The Java EE 6 Tutorial
# Contents

- **Preface** ........................................................................................................................................ 31

**Part I  Introduction** .................................................................................................................... 35

  1  **Overview** ................................................................................................................................ 37
     - Java EE 6 Platform Highlights .................................................................................................... 38
     - Java EE Application Model ......................................................................................................... 39
     - Distributed Multitiered Applications ........................................................................................ 39
       - Security ................................................................................................................................. 41
       - Java EE Components ............................................................................................................. 42
       - Java EE Clients ....................................................................................................................... 42
       - Web Components .................................................................................................................. 44
       - Business Components .......................................................................................................... 45
       - Enterprise Information System Tier ..................................................................................... 46
     - Java EE Containers .................................................................................................................. 47
       - Container Services ................................................................................................................ 47
       - Container Types ..................................................................................................................... 48
     - Web Services Support .............................................................................................................. 49
       - XML .................................................................................................................................. 49
       - SOAP Transport Protocol ....................................................................................................... 50
       - WSDL Standard Format ........................................................................................................... 50
     - Java EE Application Assembly and Deployment ....................................................................... 50
     - Packaging Applications .......................................................................................................... 51
     - Development Roles ................................................................................................................. 52
       - Java EE Product Provider ....................................................................................................... 53
       - Tool Provider ......................................................................................................................... 53
       - Application Component Provider .......................................................................................... 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Assembler</td>
<td>54</td>
</tr>
<tr>
<td>Application Deployer and Administrator</td>
<td>54</td>
</tr>
<tr>
<td>Java EE 6 APIs</td>
<td>55</td>
</tr>
<tr>
<td>Enterprise JavaBeans Technology</td>
<td>58</td>
</tr>
<tr>
<td>Java Servlet Technology</td>
<td>59</td>
</tr>
<tr>
<td>JavaServer Faces Technology</td>
<td>59</td>
</tr>
<tr>
<td>JavaServer Pages Technology</td>
<td>60</td>
</tr>
<tr>
<td>JavaServer Pages Standard Tag Library</td>
<td>60</td>
</tr>
<tr>
<td>Java Persistence API</td>
<td>61</td>
</tr>
<tr>
<td>Java Transaction API</td>
<td>61</td>
</tr>
<tr>
<td>Java API for RESTful Web Services</td>
<td>61</td>
</tr>
<tr>
<td>Managed Beans</td>
<td>61</td>
</tr>
<tr>
<td>Contexts and Dependency Injection for the Java EE Platform (JSR 299)</td>
<td>62</td>
</tr>
<tr>
<td>Dependency Injection for Java (JSR 330)</td>
<td>62</td>
</tr>
<tr>
<td>Bean Validation</td>
<td>62</td>
</tr>
<tr>
<td>Java Message Service API</td>
<td>63</td>
</tr>
<tr>
<td>Java EE Connector Architecture</td>
<td>63</td>
</tr>
<tr>
<td>JavaMail API</td>
<td>63</td>
</tr>
<tr>
<td>Java Authorization Contract for Containers</td>
<td>63</td>
</tr>
<tr>
<td>Java Authentication Service Provider Interface for Containers</td>
<td>64</td>
</tr>
<tr>
<td>Java EE 6 APIs in the Java Platform, Standard Edition 6 and 7</td>
<td>64</td>
</tr>
<tr>
<td>Java Database Connectivity API</td>
<td>64</td>
</tr>
<tr>
<td>Java Naming and Directory Interface API</td>
<td>65</td>
</tr>
<tr>
<td>JavaBeans Activation Framework</td>
<td>65</td>
</tr>
<tr>
<td>Java API for XML Processing</td>
<td>65</td>
</tr>
<tr>
<td>Java Architecture for XML Binding</td>
<td>66</td>
</tr>
<tr>
<td>SOAP with Attachments API for Java</td>
<td>66</td>
</tr>
<tr>
<td>Java API for XML Web Services</td>
<td>66</td>
</tr>
<tr>
<td>Java Authentication and Authorization Service</td>
<td>67</td>
</tr>
<tr>
<td>GlassFish Server Tools</td>
<td>67</td>
</tr>
</tbody>
</table>

## 2 Using the Tutorial Examples

### Required Software

- Java Platform, Standard Edition                                     | 69   |
- Java EE 6 Software Development Kit                                  | 70   |
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Expressions</td>
<td>132</td>
</tr>
<tr>
<td>Defining a Tag Attribute Type</td>
<td>134</td>
</tr>
<tr>
<td>Literal Expressions</td>
<td>135</td>
</tr>
<tr>
<td>Operators</td>
<td>136</td>
</tr>
<tr>
<td>Reserved Words</td>
<td>136</td>
</tr>
<tr>
<td>Examples of EL Expressions</td>
<td>137</td>
</tr>
<tr>
<td>7 Using JavaServer Faces Technology in Web Pages</td>
<td>139</td>
</tr>
<tr>
<td>Setting Up a Page</td>
<td>139</td>
</tr>
<tr>
<td>Adding Components to a Page Using HTML Tags</td>
<td>140</td>
</tr>
<tr>
<td>Common Component Tag Attributes</td>
<td>142</td>
</tr>
<tr>
<td>Adding HTML Head and Body Tags</td>
<td>145</td>
</tr>
<tr>
<td>Adding a Form Component</td>
<td>146</td>
</tr>
<tr>
<td>Using Text Components</td>
<td>147</td>
</tr>
<tr>
<td>Using Command Component Tags for Performing Actions and Navigation</td>
<td>152</td>
</tr>
<tr>
<td>Adding Graphics and Images with the <code>h:graphicImage</code> Tag</td>
<td>153</td>
</tr>
<tr>
<td>Laying Out Components with the <code>h:panelGrid</code> and <code>h:panelGroup</code> Tags</td>
<td>154</td>
</tr>
<tr>
<td>Displaying Components for Selecting One Value</td>
<td>156</td>
</tr>
<tr>
<td>Displaying Components for Selecting Multiple Values</td>
<td>158</td>
</tr>
<tr>
<td>Using the <code>f:selectItem</code> and <code>f:selectItems</code> Tags</td>
<td>159</td>
</tr>
<tr>
<td>Displaying the Results from Selection Components</td>
<td>161</td>
</tr>
<tr>
<td>Using Data-Bound Table Components</td>
<td>161</td>
</tr>
<tr>
<td>Displaying Error Messages with the <code>h:message</code> and <code>h:messages</code> Tags</td>
<td>164</td>
</tr>
<tr>
<td>Creating Bookmarkable URLs with the <code>h:button</code> and <code>h:link</code> Tags</td>
<td>165</td>
</tr>
<tr>
<td>Using View Parameters to Configure Bookmarkable URLs</td>
<td>166</td>
</tr>
<tr>
<td>The bookmarks Example Application</td>
<td>167</td>
</tr>
<tr>
<td>Resource Relocation Using <code>h:outputScript</code> and <code>h:outputStylesheet</code> Tags</td>
<td>169</td>
</tr>
<tr>
<td>Using Core Tags</td>
<td>171</td>
</tr>
<tr>
<td>8 Using Converters, Listeners, and Validators</td>
<td>175</td>
</tr>
<tr>
<td>Using the Standard Converters</td>
<td>175</td>
</tr>
<tr>
<td>Converting a Component's Value</td>
<td>176</td>
</tr>
<tr>
<td>Using <code>DateTimeConverter</code></td>
<td>177</td>
</tr>
<tr>
<td>Using <code>NumberConverter</code></td>
<td>179</td>
</tr>
<tr>
<td>Registering Listeners on Components</td>
<td>180</td>
</tr>
</tbody>
</table>
### Chapter 9: Developing with JavaServer Faces Technology

- Registering a Value-Change Listener on a Component .......................................................... 181
- Registering an Action Listener on a Component ................................................................. 182
- Using the Standard Validators .............................................................................................. 183
  - Validating a Component’s Value ....................................................................................... 185
  - Using LongRangeValidator ......................................................................................... 185
- Referencing a Managed Bean Method .................................................................................. 186
  - Referencing a Method That Performs Navigation ......................................................... 187
  - Referencing a Method That Handles an Action Event ................................................... 187
  - Referencing a Method That Performs Validation ........................................................... 187
  - Referencing a Method That Handles a Value-Change Event ......................................... 188

### Chapter 10: JavaServer Faces Technology: Advanced Concepts

- The Lifecycle of a JavaServer Faces Application ............................................................... 209
  - Overview of the JavaServer Faces Lifecycle ................................................................... 210
  - Restore View Phase ......................................................................................................... 213
  - Apply Request Values Phase ........................................................................................... 213
  - Process Validations Phase ............................................................................................... 214
  - Update Model Values Phase ............................................................................................ 214
  - Invoke Application Phase ............................................................................................... 215
11 Using Ajax with JavaServer Faces Technology ................................................................. 227
   Overview of Ajax .................................................................................................................. 228
   Using Ajax Functionality with JavaServer Faces Technology ........................................ 228
   Using Ajax with Facelets .................................................................................................... 229
      Using the f:ajax Tag ........................................................................................................ 229
   Sending an Ajax Request ...................................................................................................... 231
      Using the event Attribute .............................................................................................. 231
      Using the execute Attribute ....................................................................................... 232
      Using the immediate Attribute ............................................................................... 232
      Using the listener Attribute .................................................................................... 233
   Monitoring Events on the Client ............................................................................................ 233
   Handling Errors .................................................................................................................. 234
   Receiving an Ajax Response ............................................................................................... 234
   Ajax Request Lifecycle ........................................................................................................ 235
   Grouping of Components ..................................................................................................... 236
   Loading JavaScript as a Resource ..................................................................................... 236
      Using JavaScript API in a Facelets Application .......................................................... 237
      Using the @ResourceDependency Annotation in a Bean Class ................................ 238
   Theajaxguessnumber Example Application .................................................................... 238
      Theajaxguessnumber Source Files ........................................................................ 239
      Running theajaxguessnumber Example ................................................................. 240
   Further Information about Ajax in JavaServer Faces Technology ...................................... 242
Contents

12 Composite Components: Advanced Topics and Example .......................................................... 243
   Attributes of a Composite Component .............................................................................. 243
   Invoking a Managed Bean ............................................................................................... 244
   Validating Composite Component Values ....................................................................... 245
   The compositecomponentlogin Example Application .................................................... 245
      The Composite Component File .................................................................................. 245
      The Using Page ......................................................................................................... 246
      The Managed Bean ...................................................................................................... 247
      Running the compositecomponentlogin Example ..................................................... 248

13 Creating Custom UI Components and Other Custom Objects ............................................. 251
   Determining Whether You Need a Custom Component or Renderer ................................ 253
      When to Use a Custom Component ........................................................................... 253
      When to Use a Custom Renderer ............................................................................... 254
      Component, Renderer, and Tag Combinations .......................................................... 255
   Understanding the Image Map Example ........................................................................ 256
      Why Use JavaServer Faces Technology to Implement an Image Map? ....................... 256
   Understanding the Rendered HTML .............................................................................. 257
   Understanding the Facelets Page .................................................................................... 258
   Configuring Model Data ............................................................................................... 259
      Summary of the Image Map Application Classes ...................................................... 260
   Steps for Creating a Custom Component ....................................................................... 261
   Creating Custom Component Classes ........................................................................... 262
      Specifying the Component Family .......................................................................... 264
      Performing Encoding ................................................................................................. 265
      Performing Decoding ................................................................................................. 267
      Enabling Component Properties to Accept Expressions .......................................... 268
      Saving and Restoring State ....................................................................................... 269
   Delegating Rendering to a Renderer .............................................................................. 270
      Creating the Renderer Class ...................................................................................... 270
      Identifying the Renderer Type .................................................................................... 272
   Implementing an Event Listener ..................................................................................... 273
      Implementing Value-Change Listeners ....................................................................... 273
      Implementing Action Listeners .................................................................................. 274
   Handling Events for Custom Components .................................................................... 275
14 Configuring JavaServer Faces Applications ................................................................. 295
  Using Annotations to Configure Managed Beans ............................................................ 296
    Using Managed Bean Scopes .......................................................................................... 296
  Application Configuration Resource File .......................................................................... 297
    Ordering of Application Configuration Resource Files ............................................... 298
  Configuring Managed Beans .......................................................................................... 300
    Using the managed-bean Element ............................................................................... 300
    Initializing Properties Using the managed-property Element ..................................... 303
    Initializing Maps and Lists .......................................................................................... 308
  Registering Application Messages .................................................................................. 308
    Using FacesMessage to Create a Message .................................................................... 310
    Referencing Error Messages ....................................................................................... 310
  Using Default Validators ............................................................................................... 311
  Registering a Custom Validator ..................................................................................... 312
  Registering a Custom Converter ................................................................................... 313
  Configuring Navigation Rules ....................................................................................... 313
    ▼ To Configure a Navigation Rule ............................................................................... 315
    Implicit Navigation Rules ............................................................................................ 316
  Registering a Custom Renderer with a Render Kit ............................................................ 317
  Registering a Custom Component .................................................................................. 319
# Java Servlet Technology

What Is a Servlet? ....................................................................................................................... 328
Servlet Lifecycle ................................................................................................................................. 328
   Handling Servlet Lifecycle Events ............................................................................................ 328
   Handling Servlet Errors ............................................................................................................. 330
Sharing Information ......................................................................................................................... 330
   Using Scope Objects .................................................................................................................. 330
   Controlling Concurrent Access to Shared Resources ........................................................... 331
Creating and Initializing a Servlet ................................................................................................... 331
Writing Service Methods .................................................................................................................. 332
   Getting Information from Requests ........................................................................................ 332
   Constructing Responses .......................................................................................................... 333
Filtering Requests and Responses .................................................................................................. 334
   Programming Filters ............................................................................................................... 335
   Programming Customized Requests and Responses ............................................................ 336
   Specifying Filter Mappings ....................................................................................................... 336
Invoking Other Web Resources ...................................................................................................... 338
   Including Other Resources in the Response ........................................................................... 339
   Transferring Control to Another Web Component .............................................................. 339
Accessing the Web Context .......................................................................................................... 339
Maintaining Client State ................................................................................................................. 340
   Accessing a Session .................................................................................................................. 340
   Associating Objects with a Session .......................................................................................... 340
   Session Management ............................................................................................................... 341
   Session Tracking ..................................................................................................................... 341
Finalizing a Servlet .......................................................................................................................... 342
   Tracking Service Requests ..................................................................................................... 342
   Notifying Methods to Shut Down ............................................................................................ 343
   Creating Polite Long-Running Methods ................................................................................. 343
The mood Example Application ..................................................................................................... 344
Components of the mood Example Application ................................................................. 344
Running the mood Example ........................................................................................................ 344
Further Information about Java Servlet Technology .......................................................... 346

16 Uploading Files with Java Servlet Technology ............................................................... 347
The @MultipartConfig Annotation ....................................................................................... 347
The getParts and getPart Methods ....................................................................................... 348
The fileupload Example Application ..................................................................................... 349
Architecture of the fileupload Example Application ............................................................ 349
Running the fileupload Example .......................................................................................... 352

17 Internationalizing and Localizing Web Applications ....................................................... 355
Java Platform Localization Classes ........................................................................................ 355
Providing Localized Messages and Labels ......................................................................... 356
Establishing the Locale .......................................................................................................... 356
Setting the Resource Bundle ................................................................................................. 357
Retrieving Localized Messages ............................................................................................. 358
Date and Number Formatting ................................................................................................. 359
Character Sets and Encodings ............................................................................................... 359
Character Sets ........................................................................................................................ 359
Character Encoding ............................................................................................................... 360

Part III Web Services ............................................................................................................. 361

18 Introduction to Web Services ........................................................................................... 363
What Are Web Services? ......................................................................................................... 363
Types of Web Services ............................................................................................................. 363
“Big” Web Services ................................................................................................................ 364
RESTful Web Services .......................................................................................................... 364
Deciding Which Type of Web Service to Use ........................................................................ 366

19 Building Web Services with JAX-WS ............................................................................. 367
Creating a Simple Web Service and Clients with JAX-WS ................................................... 368
Requirements of a JAX-WS Endpoint .................................................................................... 369
Categorize the Service Endpoint Implementation Class
Building, Packaging, and Deploying the Service
Testing the Methods of a Web Service Endpoint
A Simple JAX-WS Application Client
A Simple JAX-WS Web Client
Types Supported by JAX-WS
Schema-to-Java Mapping
Java-to-Schema Mapping
Web Services Interoperability and JAX-WS
Further Information about JAX-WS

20 Building RESTful Web Services with JAX-RS
What Are RESTful Web Services?
Creating a RESTful Root Resource Class
Developing RESTful Web Services with JAX-RS
Overview of a JAX-RS Application
The @Path Annotation and URI Path Templates
Responding to HTTP Methods and Requests
Using @Consumes and @Produces to Customize Requests and Responses
Extracting Request Parameters
Example Applications for JAX-RS
A RESTful Web Service
The rsvp Example Application
Real-World Examples
Further Information about JAX-RS

21 JAX-RS: Advanced Topics and Example
Annotations for Field and Bean Properties of Resource Classes
Extracting Path Parameters
Extracting Query Parameters
Extracting Form Data
Extracting the Java Type of a Request or Response
Subresources and Runtime Resource Resolution
Subresource Methods
Subresource Locators
Integrating JAX- RS with EJB Technology and CDI .................................................................408
Conditional HTTP Requests .................................................................................................409
Runtime Content Negotiation ...............................................................................................410
Using JAX- RS With JAXB .....................................................................................................412
  Using Java Objects to Model Your Data ............................................................................414
  Starting from an Existing XML Schema Definition .........................................................415
  Using JSON with JAX- RS and JAXB ...............................................................................417
The customer Example Application .....................................................................................418
  Overview of the customer Example Application ...............................................................419
  The Customer and Address Entity Classes .......................................................................419
  The CustomerService Class .............................................................................................422
  The CustomerClientXML and CustomerClientJSON Classes ........................................424
  Modifying the Example to Generate Entity Classes from an Existing Schema .............426
  Running the customer Example .......................................................................................428

Part IV Enterprise Beans ........................................................................................................433

22 Enterprise Beans ..............................................................................................................435
  What Is an Enterprise Bean? ...............................................................................................435
    Benefits of Enterprise Beans ..........................................................................................436
    When to Use Enterprise Beans .....................................................................................436
    Types of Enterprise Beans ...........................................................................................436
  What Is a Session Bean? ....................................................................................................437
    Types of Session Beans ..................................................................................................437
    When to Use Session Beans ..........................................................................................438
  What Is a Message-Driven Bean? ......................................................................................439
    What Makes Message-Driven Beans Different from Session Beans? .........................439
    When to Use Message-Driven Beans ..........................................................................440
  Accessing Enterprise Beans ..............................................................................................441
    Using Enterprise Beans in Clients ................................................................................441
    Deciding on Remote or Local Access .............................................................................442
    Local Clients ................................................................................................................443
    Remote Clients .............................................................................................................445
    Web Service Clients .......................................................................................................446
    Method Parameters and Access ....................................................................................447
Creating Calendar-Based Timer Expressions ................................................................. 479
Programmatic Timers ........................................................................................................ 482
Automatic Timers ............................................................................................................... 483
Canceling and Saving Timers ............................................................................................ 484
Getting Timer Information ............................................................................................... 485
Transactions and Timers .................................................................................................. 485
The timersession Example ............................................................................................... 488
Running the timersession Example .................................................................................. 488
Handling Exceptions ........................................................................................................ 489

25 A Message-Driven Bean Example ............................................................................... 491
Overview of the simplemessage Example ......................................................................... 491
The simplemessage Application Client ............................................................................ 492
The Message-Driven Bean Class ..................................................................................... 493
The onMessage Method .................................................................................................... 494
Running the simplemessage Example ............................................................................. 495
Administered Objects for the simplemessage Example ..................................................... 495
▼ To Run the simplemessage Application Using NetBeans IDE .................................. 496
▼ To Run the simplemessage Application Using Ant ...................................................... 496
Removing the Administered Objects for the simplemessage Example ......................... 497

26 Using the Embedded Enterprise Bean Container ......................................................... 499
Overview of the Embedded Enterprise Bean Container .................................................. 499
Developing Embeddable Enterprise Bean Applications .................................................. 499
Running Embedded Applications .................................................................................... 500
Creating the Enterprise Bean Container ......................................................................... 500
Looking Up Session Bean References ............................................................................ 502
Shutting Down the Enterprise Bean Container ............................................................... 502
The standalone Example Application ............................................................................. 502
▼ To Run the standalone Example Application ............................................................... 503

27 Using Asynchronous Method Invocation in Session Beans ......................................... 505
Asynchronous Method Invocation .................................................................................... 505
Creating an Asynchronous Business Method ............................................................... 506
Calling Asynchronous Methods from Enterprise Bean Clients ........................................... 507
The async Example Application ..................................................................................... 508
Architecture of the async Example Application ......................................................... 508
Running the async Example ....................................................................................... 509

Part V   Contexts and Dependency Injection for the Java EE Platform ........................................ 513

28 Introduction to Contexts and Dependency Injection for the Java EE Platform ............... 515
Overview of CDI ............................................................................................................. 516
About Beans .................................................................................................................. 517
About CDI Managed Beans ............................................................................................ 517
Beans as Injectable Objects ............................................................................................ 518
Using Qualifiers .............................................................................................................. 519
Injecting Beans .............................................................................................................. 520
Using Scopes .................................................................................................................. 520
Overriding the Scope of a Bean at the Point of Injection ............................................. 522
Giving Beans EL Names ............................................................................................... 522
Adding Setter and Getter Methods ............................................................................. 523
Using a Managed Bean in a Facelets Page .................................................................. 524
Injecting Objects by Using Producer Methods ............................................................ 524
Configuring a CDI Application ....................................................................................... 525
Using the @PostConstruct and @PreDestroy Annotations With CDI Managed Bean Classes .................................................................................................................. 525
▼ To Initialize a Managed Bean Using the @PostConstruct Annotation ................... 525
▼ To Prepare for the Destruction of a Managed Bean Using the @PreDestroy Annotation 526
Further Information about CDI .................................................................................... 526

29 Running the Basic Contexts and Dependency Injection Examples ............................... 527
The simplegreeting CDI Example ............................................................................... 527
The simplegreeting Source Files ................................................................................. 528
The Facelets Template and Page .................................................................................. 528
Configuration Files ....................................................................................................... 529
Running the simplegreeting Example .......................................................................... 530
The guessnumber CDI Example ................................................................................. 531
The guessnumber Source Files ................................................................................... 532
30 Contexts and Dependency Injection for the Java EE Platform: Advanced Topics ........................................ 539
   Using Alternatives in CDI Applications ................................................................. 539
      Using Specialization ...................................................................................... 540
   Using Producer Methods, Producer Fields, and Disposer Methods in CDI Applications .... 541
      Using Producer Methods ............................................................................... 542
      Using Producer Fields to Generate Resources ............................................. 543
      Using a Disposer Method .............................................................................. 543
   Using Predefined Beans in CDI Applications ....................................................... 544
   Using Events in CDI Applications ...................................................................... 545
      Defining Events ............................................................................................... 545
      Using Observer Methods to Handle Events .................................................. 545
      Firing Events .................................................................................................... 546
   Using Interceptors in CDI Applications ............................................................... 547
   Using Decorators in CDI Applications .................................................................. 549
   Using Stereotypes in CDI Applications ............................................................... 550

31 Running the Advanced Contexts and Dependency Injection Examples .................................................. 553
   The encoder Example: Using Alternatives ......................................................... 553
      The Coder Interface and Implementations ...................................................... 554
      The encoder Facelets Page and Managed Bean .............................................. 554
      Running the encoder Example ....................................................................... 556
   The producermethods Example: Using a Producer Method To Choose a Bean
      Implementation ............................................................................................... 558
      Components of the producermethods Example ............................................ 559
      Running the producermethods Example ....................................................... 560
   The producerfields Example: Using Producer Fields to Generate Resources .......... 561
      The Producer Field for the producerfields Example ...................................... 561
      The producerfields Entity and Session Bean ................................................. 563
      The producerfields Facelets Pages and Managed Bean ................................. 564
      Running the producerfields Example .......................................................... 566
   The billpayment Example: Using Events and Interceptors ................................ 568
      The PaymentEvent Event Class .................................................................... 568
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cascade Operations in the order Application</td>
<td>610</td>
</tr>
<tr>
<td>BLOB and CLOB Database Types in the order Application</td>
<td>611</td>
</tr>
<tr>
<td>Temporal Types in the order Application</td>
<td>612</td>
</tr>
<tr>
<td>Managing the order Application’s Entities</td>
<td>612</td>
</tr>
<tr>
<td>Running the order Example</td>
<td>614</td>
</tr>
<tr>
<td>The roster Application</td>
<td>615</td>
</tr>
<tr>
<td>Relationships in the roster Application</td>
<td>616</td>
</tr>
<tr>
<td>Entity Inheritance in the roster Application</td>
<td>617</td>
</tr>
<tr>
<td>Criteria Queries in the roster Application</td>
<td>619</td>
</tr>
<tr>
<td>Automatic Table Generation in the roster Application</td>
<td>620</td>
</tr>
<tr>
<td>Running the roster Example</td>
<td>621</td>
</tr>
<tr>
<td>The address-book Application</td>
<td>623</td>
</tr>
<tr>
<td>Bean Validation Constraints in address-book</td>
<td>623</td>
</tr>
<tr>
<td>Specifying Error Messages for Constraints in address-book</td>
<td>624</td>
</tr>
<tr>
<td>Validating Contact Input from a JavaServer Faces Application</td>
<td>625</td>
</tr>
<tr>
<td>Running the address-book Example</td>
<td>625</td>
</tr>
</tbody>
</table>

34 The Java Persistence Query Language

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Language Terminology</td>
<td>630</td>
</tr>
<tr>
<td>Creating Queries Using the Java Persistence Query Language</td>
<td>630</td>
</tr>
<tr>
<td>Named Parameters in Queries</td>
<td>631</td>
</tr>
<tr>
<td>Positional Parameters in Queries</td>
<td>631</td>
</tr>
<tr>
<td>Simplified Query Language Syntax</td>
<td>632</td>
</tr>
<tr>
<td>Select Statements</td>
<td>632</td>
</tr>
<tr>
<td>Update and Delete Statements</td>
<td>632</td>
</tr>
<tr>
<td>Example Queries</td>
<td>633</td>
</tr>
<tr>
<td>Simple Queries</td>
<td>633</td>
</tr>
<tr>
<td>Queries That Navigate to Related Entities</td>
<td>634</td>
</tr>
<tr>
<td>Queries with Other Conditional Expressions</td>
<td>635</td>
</tr>
<tr>
<td>Bulk Updates and Deletes</td>
<td>637</td>
</tr>
<tr>
<td>Full Query Language Syntax</td>
<td>637</td>
</tr>
<tr>
<td>BNF Symbols</td>
<td>637</td>
</tr>
<tr>
<td>BNF Grammar of the Java Persistence Query Language</td>
<td>638</td>
</tr>
<tr>
<td>FROM Clause</td>
<td>642</td>
</tr>
<tr>
<td>Path Expressions</td>
<td>646</td>
</tr>
</tbody>
</table>
35 Using the Criteria API to Create Queries ................................................................. 661
   Overview of the Criteria and Metamodel APIs .......................................................... 661
   Using the Metamodel API to Model Entity Classes .................................................. 663
      Using Metamodel Classes ....................................................................................... 664
   Using the Criteria API and Metamodel API to Create Basic Typesafe Queries ........ 664
      Creating a Criteria Query ...................................................................................... 665
      Query Roots ........................................................................................................... 665
      Querying Relationships Using Joins ...................................................................... 666
      Path Navigation in Criteria Queries ...................................................................... 667
      Restricting Criteria Query Results ....................................................................... 667
      Managing Criteria Query Results ........................................................................ 669
      Executing Queries .................................................................................................. 671

36 Creating and Using String-Based Criteria Queries ................................................... 673
   Overview of String-Based Criteria API Queries ....................................................... 673
   Creating String-Based Queries ................................................................................ 674
   Executing String-Based Queries ............................................................................... 675

37 Controlling Concurrent Access to Entity Data with Locking ..................................... 677
   Overview of Entity Locking and Concurrency ............................................................ 677
      Using Optimistic Locking ...................................................................................... 678
      Lock Modes ............................................................................................................ 679
      Setting the Lock Mode ......................................................................................... 680
      Using Pessimistic Locking .................................................................................... 680

38 Using a Second-Level Cache with Java Persistence API Applications .................... 683
   Overview of the Second-Level Cache ...................................................................... 683
      Controlling Whether Entities May Be Cached ......................................................... 684
      Specifying the Cache Mode Settings to Improve Performance ............................ 685
Preface

This tutorial is a guide to developing enterprise applications for the Java Platform, Enterprise Edition 6 (Java EE 6) using GlassFish Server Open Source Edition.

Oracle GlassFish Server, a Java EE compatible application server, is based on GlassFish Server Open Source Edition, the leading open-source and open-community platform for building and deploying next-generation applications and services. GlassFish Server Open Source Edition, developed by the GlassFish project open-source community at http://glassfish.java.net/, is the first compatible implementation of the Java EE 6 platform specification. This lightweight, flexible, and open-source application server enables organizations not only to leverage the new capabilities introduced within the Java EE 6 specification, but also to add to their existing capabilities through a faster and more streamlined development and deployment cycle. Oracle GlassFish Server, the product version, and GlassFish Server Open Source Edition, the open-source version, are hereafter referred to as GlassFish Server.

The following topics are addressed here:

■ “Before You Read This Book” on page 31
■ “Related Documentation” on page 32
■ “Typographic Conventions” on page 32
■ “Default Paths and File Names” on page 33
■ “Third-Party Web Site References” on page 34

Before You Read This Book

Before proceeding with this tutorial, you should have a good knowledge of the Java programming language. A good way to get to that point is to work through the Java Tutorials (http://docs.oracle.com/javase/tutorial/index.html).
Related Documentation


The API documentation for packages that are provided with GlassFish Server is available as follows.

- The API specification for version 6 of Java EE is located at http://docs.oracle.com/javaee/6/api/.
- The API specification for GlassFish Server, including Java EE 6 platform packages and nonplatform packages that are specific to the GlassFish Server product, is located at http://glassfish.java.net/nonav/docs/v3/api/.

Additionally, the Java EE Specifications at http://www.oracle.com/technetwork/java/javasee/tech/index.html might be useful.

For information about creating enterprise applications in the NetBeans Integrated Development Environment (IDE), see http://www.netbeans.org/kb/.

For information about the Java DB database for use with the GlassFish Server, see http://www.oracle.com/technetwork/java/javasee/overview/index.html.

The GlassFish Samples project is a collection of sample applications that demonstrate a broad range of Java EE technologies. The GlassFish Samples are bundled with the Java EE Software Development Kit (SDK) and are also available from the GlassFish Samples project page at http://glassfish-samples.java.net/.

Typographic Conventions

Table P–1 describes the typographic changes that are used in this book.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories, and onscreen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. machine_name% you have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, contrasted with onscreen computer output</td>
<td>machine_name% su Password:</td>
</tr>
</tbody>
</table>
TABLE P–1 Typographic Conventions (Continued)

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>A placeholder to be replaced with a real name or value</td>
<td>The command to remove a file is <code>rm filename</code>.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)</td>
<td>Read Chapter 6 in the User's Guide. A cache is a copy that is stored locally. Do not save the file.</td>
</tr>
</tbody>
</table>

Default Paths and File Names

Table P–2 describes the default paths and file names that are used in this book.

TABLE P–2 Default Paths and File Names

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>as-install</td>
<td>Represents the base installation directory for the GlassFish Server or the SDK of which the GlassFish Server is a part.</td>
<td>Installations on the Solaris operating system, Linux operating system, and Mac operating system: user's-home-directory/glassfish3/glassfish Windows, all installations: SystemDrive:\glassfish3\glassfish</td>
</tr>
<tr>
<td>as-install-parent</td>
<td>Represents the parent of the base installation directory for GlassFish Server.</td>
<td>Installations on the Solaris operating system, Linux operating system, and Mac operating system: user's-home-directory/glassfish3 Windows, all installations: SystemDrive:\glassfish3</td>
</tr>
<tr>
<td>tut-install</td>
<td>Represents the base installation directory for the Java EE Tutorial after you install the GlassFish Server or the SDK and run the Update Tool.</td>
<td>as-install/docs/javaee-tutorial</td>
</tr>
<tr>
<td>domain-root-dir</td>
<td>Represents the directory in which a domain is created by default.</td>
<td>as-install/domains/</td>
</tr>
<tr>
<td>domain-dir</td>
<td>Represents the directory in which a domain's configuration is stored.</td>
<td>domain-root-dir/domain-name</td>
</tr>
</tbody>
</table>
Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.
Part I introduces the platform, the tutorial, and the examples. This part contains the following chapters:

- Chapter 1, “Overview”
- Chapter 2, “Using the Tutorial Examples”
This chapter introduces you to Java EE enterprise application development. Here you will review development basics, learn about the Java EE architecture and APIs, become acquainted with important terms and concepts, and find out how to approach Java EE application programming, assembly, and deployment.

Developers today increasingly recognize the need for distributed, transactional, and portable applications that leverage the speed, security, and reliability of server-side technology. 

**Enterprise applications** provide the business logic for an enterprise. They are centrally managed and often interact with other enterprise software. In the world of information technology, enterprise applications must be designed, built, and produced for less money, with greater speed, and with fewer resources.

With the Java Platform, Enterprise Edition (Java EE), development of Java enterprise applications has never been easier or faster. The aim of the Java EE platform is to provide developers with a powerful set of APIs while shortening development time, reducing application complexity, and improving application performance.

The Java EE platform is developed through the Java Community Process (JCP), which is responsible for all Java technologies. Expert groups, composed of interested parties, have created Java Specification Requests (JSRs) to define the various Java EE technologies. The work of the Java Community under the JCP program helps to ensure Java technology’s standard of stability and cross-platform compatibility.

The Java EE platform uses a simplified programming model. XML deployment descriptors are optional. Instead, a developer can simply enter the information as an **annotation** directly into a Java source file, and the Java EE server will configure the component at deployment and runtime. These annotations are generally used to embed in a program data that would otherwise be furnished in a deployment descriptor. With annotations, you put the specification information in your code next to the program element affected.

In the Java EE platform, dependency injection can be applied to all resources a component needs, effectively hiding the creation and lookup of resources from application code.
Dependency injection can be used in EJB containers, web containers, and application clients. Dependency injection allows the Java EE container to automatically insert references to other required components or resources, using annotations.

This tutorial uses examples to describe the features available in the Java EE platform for developing enterprise applications. Whether you are a new or experienced Enterprise developer, you should find the examples and accompanying text a valuable and accessible knowledge base for creating your own solutions.

The following topics are addressed here:

- "Java EE 6 Platform Highlights" on page 38
- "Java EE Application Model" on page 39
- "Distributed Multitiered Applications" on page 39
- "Java EE Containers" on page 47
- "Web Services Support" on page 49
- "Java EE Application Assembly and Deployment" on page 50
- "Packaging Applications" on page 51
- "Development Roles" on page 52
- "Java EE 6 APIs" on page 55
- "Java EE 6 APIs in the Java Platform, Standard Edition 6 and 7" on page 64
- "GlassFish Server Tools" on page 67

Java EE 6 Platform Highlights

The most important goal of the Java EE 6 platform is to simplify development by providing a common foundation for the various kinds of components in the Java EE platform. Developers benefit from productivity improvements with more annotations and less XML configuration, more Plain Old Java Objects (POJOs), and simplified packaging. The Java EE 6 platform includes the following new features:

- Profiles: configurations of the Java EE platform targeted at specific classes of applications. Specifically, the Java EE 6 platform introduces a lightweight Web Profile targeted at next-generation web applications, as well as a Full Profile that contains all Java EE technologies and provides the full power of the Java EE 6 platform for enterprise applications.

- New technologies, including the following:
  - Java API for RESTful Web Services (JAX-RS)
  - Managed Beans
  - Contexts and Dependency Injection for the Java EE Platform (JSR 299), informally known as CDI
  - Dependency Injection for Java (JSR 330)
  - Bean Validation (JSR 303)
Java Authentication Service Provider Interface for Containers (JASPIC)

- New features for Enterprise JavaBeans (EJB) components (see “Enterprise JavaBeans Technology” on page 58 for details)
- New features for servlets (see “Java Servlet Technology” on page 59 for details)
- New features for JavaServer Faces components (see “JavaServer Faces Technology” on page 59 for details)

Java EE Application Model

The Java EE application model begins with the Java programming language and the Java virtual machine. The proven portability, security, and developer productivity they provide forms the basis of the application model. Java EE is designed to support applications that implement enterprise services for customers, employees, suppliers, partners, and others who make demands on or contributions to the enterprise. Such applications are inherently complex, potentially accessing data from a variety of sources and distributing applications to a variety of clients.

To better control and manage these applications, the business functions to support these various users are conducted in the middle tier. The middle tier represents an environment that is closely controlled by an enterprise’s information technology department. The middle tier is typically run on dedicated server hardware and has access to the full services of the enterprise.

The Java EE application model defines an architecture for implementing services as multitier applications that deliver the scalability, accessibility, and manageability needed by enterprise-level applications. This model partitions the work needed to implement a multitier service into the following parts:

- The business and presentation logic to be implemented by the developer
- The standard system services provided by the Java EE platform

The developer can rely on the platform to provide solutions for the hard systems-level problems of developing a multitier service.

Distributed Multitiered Applications

The Java EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the application components that make up a Java EE application are installed on various machines, depending on the tier in the multitiered Java EE environment to which the application component belongs.
Figure 1–1 shows two multitiered Java EE applications divided into the tiers described in the following list. The Java EE application parts shown in Figure 1–1 are presented in "Java EE Components" on page 42.

- Client-tier components run on the client machine.
- Web-tier components run on the Java EE server.
- Business-tier components run on the Java EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a Java EE application can consist of all tiers shown in Figure 1–1, Java EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the Java EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client-and-server model by placing a multithreaded application server between the client application and back-end storage.
Security

Although other enterprise application models require platform-specific security measures in each application, the Java EE security environment enables security constraints to be defined at deployment time. The Java EE platform makes applications portable to a wide variety of security implementations by shielding application developers from the complexity of implementing security features.

The Java EE platform provides standard declarative access control rules that are defined by the developer and interpreted when the application is deployed on the server. Java EE also provides standard login mechanisms so application developers do not have to implement these mechanisms in their applications. The same application works in a variety of security environments without changing the source code.
Java EE Components

Java EE applications are made up of components. A Java EE component is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components.

The Java EE specification defines the following Java EE components:
- Application clients and applets are components that run on the client.
- Java Servlet, JavaServer Faces, and JavaServer Pages (JSP) technology components are web components that run on the server.
- Enterprise JavaBeans (EJB) components (enterprise beans) are business components that run on the server.

Java EE components are written in the Java programming language and are compiled in the same way as any program in the language. The differences between Java EE components and “standard” Java classes are that Java EE components are assembled into a Java EE application, they are verified to be well formed and in compliance with the Java EE specification, and they are deployed to production, where they are run and managed by the Java EE server.

Java EE Clients

A Java EE client is usually either a web client or an application client.

Web Clients

A web client consists of two parts:
- Dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by web components running in the web tier
- A web browser, which renders the pages received from the server

A web client is sometimes called a thin client. Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, such heavyweight operations are off-loaded to enterprise beans executing on the Java EE server, where they can leverage the security, speed, services, and reliability of Java EE server-side technologies.

Application Clients

An application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. An application client typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API, but a command-line interface is certainly possible.
Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the web tier. Application clients written in languages other than Java can interact with Java EE servers, enabling the Java EE platform to interoperate with legacy systems, clients, and non-Java languages.

**Applets**

A web page received from the web tier can include an embedded applet. Written in the Java programming language, an *applet* is a small client application that executes in the Java virtual machine installed in the web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file for the applet to successfully execute in the web browser.

Web components are the preferred API for creating a web client program, because no plug-ins or security policy files are needed on the client systems. Also, web components enable cleaner and more modular application design because they provide a way to separate applications programming from web page design. Personnel involved in web page design thus do not need to understand Java programming language syntax to do their jobs.

**The JavaBeans Component Architecture**

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans components) to manage the data flow between the following:

- An application client or applet and components running on the Java EE server
- Server components and a database

JavaBeans components are not considered Java EE components by the Java EE specification.

JavaBeans components have properties and have `get` and `set` methods for accessing the properties. JavaBeans components used in this way are typically simple in design and implementation but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

**Java EE Server Communications**

*Figure 1–2* shows the various elements that can make up the client tier. The client communicates with the business tier running on the Java EE server either directly or, as in the case of a client running in a browser, by going through web pages or servlets running in the web tier.
Web Components

Java EE web components are either servlets or web pages created using JavaServer Faces technology and/or JSP technology (JSP pages). Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. JavaServer Faces technology builds on servlets and JSP technology and provides a user interface component framework for web applications.

Static HTML pages and applets are bundled with web components during application assembly but are not considered web components by the Java EE specification. Server-side utility classes can also be bundled with web components and, like HTML pages, are not considered web components.

As shown in Figure 1–3, the web tier, like the client tier, might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.
**Business Components**

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in either the business tier or the web tier. **Figure 1–4** shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.
Enterprise Information System Tier

The enterprise information system tier handles EIS software and includes enterprise infrastructure systems, such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. For example, Java EE application components might need access to enterprise information systems for database connectivity.
Java EE Containers

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component-based and platform-independent Java EE architecture makes Java EE applications easy to write because business logic is organized into reusable components. In addition, the Java EE server provides underlying services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Container Services

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before it can be executed, a web, enterprise bean, or application client component must be assembled into a Java EE module and deployed into its container.

The assembly process involves specifying container settings for each component in the Java EE application and for the Java EE application itself. Container settings customize the underlying support provided by the Java EE server, including such services as security, transaction management, Java Naming and Directory Interface (JNDI) API lookups, and remote connectivity. Here are some of the highlights.

- The Java EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.
- The Java EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access these services.
- The Java EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

Because the Java EE architecture provides configurable services, application components within the same Java EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages nonconfigurable services, such as enterprise bean and servlet lifecycles, database connection resource pooling, data persistence, and access to the Java EE platform APIs (see “Java EE 6 APIs” on page 55).
Container Types

The deployment process installs Java EE application components in the Java EE containers as illustrated in Figure 1–5.

- **Java EE server**: The runtime portion of a Java EE product. A Java EE server provides EJB and web containers.
- **Enterprise JavaBeans (EJB) container**: Manages the execution of enterprise beans for Java EE applications. Enterprise beans and their container run on the Java EE server.
- **Web container**: Manages the execution of web pages, servlets, and some EJB components for Java EE applications. Web components and their container run on the Java EE server.
- **Application client container**: Manages the execution of application client components. Application clients and their container run on the client.
- **Applet container**: Manages the execution of applets. Consists of a web browser and Java Plug-in running on the client together.

**Web Services Support**

Web services are web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients. The Java EE platform provides the XML APIs and tools you need to quickly design, develop, test, and deploy web services and clients that fully interoperate with other web services and clients running on Java-based or non-Java-based platforms.

To write web services and clients with the Java EE XML APIs, all you do is pass parameter data to the method calls and process the data returned; for document-oriented web services, you send documents containing the service data back and forth. No low-level programming is needed, because the XML API implementations do the work of translating the application data to and from an XML-based data stream that is sent over the standardized XML-based transport protocols. These XML-based standards and protocols are introduced in the following sections.

The translation of data to a standardized XML-based data stream is what makes web services and clients written with the Java EE XML APIs fully interoperable. This does not necessarily mean that the data being transported includes XML tags, because the transported data can itself be plain text, XML data, or any kind of binary data, such as audio, video, maps, program files, computer-aided design (CAD) documents, and the like. The next section introduces XML and explains how parties doing business can use XML tags and schemas to exchange data in a meaningful way.

**XML**

Extensible Markup Language (XML) is a cross-platform, extensible, text-based standard for representing data. Parties that exchange XML data can create their own tags to describe the data, set up schemas to specify which tags can be used in a particular kind of XML document, and use XML style sheets to manage the display and handling of the data.

For example, a web service can use XML and a schema to produce price lists, and companies that receive the price lists and schema can have their own style sheets to handle the data in a way that best suits their needs. Here are examples.

- One company might put XML pricing information through a program to translate the XML to HTML so that it can post the price lists to its intranet.
- A partner company might put the XML pricing information through a tool to create a marketing presentation.
Another company might read the XML pricing information into an application for processing.

**SOAP Transport Protocol**

Client requests and web service responses are transmitted as Simple Object Access Protocol (SOAP) messages over HTTP to enable a completely interoperable exchange between clients and web services, all running on different platforms and at various locations on the Internet. HTTP is a familiar request-and-response standard for sending messages over the Internet, and SOAP is an XML-based protocol that follows the HTTP request-and-response model.

The SOAP portion of a transported message does the following:
- Defines an XML-based envelope to describe what is in the message and explain how to process the message
- Includes XML-based encoding rules to express instances of application-defined data types within the message
- Defines an XML-based convention for representing the request to the remote service and the resulting response

**WSDL Standard Format**

The Web Services Description Language (WSDL) is a standardized XML format for describing network services. The description includes the name of the service, the location of the service, and ways to communicate with the service. WSDL service descriptions can be published on the Web. GlassFish Server provides a tool for generating the WSDL specification of a web service that uses remote procedure calls to communicate with clients.

**Java EE Application Assembly and Deployment**

A Java EE application is packaged into one or more standard units for deployment to any Java EE platform-compliant system. Each unit contains
- A functional component or components, such as an enterprise bean, web page, servlet, or applet
- An optional deployment descriptor that describes its content

Once a Java EE unit has been produced, it is ready to be deployed. Deployment typically involves using a platform’s deployment tool to specify location-specific information, such as a list of local users who can access it and the name of the local database. Once deployed on a local platform, the application is ready to run.
Packaging Applications

A Java EE application is delivered in a Java Archive (JAR) file, a Web Archive (WAR) file, or an Enterprise Archive (EAR) file. A WAR or EAR file is a standard JAR (.jar) file with a .war or .ear extension. Using JAR, WAR, and EAR files and modules makes it possible to assemble a number of different Java EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various Java EE modules into Java EE JAR, WAR, or EAR files.

An EAR file (see Figure 1–6) contains Java EE modules and, optionally, deployment descriptors. A deployment descriptor, an XML document with an .xml extension, describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

The two types of deployment descriptors are Java EE and runtime. A Java EE deployment descriptor is defined by a Java EE specification and can be used to configure deployment settings on any Java EE-compliant implementation. A runtime deployment descriptor is used to configure Java EE implementation-specific parameters. For example, the GlassFish Server runtime deployment descriptor contains such information as the context root of a web application, as well as GlassFish Server implementation-specific parameters, such as caching.
directives. The GlassFish Server runtime deployment descriptors are named 
glassfish-moduleType.xml and are located in the same META-INF directory as the Java EE 
deployment descriptor.

A Java EE module consists of one or more Java EE components for the same container type and, 
optionally, one component deployment descriptor of that type. An enterprise bean module 
deployment descriptor, for example, declares transaction attributes and security authorizations 
for an enterprise bean. A Java EE module can be deployed as a stand-alone module.

Java EE modules are of the following types:

- EJB modules, which contain class files for enterprise beans and, optionally, an EJB 
deployment descriptor. EJB modules are packaged as JAR files with a .jar extension.

- Web modules, which contain servlet class files, web files, supporting class files, image and 
HTML files, and, optionally, a web application deployment descriptor. Web modules are 
packaged as JAR files with a .war (web archive) extension.

- Application client modules, which contain class files and, optionally, an application client 
deployment descriptor. Application client modules are packaged as JAR files with a .jar 
extension.

- Resource adapter modules, which contain all Java interfaces, classes, native libraries, and, 
optionally, a resource adapter deployment descriptor. Together, these implement the 
Connector architecture (see ”Java EE Connector Architecture” on page 63) for a particular 
EIS. Resource adapter modules are packaged as JAR files with an .rar (resource adapter 
archive) extension.

Development Roles

Reusable modules make it possible to divide the application development and deployment 
process into distinct roles so that different people or companies can perform different parts of 
the process.

The first two roles, Java EE product provider and tool provider, involve purchasing and 
installing the Java EE product and tools. After software is purchased and installed, Java EE 
components can be developed by application component providers, assembled by application 
assemblers, and deployed by application deployers. In a large organization, each of these roles 
might be executed by different individuals or teams. This division of labor works because each 
of the earlier roles outputs a portable file that is the input for a subsequent role. For example, in 
the application component development phase, an enterprise bean software developer delivers 
EJB JAR files. In the application assembly role, another developer may combine these EJB JAR 
files into a Java EE application and save it in an EAR file. In the application deployment role, a 
system administrator at the customer site uses the EAR file to install the Java EE application into 
a Java EE server.
The different roles are not always executed by different people. If you work for a small company, for example, or if you are prototyping a sample application, you might perform tasks in every phase.

**Java EE Product Provider**

The Java EE product provider is the company that designs and makes available for purchase the Java EE platform APIs and other features defined in the Java EE specification. Product providers are typically application server vendors that implement the Java EE platform according to the Java EE 6 Platform specification.

**Tool Provider**

The tool provider is the company or person who creates development, assembly, and packaging tools used by component providers, assemblers, and deployers.

**Application Component Provider**

The application component provider is the company or person who creates web components, enterprise beans, applets, or application clients for use in Java EE applications.

**Enterprise Bean Developer**

An enterprise bean developer performs the following tasks to deliver an EJB JAR file that contains one or more enterprise beans:

- Writes and compiles the source code
- Specifies the deployment descriptor (optional)
- Packages the `.class` files and deployment descriptor into the EJB JAR file

**Web Component Developer**

A web component developer performs the following tasks to deliver a WAR file containing one or more web components:

- Writes and compiles servlet source code
- Writes JavaServer Faces, JSP, and HTML files
- Specifies the deployment descriptor (optional)
- Packages the `.class`, `.jsp`, and `.html` files and deployment descriptor into the WAR file
Development Roles

**Application Client Developer**

An application client developer performs the following tasks to deliver a JAR file containing the application client:

- Writes and compiles the source code
- Specifies the deployment descriptor for the client (optional)
- Packages the `.class` files and deployment descriptor into the JAR file

**Application Assembler**

The application assembler is the company or person who receives application modules from component providers and may assemble them into a Java EE application EAR file. The assembler or deployer can edit the deployment descriptor directly or can use tools that correctly add XML tags according to interactive selections.

A software developer performs the following tasks to deliver an EAR file containing the Java EE application:

- Assembles EJB JAR and WAR files created in the previous phases into a Java EE application (EAR) file
- Specifies the deployment descriptor for the Java EE application (optional)
- Verifies that the contents of the EAR file are well formed and comply with the Java EE specification

**Application Deployer and Administrator**

The application deployer and administrator is the company or person who configures and deploys application clients, web applications, Enterprise JavaBeans components, and Java EE applications, administers the computing and networking infrastructure where Java EE components and applications run, and oversees the runtime environment. Duties include setting transaction controls and security attributes and specifying connections to databases.

During configuration, the deployer follows instructions supplied by the application component provider to resolve external dependencies, specify security settings, and assign transaction attributes. During installation, the deployer moves the application components to the server and generates the container-specific classes and interfaces.
A deployer or system administrator performs the following tasks to install and configure a Java EE application or components:

- Configures the Java EE application or components for the operational environment
- Verifies that the contents of the EAR, JAR, and/or WAR files are well formed and comply with the Java EE specification
- Deploys (installs) the Java EE application or components into the Java EE server

Java EE 6 APIs

Figure 1–7 shows the relationships among the Java EE containers.

Figure 1–8 shows the availability of the Java EE 6 APIs in the web container.
Figure 1–9 shows the availability of the Java EE 6 APIs in the EJB container.
Figure 1–10 shows the availability of the Java EE 6 APIs in the application client container.
The following sections give a brief summary of the technologies required by the Java EE platform and the APIs used in Java EE applications.

### Enterprise JavaBeans Technology

An Enterprise JavaBeans (EJB) component, or enterprise bean, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server.

Enterprise beans are either session beans or message-driven beans.

- A **session bean** represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone.

- A **message-driven bean** combines features of a session bean and a message listener, allowing a business component to receive messages asynchronously. Commonly, these are Java Message Service (JMS) messages.
In the Java EE 6 platform, new enterprise bean features include the following:

- The ability to package local enterprise beans in a WAR file
- Singleton session beans, which provide easy access to shared state
- A lightweight subset of Enterprise JavaBeans functionality (EJB Lite) that can be provided within Java EE Profiles, such as the Java EE Web Profile.

The Java EE 6 platform requires Enterprise JavaBeans 3.1 and Interceptors 1.1. The Interceptors specification, which is part of the EJB 3.1 specification, makes more generally available the interceptor facility originally defined as part of the EJB 3.0 specification.

**Java Servlet Technology**

Java Servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

In the Java EE 6 platform, new Java Servlet technology features include the following:

- Annotation support
- Asynchronous support
- Ease of configuration
- Enhancements to existing APIs
- Pluggability

The Java EE 6 platform requires Servlet 3.0.

**JavaServer Faces Technology**

JavaServer Faces technology is a user interface framework for building web applications. The main components of JavaServer Faces technology are as follows:

- A GUI component framework.
- A flexible model for rendering components in different kinds of HTML or different markup languages and technologies. A Renderer object generates the markup to render the component and converts the data stored in a model object to types that can be represented in a view.
- A standard RenderKit for generating HTML/4.01 markup.

The following features support the GUI components:

- Input validation
- Event handling
Data conversion between model objects and components
Managed model object creation
Page navigation configuration
Expression Language (EL)

All this functionality is available using standard Java APIs and XML-based configuration files.

In the Java EE 6 platform, new features of JavaServer Faces include the following:

- The ability to use annotations instead of a configuration file to specify managed beans and other components
- Facelets, a display technology that replaces JavaServer Pages (JSP) technology using XHTML files
- Ajax support
- Composite components
- Implicit navigation

The Java EE 6 platform requires JavaServer Faces 2.0 and Expression Language 2.2.

**JavaServer Pages Technology**

JavaServer Pages (JSP) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text:

- Static data, which can be expressed in any text-based format such as HTML or XML
- JSP elements, which determine how the page constructs dynamic content

For information about JSP technology, see the *The Java EE 5 Tutorial* at [http://docs.oracle.com/javaee/5/tutorial/doc/](http://docs.oracle.com/javaee/5/tutorial/doc/).

The Java EE 6 platform requires JavaServer Pages 2.2 for compatibility with earlier releases, but recommends the use of Facelets as the display technology in new applications.

**JavaServer Pages Standard Tag Library**

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in your JSP applications, you use a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of the tags is optimized.

JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.
The Java EE 6 platform requires JSTL 1.2.

Java Persistence API

The Java Persistence API (JPA) is a Java standards-based solution for persistence. Persistence uses an object/relational mapping approach to bridge the gap between an object-oriented model and a relational database. The Java Persistence API can also be used in Java SE applications, outside of the Java EE environment. Java Persistence consists of the following areas:

- The Java Persistence API
- The query language
- Object/relational mapping metadata

The Java EE 6 platform requires Java Persistence API 2.0.

Java Transaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The Java EE architecture provides a default auto commit to handle transaction commits and rollbacks. An auto commit means that any other applications that are viewing data will see the updated data after each database read or write operation. However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

The Java EE 6 platform requires Java Transaction API 1.1.

Java API for RESTful Web Services

The Java API for RESTful Web Services (JAX-RS) defines APIs for the development of web services built according to the Representational State Transfer (REST) architectural style. A JAX-RS application is a web application that consists of classes packaged as a servlet in a WAR file along with required libraries.

The JAX-RS API is new to the Java EE 6 platform. The Java EE 6 platform requires JAX–RS 1.1.

Managed Beans

Managed Beans, lightweight container-managed objects (POJOs) with minimal requirements, support a small set of basic services, such as resource injection, lifecycle callbacks, and
interceptors. Managed Beans represent a generalization of the managed beans specified by JavaServer Faces technology and can be used anywhere in a Java EE application, not just in web modules.

The Managed Beans specification is part of the Java EE 6 platform specification (JSR 316).

Managed Beans are new to the Java EE 6 platform. The Java EE 6 platform requires Managed Beans 1.0.

**Contexts and Dependency Injection for the Java EE Platform (JSR 299)**

Contexts and Dependency Injection (CDI) for the Java EE platform defines a set of contextual services, provided by Java EE containers, that make it easy for developers to use enterprise beans along with JavaServer Faces technology in web applications. Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate different kinds of components in a loosely coupled but type-safe way.

CDI is new to the Java EE 6 platform. The Java EE 6 platform requires CDI 1.0.

**Dependency Injection for Java (JSR 330)**

Dependency Injection for Java defines a standard set of annotations (and one interface) for use on injectable classes.

In the Java EE platform, CDI provides support for Dependency Injection. Specifically, you can use DI injection points only in a CDI-enabled application.

Dependency Injection for Java is new to the Java EE 6 platform. The Java EE 6 platform requires Dependency Injection for Java 1.0.

**Bean Validation**

The Bean Validation specification defines a metadata model and API for validating data in JavaBeans components. Instead of distributing validation of data over several layers, such as the browser and the server side, you can define the validation constraints in one place and share them across the different layers.

Bean Validation is new to the Java EE 6 platform. The Java EE 6 platform requires Bean Validation 1.0.
Java Message Service API

The Java Message Service (JMS) API is a messaging standard that allows Java EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

The Java EE 6 platform requires JMS 1.1.

Java EE Connector Architecture

The Java EE Connector architecture is used by tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged in to any Java EE product. A resource adapter is a software component that allows Java EE application components to access and interact with the underlying resource manager of the EIS. Because a resource adapter is specific to its resource manager, a different resource adapter typically exists for each type of database or enterprise information system.

The Java EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of Java EE based web services with existing EISs that can be either synchronous or asynchronous. Existing applications and EISs integrated through the Java EE Connector architecture into the Java EE platform can be exposed as XML-based web services by using JAX-WS and Java EE component models. Thus JAX-WS and the Java EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

The Java EE 6 platform requires Java EE Connector architecture 1.6.

JavaMail API

Java EE applications use the JavaMail API to send email notifications. The JavaMail API has two parts:

- An application-level interface used by the application components to send mail
- A service provider interface

The Java EE platform includes the JavaMail API with a service provider that allows application components to send Internet mail.

The Java EE 6 platform requires JavaMail 1.4.

Java Authorization Contract for Containers

The Java Authorization Contract for Containers (JACC) specification defines a contract between a Java EE application server and an authorization policy provider. All Java EE containers support this contract.
The JACC specification defines `java.security.Permission` classes that satisfy the Java EE authorization model. The specification defines the binding of container-access decisions to operations on instances of these permission classes. It defines the semantics of policy providers that use the new permission classes to address the authorization requirements of the Java EE platform, including the definition and use of roles.

The Java EE 6 platform requires JACC 1.4.

**Java Authentication Service Provider Interface for Containers**

The Java Authentication Service Provider Interface for Containers (JASPIC) specification defines a service provider interface (SPI) by which authentication providers that implement message authentication mechanisms may be integrated in client or server message-processing containers or runtimes. Authentication providers integrated through this interface operate on network messages provided to them by their calling containers. The authentication providers transform outgoing messages so that the source of each message can be authenticated by the receiving container, and the recipient of the message can be authenticated by the message sender. Authentication providers authenticate each incoming message and return to their calling containers the identity established as a result of the message authentication.

JASPIC is new to the Java EE 6 platform. The Java EE 6 platform requires JASPIC 1.0.

**Java EE 6 APIs in the Java Platform, Standard Edition 6 and 7**

Several APIs that are required by the Java EE 6 platform are included in the Java Platform, Standard Edition 6 and 7 (Java SE 6 and 7) and are thus available to Java EE applications.

**Java Database Connectivity API**

The Java Database Connectivity (JDBC) API lets you invoke SQL commands from Java programming language methods. You use the JDBC API in an enterprise bean when you have a session bean access the database. You can also use the JDBC API from a servlet or a JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts:

- An application-level interface used by the application components to access a database
- A service provider interface to attach a JDBC driver to the Java EE platform

The Java SE 6 platform requires JDBC 4.0.
Java Naming and Directory Interface API

The Java Naming and Directory Interface (JNDI) API provides naming and directory functionality, enabling applications to access multiple naming and directory services such as LDAP, DNS, and NIS. The JNDI API provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, a Java EE application can store and retrieve any type of named Java object, allowing Java EE applications to coexist with many legacy applications and systems.

Java EE naming services provide application clients, enterprise beans, and web components with access to a JNDI naming environment. A *naming environment* allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI *naming context*.

A Java EE component can locate its environment naming context by using JNDI interfaces. A component can create a `javax.naming.InitialContext` object and look up the environment naming context in `InitialContext` under the name `java:comp/env`. A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.

A Java EE component can access named system-provided and user-defined objects. The names of system-provided objects, such as JTA `UserTransaction` objects, are stored in the environment naming context `java:comp/env`. The Java EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC `DataSource` objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named within the subcontext `java:comp/env.ejb`, and JDBC `DataSource` references are named within the subcontext `java:comp/env/jdbc`.

JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is used by the JavaMail API. JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

Java API for XML Processing

The Java API for XML Processing (JAXP), part of the Java SE platform, supports the processing of XML documents using Document Object Model (DOM), Simple API for XML (SAX), and
Extensible Stylesheet Language Transformations (XSLT). JAXP enables applications to parse and transform XML documents independently of a particular XML processing implementation.

JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts. Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the Worldwide Web Consortium (W3C) schema. You can find information on the W3C schema at this URL: http://www.w3.org/XML/Schema.

Java Architecture for XML Binding

The Java Architecture for XML Binding (JAXB) provides a convenient way to bind an XML schema to a representation in Java language programs. JAXB can be used independently or in combination with JAX-WS, where it provides a standard data binding for web service messages. All Java EE application client containers, web containers, and EJB containers support the JAXB API.

The Java EE 6 platform requires JAXB 2.2.

SOAP with Attachments API for Java

The SOAP with Attachments API for Java (SAAJ) is a low-level API on which JAX-WS depends. SAAJ enables the production and consumption of messages that conform to the SOAP 1.1 and 1.2 specifications and SOAP with Attachments note. Most developers do not use the SAAJ API, instead using the higher-level JAX-WS API.

Java API for XML Web Services

The Java API for XML Web Services (JAX-WS) specification provides support for web services that use the JAXB API for binding XML data to Java objects. The JAX-WS specification defines client APIs for accessing web services as well as techniques for implementing web service endpoints. The Implementing Enterprise Web Services specification describes the deployment of JAX-WS-based services and clients. The EJB and Java Servlet specifications also describe aspects of such deployment. JAX-WS-based applications can be deployed using any of these deployment models.

The JAX-WS specification describes the support for message handlers that can process message requests and responses. In general, these message handlers execute in the same container and with the same privileges and execution context as the JAX-WS client or endpoint component with which they are associated. These message handlers have access to the same JNDI java:comp/env namespace as their associated component. Custom serializers and deserializers, if supported, are treated in the same way as message handlers.
The Java EE 6 platform requires JAX-WS 2.2.

## Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) provides a way for a Java EE application to authenticate and authorize a specific user or group of users to run it.

JAAS is a Java programming language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java Platform security architecture to support user-based authorization.

### GlassFish Server Tools

The GlassFish Server is a compliant implementation of the Java EE 6 platform. In addition to supporting all the APIs described in the previous sections, the GlassFish Server includes a number of Java EE tools that are not part of the Java EE 6 platform but are provided as a convenience to the developer.

This section briefly summarizes the tools that make up the GlassFish Server. Instructions for starting and stopping the GlassFish Server, starting the Administration Console, and starting and stopping the Java DB server are in Chapter 2, “Using the Tutorial Examples.”

The GlassFish Server contains the tools listed in Table 1–1. Basic usage information for many of the tools appears throughout the tutorial. For detailed information, see the online help in the GUI tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration Console</td>
<td>A web-based GUI GlassFish Server administration utility. Used to stop the GlassFish Server and to manage users, resources, and applications.</td>
</tr>
<tr>
<td>asadmin</td>
<td>A command-line GlassFish Server administration utility. Used to start and stop the GlassFish Server and to manage users, resources, and applications.</td>
</tr>
<tr>
<td>appclient</td>
<td>A command-line tool that launches the application client container and invokes the client application packaged in the application client JAR file.</td>
</tr>
<tr>
<td>capture-schema</td>
<td>A command-line tool to extract schema information from a database, producing a schema file that the GlassFish Server can use for container-managed persistence.</td>
</tr>
<tr>
<td>package-appclient</td>
<td>A command-line tool to package the application client container libraries and JAR files.</td>
</tr>
<tr>
<td>Java DB database</td>
<td>A copy of the Java DB server.</td>
</tr>
<tr>
<td>Tool</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>xjc</td>
<td>A command-line tool to transform, or bind, a source XML schema to a set of JAXB content classes in the Java programming language.</td>
</tr>
<tr>
<td>schemagen</td>
<td>A command-line tool to create a schema file for each namespace referenced in your Java classes.</td>
</tr>
<tr>
<td>wsimport</td>
<td>A command-line tool to generate JAX-WS portable artifacts for a given WSDL file. After generation, these artifacts can be packaged in a WAR file with the WSDL and schema documents, along with the endpoint implementation, and then deployed.</td>
</tr>
<tr>
<td>wsgen</td>
<td>A command-line tool to read a web service endpoint class and generate all the required JAX-WS portable artifacts for web service deployment and invocation.</td>
</tr>
</tbody>
</table>
Using the Tutorial Examples

This chapter tells you everything you need to know to install, build, and run the tutorial examples.

The following topics are addressed here:

- “Required Software” on page 69
- “Starting and Stopping the GlassFish Server” on page 73
- “Starting the Administration Console” on page 74
- “Starting and Stopping the Java DB Server” on page 75
- “Building the Examples” on page 75
- “Tutorial Example Directory Structure” on page 76
- “Getting the Latest Updates to the Tutorial” on page 77
- “Debugging Java EE Applications” on page 77

Required Software

The following software is required to run the examples:

- “Java Platform, Standard Edition” on page 69
- “Java EE 6 Software Development Kit” on page 70
- “Java EE 6 Tutorial Component” on page 70
- “NetBeans IDE” on page 71
- “Apache Ant” on page 72

Java Platform, Standard Edition

To build, deploy, and run the examples, you need a copy of the Java Platform, Standard Edition 6.0 Development Kit (JDK 6) or the Java Platform, Standard Edition 7.0 Development Kit (JDK 7). You can download the JDK 6 or JDK 7 software from http://www.oracle.com/technetwork/java/javase/downloads/index.html.
Download the current JDK update that does not include any other software, such as NetBeans IDE or the Java EE SDK.

**Java EE 6 Software Development Kit**

GlassFish Server Open Source Edition 3.1.2 is targeted as the build and runtime environment for the tutorial examples. To build, deploy, and run the examples, you need a copy of the GlassFish Server and, optionally, NetBeans IDE. To obtain the GlassFish Server, you must install the Java EE 6 Software Development Kit (SDK), which you can download from [http://www.oracle.com/technetwork/java/javaee/downloads/index.html](http://www.oracle.com/technetwork/java/javaee/downloads/index.html). Make sure you download the Java EE 6 SDK, not the Java EE 6 Web Profile SDK.

**SDK Installation Tips**

During the installation of the SDK, do the following:

- Allow the installer to download and configure the Update Tool. If you access the Internet through a firewall, provide the proxy host and port.
- Configure the GlassFish Server administration user name as admin, and specify no password. This is the default setting.
- Accept the default port values for the Admin Port (4848) and the HTTP Port (8080).
- Do not select the check box to create an operating system service for the domain.

You can leave the check box to start the domain after creation selected if you wish, but this is not required.

This tutorial refers to `as-install-parent`, the directory where you install the GlassFish Server. For example, the default installation directory on Microsoft Windows is `C:\glassfish3`, so `as-install-parent` is `C:\glassfish3`. The GlassFish Server itself is installed in `as-install`, the `glassfish` directory under `as-install-parent`. So on Microsoft Windows, `as-install` is `C:\glassfish3\glassfish`.

After you install the GlassFish Server, add the following directories to your PATH to avoid having to specify the full path when you use commands:

```
as-install-parent/bin
```

```
as-install/bin
```

**Java EE 6 Tutorial Component**

The tutorial example source is contained in the tutorial component. To obtain the tutorial component, use the Update Tool.
To Obtain the Tutorial Component Using the Update Tool

1 Start the Update Tool by doing one of the following:
   - From the command line, type the command `updatetool`.
   - On a Windows system, from the Start menu, select All Programs, then select Java EE 6 SDK, then select Start Update Tool.

2 Expand the Java EE 6 SDK node.

3 Select the Available Updates node.

4 From the list, select the Java EE 6 Tutorial checkbox.

5 Click Install.

6 Accept the license agreement.

   After installation, the Java EE 6 Tutorial appears in the list of installed components. The tool is installed in the `as-install/docs/javaee-tutorial` directory. This directory contains two subdirectories: docs and examples. The examples directory contains subdirectories for each of the technologies discussed in the tutorial.

Next Steps Updates to the Java EE 6 Tutorial are published periodically. For details on obtaining these updates, see “Getting the Latest Updates to the Tutorial” on page 77.

NetBeans IDE

The NetBeans integrated development environment (IDE) is a free, open-source IDE for developing Java applications, including enterprise applications. NetBeans IDE supports the Java EE platform. You can build, package, deploy, and run the tutorial examples from within NetBeans IDE.

To run the tutorial examples, you need the latest version of NetBeans IDE. You can download NetBeans IDE from [http://www.netbeans.org/downloads/index.html](http://www.netbeans.org/downloads/index.html). Make sure that you download the Java EE bundle.

To Install NetBeans IDE without GlassFish Server

When you install NetBeans IDE, do not install the version of GlassFish Server that comes with NetBeans IDE. To skip the installation of GlassFish Server, follow these steps.

1 On the first page of the NetBeans IDE Installer wizard, deselect the check box for GlassFish Server and click OK.
2 Accept both the License Agreement and the Junit License Agreement.
   A few of the tutorial examples use the Junit library, so you should install it.

3 Continue with the installation of NetBeans IDE.

To Add GlassFish Server as a Server in NetBeans IDE
To run the tutorial examples in NetBeans IDE, you must add your GlassFish Server as a server in NetBeans IDE. Follow these instructions to add the GlassFish Server to NetBeans IDE.

1 From the Tools menu, select Servers.
   The Servers wizard opens.

2 Click Add Server.

3 Under Choose Server, select GlassFish Server 3+ and click Next.

4 Under Server Location, browse to the location of the Java EE 6 SDK and click Next.

5 Under Domain Location, select Register Local Domain.

6 Click Finish.

Apache Ant
Ant is a Java technology-based build tool developed by the Apache Software Foundation (http://ant.apache.org/) and is used to build, package, and deploy the tutorial examples. To run the tutorial examples, you need Ant 1.7.1 or higher. If you do not already have Ant, you can install it from the Update Tool that is part of the GlassFish Server.

To Obtain Apache Ant

1 Start the Update Tool.
   ■ From the command line, type the command update tool.
   ■ On a Windows system, from the Start menu, select All Programs, then select Java EE 6 SDK, then select Start Update Tool.

2 Expand the Java EE 6 SDK node.

3 Select the Available Add-ons node.
4 From the list, select the Apache Ant Build Tool check box.

5 Click Install.

6 Accept the license agreement. After installation, Apache Ant appears in the list of installed components. The tool is installed in the `as-install-parent/ant` directory.

**Next Steps** To use the ant command, add `as-install-parent/ant/bin` to your PATH environment variable.

---

**Starting and Stopping the GlassFish Server**

To start the GlassFish Server from the command line, open a terminal window or command prompt and execute the following:

```
asadmin start-domain --verbose
```

A domain is a set of one or more GlassFish Server instances managed by one administration server. Associated with a domain are the following:

- The GlassFish Server’s port number. The default is 8080.
- The administration server’s port number. The default is 4848.
- An administration user name and password. The default user name is admin, and by default no password is required.

You specify these values when you install the GlassFish Server. The examples in this tutorial assume that you chose the default ports as well as the default user name and lack of password.

With no arguments, the `start-domain` command initiates the default domain, which is `domain1`. The `--verbose` flag causes all logging and debugging output to appear on the terminal window or command prompt. The output also goes into the server log, which is located in `domain-dir/logs/server.log`.

Or, on Windows, from the Start menu, select All Programs, then select Java EE 6 SDK, then select Start Application Server.

To stop the GlassFish Server, open a terminal window or command prompt and execute:

```
asadmin stop-domain domain1
```

Or, on Windows, from the Start menu, select All Programs, then select Java EE 6 SDK, then select Stop Application Server.
To Start the GlassFish Server Using NetBeans IDE

1 Click the Services tab.
2 Expand the Servers node.
3 Right-click the GlassFish Server instance and select Start.

Next Steps: To stop the GlassFish Server using NetBeans IDE, right-click the GlassFish Server instance and select Stop.

Starting the Administration Console

To administer the GlassFish Server and manage users, resources, and Java EE applications, use the Administration Console tool. The GlassFish Server must be running before you invoke the Administration Console. To start the Administration Console, open a browser at http://localhost:4848/.

Or, on Windows, from the Start menu, select All Programs, then select Java EE 6 SDK, then select Administration Console.

To Start the Administration Console Using NetBeans IDE

1 Click the Services tab.
2 Expand the Servers node.
3 Right-click the GlassFish Server instance and select View Domain Admin Console.

Note – NetBeans IDE uses your default web browser to open the Administration Console.
Starting and Stopping the Java DB Server

The GlassFish Server includes the Java DB database server.

To start the Java DB server from the command line, open a terminal window or command prompt and execute:

```
asadmin start-database
```

To stop the Java DB server from the command line, open a terminal window or command prompt and execute:

```
asadmin stop-database
```

For information about the Java DB included with the GlassFish Server, see http://www.oracle.com/technetwork/java/javadb/overview/index.html.

To Start the Database Server Using NetBeans IDE

When you start the GlassFish Server using NetBeans IDE, the database server starts automatically. If you ever need to start the database server manually, follow these steps.

1. Click the Services tab.
2. Expand the Databases node.
3. Right-click Java DB and select Start Server.

Next Steps

To stop the database using NetBeans IDE, right-click Java DB and select Stop Server.

Building the Examples

The tutorial examples are distributed with a configuration file for either NetBeans IDE or Ant. Either NetBeans IDE or Ant may be used to build, package, deploy, and run the examples. Directions for building the examples are provided in each chapter.
To facilitate iterative development and keep application source separate from compiled files, the tutorial examples use the Java BluePrints application directory structure.

Each application module has the following structure:

- **build.xml**: Ant build file
- **src/java**: Java source files for the module
- **src/conf**: configuration files for the module, with the exception of web applications
- **web**: web pages, style sheets, tag files, and images (web applications only)
- **web/WEB-INF**: configuration files for web applications (web applications only)
- **nbproject**: NetBeans project files

When an example has multiple application modules packaged into an EAR file, its submodule directories use the following naming conventions:

- **example-name-app-client**: application clients
- **example-name-ejb**: enterprise bean JAR files
- **example-name-war**: web applications

The Ant build files (build.xml) distributed with the examples contain targets to create a build subdirectory and to copy and compile files into that directory; a dist subdirectory, which holds the packaged module file; and a client-jar directory, which holds the retrieved application client JAR.

The tut-install/examples/bp-project/ directory contains additional Ant targets called by the build.xml file targets.

For some web examples, an Ant target will open the example URL in a browser if one is available. This happens automatically on Windows systems. If you are running on a UNIX system, you may want to modify a line in the tut-install/examples/bp-project/build.properties file. Remove the comment character from the line specifying the default.browser property and specify the path to the command that invokes a browser. If you do not make the change, you can open the URL in the browser yourself.
Getting the Latest Updates to the Tutorial

Check for any updates to the tutorial by using the Update Center included with the Java EE 6 SDK.

▼ To Update the Tutorial through the Update Center

1. Open the Services tab in NetBeans IDE and expand Servers.
2. Right-click the GlassFish Server instance and select View Update Center to display the Update Tool.
3. Select Available Updates in the tree to display a list of updated packages.
4. Look for updates to the Java EE 6 Tutorial (javaee-tutorial) package.
5. If there is an updated version of the Tutorial, select Java EE 6 Tutorial (javaee-tutorial) and click Install.

Debugging Java EE Applications

This section explains how to determine what is causing an error in your application deployment or execution.

Using the Server Log

One way to debug applications is to look at the server log in domain-dir/logs/server.log. The log contains output from the GlassFish Server and your applications. You can log messages from any Java class in your application with System.out.println and the Java Logging APIs (documented at http://docs.oracle.com/javase/6/docs/technotes/guides/logging/index.html) and from web components with the ServletContext.log method.

If you use NetBeans IDE, logging output appears in the Output window as well as the server log.

If you start the GlassFish Server with the -verbose flag, all logging and debugging output will appear on the terminal window or command prompt and the server log. If you start the GlassFish Server in the background, debugging information is available only in the log. You can view the server log with a text editor or with the Administration Console log viewer.
To Use the Administration Console Log Viewer

1. Select the GlassFish Server node.

2. Click the View Log Files button.
   The log viewer opens and displays the last 40 entries.

3. To display other entries, follow these steps.
   a. Click the Modify Search button.
   b. Specify any constraints on the entries you want to see.
   c. Click the Search button at the top of the log viewer.

Using a Debugger

The GlassFish Server supports the Java Platform Debugger Architecture (JPDA). With JPDA, you can configure the GlassFish Server to communicate debugging information using a socket.

To Debug an Application Using a Debugger

1. Enable debugging in the GlassFish Server using the Administration Console:
   a. Expand the Configurations node, then expand the server-config node.
   b. Select the JVM Settings node. The default debug options are set to:
      
      -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=9009
      
      As you can see, the default debugger socket port is 9009. You can change it to a port not in use by the GlassFish Server or another service.
   c. Select the Debug Enabled check box.
   d. Click the Save button.

2. Stop the GlassFish Server and then restart it.
PART II

The Web Tier

Part II explores the technologies in the web tier. This part contains the following chapters:

- Chapter 3, “Getting Started with Web Applications”
- Chapter 4, “JavaServer Faces Technology”
- Chapter 5, “Introduction to Facelets”
- Chapter 6, “Expression Language”
- Chapter 7, “Using JavaServer Faces Technology in Web Pages”
- Chapter 8, “Using Converters, Listeners, and Validators”
- Chapter 9, “Developing with JavaServer Faces Technology”
- Chapter 10, “JavaServer Faces Technology: Advanced Concepts”
- Chapter 11, “Using Ajax with JavaServer Faces Technology”
- Chapter 12, “Composite Components: Advanced Topics and Example”
- Chapter 13, “Creating Custom UI Components and Other Custom Objects”
- Chapter 14, “Configuring JavaServer Faces Applications”
- Chapter 15, “Java Servlet Technology”
- Chapter 16, “Uploading Files with Java Servlet Technology”
- Chapter 17, “Internationalizing and Localizing Web Applications”
A web application is a dynamic extension of a web or application server. Web applications are of the following types:

- **Presentation-oriented**: A presentation-oriented web application generates interactive web pages containing various types of markup language (HTML, XHTML, XML, and so on) and dynamic content in response to requests. Development of presentation-oriented web applications is covered in Chapter 4, “JavaServer Faces Technology,” through Chapter 9, “Developing with JavaServer Faces Technology.”


The following topics are addressed here:

- “Web Applications” on page 81
- “Web Application Lifecycle” on page 83
- “Web Modules: The hello1 Example” on page 84
- “Configuring Web Applications: The hello2 Example” on page 93
- “Further Information about Web Applications” on page 101

### Web Applications

In the Java EE platform, web components provide the dynamic extension capabilities for a web server. Web components can be Java servlets, web pages implemented with JavaServer Faces technology, web service endpoints, or JSP pages. Figure 3–1 illustrates the interaction between a web client and a web application that uses a servlet. The client sends an HTTP request to the web server. A web server that implements Java Servlet and JavaServer Pages technology converts the request into an HttpServletRequest object. This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic
content. The web component can then generate an `HttpServletResponse` or can pass the request to another web component. A web component eventually generates a `HttpServletResponse` object. The web server converts this object to an HTTP response and returns it to the client.

![Java Web Application Request Handling](image)

Servlets are Java programming language classes that dynamically process requests and construct responses. Java technologies, such as JavaServer Faces and Facelets, are used for building interactive web applications. (Frameworks can also be used for this purpose.) Although servlets and Java Server Faces and Facelets pages can be used to accomplish similar things, each has its own strengths. Servlets are best suited for service-oriented applications (web service endpoints can be implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. Java Server Faces and Facelets pages are more appropriate for generating text-based markup, such as XHTML, and are generally used for presentation-oriented applications.

Web components are supported by the services of a runtime platform called a web container. A web container provides such services as request dispatching, security, concurrency, and lifecycle management. A web container also gives web components access to such APIs as naming, transactions, and email.

Certain aspects of web application behavior can be configured when the application is installed, or deployed, to the web container. The configuration information can be specified using Java EE annotations or can be maintained in a text file in XML format called a web application deployment descriptor (DD). A web application DD must conform to the schema described in the Java Servlet specification.
This chapter gives a brief overview of the activities involved in developing web applications. First, it summarizes the web application lifecycle and explains how to package and deploy very simple web applications on the GlassFish Server. The chapter moves on to configuring web applications and discusses how to specify the most commonly used configuration parameters.

## Web Application Lifecycle

A web application consists of web components; static resource files, such as images; and helper classes and libraries. The web container provides many supporting services that enhance the capabilities of web components and make them easier to develop. However, because a web application must take these services into account, the process for creating and running a web application is different from that of traditional stand-alone Java classes.

The process for creating, deploying, and executing a web application can be summarized as follows:

1. Develop the web component code.
2. Develop the web application deployment descriptor, if necessary.
3. Compile the web application components and helper classes referenced by the components.
4. Optionally, package the application into a deployable unit.
5. Deploy the application into a web container.
6. Access a URL that references the web application.

Developing web component code is covered in the later chapters. Steps 2 through 4 are expanded on in the following sections and illustrated with a Hello, World-style presentation-oriented application. This application allows a user to enter a name into an HTML form and then displays a greeting after the name is submitted.

The Hello application contains two web components that generate the greeting and the response. This chapter discusses the following simple applications:

- **hello1**, a JavaServer Faces technology-based application that uses two XHTML pages and a managed bean
- **hello2**, a servlet-based web application in which the components are implemented by two servlet classes

The applications are used to illustrate tasks involved in packaging, deploying, configuring, and running an application that contains web components. The source code for the examples is in the `tut-install/examples/web/hello1/` and `tut-install/examples/web/hello2/` directories.
Web Modules: The hello1 Example

In the Java EE architecture, a web module is the smallest deployable and usable unit of web resources. A web module contains web components and static web content files, such as images, which are called web resources. A Java EE web module corresponds to a web application as defined in the Java Servlet specification.

In addition to web components and web resources, a web module can contain other files:

- Server-side utility classes, such as shopping carts
- Client-side classes, such as applets and utility classes

A web module has a specific structure. The top-level directory of a web module is the document root of the application. The document root is where XHTML pages, client-side classes and archives, and static web resources, such as images, are stored.

The document root contains a subdirectory named WEB-INF, which can contain the following files and directories:

- classes: A directory that contains server-side classes: servlets, enterprise bean class files, utility classes, and JavaBeans components
- lib: A directory that contains JAR files that contain enterprise beans, and JAR archives of libraries called by server-side classes
- Deployment descriptors, such as web.xml (the web application deployment descriptor) and ejb-jar.xml (an EJB deployment descriptor)

A web module needs a web.xml file if it uses JavaServer Faces technology, if it must specify certain kinds of security information, or if you want to override information specified by web component annotations.

You can also create application-specific subdirectories (that is, package directories) in either the document root or the WEB-INF/classes directory.

A web module can be deployed as an unpacked file structure or can be packaged in a JAR file known as a Web Archive (WAR) file. Because the contents and use of WAR files differ from those of JAR files, WAR file names use a .war extension. The web module just described is portable; you can deploy it into any web container that conforms to the Java Servlet specification.

To deploy a WAR on the GlassFish Server, the file must contain a runtime deployment descriptor. The runtime DD is an XML file that contains such information as the context root of the web application and the mapping of the portable names of an application’s resources to the GlassFish Server’s resources. The GlassFish Server web application runtime DD is named glassfish-web.xml and is located in the WEB-INF directory. The structure of a web module that can be deployed on the GlassFish Server is shown in Figure 3–2.

For example, the glassfish-web.xml file for the hello1 application specifies the following context root:
Examining the hello1 Web Module

The hello1 application is a web module that uses JavaServer Faces technology to display a greeting and response. You can use a text editor to view the application files, or you can use NetBeans IDE.

▼ To View the hello1 Web Module Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   tut-install/examples/web/

3. Select the hello1 folder.

4. Select the Open as Main Project check box.
5 Expand the Web Pages node and double-click the index.xhtml file to view it in the right-hand pane.

The index.xhtml file is the default landing page for a Facelets application. For this application, the page uses simple tag markup to display a form with a graphic image, a header, a text field, and two command buttons:

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelets Hello Greeting</title>
  </h:head>
  <h:body>
    <h:form>
      <h:graphicImage url="duke.waving.gif" alt="Duke waving his hand"/>
      <h:inputText id="username"
        title="My name is:"
        value="#{hello.name}"
        required="true"
        requiredMessage="Error: A name is required."
        maxlength="25" />
      <p></p>
      <h:commandButton id="submit" value="Submit" action="response">
      </h:commandButton>
      <h:commandButton id="reset" value="Reset" type="reset">
      </h:commandButton>
    </h:form>
  </h:body>
</html>
```

The most complex element on the page is the inputText text field. The maxLength attribute specifies the maximum length of the field. The required attribute specifies that the field must be filled out; the requiredMessage attribute provides the error message to be displayed if the field is left empty. The title attribute provides the text to be used by screen readers for the visually disabled. Finally, the value attribute contains an expression that will be provided by the Hello managed bean.

The Submit commandButton element specifies the action as response, meaning that when the button is clicked, the response.xhtml page is displayed.

6 Double-click the response.xhtml file to view it.

The response page appears. Even simpler than the greeting page, the response page contains a graphic image, a header that displays the expression provided by the managed bean, and a single button whose action element transfers you back to the index.xhtml page:

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelets Hello Greeting</title>
  </h:head>
  <h:body>
    ... 
  </h:body>
</html>
```
Expand the Source Packages node, then the hello1 node.

Double-click the Hello .java file to view it.

The Hello class, called a managed bean class, provides getter and setter methods for the name property used in the Facelets page expressions. By default, the expression language refers to the class name, with the first letter in lowercase (hello.name).

package hello1;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
@ManagedBean
@RequestScoped
public class Hello {
    private String name;
    public Hello() {
    }
    public String getName() {
        return name;
    }
    public void setName(String user_name) {
        this.name = user_name;
    }
}

Under the Web Pages node, expand the WEB-INF node and double-click the web.xml file to view it.

The web.xml file contains several elements that are required for a Facelets application. All these are created automatically when you use NetBeans IDE to create an application:

- A context parameter specifying the project stage:

```xml
<context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```
A context parameter provides configuration information needed by a web application. An application can define its own context parameters. In addition, JavaServer Faces technology and Java Servlet technology define context parameters that an application can use.

- A servlet element and its servlet-mapping element specifying the FacesServlet:

```xml
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

- A welcome-file-list element specifying the location of the landing page; note that the location is faces/index.xhtml, not just index.xhtml:

```xml
<welcome-file-list>
  <welcome-file>faces/index.xhtml</welcome-file>
</welcome-file-list>
```

### Introduction to Scopes

In the Hello.java class, the annotations `javax.faces.bean.ManagedBean` and `javax.faces.bean.RequestScoped` identify the class as a JavaServer Faces managed bean using request scope. Scope defines how application data persists and is shared.

The most commonly used scopes in JavaServer Faces applications are the following:

- **Request (@RequestScoped):** Request scope persists during a single HTTP request in a web application. In an application like hello1, where the application consists of a single request and response, the bean uses request scope.

- **Session (@SessionScoped):** Session scope persists across multiple HTTP requests in a web application. When an application consists of multiple requests and responses where data needs to be maintained, beans use session scope.

- **Application (@ApplicationScoped):** Application scope persists across all users’ interactions with a web application.

For more information on scopes in JavaServer Faces technology, see “Using Managed Bean Scopes” on page 296.

### Packaging a Web Module

A web module must be packaged into a WAR in certain deployment scenarios and whenever you want to distribute the web module. You package a web module into a WAR by executing the jar command in a directory laid out in the format of a web module, by using the Ant utility, or by using the IDE tool of your choice. This tutorial shows you how to use NetBeans IDE or Ant to build, package, and deploy the hello1 sample application.
To Set the Context Root

A context root identifies a web application in a Java EE server. A context root must start with a forward slash (/) and end with a string.

In a packaged web module for deployment on the GlassFish Server, the context root is stored in glassfish-web.xml.

To view or edit the context root, follow these steps.

1. Expand the Web Pages and WEB-INF nodes of the hello1 project.
2. Double-click glassfish-web.xml.
3. In the General tab, observe that the Context Root field is set to /hello1. If you needed to edit this value, you could do so here. When you create a new application, you type the context root here.
4. (Optional) Click the XML tab. Observe that the context root value /hello1 is enclosed by the context-root element. You could also edit the value here.

To Build and Package the hello1 Web Module Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   tut-install/examples/web/
3. Select the hello1 folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the hello1 project and select Build.

To Build and Package the hello1 Web Module Using Ant

1. In a terminal window, go to:
   tut-install/examples/web/hello1/
2. Type the following command:
   ant
This command spawns any necessary compilations, copies files to the directory
\texttt{tut-install/examples/web/hello1/build/}, creates the WAR file, and copies it to the directory
\texttt{tut-install/examples/web/hello1/dist/}.

**Deploying a Web Module**

You can deploy a WAR file to the GlassFish Server by

- Using NetBeans IDE
- Using the Ant utility
- Using the\texttt{asadmin} command
- Using the Administration Console
- Copying the WAR file into the\texttt{domain-dir/autodeploy/} directory

Throughout the tutorial, you will use NetBeans IDE or Ant for packaging and deploying.

▼ **To Deploy the hello1 Web Module Using NetBeans IDE**

1. Right-click the hello1 project and select Deploy.

▼ **To Deploy the hello1 Web Module Using Ant**

1. In a terminal window, go to:
   \texttt{tut-install/examples/web/hello1/}

2. Type the following command:
   \texttt{ant deploy}

**Running a Deployed Web Module**

Now that the web module is deployed, you can view it by opening the application in a web browser. By default, the application is deployed to host\texttt{localhost} on port 8080. The context root of the web application is\texttt{hello1}.

▼ **To Run a Deployed Web Module**

1. Open a web browser.

2. Type the following URL:
   \texttt{http://localhost:8080/hello1/}

3. Type your name and click Submit.
   The response page displays the name you submitted. Click the Back button to try again.
Listing Deployed Web Modules

The GlassFish Server provides two ways to view the deployed web modules: the Administration Console and the asadmin command.

▼ To List Deployed Web Modules Using the Administration Console

2. Select the Applications node.
   
   The deployed web modules appear in the Deployed Applications table.

▼ To List Deployed Web Modules Using the asadmin Command

- Type the following command:
  
  `asadmin list-applications`

Updating a Web Module

A typical iterative development cycle involves deploying a web module and then making changes to the application components. To update a deployed web module, follow these steps.

▼ To Update a Deployed Web Module

1. Recompile any modified classes.
2. Redeploy the module.
3. Reload the URL in the client.

Dynamic Reloading

If dynamic reloading is enabled, you do not have to redeploy an application or module when you change its code or deployment descriptors. All you have to do is copy the changed pages or class files into the deployment directory for the application or module. The deployment directory for a web module named `context-root` is `domain-dir/applications/context-root`. The server checks for changes periodically and redeploy automatically and dynamically, with the changes.
This capability is useful in a development environment because it allows code changes to be tested quickly. Dynamic reloading is not recommended for a production environment, however, because it may degrade performance. In addition, whenever a reload is done, the sessions at that time become invalid, and the client must restart the session.

In the GlassFish Server, dynamic reloading is enabled by default.

▼ To Disable or Modify Dynamic Reloading
If for some reason you do not want the default dynamic reloading behavior, follow these steps in the Administration Console.

2. Select the GlassFish Server node.
3. Select the Advanced tab.
4. To disable dynamic reloading, deselect the Reload Enabled check box.
5. To change the interval at which applications and modules are checked for code changes and dynamically reloaded, type a number of seconds in the Reload Poll Interval field. The default value is 2 seconds.
6. Click the Save button.

Undeploying Web Modules
You can undeploy web modules and other types of enterprise applications by using either NetBeans IDE or the Ant tool.

▼ To Undeploy the hello1 Web Module Using NetBeans IDE
1. Ensure that the GlassFish Server is running.
2. In the Services window, expand the Servers node, GlassFish Server instance, and the Applications node.
3. Right-click the hello1 module and choose Undeploy.
4. To delete the class files and other build artifacts, right-click the project and choose Clean.
To Undeploy the hello1 Web Module Using Ant

1. In a terminal window, go to:
   `tut-install/examples/web/hello1/`
2. Type the following command:
   `ant undeploy`
3. To delete the class files and other build artifacts, type the following command:
   `ant clean`

Configuring Web Applications: The hello2 Example

Web applications are configured by means of annotations or by elements contained in the web application deployment descriptor.

The following sections give a brief introduction to the web application features you will usually want to configure. Examples demonstrate procedures for configuring the Hello, World application.

Mapping URLs to Web Components

When it receives a request, the web container must determine which web component should handle the request. The web container does so by mapping the URL path contained in the request to a web application and a web component. A URL path contains the context root and, optionally, a URL pattern:

```
http://host:port/context-root[/url-pattern]
```

You set the URL pattern for a servlet by using the `@WebServlet` annotation in the servlet source file. For example, the `GreetingServlet.java` file in the `hello2` application contains the following annotation, specifying the URL pattern as `/greeting`:

```
@WebServlet("/greeting")
public class GreetingServlet extends HttpServlet {

    ...

This annotation indicates that the URL pattern `/greeting` follows the context root. Therefore, when the servlet is deployed locally, it is accessed with the following URL:

```
http://localhost:8080/hello2/greeting
```

To access the servlet by using only the context root, specify `"/"` as the URL pattern.
Examining the hello2 Web Module

The hello2 application behaves almost identically to the hello1 application, but it is implemented using Java Servlet technology instead of JavaServer Faces technology. You can use a text editor to view the application files, or you can use NetBeans IDE.

▼ To View the hello2 Web Module Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   tut-install/examples/web/

3. Select the hello2 folder.

4. Select the Open as Main Project check box.

5. Expand the Source Packages node, then the servlets node.

6. Double-click the GreetingServlet.java file to view it.

This servlet overrides the doGet method, implementing the GET method of HTTP. The servlet displays a simple HTML greeting form whose Submit button, like that of hello1, specifies a response page for its action. The following excerpt begins with the @WebServlet annotation that specifies the URL pattern, relative to the context root:

```java
@WebServlet("/greeting")
public class GreetingServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        response.setBufferSize(8192);
        PrintWriter out = response.getWriter();

        // then write the data of the response
        out.println("<html lang="en">
            + "<head><title>Servlet Hello</title></head>
            + "<body bgcolor="#ffffff">
            + "<img src="duke.waving.gif" alt="Duke waving his hand">
            + "<h2>Hello, my name is Duke. What's yours?</h2>
            + "<input title="My name is: " type="text" name="username" size="25">
            + "<p>
            + "<input type="submit" value="Submit">
            + "</body>
            + "</html>"
        );
```
String username = request.getParameter("username");
if (username != null && username.length() > 0) {
    RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher("/response");
    if (dispatcher != null) {
        dispatcher.include(request, response);
    }
    out.println("<h2>Hello, " + username + "!</h2>";
    out.close();
}
...

7 Double-click the ResponseServlet.java file to view it.
This servlet also overrides the doGet method, displaying only the response. The following excerpt begins with the @WebServlet annotation, which specifies the URL pattern, relative to the context root:

```java
@WebServlet("/response")
public class ResponseServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        // then write the data of the response
        String username = request.getParameter("username");
        if (username != null && username.length() > 0) {
            out.println("<h2>Hello, " + username + "!</h2>";
        }
    }
    ...
```

8 Under the Web Pages node, expand the WEB-INF node and double-click the glassfish-web.xml file to view it.
In the General tab, observe that the Context Root field is set to /hello2.
For this simple servlet application, a web.xml file is not required.

### Running the hello2 Example
You can use either NetBeans IDE or Ant to build, package, deploy, and run the hello2 example.
To Run the hello2 Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   tut-install/examples/web/
3. Select the hello2 folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the hello2 project and select Build.
7. Right-click the project and select Deploy.
8. In a web browser, open the URL http://localhost:8080/hello2/greeting.

   The URL specifies the context root, followed by the URL pattern.
   The application looks much like the hello1 application. The major difference is that after you click the Submit button, the response appears below the greeting, not on a separate page.

To Run the hello2 Example Using Ant

1. In a terminal window, go to:
   tut-install/examples/web/hello2/
2. Type the following command:
   `ant`
   This target builds the WAR file and copies it to the tut-install/examples/web/hello2/dist/directory.
3. Type `ant deploy`.
   Ignore the URL shown in the deploy target output.
4. In a web browser, open the URL http://localhost:8080/hello2/greeting.

   The URL specifies the context root, followed by the URL pattern.
   The application looks much like the hello1 application. The major difference is that after you click the Submit button, the response appears below the greeting, not on a separate page.
Declaring Welcome Files

The welcome files mechanism allows you to specify a list of files that the web container will use for appending to a request for a URL (called a valid partial request) that is not mapped to a web component. For example, suppose that you define a welcome file welcome.html. When a client requests a URL such as host:port/webapp/directory, where directory is not mapped to a servlet or XHTML page, the file host:port/webapp/directory/welcome.html is returned to the client.

If a web container receives a valid partial request, the web container examines the welcome file list and appends to the partial request each welcome file in the order specified and checks whether a static resource or servlet in the WAR is mapped to that request URL. The web container then sends the request to the first resource that matches in the WAR.

If no welcome file is specified, the GlassFish Server will use a file named index.html as the default welcome file. If there is no welcome file and no file named index.html, the GlassFish Server returns a directory listing.

By convention, you specify the welcome file for a JavaServer Faces application as faces/file-name.xhtml.

Setting Context Parameters

The web components in a web module share an object that represents their application context. You can pass context parameters to the context, or initialization parameters to a servlet. Context parameters are available to the entire web application. For information on initialization parameters, see “Creating and Initializing a Servlet” on page 331.

▼ To Add a Context Parameter Using NetBeans IDE

These steps apply generally to web applications, but do not apply specifically to the examples in this chapter.

1 Open the project.

2 Expand the project’s node in the Projects pane.

3 Expand the Web Pages node and then the WEB-INF node.

4 Double-click web.xml.

   If the project does not have a web.xml file, follow the steps in “To Create a web.xml File Using NetBeans IDE” on page 98.

5 Click General at the top of the editor pane.

6 Expand the Context Parameters node.
Click Add.

An Add Context Parameter dialog opens.

In the Parameter Name field, type the name that specifies the context object.

In the Parameter Value field, type the parameter to pass to the context object.

Click OK.

▼ To Create a `web.xml` File Using NetBeans IDE

1 From the File menu, choose New File.

2 In the New File wizard, select the Web category, then select Standard Deployment Descriptor under File Types.

3 Click Next.

4 Click Finish.

A basic `web.xml` file appears in `web/WEB-INF/`.

**Mapping Errors to Error Screens**

When an error occurs during execution of a web application, you can have the application display a specific error screen according to the type of error. In particular, you can specify a mapping between the status code returned in an HTTP response or a Java programming language exception returned by any web component and any type of error screen.

You can have multiple `error-page` elements in your deployment descriptor. Each element identifies a different error that causes an error page to open. This error page can be the same for any number of `error-page` elements.

▼ To Set Up Error Mapping Using NetBeans IDE

These steps apply generally to web applications, but do not apply specifically to the examples in this chapter.

1 Open the project.

2 Expand the project’s node in the Projects pane.

3 Expand the Web Pages node and then the WEB-INF node.
Double-click `web.xml`. If the project does not have a `web.xml` file, follow the steps in “To Create a `web.xml` File Using NetBeans IDE” on page 98.

Click Pages at the top of the editor pane.

Expand the Error Pages node.

Click Add. The Add Error Page dialog opens.

Click Browse to locate the page that you want to act as the error page.

Specify either an error code or an exception type:

- To specify an error code, in the Error Code field, type the HTTP status code that will cause the error page to be opened, or leave the field blank to include all error codes.
- To specify an exception type, in the Exception Type field, type the exception that will cause the error page to load. To specify all throwable errors and exceptions, type `java.lang.Throwable`.

Click OK.

Declaring Resource References

If your web component uses such objects as enterprise beans, data sources, or web services, you use Java EE annotations to inject these resources into your application. Annotations eliminate a lot of the boilerplate lookup code and configuration elements that previous versions of Java EE required.

Although resource injection using annotations can be more convenient for the developer, there are some restrictions on using it in web applications. First, you can inject resources only into container-managed objects, since a container must have control over the creation of a component so that it can perform the injection into a component. As a result, you cannot inject resources into such objects as simple JavaBeans components. However, JavaServer Faces managed beans and CDI managed beans are managed by the container; therefore, they can accept resource injections.

Components that can accept resource injections are listed in Table 3–1.

This section explains how to use a couple of the annotations supported by a web container to inject resources. Chapter 33, “Running the Persistence Examples,” explains how web applications use annotations supported by the Java Persistence API. Chapter 40, “Getting
Started Securing Web Applications,” explains how to use annotations to specify information about securing web applications. See Chapter 45, “Resources and Resource Adapters,” for more information on resources.

### TABLE 3–1 Web Components That Accept Resource Injections

<table>
<thead>
<tr>
<th>Component</th>
<th>Interface/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servlets</td>
<td>javax.servlet.Servlet</td>
</tr>
<tr>
<td>Servlet filters</td>
<td>javax.servlet.ServletFilter</td>
</tr>
<tr>
<td>Event listeners</td>
<td>javax.servlet.ServletContextListener</td>
</tr>
<tr>
<td></td>
<td>javax.servlet.ServletContextAttributeListener</td>
</tr>
<tr>
<td></td>
<td>javax.servlet.ServletRequestListener</td>
</tr>
<tr>
<td></td>
<td>javax.servlet.ServletRequestAttributeListener</td>
</tr>
<tr>
<td></td>
<td>javax.servlet.http.HttpSessionListener</td>
</tr>
<tr>
<td></td>
<td>javax.servlet.http.HttpSessionAttributeListener</td>
</tr>
<tr>
<td></td>
<td>javax.servlet.http.HttpSessionBindingListener</td>
</tr>
<tr>
<td>Managed beans</td>
<td>Plain Old Java Objects</td>
</tr>
</tbody>
</table>

#### Declaring a Reference to a Resource

The `@Resource` annotation is used to declare a reference to a resource, such as a data source, an enterprise bean, or an environment entry.

The `@Resource` annotation is specified on a class, a method, or a field. The container is responsible for injecting references to resources declared by the `@Resource` annotation and mapping it to the proper JNDI resources.

In the following example, the `@Resource` annotation is used to inject a data source into a component that needs to make a connection to the data source, as is done when using JDBC technology to access a relational database:

```java
@Resource javax.sql.DataSource catalogDS;
public getProductsByCategory() {
    // get a connection and execute the query
    Connection conn = catalogDS.getConnection();
    ...
}
```

The container injects this data source prior to the component's being made available to the application. The data source JNDI mapping is inferred from the field name `catalogDS` and the type, `javax.sql.DataSource`.

If you have multiple resources that you need to inject into one component, you need to use the `@Resources` annotation to contain them, as shown by the following example:
The web application examples in this tutorial use the Java Persistence API to access relational databases. This API does not require you to explicitly create a connection to a data source. Therefore, the examples do not use the @Resource annotation to inject a data source. However, this API supports the @PersistenceUnit and @PersistenceContext annotations for injecting EntityManagerFactory and EntityManager instances, respectively. Chapter 33, “Running the Persistence Examples,” describes these annotations and the use of the Java Persistence API in web applications.

### Declaring a Reference to a Web Service

The @WebServiceRef annotation provides a reference to a web service. The following example shows uses the @WebServiceRef annotation to declare a reference to a web service. 

```java
import javax.xml.ws.WebServiceRef;

public class ResponseServlet extends HTTPServlet {
  @WebServiceRef(wsdlLocation=
                  "http://localhost:8080/helloservice/hello?wsdl")
  static HelloService service;
}
```

For more information on web applications, see

JavaServer Faces technology is a server-side component framework for building Java technology-based web applications.

JavaServer Faces technology consists of the following:

- An API for representing components and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all these features
- Tag libraries for adding components to web pages and for connecting components to server-side objects

JavaServer Faces technology provides a well-defined programming model and various tag libraries. The tag libraries contain tag handlers that implement the component tags. These features significantly ease the burden of building and maintaining web applications with server-side user interfaces (UIs). With minimal effort, you can complete the following tasks.

- Create a web page.
- Drop components onto a web page by adding component tags.
- Bind components on a page to server-side data.
- Wire component-generated events to server-side application code.
- Save and restore application state beyond the life of server requests.
- Reuse and extend components through customization.

This chapter provides an overview of JavaServer Faces technology. After explaining what a JavaServer Faces application is and reviewing some of the primary benefits of using JavaServer Faces technology, this chapter describes the process of creating a simple JavaServer Faces application. This chapter also introduces the JavaServer Faces lifecycle by describing the example JavaServer Faces application progressing through the lifecycle stages.

The following topics are addressed here:

- "What Is a JavaServer Faces Application?" on page 104
- "JavaServer Faces Technology Benefits" on page 105
What Is a JavaServer Faces Application?

The functionality provided by a JavaServer Faces application is similar to that of any other Java web application. A typical JavaServer Faces application includes the following parts:

- A set of web pages in which components are laid out
- A set of tags to add components to the web page
- A set of managed beans, which are lightweight container-managed objects (POJOs) with minimal requirements. They support a small set of basic services, such as resource injection, lifecycle callbacks and interceptors.
- A web deployment descriptor (web.xml file)
- Optionally, one or more application configuration resource files, such as a faces-config.xml file, which can be used to define page navigation rules and configure beans and other custom objects, such as custom components
- Optionally, a set of custom objects, which can include custom components, validators, converters, or listeners, created by the application developer
- Optionally, a set of custom tags for representing custom objects on the page

Figure 4–1 shows the interaction between client and server in a typical JavaServer Faces application. In response to a client request, a web page is rendered by the web container that implements JavaServer Faces technology.

![Figure 4-1](image-url)
The web page, myfacelet.xhtml, is built using JavaServer Faces component tags. Component tags are used to add components to the view (represented by myUI in the diagram), which is the server-side representation of the page. In addition to components, the web page can also reference objects, such as the following:

- Any event listeners, validators, and converters that are registered on the components
- The JavaBeans components that capture the data and process the application-specific functionality of the components

On request from the client, the view is rendered as a response. Rendering is the process whereby, based on the server-side view, the web container generates output, such as HTML or XHTML, that can be read by the client, such as a browser.

**JavaServer Faces Technology Benefits**

One of the greatest advantages of JavaServer Faces technology is that it offers a clean separation between behavior and presentation for web applications. A JavaServer Faces application can map HTTP requests to component-specific event handling and manage components as stateful objects on the server. JavaServer Faces technology allows you to build web applications that implement the finer-grained separation of behavior and presentation that is traditionally offered by client-side UI architectures.

The separation of logic from presentation also allows each member of a web application development team to focus on a single piece of the development process and provides a simple programming model to link the pieces. For example, page authors with no programming expertise can use JavaServer Faces technology tags in a web page to link to server-side objects without writing any scripts.

Another important goal of JavaServer Faces technology is to leverage familiar component and web-tier concepts without limiting you to a particular scripting technology or markup language. JavaServer Faces technology APIs are layered directly on top of the Servlet API, as shown in Figure 4–2.

**FIGURE 4-2  Java Web Application Technologies**
This layering of APIs enables several important application use cases, such as using different presentation technologies, creating your own custom components directly from the component classes, and generating output for various client devices.

Facelets technology, available as part of JavaServer Faces 2.0, is now the preferred presentation technology for building JavaServer Faces technology-based web applications. For more information on Facelets technology features, see Chapter 5, “Introduction to Facelets.”

Facelets technology offers several advantages.

- Code can be reused and extended for components through the templating and composite component features.
- When you use the JavaServer Faces Annotations feature, you can automatically register the managed bean as a resource available for JavaServer Faces applications. In addition, implicit navigation rules allow developers to quickly configure page navigation. These features reduce the manual configuration process for applications.
- Most important, JavaServer Faces technology provides a rich architecture for managing component state, processing component data, validating user input, and handling events.

Creating a Simple JavaServer Faces Application

JavaServer Faces technology provides an easy and user-friendly process for creating web applications. Developing a simple JavaServer Faces application typically requires the following tasks:

- Developing managed beans
- Creating web pages using component tags
- Mapping the javax.faces.webapp.FacesServlet instance

This section describes those tasks through the process of creating a simple JavaServer Faces Facelets application.

The example is a Hello application that includes a managed bean and a web page. When accessed by a client, the web page prints out a Hello World message. The example application is located in the tut-install/examples/web/hello/ directory. The tasks involved in developing this application can be examined by looking at the application components in detail.

Developing the Managed Bean

As mentioned earlier in this chapter, a managed bean is a lightweight container-managed object. Components in a page are associated with managed beans that provide application logic. The example managed bean, Hello.java, contains the following code:
package hello;

import javax.faces.bean.ManagedBean;

@ManagedBean
public class Hello {
    final String world = "Hello World!";
    
    public String getWorld() {
        return world;
    }
}

The example managed bean sets the value of the variable world with the string "Hello World!". The @ManagedBean annotation registers the managed bean as a resource with the JavaServer Faces implementation. For more information on managed beans and annotations, see Chapter 9, "Developing with JavaServer Faces Technology."

Creating the Web Page

In a typical Facelets application, web pages are created in XHTML. The example web page, beanhello.xhtml, is a simple XHTML page. It has the following content:

```
<html lang="en" xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Facelets Hello World</title>
    </h:head>
    <h:body>
        #{hello.world}
    </h:body>
</html>
```

A Facelets XHTML web page can also contain several other elements, which are covered later in this tutorial.

The web page connects to the managed bean through the Expression Language (EL) value expression #{hello.world}, which retrieves the value of the world property from the managed bean Hello. Note the use of hello to reference the managed bean Hello. If no name is specified in the @ManagedBean annotation, the managed bean is always accessed with the first letter of the class name in lowercase.

For more information on using EL expressions, see Chapter 6, “Expression Language.” For more information about Facelets technology, see Chapter 5, “Introduction to Facelets.” For more information about the JavaServer Faces programming model and building web pages using JavaServer Faces technology, see Chapter 7, “Using JavaServer Faces Technology in Web Pages.”
Mapping the FacesServlet Instance

The final task requires mapping the FacesServlet, which is done through the web deployment descriptor (web.xml). A typical mapping of FacesServlet is as follows:

```xml
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

The preceding file segment represents part of a typical JavaServer Faces web deployment descriptor. The web deployment descriptor can also contain other content relevant to a JavaServer Faces application configuration, but that information is not covered here.

Mapping the FacesServlet is automatically done for you if you are using an IDE such as NetBeans IDE.

The Lifecycle of the hello Application

Every web application has a lifecycle. Common tasks, such as handling incoming requests, decoding parameters, modifying and saving state, and rendering web pages to the browser, are all performed during a web application lifecycle. Some web application frameworks hide the details of the lifecycle from you, whereas others require you to manage them manually.

By default, JavaServer Faces automatically handles most of the lifecycle actions for you. However, it also exposes the various stages of the request lifecycle, so that you can modify or perform different actions if your application requirements warrant it.

It is not necessary for the beginning user to understand the lifecycle of a JavaServer Faces application, but the information can be useful for creating more complex applications.

The lifecycle of a JavaServer Faces application starts and ends with the following activity: The client makes a request for the web page, and the server responds with the page. The lifecycle consists of two main phases: `execute` and `render`.

During the execute phase, several actions can take place:

- The application view is built or restored.
- The request parameter values are applied.
- Conversions and validations are performed for component values.
- Managed beans are updated with component values.
- Application logic is invoked.
For a first (initial) request, only the view is built. For subsequent (postback) requests, some or all of the other actions can take place.

In the render phase, the requested view is rendered as a response to the client. Rendering is typically the process of generating output, such as HTML or XHTML, that can be read by the client, usually a browser.

The following short description of the example JavaServer Faces application passing through its lifecycle summarizes the activity that takes place behind the scenes.

The hello example application goes through the following stages when it is deployed on the GlassFish Server.

1. When the hello application is built and deployed on the GlassFish Server, the application is in an uninitiated state.
2. When a client makes an initial request for the bean hello.xhtml webpage, the hello Facelets application is compiled.
3. The compiled Facelets application is executed, and a new component tree is constructed for the hello application and is placed in a javax.faces.context.FacesContext.
4. The component tree is populated with the component and the managed bean property associated with it, represented by the EL expression hello.world.
5. A new view is built, based on the component tree.
6. The view is rendered to the requesting client as a response.
7. The component tree is destroyed automatically.
8. On subsequent (postback) requests, the component tree is rebuilt, and the saved state is applied.

For more detailed information on the JavaServer Faces lifecycle, see Chapter 10, “JavaServer Faces Technology: Advanced Concepts.”

Running the hello Application

You can use either NetBeans IDE or Ant to build, package, deploy, and run the hello example.

To Run the hello Application in NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog box, navigate to:
   tut-install/examples/web
3. Select the hello folder.
4 Select the Open as Main Project check box.

5 Click Open Project.

6 In the Projects tab, right-click the hello project and select Run.
   This step compiles, assembles, and deploys the application and then brings up a web browser
   window displaying the following URL:
   http://localhost:8080/hello
   The output looks like this:

   Hello World!

▼ To Run the hello Example Using Ant

1 In a terminal window, go to:
   tut-install/examples/web/hello/

2 Type the following command:
   ant
   This target builds the WAR file and copies it to the tut-install/examples/web/hello/dist/
   directory.

3 Type ant deploy.

4 In a web browser, type the following URL:
   http://localhost:8080/hello/
   The output looks like this:

   Hello World!

Further Information about JavaServer Faces Technology

For more information on JavaServer Faces technology, see

- JavaServer Faces 2.0 specification:

- JavaServer Faces project web site:
  http://javaserverfaces.java.net/

- Mojarra (JavaServer Faces 2.0 implementation) Release Notes:
  http://javaserverfaces.java.net/nonav/rlnotes/2.1.4/
Introduction to Facelets

The term *Facelets* refers to the view declaration language for JavaServer Faces technology. JavaServer Pages (JSP) technology, previously used as the presentation technology for JavaServer Faces, does not support all the new features available in JavaServer Faces in the Java EE 6 platform. JSP technology is considered to be a deprecated presentation technology for JavaServer Faces. Facelets is a part of the JavaServer Faces specification and also the preferred presentation technology for building JavaServer Faces technology-based applications.

The following topics are addressed here:

- “What Is Facelets?” on page 111
- “Developing a Simple Facelets Application” on page 113
- “Using Facelets Templates” on page 119
- “Composite Components” on page 121
- “Web Resources” on page 123

**What Is Facelets?**

Facelets is a powerful but lightweight page declaration language that is used to build JavaServer Faces views using HTML style templates and to build component trees. Facelets features include the following:

- Use of XHTML for creating web pages
- Support for Facelets tag libraries in addition to JavaServer Faces and JSTL tag libraries
- Support for the Expression Language (EL)
- Templating for components and pages

Advantages of Facelets for large-scale development projects include the following:

- Support for code reuse through templating and composite components
- Functional extensibility of components and other server-side objects through customization
- Faster compilation time
- Compile-time EL validation
- High-performance rendering

In short, the use of Facelets reduces the time and effort that needs to be spent on development and deployment.

Facelets views are usually created as XHTML pages. JavaServer Faces implementations support XHTML pages created in conformance with the XHTML Transitional Document Type Definition (DTD), as listed at [http://www.w3.org/TR/xhtml1/#a_dtd_XHTML-1.0-Transitional](http://www.w3.org/TR/xhtml1/#a_dtd_XHTML-1.0-Transitional). By convention, web pages built with XHTML have an .xhtml extension.

JavaServer Faces technology supports various tag libraries to add components to a web page. To support the JavaServer Faces tag library mechanism, Facelets uses XML namespace declarations. Table 5–1 lists the tag libraries supported by Facelets.

<table>
<thead>
<tr>
<th>Tag Library</th>
<th>URI</th>
<th>Prefix</th>
<th>Example</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaServer Faces Core Tag Library</td>
<td><a href="http://java.sun.com/jsf/core">http://java.sun.com/jsf/core</a></td>
<td>f:</td>
<td>f:actionListener, f:attribute</td>
<td>Tags for JavaServer Faces custom actions that are independent of any particular render kit</td>
</tr>
<tr>
<td>JSTL Core Tag Library</td>
<td><a href="http://java.sun.com/jsp/jstl/core">http://java.sun.com/jsp/jstl/core</a></td>
<td>c:</td>
<td>c:forEach, c:catch</td>
<td>JSTL 1.2 Core Tags</td>
</tr>
<tr>
<td>JSTL Functions Tag Library</td>
<td><a href="http://java.sun.com/jsp/jstl/functions">http://java.sun.com/jsp/jstl/functions</a></td>
<td>fn:</td>
<td>fn:toUpperCase, fn:toLowerCase</td>
<td>JSTL 1.2 Functions Tags</td>
</tr>
</tbody>
</table>
In addition, Facelets supports tags for composite components, for which you can declare custom prefixes. For more information on composite components, see “Composite Components” on page 121.

Based on the JavaServer Faces support for Expression Language (EL) syntax, Facelets uses EL expressions to reference properties and methods of managed beans. EL expressions can be used to bind component objects or values to methods or properties of managed beans. For more information on using EL expressions, see “Using the EL to Reference Managed Beans” on page 191.

### Developing a Simple Facelets Application

This section describes the general steps involved in developing a JavaServer Faces application. The following tasks are usually required:

- Developing the managed beans
- Creating the pages using the component tags
- Defining page navigation
- Mapping the `javax.faces.webapp.FacesServlet` instance
- Adding managed bean declarations

### Creating a Facelets Application

The example used in this tutorial is the `guessnumber` application. The application presents you with a page that asks you to guess a number between 0 and 10, validates your input against a random number, and responds with another page that informs you whether you guessed the number correctly or incorrectly.

### Developing a Managed Bean

In a typical JavaServer Faces application, each page of the application connects to a managed bean. The managed bean defines the methods and properties that are associated with the components. In this example, both pages use the same managed bean.

The following managed bean class, `UserNumberBean.java`, generates a random number from 0 to 10:

```java
package guessNumber;
import java.io.Serializable;
import java.util.Random;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean
```
public class UserNumberBean implements Serializable {
    private static final long serialVersionUID = 5443351151396868724L;
    Integer randomInt = null;
    Integer userNumber = null;
    String response = null;
    private long maximum=10;
    private long minimum=0;

    public UserNumberBean() {
        Random randomGR = new Random();
        randomInt = new Integer(randomGR.nextInt(10));
        System.out.println("Duke’s number: " + randomInt);
    }

    public void setUserNumber(Integer user_number) {
        userNumber = user_number;
    }

    public Integer getUserNumber() {
        return userNumber;
    }

    public String getResponse() {
        if ((userNumber != null) && (userNumber.compareTo(randomInt) == 0)) {
            return "Yay! You got it!";
        } else {
            return "Sorry, " + userNumber + " is incorrect.";
        }
    }

    public long getMaximum() {
        return (this.maximum);
    }

    public void setMaximum(long maximum) {
        this.maximum = maximum;
    }

    public long getMinimum() {
        return (this.minimum);
    }

    public void setMinimum(long minimum) {
        this.minimum = minimum;
    }
}

Note the use of the @ManagedBean annotation, which registers the managed bean as a resource with the JavaServer Faces implementation. The @SessionScoped annotation registers the bean scope as session.

Creating Facelets Views
To create a page or view, you add components to the pages, wire the components to managed bean values and properties, and register converters, validators, or listeners on the components.
For the example application, XHTML web pages serve as the front end. The first page of the example application is a page called `greeting.xhtml`. A closer look at various sections of this web page provides more information.

The first section of the web page declares the content type for the page, which is XHTML:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

The next section specifies the language of the XHTML page, then declares the XML namespace for the tag libraries that are used in the web page:

```html
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core">
```

The next section uses various tags to insert components into the web page:

```html
<h:head>
  <h:outputStylesheet library="css" name="default.css"/>
  <title>Guess Number Facelets Application</title>
</h:head>
<h:body>
  <h:form>
    <h:graphicImage library="images" name="wave.med.gif"
      alt="Duke waving his hand"/>
    <h2>
      Hi, my name is Duke. I am thinking of a number from
      #{userNumberBean.minimum} to #{userNumberBean.maximum}.
      Can you guess it?
    </h2>
    <p>
      <h:inputText
        id="userNo"
        title="Type a number from 0 to 10:"
        value="#{userNumberBean.userNumber}"
        f:validateLongRange
        minimum="#{userNumberBean.minimum}"
        maximum="#{userNumberBean.maximum}"/>
    </p>
    <h:commandButton id="submit" value="Submit"
      action="response"/>
    <h:message showSummary="true" showDetail="false"
      style="color: #d20005;
      font-family: 'New Century Schoolbook', serif;
      font-style: oblique;
      text-decoration: overline"
      id="errors1"
      for="userNo"/>
  </h:form>
</h:body>
```
Note the use of the following tags:

- Facelets HTML tags (those beginning with h:) to add components
- The Facelets core tag f:validateLongRange to validate the user input

An h:inputText tag accepts user input and sets the value of the managed bean property `userNumber` through the EL expression `#{userNumberBean.userNumber}`. The input value is validated for value range by the JavaServer Faces standard validator tag f:validateLongRange.

The image file, `wave.med.gif`, is added to the page as a resource; so is the style sheet. For more details about the resources facility, see "Web Resources" on page 123.

An h:commandButton tag with the ID submit starts validation of the input data when a user clicks the button. Using implicit navigation, the tag redirects the client to another page, `response.xhtml`, which shows the response to your input. The page specifies only `response`, which by default causes the server to look for `response.xhtml`.

You can now create the second page, `response.xhtml`, with the following content:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  <h:head>
    <h:outputStylesheet library="css" name="default.css"/>
    <title>Guess Number Facelets Application</title>
  </h:head>
  <h:body>
    <h:form>
      <h:graphicImage library="images" name="wave.med.gif" alt="Duke waving his hand"/>
      <h2>
        <h:outputText id="result" value="#{userNumberBean.response}"/>
      </h2>
      <h:commandButton id="back" value="Back" action="greeting"/>
    </h:form>
  </h:body>
</html>
```

### Configuring the Application

Configuring a JavaServer Faces application involves mapping the Faces Servlet in the web deployment descriptor file, such as a `web.xml` file, and possibly adding managed bean declarations, navigation rules, and resource bundle declarations to the application configuration resource file, `faces-config.xml`. 

---

The Java EE 6 Tutorial • January 2013
If you are using NetBeans IDE, a web deployment descriptor file is automatically created for you. In such an IDE-created web.xml file, change the default greeting page, which is index.xhtml, to greeting.xhtml. Here is an example web.xml file, showing this change in bold.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>faces/greeting.xhtml</welcome-file>
  </welcome-file-list>
</web-app>
```

Note the use of the context parameter PROJECT_STAGE. This parameter identifies the status of a JavaServer Faces application in the software lifecycle.

The stage of an application can affect the behavior of the application. For example, if the project stage is defined as Development, debugging information is automatically generated for the user. If not defined by the user, the default project stage is Production.

**Running the guessnumber Facelets Example**

You can use either NetBeans IDE or Ant to build, package, deploy, and run the guessnumber example. The source code for this example is available in the tut-install/examples/web/guessnumber/ directory.
To Build, Package, and Deploy the guessnumber Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   tut-install/examples/web/
3. Select the guessnumber folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the guessnumber project and select Deploy.
   This option builds and deploys the example application to your GlassFish Server instance.

To Build, Package, and Deploy the guessnumber Example Using Ant

1. In a terminal window, go to:
   tut-install/examples/web/guessnumber/
2. Type the following command:
   ant
   This command calls the default target, which builds and packages the application into a WAR file, guessnumber.war, that is located in the dist directory.
3. Make sure that the GlassFish Server is started.
4. To deploy the application, type the following command:
   ant deploy

To Run the guessnumber Example

1. Open a web browser.
2. Type the following URL in your web browser:
   http://localhost:8080/guessnumber
   A web page opens.
3 **In the text field, type a number from 0 to 10 and click Submit.**

Another page appears, reporting whether your guess is correct or incorrect.

4 **If you guessed incorrectly, click the Back button to return to the main page.**

You can continue to guess until you get the correct answer.

### Using Facelets Templates

JavaServer Faces technology provides the tools to implement user interfaces that are easy to extend and reuse. Templating is a useful Facelets feature that allows you to create a page that will act as the base, or *template*, for the other pages in an application. By using templates, you can reuse code and avoid recreating similarly constructed pages. Templating also helps in maintaining a standard look and feel in an application with a large number of pages.

*Table 5–2* lists Facelets tags that are used for templating and their respective functionality.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui:component</td>
<td>Defines a component that is created and added to the component tree.</td>
</tr>
<tr>
<td>ui:composition</td>
<td>Defines a page composition that optionally uses a template. Content outside of this tag is ignored.</td>
</tr>
<tr>
<td>ui:debug</td>
<td>Defines a debug component that is created and added to the component tree.</td>
</tr>
<tr>
<td>ui:decorate</td>
<td>Similar to the composition tag but does not disregard content outside this tag.</td>
</tr>
<tr>
<td>ui:define</td>
<td>Defines content that is inserted into a page by a template.</td>
</tr>
<tr>
<td>ui:fragment</td>
<td>Similar to the component tag but does not disregard content outside this tag.</td>
</tr>
<tr>
<td>ui:include</td>
<td>Encapsulate and reuse content for multiple pages.</td>
</tr>
<tr>
<td>ui:insert</td>
<td>Inserts content into a template.</td>
</tr>
<tr>
<td>ui:param</td>
<td>Used to pass parameters to an included file.</td>
</tr>
<tr>
<td>ui:repeat</td>
<td>Used as an alternative for loop tags, such as c:forEach or h: dataTable.</td>
</tr>
<tr>
<td>ui:remove</td>
<td>Removes content from a page.</td>
</tr>
</tbody>
</table>

For more information on Facelets templating tags, see the documentation at [http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/](http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/).

The Facelets tag library includes the main templating tag `ui:insert`. A template page that is created with this tag allows you to define a default structure for a page. A template page is used as a template for other pages, usually referred to as client pages.
Using Facelets Templates

Here is an example of a template saved as template.xhtml:
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html">
<h:head>
<meta http-equiv="Content-Type"
content="text/html; charset=UTF-8" />
<h:outputStylesheet library="css" name="default.css"/>
<h:outputStylesheet library="css" name="cssLayout.css"/>
<title>Facelets Template</title>
</h:head>
<h:body>
<div id="top" class="top">
<ui:insert name="top">Top Section</ui:insert>
</div>
<div>
<div id="left">
<ui:insert name="left">Left Section</ui:insert>
</div>
<div id="content" class="left_content">
<ui:insert name="content">Main Content</ui:insert>
</div>
</div>
</h:body>
</html>

The example page defines an XHTML page that is divided into three sections: a top section, a
left section, and a main section. The sections have style sheets associated with them. The same
structure can be reused for the other pages of the application.
The client page invokes the template by using the ui:composition tag. In the following
example, a client page named templateclient.xhtml invokes the template page named
template.xhtml from the preceding example. A client page allows content to be inserted with
the help of the ui:define tag.
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html">
<h:body>
<ui:composition template="./template.xhtml">
<ui:define name="top">
Welcome to Template Client Page
</ui:define>
<ui:define name="left">
<h:outputLabel value="You are in the Left Section"/>
</ui:define>
120

The Java EE 6 Tutorial • January 2013


Composite Components

JavaServer Faces technology offers the concept of composite components with Facelets. A composite component is a special type of template that acts as a component.

Any component is essentially a piece of reusable code that behaves in a particular way. For example, an input component accepts user input. A component can also have validators, converters, and listeners attached to it to perform certain defined actions.

A composite component consists of a collection of markup tags and other existing components. This reusable, user-created component has a customized, defined functionality and can have validators, converters, and listeners attached to it like any other component.

With Facelets, any XHTML page that contains markup tags and other components can be converted into a composite component. Using the resources facility, the composite component can be stored in a library that is available to the application from the defined resources location.

Table 5–3 lists the most commonly used composite tags and their functions.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>composite:interface</td>
<td>Declares the usage contract for a composite component. The composite component can be used as a single component whose feature set is the union of the features declared in the usage contract.</td>
</tr>
<tr>
<td>composite:implementation</td>
<td>Defines the implementation of the composite component. If a composite:interface element appears, there must be a corresponding composite:implementation.</td>
</tr>
<tr>
<td>composite:attribute</td>
<td>Declares an attribute that may be given to an instance of the composite component in which this tag is declared.</td>
</tr>
<tr>
<td>composite:insertChildren</td>
<td>Any child components or template text within the composite component tag in the using page will be reparented into the composite component at the point indicated by this tag's placement within the composite:implementation section.</td>
</tr>
</tbody>
</table>
TABLE 5–3 Composite Component Tags (Continued)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>composite:valueHolder</td>
<td>Declares that the composite component whose contract is declared by the composite:interface in which this element is nested exposes an implementation of javax.faces.component.ValueHolder suitable for use as the target of attached objects in the using page.</td>
</tr>
<tr>
<td>composite:editableValueHolder</td>
<td>Declares that the composite component whose contract is declared by the composite:interface in which this element is nested exposes an implementation of javax.faces.component.EditableValueHolder suitable for use as the target of attached objects in the using page.</td>
</tr>
<tr>
<td>composite:actionSource</td>
<td>Declares that the composite component whose contract is declared by the composite:interface in which this element is nested exposes an implementation of javax.faces.component.ActionSource2 suitable for use as the target of attached objects in the using page.</td>
</tr>
</tbody>
</table>

For more information and a complete list of Facelets composite tags, see the documentation at http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/.

The following example shows a composite component that accepts an email address as input:

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:composite="http://java.sun.com/jsf/composite"
     xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
      <title>This content will not be displayed</title>
    </h:head>
    <h:body>
      <composite:interface>
        <composite:attribute name="value" required="false"/>
      </composite:interface>
      <composite:implementation>
        <h:outputLabel value="Email id: ">
        <h:inputText value="#{cc.attrs.value}"/>
      </composite:implementation>
      </h:body>
    </html>
```

Note the use of `cc.attrs.value` when defining the value of the `inputText` component. The word `cc` in JavaServer Faces is a reserved word for composite components. The `#{cc.attrs.attribute-name}` expression is used to access the attributes defined for the composite component’s interface, which in this case happens to be `value`.

The preceding example content is stored as a file named `email.xhtml` in a folder named `resources/emcomp`, under the application web root directory. This directory is considered a
library by JavaServer Faces, and a component can be accessed from such a library. For more information on resources, see "Web Resources" on page 123.

The web page that uses this composite component is generally called a using page. The using page includes a reference to the composite component, in the xml namespace declarations:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  <h:head>
    <title>Using a sample composite component</title>
  </h:head>
  <body>
    <h:form>
      <em:email value="Enter your email id" />
    </h:form>
  </body>
</html>
```

The local composite component library is defined in the xmlns namespace with the declaration xmlns:em="http://java.sun.com/jsf/composite/emcomp/". The component itself is accessed through the em:email tag. The preceding example content can be stored as a web page named emuserpage.xhtml under the web root directory. When compiled and deployed on a server, it can be accessed with the following URL:

http://localhost:8080/application-name/faces/emuserpage.xhtml

**Web Resources**

Web resources are any software artifacts that the web application requires for proper rendering, including images, script files, and any user-created component libraries. Resources must be collected in a standard location, which can be one of the following:

- A resource packaged in the web application root must be in a subdirectory of a resources directory at the web application root: resources/resource-identifier.

- A resource packaged in the web application’s classpath must be in a subdirectory of the META-INF/resources directory within a web application: META-INF/resources/resource-identifier. You can use this file structure to package resources in a JAR file bundled in the web application. See Chapter 53, “Duke’s Forest Case Study Example,” for an application that uses this mechanism.

The JavaServer Faces runtime will look for the resources in the preceding listed locations, in that order.

Resource identifiers are unique strings that conform to the following format:
Elements of the resource identifier in brackets ([ ]) are optional, indicating that only a resource-name, which is usually a file name, is a required element. For example, the most common way to specify a style sheet, image, or script is to use the library and name attributes, as in the following tag from the guessnumber example:

```xml
<h:outputStylesheet library="css" name="default.css"/>
```

This tag specifies that the default.css style sheet is in the directory web/resources/css.

You can also specify the location of an image using the following syntax, also from the guessnumber example:

```xml
<h:graphicImage value="#{resource['images:wave.med.gif']}"/>
```

This tag specifies that the image named wave.med.gif is in the directory web/resources/images.

Resources can be considered as a library location. Any artifact, such as a composite component or a template that is stored in the resources directory, becomes accessible to the other application components, which can use it to create a resource instance.
This chapter introduces the Expression Language (also referred to as the EL), which provides an important mechanism for enabling the presentation layer (web pages) to communicate with the application logic (managed beans). The EL is used by both JavaServer Faces technology and JavaServer Pages (JSP) technology. The EL represents a union of the expression languages offered by JavaServer Faces technology and JSP technology.

The following topics are addressed here:

- "Overview of the EL" on page 125
- "Immediate and Deferred Evaluation Syntax" on page 126
- "Value and Method Expressions" on page 128
- "Defining a Tag Attribute Type" on page 134
- "Literal Expressions" on page 135
- "Operators" on page 136
- "Reserved Words" on page 136
- "Examples of EL Expressions" on page 137

**Overview of the EL**

The EL allows page authors to use simple expressions to dynamically access data from JavaBeans components. For example, the `test` attribute of the following conditional tag is supplied with an EL expression that compares 0 with the number of items in the session-scoped bean named `cart`.

```xml
<c:if test="${sessionScope.cart.numberOfItems > 0}">
  ...
</c:if>
```

JavaServer Faces technology uses the EL for the following functions:

- Deferred and immediate evaluation of expressions
- The ability to set as well as get data
The ability to invoke methods

See “Using the EL to Reference Managed Beans” on page 191 for more information on how to use the EL in JavaServer Faces applications.

To summarize, the EL provides a way to use simple expressions to perform the following tasks:

- Dynamically read application data stored in JavaBeans components, various data structures, and implicit objects
- Dynamically write data, such as user input into forms, to JavaBeans components
- Invoke arbitrary static and public methods
- Dynamically perform arithmetic operations

The EL is also used to specify the following kinds of expressions that a custom tag attribute will accept:

- **Immediate evaluation expressions** or **deferred evaluation expressions**. An immediate evaluation expression is evaluated at once by the underlying technology, such as JavaServer Faces. A deferred evaluation expression can be evaluated later by the underlying technology using the EL.
- **Value expression** or **method expression**. A value expression references data, whereas a method expression invokes a method.
- **Rvalue expression** or **lvalue expression**. An rvalue expression can only read a value, whereas an lvalue expression can both read and write that value to an external object.

Finally, the EL provides a pluggable API for resolving expressions so custom resolvers that can handle expressions not already supported by the EL can be implemented.

### Immediate and Deferred Evaluation Syntax

The EL supports both immediate and deferred evaluation of expressions. Immediate evaluation means that the expression is evaluated and the result returned as soon as the page is first rendered. Deferred evaluation means that the technology using the expression language can use its own machinery to evaluate the expression sometime later during the page’s lifecycle, whenever it is appropriate to do so.

Those expressions that are evaluated immediately use the ${} syntax. Expressions whose evaluation is deferred use the #{} syntax.

Because of its multiphase lifecycle, JavaServer Faces technology uses mostly deferred evaluation expressions. During the lifecycle, component events are handled, data is validated, and other tasks are performed in a particular order. Therefore, a JavaServer Faces implementation must defer evaluation of expressions until the appropriate point in the lifecycle.

Other technologies using the EL might have different reasons for using deferred expressions.
Immediate Evaluation

All expressions using the ${} syntax are evaluated immediately. These expressions can be used only within template text or as the value of a tag attribute that can accept runtime expressions.

The following example shows a tag whose value attribute references an immediate evaluation expression that gets the total price from the session-scoped bean named cart:

```xml
<fmt:formatNumber value="${sessionScope.cart.total}"/>
```

The JavaServer Faces implementation evaluates the expression ${sessionScope.cart.total}, converts it, and passes the returned value to the tag handler.

Immediate evaluation expressions are always read-only value expressions. The preceding example expression cannot set the total price, but instead can only get the total price from the cart bean.

Deferred Evaluation

Deferred evaluation expressions take the form #{expr} and can be evaluated at other phases of a page lifecycle as defined by whatever technology is using the expression. In the case of JavaServer Faces technology, its controller can evaluate the expression at different phases of the lifecycle, depending on how the expression is being used in the page.

The following example shows a JavaServer Faces h:inputText tag, which represents a text field component into which a user enters a value. The h:inputText tag's value attribute references a deferred evaluation expression that points to the name property of the customer bean:

```xml
<h:inputText id="name" value="#{customer.name}" />
```

For an initial request of the page containing this tag, the JavaServer Faces implementation evaluates the #{customer.name} expression during the render-response phase of the lifecycle. During this phase, the expression merely accesses the value of name from the customer bean, as is done in immediate evaluation.

For a postback request, the JavaServer Faces implementation evaluates the expression at different phases of the lifecycle, during which the value is retrieved from the request, validated, and propagated to the customer bean.

As shown in this example, deferred evaluation expressions can be

- Value expressions that can be used to both read and write data
- Method expressions

Value expressions (both immediate and deferred) and method expressions are explained in the next section.
Value and Method Expressions

The EL defines two kinds of expressions: value expressions and method expressions. Value expressions can either yield a value or set a value. Method expressions reference methods that can be invoked and can return a value.

Value Expressions

Value expressions can be further categorized into rvalue and lvalue expressions. Rvalue expressions can read data but cannot write it. Lvalue expressions can both read and write data.

All expressions that are evaluated immediately use the ${} delimiters and are always rvalue expressions. Expressions whose evaluation can be deferred use the #{} delimiters and can act as both rvalue and lvalue expressions. Consider the following two value expressions:

${customer.name}
#{customer.name}

The former uses immediate evaluation syntax, whereas the latter uses deferred evaluation syntax. The first expression accesses the name property, gets its value, adds the value to the response, and gets rendered on the page. The same can happen with the second expression. However, the tag handler can defer the evaluation of this expression to a later time in the page lifecycle, if the technology using this tag allows.

In the case of JavaServer Faces technology, the latter tag’s expression is evaluated immediately during an initial request for the page. In this case, this expression acts as an rvalue expression. During a postback request, this expression can be used to set the value of the name property with user input. In this case, the expression acts as an lvalue expression.

Referencing Objects Using Value Expressions

Both rvalue and lvalue expressions can refer to the following objects and their properties or attributes:

- JavaBeans components
- Collections
- Java SE enumerated types
- Implicit objects

To refer to these objects, you write an expression using a variable that is the name of the object. The following expression references a managed bean called customer:

${customer}
The web container evaluates the variable that appears in an expression by looking up its value according to the behavior of `PageContext.findAttribute(String)`, where the String argument is the name of the variable. For example, when evaluating the expression `${customer}`, the container will look for `customer` in the page, request, session, and application scopes and will return its value. If `customer` is not found, a null value is returned.

You can use a custom EL resolver to alter the way variables are resolved. For instance, you can provide an EL resolver that intercepts objects with the name `customer`, so that `${customer}` returns a value in the EL resolver instead.

To reference an enum constant with an expression, use a `String` literal. For example, consider this Enum class:

```java
public enum Suit {hearts, spades, diamonds, clubs}
```

To refer to the `Suit.hearts` constant with an expression, use the `String` literal "hearts". Depending on the context, the `String` literal is converted to the enum constant automatically. For example, in the following expression in which `mySuit` is an instance of `Suit`, "hearts" is first converted to `Suit.hearts` before it is compared to the instance:

`${mySuit == "hearts"}`

### Referring to Object Properties Using Value Expressions

To refer to properties of a bean or an enum instance, items of a collection, or attributes of an implicit object, you use the `.or []` notation.

To reference the name property of the `customer` bean, use either the expression `${customer.name}` or the expression `${customer["name"]}`. The part inside the brackets is a `String` literal that is the name of the property to reference.

You can use double or single quotes for the `String` literal. You can also combine the `[]` and `.notations, as shown here:

`${customer.address["street"]}`

Properties of an enum constant can also be referenced in this way. However, as with JavaBeans component properties, the properties of an Enum class must follow JavaBeans component conventions. This means that a property must at least have an accessor method called `getProperty`, where `Property` is the name of the property that can be referenced by an expression.

For example, consider an Enum class that encapsulates the names of the planets of our galaxy and includes a method to get the mass of a planet. You can use the following expression to reference the method `getMass` of the Enum class `Planet`:

`${myPlanet.mass}`
If you are accessing an item in an array or list, you must use either a literal value that can be converted to int or the [] notation with an int and without quotes. The following examples could resolve to the same item in a list or array, assuming that socks can be converted to int:

- ${customer.orders[1]}
- ${customer.orders.socks}

In contrast, an item in a Map can be accessed using a string literal key; no coercion is required:

- ${customer.orders["socks"]}

An rvalue expression also refers directly to values that are not objects, such as the result of arithmetic operations and literal values, as shown by these examples:

- ${"literal"}
- ${customer.age + 20}
- ${true}
- ${57}

The EL defines the following literals:

- Boolean: true and false
- Integer: as in Java
- Floating-point: as in Java
- String: with single and double quotes; " is escaped as ", ' is escaped as \', and \ is escaped as \"
- Null: null

You can also write expressions that perform operations on an enum constant. For example, consider the following Enum class:

```java
public enum Suit {club, diamond, heart, spade}
```

After declaring an enum constant called mySuit, you can write the following expression to test whether mySuit is spade:

- $(mySuit == "spade")

When it resolves this expression, the EL resolving mechanism will invoke the valueOf method of the Enum class with the Suit class and the spade type, as shown here:

- mySuit.valueOf(Suit.class, "spade")
Where Value Expressions Can Be Used

Value expressions using the ${} delimiters can be used in

- Static text
- Any standard or custom tag attribute that can accept an expression

The value of an expression in static text is computed and inserted into the current output. Here is an example of an expression embedded in static text:

```xml
<some:tag>
  some text ${expr} some text
</some:tag>
```

If the static text appears in a tag body, note that an expression will not be evaluated if the body is declared to be tagdependent.

Lvalue expressions can be used only in tag attributes that can accept lvalue expressions.

A tag attribute value using either an rvalue or lvalue expression can be set in the following ways:

- With a single expression construct:
  ```xml
  <some:tag value="${expr}"/>
  <another:tag value="#{expr}"/>
  ```
  These expressions are evaluated, and the result is converted to the attribute’s expected type.

- With one or more expressions separated or surrounded by text:
  ```xml
  <some:tag value="some${expr}${expr}text${expr}"/>
  <another:tag value="some#{expr}#{expr}text#{expr}"/>
  ```
  These kinds of expression, called composite expressions, are evaluated from left to right. Each expression embedded in the composite expression is converted to a String and then concatenated with any intervening text. The resulting String is then converted to the attribute's expected type.

- With text only:
  ```xml
  <some:tag value="sometext"/>
  ```
  This expression is called a literal expression. In this case, the attribute’s String value is converted to the attribute’s expected type. Literal value expressions have special syntax rules. See “Literal Expressions” on page 135 for more information. When a tag attribute has an enum type, the expression that the attribute uses must be a literal expression. For example, the tag attribute can use the expression "hearts" to mean Suit.hearts. The literal is converted to Suit, and the attribute gets the value Suit.hearts.

All expressions used to set attribute values are evaluated in the context of an expected type. If the result of the expression evaluation does not match the expected type exactly, a type conversion will be performed. For example, the expression ${1.2E4} provided as the value of an attribute of type float will result in the following conversion:
Method Expressions

Another feature of the EL is its support of deferred method expressions. A method expression is used to invoke an arbitrary public method of a bean, which can return a result.

In JavaServer Faces technology, a component tag represents a component on a page. The component tag uses method expressions to invoke methods that perform some processing for the component. These methods are necessary for handling events that the components generate and for validating component data, as shown in this example:

```xml
<h:form>
  <h:inputText id="name" value="#{customer.name}"
               validator="#{customer.validateName}"/>
  <h:commandButton id="submit" action="#{customer.submit}" />
</h:form>
```

The `h:inputText` tag displays as a text field. The `validator` attribute of this `h:inputText` tag references a method, called `validateName`, in the bean, called `customer`.

Because a method can be invoked during different phases of the lifecycle, method expressions must always use the deferred evaluation syntax.

Like `lvalue` expressions, method expressions can use the . and the [] operators. For example, `#{object.method}` is equivalent to `#{object["method"]}`. The literal inside the [] is converted to `String` and is used to find the name of the method that matches it. Once the method is found, it is invoked, or information about the method is returned.

Method expressions can be used only in tag attributes and only in the following ways:

- With a single expression construct, where `bean` refers to a JavaBeans component and `method` refers to a method of the JavaBeans component:
  ```xml
  <some:tag value="#{bean.method}"/>
  ```
  The expression is evaluated to a method expression, which is passed to the tag handler. The method represented by the method expression can then be invoked later.

- With text only:
  ```xml
  <some:tag value="sometext"/>
  ```
Method expressions support literals primarily to support action attributes in JavaServer Faces technology. When the method referenced by this method expression is invoked, the method returns the String literal, which is then converted to the expected return type, as defined in the tag’s tag library descriptor.

**Parameterized Method Calls**

The EL offers support for parameterized method calls. Method calls can use parameters without having to use static EL functions.

Both the . and [] operators can be used for invoking method calls with parameters, as shown in the following expression syntax:

- `expr-a[expr-b](parameters)`
- `expr-a.identifier-b(parameters)`

In the first expression syntax, `expr-a` is evaluated to represent a bean object. The expression `expr-b` is evaluated and cast to a string that represents a method in the bean represented by `expr-a`. In the second expression syntax, `expr-a` is evaluated to represent a bean object, and `identifier-b` is a string that represents a method in the bean object. The `parameters` in parentheses are the arguments for the method invocation. Parameters can be zero or more values or expressions, separated by commas.

Parameters are supported for both value expressions and method expressions. In the following example, which is a modified tag from the guessnumber application, a random number is provided as an argument rather than from user input to the method call:

```xml
<h:inputText value="#{userNumberBean.userNumber('5')}">
```

The preceding example uses a value expression.

Consider the following example of a JavaServer Faces component tag that uses a method expression:

```xml
<h:commandButton action="#{trader.buy}" value="buy"/>
```

The EL expression `trader.buy` calls the `trader` bean’s `buy` method. You can modify the tag to pass on a parameter. Here is the revised tag where a parameter is passed:

```xml
<h:commandButton action="#{trader.buy('SOMESTOCK')}" value="buy"/>
```

In the preceding example, you are passing the string ‘SOMESTOCK’ (a stock symbol) as a parameter to the `buy` method.

For more information on the updated EL, see [http://uel.java.net/](http://uel.java.net/).
Defining a Tag Attribute Type

As explained in the previous section, all kinds of expressions can be used in tag attributes. Which kind of expression and how it is evaluated, whether immediately or deferred, are determined by the type attribute of the tag’s definition in the View Description Language (VDL) that defines the tag.

If you plan to create custom tags, for each tag in the VDL, you need to specify what kind of expression to accept. Table 6–1 shows the kinds of tag attributes that accept EL expressions, gives examples of expressions they accept, and provides the type definitions of the attributes that must be added to the VDL. You cannot use #{} syntax for a dynamic attribute, meaning an attribute that accepts dynamically calculated values at runtime. Similarly, you also cannot use the ${} syntax for a deferred attribute.

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Example Expression</th>
<th>Type Attribute Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>&quot;literal&quot;</td>
<td>&lt;rtexprvalue&gt;true&lt;/rtexprvalue&gt;</td>
</tr>
<tr>
<td>Dynamic</td>
<td>${literal}</td>
<td>&lt;rtexprvalue&gt;true&lt;/rtexprvalue&gt;</td>
</tr>
<tr>
<td>Deferred value</td>
<td>&quot;literal&quot;</td>
<td>&lt;deferred-value&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;type&gt;java.lang.String&lt;/type&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/deferred-value&gt;</td>
</tr>
<tr>
<td>Deferred value</td>
<td>#{customer.age}</td>
<td>&lt;deferred-value&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;type&gt;int&lt;/type&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/deferred-value&gt;</td>
</tr>
<tr>
<td>Deferred method</td>
<td>&quot;literal&quot;</td>
<td>&lt;deferred-method&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;method-signature&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>java.lang.String submit()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/method-signature&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/deferred-method&gt;</td>
</tr>
<tr>
<td>Deferred method</td>
<td>#{customer.calcTotal}</td>
<td>&lt;deferred-method&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;method-signature&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>double calcTotal(int, double)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/method-signature&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;/deferred-method&gt;</td>
</tr>
</tbody>
</table>

In addition to the tag attribute types shown in Table 6–1, you can define an attribute to accept both dynamic and deferred expressions. In this case, the tag attribute definition contains both an rtexprvalue definition set to true and either a deferred-value or deferred-method definition.
Literal Expressions

A literal expression is evaluated to the text of the expression, which is of type String. A literal expression does not use the ${} or #{} delimiters.

If you have a literal expression that includes the reserved ${} or #{} syntax, you need to escape these characters as follows:

- By creating a composite expression as shown here:
  
  ```
  ${'${'}exprA}
  
  #{'#{'}exprB}
  ```

  The resulting values would then be the strings ${exprA} and #{exprB}.

- By using the escape characters \$ and \# to escape what would otherwise be treated as an eval-expression:
  
  ```
  \${exprA}
  
  \#{exprB}
  ```

  The resulting values would again be the strings ${exprA} and #{exprB}.

When a literal expression is evaluated, it can be converted to another type. Table 6–2 shows examples of various literal expressions and their expected types and resulting values.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expected Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi</td>
<td>String</td>
<td>Hi</td>
</tr>
<tr>
<td>true</td>
<td>Boolean</td>
<td>Boolean.TRUE</td>
</tr>
<tr>
<td>42</td>
<td>int</td>
<td>42</td>
</tr>
</tbody>
</table>

Literal expressions can be evaluated immediately or deferred and can be either value or method expressions. At what point a literal expression is evaluated depends on where it is being used. If the tag attribute that uses the literal expression is defined to accept a deferred value expression, when referencing a value, the literal expression is evaluated at a point in the lifecycle that is determined by other factors, such as where the expression is being used and to what it is referring.

In the case of a method expression, the method that is referenced is invoked and returns the specified String literal. For example, the h:commandButton tag of the guess number application uses a literal method expression as a logical outcome to tell the JavaServer Faces navigation system which page to display next.
Operators

In addition to the . and [ ] operators discussed in “Value and Method Expressions” on page 128, the EL provides the following operators, which can be used in rvalue expressions only:

- **Arithmetic**: +, - (binary), *, / and div, % and mod, - (unary)
- **Logical**: and, &&, or, ||, not, !
- **Relational**: ==, eq, !=, ne, <, lt, , gt, <=, ge, >=, le. Comparisons can be made against other values or against Boolean, string, integer, or floating-point literals.
- **Empty**: The empty operator is a prefix operation that can be used to determine whether a value is null or empty.
- **Conditional**: A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.

The precedence of operators highest to lowest, left to right is as follows:

- [ ]
- () (used to change the precedence of operators)
- - (unary) not ! empty
- * / div % mod
- + - (binary)
- < > <= >= lt lt ge
- == != eq ne
- && and
- || or
- ? :

Reserved Words

The following words are reserved for the EL and should not be used as identifiers:

```plaintext
and       or       not       eq
ne        lt        gt        le
ge        true      false     null
instanceof empty     div       mod
```
### Examples of EL Expressions

Table 6–3 contains example EL expressions and the result of evaluating them.

<table>
<thead>
<tr>
<th>EL Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1 &gt; (4/2))$</td>
<td>false</td>
</tr>
<tr>
<td>$(4.0 &gt;= 3)$</td>
<td>true</td>
</tr>
<tr>
<td>$(100.0 == 100)$</td>
<td>true</td>
</tr>
<tr>
<td>$(10*10) ne 100$</td>
<td>false</td>
</tr>
<tr>
<td>${'a' &lt; 'b'}$</td>
<td>true</td>
</tr>
<tr>
<td>${'hip' gt 'hit'}$</td>
<td>false</td>
</tr>
<tr>
<td>$(4 &gt; 3)$</td>
<td>true</td>
</tr>
<tr>
<td>$(1.2E4 + 1.4)$</td>
<td>12001.4</td>
</tr>
<tr>
<td>$(3 div 4)$</td>
<td>0.75</td>
</tr>
<tr>
<td>$(10 mod 4)$</td>
<td>2</td>
</tr>
<tr>
<td>$(!empty param.Add)$</td>
<td>False if the request parameter named Add is null or an empty string.</td>
</tr>
<tr>
<td>${pageContext.request.contextPath}$</td>
<td>The context path.</td>
</tr>
<tr>
<td>${sessionScope.cart.numberOfItems}$</td>
<td>The value of the numberOfItems property of the session-scoped attribute named cart.</td>
</tr>
<tr>
<td>${param['mycom.productId']}$</td>
<td>The value of the request parameter named mycom.productId.</td>
</tr>
<tr>
<td>${header['host']}$</td>
<td>The host.</td>
</tr>
<tr>
<td>${departments[deptName]}$</td>
<td>The value of the entry named deptName in the departments map.</td>
</tr>
<tr>
<td>${requestScope['javax.servlet.forward.servlet_path']}$</td>
<td>The value of the request-scoped attribute named javax.servlet.forward.servlet_path.</td>
</tr>
<tr>
<td>#{customer.lName}</td>
<td>Gets the value of the property lName from the customer bean during an initial request. Sets the value of lName during a postback.</td>
</tr>
<tr>
<td>#{customer.calcTotal}</td>
<td>The return value of the method calcTotal of the customer bean.</td>
</tr>
</tbody>
</table>
Web pages represent the presentation layer for web applications. The process of creating web pages for a JavaServer Faces application includes adding components to the page and wiring them to managed beans, validators, listeners, converters, and other server-side objects that are associated with the page.

This chapter explains how to create web pages using various types of component and core tags. In the next chapter, you will learn about adding converters, validators, and listeners to component tags to provide additional functionality to components.

Many of the examples in this chapter are taken from Chapter 51, “Duke’s Bookstore Case Study Example.”

The following topics are addressed here:

- “Setting Up a Page” on page 139
- “Adding Components to a Page Using HTML Tags” on page 140
- “Using Core Tags” on page 171

**Setting Up a Page**

A typical JavaServer Faces web page includes the following elements:

- A set of namespace declarations that declare the JavaServer Faces tag libraries
- Optionally, the HTML head (h:head) and body (h:body) tags
- A form tag (h:form) that represents the user input components

To add the JavaServer Faces components to your web page, you need to provide the page access to the two standard tag libraries: the JavaServer Faces HTML render kit tag library and the JavaServer Faces core tag library. The JavaServer Faces standard HTML tag library defines tags that represent common HTML user interface components. This library is linked to the HTML
Adding Components to a Page Using HTML Tags

The JavaServer Faces core tag library defines tags that perform core actions and are independent of a particular render kit at http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/renderkitdocs/. For a complete list of JavaServer Faces Facelets tags and their attributes, refer to the documentation at http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/.

To use any of the JavaServer Faces tags, you need to include appropriate directives at the top of each page specifying the tag libraries.

For Facelets applications, the XML namespace directives uniquely identify the tag library URI and the tag prefix.

For example, when you create a Facelets XHTML page, include namespace directives as follows:

```xml
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://java.sun.com/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core">

The XML namespace URI identifies the tag library location, and the prefix value is used to distinguish the tags belonging to that specific tag library. You can also use other prefixes instead of the standard h or f. However, when including the tag in the page, you must use the prefix that you have chosen for the tag library. For example, in the following web page, the form tag must be referenced using the h prefix because the preceding tag library directive uses the h prefix to distinguish the tags defined in HTML tag library:

```xml
<h:form ...>

The sections “Adding Components to a Page Using HTML Tags” on page 140 and “Using Core Tags” on page 171 describe how to use the component tags from the JavaServer Faces standard HTML tag library and the core tags from the JavaServer Faces core tag library.

Adding Components to a Page Using HTML Tags

The tags defined by the JavaServer Faces standard HTML tag library represent HTML form components and other basic HTML elements. These components display data or accept data from the user. This data is collected as part of a form and is submitted to the server, usually when the user clicks a button. This section explains how to use each of the component tags shown in Table 7–1.
<table>
<thead>
<tr>
<th>Tag</th>
<th>Functions</th>
<th>Rendered as</th>
<th>Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>h:column</td>
<td>Represents a column of data in a data component</td>
<td>A column of data in an HTML table</td>
<td>A column in a table</td>
</tr>
<tr>
<td>h:commandButton</td>
<td>Submits a form to the application</td>
<td>An HTML <code>&lt;input type=&quot;type&quot;&gt;</code> element, where the type value can be &quot;submit&quot;, &quot;reset&quot;, or &quot;image&quot;</td>
<td>A button</td>
</tr>
<tr>
<td>h:commandLink</td>
<td>Links to another page or location on a page</td>
<td>An HTML <code>&lt;a href&gt;</code> element</td>
<td>A hyperlink</td>
</tr>
<tr>
<td>h:dataTable</td>
<td>Represents a data wrapper</td>
<td>An HTML <code>&lt;table&gt;</code> element</td>
<td>A table that can be updated dynamically</td>
</tr>
<tr>
<td>h:form</td>
<td>Represents an input form (inner tags of the form receive the data that will be submitted with the form)</td>
<td>An HTML <code>&lt;form&gt;</code> element</td>
<td>No appearance</td>
</tr>
<tr>
<td>h:graphicImage</td>
<td>Displays an image</td>
<td>An HTML <code>&lt;img&gt;</code> element</td>
<td>An image</td>
</tr>
<tr>
<td>h:inputHidden</td>
<td>Allows a page author to include a hidden variable in a page</td>
<td>An HTML <code>&lt;input type=&quot;hidden&quot;&gt;</code> element</td>
<td>No appearance</td>
</tr>
<tr>
<td>h:inputSecret</td>
<td>Allows a user to input a string without the actual string appearing in the field</td>
<td>An HTML <code>&lt;input type=&quot;password&quot;&gt;</code> element</td>
<td>A text field, which displays a row of characters instead of the actual string entered</td>
</tr>
<tr>
<td>h:inputText</td>
<td>Allows a user to input a string</td>
<td>An HTML <code>&lt;input type=&quot;text&quot;&gt;</code> element</td>
<td>A text field</td>
</tr>
<tr>
<td>h:inputTextarea</td>
<td>Allows a user to enter a multiline string</td>
<td>An HTML <code>&lt;textarea&gt;</code> element</td>
<td>A multi-row text field</td>
</tr>
<tr>
<td>h:message</td>
<td>Displays a localized message</td>
<td>An HTML <code>&lt;span&gt;</code> tag if styles are used</td>
<td>A text string</td>
</tr>
<tr>
<td>h:messages</td>
<td>Displays localized messages</td>
<td>A set of HTML <code>&lt;span&gt;</code> tags if styles are used</td>
<td>A text string</td>
</tr>
<tr>
<td>h:outputFormat</td>
<td>Displays a localized message</td>
<td>Plain text</td>
<td>Plain text</td>
</tr>
<tr>
<td>h:outputLabel</td>
<td>Displays a nested component as a label for a specified input field</td>
<td>An HTML <code>&lt;label&gt;</code> element</td>
<td>Plain text</td>
</tr>
<tr>
<td>Tag</td>
<td>Functions</td>
<td>Rendered as</td>
<td>Appearance</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>h:outputLink</td>
<td>Links to another page or location on a page without generating an action event</td>
<td>An HTML <code>&lt;a&gt;</code> element</td>
<td>A hyperlink</td>
</tr>
<tr>
<td>h:outputText</td>
<td>Displays a line of text</td>
<td>Plain text</td>
<td>Plain text</td>
</tr>
<tr>
<td>h:panelGrid</td>
<td>Displays a table</td>
<td>An HTML <code>&lt;table&gt;</code> element with <code>&lt;tr&gt;</code> and <code>&lt;td&gt;</code> elements</td>
<td>A table</td>
</tr>
<tr>
<td>h:panelGroup</td>
<td>Groups a set of components under one parent</td>
<td>A HTML <code>&lt;div&gt;</code> or <code>&lt;span&gt;</code> element</td>
<td>A row in a table</td>
</tr>
<tr>
<td>h:selectBooleanCheckbox</td>
<td>Allows a user to change the value of a Boolean choice</td>
<td>An HTML <code>&lt;input type=&quot;checkbox&quot;&gt;</code> element</td>
<td>A check box</td>
</tr>
<tr>
<td>h:selectManyCheckbox</td>
<td>Displays a set of check boxes from which the user can select multiple values</td>
<td>A set of HTML <code>&lt;input&gt;</code> elements of type checkbox</td>
<td>A set of check boxes</td>
</tr>
<tr>
<td>h:selectManyListbox</td>
<td>Allows a user to select multiple items from a set of items, all displayed at once</td>
<td>An HTML <code>&lt;select&gt;</code> element</td>
<td>A list box</td>
</tr>
<tr>
<td>h:selectManyMenu</td>
<td>Allows a user to select multiple items from a set of items</td>
<td>An HTML <code>&lt;select&gt;</code> element</td>
<td>A scrollable combo box</td>
</tr>
<tr>
<td>h:selectOneListbox</td>
<td>Allows a user to select one item from a set of items, all displayed at once</td>
<td>An HTML <code>&lt;select&gt;</code> element</td>
<td>A list box</td>
</tr>
<tr>
<td>h:selectOneMenu</td>
<td>Allows a user to select one item from a set of items</td>
<td>An HTML <code>&lt;select&gt;</code> element</td>
<td>A scrollable combo box</td>
</tr>
<tr>
<td>h:selectOneRadio</td>
<td>Allows a user to select one item from a set of items</td>
<td>An HTML <code>&lt;input type=&quot;radio&quot;&gt;</code> element</td>
<td>A set of radio buttons</td>
</tr>
</tbody>
</table>

The next section explains the important tag attributes that are common to most component tags. For each of the components discussed in the following sections, “Writing Bean Properties” on page 192 explains how to write a bean property bound to a particular component or its value.

**Common Component Tag Attributes**

Most of the component tags support the attributes shown in Table 7–2.
TABLE 7–2  Common Component Tag Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binding</td>
<td>Identifies a bean property and binds the component instance to it.</td>
</tr>
<tr>
<td>id</td>
<td>Uniquely identifies the component.</td>
</tr>
<tr>
<td>immediate</td>
<td>If set to true, indicates that any events, validation, and conversion associated with the component should happen when request parameter values are applied.</td>
</tr>
<tr>
<td>rendered</td>
<td>Specifies a condition under which the component should be rendered. If the condition is not satisfied, the component is not rendered.</td>
</tr>
<tr>
<td>style</td>
<td>Specifies a Cascading Style Sheet (CSS) style for the tag.</td>
</tr>
<tr>
<td>styleClass</td>
<td>Specifies a CSS class that contains definitions of the styles.</td>
</tr>
<tr>
<td>value</td>
<td>Specifies the value of the component, in the form of a value expression.</td>
</tr>
</tbody>
</table>

All the tag attributes (except id) can accept expressions, as defined by the EL, described in Chapter 6, “Expression Language.”

The id Attribute
The id attribute is not usually required for a component tag but is used when another component or a server-side class must refer to the component. If you don’t include an id attribute, the JavaServer Faces implementation automatically generates a component ID. Unlike most other JavaServer Faces tag attributes, the id attribute takes expressions using only the evaluation syntax described in “Immediate Evaluation” on page 127, which uses the ${} delimiters. For more information on expression syntax, see “Value Expressions” on page 128.

The immediate Attribute
Input components and command components (those that implement the javax.faces.component. ActionSource interface, such as buttons and hyperlinks) can set the immediate attribute to true to force events, validations, and conversions to be processed when request parameter values are applied.

You need to carefully consider how the combination of an input component’s immediate value and a command component’s immediate value determines what happens when the command component is activated.

Assume that you have a page with a button and a field for entering the quantity of a book in a shopping cart. If the immediate attributes of both the button and the field are set to true, the new value entered in the field will be available for any processing associated with the event that is generated when the button is clicked. The event associated with the button as well as the events, validation, and conversion associated with the field are all handled when request parameter values are applied.
If the button’s immediate attribute is set to true but the field’s immediate attribute is set to false, the event associated with the button is processed without updating the field’s local value to the model layer. The reason is that any events, validation, or conversion associated with the field occur after request parameter values are applied.

The bookshowcart.xhtml page of the Duke’s Bookstore case study has examples of components using the immediate attribute to control which component’s data is updated when certain buttons are clicked. The quantity field for each book does not set the immediate attribute, so the value is false (the default).

```xml
<h:inputText id="quantity"
    size="4"
    value="#{item.quantity}"
    title="#{bundle.ItemQuantity}"
    <f:validateLongRange minimum="1"/>
</h:inputText>
```

The immediate attribute of the Continue Shopping hyperlink is set to true, while the immediate attribute of the Update Quantities hyperlink is set to false:

```xml
<h:commandLink id="continue"
    action="bookcatalog"
    immediate="true">
    <h:outputText value="#{bundle.ContinueShopping}"/>
</h:commandLink>
...     <h:commandLink id="update"
        action="#{showcart.update}"
        immediate="false">
        <h:outputText value="#{bundle.UpdateQuantities}"/>
    </h:commandLink>
```

If you click the Continue Shopping hyperlink, none of the changes entered into the quantity input fields will be processed. If you click the Update Quantities hyperlink, the values in the quantity fields will be updated in the shopping cart.

**The rendered Attribute**

A component tag uses a Boolean EL expression along with the rendered attribute to determine whether the component will be rendered. For example, the commandLink component in the following section of a page is not rendered if the cart contains no items:

```xml
<h:commandLink id="check"
    rendered="#{cart.numberOfItems > 0}"
>
    <h:outputText value="#{bundle.CartCheck}"/>
</h:commandLink>
```

Unlike nearly every other JavaServer Faces tag attribute, the rendered attribute is restricted to using rvalue expressions. As explained in “Value and Method Expressions” on page 128, these rvalue expressions can only read data; they cannot write the data back to the data source.
Therefore, expressions used with rendered attributes can use the arithmetic operators and literals that rvalue expressions can use but lvalue expressions cannot use. For example, the expression in the preceding example uses the > operator.

Note – In this example and others, bundle refers to a java.util.ResourceBundle file that contains locale-specific strings to be displayed. Resource bundles are discussed in Chapter 17, “Internationalizing and Localizing Web Applications.”

The style and styleClass Attributes

The style and styleClass attributes allow you to specify CSS styles for the rendered output of your tags. “Displaying Error Messages with the h:message and h:messages Tags” on page 164 describes an example of using the style attribute to specify styles directly in the attribute. A component tag can instead refer to a CSS class.

The following example shows the use of a dataTable tag that references the style class list-background:

```xml
<h:dataTable id="items"
    styleClass="list-background"
    value="#{cart.items}"
    var="book">
```

The style sheet that defines this class is stylesheet.css, which will be included in the application. For more information on defining styles, see Cascading Style Sheets Specification at http://www.w3.org/Style/CSS/.

The value and binding Attributes

A tag representing an output component uses the value and binding attributes to bind its component’s value or instance, respectively, to a data object.

Adding HTML Head and Body Tags

The HTML head (h:head) and body (h:body) tags add HTML page structure to JavaServer Faces web pages.

- The h:head tag represents the head element of an HTML page
- The h:body tag represents the body element of an HTML page

The following is an example of an XHTML page using the usual head and body markup tags:

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```
The following is an example of an XHTML page using h:head and h:body tags:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
      Add a title
    </h:head>
    <h:body>
      Add Content
    </h:body>
</html>
```

Both of the preceding example code segments render the same HTML elements. The head and body tags are useful mainly for resource relocation. For more information on resource relocation, see “Resource Relocation Using h:outputScript and h:outputStylesheet Tags” on page 169.

## Adding a Form Component

An h:form tag represents an input form, which includes child components that can contain data that is either presented to the user or submitted with the form.

Figure 7–1 shows a typical login form in which a user enters a user name and password, then submits the form by clicking the Login button.

![Figure 7–1  A Typical Form](image)

The h:form tag represents the form on the page and encloses all the components that display or collect data from the user, as shown here:
The `<h:form>` tag can also include HTML markup to lay out the components on the page. Note that the `<h:form>` tag itself does not perform any layout; its purpose is to collect data and to declare attributes that can be used by other components in the form.

A page can include multiple `<h:form>` tags, but only the values from the form submitted by the user will be included in the postback request.

### Using Text Components

Text components allow users to view and edit text in web applications. The basic types of text components are as follows:

- **Label**, which displays read-only text
- **Text field**, which allows users to enter text, often to be submitted as part of a form
- **Text area**, which is a type of text field that allows users to enter multiple lines of text
- **Password field**, which is a type of text field that displays a set of characters, such as asterisks, instead of the password text that the user enters

**Figure 7–2** shows examples of these text components.

**FIGURE 7–2  Example Text Components**

<table>
<thead>
<tr>
<th>Label</th>
<th>User Name: Duke</th>
<th>Text Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Password:*******</td>
<td>Password Field</td>
</tr>
<tr>
<td>Comments: A user can enter text across multiple lines.</td>
<td>Text Area</td>
<td></td>
</tr>
</tbody>
</table>

Text components can be categorized as either input or output. A JavaServer Faces output component is rendered as read-only text. An example is a label. A JavaServer Faces input component is rendered as editable text. An example is a text field.

The input and output components can each be rendered in various ways to display more specialized text.

**Table 7–3** lists the tags that represent the input components.
### TABLE 7–3  Input Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>h:inputHidden</td>
<td>Allows a page author to include a hidden variable in a page</td>
</tr>
<tr>
<td>h:inputSecret</td>
<td>The standard password field: accepts one line of text with no spaces and displays it as a set of asterisks as it is typed</td>
</tr>
<tr>
<td>h:inputText</td>
<td>The standard text field: accepts a one-line text string</td>
</tr>
<tr>
<td>h:inputTextarea</td>
<td>The standard text area: accepts multiple lines of text</td>
</tr>
</tbody>
</table>

The input tags support the tag attributes shown in Table 7–4 in addition to those described in “Common Component Tag Attributes” on page 142. Note that this table does not include all the attributes supported by the input tags but just those that are used most often. For the complete list of attributes, refer to the documentation at [http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/](http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/).

### TABLE 7–4  Input Tag Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>converter</td>
<td>Identifies a converter that will be used to convert the component’s local data. See “Using the Standard Converters” on page 175 for more information on how to use this attribute.</td>
</tr>
<tr>
<td>converterMessage</td>
<td>Specifies an error message to display when the converter registered on the component fails.</td>
</tr>
<tr>
<td>dir</td>
<td>Specifies the direction of the text displayed by this component. Acceptable values are LTR, meaning left-to-right, and RTL, meaning right-to-left.</td>
</tr>
<tr>
<td>label</td>
<td>Specifies a name that can be used to identify this component in error messages.</td>
</tr>
<tr>
<td>lang</td>
<td>Specifies the code for the language used in the rendered markup, such as en-US.</td>
</tr>
<tr>
<td>required</td>
<td>Takes a boolean value that indicates whether the user must enter a value in this component.</td>
</tr>
<tr>
<td>requiredMessage</td>
<td>Specifies an error message to display when the user does not enter a value into the component.</td>
</tr>
<tr>
<td>validator</td>
<td>Identifies a method expression pointing to a managed bean method that performs validation on the component’s data. See “Referencing a Method That Performs Validation” on page 187 for an example of using the f:validator tag.</td>
</tr>
</tbody>
</table>
**TABLE 7–4  Input Tag Attributes (Continued)**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validatorMessage</td>
<td>Specifies an error message to display when the validator registered on the component fails to validate the component's local value.</td>
</tr>
<tr>
<td>valueChangeListener</td>
<td>Identifies a method expression that points to a managed bean method that handles the event of entering a value in this component. See &quot;Referencing a Method That Handles a Value-Change Event&quot; on page 188 for an example of using valueChangeListener.</td>
</tr>
</tbody>
</table>

Table 7–5 lists the tags that represent the output components.

**TABLE 7–5  Output Tags**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>h:outputFormat</td>
<td>Displays a formatted message</td>
</tr>
<tr>
<td>h:outputLabel</td>
<td>The standard read-only label: displays a component as a label for a specified input field</td>
</tr>
<tr>
<td>h:outputLink</td>
<td>Displays an &lt;a href&gt; tag that links to another page without generating an action event</td>
</tr>
<tr>
<td>h:outputText</td>
<td>Displays a one-line text string</td>
</tr>
</tbody>
</table>

The output tags support the converter tag attribute in addition to those listed in “Common Component Tag Attributes” on page 142.

The rest of this section explains how to use some of the tags listed in Table 7–3 and Table 7–5. The other tags are written in a similar way.

**Rendering a Text Field with the h:inputText Tag**

The h:inputText tag is used to display a text field. A similar tag, the h:outputText tag, displays a read-only, single-line string. This section shows you how to use the h:inputText tag. The h:outputText tag is written in a similar way.

Here is an example of an h:inputText tag:

```xml
<h:inputText id="name"
    label="Customer Name"
    size="30"
    value="#{cashier.name}"
    required="true"
    requiredMessage="#{bundle.ReqCustomerName}"
    valueChangeListener="dukesbookstore.listeners.NameChanged" />
</h:inputText>
```
The label attribute specifies a user-friendly name that will be used in the substitution parameters of error messages displayed for this component.

The value attribute refers to the name property of a managed bean named CashierBean. This property holds the data for the name component. After the user submits the form, the value of the name property in CashierBean will be set to the text entered in the field corresponding to this tag.

The required attribute causes the page to reload, displaying errors, if the user does not enter a value in the name text field. The JavaServer Faces implementation checks whether the value of the component is null or is an empty string.

If your component must have a non-null value or a String value at least one character in length, you should add a required attribute to your tag and set its value to true. If your tag has a required attribute that is set to true and the value is null or a zero-length string, no other validators that are registered on the tag are called. If your tag does not have a required attribute set to true, other validators that are registered on the tag are called, but those validators must handle the possibility of a null or zero-length string. See “Validating Null and Empty Strings” on page 207 for more information.

**Rendering a Password Field with the h:inputSecret Tag**

The h:inputSecret tag renders an <input type="password"> HTML tag. When the user types a string into this field, a row of asterisks is displayed instead of the text typed by the user. Here is an example:

```xml
<h:inputSecret redisplay="false" value="#{LoginBean.password}" />
```

In this example, the redisplay attribute is set to false. This will prevent the password from being displayed in a query string or in the source file of the resulting HTML page.

**Rendering a Label with the h:outputLabel Tag**

The h:outputLabel tag is used to attach a label to a specified input field for the purpose of making it accessible. The following page uses an h:outputLabel tag to render the label of a check box:

```xml
<h:selectBooleanCheckbox id="fanClub" binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClub" binding="#{cashier.specialOfferText}" >
  <h:outputText id="fanClubLabel" value="#{bundle.DukeFanClub}" />
</h:outputLabel>
...
The `for` attribute of the `h:outputLabel` tag maps to the `id` of the input field to which the label is attached. The `h:outputText` tag nested inside the `h:outputLabel` tag represents the label component. The `value` attribute on the `h:outputText` tag indicates the text that is displayed next to the input field.

Instead of using an `h:outputText` tag for the text displayed as a label, you can simply use the `h:outputLabel` tag's `value` attribute. The following code snippet shows what the previous code snippet would look like if it used the `value` attribute of the `h:outputLabel` tag to specify the text of the label:

```xml
<h:selectBooleanCheckbox id="fanClub" binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClub" binding="#{(cashier.specialOfferText)}"
                value="#{bundle.DukeFanClub}" />
... 
```

**Rendering a Hyperlink with the `h:outputLink` Tag**

The `h:outputLink` tag is used to render a hyperlink that, when clicked, loads another page but does not generate an action event. You should use this tag instead of the `h:commandLink` tag if you always want the URL specified by the `h:outputLink` tag's `value` attribute to open and do not want any processing to be performed when the user clicks the link. Here is an example:

```xml
<h:outputLink value="javadocs"> 
    Documentation for this demo
</h:outputLink>
```

The text in the body of the `h:outputLink` tag identifies the text that the user clicks to get to the next page.

**Displaying a Formatted Message with the `h:outputFormat` Tag**

The `h:outputFormat` tag allows display of concatenated messages as a `MessageFormat` pattern, as described in the API documentation for `java.text.MessageFormat`. Here is an example of an `h:outputFormat` tag:

```xml
<h:outputFormat value="Hello, {0}!">
    <f:param value="#{hello.name}"/>
</h:outputFormat>
```

The `value` attribute specifies the `MessageFormat` pattern. The `f:param` tag specifies the substitution parameters for the message. The value of the parameter replaces the `{0}` in the sentence. If the value of "#{hello.name}" is "Bill", the message displayed in the page is as follows:

Hello, Bill!
An h:outputFormat tag can include more than one f:param tag for those messages that have more than one parameter that must be concatenated into the message. If you have more than one parameter for one message, make sure that you put the f:param tags in the proper order so that the data is inserted in the correct place in the message. Here is the preceding example modified with an additional parameter:

```xml
<h:outputFormat value="Hello, {0}! You are visitor number {1} to the page."/>
  <f:param value="#{hello.name}"/>
  <f:param value="#{bean.numVisitor}"/>
</h:outputFormat>
```

The value of {1} is replaced by the second parameter. The parameter is an EL expression, bean.numVisitor, where the property numVisitor of the managed bean bean keeps track of visitors to the page. This is an example of a value-expression-enabled tag attribute accepting an EL expression. The message displayed in the page is now as follows:

Hello, Bill! You are visitor number 10 to the page.

**Using Command Component Tags for Performing Actions and Navigation**

In JavaServer Faces applications, the button and hyperlink component tags are used to perform actions, such as submitting a form, and for navigating to another page. These tags are called command component tags because they perform an action when activated.

The h:commandButton tag is rendered as a button. The h:commandLink tag is rendered as a hyperlink.

In addition to the tag attributes listed in "Common Component Tag Attributes" on page 142, the h:commandButton and h:commandLink tags can use the following attributes:

- **action**, which is either a logical outcome String or a method expression pointing to a bean method that returns a logical outcome String. In either case, the logical outcome String is used to determine what page to access when the command component tag is activated.
- **actionListener**, which is a method expression pointing to a bean method that processes an action event fired by the command component tag.

See "Referencing a Method That Performs Navigation" on page 187 for more information on using the action attribute. See "Referencing a Method That Handles an Action Event" on page 187 for details on using the actionListener attribute.

**Rendering a Button with the h:commandButton Tag**

If you are using an h:commandButton component tag, the data from the current page is processed when a user clicks the button, and the next page is opened. Here is an example of the h:commandButton tag:
Clicking the button will cause the submit method of CashierBean to be invoked because the action attribute references this method. The submit method performs some processing and returns a logical outcome.

The value attribute of the example h:commandButton tag references the button's label. For information on how to use the action attribute, see "Referencing a Method That Performs Navigation" on page 187.

Rendering a Hyperlink with the h:commandLink Tag

The h:commandLink tag represents an HTML hyperlink and is rendered as an HTML <a> element.

A h:commandLink tag must include a nested h:outputText tag, which represents the text that the user clicks to generate the event. Here is an example:

```xml
<h:commandLink id="Duke" action="bookstore">
    <f:actionListener
        type="dukesbookstore.listeners.LinkBookChangeListener" />
    <h:outputText value="#{bundle.Book201}"/>
</h:commandLink>
```

This tag will render the following HTML:

```html
<a id="_idt16:Duke" href="#" onclick="mojarra.jsfcljs(document.getElementById('j_idt16'),
    {'j_idt16:Duke':'j_idt16:Duke'},''); return false;">My Early Years: Growing Up on Star7, by Duke</a>
```

Note – The h:commandLink tag will render JavaScript scripting language. If you use this tag, make sure that your browser is enabled for JavaScript technology.

Adding Graphics and Images with the h:graphicImage Tag

In a JavaServer Faces application, use the h:graphicImage tag to render an image on a page:

```xml
<h:graphicImage id="mapImage" url="/resources/images/book_all.jpg"/>
```

In this example, the url attribute specifies the path to the image. The URL of the example tag begins with a slash (/), which adds the relative context path of the web application to the beginning of the path to the image.

Alternatively, you can use the facility described in "Web Resources" on page 123 to point to the image location. Here are two examples:
You can use similar syntax to refer to an image in a style sheet. The following syntax in a style sheet specifies that the image is to be found at resources/img/top-background.jpg:

```html
header {
  position: relative;
  height: 150px;
  background: #fff url({resource['img:top-background.jpg']}) repeat-x;
...
```

### Laying Out Components with the h:panelGrid and h:panelGroup Tags

In a JavaServer Faces application, you use a panel as a layout container for a set of other components. A panel is rendered as an HTML table. Table 7–6 lists the tags used to create panels.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Attributes</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>h:panelGrid</td>
<td>columns, columnClasses, footerClass, headerClass, panelClass, rowClasses</td>
<td>Displays a table</td>
</tr>
<tr>
<td>h:panelGroup</td>
<td>layout</td>
<td>Groups a set of components under one parent</td>
</tr>
</tbody>
</table>

The `h:panelGrid` tag is used to represent an entire table. The `h:panelGroup` tag is used to represent rows in a table. Other tags are used to represent individual cells in the rows.

The `columns` attribute defines how to group the data in the table and therefore is required if you want your table to have more than one column. The `h:panelGrid` tag also has a set of optional attributes that specify CSS classes: `columnClasses`, `footerClass`, `headerClass`, `panelClass`, and `rowClasses`.

If the `headerClass` attribute value is specified, the `h:panelGrid` tag must have a header as its first child. Similarly, if a `footerClass` attribute value is specified, the `h:panelGrid` tag must have a footer as its last child.

Here is an example:
The preceding `<h:panelGrid>` tag is rendered as a table that contains components in which a customer inputs personal information. This `<h:panelGrid>` tag uses style sheet classes to format the table. The following code shows the `list-header` definition:

```html
.list-header {
    background-color: #ffffff;
    color: #000000;
    text-align: center;
}
```

Because the `<h:panelGrid>` tag specifies a `headerClass`, the `<h:panelGrid>` tag must contain a header. The example `<h:panelGrid>` tag uses an `<f:facet>` tag for the header. Facets can have only one child, so an `<h:panelGroup>` tag is needed if you want to group more than one component within an `<f:facet>`. The example `<h:panelGrid>` tag has only one cell of data, so an `<h:panelGroup>` tag is not needed. (For more information about facets, see "Using Data-Bound Table Components" on page 161.)

The `<h:panelGroup>` tag has an attribute, `layout`, in addition to those listed in "Common Component Tag Attributes" on page 142. If the `layout` attribute has the value `block`, an HTML `div` element is rendered to enclose the row; otherwise, an HTML `span` element is rendered to...
enclose the row. If you are specifying styles for the h:panelGroup tag, you should set the layout attribute to block in order for the styles to be applied to the components within the h:panelGroup tag. You should do this because styles, such as those that set width and height, are not applied to inline elements, which is how content enclosed by the span element is defined.

An h:panelGroup tag can also be used to encapsulate a nested tree of components so that the tree of components appears as a single component to the parent component.

Data, represented by the nested tags, is grouped into rows according to the value of the columns attribute of the h:panelGrid tag. The columns attribute in the example is set to 2, and therefore the table will have two columns. The column in which each component is displayed is determined by the order in which the component is listed on the page modulo 2. So, if a component is the fifth one in the list of components, that component will be in the 5 modulo 2 column, or column 1.

## Displaying Components for Selecting One Value

Another commonly used component is one that allows a user to select one value, whether it is the only value available or one of a set of choices. The most common tags for this kind of component are as follows:

- An h:selectBooleanCheckbox tag, displayed as a check box, which represents a Boolean state
- An h:selectOneRadio tag, displayed as a set of radio buttons
- An h:selectOneMenu tag, displayed as a drop-down menu, with a scrollable list
- An h:selectOneListBox tag, displayed as a list box, with an unscrollable list

Figure 7–3 shows examples of these components.
Displaying a Check Box Using the h:selectBooleanCheckbox Tag

The h:selectBooleanCheckbox tag is the only tag that JavaServer Faces technology provides for representing a Boolean state.

Here is an example that shows how to use the h:selectBooleanCheckbox tag:

```xml
<h:selectBooleanCheckbox id="fanClub" binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClub" binding="#{cashier.specialOfferText}" value="#{bundle.DukeFanClub}" />
```

This example tag displays a check box to allow users to indicate whether they want to join the Duke Fan Club. The label for the check box is rendered by the h:outputLabel tag. The text is represented by the value attribute.

Displaying a Menu Using the h:selectOneMenu Tag

A component that allows the user to select one value from a set of values can be rendered as a list box, a set of radio buttons, or a menu. This section describes the h:selectOneMenu tag. The h:selectOneRadio and h:selectOneListbox tags are used in a similar way. The h:selectOneListbox tag is similar to the h:selectOneMenu tag except that h:selectOneListbox defines a size attribute that determines how many of the items are displayed at once.

The h:selectOneMenu tag represents a component that contains a list of items from which a user can choose one item. This menu component is also commonly known as a drop-down list or a combo box. The following code snippet shows how the h:selectOneMenu tag is used to allow the user to select a shipping method:

```xml
<h:selectOneMenu id="shippingOption" required="true" value="#{cashier.shippingOption}"
    <f:selectItem itemValue="2" itemLabel="#{bundle.QuickShip}"/>
```
The value attribute of the h:selectOneMenu tag maps to the property that holds the currently selected item’s value. You are not required to provide a value for the currently selected item. If you don’t provide a value, the first item in the list is selected by default.

Like the h:selectOneRadio tag, the h:selectOneMenu tag must contain either an f:selectItems tag or a set of f:selectItem tags for representing the items in the list. "Using the f:selectItem and f:selectItems Tags" on page 159 describes these tags.

Displaying Components for Selecting Multiple Values

In some cases, you need to allow your users to select multiple values rather than just one value from a list of choices. You can do this using one of the following component tags:

- An h:selectManyCheckbox tag, displayed as a set of check boxes
- An h:selectManyMenu tag, displayed as a drop-down menu
- An h:selectManyListBox tag, displayed as a list box

Figure 7–4 shows examples of these components.

FIGURE 7–4 Example Components for Selecting Multiple Values

These tags allow the user to select zero or more values from a set of values. This section explains the h:selectManyCheckbox tag. The h:selectManyListBox and h:selectManyMenu tags are used in a similar way.

Unlike a menu, a list box displays a subset of items in a box; a menu displays only one item at a time when the user is not selecting the menu. The size attribute of the h:selectManyListBox tag determines the number of items displayed at one time. The list box includes a scroll bar for scrolling through any remaining items in the list.
The `h:selectManyCheckbox` tag renders a set of check boxes, with each check box representing one value that can be selected:

```xml
<h:selectManyCheckbox id="newslettercheckbox" layout="pageDirection" value="#{cashier.newsletters}"
  <f:selectItems value="#{cashier.newsletterItems}"/>
</h:selectManyCheckbox>
```

The value attribute of the `h:selectManyCheckbox` tag identifies the newsletters property of the `CashierBean` managed bean. This property holds the values of the currently selected items from the set of check boxes. You are not required to provide a value for the currently selected items. If you don’t provide a value, the first item in the list is selected by default. In the `CashierBean` managed bean, this value is instantiated to 0, so no items are selected by default.

The layout attribute indicates how the set of check boxes is arranged on the page. Because layout is set to `pageDirection`, the check boxes are arranged vertically. The default is `lineDirection`, which aligns the check boxes horizontally.

The `h:selectManyCheckbox` tag must also contain a tag or set of tags representing the set of check boxes. To represent a set of items, you use the `f:selectItems` tag. To represent each item individually, you use the `f:selectItem` tag. The following section explains these tags in more detail.

### Using the `f:selectItem` and `f:selectItems` Tags

The `f:selectItem` and `f:selectItems` tags represent components that can be nested inside a component that allows you to select one or multiple items. An `f:selectItem` tag contains the value, label, and description of a single item. An `f:selectItems` tag contains the values, labels, and descriptions of the entire list of items.

You can use either a set of `f:selectItem` tags or a single `f:selectItems` tag within your component tag.

The advantages of using the `f:selectItems` tag are as follows.

- Items can be represented by using different data structures, including `Array`, `Map`, and `Collection`. The value of the `f:selectItems` tag can represent even a generic collection of `POJOs`.
- Different lists can be concatenated into a single component, and the lists can be grouped within the component.
- Values can be generated dynamically at runtime.
Adding Components to a Page Using HTML Tags

The advantages of using f:selectItem are as follows:

- Items in the list can be defined from the page.
- Less code is needed in the bean for the f:selectItem properties.

The rest of this section shows you how to use the f:selectItems and f:selectItem tags.

**Using the f:selectItems Tag**

The following example from "Displaying Components for Selecting Multiple Values" on page 158 shows how to use the h:selectManyCheckbox tag:

```xml
<h:selectManyCheckbox id="newsletterCheckbox"
  layout="pageDirection"
  value="#{cashier.newsletters}">
  <f:selectItems value="#{cashier.newsletterItems}"/>
</h:selectManyCheckbox>
```

The value attribute of the f:selectItems tag is bound to the managed bean property `cashier.newsletterItems`. The individual SelectItem objects are created programmatically in the managed bean.

See "UISelectItems Properties" on page 198 for information on how to write a managed bean property for one of these tags.

**Using the f:selectItem Tag**

The f:selectItem tag represents a single item in a list of items. Here is the example from "Displaying a Menu Using the h:selectOneMenu Tag" on page 157 once again:

```xml
<h:selectOneMenu id="shippingOption"
  required="true"
  value="#{cashier.shippingOption}"
  itemValue="2"
  itemLabel="#{bundle.QuickShip}"
  itemValue="5"
  itemLabel="#{bundle.NormalShip}"
  itemValue="7"
  itemLabel="#{bundle.SaverShip}"
</h:selectOneMenu>
```

The itemValue attribute represents the value for the f:selectItem tag. The itemLabel attribute represents the String that appears in the drop-down menu component on the page.

The itemValue and itemLabel attributes are value-binding-enabled, meaning that they can use value-binding expressions to refer to values in external objects. These attributes can also define literal values, as shown in the example h:selectOneMenu tag.
Displaying the Results from Selection Components

If you display components that allow a user to select values, you may also want to display the result of the selection.

For example, you might want to thank a user who selected the checkbox to join the Duke Fan Club, as described in "Displaying a Check Box Using the h:selectBooleanCheckbox Tag" on page 157. Because the checkbox is bound to the specialOffer property of CashierBean, a javax.faces.component.UISelectBoolean value, you can call the isSelected method of the property to determine whether to render a thank-you message:

```html
<h:outputText value="#{bundle.DukeFanClubThanks}" 
    rendered="#{cashier.specialOffer.isSelected()}"/>
```

Similarly, you might want to acknowledge that a user subscribed to newsletters using the h:selectManyCheckbox tag, as described in "Displaying Components for Selecting Multiple Values" on page 158. To do so, you can retrieve the value of the newsletters property, the String array that holds the selected items:

```html
<h:outputText value="#{bundle.NewsletterThanks}" 
    rendered="#{!empty cashier.newsletters}"/>
  </ul>
  <ui:repeat value="#{cashier.newsletters}" var="nli">
    <li><h:outputText value="#{nli}" /></li>
  </ui:repeat>
</ul>
```

An introductory thank-you message is displayed only if the newsletters array is not empty. Then a ui:repeat tag, a simple way to show values in a loop, displays the contents of the selected items in an itemized list. (This tag is listed in Table 5–2.)

Using Data-Bound Table Components

Data-bound table components display relational data in a tabular format. In a JavaServer Faces application, the h: dataTable component tag supports binding to a collection of data objects and displays the data as an HTML table. The h:column tag represents a column of data within the table, iterating over each record in the data source, which is displayed as a row. Here is an example:

```html
<h:dataTable id="items" 
    captionStyle="font-weight: bold"
    columnClasses="list-column-center, list-column-left, list-column-right, list-column-center"
    footerClass="list-footer"
    headerClass="list-header"
    rowClasses="list-row-even, list-row-odd"
    styleClass="list-background"
    summary="#{bundle.ShoppingCart}"
```
The example `h:dataTable` tag displays the books in the shopping cart, as well as the quantity of each book in the shopping cart, the prices, and a set of buttons the user can click to remove books from the shopping cart.

The `h:column` tags represent columns of data in a data component. While the data component is iterating over the rows of data, it processes the column component associated with each `h:column` tag for each row in the table.

The `h:dataTable` tag shown in the preceding code example iterates through the list of books (`cart.items`) in the shopping cart and displays their titles, authors, and prices. Each time the `h:dataTable` tag iterates through the list of books, it renders one cell in each column.

The `h:dataTable` and `h:column` tags use facets to represent parts of the table that are not repeated or updated. These parts include headers, footers, and captions.

In the preceding example, `h:column` tags include `f:facet` tags for representing column headers or footers. The `h:column` tag allows you to control the styles of these headers and footers by
supporting the headerClass and footerClass attributes. These attributes accept space-separated lists of CSS classes, which will be applied to the header and footer cells of the corresponding column in the rendered table.

Facets can have only one child, so an h:panelGroup tag is needed if you want to group more than one component within an f:facet. Because the facet tag representing the footer includes more than one tag, the h:panelGroup tag is needed to group those tags. Finally, this h:dataTable tag includes an f:facet tag with its name attribute set to caption, causing a table caption to be rendered above the table.

This table is a classic use case for a data component because the number of books might not be known to the application developer or the page author when that application is developed. The data component can dynamically adjust the number of rows of the table to accommodate the underlying data.

The value attribute of an h:dataTable tag references the data to be included in the table. This data can take the form of any of the following:

- A list of beans
- An array of beans
- A single bean
- A javax.faces.model.DataModel object
- A java.sql.ResultSet object
- A javax.servlet.jsp.jstl.sql.Result object
- A javax.sql.RowSet object

All data sources for data components have a javax.faces.model.DataModel wrapper. Unless you explicitly construct a DataModel wrapper, the JavaServer Faces implementation will create one around data of any of the other acceptable types. See "Writing Bean Properties" on page 192 for more information on how to write properties for use with a data component.

The var attribute specifies a name that is used by the components within the h:dataTable tag as an alias to the data referenced in the value attribute of h:dataTable.

In the example h:dataTable tag, the value attribute points to a list of books. The var attribute points to a single book in that list. As the h:dataTable tag iterates through the list, each reference to item points to the current book in the list.

The h:dataTable tag also has the ability to display only a subset of the underlying data. This feature is not shown in the preceding example. To display a subset of the data, you use the optional first and rows attributes.

The first attribute specifies the first row to be displayed. The rows attribute specifies the number of rows, starting with the first row, to be displayed. For example, if you wanted to display records 2 through 10 of the underlying data, you would set first to 2 and rows to 9.

When you display a subset of the data in your pages, you might want to consider including a link or button that causes subsequent rows to display when clicked. By default, both first and rows are set to zero, and this causes all the rows of the underlying data to display.
Table 7–7 shows the optional attributes for the h:dataTable tag.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Defines Styles for</th>
</tr>
</thead>
<tbody>
<tr>
<td>captionClass</td>
<td>Table caption</td>
</tr>
<tr>
<td>columnClasses</td>
<td>All the columns</td>
</tr>
<tr>
<td>footerClass</td>
<td>Footer</td>
</tr>
<tr>
<td>headerClass</td>
<td>Header</td>
</tr>
<tr>
<td>rowClasses</td>
<td>Rows</td>
</tr>
<tr>
<td>styleClass</td>
<td>The entire table</td>
</tr>
</tbody>
</table>

Each of the attributes in Table 7–7 can specify more than one style. If columnClasses or rowClasses specifies more than one style, the styles are applied to the columns or rows in the order that the styles are listed in the attribute. For example, if columnClasses specifies styles list-column-center and list-column-right and if the table has two columns, the first column will have style list-column-center, and the second column will have style list-column-right.

If the style attribute specifies more styles than there are columns or rows, the remaining styles will be assigned to columns or rows starting from the first column or row. Similarly, if the style attribute specifies fewer styles than there are columns or rows, the remaining columns or rows will be assigned styles starting from the first style.

### Displaying Error Messages with the h:message and h:messages Tags

The h:message and h:messages tags are used to display error messages when conversion or validation fails. The h:message tag displays error messages related to a specific input component, whereas the h:messages tag displays the error messages for the entire page.

Here is an example h:message tag from the guessnumber application:

```xml
<p>
  <h:inputText id="userNo"
              title="Type a number from 0 to 10:"
              value="#{userNumberBean.userNumber}"
              
              <f:validateLongRange minimum="#{userNumberBean.minimum}"
                             maximum="#{userNumberBean.maximum}" />
  </h:inputText>
  <h:commandButton id="submit" value="Submit"
                   action="response"/>
</p>
```
The `for` attribute refers to the ID of the component that generated the error message. The error message is displayed at the same location that the `h:message` tag appears in the page. In this case, the error message will appear after the Submit button.

The `style` attribute allows you to specify the style of the text of the message. In the example in this section, the text will be a shade of red, New Century Schoolbook, serif font family, and oblique style, and a line will appear over the text. The message and messages tags support many other attributes for defining styles. For more information on these attributes, refer to the documentation at http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/.

Another attribute supported by the `h:messages` tag is the `layout` attribute. Its default value is `list`, which indicates that the messages are displayed in a bullet list using the HTML `ul` and `li` elements. If you set the attribute value to `table`, the messages will be rendered in a table using the HTML `table` element.

The preceding example shows a standard validator that is registered on the input component. The message tag displays the error message that is associated with this validator when the validator cannot validate the input component's value. In general, when you register a converter or validator on a component, you are queueing the error messages associated with the converter or validator on the component. The `h:message` and `h:messages` tags display the appropriate error messages that are queued on the component when the validators or converters registered on that component fail to convert or validate the component's value.

Standard error messages are provided with standard converters and standard validators. An application architect can override these standard messages and supply error messages for custom converters and validators by registering custom error messages with the application.

### Creating Bookmarkable URLs with the h:button and h:link Tags

The ability to create bookmarkable URLs refers to the ability to generate hyperlinks based on a specified navigation outcome and on component parameters.

In HTTP, most browsers by default send GET requests for URL retrieval and POST requests for data processing. The GET requests can have query parameters and can be cached, which is not advised for POST requests, which send data to servers for processing. The other JavaServer Faces Technology in Web Pages
Faces tags capable of generating hyperlinks use either simple GET requests, as in the case of h:outputLink, or POST requests, as in the case of h:commandLink or h:commandButton tags. GET requests with query parameters provide finer granularity to URL strings. These URLs are created with one or more name=value parameters appended to the simple URL after a ? character and separated by either &; or & strings.

To create a bookmarkable URL, use an h:link or h:button tag. Both of these tags can generate a hyperlink based on the outcome attribute of the component. For example:

```html
<h:link outcome="somepage" value="Message" />
```

The h:link tag will generate a URL link that points to the somepage.xhtml file on the same server. The following sample HTML is generated from the preceding tag, assuming that the application name is simplebookmark:

```html
<a href="/simplebookmark/faces/somepage.xhtml">Message</a>
```

This is a simple GET request that cannot pass any data from page to page. To create more complex GET requests and utilize the complete functionality of the h:link tag, use view parameters.

### Using View Parameters to Configure Bookmarkable URLs

To pass a parameter from one page to another, use the `includeViewParams` attribute in your h:link tag and, in addition, use an f:param tag to specify the name and value to be passed. Here the h:link tag specifies the outcome page as personal.xhtml and provides a parameter named `Result` whose value is a managed bean property:

```html
<h:body>
  <h:form>
    <h:graphicImage url="duke.waving.gif" alt="Duke waving his hand"/>
    <h:body>
      <h2>Hello, #{hello.name}!</h2>
      <p>I've made your personal greeting page!</p>
      <h:link outcome="personal" value="personal greeting page!"
        includeViewParams="true">
        <f:param name="Result" value="#{hello.name}"/>
      </h:link>
      <h:commandButton id="back" value="Back" action="/index" />
    </h:body>
  </h:form>
</h:body>
```

If the includeViewParams attribute is set on the component, the view parameters are added to the hyperlink. Therefore, the resulting URL will look something like this if the value of hello.name is Timmy:

```
```
On the outcome page, specify the core tags `f:metadata` and `f:viewparam` as the source of parameters for configuring the URLs. View parameters are declared as part of `f:metadata` for a page, as shown in the following example:

```xml
<f:metadata>
  <f:viewParam name="Result" value="#{hello.name}"/>
</f:metadata>
```

This allows you to specify the bean property value on the page:

```xml
<h:outputText value="Howdy, #{hello.name}"/>
```

As a view parameter, the name also appears in the page's URL. If you edit the URL, you change the output on the page.

Because the URL can be the result of various parameter values, the order of the URL creation has been predefined. The order in which the various parameter values are read is as follows:

1. Component
2. Navigation-case parameters
3. View parameters

### The bookmarks Example Application

The bookmarks example application modifies the `hello1` application described in “Web Modules: The `hello1` Example” on page 84 to use a bookmarkable URL that uses view parameters.

Like `hello1`, the application includes the `Hello.java` managed bean, an `index.xhtml` page, and a `response.xhtml` page. In addition, it includes a `personal.xhtml` page, to which a bookmarkable URL and view parameters are passed from the `response.xhtml` page, as described in “Using View Parameters to Configure Bookmarkable URLs” on page 166.

### Building, Packaging, Deploying, and Running the bookmarks Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the bookmarks example. The source code for this example is available in the `tut-install/examples/web/bookmarks/` directory.

#### To Build, Package, and Deploy the bookmarks Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:

   `tut-install/examples/web/bookmarks/`
Select the bookmarks folder.

Select the Open as Main Project check box.

Click Open Project.

In the Projects tab, right-click the bookmarks project and select Deploy. This option builds and deploys the example application to your GlassFish Server instance.

▼ To Build, Package, and Deploy the bookmarks Example Using Ant

1 In a terminal window, go to:
   tut-install/examples/web/bookmarks/

2 Type the following command:
   ant
   This command calls the default target, which builds and packages the application into a WAR file, bookmarks.war, that is located in the dist directory.

3 Make sure that the GlassFish Server is started.

4 To deploy the application, type the following command:
   ant deploy

▼ To Run the bookmarks Example

1 Open a web browser.

2 Type the following URL in your web browser:
   http://localhost:8080/bookmarks

3 In the text field, type a name and click Submit.

4 On the response page, move your mouse over the “personal greeting page” link to view the URL with the view parameter, then click the link.
   The personal.xhtml page opens, displaying a greeting to the name you typed.

5 In the URL field, modify the Result parameter value and press Enter.
   The name in the greeting changes to what you typed.
Resource Relocation Using h:outputScript and h:outputStylesheet Tags

Resource relocation refers to the ability of a JavaServer Faces application to specify the location where a resource can be rendered. Resource relocation can be defined with the following HTML tags:

- h:outputScript
- h:outputStylesheet

These tags have name and target attributes, which can be used to define the render location. For a complete list of attributes for these tags, see the documentation at http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/.

For the h:outputScript tag, the name and target attributes define where the output of a resource may appear. Here is an example:

```html
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://java.sun.com/jsf/html">
  <h:head id="head">
    <title>Resource Relocation</title>
  </h:head>
  <h:body id="body">
    <h:form id="form" name="form">
      <h:outputScript name="hello.js"/>
      <h:outputStylesheet name="hello.css"/>
    </h:form>
  </h:body>
</html>
```

Since the target attribute is not defined in the tags, the style sheet hello.css is rendered in the head element of the page, and the hello.js script is rendered in the body of the page.

Here is the HTML generated by the preceding code:

```html
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Resource Relocation</title>
    <link type="text/css" rel="stylesheet" href="/context-root/faces/javax.faces.resource/hello.css"/>
  </head>
  <body>
    <form id="form" name="form" method="post" action="..." enctype="...">
      <script type="text/javascript" src="/context-root/face..." charset="...">
      </script>
    </form>
  </body>
</html>
```
If you set the `target` attribute for the `h:outputScript` tag, the incoming GET request provides the location parameter. Here is an example:

```html
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://java.sun.com/jsf/html">
  <h:head id="head">
    <title>Resource Relocation</title>
  </h:head>
  <h:body id="body">
    <h:form id="form">
      <h:outputScript name="hello.js" target="#{param.location}"/>
      <h:outputStylesheet name="hello.css"/>
    </h:form>
  </h:body>
</html>
```

In this case, if the incoming request does not provide a location parameter, the default locations will still apply: The style sheet is rendered in the head, and the script is rendered inline. However, if the incoming request specifies the location parameter as the head, both the style sheet and the script will be rendered in the head element.

The HTML generated by the preceding code is as follows:

```html
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Resource Relocation</title>
    <link type="text/css" rel="stylesheet" href="/context-root/faces/javax.faces.resource/hello.css"/>
    <script type="text/javascript" src="/context-root/faces/javax.faces.resource/hello.js"></script>
  </head>
  <body>
    <form id="form" name="form" method="post" action="..." enctype="...">
    </form>
  </body>
</html>
```

Similarly, if the incoming request provides the location parameter as the body, the script will be rendered in the body element.

The preceding section describes simple uses for resource relocation. That feature can add even more functionality for the components and pages. A page author does not have to know the location of a resource or its placement.

By using a `@ResourceDependency` annotation for the components, component authors can define the resources for the component, such as a style sheet and script. This allows the page authors freedom from defining resource locations.
Using Core Tags

The tags included in the JavaServer Faces core tag library are used to perform core actions that are not performed by HTML tags.

Table 7–8 lists the event handling core tags.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>f:actionListener</td>
<td>Adds an action listener to a parent component</td>
</tr>
<tr>
<td>f:phaseListener</td>
<td>Adds a PhaseListener to a page</td>
</tr>
<tr>
<td>f:setPropertyActionListener</td>
<td>Registers a special action listener whose sole purpose is to push a value into a managed bean when a form is submitted</td>
</tr>
<tr>
<td>f:valueChangeListener</td>
<td>Adds a value-change listener to a parent component</td>
</tr>
</tbody>
</table>

Table 7–9 lists the data conversion core tags.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>f:converter</td>
<td>Adds an arbitrary converter to the parent component</td>
</tr>
<tr>
<td>f:convertDateTime</td>
<td>Adds a DateTimeConverter instance to the parent component</td>
</tr>
<tr>
<td>f:convertNumber</td>
<td>Adds a NumberConverter instance to the parent component</td>
</tr>
</tbody>
</table>

Table 7–10 lists the facet core tags.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>f:facet</td>
<td>Adds a nested component that has a special relationship to its enclosing tag</td>
</tr>
<tr>
<td>f:metadata</td>
<td>Registers a facet on a parent component</td>
</tr>
</tbody>
</table>

Table 7–11 lists the core tags that represent items in a list.
TABLE 7–11 Core Tags that Represent Items in a List

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>f:selectItem</td>
<td>Represents one item in a list of items</td>
</tr>
<tr>
<td>f:selectItems</td>
<td>Represents a set of items</td>
</tr>
</tbody>
</table>

Table 7–12 lists the validator core tags.

TABLE 7–12 Validator Core Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>f:validateDoubleRange</td>
<td>Adds a DoubleRangeValidator to a component</td>
</tr>
<tr>
<td>f:validateLength</td>
<td>Adds a LengthValidator to a component</td>
</tr>
<tr>
<td>f:validateLongRange</td>
<td>Adds a LongRangeValidator to a component</td>
</tr>
<tr>
<td>f:validator</td>
<td>Adds a custom validator to a component</td>
</tr>
<tr>
<td>f:validateRegEx</td>
<td>Adds a RegExValidator to a component</td>
</tr>
<tr>
<td>f:validateBean</td>
<td>Delegates the validation of a local value to a BeanValidator</td>
</tr>
<tr>
<td>f:validateRequired</td>
<td>Ensures the presence of a value in a component</td>
</tr>
</tbody>
</table>

Table 7–13 lists the core tags that fall into other categories.

TABLE 7–13 Miscellaneous Core Tags

<table>
<thead>
<tr>
<th>Tag Category</th>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute configuration</td>
<td>f:attribute</td>
<td>Adds configurable attributes to a parent component</td>
</tr>
<tr>
<td>Localization</td>
<td>f:loadBundle</td>
<td>Specifies a ResourceBundle that is exposed as a Map</td>
</tr>
<tr>
<td>Parameter substitution</td>
<td>f:param</td>
<td>Substitutes parameters into a MessageFormat instance and adds query string name-value pairs to a URL</td>
</tr>
<tr>
<td>Ajax</td>
<td>f:ajax</td>
<td>Associates an Ajax action with a single component or a group of components based on placement</td>
</tr>
</tbody>
</table>
TABLE 7–13  Miscellaneous Core Tags  (Continued)

<table>
<thead>
<tr>
<th>Tag Category</th>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>f:event</td>
<td>Allows installing a ComponentSystemEventListener on a component</td>
</tr>
</tbody>
</table>

These tags, which are used in conjunction with component tags, are explained in other sections of this tutorial. Table 7–14 lists the sections that explain how to use specific core tags.

TABLE 7–14  Where the Core Tags Are Explained

<table>
<thead>
<tr>
<th>Tags</th>
<th>Where Explained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event handling tags</td>
<td>&quot;Registering Listeners on Components” on page 180</td>
</tr>
<tr>
<td>Data conversion tags</td>
<td>“Using the Standard Converters” on page 175</td>
</tr>
<tr>
<td>f:facet</td>
<td>&quot;Using Data-Bound Table Components” on page 161 and &quot;Laying Out Components with the h:panelGrid and h:panelGroup Tags” on page 154</td>
</tr>
<tr>
<td>f:loadBundle</td>
<td>&quot;Setting the Resource Bundle” on page 357</td>
</tr>
<tr>
<td>f:metadata</td>
<td>“Using View Parameters to Configure Bookmarkable URLs” on page 166</td>
</tr>
<tr>
<td>f:param</td>
<td>“Displaying a Formatted Message with the h:outputFormat Tag” on page 151</td>
</tr>
<tr>
<td>f:selectItem, f:selectItems</td>
<td>“Using the f:selectItem and f:selectItems Tags” on page 159</td>
</tr>
<tr>
<td>Validator tags</td>
<td>&quot;Using the Standard Validators” on page 183</td>
</tr>
<tr>
<td>f:ajax</td>
<td>Chapter 11, “Using Ajax with JavaServer Faces Technology”</td>
</tr>
</tbody>
</table>
Using Converters, Listeners, and Validators

The previous chapter described components and explained how to add them to a web page. This chapter provides information on adding more functionality to the components through converters, listeners, and validators.

- Converters are used to convert data that is received from the input components.
- Listeners are used to listen to the events happening in the page and perform actions as defined.
- Validators are used to validate the data that is received from the input components.

The following topics are addressed here:

- “Using the Standard Converters” on page 175
- “Registering Listeners on Components” on page 180
- “Using the Standard Validators” on page 183
- “Referencing a Managed Bean Method” on page 186

Using the Standard Converters

The JavaServer Faces implementation provides a set of Converter implementations that you can use to convert component data. For more information on the conceptual details of the conversion model, see “Conversion Model” on page 220. The standard Converter implementations, located in the javax.faces.convert package, are as follows:

- BigDecimalConverter
- BigIntegerConverter
- BooleanConverter
- ByteConverter
- CharacterConverter
- DateTimeConverter
- DoubleConverter
- EnumConverter
Using the Standard Converters

- FloatConverter
- IntegerConverter
- LongConverter
- NumberConverter
- ShortConverter

A standard error message is associated with each of these converters. If you have registered one of these converters onto a component on your page, and the converter is not able to convert the component’s value, the converter’s error message will display on the page. For example, the following error message appears if BigIntegerConverter fails to convert a value:

{0} must be a number consisting of one or more digits

In this case, the {0} substitution parameter will be replaced with the name of the input component on which the converter is registered.

Two of the standard converters (DateTimeConverter and NumberConverter) have their own tags, which allow you to configure the format of the component data using the tag attributes. For more information about using DateTimeConverter, see “Using DateTimeConverter” on page 177. For more information about using NumberConverter, see “Using NumberConverter” on page 179. The following section explains how to convert a component’s value, including how to register other standard converters with a component.

Converting a Component’s Value

To use a particular converter to convert a component’s value, you need to register the converter onto the component. You can register any of the standard converters in one of the following ways:

- Nest one of the standard converter tags inside the component’s tag. These tags are `convertDateTime` and `convertNumber`, which are described in “Using DateTimeConverter” on page 177 and “Using NumberConverter” on page 179, respectively.
- Bind the value of the component to a managed bean property of the same type as the converter.
- Refer to the converter from the component tag’s converter attribute.
- Nest a converter tag inside of the component tag, and use either the converter tag’s converterId attribute or its binding attribute to refer to the converter.

As an example of the second technique, if you want a component’s data to be converted to an Integer, you can simply bind the component’s value to a managed bean property. Here is an example:

```
Integer age = 0;
pUBLIC Integer getAge(){ return age;
PUBLIC void setAge(Integer age) {this.age = age;
```
If the component is not bound to a bean property, you can use the third technique by using the converter attribute directly on the component tag:

```html
<h:inputText
   converter="javax.faces.convert.IntegerConverter" />
```

This example shows the converter attribute referring to the fully qualified class name of the converter. The converter attribute can also take the ID of the component.

The data from the inputText tag in this example will be converted to a `java.lang.Integer` value. The Integer type is a supported type of NumberConverter. If you don’t need to specify any formatting instructions using the convertNumber tag attributes, and if one of the standard converters will suffice, you can simply reference that converter by using the component tag’s converter attribute.

Finally, you can nest a converter tag within the component tag and use either the converter tag’s converterId attribute or its binding attribute to reference the converter.

The converterId attribute must reference the converter’s ID. Here is an example:

```html
<h:inputText value="#{loginBean.age}" />
<f:converter converterId="Integer" />
</h:inputText>
```

Instead of using the converterId attribute, the converter tag can use the binding attribute. The binding attribute must resolve to a bean property that accepts and returns an appropriate Converter instance.

### Using DateTimeConverter

You can convert a component’s data to a `java.util.Date` by nesting the convertDateTime tag inside the component tag. The convertDateTime tag has several attributes that allow you to specify the format and type of the data. Table 8–1 lists the attributes.

Here is a simple example of a convertDateTime tag:

```html
<h:outputText value="#{cashier.shipDate}"
   <f:convertDateTime type="date" dateStyle="full" />
</h:outputText>
```

When binding the DateTimeConverter to a component, ensure that the managed bean property to which the component is bound is of type `java.util.Date`. In the preceding example, `cashier.shipDate` must be of type `java.util.Date`.

The example tag can display the following output:

Saturday, September 25, 2011
You can also display the same date and time by using the following tag where the date format is specified:

```html
<h:outputText value="#{cashier.shipDate}"
    <f:convertDateTime
        pattern="EEEEEEE, MMM dd, yyyy" />
</h:outputText>
```

If you want to display the example date in Spanish, you can use the `locale` attribute:

```html
<h:outputText value="#{cashier.shipDate}"
    <f:convertDateTime dateStyle="full"
        locale="es"
        timeStyle="long" type="both" />
</h:outputText>
```

This tag would display the following output:

jueves 27 de octubre de 2011 15:07:04 GMT

Refer to the “Customizing Formats” lesson of the Java Tutorial at [http://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html](http://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html) for more information on how to format the output using the `pattern` attribute of the `convertDateTime` tag.

### TABLE 8–1  Attributes for the `convertDateTime` Tag

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binding</td>
<td>DateTimeConverter</td>
<td>Used to bind a converter to a managed bean property.</td>
</tr>
<tr>
<td>dateStyle</td>
<td>String</td>
<td>Defines the format, as specified by <code>java.text.SimpleDateFormat</code>, of a date or the date part of a date string. Applied only if <code>type</code> is <code>date</code> or both and if <code>pattern</code> is not defined. Valid values: <code>default</code>, <code>short</code>, <code>medium</code>, <code>long</code>, and <code>full</code>. If no value is specified, <code>default</code> is used.</td>
</tr>
<tr>
<td>for</td>
<td>String</td>
<td>Used with composite components. Refers to one of the objects within the composite component inside which this tag is nested.</td>
</tr>
<tr>
<td>locale</td>
<td>String or Locale</td>
<td>Locale whose predefined styles for dates and times are used during formatting or parsing. If not specified, the <code>Locale</code> returned by <code>javax.faces.context.FacesContext.getLocale</code> will be used.</td>
</tr>
<tr>
<td>pattern</td>
<td>String</td>
<td>Custom formatting pattern that determines how the date/time string should be formatted and parsed. If this attribute is specified, <code>dateStyle</code>, <code>timeStyle</code>, and <code>type</code> attributes are ignored.</td>
</tr>
<tr>
<td>timeStyle</td>
<td>String</td>
<td>Defines the format, as specified by <code>java.text.SimpleDateFormat</code>, of a time or the time part of a date string. Applied only if <code>type</code> is <code>time</code> and if <code>pattern</code> is not defined. Valid values: <code>default</code>, <code>short</code>, <code>medium</code>, <code>long</code>, and <code>full</code>. If no value is specified, <code>default</code> is used.</td>
</tr>
</tbody>
</table>
TABLE 8–1 Attributes for the convertDateTime Tag  (Continued)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeZone</td>
<td>String or TimeZone</td>
<td>Time zone in which to interpret any time information in the date string.</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Specifies whether the string value will contain a date, a time, or both.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid values are date, time, or both. If no value is specified, date is used.</td>
</tr>
</tbody>
</table>

Using NumberConverter

You can convert a component’s data to a java.lang.Number by nesting the convertNumber tag inside the component tag. The convertNumber tag has several attributes that allow you to specify the format and type of the data. Table 8–2 lists the attributes.

The following example uses a convertNumber tag to display the total prices of the contents of a shopping cart:

```xml
<h:outputText value="#{cart.total}">
    <f:convertNumber currencySymbol="$
    type="currency"/>
</h:outputText>
```

When binding the NumberConverter to a component, ensure that the managed bean property to which the component is bound is of a primitive type or has a type of java.lang.Number. In the preceding example, cart.total is of type double.

Here is an example of a number that this tag can display:

$934

This result can also be displayed by using the following tag, where the currency pattern is specified:

```xml
<h:outputText id="cartTotal" value="#{cart.Total}"
    <f:convertNumber pattern="$##" />
</h:outputText>
```

See the “Customizing Formats” lesson of the Java Tutorial at http://docs.oracle.com/javase/tutorial/i18n/format/decimalFormat.html for more information on how to format the output by using the pattern attribute of the convertNumber tag.

TABLE 8–2 Attributes for the convertNumber Tag

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binding</td>
<td>NumberConverter</td>
<td>Used to bind a converter to a managed bean property.</td>
</tr>
</tbody>
</table>
### TABLE 8–2 Attributes for the convertNumber Tag  *(Continued)*

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>currencyCode</td>
<td>String</td>
<td>ISO 4217 currency code, used only when formatting currencies.</td>
</tr>
<tr>
<td>currencySymbol</td>
<td>String</td>
<td>Currency symbol, applied only when formatting currencies.</td>
</tr>
<tr>
<td>for</td>
<td>String</td>
<td>Used with composite components. Refers to one of the objects within the composite component inside which this tag is nested.</td>
</tr>
<tr>
<td>groupingUsed</td>
<td>Boolean</td>
<td>Specifies whether formatted output contains grouping separators.</td>
</tr>
<tr>
<td>integerOnly</td>
<td>Boolean</td>
<td>Specifies whether only the integer part of the value will be parsed.</td>
</tr>
<tr>
<td>locale</td>
<td>String or Locale</td>
<td>Locale whose number styles are used to format or parse data.</td>
</tr>
<tr>
<td>maxFractionDigits</td>
<td>int</td>
<td>Maximum number of digits formatted in the fractional part of the output.</td>
</tr>
<tr>
<td>maxIntegerDigits</td>
<td>int</td>
<td>Maximum number of digits formatted in the integer part of the output.</td>
</tr>
<tr>
<td>minFractionDigits</td>
<td>int</td>
<td>Minimum number of digits formatted in the fractional part of the output.</td>
</tr>
<tr>
<td>minIntegerDigits</td>
<td>int</td>
<td>Minimum number of digits formatted in the integer part of the output.</td>
</tr>
<tr>
<td>pattern</td>
<td>String</td>
<td>Custom formatting pattern that determines how the number string is formatted and parsed.</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Specifies whether the string value is parsed and formatted as a number, currency, or percentage. If not specified, number is used.</td>
</tr>
</tbody>
</table>

### Registering Listeners on Components

An application developer can implement listeners as classes or as managed bean methods. If a listener is a managed bean method, the page author references the method from either the component’s `valueChangeListener` attribute or its `actionListener` attribute. If the listener is a class, the page author can reference the listener from either an `<f:valueChangeListener>` tag or an `<f:actionListener>` tag and nest the tag inside the component tag to register the listener on the component.
"Referencing a Method That Handles an Action Event" on page 187 and "Referencing a Method That Handles a Value-Change Event" on page 188 explain how a page author uses the valueChangeListener and actionListener attributes to reference managed bean methods that handle events.

This section explains how to register a NameChanged value-change listener and a BookChange action listener implementation on components. The Duke's Bookstore case study includes both of these listeners.

**Registering a Value-Change Listener on a Component**

A page author can register a ValueChangeListener implementation on a component that implements EditableValueHolder by nesting an f:valueChangeListener tag within the component's tag on the page. The f:valueChangeListener tag supports the attributes shown in Table 8–3, one of which must be used.

**TABLE 8–3** Attributes for the f:valueChangeListener Tag

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>References the fully qualified class name of a ValueChangeListener implementation. Can accept a literal or a value expression.</td>
</tr>
<tr>
<td>binding</td>
<td>References an object that implements ValueChangeListener. Can accept only a value expression, which must point to a managed bean property that accepts and returns a ValueChangeListener implementation.</td>
</tr>
</tbody>
</table>

The following example shows a value-change listener registered on a component:

```xml
<h:inputText id="name"
    size="30"
    value="#{cashier.name}"
    required="true"
    requiredMessage="#{bundle.ReqCustomerName}" >
    <f:valueChangeListener
type="dukesbookstore.listeners.NameChanged" />
</h:inputText>
```

In the example, the core tag type attribute specifies the custom NameChanged listener as the javax.faces.event.ValueChangeListener implementation registered on the name component.

After this component tag is processed and local values have been validated, its corresponding component instance will queue the javax.faces.event.ValueChangeEvent associated with the specified ValueChangeListener to the component.
The binding attribute is used to bind a ValueChangeListener implementation to a managed bean property. This attribute works in a similar way to the binding attribute supported by the standard converter tags. See “Binding Component Values and Instances to Managed Bean Properties” on page 288 for more information.

Registering an Action Listener on a Component

A page author can register a javax.faces.event.ActionListener implementation on a command component by nesting an f:actionListener tag within the component’s tag on the page. Similarly to the f:valueChangeListener tag, the f:actionListener tag supports both the type and binding attributes. One of these attributes must be used to reference the action listener.

Here is an example of an h:commandLink tag that references an ActionListener implementation:

```xml
<h:commandLink id="Duke" action="bookstore">
    <f:actionListener type="dukesbookstore.listeners.LinkBookChangeListener" />
    <h:outputText value="#{bundle.Book201}"/>
</h:commandLink>
```

The type attribute of the f:actionListener tag specifies the fully qualified class name of the ActionListener implementation. Similarly to the f:valueChangeListener tag, the f:actionListener tag also supports the binding attribute. See “Binding Converters, Listeners, and Validators to Managed Bean Properties” on page 293 for more information about binding listeners to managed bean properties.

In addition to the actionListener tag that allows you register a custom listener onto a component, the core tag library includes the f:setPropertyActionListener tag. You use this tag to register a special action listener onto the ActionSource instance associated with a component. When the component is activated, the listener will store the object referenced by the tag’s value attribute into the object referenced by the tag’s target attribute.

The bookcatalog.xhtml page of the Duke’s Bookstore application uses f:setPropertyActionListener with two components: the h:commandLink component used to link to the bookdetails.xhtml page and the h:commandButton component used to add a book to the cart:

```xml
<h:dataTable id="books"
    value="#{bookRequestBean.books}"
    var="book"
    headerClass="list-header"
    styleClass="list-background"
    rowClasses="list-row-even, list-row-odd"
    border="1"
    summary="#{bundle.BookCatalog}" >
```
The `h:commandLink` and `h:commandButton` tags are within an `h:dataTable` tag, which iterates over the list of books. The `var` attribute refers to a single book in the list of books.

The object referenced by the `var` attribute of an `h:dataTable` tag is in page scope. However, in this case, you need to put this object into request scope so that when the user activates the `commandLink` component to go to `bookdetails.xhtml` or activates the `commandButton` component to go to `bookcatalog.xhtml`, the book data is available to those pages. Therefore, the `f:setPropertyActionListener` tag is used to set the current book object into request scope when the `commandLink` or `commandButton` component is activated.

In the preceding example, the `f:setPropertyActionListener` tag's `value` attribute references the book object. The `f:setPropertyActionListener` tag's `target` attribute references the value expression `requestScope.book`, which is where the book object referenced by the `value` attribute is stored when the `commandLink` or the `commandButton` component is activated.

### Using the Standard Validators

JavaServer Faces technology provides a set of standard classes and associated tags that page authors and application developers can use to validate a component's data. Table 8–4 lists all the standard `javax.faces.validator` classes and the tags that allow you to use the validators from the page.
TABLE 8–4  The Validator Classes

<table>
<thead>
<tr>
<th>Validator Class</th>
<th>Tag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeanValidator</td>
<td>validateBean</td>
<td>Registers a bean validator for the component.</td>
</tr>
<tr>
<td>DoubleRangeValidator</td>
<td>validateDoubleRange</td>
<td>Checks whether the local value of a component is within a certain range. The value must be floating-point or convertible to floating-point.</td>
</tr>
<tr>
<td>LengthValidator</td>
<td>validateLength</td>
<td>Checks whether the length of a component's local value is within a certain range. The value must be a java.lang.String.</td>
</tr>
<tr>
<td>LongRangeValidator</td>
<td>validateLongRange</td>
<td>Checks whether the local value of a component is within a certain range. The value must be any numeric type or String that can be converted to a long.</td>
</tr>
<tr>
<td>RegexValidator</td>
<td>validateRegEx</td>
<td>Checks whether the local value of a component is a match against a regular expression from the java.util.regex package.</td>
</tr>
<tr>
<td>RequiredValidator</td>
<td>validateRequired</td>
<td>Ensures that the local value is not empty on a javax.faces.componentEditableValueHolder component.</td>
</tr>
</tbody>
</table>

All these validator classes implement the javax.faces.validator.Validator interface. Component writers and application developers can also implement this interface to define their own set of constraints for a component's value.

Similar to the standard converters, each of these validators has one or more standard error messages associated with it. If you have registered one of these validators onto a component on your page, and the validator is unable to validate the component's value, the validator's error message will display on the page. For example, the error message that displays when the component's value exceeds the maximum value allowed by LongRangeValidator is as follows:

{(1): Validation Error: Value is greater than allowable maximum of "{0}"}

In this case, the {1} substitution parameter is replaced by the component's label or id, and the {0} substitution parameter is replaced with the maximum value allowed by the validator.

See "Displaying Error Messages with the h:message and h:messages Tags" on page 164 for information on how to display validation error messages on the page when validation fails.

Instead of using the standard validators, you can use Bean Validation to validate data. See "Using Bean Validation" on page 204 for more information.
Validating a Component’s Value

To validate a component’s value using a particular validator, you need to register that validator on the component. You can do this in one of the following ways:

- Nest the validator’s corresponding tag (shown in Table 8–4) inside the component’s tag. “Using LongRangeValidator” on page 185 explains how to use the validateLongRange tag. You can use the other standard tags in the same way.
- Refer to a method that performs the validation from the component tag’s validator attribute.
- Nest a validator tag inside the component tag, and use either the validator tag’s validatorId attribute or its binding attribute to refer to the validator.

See “Referencing a Method That Performs Validation” on page 187 for more information on using the validator attribute.

The validatorId attribute works similarly to the converterId attribute of the converter tag, as described in “Converting a Component’s Value” on page 176.

Keep in mind that validation can be performed only on components that implement EditableValueHolder, because these components accept values that can be validated.

Using LongRangeValidator

The following example shows how to use the validateLongRange validator on an input component named quantity:

```xml
<
<h:inputText id="quantity" size="4"
   value="#{item.quantity}"/>
   <f:validateLongRange minimum="1"/>
</h:inputText>
<h:message for="quantity"/>
```

This tag requires the user to enter a number that is at least 1. The validateLongRange tag also has a maximum attribute, which sets a maximum value for the input.

The attributes of all the standard validator tags accept EL value expressions. This means that the attributes can reference managed bean properties rather than specify literal values. For example, the validateLongRange tag in the preceding example can reference managed bean properties called minimum and maximum to get the minimum and maximum values acceptable to the validator implementation, as shown in this snippet from the guess number example:

```xml
<h:inputText
   id="userNo"
   title="Type a number from 0 to 10:"
   value="#{userNumberBean.userNumber}"
```
Referencing a Managed Bean Method

A component tag has a set of attributes for referencing managed bean methods that can perform certain functions for the component associated with the tag. These attributes are summarized in **Table 8–5**.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>Refers to a managed bean method that performs navigation processing for the component and returns a logical outcome String</td>
</tr>
<tr>
<td>actionListener</td>
<td>Refers to a managed bean method that handles action events</td>
</tr>
<tr>
<td>validator</td>
<td>Refers to a managed bean method that performs validation on the component’s value</td>
</tr>
<tr>
<td>valueChangeListener</td>
<td>Refers to a managed bean method that handles value-change events</td>
</tr>
</tbody>
</table>

Only components that implement `javax.faces.component.ActionSource` can use the `action` and `actionListener` attributes. Only components that implement `javax.faces.component.EditableValueHolder` can use the `validator` or `valueChangeListener` attributes.

The component tag refers to a managed bean method using a method expression as a value of one of the attributes. The method referenced by an attribute must follow a particular signature, which is defined by the tag attribute’s definition in the documentation at [http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/](http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/). For example, the definition of the `validator` attribute of the `inputText` tag is the following:

```java
void validate(javax.faces.context.FacesContext, javax.faces.component.UIComponent, java.lang.Object)
```

The following sections give examples of how to use the attributes.
Referencing a Method That Performs Navigation

If your page includes a component, such as a button or a hyperlink, that causes the application to navigate to another page when the component is activated, the tag corresponding to this component must include an action attribute. This attribute does one of the following:

- Specifies a logical outcome String that tells the application which page to access next
- References a managed bean method that performs some processing and returns a logical outcome String

The following example shows how to reference a navigation method:

```xml
<h:commandButton
    value="#{bundle.Submit}"
    action="#{cashier.submit}" />
```

See “Writing a Method to Handle Navigation” on page 201 for information on how to write such a method.

Referencing a Method That Handles an Action Event

If a component on your page generates an action event, and if that event is handled by a managed bean method, you refer to the method by using the component’s actionListener attribute.

The following example shows how such a method could be referenced:

```xml
<h:commandLink id="Duke" action="bookstore"
    actionListener="#{actionBean.chooseBookFromLink}"
    />
```

The actionListener attribute of this component tag references the chooseBookFromLink method using a method expression. The chooseBookFromLink method handles the event when the user clicks the hyperlink rendered by this component. See “Writing a Method to Handle an Action Event” on page 202 for information on how to write such a method.

Referencing a Method That Performs Validation

If the input of one of the components on your page is validated by a managed bean method, refer to the method from the component’s tag by using the validator attribute.

The following example from “The guessnumber CDI Example” on page 531 shows how to reference a method that performs validation on inputGuess, an input component:

```xml
<h:inputText id="inputGuess"
    value="#{userNumberBean.userNumber}"
    required="true" size="3"
```
The managed bean method validateNumberRange verifies that the input value is within the valid range, which changes each time another guess is made. See “Writing a Method to Perform Validation” on page 202 for information on how to write such a method.

**Referencing a Method That Handles a Value-Change Event**

If you want a component on your page to generate a value-change event and you want that event to be handled by a managed bean method instead of a javax.faces.event.ValueChangeListener implementation, you refer to the method by using the component’s valueChangeListener attribute:

```html
<h:inputText id="name" size="30"
  value="#{cashier.name}"
  required="true"
  valueChangeListener="#{cashier.processValueChange}" />
</h:inputText>
```

The valueChangeListener attribute of this component tag references the processValueChange method of CashierBean by using a method expression. The processValueChange method handles the event of a user entering a name in the input field rendered by this component.

“Writing a Method to Handle a Value-Change Event” on page 203 describes how to implement a method that handles a javax.faces.event.ValueChangeEvent.
This chapter provides an overview of managed beans and explains how to write methods and properties of managed beans that are used by a JavaServer Faces application. This chapter also introduces the Bean Validation feature.

Chapter 7, "Using JavaServer Faces Technology in Web Pages," and Chapter 8, "Using Converters, Listeners, and Validators," showed how to add components to a page and connect them to server-side objects by using component tags and core tags, as well as how to provide additional functionality to the components through converters, listeners, and validators. Developing a JavaServer Faces application also involves the task of programming the server-side objects: managed beans, converters, event handlers, and validators.

The following topics are addressed here:
- “Managed Beans in JavaServer Faces Technology” on page 189
- “Writing Bean Properties” on page 192
- “Writing Managed Bean Methods” on page 200
- “Using Bean Validation” on page 204

Managed Beans in JavaServer Faces Technology

A typical JavaServer Faces application includes one or more managed beans, each of which can be associated with the components used in a particular page. This section introduces the basic concepts of creating, configuring, and using managed beans in an application.
Creating a Managed Bean

A managed bean is created with a constructor with no arguments, a set of properties, and a set of methods that perform functions for a component. Each of the managed bean properties can be bound to one of the following:

- A component value
- A component instance
- A converter instance
- A listener instance
- A validator instance

The most common functions that managed bean methods perform include the following:

- Validating a component's data
- Handling an event fired by a component
- Performing processing to determine the next page to which the application must navigate

As with all JavaBeans components, a property consists of a private data field and a set of accessor methods, as shown by this code:

```java
private Integer userNumber = null;
...
public void setUserNumber(Integer user_number) {
    userNumber = user_number;
}
public Integer getUserNumber() {
    return userNumber;
}
```

When bound to a component's value, a bean property can be any of the basic primitive and numeric types or any Java object type for which the application has access to an appropriate converter. For example, a property can be of type `java.util.Date` if the application has access to a converter that can convert the `Date` type to a `String` and back again. See “Writing Bean Properties” on page 192 for information on which types are accepted by which component tags.

When a bean property is bound to a component instance, the property's type must be the same as the component object. For example, if a `javax.faces.component.UISelectBoolean` component is bound to the property, the property must accept and return a `UISelectBoolean` object. Likewise, if the property is bound to a converter, validator, or listener instance, the property must be of the appropriate converter, validator, or listener type.

For more information on writing beans and their properties, see “Writing Bean Properties” on page 192.
Using the EL to Reference Managed Beans

To bind component values and objects to managed bean properties or to reference managed bean methods from component tags, page authors use the Expression Language syntax. As explained in “Overview of the EL” on page 125, the following are some of the features that EL offers:

- Deferred evaluation of expressions
- The ability to use a value expression to both read and write data
- Method expressions

Deferred evaluation of expressions is important because the JavaServer Faces lifecycle is split into several phases in which component event handling, data conversion and validation, and data propagation to external objects are all performed in an orderly fashion. The implementation must be able to delay the evaluation of expressions until the proper phase of the lifecycle has been reached. Therefore, the implementation’s tag attributes always use deferred-evaluation syntax, which is distinguished by the `#{}` delimiter.

To store data in external objects, almost all JavaServer Faces tag attributes use Ivalue expressions, which are expressions that allow both getting and setting data on external objects.

Finally, some component tag attributes accept method expressions that reference methods that handle component events or validate or convert component data.

To illustrate a JavaServer Faces tag using the EL, the following tag references a method that validates user input:

```xml
<h:inputText id="inputGuess"
   value="#{userNumberBean.userNumber}"
   required="true" size="3"
   disabled="#{userNumberBean.number eq userNumberBean.userNumber}"
   validator="#{userNumberBean.validateNumberRange}">
</h:inputText>
```

This tag binds the inputGuess component’s value to the UserNumberBean.userNumber managed bean property by using an Ivalue expression. The tag uses a method expression to refer to the UserNumberBean.validateNumberRange method, which performs validation of the component’s local value. The local value is whatever the user types into the field corresponding to this tag. This method is invoked when the expression is evaluated.

Nearly all JavaServer Faces tag attributes accept value expressions. In addition to referencing bean properties, value expressions can reference lists, maps, arrays, implicit objects, and resource bundles.

Another use of value expressions is to bind a component instance to a managed bean property. A page author does this by referencing the property from the binding attribute:

```xml
<h:outputLabel for="fanClub"
   rendered="false"
   binding="#{cashier.specialOfferText}"
>`
<h:outputText id="fanClubLabel" value="#{bundle.DukeFanClub}"/>

In addition to using expressions with the standard component tags, you can configure your custom component properties to accept expressions by creating `javax.el.ValueExpression` or `javax.el.MethodExpression` instances for them.

For information on the EL, see Chapter 6, "Expression Language."

For information on referencing managed bean methods from component tags, see “Referencing a Managed Bean Method” on page 186.

### Writing Bean Properties

As explained in “Managed Beans in JavaServer Faces Technology” on page 189, a managed bean property can be bound to one of the following items:

- A component value
- A component instance
- A converter implementation
- A listener implementation
- A validator implementation

These properties follow the conventions of JavaBeans components (also called beans). For more information on JavaBeans components, see the [JavaBeans Tutorial](http://docs.oracle.com/javase/tutorial/javabeans/index.html).

The component's tag binds the component's value to a managed bean property by using its `value` attribute and binds the component's instance to a managed bean property by using its `binding` attribute. Likewise, all the converter, listener, and validator tags use their `binding` attributes to bind their associated implementations to managed bean properties. See “Binding Component Values and Instances to Managed Bean Properties” on page 288 and "Binding Converters, Listeners, and Validators to Managed Bean Properties" on page 293 for more information.

To bind a component's value to a managed bean property, the type of the property must match the type of the component's value to which it is bound. For example, if a managed bean property is bound to a `UISelectBoolean` component's value, the property should accept and return a boolean value or a `Boolean` wrapper `Object` instance.

To bind a component instance to a managed bean property, the property must match the type of component. For example, if a managed bean property is bound to a `UISelectBoolean` instance, the property should accept and return a `UISelectBoolean` value.

Similarly, to bind a converter, listener, or validator implementation to a managed bean property, the property must accept and return the same type of converter, listener, or validator.
object. For example, if you are using the convertDateTime tag to bind a
javax.faces.convert.DateTimeConverter to a property, that property must accept and
return a DateTimeConverter instance.

The rest of this section explains how to write properties that can be bound to component values,
to component instances for the component objects described in “Adding Components to a Page
Using HTML Tags” on page 140, and to converter, listener, and validator implementations.

**Writing Properties Bound to Component Values**

To write a managed bean property that is bound to a component’s value, you must match the
property type to the component’s value.

Table 9–1 lists the javax.faces.component classes and the acceptable types of their values.

<table>
<thead>
<tr>
<th>Component Class</th>
<th>Acceptable Types of Component Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIInput, UIOutput, UISelectItem, UISelectOne</td>
<td>Any of the basic primitive and numeric types or any Java programming language object type for which an appropriate javax.faces.convert.Converter implementation is available</td>
</tr>
<tr>
<td>UIData</td>
<td>array of beans, List of beans, single bean, java.sql.ResultSet, javax.servlet.jsp.jstl.sql.ResultSet, java.sql.ResultSet</td>
</tr>
<tr>
<td>UISelectBoolean</td>
<td>boolean or Boolean</td>
</tr>
<tr>
<td>UISelectItems</td>
<td>java.lang.String, Collection, Array, Map</td>
</tr>
<tr>
<td>UISelectMany</td>
<td>array or List, though elements of the array or List can be any of the standard types</td>
</tr>
</tbody>
</table>

When they bind components to properties by using the value attributes of the component tags,
page authors need to ensure that the corresponding properties match the types of the
components’ values.

**UIInput and UIOutput Properties**

The UIInput and UIOutput component classes are represented by the component tags that
begin with h:input and h:output, respectively (for example, h:inputText and h:outputText).

In the following example, an h:inputText tag binds the name component to the name property
of a managed bean called CashierBean.

```xml
<h:inputText id="name"
size="30"/>
The following code snippet from the managed bean CashierBean shows the bean property type bound by the preceding component tag:

```java
private String name = null;
public void setName(String name) {
    this.name = name;
}
public String getName() {
    return this.name;
}
```

As described in “Using the Standard Converters” on page 175, to convert the value of an input or output component, you can either apply a converter or create the bean property bound to the component with the matching type. Here is the example tag, from “Using DateTimeConverter” on page 177, that displays the date when items will be shipped.

```html
<h:outputText value="#{cashier.shipDate}">
    <f:convertDateTime type="date" dateStyle="full" /> 
</h:outputText>
```

The bean property represented by this tag must have a type of java.util.Date. The following code snippet shows the shipDate property, from the managed bean CashierBean, that is bound by the tag's value in the preceding example:

```java
private Date shipDate;
public Date getShipDate() {
    return this.shipDate;
}
public void setShipDate(Date shipDate) {
    this.shipDate = shipDate;
}
```

**UIData Properties**

The UIData component class is represented by the `h:dataTable` component tag.

UIData components must be bound to one of the managed bean property types listed in Table 9–1. Data components are discussed in "Using Data-Bound Table Components" on page 161. Here is part of the start tag of `dataTable` from that section:

```html
<h:dataTable id="items"
    value="#{cart.items}"
    var="item">
```

The value expression points to the items property of a shopping cart bean named cart. The cart bean maintains a map of ShoppingCartItem beans.
The `getItems` method from the `cart` bean populates a `List` with `ShoppingCartItem` instances that are saved in the `items` map when the customer adds books to the cart, as shown in the following code segment:

```java
public synchronized List<ShoppingCartItem> getItems() {
    List<ShoppingCartItem> results = new ArrayList<ShoppingCartItem>();
    results.addAll(this.items.values());
    return results;
}
```

All the components contained in the `UIData` component are bound to the properties of the `cart` bean that is bound to the entire `UIData` component. For example, here is the `h:outputText` tag that displays the book title in the table:

```xml
<h:commandLink action="#{showcart.details}"
    <h:outputText value="#{item.item.title}"
</h:commandLink>
```

The title is actually a hyperlink to the `bookdetails.xhtml` page. The `h:outputText` tag uses the value expression `#{item.item.title}` to bind its `UIOutput` component to the `title` property of the `Book` entity. The first item in the expression is the `ShoppingCartItem` instance that the `h: dataTable` tag is referencing while rendering the current row. The second item in expression refers to the `item` property of `ShoppingCartItem`, which returns an `Object` (in this case, a `Book`). The `title` part of the expression refers to the `title` property of `Book`. The value of the `UIOutput` component corresponding to this tag is bound to the `title` property of the `Book` entity:

```java
private String title;
...
public String getTitle() {
    return title;
}
public void setTitle(String title) {
    this.title = title;
}
```

**UISelectBoolean Properties**

The `UISelectBoolean` component class is represented by the component tag `h:selectBooleanCheckbox`.

Managed bean properties that hold a `UISelectBoolean` component's data must be of `boolean` or `Boolean` type. The example `selectBooleanCheckbox` tag from the section "Displaying Components for Selecting One Value" on page 156 binds a component to a property. The following example shows a tag that binds a component value to a boolean property:

```xml
<h:selectBooleanCheckbox title="#{bundle.receiveEmails}" value="#{custFormBean.receiveEmails}" />
</h:selectBooleanCheckbox>
<h:outputText value="#{bundle.receiveEmails}"/>
```
Here is an example property that can be bound to the component represented by the example tag:

```java
private boolean receiveEmails = false;
...
public void setReceiveEmails(boolean receiveEmails) {
    this.receiveEmails = receiveEmails;
}
public boolean getReceiveEmails() {
    return receiveEmails;
}
```

**UISelectMany Properties**

The UISelectMany component class is represented by the component tags that begin with `h:selectMany` (for example, `h:selectManyRadio` and `h:selectManyListbox`).

Because a UISelectMany component allows a user to select one or more items from a list of items, this component must map to a bean property of type `List` or `array`. This bean property represents the set of currently selected items from the list of available items.

The following example of the `selectManyCheckbox` tag comes from “Displaying Components for Selecting Multiple Values” on page 158:

```xml
<h:selectManyCheckbox id="newsletterCheckbox" layout="pageDirection" value="#{cashier.newsletters}">
    <f:selectItems value="#{cashier.newsletterItems}"/>
</h:selectManyCheckbox>
```

Here is the bean property that maps to the value of the `selectManyCheckbox` tag from the preceding example:

```java
private String newsletters[] = new String[0];
public void setNewsletters(String newsletters[]) {
    this.newsletters = newsletters;
}
public String[] getNewsletters() {
    return this.newsletters;
}
```

The UISelectItem and UISelectItems components are used to represent all the values in a UISelectMany component. See “UISelectItem Properties” on page 197 and “UISelectItems Properties” on page 198 for information on writing the bean properties for the UISelectItem and UISelectItems components.

**UISelectOne Properties**

The UISelectOne component class is represented by the component tags that begin with `h:selectOne` (for example, `h:selectOneRadio` and `h:selectOneListbox`).
UISelectOne properties accept the same types as UIInput and UIOutput properties, because a UISelectOne component represents the single selected item from a set of items. This item can be any of the primitive types and anything else for which you can apply a converter.

Here is an example of the h:selectOneMenu tag from “Displaying a Menu Using the h:selectOneMenu Tag” on page 157:

```xml
<h:selectOneMenu id="shippingOption"
  required="true"
  value="#{cashier.shippingOption}">
  <f:selectItem itemValue="2"
    itemLabel="#{bundle.QuickShip}"/>
  <f:selectItem itemValue="5"
    itemLabel="#{bundle.NormalShip}"/>
  <f:selectItem itemValue="7"
    itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

Here is the bean property corresponding to this tag:

```java
private String shippingOption = "2";
public void setShippingOption(String shippingOption) {
  this.shippingOption = shippingOption;
}
public String getShippingOption() {
  return this.shippingOption;
}
```

Note that shippingOption represents the currently selected item from the list of items in the UISelectOne component.

The UISelectItem and UISelectItems components are used to represent all the values in a UISelectOne component. This is explained in the section “Displaying a Menu Using the h:selectOneMenu Tag” on page 157.

For information on how to write the managed bean properties for the UISelectItem and UISelectItems components, see “UISelectItem Properties” on page 197 and “UISelectItems Properties” on page 198.

**UISelectItem Properties**

A UISelectItem component represents a single value in a set of values in a UISelectMany or a UISelectOne component. A UISelectItem component must be bound to a managed bean property of type javax.faces.model.SelectItem. A SelectItem object is composed of an Object representing the value, along with two String values representing the label and description of the UISelectItem object.

The example selectOneMenu tag from “UISelectOne Properties” on page 196 contains selectItem tags that set the values of the list of items in the page. Here is an example of a bean property that can set the values for this list in the bean:
Writing Bean Properties

```java
SelectItem itemOne = null;
SelectItem getItemOne()
    return itemOne;
}
void setItemOne(SelectItem item) {
    itemOne = item;
}

UISelectItems Properties

UISelectItems components are children of UISelectMany and UISelectOne components. Each UISelectItems component is composed of a set of either UISelectItem instances or any collection of objects, such as an array, a list, or even POJOs.

The following code snippet from CashierBean shows how to write the properties for selectItems tags containing SelectItem instances.

```java
private String[] newsletters = new String[0];
...
private static SelectItem[] newsletterItems = {
    new SelectItem("Duke's Quarterly"),
    new SelectItem("Innovator's Almanac"),
    new SelectItem("Duke's Diet and Exercise Journal"),
    new SelectItem("Random Ramblings")
};
...
public void setNewsletters(String[] newsletters) {
    this.newsletters = newsletters;
}

public String[] getNewsletters() {
    return this.newsletters;
}

public SelectItem[] getNewsletterItems() {
    return newsletterItems;
}
```

Here, the newsletters property represents the SelectItems object, while the newsletterItems property represents a static array of SelectItem objects. The SelectItem class has several constructors; in this example, the argument is an Object that represents both the value of the item and the label that appears in the UISelectMany component on the page.

Writing Properties Bound to Component Instances

A property bound to a component instance returns and accepts a component instance rather than a component value. The following components bind a component instance to a managed bean property:

```html
<h:selectBooleanCheckbox id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
```
The selectBooleanCheckbox tag renders a check box and binds the fanClub UISelectBoolean component to the specialOffer property of CashierBean. The outputLabel tag binds the value of the value attribute, which represents the check box's label, to the specialOfferText property of CashierBean. If the user orders more than $100 worth of books and clicks the Submit button, the submit method of CashierBean sets both components' rendered properties to true, causing the check box and label to display when the page is rerendered.

Because the components corresponding to the example tags are bound to the managed bean properties, these properties must match the components' types. This means that the specialOfferText property must be of type UIOutput, and the specialOffer property must be of type UISelectBoolean:

```java
UIOutput specialOfferText = null;
public UIOutput getSpecialOfferText() {
    return this.specialOfferText;
}
public void setSpecialOfferText(UIOutput specialOfferText) {
    this.specialOfferText = specialOfferText;
}

UISelectBoolean specialOffer = null;
public UISelectBoolean getSpecialOffer() {
    return this.specialOffer;
}
public void setSpecialOffer(UISelectBoolean specialOffer) {
    this.specialOffer = specialOffer;
}
```

For more general information on component binding, see “Managed Beans in JavaServer Faces Technology” on page 189.

For information on how to reference a managed bean method that performs navigation when a button is clicked, see “Referencing a Method That Performs Navigation” on page 187.

For more information on writing managed bean methods that handle navigation, see “Writing a Method to Handle Navigation” on page 201.

**Writing Properties Bound to Converters, Listeners, or Validators**

All the standard converter, listener, and validator tags included with JavaServer Faces technology support binding attributes that allow you to bind converter, listener, or validator implementations to managed bean properties.
The following example shows a standard `convertDateTime` tag using a value expression with its binding attribute to bind the `javax.faces.convert.DateTimeConverter` instance to the `convertDate` property of `LoginBean`:

```html
<h:inputText value="#{LoginBean.birthDate}"
     <f:convertDateTime binding="#{LoginBean.convertDate}" />
</h:inputText>
```

The `convertDate` property must therefore accept and return a `DateTimeConverter` object, as shown here:

```java
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
    ... return convertDate;
}
public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
```

Because the converter is bound to a managed bean property, the managed bean property can modify the attributes of the converter or add new functionality to it. In the case of the preceding example, the property sets the date pattern that the converter uses to parse the user's input into a `Date` object.

The managed bean properties that are bound to validator or listener implementations are written in the same way and have the same general purpose.

**Writing Managed Bean Methods**

Methods of a managed bean can perform several application-specific functions for components on the page. These functions include

- Performing processing associated with navigation
- Handling action events
- Performing validation on the component's value
- Handling value-change events

By using a managed bean to perform these functions, you eliminate the need to implement the `javax.faces.validator.Validator` interface to handle the validation or one of the listener interfaces to handle events. Also, by using a managed bean instead of a `Validator` implementation to perform validation, you eliminate the need to create a custom tag for the `Validator` implementation.

In general, it is good practice to include these methods in the same managed bean that defines the properties for the components referencing these methods. The reason for doing so is that the methods might need to access the component's data to determine how to handle the event or to perform the validation associated with the component.
The following sections explain how to write various types of managed bean methods.

**Writing a Method to Handle Navigation**

An *action method*, a managed bean method that handles navigation processing, must be a public method that takes no parameters and returns an *Object*, which is the logical outcome that the navigation system uses to determine the page to display next. This method is referenced using the component tag’s *action* attribute.

The following action method is from the managed bean *CashierBean*, which is invoked when a user clicks the Submit button on the page. If the user has ordered more than $100 worth of books, this method sets the *rendered* properties of the *fanClub* and *specialOffer* components to *true*, causing them to be displayed on the page the next time that page is rendered.

After setting the components’ *rendered* properties to *true*, this method returns the logical outcome *null*. This causes the JavaServer Faces implementation to rerender the page without creating a new view of the page, retaining the customer’s input. If this method were to return *purchase*, which is the logical outcome to use to advance to a payment page, the page would rerender without retaining the customer’s input. In this case, you want to rerender the page without clearing the data.

If the user does not purchase more than $100 worth of books, or if the *thankYou* component has already been rendered, the method returns *bookreceipt*. The JavaServer Faces implementation loads the *bookreceipt.xhtml* page after this method returns:

```java
public String submit() {
    ...
    if ((cart.getTotal() > 100.00) && !specialOffer.isRendered()) {
        specialOfferText.setRendered(true);
        specialOffer.setRendered(true);
        return null;
    } else if (specialOffer.isRendered() && !thankYou.isRendered()) {
        thankYou.setRendered(true);
        return null;
    } else {
        ...
        cart.clear();
        return "bookreceipt";
    }
}
```

Typically, an action method will return a *String* outcome, as shown in the previous example. Alternatively, you can define an *Enum* class that encapsulates all possible outcome strings and then make an action method return an *enum* constant, which represents a particular *String* outcome defined by the *Enum* class.

The following example uses an *Enum* class to encapsulate all logical outcomes:
public enum Navigation {
    main, accountHist, accountList, atm, atmAck, transferFunds,
    transferAck, error
}

When it returns an outcome, an action method uses the dot notation to reference the outcome from the Enum class:

```java
public Object submit(){
    ...
    return Navigation.accountHist;
}
```

The section “Referencing a Method That Performs Navigation” on page 187 explains how a component tag references this method. The section “Writing Properties Bound to Component Instances” on page 198 explains how to write the bean properties to which the components are bound.

## Writing a Method to Handle an Action Event

A managed bean method that handles an action event must be a public method that accepts an action event and returns void. This method is referenced using the component tag’s actionListener attribute. Only components that implement javax.faces.component.ActionSource can refer to this method.

In the following example, a method from a managed bean named ActionBean processes the event of a user clicking one of the hyperlinks on the page:

```java
public void chooseBookFromLink(ActionEvent event) {
    String current = event.getComponent().getId();
    FacesContext context = FacesContext.getCurrentInstance();
    String bookId = books.get(current);
    context.getExternalContext().getSessionMap().put("bookId", bookId);
}
```

This method gets the component that generated the event from the event object; then it gets the component’s ID, which is a code for the book. The method matches the code against a HashMap object that contains the book codes and corresponding book ID values. Finally, the method sets the book ID by using the selected value from the HashMap object.

“Referencing a Method That Handles an Action Event” on page 187 explains how a component tag references this method.

## Writing a Method to Perform Validation

Instead of implementing the javax.faces.validator.Validator interface to perform validation for a component, you can include a method in a managed bean to take care of validating input for the component. A managed bean method that performs validation must
accept a javax.faces.context.FacesContext, the component whose data must be validated, and the data to be validated, just as the validate method of the Validator interface does. A component refers to the managed bean method by using its validator attribute. Only values of UIInput components or values of components that extend UIInput can be validated.

Here is an example of a managed bean method that validates user input, from “The guessnumber CDI Example” on page 531:

```java
class GuessingGameController {
    public void validateNumberRange(FacesContext context, 
                                     UIComponent toValidate, 
                                     Object value) {
        if (remainingGuesses <= 0) {
            FacesMessage message = new FacesMessage("No guesses left!");
            context.addMessage(toValidate.getClientId(context), message);
            ((UIInput) toValidate).setValid(false);
            return;
        }
        int input = (Integer) value;
        if (input < minimum || input > maximum) {
            ((UIInput) toValidate).setValid(false);
            FacesMessage message = new FacesMessage("Invalid guess");
            context.addMessage(toValidate.getClientId(context), message);
        }
    }
}
```

The `validateNumberRange` method performs two different validations:

1. If the user has run out of guesses, the method sets the valid property of the UIInput component to false. Then it queues a message onto the FacesContext instance, associating the message with the component ID, and returns.
2. If the user has some remaining guesses, the method then retrieves the local value of the component. If the input value is outside the allowable range, the method again sets the valid property of the UIInput component to false, queues a different message on the FacesContext instance, and returns.

See “Referencing a Method That Performs Validation” on page 187 for information on how a component tag references this method.

## Writing a Method to Handle a Value-Change Event

A managed bean that handles a value-change event must use a public method that accepts a value-change event and returns void. This method is referenced using the component’s valueChangeListener attribute. This section explains how to write a managed bean method to replace the javax.faces.event.ValueChangeListener implementation.

The following example tag comes from “Registering a Value-ChangeListener on a Component” on page 181, where the h:inputText tag with the id of name has a ValueChangeListener instance registered on it. This ValueChangeListener instance handles
the event of entering a value in the field corresponding to the component. When the user enters a value, a value-change event is generated, and the processValueChange(ValueChangeEvent) method of the ValueChangeListener class is invoked:

```<h:inputText id="name" size="30" value="#{cashier.name}" required="true" requiredMessage="#{bundle.ReqCustomerName}"/>
<f:valueChangeListener type="dukesbookstore.listeners.NameChanged"/></h:inputText>
```

Instead of implementing ValueChangeListener, you can write a managed bean method to handle this event. To do this, you move the processValueChange(ValueChangeEvent) method from the ValueChangeListener class, called NameChanged, to your managed bean.

Here is the managed bean method that processes the event of entering a value in the name field on the page:

```java
public void processValueChange(ValueChangeEvent event) throws AbortProcessingException {
if (null != event.getNewValue()) {
FacesContext.getCurrentInstance().getExternalContext().getSessionMap().put("name", event.getNewValue());
}
}
```

To make this method handle the javax.faces.event.ValueChangeEvent generated by an input component, reference this method from the component tag's valueChangeListener attribute. See "Referencing a Method That Handles a Value-Change Event" on page 188 for more information.

Using Bean Validation

Validating input received from the user to maintain data integrity is an important part of application logic. Validation of data can take place at different layers in even the simplest of applications, as shown in "Developing a Simple Facelets Application" on page 113. The guessnumber example application validates the user input (in the h:inputText tag) for numerical data at the presentation layer and for a valid range of numbers at the business layer.

JavaBeans Validation (Bean Validation) is a new validation model available as part of Java EE 6 platform. The Bean Validation model is supported by constraints in the form of annotations placed on a field, method, or class of a JavaBeans component, such as a managed bean.

Constraints can be built in or user defined. User-defined constraints are called custom constraints. Several built-in constraints are available in the java.validation.constraints package. Table 9–2 lists all the built-in constraints.
### TABLE 9–2  Built-In Bean Validation Constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>@AssertFalse</td>
<td>The value of the field or property must be false.</td>
<td>@AssertFalse boolean isUnsupported;</td>
</tr>
<tr>
<td>@AssertTrue</td>
<td>The value of the field or property must be true.</td>
<td>@AssertTrue boolean isActive;</td>
</tr>
<tr>
<td>@DecimalMax</td>
<td>The value of the field or property must be a decimal value lower than or equal to the number in the value element.</td>
<td>@DecimalMax(&quot;30.00&quot;) BigDecimal discount;</td>
</tr>
<tr>
<td>@DecimalMin</td>
<td>The value of the field or property must be a decimal value greater than or equal to the number in the value element.</td>
<td>@DecimalMin(&quot;5.00&quot;) BigDecimal discount;</td>
</tr>
<tr>
<td>@Digits</td>
<td>The value of the field or property must be a number within a specified range. The integer element specifies the maximum integral digits for the number, and the fraction element specifies the maximum fractional digits for the number.</td>
<td>@Digits(integer=6, fraction=2) BigDecimal price;</td>
</tr>
<tr>
<td>@Future</td>
<td>The value of the field or property must be a date in the future.</td>
<td>@Future Date eventDate;</td>
</tr>
<tr>
<td>@Max</td>
<td>The value of the field or property must be an integer value lower than or equal to the number in the value element.</td>
<td>@Max(10) int quantity;</td>
</tr>
<tr>
<td>@Min</td>
<td>The value of the field or property must be an integer value greater than or equal to the number in the value element.</td>
<td>@Min(5) int quantity;</td>
</tr>
<tr>
<td>@NotNull</td>
<td>The value of the field or property must not be null.</td>
<td>@NotNull String username;</td>
</tr>
<tr>
<td>@Null</td>
<td>The value of the field or property must be null.</td>
<td>@Null String unusedString;</td>
</tr>
<tr>
<td>@Past</td>
<td>The value of the field or property must be a date in the past.</td>
<td>@Past Date birthday;</td>
</tr>
</tbody>
</table>
TABLE 9–2  Built-In Bean Validation Constraints  (Continued)

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| @Pattern   | The value of the field or property must match the regular expression defined in the regexp element. | @Pattern(regexp="\(\d{3}\)\d{3}-\d{4}\)"

String phoneNumber; |
| @Size      | The size of the field or property is evaluated and must match the specified boundaries. If the field or property is a String, the size of the string is evaluated. If the field or property is a Collection, the size of the Collection is evaluated. If the field or property is a Map, the size of the Map is evaluated. If the field or property is an array, the size of the array is evaluated. Use one of the optional max or min elements to specify the boundaries. | @Size(min=2, max=240)

String briefMessage; |

In the following example, a constraint is placed on a field using the built-in @NotNull constraint:

```java
public class Name {
    @NotNull
    private String firstname;

    @NotNull
    private String lastname;
}
```

You can also place more than one constraint on a single JavaBeans component object. For example, you can place an additional constraint for size of field on the firstname and the lastname fields:

```java
public class Name {
    @NotNull
    @Size(min=1, max=16)
    private String firstname;

    @NotNull
    @Size(min=1, max=16)
    private String lastname;
}
```

The following example shows a method with a user-defined constraint that checks for a predefined email address pattern such as a corporate email account:
@ValidEmail
public String getEmailAddress() {
    return emailAddress;
}

For a built-in constraint, a default implementation is available. A user-defined or custom constraint needs a validation implementation. In the above example, the @ValidEmail custom constraint needs an implementation class.

Any validation failures are gracefully handled and can be displayed by the h:messages tag.

Any managed bean that contains Bean Validation annotations automatically gets validation constraints placed on the fields on a JavaServer Faces application’s web pages.

See “Validating Persistent Fields and Properties” on page 583 for more information on using validation constraints.

Validating Null and Empty Strings

The Java programming language distinguishes between null and empty strings. An empty string is a string instance of zero length, whereas a null string has no value at all.

An empty string is represented as "". It is a character sequence of zero characters. A null string is represented by null. It can be described as the absence of a string instance.

Managed bean elements represented as a JavaServer Faces text component such as inputText are initialized with the value of the empty string by the JavaServer Faces implementation. Validating these strings can be an issue when user input for such fields is not required. Consider the following example, where the string testString is a bean variable that will be set using input typed by the user. In this case, the user input for the field is not required.

```java
if (testString == null) {
    doSomething();
} else {
    doAnotherThing();
}
```

By default, the doAnotherThing method is called even when the user enters no data, because the testString element has been initialized with the value of an empty string.

In order for the Bean Validation model to work as intended, you must set the context parameter javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL to true in the web deployment descriptor file, web.xml:

```xml
<context-param>
    <param-name>
        javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
    </param-name>
    <param-value>true</param-value>
</context-param>
```
This parameter enables the JavaServer Faces implementation to treat empty strings as null.

Suppose, on the other hand, that you have a `@NotNull` constraint on an element, meaning that input is required. In this case, an empty string will pass this validation constraint. However, if you set the context parameter `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` to true, the value of the managed bean attribute is passed to the Bean Validation runtime as a null value, causing the `@NotNull` constraint to fail.
Previous chapters have introduced JavaServer Faces technology and Facelets, the preferred presentation layer for the Java EE platform. This chapter describes the JavaServer Faces lifecycle in detail. Some of the complex JavaServer Faces applications use the well-defined lifecycle phases to customize application behavior.

The following chapters introduce additional advanced concepts in this area:

- **Chapter 11, “Using Ajax with JavaServer Faces Technology,”** introduces Ajax concepts and the use of Ajax in JavaServer Faces applications.
- **Chapter 12, “Composite Components: Advanced Topics and Example,”** introduces advanced features of composite components.
- **Chapter 13, “Creating Custom UI Components and Other Custom Objects,”** describes the process of creating new components, renderers, converters, listeners, and validators from scratch.
- **Chapter 14, “Configuring JavaServer Faces Applications,”** introduces the process of creating and deploying JavaServer Faces applications, the use of various configuration files, and the deployment structure.

The following topics are addressed here:

- “The Lifecycle of a JavaServer Faces Application” on page 210
- “Partial Processing and Partial Rendering” on page 216
- “The Lifecycle of a Facelets Application” on page 216
- “User Interface Component Model” on page 217
The Lifecycle of a JavaServer Faces Application

The lifecycle of an application refers to the various stages of processing of that application, from its initiation to its conclusion. All applications have lifecycles. During a web application lifecycle, common tasks such as the following are performed:

- Handling incoming requests
- Decoding parameters
- Modifying and saving state
- Rendering web pages to the browser

The JavaServer Faces web application framework manages lifecycle phases automatically for simple applications or allows you to manage them manually for more complex applications as required.

JavaServer Faces applications that use advanced features may require interaction with the lifecycle at certain phases. For example, Ajax applications use partial processing features of the lifecycle. A clearer understanding of the lifecycle phases is key to creating well-designed components.

A simplified view of the JavaServer faces lifecycle, consisting of the two main phases of a JavaServer Faces web application, is introduced in “The Lifecycle of the hello Application” on page 108. This section examines the JavaServer Faces lifecycle in more detail.

Overview of the JavaServer Faces Lifecycle

The lifecycle of a JavaServer Faces application begins when the client makes an HTTP request for a page and ends when the server responds with the page, translated to HTML.

The lifecycle can be divided into two main phases, execute and render. The execute phase is further divided into subphases to support the sophisticated component tree. This structure requires that component data be converted and validated, component events be handled, and component data be propagated to beans in an orderly fashion.

A JavaServer Faces page is represented by a tree of components, called a view. During the lifecycle, the JavaServer Faces implementation must build the view while considering the state saved from a previous submission of the page. When the client requests a page, the JavaServer Faces implementation performs several tasks, such as validating the data input of components in the view and converting input data to types specified on the server side.

The JavaServer Faces implementation performs all these tasks as a series of steps in the JavaServer Faces request-response lifecycle. Figure 10–1 illustrates these steps.
The Lifecycle of a JavaServer Faces Application

FIGURE 10–1  JavaServer Faces Standard Request-Response Lifecycle
The request-response lifecycle handles two kinds of requests: initial requests and postbacks. An initial request occurs when a user makes a request for a page for the first time. A postback request occurs when a user submits the form contained on a page that was previously loaded into the browser as a result of executing an initial request.

When the lifecycle handles an initial request, it executes only the Restore View and Render Response phases, because there is no user input or action to process. Conversely, when the lifecycle handles a postback, it executes all of the phases.

Usually, the first request for a JavaServer Faces page comes in from a client, as a result of clicking a link or button component on a JavaServer Faces page. To render a response that is another JavaServer Faces page, the application creates a new view and stores it in the javax.faces.context.FacesContext instance, which represents all of the information associated with processing an incoming request and creating a response. The application then acquires object references needed by the view and calls the FacesContext.renderResponse method, which forces immediate rendering of the view by skipping to the Render Response phase of the lifecycle, as is shown by the arrows labelled Render Response in the diagram.

Sometimes, an application might need to redirect to a different web application resource, such as a web service, or generate a response that does not contain JavaServer Faces components. In these situations, the developer must skip the Render Response phase by calling the FacesContext.responseComplete method. This situation is also shown in the diagram, with the arrows labelled Response Complete.

The most common situation is that a JavaServer Faces component submits a request for another JavaServer Faces page. In this case, the JavaServer Faces implementation handles the request and automatically goes through the phases in the lifecycle to perform any necessary conversions, validations, and model updates, and to generate the response.

There is one exception to the lifecycle described in this section. When a component’s immediate attribute is set to true, the validation, conversion, and events associated with these components are processed during the Apply Request Values phase rather than in a later phase.

The details of the lifecycle explained in the following sections are primarily intended for developers who need to know information such as when validations, conversions, and events are usually handled and ways to change how and when they are handled. For more information on each of the lifecycle phases, download the latest JavaServer Faces Specification documentation from http://jcp.org/en/jsr/detail?id=314.

The JavaServer Faces application lifecycle execute phase contains the following subphases:

- “Restore View Phase” on page 213
- “Apply Request Values Phase” on page 213
- “Process Validations Phase” on page 214
- “Update Model Values Phase” on page 214
- “Invoke Application Phase” on page 215
- “Render Response Phase” on page 215
Restore View Phase

When a request for a JavaServer Faces page is made, usually by an action such as when a link or a button component is clicked, the JavaServer Faces implementation begins the Restore View phase.

During this phase, the JavaServer Faces implementation builds the view of the page, wires event handlers and validators to components in the view, and saves the view in the FacesContext instance, which contains all the information needed to process a single request. All the application's components, event handlers, converters, and validators have access to the FacesContext instance.

If the request for the page is an initial request, the JavaServer Faces implementation creates an empty view during this phase and the lifecycle advances to the Render Response phase, during which the empty view is populated with the components referenced by the tags in the page.

If the request for the page is a postback, a view corresponding to this page already exists in the FacesContext instance. During this phase, the JavaServer Faces implementation restores the view by using the state information saved on the client or the server.

Apply Request Values Phase

After the component tree is restored during a postback request, each component in the tree extracts its new value from the request parameters by using its decode (processDecodes()) method. The value is then stored locally on each component.

If any decode methods or event listeners have called the renderResponse method on the current FacesContext instance, the JavaServer Faces implementation skips to the Render Response phase.

If any events have been queued during this phase, the JavaServer Faces implementation broadcasts the events to interested listeners.

If some components on the page have their immediate attributes (see “The immediate Attribute” on page 143) set to true, then the validations, conversions, and events associated with these components will be processed during this phase. If any conversion fails, an error message associated with the component is generated and queued on FacesContext. This message will be displayed during the Render Response phase, along with any validation errors resulting from the Process Validations phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete method.

At the end of this phase, the components are set to their new values, and messages and events have been queued.
If the current request is identified as a partial request, the partial context is retrieved from the FacesContext, and the partial processing method is applied.

**Process Validations Phase**

During this phase, the JavaServer Faces implementation processes all validators registered on the components in the tree, by using its validate (processValidators) method. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component. The JavaServer Faces implementation also completes conversions for input components that do not have the immediate attribute set to true.

If the local value is invalid, or if any conversion fails, the JavaServer Faces implementation adds an error message to the FacesContext instance, and the lifecycle advances directly to the Render Response phase so that the page is rendered again with the error messages displayed. If there were conversion errors from the Apply Request Values phase, the messages for these errors are also displayed.

If any validate methods or event listeners have called the renderResponse method on the current FacesContext, the JavaServer Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete method.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the FacesContext, and the partial processing method is applied.

**Update Model Values Phase**

After the JavaServer Faces implementation determines that the data is valid, it traverses the component tree and sets the corresponding server-side object properties to the components’ local values. The JavaServer Faces implementation updates only the bean properties pointed at by an input component’s value attribute. If the local data cannot be converted to the types specified by the bean properties, the lifecycle advances directly to the Render Response phase so that the page is re-rendered with errors displayed. This is similar to what happens with validation errors.

If any updateModels methods or any listeners have called the renderResponse method on the current FacesContext instance, the JavaServer Faces implementation skips to the Render Response phase.
At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the `FacesContext.responseComplete` method.

If any events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

## Invoke Application Phase

During this phase, the JavaServer Faces implementation handles any application-level events, such as submitting a form or linking to another page.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the `FacesContext.responseComplete` method.

If the view being processed was reconstructed from state information from a previous request and if a component has fired an event, these events are broadcast to interested listeners.

Finally, the JavaServer Faces implementation transfers control to the Render Response phase.

## Render Response Phase

During this phase, JavaServer Faces builds the view and delegates authority to the appropriate resource for rendering the pages.

If this is an initial request, the components that are represented on the page will be added to the component tree. If this is not an initial request, the components are already added to the tree, so they need not be added again.

If the request is a postback and errors were encountered during the Apply Request Values phase, Process Validations phase, or Update Model Values phase, the original page is rendered again during this phase. If the pages contain `h:message` or `h:messages` tags, any queued error messages are displayed on the page.

After the content of the view is rendered, the state of the response is saved so that subsequent requests can access it. The saved state is available to the Restore View phase.
Partial Processing and Partial Rendering

The JavaServer Faces lifecycle spans all of the execute and render processes of an application. It is also possible to process and render only parts of an application, such as a single component. For example, the JavaServer Faces Ajax framework can generate requests containing information on which particular component may be processed and which particular component may be rendered back to the client.

Once such a partial request enters the JavaServer Faces lifecycle, the information is identified and processed by a javax.faces.context.PartialViewContext object. The JavaServer Faces lifecycle is still aware of such Ajax requests and modifies the component tree accordingly.

The execute and render attributes of the <f:ajax> tag are used to identify which components may be executed and rendered. For more information on these attributes, see Chapter 11, "Using Ajax with JavaServer Faces Technology."

The Lifecycle of a Facelets Application

The JavaServer Faces specification defines the lifecycle of a JavaServer Faces application. For more information on this lifecycle, see “The Lifecycle of a JavaServer Faces Application” on page 210. The following steps describe that process as applied to a Facelets-based application.

1. When a client, such as a browser, makes a new request to a page that is created using Facelets, a new component tree or javax.faces.component.UIViewRoot is created and placed in the FacesContext.
2. The UIViewRoot is applied to the Facelets, and the view is populated with components for rendering.
3. The newly built view is rendered back as a response to the client.
4. On rendering, the state of this view is stored for the next request. The state of input components and form data is stored.
5. The client may interact with the view and request another view or change from the JavaServer Faces application. At this time the saved view is restored from the stored state.
6. The restored view is once again passed through the JavaServer Faces lifecycle, which eventually will either generate a new view or re-render the current view if there were no validation problems and no action was triggered.
7. If the same view is requested, the stored view is rendered once again.
8. If a new view is requested, then the process described in Step 2 is continued.
9. The new view is then rendered back as a response to the client.
User Interface Component Model

In addition to the lifecycle description, an overview of JavaServer Faces architecture provides better understanding of the technology.

JavaServer Faces components are the building blocks of a JavaServer Faces view. A component can be a user interface (UI) component or a non-UI component.

JavaServer Faces UI components are configurable, reusable elements that compose the user interfaces of JavaServer Faces applications. A component can be simple, such as a button, or can be compound, such as a table, composed of multiple components.

JavaServer Faces technology provides a rich, flexible component architecture that includes the following:

- A set of `javax.faces.component.UIComponent` classes for specifying the state and behavior of UI components
- A rendering model that defines how to render the components in various ways
- A conversion model that defines how to register data converters onto a component
- An event and listener model that defines how to handle component events
- A validation model that defines how to register validators onto a component
- A navigation model that defines page navigation and the sequence in which pages are loaded

This section briefly describes each of these pieces of the component architecture.

User Interface Component Classes

JavaServer Faces technology provides a set of UI component classes and associated behavioral interfaces that specify all the UI component functionality, such as holding component state, maintaining a reference to objects, and driving event handling and rendering for a set of standard components.

The component classes are completely extensible, allowing component writers to create their own custom components. See Chapter 13, “Creating Custom UI Components and Other Custom Objects,” for more information.

The abstract base class for all components is `javax.faces.component.UIComponent`. JavaServer Faces UI component classes extend the `UIComponentBase` class (a subclass of `UIComponent`), which defines the default state and behavior of a component. The following set of component classes is included with JavaServer Faces technology:

- `UIComponent`: Represents a single column of data in a `UIData` component.
- `UICommand`: Represents a control that fires actions when activated.
- `UIData`: Represents a data binding to a collection of data represented by a `javax.faces.model.DataModel` instance.
User Interface Component Model

- **UIForm**: Represents an input form to be presented to the user. Its child components represent (among other things) the input fields to be included when the form is submitted. This component is analogous to the `<form>` tag in HTML.
- **UIGraphic**: Displays an image.
- **UIInput**: Takes data input from a user. This class is a subclass of **UIOutput**.
- **UIMessage**: Displays a localized error message.
- **UIMessages**: Displays a set of localized error messages.
- **UIOutcomeTarget**: Displays a hyperlink in the form of a link or a button.
- **UIOutput**: Displays data output on a page.
- **UIPanel**: Manages the layout of its child components.
- **UIParameter**: Represents substitution parameters.
- **UISelectBoolean**: Allows a user to set a boolean value on a control by selecting or deselecting it. This class is a subclass of the **UIInput** class.
- **UISelectItem**: Represents a single item in a set of items.
- **UISelectItems**: Represents an entire set of items.
- **UISelectMany**: Allows a user to select multiple items from a group of items. This class is a subclass of the **UIInput** class.
- **UISelectOne**: Allows a user to select one item from a group of items. This class is a subclass of the **UIInput** class.
- **UIViewParameter**: Represents the query parameters in a request. This class is a subclass of the **UIInput** class.
- **UIViewRoot**: Represents the root of the component tree.

In addition to extending **UIComponentBase**, the component classes also implement one or more behavioral interfaces, each of which defines certain behavior for a set of components whose classes implement the interface.

These behavioral interfaces, all defined in the `javax.faces.component` package unless otherwise stated, are as follows:

- **ActionSource**: Indicates that the component can fire an action event. This interface is intended for use with components based on JavaServer Faces technology 1.1_01 and earlier versions. This interface is deprecated in JavaServer Faces 2.
- **ActionSource2**: Extends **ActionSource**, and therefore provides the same functionality. However, it allows components to use the Expression Language (EL) when they are referencing methods that handle action events.
- **EditableValueHolder**: Extends **ValueHolder** and specifies additional features for editable components, such as validation and emitting value-change events.
- **NamingContainer**: Mandates that each component rooted at this component have a unique ID.
- **StateHolder**: Denotes that a component has state that must be saved between requests.
- **ValueHolder**: Indicates that the component maintains a local value as well as the option of accessing data in the model tier.
- **javax.faces.event.SystemEventListenerHolder**: Maintains a list of javax.faces.event.SystemEventListener instances for each type of javax.faces.event.SystemEvent defined by that class.
- **javax.faces.component.behavior.ClientBehaviorHolder**: Adds the ability to attach javax.faces.component.behavior.ClientBehavior instances such as a reusable script.

UICommand implements ActionSource2 and StateHolder. UIOutput and component classes that extend UIOutput implement StateHolder and ValueHolder. UIInput and component classes that extend UIInput implement EditableValueHolder, StateHolder, and ValueHolder. UIComponentBase implements StateHolder.

Only component writers will need to use the component classes and behavioral interfaces directly. Page authors and application developers will use a standard component by including a tag that represents it on a page. Most of the components can be rendered in different ways on a page. For example, a UICommand component can be rendered as a button or a hyperlink.

The next section explains how the rendering model works and how page authors can choose to render the components by selecting the appropriate tags.

### Component Rendering Model

The JavaServer Faces component architecture is designed such that the functionality of the components is defined by the component classes, whereas the component rendering can be defined by a separate renderer class. This design has several benefits, including the following:

- Component writers can define the behavior of a component once but create multiple renderers, each of which defines a different way to render the component to the same client or to different clients.
- Page authors and application developers can change the appearance of a component on the page by selecting the tag that represents the appropriate combination of component and renderer.

A renderer kit defines how component classes map to component tags that are appropriate for a particular client. The JavaServer Faces implementation includes a standard HTML renderer kit for rendering to an HTML client.

The render kit defines a set of javax.faces.render.Renderer classes for each component that it supports. Each Renderer class defines a different way to render the particular component to the output defined by the render kit. For example, a SelectOne component has three different renderers. One of them renders the component as a set of radio buttons. Another renders the component as a combo box. The third one renders the component as a list box.
Similarly, a UICommand component can be rendered as a button or a hyperlink, using the h:commandButton or h:commandLink tag. The command part of each tag corresponds to the UICommand class, specifying the functionality, which is to fire an action. The Button or Link part of each tag corresponds to a separate Renderer class that defines how the component appears on the page.

Each custom tag defined in the standard HTML render kit is composed of the component functionality (defined in the UIComponent class) and the rendering attributes (defined by the Renderer class).

The section "Adding Components to a Page Using HTML Tags" on page 140 lists all supported component tags and illustrates how to use the tags in an example.

The JavaServer Faces implementation provides a custom tag library for rendering components in HTML.

**Conversion Model**

A JavaServer Faces application can optionally associate a component with server-side object data. This object is a JavaBeans component, such as a managed bean. An application gets and sets the object data for a component by calling the appropriate object properties for that component.

When a component is bound to an object, the application has two views of the component’s data:

- The model view, in which data is represented as data types, such as int or long.
- The presentation view, in which data is represented in a manner that can be read or modified by the user. For example, a java.util.Date might be represented as a text string in the format mm/dd/yy or as a set of three text strings.

The JavaServer Faces implementation automatically converts component data between these two views when the bean property associated with the component is of one of the types supported by the component’s data. For example, if a UISelectBoolean component is associated with a bean property of type java.lang.Boolean, the JavaServer Faces implementation will automatically convert the component’s data from String to Boolean. In addition, some component data must be bound to properties of a particular type. For example, a UISelectBoolean component must be bound to a property of type boolean or java.lang.Boolean.

Sometimes you might want to convert a component’s data to a type other than a standard type, or you might want to convert the format of the data. To facilitate this, JavaServer Faces technology allows you to register a javax.faces.converter.converter implementation on UIOutput components and components whose classes subclass UIOutput. If you register the Converter implementation on a component, the Converter implementation converts the component’s data between the two views.
You can either use the standard converters supplied with the JavaServer Faces implementation or create your own custom converter. Custom converter creation is covered in Chapter 13, “Creating Custom UI Components and Other Custom Objects.”

**Event and Listener Model**

The JavaServer Faces event and listener model is similar to the JavaBeans event model in that it has strongly typed event classes and listener interfaces that an application can use to handle events generated by components.

- **Application events** are tied to a particular application and are generated by a `UIComponent`. They represent the standard events available in previous versions of JavaServer Faces technology.

  An event object identifies the component that generated the event and stores information about the event. To be notified of an event, an application must provide an implementation of the listener class and must register it on the component that generates the event. When the user activates a component, such as by clicking a button, an event is fired. This causes the JavaServer Faces implementation to invoke the listener method that processes the event.

  JavaServer Faces supports two kinds of application events: action events and value-change events.

  - **An action event** (class `javax.faces.event.ActionEvent`) occurs when the user activates a component that implements `javax.faces.component.ActionSource`. These components include buttons and hyperlinks.

  - **A value-change event** (class `javax.faces.event.ValueChangeEvent`) occurs when the user changes the value of a component represented by `UIInput` or one of its subclasses. An example is selecting a check box, an action that results in the component’s value changing to `true`. The component types that can generate these types of events are the `UIInput`, `UISelectOne`, `UISelectMany`, and `UISelectBoolean` components. Value-change events are fired only if no validation errors are detected.

  Depending on the value of the **immediate property** (see “The immediate Attribute” on page 143) of the component emitting the event, action events can be processed during the invoke application phase or the apply request values phase, and value-change events can be processed during the process validations phase or the apply request values phase.

- **System events** are generated by an `Object` rather than a `UIComponent`. They are generated during the execution of an application at predefined times. They are applicable to the entire application rather than to a specific component.
A *data-model event* occurs when a new row of a `UIData` component is selected.

There are two ways to cause your application to react to action events or value-change events that are emitted by a standard component:

- Implement an event listener class to handle the event and register the listener on the component by nesting either an `f:valueChangeListener` tag or an `f:actionListener` tag inside the component tag.
- Implement a method of a managed bean to handle the event and refer to the method with a method expression from the appropriate attribute of the component's tag.

See "Implementing an Event Listener" on page 273 for information on how to implement an event listener. See "Registering Listeners on Components" on page 180 for information on how to register the listener on a component.

See "Writing a Method to Handle an Action Event" on page 202 and "Writing a Method to Handle a Value-Change Event" on page 203 for information on how to implement managed bean methods that handle these events.

See "Referencing a Managed Bean Method" on page 186 for information on how to refer to the managed bean method from the component tag.

When emitting events from custom components, you must implement the appropriate event class and manually queue the event on the component in addition to implementing an event listener class or a managed bean method that handles the event. "Handling Events for Custom Components" on page 275 explains how to do this.

**Validation Model**

JavaServer Faces technology supports a mechanism for validating the local data of editable components (such as text fields). This validation occurs before the corresponding model data is updated to match the local value.

Like the conversion model, the validation model defines a set of standard classes for performing common data validation checks. The JavaServer Faces core tag library also defines a set of tags that correspond to the standard `javax.faces.validator.Validator` implementations. See "Using the Standard Validators" on page 183 for a list of all the standard validation classes and corresponding tags.

Most of the tags have a set of attributes for configuring the validator's properties, such as the minimum and maximum allowable values for the component's data. The page author registers the validator on a component by nesting the validator's tag within the component's tag.

In addition to validators that are registered on the component, you can declare a default validator which is registered on all `UIInput` components in the application. For more information on default validators, see "Using Default Validators" on page 311.
The validation model also allows you to create your own custom validator and corresponding tag to perform custom validation. The validation model provides two ways to implement custom validation:

- Implement a `Validator` interface that performs the validation.
- Implement a managed bean method that performs the validation.

If you are implementing a `Validator` interface, you must also:

- Register the `Validator` implementation with the application.
- Create a custom tag or use an `f:validator` tag to register the validator on the component.

In the previously described standard validation model, the validator is defined for each input component on a page. The Bean Validation model allows the validator to be applied to all fields in a page. See “Using Bean Validation” on page 204 and Chapter 49, "Bean Validation: Advanced Topics," for more information on Bean Validation.

**Navigation Model**

The JavaServer Faces navigation model makes it easy to define page navigation and to handle any additional processing that is needed to choose the sequence in which pages are loaded.

In JavaServer Faces technology, `navigation` is a set of rules for choosing the next page or view to be displayed after an application action, such as when a button or hyperlink is clicked.

Navigation can be implicit or user-defined. Implicit navigation comes into play when user-defined navigation rules are not available. For more information on implicit navigation, see "Implicit Navigation Rules” on page 316.

User-defined navigation rules are declared in zero or more application configuration resource files, such as `faces-config.xml`, by using a set of XML elements. The default structure of a navigation rule is as follows:

```xml
<navigation-rule>
  <description></description>
  <from-view-id></from-view-id>
  <navigation-case>
    <from-action></from-action>
    <from-outcome></from-outcome>
    <if></if>
    <to-view-id></to-view-id>
  </navigation-case>
</navigation-rule>
```
User-defined navigation is handled as follows:

- Define the rules in the application configuration resource file.
- Refer to an outcome String from the button or hyperlink component's action attribute. This outcome String is used by the JavaServer Faces implementation to select the navigation rule.

Here is an example navigation rule:

```xml
<navigation-rule>
  <from-view-id>/greeting.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/response.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

This rule states that when a command component (such as an h:commandButton or an h:commandLink) on greeting.xhtml is activated, the application will navigate from the greeting.xhtml page to the response.xhtml page if the outcome referenced by the button component's tag is success. Here is the h:commandButton tag from greeting.xhtml that specifies a logical outcome of success:

```xml
<h:commandButton id="submit" action="success"
  value="Submit" />
```

As the example demonstrates, each navigation-rule element defines how to get from one page (specified in the from-view-id element) to the other pages of the application. The navigation-rule elements can contain any number of navigation-case elements, each of which defines the page to open next (defined by to-view-id) based on a logical outcome (defined by from-outcome).

In more complicated applications, the logical outcome can also come from the return value of an action method in a managed bean. This method performs some processing to determine the outcome. For example, the method can check whether the password the user entered on the page matches the one on file. If it does, the method might return success; otherwise, it might return failure. An outcome of failure might result in the logon page being reloaded. An outcome of success might cause the page displaying the user's credit card activity to open. If you want the outcome to be returned by a method on a bean, you must refer to the method using a method expression, with the action attribute, as shown by this example:

```xml
<h:commandButton id="submit"
  action="#{userNumberBean.getOrderStatus}" value="Submit" />
```

When the user clicks the button represented by this tag, the corresponding component generates an action event. This event is handled by the default javax.faces.event.ActionListener instance, which calls the action method referenced by the component that triggered the event. The action method returns a logical outcome to the action listener.
The listener passes the logical outcome and a reference to the action method that produced the outcome to the default `javax.faces.application.NavigationHandler`. The `NavigationHandler` selects the page to display next by matching the outcome or the action method reference against the navigation rules in the application configuration resource file by the following process:

1. The `NavigationHandler` selects the navigation rule that matches the page currently displayed.
2. It matches the outcome or the action method reference that it received from the default `javax.faces.event.ActionListener` with those defined by the navigation cases.
3. It tries to match both the method reference and the outcome against the same navigation case.
4. If the previous step fails, the navigation handler attempts to match the outcome.
5. Finally, the navigation handler attempts to match the action method reference if the previous two attempts failed.
6. If no navigation case is matched, it displays the same view again.

When the `NavigationHandler` achieves a match, the render response phase begins. During this phase, the page selected by the `NavigationHandler` will be rendered.

The Duke's Tutoring case study example application uses navigation rules in the business methods that handle creating, editing, and deleting the users of the application. For example, the form for creating a student has the following `h:commandButton` tag:

```html
<h:commandButton id="submit"
    action="#{adminBean.createStudent(studentManager.newStudent)}"
    value="#{bundle['action.submit']}"/>
```

The action event calls the `dukestutoring.ejb.AdminBean.createStudent` method:

```java
public String createStudent(Student student) {
    em.persist(student);
    return "createdStudent";
}
```

The return value of `createdStudent` has a corresponding navigation case in the `faces-config.xml` configuration file:

```xml
<navigation-rule>
    <from-view-id>/admin/student/createStudent.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>createdStudent</from-outcome>
        <to-view-id>/admin/index.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

After the student is created, the user is returned to the Administration index page.
For more information on how to define navigation rules, see “Configuring Navigation Rules” on page 313.

For more information on how to implement action methods to handle navigation, see “Writing a Method to Handle an Action Event” on page 202.

For more information on how to reference outcomes or action methods from component tags, see “Referencing a Method That Performs Navigation” on page 187.
Using Ajax with JavaServer Faces Technology

Ajax is an acronym for Asynchronous JavaScript and XML, a group of web technologies that enable creation of dynamic and highly responsive web applications. Using Ajax, web applications can retrieve content from the server without interfering with the display on the client. In the Java EE 6 platform, JavaServer Faces provides built-in support for Ajax.

Early web applications were created mostly as static web pages. When a static web page is updated by a client, the entire page has to reload to reflect the update. In effect, every update needs a page reload to reflect the change. Repetitive page reloads can result in excessive network access and can impact application performance. Technologies such as Ajax were created to overcome these deficiencies.

This chapter describes using Ajax functionality in JavaServer Faces web applications.

The following topics are addressed here:

- "Overview of Ajax" on page 228
- "Using Ajax Functionality with JavaServer Faces Technology" on page 228
- "Using Ajax with Facelets" on page 229
- "Sending an Ajax Request" on page 231
- "Monitoring Events on the Client" on page 233
- "Handling Errors" on page 234
- "Receiving an Ajax Response" on page 234
- "Ajax Request Lifecycle" on page 235
- "Grouping of Components" on page 236
- "Loading JavaScript as a Resource" on page 236
- "The AjaxguessNumber Example Application" on page 238
- "Further Information about Ajax in JavaServer Faces Technology" on page 242
Overview of Ajax

Ajax refers to JavaScript and XML, technologies that are widely used for creating dynamic and asynchronous web content. While Ajax is not limited to JavaScript and XML technologies, more often than not they are used together by web applications. The focus of this tutorial is on using JavaScript based Ajax functionality in JavaServer Faces web applications.

JavaScript is a dynamic scripting language for web applications. It allows users to add enhanced functionality to user interfaces and allows web pages to interact with clients asynchronously. JavaScript runs mainly on the client side (as in a browser) and thereby reduces server access by clients.

When a JavaScript function sends an asynchronous request from the client to the server, the server sends back a response that is used to update the page’s Document Object Model (DOM). This response is often in the format of an XML document. The term Ajax refers to this interaction between the client and server.

The server response need not be in XML only; it can also be in other formats, such as JSON. This tutorial does not focus on the response formats.

Ajax enables asynchronous and partial updating of web applications. Such functionality allows for highly responsive web pages that are rendered in near real time. Ajax-based web applications can access server and process information and can also retrieve data without interfering with the display and rendering of the current web page on a client (such as a browser).

Some of the advantages of using Ajax are as follows:

■ Form data validation in real time, eliminating the need to submit the form for verification
■ Enhanced functionality for web pages, such as user name and password prompts
■ Partial update of the web content, avoiding complete page reloads

Using Ajax Functionality with JavaServer Faces Technology

Ajax functionality can be added to a JavaServer Faces application in one of the following ways:

■ Adding the required JavaScript code to an application
■ Using the built-in Ajax resource library

In earlier releases of the Java EE platform, JavaServer Faces applications provided Ajax functionality by adding the necessary JavaScript to the web page. In the Java EE 6 platform, standard Ajax support is provided by a built-in JavaScript resource library.

With the support of this JavaScript resource library, JavaServer Faces standard UI components, such as buttons, labels, or text fields, can be enabled for Ajax functionality. You can also load
this resource library and use its methods directly from within the managed bean code. The next sections of the tutorial describe the use of the built-in Ajax resource library.

In addition, because the JavaServer Faces technology component model can be extended, custom components can be created with Ajax functionality.

An Ajax version of the guessnumber application, ajaxguessnumber, is available in the example repository. See “The ajaxguessnumber Example Application” on page 238 for more information.

The Ajax specific f:ajax tag and its attributes are explained in the next sections.

Using Ajax with Facelets

As mentioned in the previous section, JavaServer Faces technology supports Ajax by using a built-in JavaScript resource library that is provided as part of the JavaServer Faces core libraries. This built-in Ajax resource can be used in JavaServer Faces web applications in one of the following ways:

- By using the f:ajax tag along with another standard component in a Facelets application. This method adds Ajax functionality to any UI component without additional coding and configuration.
- By using the JavaScript API method jsf.ajax.request() directly within the Facelets application. This method provides direct access to Ajax methods, and allows customized control of component behavior.

Using the f:ajax Tag

The f:ajax tag is a JavaServer Faces core tag that provides Ajax functionality to any regular UI component when used in conjunction with that component. In the following example, Ajax behavior is added to an input component by including the f:ajax core tag:

```xml
<h:inputText value="#{bean.message}"
   f:ajax />
</h:inputText>
```

In this example, although Ajax is enabled, the other attributes of the f:ajax tag are not defined. If an event is not defined, the default action for the component is performed. For the inputText component, when no event attribute is specified, the default event is valueChange. Table 11–1 lists the attributes of the f:ajax tag and their default actions.
### Attributes of the f:ajax Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>javax.el.ValueExpression that evaluates to a Boolean</td>
<td>A Boolean value that identifies the tag status. A value of <code>true</code> indicates that the Ajax behavior should not be rendered. A value of <code>false</code> indicates that the Ajax behavior should be rendered. The default value is <code>false</code>.</td>
</tr>
<tr>
<td>event</td>
<td>javax.el.ValueExpression that evaluates to a String</td>
<td>A String that identifies the type of event to which the Ajax action will apply. If specified, it must be one of the events supported by the component. If not specified, the default event (the event that triggers the Ajax request) is determined for the component. The default event is <code>action</code> for <code>javax.faces.component.ActionSource</code> components and <code>valueChange</code> for <code>javax.faces.component.EditableValueHolder</code> components.</td>
</tr>
<tr>
<td>execute</td>
<td>javax.el.ValueExpression that evaluates to an Object</td>
<td>A Collection that identifies a list of components to be executed on the server. If a literal is specified, it must be a space-delimited String of component identifiers and/or one of the keywords. If a ValueExpression is specified, it must refer to a property that returns a Collection of String objects. If not specified, the default value is <code>@this</code>.</td>
</tr>
<tr>
<td>immediate</td>
<td>javax.el.ValueExpression that evaluates to a Boolean</td>
<td>A Boolean value that indicates whether inputs are to be processed early in the lifecycle. If <code>true</code>, behavior events generated from this behavior are broadcast during the Apply Request Values phase. Otherwise, the events will be broadcast during the Invoke Applications phase.</td>
</tr>
<tr>
<td>listener</td>
<td>javax.el.MethodExpression</td>
<td>The name of the listener method that is called when a <code>javax.faces.event.AjaxBehaviorEvent</code> has been broadcast for the listener.</td>
</tr>
<tr>
<td>onevent</td>
<td>javax.el.ValueExpression that evaluates to a String</td>
<td>The name of the JavaScript function that handles UI events.</td>
</tr>
<tr>
<td>onerror</td>
<td>javax.el.ValueExpression that evaluates to a String</td>
<td>The name of the JavaScript function that handles errors.</td>
</tr>
<tr>
<td>render</td>
<td>javax.el.ValueExpression that evaluates to an Object</td>
<td>A Collection that identifies a list of components to be rendered on the client. If a literal is specified, it must be a space-delimited String of component identifiers and/or one of the keywords. If a ValueExpression is specified, it must refer to a property that returns a Collection of String objects. If not specified, the default value is <code>@none</code>.</td>
</tr>
</tbody>
</table>
The keywords listed in Table 11–2 can be used with the `execute` and `render` attributes of the `f:ajax` tag.

**TABLE 11–2  Execute and Render Keywords**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@all</td>
<td>All component identifiers</td>
</tr>
<tr>
<td>@form</td>
<td>The form that encloses the component</td>
</tr>
<tr>
<td>@none</td>
<td>No component identifiers</td>
</tr>
<tr>
<td>@this</td>
<td>The element that triggered the request</td>
</tr>
</tbody>
</table>

Note that when you use the `f:ajax` tag in a Facelets page, the JavaScript resource library is loaded implicitly. This resource library can also be loaded explicitly as described in “Loading JavaScript as a Resource” on page 236.

## Sending an Ajax Request

To activate Ajax functionality, the web application must create an Ajax request and send it to the server. The server then processes the request.

The application uses the attributes of the `f:ajax` tag listed in Table 11–1 to create the Ajax request. The following sections explain the process of creating and sending an Ajax request using some of these attributes.

**Note** – Behind the scenes, the `jsf.ajax.request()` method of the JavaScript resource library collects the data provided by the `f:ajax` tag and posts the request to the JavaServer Faces lifecycle.

## Using the event Attribute

The event attribute defines the event that triggers the Ajax action. Some of the possible values for this attribute are `click`, `keyup`, `mouseover`, `focus`, and `blur`.

If not specified, a default event based on the parent component will be applied. The default event is `action` for `javax.faces.component.ActionSource` components such as a `commandButton`, and `valueChange` for `javax.faces.component.EditableValueHolder` components such as `inputText`. In the following example, an Ajax tag is associated with the button component, and the event that triggers the Ajax action is a mouse click:
Sending an Ajax Request

```xml
<h:commandButton id="submit" value="Submit">
  <f:ajax event="click"/>
</h:commandButton>
<h:outputText id="result" value="#{userNumberBean.response}"/>

Note – You may have noticed that the listed events are very similar to JavaScript events. In fact, they are based on JavaScript events, but do not have the on prefix.

For a command button, the default event is click, so that you do not actually need to specify event="click" to obtain the desired behavior.

Using the execute Attribute

The execute attribute defines the component or components to be executed on the server. The component is identified by its id attribute. You can specify more than one executable component. If more than one component is to be executed, specify a space-delimited list of components.

When a component is executed, it participates in all phases of the request processing lifecycle except the Render Response phase.

The execute attribute can also be a keyword, such as @all, @none, @this, or @form. The default value is @this, which refers to the component within which the f:ajax tag is nested.

The following code specifies that the h:inputText component with the id value of userNo should be executed when the button is clicked:

```xml
<h:inputText id="userNo"
  title="Type a number from 0 to 10:"
  value="#{userNumberBean.userNumber}"
/>
...
</h:inputText>
<h:commandButton id="submit" value="Submit">
  <f:ajax event="click" execute="userNo"/>
</h:commandButton>

Using the immediate Attribute

The immediate attribute indicates whether user inputs are to be processed early in the application lifecycle or later. If the attribute is set to true, events generated from this component are broadcast during the Apply Request Values phase. Otherwise, the events will be broadcast during the Invoke Applications phase.

If not defined, the default value of this attribute is false.
Using the **listener** Attribute

The **listener** attribute refers to a method expression that is executed on the server side in response to an Ajax action on the client. The listener's `javax.faces.event.AjaxBehaviorListener.processAjaxBehavior` method is called once during the Invoke Application phase of the lifecycle. In the following example, a **listener** attribute is defined by an `f:ajax` tag, which refers to a method from the bean.

```xml
<f:ajax listener="#{mybean.someaction}" render="somecomponent" />
```

The following code represents the **someaction** method in **mybean**.

```java
public void someaction(AjaxBehaviorEvent event) {
    dosomething;
}
```

Monitoring Events on the Client

The ongoing Ajax requests can be monitored by using the **onevent** attribute of the `f:ajax` tag. The value of this attribute is the name of a JavaScript function. JavaServer Faces calls the **onevent** function at each stage of the processing of an Ajax request: begin, complete, and success.

When calling the JavaScript function assigned to the **onevent** property, JavaServer Faces passes a data object to it. The data object contains the properties listed in Table 11–3.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>responseXML</td>
<td>The response to the Ajax call in XML format</td>
</tr>
<tr>
<td>responseText</td>
<td>The response to the Ajax call in text format</td>
</tr>
<tr>
<td>responseCode</td>
<td>The response to the Ajax call in numeric code</td>
</tr>
<tr>
<td>source</td>
<td>The source of the current Ajax event: the DOM element</td>
</tr>
<tr>
<td>status</td>
<td>The status of the current Ajax call: begin, complete, or success</td>
</tr>
<tr>
<td>type</td>
<td>The type of the Ajax call: event</td>
</tr>
</tbody>
</table>

By using the **status** property of the data object, you can identify the current status of the Ajax request and monitor its progress. In the following example, `monitormyajaxevent` is a JavaScript function that monitors the Ajax request sent by the event:

```xml
<f:ajax event="click" render="errormessage" onevent="monitormyajaxevent"/>
```
Handling Errors

JavaServer Faces handles Ajax errors through use of the `onerror` attribute of the `f:ajax` tag. The value of this attribute is the name of a JavaScript function.

When there is an error in processing a Ajax request, JavaServer Faces calls the defined `onerror` JavaScript function and passes a data object to it. The data object contains all the properties available for the `onevent` attribute, and in addition, the following properties:

- `description`
- `errorName`
- `errorMessage`

The `type` is `error`. The `status` property of the data object contains one of the valid error values listed in Table 11–4.

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>emptyResponse</td>
<td>No Ajax response from server.</td>
</tr>
<tr>
<td>httpError</td>
<td>One of the valid HTTP errors: <code>request.status==null</code> or <code>request.status==undefined</code> or <code>request.status &lt; 200</code> or <code>request.status &gt;= 300</code></td>
</tr>
<tr>
<td>malformedXML</td>
<td>The Ajax response is not well formed.</td>
</tr>
<tr>
<td>serverError</td>
<td>The Ajax response contains an <code>error</code> element.</td>
</tr>
</tbody>
</table>

In the following example, any errors that occurred in processing the Ajax request are handled by the `handlemyajaxerror` JavaScript function:

```xml
<f:ajax event="click" render="test" onerror="handlemyajaxerror"/>
```

Receiving an Ajax Response

After the application sends an Ajax request, it is processed on the server side, and a response is sent back to the client. As described earlier, Ajax allows for partial updating of web pages. To enable such partial updating, JavaServer Faces technology allows for partial processing of the view. The handling of the response is defined by the `render` attribute of the `f:ajax` tag.

Similar to the `execute` attribute, the `render` attribute defines which sections of the page will be updated. The value of a `render` attribute can be one or more component id values, one of the keywords `@this`, `@all`, `@none`, and `@form`, or an EL expression. In the following example, the `render` attribute simply identifies an output component to be displayed when the Ajax action has successfully completed.
However, more often than not, the render attribute is likely to be associated with an event attribute. In the following example, an output component is displayed when the button component is clicked.

```
<h:commandButton id="submit" value="Submit">
    <f:ajax execute="userNo" render="result" />
</h:commandButton>
<h:outputText id="result" value="#{userNumberBean.response}" />
```

Note – Behind the scenes, once again the jsf.ajax.request() method handles the response. It registers a response-handling callback when the original request is created. When the response is sent back to the client, the callback is invoked. This callback automatically updates the client-side DOM to reflect the rendered response.

Ajax Request Lifecycle

An Ajax request varies from other typical JavaServer Faces requests, and its processing is also handled differently by the JavaServer Faces lifecycle.

As described in “Partial Processing and Partial Rendering” on page 216, when an Ajax request is received, the state associated with that request is captured by the javax.faces.context PartialViewContext. This object provides access to information such as which components are targeted for processing/rendering. The processPartial method of PartialViewContext uses this information to perform partial component tree processing and rendering.

The execute attribute of the f:ajax tag identifies which segments of the server side component tree should be processed. Because components can be uniquely identified in the JavaServer Faces component tree, it is easy to identify and process a single component, a few components, or a whole tree. This is made possible by the visitTree method of the javax.faces.component.UICOMPONENT class. The identified components then run through the JavaServer Faces request lifecycle phases.

Similar to the execute attribute, the render attribute identifies which segments of the JavaServer Faces component tree need to be rendered during the render response phase.

During the render response phase, the render attribute is examined. The identified components are found and asked to render themselves and their children. The components are then packaged up and sent back to the client as a response.
Grouping of Components

The previous sections describe how to associate a single UI component with Ajax functionality. You can also associate Ajax with more than one component at a time by grouping them together on a page. The following example shows how a number of components can be grouped by using the \texttt{f:ajax} tag.

\begin{verbatim}
<f:ajax>
  <h:form>
    <h:inputText id="input1"/>
    <h:commandButton id="Submit"/>
  </h:form>
</f:ajax>
\end{verbatim}

In the example, neither component is associated with any Ajax event or render attributes yet. Therefore, no action will take place in case of user input. You can associate the above components with an event and a render attribute as follows:

\begin{verbatim}
<f:ajax event="click" render="@all">
  <h:form>
    <h:inputText id="input1" value="#{user.name}"/>
    <h:commandButton id="Submit"/>
  </h:form>
</f:ajax>
\end{verbatim}

In the updated example, when the user clicks either component, the updated results will be displayed for all components. You can further fine tune the Ajax action by adding specific events to each of the components, in which case Ajax functionality becomes cumulative. Consider the following example:

\begin{verbatim}
<f:ajax event="click" render="@all">
  ...
  <h:commandButton id="Submit">
    <f:ajax event="mouseover"/>
  </h:commandButton>
  ...
</f:ajax>
\end{verbatim}

Now the button component will fire an Ajax action in case of a mouseover event as well as a mouse click event.

Loading JavaScript as a Resource

The JavaScript resource file bundled with JavaServer Faces technology is named \texttt{jsf.js} and is available in the \texttt{javax.faces} library. This resource library supports Ajax functionality in JavaServer Faces applications.
In order to use this resource directly with a component or a bean class, you need to explicitly load the resource library. The resource can be loaded in one of the following ways:

- By using the resource API directly in a Facelets page
- By using the `javax.faces.application.ResourceDependency` annotation and the resource API in a bean class

### Using JavaScript API in a Facelets Application

To use the bundled JavaScript resource API directly in a web application, such as a Facelets page, you need to first identify the default JavaScript resource for the page with the help of the `h:outputScript` tag. For example, consider the following section of a Facelets page:

```xml
<h:form>
    <h:outputScript name="jsf.js" library="javax.faces" target="head"/>
</h:form>
```

Specifying the target as `head` causes the script resource to be rendered within the `head` element on the HTML page.

In the next step, identify the component to which you would like to attach the Ajax functionality. Add the Ajax functionality to the component by using the JavaScript API. For example, consider the following:

```xml
<h:form>
    <h:outputScript name="jsf.js" library="javax.faces" target="head">
        <h:inputText id="inputname" value="#{userBean.name}"/>
        <h:outputText id="outputname" value="#{userBean.name}"/>
        <h:commandButton id="submit" value="Submit"
        onclick="jsf.ajax.request(this, event,
            {execute:'inputname',render:'outputname'});
            return false;"/>
    </h:outputScript>
</h:form>
```

The `jsf.ajax.request` method takes up to three parameters that specify source, event, and options. The source parameter identifies the DOM element that triggered the Ajax request, typically `this`. The optional event parameter identifies the DOM event that triggered this request. The optional options parameter contains a set of name/value pairs from Table 11–5.

### Possible Values for the Options Parameter

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute</td>
<td>A space-delimited list of client identifiers or one of the keywords listed in Table 11–2. The identifiers reference the components that will be processed during the execute phase of the lifecycle.</td>
</tr>
</tbody>
</table>
Possible Values for the Options Parameter  

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>render</td>
<td>A space-delimited list of client identifiers or one of the keywords listed in Table 11-2. The identifiers reference the components that will be processed during the render phase of the lifecycle.</td>
</tr>
<tr>
<td>onevent</td>
<td>A String that is the name of the JavaScript function to call when an event occurs.</td>
</tr>
<tr>
<td>onerror</td>
<td>A String that is the name of the JavaScript function to call when an error occurs.</td>
</tr>
<tr>
<td>params</td>
<td>An object that may include additional parameters to include in the request.</td>
</tr>
</tbody>
</table>

If no identifier is specified, the default assumed keyword for the execute attribute is @this, and for the render attribute it is @none.

You can also place the JavaScript method in a file and include it as a resource.

**Using the @ResourceDependency Annotation in a Bean Class**

Use the `javax.faces.application.ResourceDependency` annotation to cause the bean class to load the default `jsf.js` library.

To load the Ajax resource from the server side, use the `javax.faces.ajax.request` method within the bean class. This method is usually used when creating a custom component or a custom renderer for a component.

The following example shows how the resource is loaded in a bean class:

```java
@ResourceDependency(name="jsf.js" library="javax.faces" target="head")
```

**The ajaxguessnumber Example Application**

To demonstrate the advantages of using Ajax, revisit the `guessnumber` example from Chapter 5, “Introduction to Facelets.” If you modify this example to use Ajax, the response need not be displayed in the `response.xhtml` page. Instead, an asynchronous call is made to the bean on the server side, and the response is displayed in the originating page by executing just the input component rather than by form submission.

The source code for this application is in the `tut-install/examples/web/ajaxguessnumber/` directory.
The ajaxguessnumber Source Files

The changes to the guessnumber application occur in two source files, as well as with the addition of a JavaScript file.

The ajaxgreeting.xhtml Facelets Page

The Facelets page for ajaxguessnumber, web/ajaxgreeting.xhtml, is almost the same as the greeting.xhtml page for the guessnumber application:

```xml
<h:head>
    <h:outputStylesheet library="css" name="default.css"/>
    <title>Ajax Guess Number Facelets Application</title>
</h:head>
<h:body>

    <h:form id="AjaxGuess">
        <h:outputScript name="ui.js" target="head"/>
        <h:graphicImage library="images" name="wave.med.gif"
                        alt="Duke waving his hand"/>

        <h2>
            Hi, my name is Duke. I am thinking of a number from
            #{userNumberBean.minimum} to #{userNumberBean.maximum}. Can you guess it?
        </h2>

        <p>
            <h:inputText
                id="userNo"
                title="Type a number from 0 to 10:"
                value="#{userNumberBean.userNumber}"
                <f:validateLongRange
                    minimum="#{userNumberBean.minimum}"
                    maximum="#{userNumberBean.maximum}"/>
            </h:inputText>

            <h:commandButton id="submit" value="Submit"/>
            <!--f:ajax execute="userNo" render="result errors1" onevent="msg"/>-->
            <f:ajax execute="userNo" render="result errors1"/>
        </h:commandButton>
    </p>

    <p>
        <h:message id="errors1" showSummary="true" showDetail="false"
                   style="color: #d20005;"
                   font-family: 'New Century Schoolbook', serif;
                   font-style: oblique;
                   text-decoration: overline"
                   for="userNo"/>
    </p>

</h:form>
</h:body>
```

The most important change is in the h:commandButton tag. The action attribute is removed from the tag, and f:ajax tag is added.
The f:ajax tag specifies that when the button is clicked, the h:inputText component with the id value userNo is executed. The components with the id values result and errors1 are then rendered. If that was all you did (as in the commented-out version of the tag), you would see the output from both the result and errors1 components, although only one output is valid; if a validation error occurs, the managed bean is not executed, so the result output is stale.

To solve this problem, the tag also calls the JavaScript function named msg, in the file ui.js, as described in the next section. The h:outputScript tag at the top of the form calls in this script.

The ui.js JavaScript File

The ui.js file specified in the h:outputScript tag of the ajaxgreeting.xhtml file is located in the web/resources directory of the application. The file contains just one function, msg:

```javascript
var msg = function msg(data) {
  var resultArea = document.getElementById("AjaxGuess:result");
  var errorArea = document.getElementById("AjaxGuess:errors1");
  if (errorArea.innerHTML !== null && errorArea.innerHTML !== 
    
  resultArea.innerHTML = "";
}
```

The msg function obtains a handle to both the result and errors1 elements. If the errors1 element has any content, the function erases the content of the result element, so the stale output does not appear in the page.

The UserNumberBean Managed Bean

A small change is also made in the UserNumberBean code so that the output component does not display any message for the default (null) value of the property response. Here is the modified bean code:

```java
public String getResponse() {
  if ((userNumber != null) && (userNumber.compareTo(randomInt) == 0)) {
    return "Yay! You got it!";
  }
  if (userNumber == null) {
    return null;
  } else {
    return "Sorry, " + userNumber + " is incorrect.";
  }
}
```

Running the ajaxguessnumber Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the ajaxguessnumber example.
To Build, Package, and Deploy the ajaxguessnumber Example Using NetBeans IDE

This procedure builds the application into the tut-install/examples/web/ajaxguessnumber/build/web/ directory. The contents of this directory are deployed to the GlassFish Server.

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   tut-install/examples/web/

3. Select the ajaxguessnumber folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the ajaxguessnumber project and select Deploy.

To Build, Package, and Deploy the ajaxguessnumber Example Using Ant

1. In a terminal window, go to:
   tut-install/examples/web/ajaxguessnumber/

2. Type the following command:
   
   ant
   
   This command calls the default target, which builds and packages the application into a WAR file, ajaxguessnumber.war, located in the dist directory.

3. Type the following command:

   ant deploy
   
   Typing this command deploys ajaxguessnumber.war to the GlassFish Server.

To Run the ajaxguessnumber Example

1. In a web browser, type the following URL:
   http://localhost:8080/ajaxguessnumber

2. Type a value in the input field and click Submit.
   If the value is in the range 0 to 10, a message states whether the guess is correct or incorrect. If the value is outside that range, or if the value is not a number, an error message appears in red.
To see what would happen if the JavaScript function were not included, remove the comment marks from the first `<f:ajax>` tag in `ajaxgreeting.xhtml` and place them around the second tag, as follows:

```html
<f:ajax execute="userNo" render="result errors1"/>
<!----<f:ajax execute="userNo" render="result errors1" onevent="msg"/>---->
```

If you then redeploy the application, you can see that stale output from valid guesses continues to appear if you subsequently type erroneous input.

---

**Further Information about Ajax in JavaServer Faces Technology**

For more information on Ajax in JavaServer Faces Technology, see

- JavaServer Faces project web site:
  
  [http://javaserverfaces.java.net/](http://javaserverfaces.java.net/)

- JavaServer Faces JavaScript Library APIs:

  [http://javaserverfaces.java.net/nonav/docs/2.1/jsdocs/symbols/jsf.ajax.html](http://javaserverfaces.java.net/nonav/docs/2.1/jsdocs/symbols/jsf.ajax.html)
This chapter describes the advanced features of composite components in JavaServer Faces technology.

A composite component is a special type of JavaServer Faces template that acts as a component. If you are new to composite components, see “Composite Components” on page 121 before you proceed with this chapter.

The following topics are addressed here:

- “Attributes of a Composite Component” on page 243
- “Invoking a Managed Bean” on page 244
- “Validating Composite Component Values” on page 245
- “The composite component login Example Application” on page 245

Attributes of a Composite Component

You define an attribute of a composite component by using the composite:attribute tag. Table 12-1 lists the commonly used attributes of this tag.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Specifies the name of the composite component attribute to be used in the using page. Alternatively, the name attribute can specify standard event handlers such as action, actionListener, and managed bean.</td>
</tr>
<tr>
<td>default</td>
<td>Specifies the default value of the composite component attribute.</td>
</tr>
<tr>
<td>required</td>
<td>Specifies whether it is mandatory to provide a value for the attribute.</td>
</tr>
</tbody>
</table>
The following code snippet defines a composite component attribute and assigns it a default value:

```xml
<composite:attribute name="username" default="admin"/>
```

The following code snippet uses the method-signature element:

```xml
<composite:attribute name="myaction"
    method-signature="java.lang.String action()"/>
```

The following code snippet uses the type element:

```xml
<composite:attribute name="dateofjoining" type="java.util.Date"/>
```

### Invoking a Managed Bean

To enable a composite component to handle server-side data, you can invoke a managed bean in one of the following ways:

- Pass the reference of the managed bean to the composite component.
- Directly use the properties of the managed bean.

The example application described in “The composite component login Example Application” on page 245 shows how to use a managed bean with a composite component by passing the reference of the managed bean to the component.
Validating Composite Component Values

JavaServer Faces provides the following tags for validating values of input components. These tags can be used with the composite:valueHolder or the composite:editableValueHolder tag.

Table 12–2 lists commonly used validator tags.

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f:validateBean</td>
<td>Delegates the validation of the local value to the Bean Validation API.</td>
</tr>
<tr>
<td>f:validateRegex</td>
<td>Uses the pattern attribute to validate the wrapping component. The entire</td>
</tr>
<tr>
<td></td>
<td>pattern is matched against the String value of the component. If it</td>
</tr>
<tr>
<td></td>
<td>matches, it is valid.</td>
</tr>
<tr>
<td>f:validateRequired</td>
<td>Enforces the presence of a value. Has the same effect as setting the</td>
</tr>
<tr>
<td></td>
<td>required element of a composite component's attribute to true.</td>
</tr>
</tbody>
</table>

The compositeComponentLogin Example Application

The compositeComponentLogin application creates a composite component that accepts a user name and a password. The component interacts with a managed bean. The component stores the user name and password in the managed bean, retrieves the values from the bean, and displays these values on the Login page.

The compositeComponentLogin application has a composite component file, a using page, and a managed bean.

The source code for this application is in the tut-install/examples/web/compositeComponentLogin/ directory.

The Composite Component File

The composite component file is an XHTML file, /web/resources/ezcomp/LoginPanel.xhtml. It has a composite:interface section that declares the labels for the user name, password, and login button. It also declares a managed bean, which defines properties for the user name and password.

```xml
<composite:interface>
  <composite:attribute name="namePrompt" default="User Name: "/>
  <composite:attribute name="passwordPrompt" default="Password: "/>
  <composite:attribute name="loginButtonText" default="Log In"/>
```
The composite component implementation accepts input values for the user name and password properties of the managed bean.

<composite:implmentation>
    <h:form id="form">
        <table columns="2" role="presentation">
            <tr>
                <td><h:outputLabel for="name" value="#{cc.attrs.namePrompt}" /></td>
                <td><h:inputText id="name" value="#{cc.attrs.myLoginBean.name}" required="true" /></td>
            </tr>
            <tr>
                <td><h:outputLabel for="password" value="#{cc.attrs.passwordPrompt}" /></td>
                <td><h:inputSecret id="password" value="#{cc.attrs.myLoginBean.password}" required="true" /></td>
            </tr>
        </table>
        <p>
            <h:commandButton id="loginButton" value="#{cc.attrs.loginButtonText}" action="#{cc.attrs.loginAction}"
        </p>
    </h:form>
</composite:implmentation>

The Using Page

The using page in this example application, web/index.xhtml, is an XHTML file that invokes the login composite component file along with the managed bean. It validates the user’s input.

<ez:LoginPanel myLoginBean="#{myLoginBean}"
    loginAction="#{myLoginBean.login}"
    />

The f:validateLength tag requires the user name to have from 4 to 10 characters.

The f:validateRegex tag requires the password to have from 4 to 10 characters and to contain at least one digit, one lowercase letter, and one uppercase letter.
The Managed Bean

The managed bean, `src/java/compositecomponentlogin/MyLoginBean.java`, defines a method called `login`, which retrieves the values of the user name and password.

```java
@ManagedBean
@RequestScoped
public class MyLoginBean {

    private String name;
    private String password;

    public MyLoginBean() {
    }

    public MyLoginBean(String name, String password) {
        this.name = name;
        this.password = password;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String newValue) {
        password = newValue;
    }

    public String getName() {
        return name;
    }

    public void setName(String newValue) {
        name = newValue;
    }

    public String login() {
        if (getName().equals("javaee")) {
            String msg = "Success. Your user name is " + getName()
                + " and your password is " + getPassword();
            FacesMessage facesMsg = new FacesMessage(msg, msg);
            FacesContext.getCurrentInstance().addMessage(null, facesMsg);
            return "index";
        } else {
            String msg = "Failure. Your user name is " + getName()
                + " and your password is " + getPassword();
            FacesMessage facesMsg =
                    new FacesMessage(FacesMessage.SEVERITY_ERROR, msg, msg);
            FacesContext.getCurrentInstance().addMessage(null, facesMsg);
            return "index";
        }
    }
}
```
Running the compositecomponentlogin Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the compositecomponentlogin example.

▼ To Build, Package, and Deploy the compositecomponentlogin Example Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   
tut-install/examples/web/

3. Select the compositecomponentlogin folder.

4. Select the Open as Main Project checkbox.

5. Click Open Project.

6. In the Projects tab, right-click compositecomponentlogin and select Deploy.

▼ To Build, Package, and Deploy the compositecomponentlogin Example Using Ant

1. In a terminal window, go to:
   
tut-install/examples/web/compositecomponentlogin/

2. Type the following command:
   
ant

3. Type the following command:
   
ant deploy

▼ To Run the compositecomponentlogin Example

1. In a web browser, type the following URL:
   
http://localhost:8080/compositecomponentlogin/

   The Login Component page opens.

2. Type values in the User Name and Password fields, then click the Log In button.

   Because of the way the login method is coded, the login succeeds only if the user name is javae.
Because of the `f:validateLength` tag, if the username has fewer than 4 characters or more than 10 characters, a validation error message appears.

Because of the `f:validateRegex` tag, if the password has fewer than 4 characters or more than 10 characters or does not contain at least one digit, one lowercase letter, and one uppercase letter, a "Regex Pattern not matched" error message appears.
Creating Custom UI Components and Other Custom Objects

JavaServer Faces technology offers a basic set of standard, reusable UI components that enable quick and easy construction of user interfaces for web applications. These components mostly map one-to-one to the elements in HTML 4. However, an application often requires a component that has additional functionality or requires a completely new component. JavaServer Faces technology allows extension of standard components to enhance their functionality or to create custom components. A rich ecosystem of third party component libraries is built on this extension capability, but it is beyond the scope of this tutorial to examine them. A web search for “JSF Component Libraries” is a good starting point to learn more about this important aspect of using JavaServer Faces technology.

In addition to extending the functionality of standard components, a component writer might want to give a page author the ability to change the appearance of the component on the page or to alter listener behavior. Alternatively, the component writer might want to render a component to a different kind of client device type, such as a smartphone or a tablet instead of a desktop computer. Enabled by the flexible JavaServer Faces architecture, a component writer can separate the definition of the component behavior from its appearance by delegating the rendering of the component to a separate renderer. In this way, a component writer can define the behavior of a custom component once but create multiple renderers, each of which defines a different way to render the component to a particular kind of client device.

A `javax.faces.component.UIComponent` is a Java class that is responsible for representing a self-contained piece of the user interface during the request processing lifecycle. It is intended to represent the meaning of the component; the visual representation of the component is the responsibility of the `javax.faces.render.Renderer`. There can be multiple instances of the same `UIComponent` class in any given JavaServer Faces view, just as there can be multiple instances of any Java class in any given Java program.

JavaServer Faces technology provides the ability to create custom components by extending the `UIComponent` class, the base class for all standard UI components. A custom component can be used anywhere an ordinary component can be used, such as within a composite component. A `UIComponent` is identified by two names: `component-family` specifies the purpose of the
component (input or output, for instance), while component-type indicates the specific purpose of a component, such as a text input field or a command button.

A Renderer is a helper to the UIComponent that deals with how that specific UIComponent class should appear in a specific kind of client device. Like components, renderers are identified by two names: render-kit-id and renderer-type. A render kit is just a bucket into which a particular group of renderers is placed, and the render-kit-id identifies the group. Most JavaServer Faces component libraries provide their own render kits.

A javax.faces.view.facelets.Tag object is a helper to the UIComponent and Renderer that allows the page author to include an instance of a UIComponent in a JavaServer Faces view. A tag represents a specific combination of component-type and renderer-type.

See “Component, Renderer, and Tag Combinations” on page 255 for information on how components, renderers, and tags interact.

This chapter uses the image map component from the Duke’s Bookstore case study example to explain how you can create simple custom components, custom renderers, and associated custom tags, and take care of all the other details associated with using the components and renderers in an application. See Chapter 51, “Duke’s Bookstore Case Study Example,” for more information about this example.

The chapter also describes how to create other custom objects: custom converters, custom listeners, and custom validators. It also describes how to bind component values and instances to data objects and how to bind custom objects to managed bean properties.

The following topics are addressed here:

- “Determining Whether You Need a Custom Component or Renderer” on page 253
- “Understanding the Image Map Example” on page 256
- “Steps for Creating a Custom Component” on page 261
- “Creating Custom Component Classes” on page 262
- “Delegating Rendering to a Renderer” on page 270
- “Implementing an Event Listener” on page 273
- “Handling Events for Custom Components” on page 275
- “Defining the Custom Component Tag in a Tag Library Descriptor” on page 276
- “Using a Custom Component” on page 277
- “Creating and Using a Custom Converter” on page 279
- “Creating and Using a Custom Validator” on page 283
- “Binding Component Values and Instances to Managed Bean Properties” on page 288
- “Binding Converters, Listeners, and Validators to Managed Bean Properties” on page 293
Determining Whether You Need a Custom Component or Renderer

The JavaServer Faces implementation supports a very basic set of components and associated renderers. This section helps you to decide whether you can use standard components and renderers in your application or need a custom component or custom renderer.

When to Use a Custom Component

A component class defines the state and behavior of a UI component. This behavior includes converting the value of a component to the appropriate markup, queuing events on components, performing validation, and any other behavior related to how the component interacts with the browser and the request processing lifecycle.

You need to create a custom component in the following situations:

- You need to add new behavior to a standard component, such as generating an additional type of event (for example, notifying another part of the page that something changed in this component as a result of user interaction).
- You need to take a different action in the request processing of the value of a component from what is available in any of the existing standard components.
- You want to take advantage of an HTML capability offered by your target browser, but none of the standard JavaServer Faces components take advantage of the capability in the way you want, if at all. The current release does not contain standard components for complex HTML components, such as frames; however, because of the extensibility of the component architecture, you can use JavaServer Faces technology to create components like these. The Duke’s Bookstore case study creates custom components that correspond to the HTML map and area tags.
- You need to render to a non-HTML client that requires extra components not supported by HTML. Eventually, the standard HTML render kit will provide support for all standard HTML components. However, if you are rendering to a different client, such as a phone, you might need to create custom components to represent the controls uniquely supported by the client. For example, some component architectures for wireless clients include support for tickers and progress bars, which are not available on an HTML client. In this case, you might also need a custom renderer along with the component; or you might need only a custom renderer.
You do not need to create a custom component in these cases:

- You need to aggregate components to create a new component that has its own unique behavior. In this situation, you can use a composite component to combine existing standard components. For more information on composite components, see “Composite Components” on page 121 and Chapter 12, “Composite Components: Advanced Topics and Example.”

- You simply need to manipulate data on the component or add application-specific functionality to it. In this situation, you should create a managed bean for this purpose and bind it to the standard component rather than create a custom component. See “Managed Beans in JavaServer Faces Technology” on page 189 for more information on managed beans.

- You need to convert a component’s data to a type not supported by its renderer. See “Using the Standard Converters” on page 175 for more information about converting a component’s data.

- You need to perform validation on the component data. Standard validators and custom validators can be added to a component by using the validator tags from the page. See “Using the Standard Validators” on page 183 and “Creating and Using a Custom Validator” on page 283 for more information about validating a component’s data.

- You need to register event listeners on components. You can either register event listeners on components using the `f:valueChangeListener` and `f:actionListener` tags, or you can point at an event-processing method on a managed bean using the component’s `actionListener` or `valueChangeListener` attributes. See “Implementing an Event Listener” on page 273 and “Writing Managed Bean Methods” on page 200 for more information.

### When to Use a Custom Renderer

A renderer, which generates the markup to display a component on a web page, allows you to separate the semantics of a component from its appearance. By keeping this separation, you can support different kinds of client devices with the same kind of authoring experience. You can think of a renderer as a “client adapter.” It produces output suitable for consumption and display by the client, and accepts input from the client when the user interacts with that component.

If you are creating a custom component, you need to ensure, among other things, that your component class performs these operations that are central to rendering the component:

- **Decoding:** Converting the incoming request parameters to the local value of the component

- **Encoding:** Converting the current local value of the component into the corresponding markup that represents it in the response
The JavaServer Faces specification supports two programming models for handling encoding and decoding:

- **Direct implementation**: The component class itself implements the decoding and encoding.
- **Delegated implementation**: The component class delegates the implementation of encoding and decoding to a separate renderer.

By delegating the operations to the renderer, you have the option of associating your custom component with different renderers so that you can render the component on different clients. If you don’t plan to render a particular component on different clients, it may be simpler to let the component class handle the rendering. However, a separate renderer enables you to preserve the separation of semantics from appearance. The Duke’s Bookstore application separates the renderers from the components, although it renders only to HTML 4 web browsers.

If you aren’t sure whether you will need the flexibility offered by separate renderers but you want to use the simpler direct-implementation approach, you can actually use both models. Your component class can include some default rendering code, but it can delegate rendering to a renderer if there is one.

### Component, Renderer, and Tag Combinations

When you create a custom component, you can create a custom renderer to go with it. To associate the component with the renderer and to reference the component from the page, you will also need a custom tag.

Although you need to write the custom component and renderer, there is no need to write code for a custom tag (called a tag handler). If you specify the component and renderer combination, Facelets creates the tag handler automatically.

In rare situations, you might use a custom renderer with a standard component rather than a custom component. Or you might use a custom tag without a renderer or a component. This section gives examples of these situations and summarizes what’s required for a custom component, renderer, and tag.

You would use a custom renderer without a custom component if you wanted to add some client-side validation on a standard component. You would implement the validation code with a client-side scripting language, such as JavaScript, and then render the JavaScript with the custom renderer. In this situation, you need a custom tag to go with the renderer so that its tag handler can register the renderer on the standard component.

Custom components as well as custom renderers need custom tags associated with them. However, you can have a custom tag without a custom renderer or custom component. For example, suppose that you need to create a custom validator that requires extra attributes on the
validator tag. In this case, the custom tag corresponds to a custom validator and not to a custom component or custom renderer. In any case, you still need to associate the custom tag with a server-side object.

Table 13–1 summarizes what you must or can associate with a custom component, custom renderer, or custom tag.

<table>
<thead>
<tr>
<th>Custom Item</th>
<th>Must Have</th>
<th>Can Have</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom component</td>
<td>Custom tag</td>
<td>Custom renderer or standard renderer</td>
</tr>
<tr>
<td>Custom renderer</td>
<td>Custom tag</td>
<td>Custom component or standard component</td>
</tr>
<tr>
<td>Custom JavaServer Faces tag</td>
<td>Some server-side object, like a component, a custom renderer, or custom validator</td>
<td>Custom component or standard component associated with a custom renderer</td>
</tr>
</tbody>
</table>

Understanding the Image Map Example

Duke’s Bookstore includes a custom image map component on the index.xhtml page. This image map displays a selection of six book titles. When the user clicks one of the book titles in the image map, the application goes to a page that displays the title of the selected book as well as information about a featured book. The page allows the user to add either book (or none) to the shopping cart.

Why Use JavaServer Faces Technology to Implement an Image Map?

JavaServer Faces technology is an ideal framework to use for implementing this kind of image map because it can perform the work that must be done on the server without requiring you to create a server-side image map.

In general, client-side image maps are preferred over server-side image maps for several reasons. One reason is that the client-side image map allows the browser to provide immediate feedback when a user positions the mouse over a hotspot. Another reason is that client-side image maps perform better because they don’t require round-trips to the server. However, in some situations, your image map might need to access the server to retrieve data or to change the appearance of non-form controls, tasks that a client-side image map cannot do.
Because the image map custom component uses JavaServer Faces technology, it has the best of both styles of image maps: It can handle the parts of the application that need to be performed on the server, while allowing the other parts of the application to be performed on the client side.

**Understanding the Rendered HTML**

Here is an abbreviated version of the form part of the HTML page that the application needs to render:

```html
<form id="j_idt13" name="j_idt13" method="post"
    action="/dukesbookstore/faces/index.xhtml" ... >
    ...
    <img id="j_idt13:mapImage"
        src="/dukesbookstore/faces/javax.faces.resource/book_all.jpg?ln=images"
        alt="Choose a Book from our Catalog"
        usemap="#bookMap" />
    ...
    <map name="bookMap">
        <area alt="Duke"
            coords="67,23,212,268"
            shape="rect"
            onmouseout=
                "document.forms[0]['j_idt13:mapImage'].src='resources/images/book_all.jpg'"
            onmouseover=
                "document.forms[0]['j_idt13:mapImage'].src='resources/images/book_201.jpg'"
            onclick=
                "document.forms[0]['bookMap_current'].value='Duke'; document.forms[0].submit()"
        />
        ...
    </map>
    ...
</form>
```

The `img` tag associates an image (`book_all.jpg`) with the image map referenced in the `usemap` attribute value.

The `map` tag specifies the image map and contains a set of `area` tags.

Each `area` tag specifies a region of the image map. The `onmouseover`, `onmouseout`, and `onclick` attributes define which JavaScript code is executed when these events occur. When the user moves the mouse over a region, the `onmouseover` function associated with the region displays the map with that region highlighted. When the user moves the mouse out of a region, the `onmouseout` function redisplay the original image. If the user clicks on a region, the `onclick` function sets the value of the `input` tag to the ID of the selected area and submits the page.

The `input` tag represents a hidden control that stores the value of the currently selected area between client-server exchanges so that the server-side component classes can retrieve the value.
The server-side objects retrieve the value of `bookMap_current` and set the locale in the `javax.faces.context.FacesContext` instance according to the region that was selected.

**Understanding the Facelets Page**

Here is an abbreviated form of the Facelets page that the image map component uses to generate the HTML page shown in the preceding section. It uses custom `bookstore:map` and `bookstore:area` tags to represent the custom components:

```html
<h:form>
  ...
  <h:graphicImage id="mapImage"
                   name="book_all.jpg"
                   library="images"
                   alt="#{bundle.ChooseBook}"
                   usemap="#bookMap" />
  <bookstore:map id="bookMap"
                 current="map1"
                 immediate="true"
                 action="bookstore">
    <f:actionListener
        type="dukesbookstore.listeners.MapBookChangeListener" />
    <bookstore:area id="map1" value="#{Book201}"
                    onmouseover="resources/images/book_201.jpg"
                    onMouseOut="resources/images/book_all.jpg"
                    targetImage="mapImage" />
    <bookstore:area id="map2" value="#{Book202}"
                    onmouseover="resources/images/book_202.jpg"
                    onMouseOut="resources/images/book_all.jpg"
                    targetImage="mapImage" />
  </bookstore:map>
  ...
</h:form>
```

The `alt` attribute of the `h:graphicImage` tag maps to the localized string "Choose a Book from our Catalog".

The `f:actionListener` tag within the `bookstore:map` tag points to a listener class for an action event. The processAction method of the listener places the book ID for the selected map area into the session map. The way this event is handled is explained more in "Handling Events for Custom Components" on page 275.

The `action` attribute of the `bookstore:map` tag specifies a logical outcome String, "bookstore", which by implicit navigation rules sends the application to the page `bookstore.xhtml`. For more information on navigation, see the section "Configuring Navigation Rules" on page 313.

The `immediate` attribute of the `bookstore:map` tag is set to true, which indicates that the default `javax.faces.event.ActionListener` implementation should execute during the Apply Request Values phase of the request-processing lifecycle, instead of waiting for the
Invoke Application phase. Because the request resulting from clicking the map does not require any validation, data conversion, or server-side object updates, it makes sense to skip directly to the Invoke Application phase.

The current attribute of the bookstore:map tag is set to the default area, which is map1 (the book *My Early Years: Growing Up on Star7*, by Duke).

Notice that the bookstore:area tags do not contain any of the JavaScript, coordinate, or shape data that is displayed on the HTML page. The JavaScript is generated by the dukesbookstore.renderers.AreaRenderer class. The onmouseover and onmouseout attribute values indicate the image to be loaded when these events occur. How the JavaScript is generated is explained more in "Performing Encoding" on page 265.

The coordinate, shape, and alternate text data are obtained through the value attribute, whose value refers to an attribute in application scope. The value of this attribute is a bean, which stores the coords, shape, and alt data. How these beans are stored in the application scope is explained more in the next section.

### Configuring Model Data

In a JavaServer Faces application, data such as the coordinates of a hotspot of an image map is retrieved from the value attribute through a bean. However, the shape and coordinates of a hotspot should be defined together because the coordinates are interpreted differently depending on what shape the hotspot is. Because a component’s value can be bound only to one property, the value attribute cannot refer to both the shape and the coordinates.

To solve this problem, the application encapsulates all of this information in a set of ImageArea objects. These objects are initialized into application scope by the managed bean creation facility (see “Managed Beans in JavaServer Faces Technology” on page 189). Here is part of the managed bean declaration for the ImageArea bean corresponding to the South America hotspot:

```xml
<managed-bean eager="true">
  ...
  <managed-bean-name> Book201 </managed-bean-name>
  <managed-bean-class> dukesbookstore.model.ImageArea </managed-bean-class>
  <managed-bean-scope> application </managed-bean-scope>
  <managed-property>
    ...
    <property-name>shape</property-name>
    <value>rect</value>
  </managed-property>
  ...
  <managed-bean-name> alt </managed-bean-name>
  <managed-bean-class> Duke </managed-bean-class>
</managed-bean>
```
For more information on initializing managed beans with the managed bean creation facility, see the section "Application Configuration Resource File" on page 297.

The value attributes of the bookstore:area tags refer to the beans in the application scope, as shown in this bookstore:area tag from index.xhtml:

```xml
<bookstore:area id="map1" value="#{Book201}" onmouseover="resources/images/book_201.jpg" onmouseout="resources/images/book_all.jpg" targetImage="mapImage" />
```

To reference the ImageArea model object bean values from the component class, you implement a `getValue` method in the component class. This method calls `super.getValue`. The superclass of `tut-install/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/components/AreaComponent.java`, `UIOutput`, has a `getValue` method that does the work of finding the ImageArea object associated with AreaComponent. The AreaRenderer class, which needs to render the alt, shape, and coords values from the ImageArea object, calls the `getValue` method of AreaComponent to retrieve the ImageArea object.

```java
ImageArea iarea = (ImageArea) area.getValue();
```

ImageArea is a simple bean, so you can access the shape, coordinates, and alternative text values by calling the appropriate accessor methods of ImageArea. "Creating the Renderer Class" on page 270 explains how to do this in the AreaRenderer class.

## Summary of the Image Map Application Classes

Table 13–2 summarizes all the classes needed to implement the image map component.

<table>
<thead>
<tr>
<th>Class</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AreaSelectedEvent</td>
<td>The javax.faces.event.ActionEvent indicating that an AreaComponent from the MapComponent has been selected.</td>
</tr>
<tr>
<td>AreaComponent</td>
<td>The class that defines AreaComponent, which corresponds to the bookstore:area custom tag.</td>
</tr>
<tr>
<td>MapComponent</td>
<td>The class that defines MapComponent, which corresponds to the bookstore:map custom tag.</td>
</tr>
</tbody>
</table>
### Steps for Creating a Custom Component

You can apply the following steps while developing your own custom component.

1. Create a custom component class that does the following:
   a. Overrides the `getFamily` method to return the component family, which is used to look up renderers that can render the component.
   b. Includes the rendering code or delegates it to a renderer (explained in step 2).
   c. Enables component attributes to accept expressions.
   d. Queues an event on the component if the component generates events.
   e. Saves and restores the component state.

2. Delegate rendering to a renderer if your component does not handle the rendering. To do this:
   a. Create a custom renderer class by extending `javax.faces.render.Renderer`.
   b. Register the renderer to a render kit.

3. Register the component.

4. Create an event handler if your component generates events.

5. Create a tag library descriptor (TLD) that defines the custom tag.

See "Registering a Custom Component" on page 319 and "Registering a Custom Renderer with a Render Kit" on page 317 for information on registering the custom component and the renderer. The section "Using a Custom Component" on page 277 discusses how to use the custom component in a JavaServer Faces page.
Creating Custom Component Classes

As explained in “When to Use a Custom Component” on page 253, a component class defines the state and behavior of a UI component. The state information includes the component’s type, identifier, and local value. The behavior defined by the component class includes the following:

- Decoding (converting the request parameter to the component’s local value)
- Encoding (converting the local value into the corresponding markup)
- Saving the state of the component
- Updating the bean value with the local value
- Processing validation on the local value
- Queueing events

The `javax.faces.component.UIComponentBase` class defines the default behavior of a component class. All the classes representing the standard components extend from `UIComponentBase`. These classes add their own behavior definitions, as your custom component class will do.

Your custom component class must either extend `UIComponentBase` directly or extend a class representing one of the standard components. These classes are located in the `javax.faces.component` package and their names begin with `UI`.

If your custom component serves the same purpose as a standard component, you should extend that standard component rather than directly extend `UIComponentBase`. For example, suppose you want to create an editable menu component. It makes sense to have this component extend `UISelectOne` rather than `UIComponentBase` because you can reuse the behavior already defined in `UISelectOne`. The only new functionality you need to define is to make the menu editable.

Whether you decide to have your component extend `UIComponentBase` or a standard component, you might also want your component to implement one or more of these behavioral interfaces defined in the `javax.faces.component` package:

- `ActionSource`: Indicates that the component can fire a `javax.faces.event.ActionEvent`.
- `ActionSource2`: Extends `ActionSource` and allows component properties referencing methods that handle action events to use method expressions as defined by the unified EL.
- `EditableValueHolder`: Extends `ValueHolder` and specifies additional features for editable components, such as validation and emitting value-change events.
- `NamingContainer`: Mandates that each component rooted at this component have a unique ID.
- `StateHolder`: Denotes that a component has state that must be saved between requests.
- `ValueHolder`: Indicates that the component maintains a local value as well as the option of accessing data in the model tier.
If your component extends UIComponentBase, it automatically implements only StateHolder. Because all components directly or indirectly extend UIComponentBase, they all implement StateHolder. Any component that implements StateHolder also implements the StateHelper interface, which extends StateHolder and defines a Map-like contract that makes it easy for components to save and restore a partial view state.

If your component extends one of the other standard components, it might also implement other behavioral interfaces in addition to StateHolder. If your component extends UICommand, it automatically implements ActionSource2. If your component extends UIOutput or one of the component classes that extend UIOutput, it automatically implements ValueHolder. If your component extends UIInput, it automatically implements EditableValueHolder and ValueHolder. See the JavaServer Faces API documentation to find out what the other component classes implement.

You can also make your component explicitly implement a behavioral interface that it doesn’t already by virtue of extending a particular standard component. For example, if you have a component that extends UIInput and you want it to fire action events, you must make it explicitly implement ActionSource2 because a UIInput component doesn’t automatically implement this interface.

The Duke’s Bookstore image map example has two component classes: AreaComponent and MapComponent. The MapComponent class extends UICommand and therefore implements ActionSource2, which means it can fire action events when a user clicks on the map. The AreaComponent class extends the standard component UIOutput. The @FacesComponent annotation registers the components with the JavaServer Faces implementation:

```java
@FacesComponent("DemoMap")
public class MapComponent extends UICommand {...}

@FacesComponent("DemoArea")
public class AreaComponent extends UIOutput {...}
```

The MapComponent class represents the component corresponding to the bookstore:map tag:

```xml
<bookstore:map id="bookMap"
    current="map1"
    immediate="true"
    action="bookstore">
    ...
</bookstore:map>
```
The AreaComponent class represents the component corresponding to the bookstore:area tag:

```xml
<bookstore:area id="map1" value="#{Book201}"
    onmouseover="resources/images/book_201.jpg"
    onmouseout="resources/images/book_all.jpg"
    targetImage="mapImage"/>
```

MapComponent has one or more AreaComponent instances as children. Its behavior consists of the following actions:

- Retrieving the value of the currently selected area
- Defining the properties corresponding to the component’s values
- Generating an event when the user clicks on the image map
- Queuing the event
- Saving its state
- Rendering the HTML map tag and the HTML input tag

MapComponent delegates the rendering of the HTML map and input tags to the MapRenderer class.

AreaComponent is bound to a bean that stores the shape and coordinates of the region of the image map. You will see how all this data is accessed through the value expression in “Creating the Renderer Class” on page 270. The behavior of AreaComponent consists of the following:

- Retrieving the shape and coordinate data from the bean
- Setting the value of the hidden tag to the id of this component
- Rendering the area tag, including the JavaScript for the onmouseover, onmouseout, and onclick functions

Although these tasks are actually performed by AreaRenderer, AreaComponent must delegate the tasks to AreaRenderer. See “Delegating Rendering to a Renderer” on page 270 for more information.

The rest of this section describes the tasks that MapComponent performs as well as the encoding and decoding that it delegates to MapRenderer. “Handling Events for Custom Components” on page 275 details how MapComponent handles events.

### Specifying the Component Family

If your custom component class delegates rendering, it needs to override the getFamily method of UIComponent to return the identifier of a component family, which is used to refer to a component or set of components that can be rendered by a renderer or set of renderers.

The component family is used along with the renderer type to look up renderers that can render the component.
public String getFamily() {
    return "Map";
}

The component family identifier, Map, must match that defined by the component-family elements included in the component and renderer configurations in the application configuration resource file. “Registering a Custom Renderer with a Render Kit” on page 317 explains how to define the component family in the renderer configuration. “Registering a Custom Component” on page 319 explains how to define the component family in the component configuration.

Performing Encoding

During the Render Response phase, the JavaServer Faces implementation processes the encoding methods of all components and their associated renderers in the view. The encoding methods convert the current local value of the component into the corresponding markup that represents it in the response.

The UIComponentBase class defines a set of methods for rendering markup: encodeBegin, encodeChildren, and encodeEnd. If the component has child components, you might need to use more than one of these methods to render the component; otherwise, all rendering should be done in encodeEnd. Alternatively, you can use the encodeALL method, which encompasses all the methods.

Because MapComponent is a parent component of AreaComponent, the area tags must be rendered after the beginning map tag and before the ending map tag. To accomplish this, the MapRenderer class renders the beginning map tag in encodeBegin and the rest of the map tag in encodeEnd.

The JavaServer Faces implementation automatically invokes the encodeEnd method of AreaComponent’s renderer after it invokes MapRenderer’s encodeBegin method and before it invokes MapRenderer’s encodeEnd method. If a component needs to perform the rendering for its children, it does this in the encodeChildren method.

Here are the encodeBegin and encodeEnd methods of MapRenderer:

@override
public void encodeBegin(FacesContext context, UIComponent component) throws IOException {
    if ((context == null)|| (component == null)){
        throw new NullPointerException();
    }
    MapComponent map = (MapComponent) component;
    ResponseWriter writer = context.getResponseWriter();
    writer.startElement("map", map);
    writer.writeAttribute("name", map.getId(), "id");
}
@Override
public void encodeEnd(FacesContext context, UIComponent component)
  throws IOException {
  if ((context == null) || (component == null)) {
    throw new NullPointerException();
  }
  MapComponent map = (MapComponent) component;
  ResponseWriter writer = context.getResponseWriter();
  writer.startElement("input", map);
  writer.writeAttribute("type", "hidden", null);
  writer.writeAttribute("name", getName(context, map), "clientId");
  writer.endElement("input");
  writer.endElement("map");
}

Notice that encodeBegin renders only the beginning map tag. The encodeEnd method renders the input tag and the ending map tag.

The encoding methods accept a UIComponent argument and a javax.faces.context.FacesContext argument. The FacesContext instance contains all the information associated with the current request. The UIComponent argument is the component that needs to be rendered.

The rest of the method renders the markup to the javax.faces.context.ResponseWriter instance, which writes out the markup to the current response. This basically involves passing the HTML tag names and attribute names to the ResponseWriter instance as strings, retrieving the values of the component attributes, and passing these values to the ResponseWriter instance.

The startElement method takes a String (the name of the tag) and the component to which the tag corresponds (in this case, map). (Passing this information to the ResponseWriter instance helps design-time tools know which portions of the generated markup are related to which components.)

After calling startElement, you can call writeAttribute to render the tag's attributes. The writeAttribute method takes the name of the attribute, its value, and the name of a property or attribute of the containing component corresponding to the attribute. The last parameter can be null, and it won't be rendered.

The name attribute value of the map tag is retrieved using the getId method of UIComponent, which returns the component's unique identifier. The name attribute value of the input tag is retrieved using the getName(FacesContext, UIComponent) method of MapRenderer.
If you want your component to perform its own rendering but delegate to a renderer if there is one, include the following lines in the encoding method to check whether there is a renderer associated with this component:

```java
if (getRendererType() != null) {
    super.encodeEnd(context);
    return;
}
```

If there is a renderer available, this method invokes the superclass's `encodeEnd` method, which does the work of finding the renderer. The `MapComponent` class delegates all rendering to `MapRenderer`, so it does not need to check for available renderers.

In some custom component classes that extend standard components, you might need to implement other methods in addition to `encodeEnd`. For example, if you need to retrieve the component's value from the request parameters, you must also implement the `decode` method.

### Performing Decoding

During the Apply Request Values phase, the JavaServer Faces implementation processes the decode methods of all components in the tree. The `decode` method extracts a component's local value from incoming request parameters and uses a `javax.faces.convert.Converter` implementation to convert the value to a type that is acceptable to the component class.

A custom component class or its renderer must implement the `decode` method only if it must retrieve the local value or if it needs to queue events. The component queues the event by calling `queueEvent`.

Here is the `decode` method of `MapRenderer`:

```java
@Override
public void decode(FacesContext context, UIComponent component) {
    if ((context == null) || (component == null)) {
        throw new NullPointerException();
    }
    MapComponent map = (MapComponent) component;
    String key = getName(context, map);
    String value = (String) context.getExternalContext().
                   getRequestParameterMap().get(key);
    if (value != null)
        map.setCurrent(value);
}
```

The `decode` method first gets the name of the hidden input field by calling `getName(FacesContext, UIComponent)`. It then uses that name as the key to the request parameter map to retrieve the current value of the input field. This value represents the currently selected area. Finally, it sets the value of the `MapComponent` class's `current` attribute to the value of the input field.
Enabling Component Properties to Accept Expressions

Nearly all the attributes of the standard JavaServer Faces tags can accept expressions, whether they are value expressions or method expressions. It is recommended that you also enable your component attributes to accept expressions because it gives you much more flexibility when you write Facelets pages.

To enable the attributes to accept expressions, the component class must implement getter and setter methods for the component properties. These methods can use the facilities offered by the StateHelper interface to store and retrieve not only the values for these properties, but also the state of the components across multiple requests.

Because MapComponent extends UICommand, the UICommand class already does the work of getting the ValueExpression and MethodExpression instances associated with each of the attributes that it supports. Similarly, the UIOutput class that AreaComponent extends already obtains the ValueExpression instances for its supported attributes. For both components, the simple getter and setter methods store and retrieve the key values and state for the attributes, as shown in this code fragment from AreaComponent:

```java
enum PropertyKeys {
    alt, coords, shape, targetImage;
}

public String getAlt() {
    return (String) getStateHelper().eval(PropertyKeys.alt, null);
}

public void setAlt(String alt) {
    getStateHelper().put(PropertyKeys.alt, alt);
}

...'
```

However, if you have a custom component class that extends UIComponentBase, you will need to implement the methods that get the ValueExpression and MethodExpression instances associated with those attributes that are enabled to accept expressions. For example, you could include a method that gets the ValueExpression instance for the immediate attribute:

```java
public boolean isImmediate() {
    if (this.immediateSet) {
        return (this.immediate);
    }
    ValueExpression ve = getValueExpression("immediate");
    if (ve != null) {
        Boolean value = (Boolean) ve.getValue(
            getFacesContext().getELContext());
        return (value.booleanValue());
    } else {
        return (this.immediate);
    }
}
```
The properties corresponding to the component attributes that accept method expressions must accept and return a MethodExpression object. For example, if MapComponent extended UIComponentBase instead of UICommand, it would need to provide an action property that returns and accepts a MethodExpression object:

```java
public MethodExpression getAction() {
    return (this.action);
}
public void setAction(MethodExpression action) {
    this.action = action;
}
```

### Saving and Restoring State

As described in “Enabling Component Properties to Accept Expressions” on page 268, use of the StateHelper interface facilities allows you to save the component’s state at the same time you set and retrieve property values. The StateHelper implementation allows partial state saving: it saves only the changes in the state since the initial request, not the entire state, because the full state can be restored during the Restore View phase.

Component classes that implement StateHolder may prefer to implement the saveState(FacesContext) and restoreState(FacesContext, Object) methods to help the JavaServer Faces implementation save and restore the full state of components across multiple requests.

To save a set of values, you can implement the saveState(FacesContext) method. This method is called during the Render Response phase, during which the state of the response is saved for processing on subsequent requests. Here is a hypothetical method from MapComponent, which has only one attribute, current:

```java
@Override
public Object saveState(FacesContext context) {
    Object values[] = new Object[2];
    values[0] = super.saveState(context);
    values[1] = current;
    return (values);
}
```

This method initializes an array, which will hold the saved state. It next saves all of the state associated with the component.

A component that implements StateHolder may also provide an implementation for restoreState(FacesContext, Object), which restores the state of the component to that saved with the saveState(FacesContext) method. The restoreState(FacesContext, Object) method is called during the Restore View phase, during which the JavaServer Faces implementation checks whether there is any state that was saved during the last Render Response phase and needs to be restored in preparation for the next postback.
Here is a hypothetical `restoreState(FacesContext, Object)` method from `MapComponent`:

```java
public void restoreState(FacesContext context, Object state) {
    Object values[] = (Object[]) state;
    super.restoreState(context, values[0]);
    current = (String) values[1];
}
```

This method takes a `FacesContext` and an `Object` instance, representing the array that is holding the state for the component. This method sets the component's properties to the values saved in the `Object` array.

Whether or not you implement these methods in a component class, you can use the `javax.faces.STATE_SAVING_METHOD` context parameter to specify in the deployment descriptor where you want the state to be saved: either `client` or `server`. If state is saved on the client, the state of the entire view is rendered to a hidden field on the page. By default, the state is saved on the server.

The web applications in the Duke's Forest case study save their view state on the client.

Saving state on the client uses more bandwidth as well as more client resources, while saving it on the server uses more server resources. You may also want to save state on the client if you expect your users to disable cookies.

---

### Delegating Rendering to a Renderer

Both `MapComponent` and `AreaComponent` delegate all of their rendering to a separate renderer. The section "Performing Encoding" on page 265 explains how `MapRenderer` performs the encoding for `MapComponent`. This section explains in detail the process of delegating rendering to a renderer using `AreaRenderer`, which performs the rendering for `AreaComponent`.

To delegate rendering, you perform these tasks:

- Create the `Renderer` class.
- Register the renderer with a render kit by using the `@FacesRenderer` annotation (or by using the application configuration resource file, as explained in "Registering a Custom Renderer with a Render Kit" on page 317).
- Identify the renderer type in the `@FacesRenderer` annotation.

### Creating the Renderer Class

When delegating rendering to a renderer, you can delegate all encoding and decoding to the renderer, or you can choose to do part of it in the component class. The `AreaComponent` class delegates encoding to the `AreaRenderer` class.
The renderer class begins with a @FacesRenderer annotation:

```java
@FacesRenderer(componentFamily = "Area",
rendererType = "dukesbookstore.renderers.AreaRenderer")
public class AreaRenderer extends Renderer {

The @FacesRenderer annotation registers the renderer class with the JavaServer Faces implementation as a renderer class. The annotation identifies the component family as well as the renderer type.

To perform the rendering for AreaComponent, AreaRenderer must implement an encodeEnd method. The encodeEnd method of AreaRenderer retrieves the shape, coordinates, and alternative text values stored in the ImageArea bean that is bound to AreaComponent. Suppose that the area tag currently being rendered has a value attribute value of "book203". The following line from encodeEnd gets the value of the attribute "book203" from the FacesContext instance.

```java
ImageArea ia = (ImageArea)area.getValue();
```

The attribute value is the ImageArea bean instance, which contains the shape, coords, and alt values associated with the book203 AreaComponent instance. "Configuring Model Data" on page 259 describes how the application stores these values.

After retrieving the ImageArea object, the method renders the values for shape, coords, and alt by simply calling the associated accessor methods and passing the returned values to the `javax.faces.context.ResponseWriter` instance, as shown by these lines of code, which write out the shape and coordinates:

```java
writer.startElement("area", area);
writer.writeAttribute("alt", iarea.getAlt(), "alt");
writer.writeAttribute("coords", iarea.getCoords(), "coords");
writer.writeAttribute("shape", iarea.getShape(), "shape");
```

The encodeEnd method also renders the JavaScript for the onmouseover, onmouseout, and onclick attributes. The Facelets page need only provide the path to the images that are to be loaded during an onmouseover or onmouseout action:

```xml
<bookstore:area id="map3" value="#{Book203}"
    onmouseover="resources/images/book_203.jpg"
    onmouseout="resources/images/book_all.jpg"
    targetImage="mapImage"/>
```

The AreaRenderer class takes care of generating the JavaScript for these actions, as shown in the following code from encodeEnd. The JavaScript that AreaRenderer generates for the onclick action sets the value of the hidden field to the value of the current area's component ID and submits the page.

```java
sb = new StringBuffer("document.forms[0][".append(targetImageId).
    append("].src=");
sb.append("Delegating Rendering to a Renderer

Chapter 13 • Creating Custom UI Components and Other Custom Objects 271

```
getURI(context, (String) area.getAttributes().get("onmouseout"));
    sb.append("'");
    writer.writeAttribute("onmouseout", sb.toString(), "onmouseout");
    sb = new StringBuffer("document.forms[0]["\].append(targetImageId).
                        append("'.src='");
    sb.append(getURI(context, (String) area.getAttributes().get("onmouseover"));
    sb.append("'");
    writer.writeAttribute("onmouseover", sb.toString(), "onmouseover");
    sb = new StringBuffer("document.forms[0]['").append(targetImageId).
                        append("'].value='");
    sb.append(iarea.getAlt());
    sb.append("'; document.forms[0].submit()");
    writer.writeAttribute("onclick", sb.toString(), "value");
    writer.endElement("area");

By submitting the page, this code causes the JavaServer Faces lifecycle to return back to the Restore View phase. This phase saves any state information, including the value of the hidden field, so that a new request component tree is constructed. This value is retrieved by the decode method of the MapComponent class. This decode method is called by the JavaServer Faces implementation during the Apply Request Values phase, which follows the Restore View phase.

In addition to the encodeEnd method, AreaRenderer contains an empty constructor. This is used to create an instance of AreaRenderer so that it can be added to the render kit.

The @FacesRenderer annotation registers the renderer class with the JavaServer Faces implementation as a renderer class. The annotation identifies the component family as well as the renderer type.

**Identifying the Renderer Type**

During the Render Response phase, the JavaServer Faces implementation calls the getRendererType method of the component's tag handler to determine which renderer to invoke, if there is one.

You identify the type associated with the renderer in the rendererType element of the @FacesRenderer annotation for AreaRenderer as well as in the renderer-type element of the tag library descriptor file.
Implementing an Event Listener

The JavaServer Faces technology supports action events and value-change events for components.

Action events occur when the user activates a component that implements `javax.faces.component.ActionSource`. These events are represented by the class `javax.faces.event.ActionEvent`.

Value-change events occur when the user changes the value of a component that implements `javax.faces.component.EditableValueHolder`. These events are represented by the class `javax.faces.event.ValueChangeEvent`.

One way to handle events is to implement the appropriate listener classes. Listener classes that handle the action events in an application must implement the interface `javax.faces.event.ActionListener`. Similarly, listeners that handle the value-change events must implement the interface `javax.faces.event.ValueChangeListener`.

This section explains how to implement the two listener classes.

To handle events generated by custom components, you must implement an event listener and an event handler and manually queue the event on the component. See "Handling Events for Custom Components" on page 275 for more information.

**Note** – You do not need to create an `ActionListener` implementation to handle an event that results solely in navigating to a page and does not perform any other application-specific processing. See "Writing a Method to Handle Navigation" on page 201 for information on how to manage page navigation.

Implementing Value-Change Listeners

A `javax.faces.event.ValueChangeListener` implementation must include a `processValueChange(ValueChangeEvent)` method. This method processes the specified value-change event and is invoked by the JavaServer Faces implementation when the value-change event occurs. The `ValueChangeEvent` instance stores the old and the new values of the component that fired the event.

In the Duke’s Bookstore case study, the `NameChanged` listener implementation is registered on the name `UIInput` component on the `bookcashier.xhtml` page. This listener stores into session scope the name the user entered in the text field corresponding to the name component.

The `bookreceipt.xhtml` subsequently retrieves the name from the session scope:
Implementing an Event Listener

```
<h:outputFormat title="thanks"
    value="#{bundle.ThankYouParam}"/>
<f:param value="#{sessionScope.name}"/>
</h:outputFormat>

When the bookreceipt.xhtml page is loaded, it displays the name inside the message:

"Thank you, {0}, for purchasing your books from us."

Here is part of the NameChanged listener implementation:

```java
public class NameChanged extends Object implements ValueChangeListener {
    @Override
    public void processValueChange(ValueChangeEvent event)
        throws AbortProcessingException {
        if (null != event.getNewValue()) {
            FacesContext.getCurrentInstance().getExternalContext().
                getSessionMap().put("name", event.getNewValue());
        }
    }
}
```

When the user enters the name in the text field, a value-change event is generated, and the `processValueChange(ValueChangeEvent)` method of the NameChanged listener implementation is invoked. This method first gets the ID of the component that fired the event from the `ValueChangeEvent` object, and it puts the value, along with an attribute name, into the session map of the `FacesContext` instance.

"Registering a Value-Change Listener on a Component" on page 181 explains how to register this listener onto a component.

Implementing Action Listeners


The Duke's Bookstore case study uses two `ActionListener` implementations, `LinkBookChangeListener` and `MapBookChangeListener`. See "Handling Events for Custom Components" on page 275 for details on `MapBookChangeListener`.

"Registering an Action Listener on a Component" on page 182 explains how to register this listener onto a component.
Handling Events for Custom Components

As explained in "Implementing an Event Listener" on page 273, events are automatically queued on standard components that fire events. A custom component, on the other hand, must manually queue events from its decode method if it fires events.

"Performing Decoding" on page 267 explains how to queue an event on MapComponent using its decode method. This section explains how to write the class that represents the event of clicking on the map and how to write the method that processes this event.

As explained in "Understanding the Facelets Page" on page 258, the actionListener attribute of the bookstore:map tag points to the MapBookChangeListener class. The listener class's processAction method processes the event of clicking the image map. Here is the processAction method:

```java
@override
public void processAction(ActionEvent actionEvent)
    throws AbortProcessingException {
    AreaSelectedEvent event = (AreaSelectedEvent) actionEvent;
    String current = event.getMapComponent().getCurrent();
    FacesContext context = FacesContext.getCurrentInstance();
    String bookId = books.get(current);
    context.getExternalContext().getSessionMap().put("bookId", bookId);
}
```

When the JavaServer Faces implementation calls this method, it passes in an ActionEvent object that represents the event generated by clicking on the image map. Next, it casts it to an AreaSelectedEvent object (see tut-install/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/listeners/AreaSelectedEvent.java). Then this method gets the MapComponent associated with the event. It then gets the value of the MapComponent object's current attribute, which indicates the currently selected area. The method then uses the value of the current attribute to get the book's ID value from a HashMap object, which is constructed elsewhere in the MapBookChangeListener class. Finally, the method places the ID obtained from the HashMap object into the session map for the application.

In addition to the method that processes the event, you need the event class itself. This class is very simple to write: You have it extend ActionEvent and provide a constructor that takes the component on which the event is queued and a method that returns the component.
Here is the `AreaSelectedEvent` class used with the image map:

```java
public class AreaSelectedEvent extends ActionEvent {
    public AreaSelectedEvent(MapComponent map) {
        super(map);
    }
    public MapComponent getMapComponent() {
        return ((MapComponent) getComponent());
    }
}
```

As explained in the section "Creating Custom Component Classes" on page 262, in order for `MapComponent` to fire events in the first place, it must implement `ActionSource`. Because `MapComponent` extends `UICommand`, it also implements `ActionSource`.

### Defining the Custom Component Tag in a Tag Library Descriptor

To use a custom tag, you declare it in a Tag Library Descriptor (TLD). The TLD file defines how the custom tag is used in a JavaServer Faces page. The web container uses the TLD to validate the tag. The set of tags that are part of the HTML render kit are defined in the HTML_BASIC TLD, available at [http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/renderkitdocs/](http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/renderkitdocs/).

The TLD file name must end with `taglib.xml`. In the Duke's Bookstore case study, the custom tags `area` and `map` are defined in the file `web/WEB-INF/bookstore.taglib.xml`.

All tag definitions must be nested inside the `facelet-taglib` element in the TLD. Each tag is defined by a `tag` element that specifies a particular combination of a component type and a renderer type. Here are the tag definitions for the `area` and `map` components:

```xml
<facelet-taglib xmlns="http://java.sun.com/xml/ns/javae"

...>
    <namespace>http://dukesbookstore</namespace>
    <tag>
        <tag-name>area</tag-name>
        <component>
            <component-type>DemoArea</component-type>
            <renderer-type>DemoArea</renderer-type>
        </component>
    </tag>
    <tag>
        <tag-name>map</tag-name>
        <component>
            <component-type>DemoMap</component-type>
            <renderer-type>DemoMap</renderer-type>
        </component>
    </tag>
</facelet-taglib>
```
The component-type element specifies the name defined in the @FacesComponent annotation, while the renderer-type element specifies the rendererType defined in the @FacesRenderer annotation.

The facelet-taglib element must also include a namespace element, which defines the namespace to be specified in pages that use the custom component. See "Using a Custom Component" on page 277 for information on specifying the namespace in pages.

The TLD file is located in the WEB-INF directory. In addition, an entry is included in the web deployment descriptor (web.xml) to identify the custom tag library descriptor file, as follows:

```xml
<context-param>
  <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
  <param-value>/WEB-INF/bookstore.taglib.xml</param-value>
</context-param>
```

### Using a Custom Component

To use a custom component in a page, you add the custom tag associated with the component to the page.

As explained in "Defining the Custom Component Tag in a Tag Library Descriptor" on page 276, you must ensure that the TLD that defines any custom tags is packaged in the application if you intend to use the tags in your pages. TLD files are stored in the WEB-INF directory or subdirectory of the WAR file or in the META-INF directory or subdirectory of a tag library packaged in a JAR file.

You also need to include a namespace declaration in the page so that the page has access to the tags. The custom tags for the Duke's Bookstore case study are defined in bookstore.taglib.xml. The ui:composition tag on the index.xhtml page declares the namespace defined in the tag library:

```xml
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://java.sun.com/jsf.facelets"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:f="http://java.sun.com/jsf/core"
                xmlns:bookstore="http://dukesbookstore"
                template="./bookstoreTemplate.xhtml">
```

Finally, to use a custom component in a page, you add the component's tag to the page.
The Duke's Bookstore case study includes a custom image map component on the index.xhtml page. This component allows you to select a book by clicking on a region of the image map:

```xml
...<h:graphicImage id="mapImage"
    name="book_all.jpg"
    library="images"
    alt="#{bundle.chooseLocale}"
    usemap="#bookMap" />
<bookstore:map id="bookMap"
    current="map1"
    immediate="true"
    action="bookstore">
    <f:actionListener
        type="dukesbookstore.listeners.MapBookChangeListener" />
    <bookstore:area id="map1" value="#{Book201}"
        onmouseover="resources/images/book_201.jpg"
        onmouseout="resources/images/book_all.jpg"
        targetImage="mapImage" />
    ...
    <bookstore:area id="map6" value="#{Book207}"
        onmouseover="resources/images/book_207.jpg"
        onmouseout="resources/images/book_all.jpg"
        targetImage="mapImage" />
</bookstore:map>
...```

The standard h:graphicImage tag associates an image (book_all.jpg) with an image map that is referenced in the usemap attribute value.

The custom bookstore:map tag that represents the custom component, MapComponent, specifies the image map, and contains a set of area tags. Each custom bookstore:area tag represents a custom AreaComponent and specifies a region of the image map.

On the page, the onmouseover and onmouseout attributes specify the image that is displayed when the user performs the actions described by the attributes. The custom renderer also renders an onclick attribute.

In the rendered HTML page, the onmouseover, onmouseout, and onclick attributes define which JavaScript code is executed when these events occur. When the user moves the mouse over a region, the onmouseover function associated with the region displays the map with that region highlighted. When the user moves the mouse out of a region, the onmouseout function redisplaysthe original image. When the user clicks a region, the onclick function sets the value of a hidden input tag to the ID of the selected area and submits the page.

When the custom renderer renders these attributes in HTML, it also renders the JavaScript code. The custom renderer also renders the entire onclick attribute rather than let the page author set it.

The custom renderer that renders the HTML map tag also renders a hidden input component that holds the current area. The server-side objects retrieve the value of the hidden input field and set the locale in the FacesContext instance according to which region was selected.
Creating and Using a Custom Converter

A JavaServer Faces converter class converts strings to objects and objects to strings as required. Several standard converters are provided by JavaServer Faces for this purpose. See for more information on these included converters.

As explained in “Conversion Model” on page 220, if the standard converters included with JavaServer Faces cannot perform the data conversion that you need, you can create a custom converter to perform this specialized conversion. This implementation, at a minimum, must define how to convert data both ways between the two views of the data described in “Conversion Model” on page 220.

All custom converters must implement the javax.faces.convert.Converter interface. This section explains how to implement this interface to perform a custom data conversion.

The Duke’s Bookstore case study uses a custom Converter implementation, located in tut-install/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/converters/CreditCardConverter.java, to convert the data entered in the Credit Card Number field on the bookcashier.xhtml page. It strips blanks and hyphens from the text string and formats it so that a blank space separates every four characters.

Another common use case for a custom converter is in a drop-down menu for a nonstandard object type. In the Duke’s Tutoring case study, the Student and Guardian entities require a custom converter so they can be converted to and from a UISelectItems input component.

Creating a Custom Converter

The CreditCardConverter custom converter class is created as follows:

```java
@FacesConverter("ccno")
public class CreditCardConverter implements Converter {
    ...
}
```

The @FacesConverter annotation registers the custom converter class as a converter with the name of ccno with the JavaServer Faces implementation. Alternatively, you can register the converter with entries in the application configuration resource file, as shown in “Registering a Custom Converter” on page 313.

To define how the data is converted from the presentation view to the model view, the Converter implementation must implement the getAsObject(FacesContext, UIComponent, String) method from the Converter interface. Here is the implementation of this method from CreditCardConverter:

```java
@Override
public Object getAsObject(FacesContext context, 
        UIComponent component, String newValue)
```
During the Apply Request Values phase, when the components' decode methods are processed, the JavaServer Faces implementation looks up the component's local value in the request and calls the getAsObject method. When calling this method, the JavaServer Faces implementation passes in the current FacesContext instance, the component whose data needs conversion, and the local value as a String. The method then writes the local value to a character array, trims the hyphens and blanks, adds the rest of the characters to a String, and returns the String.

To define how the data is converted from the model view to the presentation view, the Converter implementation must implement the getAsString(FacesContext, UIComponent, Object) method from the Converter interface. Here is an implementation of this method:

```java
@Override
public String getAsString(FacesContext context, UIComponent component, Object value)
    throws ConverterException {
    String inputVal = null;
    if ( value == null ) {
        return null;
    }
    // value must be of a type that can be cast to a String.
    try {
        inputVal = (String)value;
    } catch (ClassCastException ce) {
        FacesMessage errMsg = new FacesMessage(CONVERSION_ERROR_MESSAGE_ID);
        FacesContext.getCurrentInstance().addMessage(null, errMsg);
        throw new ConverterException(errMsg.getSummary());
    }
    // insert spaces after every four characters for better
// readability if they are not already present.
char[] input = inputVal.toCharArray();
StringBuilder builder = new StringBuilder(input.length + 3);
for (int i = 0; i < input.length; ++i) {
    if ((i % 4) == 0 && i != 0) {
        if (input[i] != '' || input[i] != -') {
            builder.append(' ');
            // if there are any "-"s convert them to blanks.
        } else if (input[i] == '-') {
            builder.append(' ');
        }
    }
    builder.append(input[i]);
}
String convertedValue = builder.toString();
return convertedValue;

During the Render Response phase, in which the components’ encode methods are called, the JavaServer Faces implementation calls the getAsString method in order to generate the appropriate output. When the JavaServer Faces implementation calls this method, it passes in the current FacesContext, the UIComponent whose value needs to be converted, and the bean value to be converted. Because this converter does a String-to-String conversion, this method can cast the bean value to a String.

If the value cannot be converted to a String, the method throws an exception, passing an error message from the resource bundle that is registered with the application. "Registering Application Messages" on page 308 explains how to register custom error messages with the application.

If the value can be converted to a String, the method reads the String to a character array and loops through the array, adding a space after every four characters.

You can also create a custom converter with a @FacesConverter annotation that specifies the forClass attribute, as shown in the following example from the Duke's Tutoring case study:

@FacesConverter(forClass=Guardian.class)
public class GuardianConverter implements Converter { ...
Using a Custom Converter

To apply the data conversion performed by a custom converter to a particular component’s value, you must do one of the following:

- Reference the converter from the component tag’s converter attribute.
- Nest an f:converter tag inside the component’s tag and reference the custom converter from one of the f:converter tag’s attributes.

If you are using the component tag’s converter attribute, this attribute must reference the Converter implementation’s identifier or the fully-qualified class name of the converter. “Creating and Using a Custom Converter” on page 279 explains how to implement a custom converter.

The identifier for the credit card converter class is ccno, the value specified in the @FacesConverter annotation:

```java
@FacesConverter("ccno")
public class CreditCardConverter implements Converter {
    ...
}
```

Therefore, the CreditCardConverter instance can be registered on the ccno component as shown in the following example:

```html
<h:inputText id="ccno"
    size="19"
    converter="ccno"
    value="#{cashier.creditCardNumber}"
    required="true"
    requiredMessage="#{bundle.ReqCreditCard}">
    ...
</h:inputText>
```

By setting the converter attribute of a component’s tag to the converter’s identifier or its class name, you cause that component’s local value to be automatically converted according to the rules specified in the Converter implementation.

Instead of referencing the converter from the component tag’s converter attribute, you can reference the converter from an f:converter tag nested inside the component’s tag. To reference the custom converter using the f:converter tag, you do one of the following:

- Set the f:converter tag’s converterId attribute to the Converter implementation’s identifier defined in the @FacesConverter annotation or in the application configuration resource file. This method is shown in bookcashier.xhtml:

  ```html
  <h:inputText id="ccno"
      size="19"
      value="#{cashier.creditCardNumber}"
      required="true"
      requiredMessage="#{bundle.ReqCreditCard}">
      <f:converter converterId="ccno"/>
  ```
Creating and Using a Custom Validator

If the standard validators or Bean Validation don’t perform the validation checking you need, you can create a custom validator to validate user input. As explained in “Validation Model” on page 222, there are two ways to implement validation code:

- Implement a managed bean method that performs the validation.
- Provide an implementation of the javax.faces.validator.Validator interface to perform the validation.

“Writing a Method to Perform Validation” on page 202 explains how to implement a managed bean method to perform validation. The rest of this section explains how to implement the Validator interface.

If you choose to implement the Validator interface and you want to allow the page author to configure the validator’s attributes from the page, you also must specify a custom tag for registering the validator on a component.

If you prefer to configure the attributes in the Validator implementation, you can forgo specifying a custom tag and instead let the page author register the validator on a component using the f:validator tag, as described in “Using a Custom Validator” on page 287.
You can also create a managed bean property that accepts and returns the Validator implementation you create, as described in “Writing Properties Bound to Converters, Listeners, or Validators” on page 199. You can use the f:validate tag's binding attribute to bind the Validator implementation to the managed bean property.

Usually, you will want to display an error message when data fails validation. You need to store these error messages in a resource bundle.

After creating the resource bundle, you have two ways to make the messages available to the application. You can queue the error messages onto the FacesContext programmatically, or you can register the error messages in the application configuration resource file, as explained in “Registering Application Messages” on page 308.

For example, an e-commerce application might use a general-purpose custom validator called FormatValidator.java to validate input data against a format pattern that is specified in the custom validator tag. This validator would be used with a Credit Card Number field on a Facelets page. Here is the custom validator tag:

```xml
<mystore:formatValidator
   formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>
```

According to this validator, the data entered in the field must be one of the following:

- A 16-digit number with no spaces
- A 16-digit number with a space between every four digits
- A 16-digit number with hyphens between every four digits

The f:validateRegex tag makes a custom validator unnecessary in this situation. However, the rest of this section describes how this validator would be implemented and how to specify a custom tag so that the page author could register the validator on a component.

### Implementing the Validator Interface

A Validator implementation must contain a constructor, a set of accessor methods for any attributes on the tag, and a validate method, which overrides the validate method of the Validator interface.

The hypothetical FormatValidator class also defines accessor methods for setting the formatPatterns attribute, which specifies the acceptable format patterns for input into the fields. The setter method calls the parseFormatPatterns method, which separates the components of the pattern string into a string array, formatPatternsList.

```java
public String getFormatPatterns() {
    return (this.formatPatterns);
}
public void setFormatPatterns(String formatPatterns) {
    this.formatPatterns = formatPatterns;
    parseFormatPatterns();
}
```
In addition to defining accessor methods for the attributes, the class overrides the validate method of the Validator interface. This method validates the input and also accesses the custom error messages to be displayed when the String is invalid.

The validate method performs the actual validation of the data. It takes the FacesContext instance, the component whose data needs to be validated, and the value that needs to be validated. A validator can validate only data of a component that implements javax.faces.component.EditableValueHolder.

Here is an implementation of the validate method:

```java
@FacesValidator
public class FormatValidator implements Validator, StateHolder {
    ...
    public void validate(FacesContext context, UIComponent component,
                          Object toValidate) {
        boolean valid = false;
        String value = null;
        if ((context == null) || (component == null)) {
            throw new NullPointerException();
        }
        if (!(component instanceof UIInput)) {
            return;
        }
        if (null == formatPatternsList || null == toValidate) {
            return;
        }
        value = toValidate.toString();
        // validate the value against the list of valid patterns.
        Iterator patternIt = formatPatternsList.iterator();
        while (patternIt.hasNext()) {
            valid = isFormatValid((String)patternIt.next(), value);
            if (valid) {
                break;
            }
        }
        if ( !valid ) {
            FacesMessage errMsg =
                new FacesMessage(FORMAT_INVALID_MESSAGE_ID);
            FacesContext.getCurrentInstance().addMessage(null, errMsg);
            throw new ValidatorException(errMsg);
        }
    }
}
```

The @FacesValidator annotation registers the FormatValidator class as a validator with the JavaServer Faces implementation. The validate method gets the local value of the component and converts it to a String. It then iterates over the formatPatternsList list, which is the list of acceptable patterns that was parsed from the formatPatterns attribute of the custom validator tag.

While iterating over the list, this method checks the pattern of the component's local value against the patterns in the list. If the pattern of the local value does not match any pattern in the
list, this method generates an error message. It then creates a
javax.faces.application.FacesMessage and queues it on the FacesContext for display
during the Render Response phase, using a String that represents the key in the Properties
file:

    public static final String FORMAT_INVALID_MESSAGE_ID =
            "FormatInvalid";
    }

Finally, the method passes the message to the constructor of
javax.faces.validator.ValidatorException.

When the error message is displayed, the format pattern will be substituted for the {0} in the
error message, which, in English, is as follows:

Input must match one of the following patterns: {0}

You may wish to save and restore state for your validator, although state saving is not usually
necessary. To do so, you will need to implement the StateHolder interface in addition to the
Validator interface. To implement StateHolder, you would need to implement its four
methods: saveState(FacesContext), restoreState(FacesContext, Object), isTransient,
and setTransient(boolean). See “Saving and Restoring State” on page 269 for more
information.

**Specifying a Custom Tag**

If you implemented a Validator interface rather than implementing a managed bean method
that performs the validation, you need to do one of the following:

- Allow the page author to specify the Validator implementation to use with the
  f:validator tag. In this case, the Validator implementation must define its own
  properties. “Using a Custom Validator” on page 287 explains how to use the f:validator
tag.

- Specify a custom tag that provides attributes for configuring the properties of the validator
  from the page.

To specify a custom tag, you need to add the tag to the tag library descriptor for the application,
bookstore.taglib.xml.

```xml
<tag>
    <tag-name>formatValidator</tag-name>
    <validator>
        <validator-id>formatValidator</validator-id>
        <validator-class>dukesbookstore.validators.FormatValidator</validator-class>
    </validator>
</tag>
```
The `tag-name` element defines the name of the tag as it must be used in a Facelets page. The `validator-id` element identifies the custom validator. The `validator-class` element wires the custom tag to its implementation class.

"Using a Custom Validator" on page 287 explains how to use the custom validator tag on the page.

**Using a Custom Validator**

To register a custom validator on a component, you must do one of the following:

- Nest the validator's custom tag inside the tag of the component whose value you want to be validated.
- Nest the standard `f:validator` tag within the tag of the component and reference the custom `Validator` implementation from the `f:validator` tag.

Here is a hypothetical custom `formatValidator` tag for the Credit Card Number field, nested within the `h:inputText` tag:

```xml
<h:inputText id="ccno" size="19"
   required="true">
   <mystore:formatValidator
      formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999" />
</h:inputText>
<h:message styleClass="validationMessage" for="ccno"/>
```

This tag validates the input of the `ccno` field against the patterns defined by the page author in the `formatPatterns` attribute.

You can use the same custom validator for any similar component by simply nesting the custom validator tag within the component tag.

If the application developer who created the custom validator prefers to configure the attributes in the `Validator` implementation rather than allow the page author to configure the attributes from the page, the developer will not create a custom tag for use with the validator.

In this case, the page author must nest the `f:validator` tag inside the tag of the component whose data needs to be validated. Then the page author needs to do one of the following:

- Set the `f:validator` tag's `validatorId` attribute to the ID of the validator that is defined in the application configuration resource file.
- Bind the custom `Validator` implementation to a managed bean property using the `f:validator` tag's `binding` attribute, as described in "Binding Converters, Listeners, and Validators to Managed Bean Properties" on page 293.
The following tag registers a hypothetical validator on a component using a validator tag and references the ID of the validator:

```xml
<h:inputText id="name" value="#{CustomerBean.name}"
           size="10" ... >
  <f:validator validatorId="customValidator" />
...  
</h:inputText>
```

**Binding Component Values and Instances to Managed Bean Properties**

A component tag can wire its data to a managed bean by one of the following methods:

- Binding its component’s value to a bean property
- Binding its component’s instance to a bean property

To bind a component’s value to a managed bean property, a component tag’s value attribute uses a EL value expression. To bind a component instance to a bean property, a component tag’s binding attribute uses a value expression.

When a component instance is bound to a managed bean property, the property holds the component’s local value. Conversely, when a component’s value is bound to a managed bean property, the property holds the value stored in the managed bean. This value is updated with the local value during the Update Model Values phase of the lifecycle. There are advantages to both of these methods.

Binding a component instance to a bean property has these advantages:

- The managed bean can programmatically modify component attributes.
- The managed bean can instantiate components rather than let the page author do so.

Binding a component’s value to a bean property has these advantages:

- The page author has more control over the component attributes.
- The managed bean has no dependencies on the JavaServer Faces API (such as the component classes), allowing for greater separation of the presentation layer from the model layer.
- The JavaServer Faces implementation can perform conversions on the data based on the type of the bean property without the developer needing to apply a converter.

In most situations, you will bind a component’s value rather than its instance to a bean property. You’ll need to use a component binding only when you need to change one of the component’s attributes dynamically. For example, if an application renders a component only under certain conditions, it can set the component’s rendered property accordingly by accessing the property to which the component is bound.
When referencing the property using the component tag's `value` attribute, you need to use the proper syntax. For example, suppose a managed bean called `MyBean` has this `int` property:

```java
protected int currentOption = null;
public int getCurrentOption() {...}
public void setCurrentOption(int option){...}
```

The `value` attribute that references this property must have this value-binding expression:

```el
#{myBean.currentOption}
```

In addition to binding a component's value to a bean property, the `value` attribute can specify a literal value or can map the component's data to any primitive (such as `int`), structure (such as an array), or collection (such as a list), independent of a JavaBeans component. Table 13–3 lists some example value-binding expressions that you can use with the `value` attribute.

<table>
<thead>
<tr>
<th>Value Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cart.numberOfItems &gt; 0</code></td>
<td>A Boolean</td>
</tr>
<tr>
<td><code>initParam.quantity</code></td>
<td>A property initialized from a context initialization parameter</td>
</tr>
<tr>
<td><code>cashierBean.name</code></td>
<td>A bean property</td>
</tr>
<tr>
<td><code>books[3]</code></td>
<td>Value in an array</td>
</tr>
<tr>
<td><code>books[&quot;fiction&quot;]</code></td>
<td>Value in a collection</td>
</tr>
<tr>
<td><code>books[3].price</code></td>
<td>Property of an object in an array of objects</td>
</tr>
</tbody>
</table>

The next two sections explain how to use the `value` attribute to bind a component's value to a bean property or other data objects, and how to use the `binding` attribute to bind a component instance to a bean property.

**Binding a Component Value to a Property**

To bind a component's value to a managed bean property, you specify the name of the bean and the property using the `value` attribute.

This means that the first part of the EL value expression must match the name of the managed bean up to the first period (.), and the part of the value expression after the period must match the property of the managed bean.

For example, in the Duke's Bookstore case study, the `h: dataTable` tag in `bookcatalog.xhtml` sets the value of the component to the value of the `books` property of the stateless session bean `BookRequestBean`:
The value is obtained by calling the bean's `getBooks` method.

If you use the application configuration resource file to configure managed beans instead of defining them in managed bean classes, the name of the bean in the value expression must match the `managed-bean-name` element of the managed bean declaration up to the first period (.) in the expression. Similarly, the part of the value expression after the period must match the name specified in the corresponding `property-name` element in the application configuration resource file.

For example, consider this managed bean configuration, which configures the `ImageArea` bean corresponding to the top left book in the image map on the `index.html` page of the Duke’s Bookstore case study:

```xml
<managed-bean eager="true">
  ...
  <managed-bean-name> Book201 </managed-bean-name>
  <managed-bean-class> dukesh bookstore.model.ImageArea </managed-bean-class>
  <managed-bean-scope> application </managed-bean-scope>
  <managed-property>
    <property-name>shape</property-name>
    <value>rect</value>
  </managed-property>
  <managed-property>
    ...
    <property-name>alt</property-name>
    <value>Duke</value>
  </managed-property>
  ...
</managed-bean>
```

This example configures a bean called `Book201`, which has several properties, one of which is called `shape`. 
Although the `bookstore:area` tags on the `index.xhtml` page do not bind to an `ImageArea` property (they bind to the bean itself), you could refer to the property using a value expression from the `value` attribute of the component's tag:

```xml
<h:outputText value="#{Book201.shape}" />
```

See “Configuring Managed Beans” on page 300 for information on how to configure beans in the application configuration resource file.

**Binding a Component Value to an Implicit Object**

One external data source that a `value` attribute can refer to is an implicit object.

The `bookreceipt.xhtml` page of the Duke's Bookstore case study has a reference to an implicit object:

```xml
<h:outputFormat title="thanks"
  value="#{bundle.ThankYouParam}"
  <f:param value="#{sessionScope.name}"/>
</h:outputFormat>
```

This tag gets the name of the customer from the session scope and inserts it into the parameterized message at the key `ThankYouParam` from the resource bundle. For example, if the name of the customer is Gwen Canigetit, this tag will render:

Thank you, Gwen Canigetit, for purchasing your books from us.

Retrieving values from other implicit objects is done in a similar way to the example shown in this section. Table 13–4 lists the implicit objects to which a `value` attribute can refer. All of the implicit objects, except for the scope objects, are read-only and therefore should not be used as a value for a `UIInput` component.

**Table 13–4** Implicit Objects

<table>
<thead>
<tr>
<th>Implicit Object</th>
<th>What It Is</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>applicationScope</code></td>
<td>A Map of the application scope attribute values, keyed by attribute name</td>
</tr>
<tr>
<td><code>cookie</code></td>
<td>A Map of the cookie values for the current request, keyed by cookie name</td>
</tr>
<tr>
<td><code>facesContext</code></td>
<td>The <code>FacesContext</code> instance for the current request</td>
</tr>
<tr>
<td><code>header</code></td>
<td>A Map of HTTP header values for the current request, keyed by header name</td>
</tr>
<tr>
<td><code>headerValues</code></td>
<td>A Map of <code>String</code> arrays containing all the header values for HTTP headers in the current request, keyed by header name</td>
</tr>
<tr>
<td><code>initParam</code></td>
<td>A Map of the context initialization parameters for this web application</td>
</tr>
<tr>
<td><code>param</code></td>
<td>A Map of the request parameters for this request, keyed by parameter name</td>
</tr>
</tbody>
</table>
TABLE 13-4 Implicit Objects  (Continued)

<table>
<thead>
<tr>
<th>Implicit Object</th>
<th>What It Is</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramValues</td>
<td>A Map of String arrays containing all the parameter values for request parameters in the current request, keyed by parameter name</td>
</tr>
<tr>
<td>requestScope</td>
<td>A Map of the request attributes for this request, keyed by attribute name</td>
</tr>
<tr>
<td>sessionScope</td>
<td>A Map of the session attributes for this request, keyed by attribute name</td>
</tr>
<tr>
<td>view</td>
<td>The root UIComponent in the current component tree stored in the FacesRequest for this request</td>
</tr>
</tbody>
</table>

Binding a Component Instance to a Bean Property

A component instance can be bound to a bean property using a value expression with the binding attribute of the component's tag. You usually bind a component instance rather than its value to a bean property if the bean must dynamically change the component's attributes.

Here are two tags from the bookcashier.xhtml page that bind components to bean properties:

```.xhtml
<h:selectBooleanCheckbox id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
<h:outputLabel for="fanClub"
    rendered="false"
    binding="#{cashier.specialOfferText}"
    value="#{bundle.DukeFanClub}"/>
```

The `h:selectBooleanCheckbox` tag renders a check box and binds the `fanClub` UISelectBoolean component to the `specialOffer` property of the `cashier` bean. The `h:outputLabel` tag binds the component representing the check box's label to the `specialOfferText` property of the `cashier` bean. If the application's locale is English, the `h:outputLabel` tag renders:

I'd like to join the Duke Fan Club, free with my purchase of over $100

The `rendered` attributes of both tags are set to `false`, to prevent the check box and its label from being rendered. If the customer makes a large order and clicks the Submit button, the submit method of `CashierBean` sets both components' `rendered` properties to true, causing the check box and its label to be rendered.

These tags use component bindings rather than value bindings, because the managed bean must dynamically set the values of the components' `rendered` properties.

If the tags were to use value bindings instead of component bindings, the managed bean would not have direct access to the components, and would therefore require additional code to access the components from the `FacesContext` instance to change the components' `rendered` properties.
"Writing Properties Bound to Component Instances" on page 198 explains how to write the bean properties bound to the example components.

## Binding Converters, Listeners, and Validators to Managed Bean Properties

As described in "Adding Components to a Page Using HTML Tags" on page 140, a page author can bind converter, listener, and validator implementations to managed bean properties using the binding attributes of the tags that are used to register the implementations on components.

This technique has similar advantages to binding component instances to managed bean properties, as described in "Binding Component Values and Instances to Managed Bean Properties" on page 288. In particular, binding a converter, listener, or validator implementation to a managed bean property yields the following benefits:

- The managed bean can instantiate the implementation instead of allowing the page author to do so.
- The managed bean can programmatically modify the attributes of the implementation. In the case of a custom implementation, the only other way to modify the attributes outside of the implementation class would be to create a custom tag for it and require the page author to set the attribute values from the page.

Whether you are binding a converter, listener, or validator to a managed bean property, the process is the same for any of the implementations:

- Nest the converter, listener, or validator tag within an appropriate component tag.
- Make sure that the managed bean has a property that accepts and returns the converter, listener, or validator implementation class that you want to bind to the property.
- Reference the managed bean property using a value expression from the binding attribute of the converter, listener, or validator tag.

For example, say that you want to bind the standard `DateTime` converter to a managed bean property because you want to set the formatting pattern of the user's input in the managed bean rather than on the Facelets page. First, the page registers the converter onto the component by nesting the `f:convertDateTime` tag within the component tag.
Then, the page references the property with the binding attribute of the \texttt{f:convertDateTime} tag:

\begin{verbatim}
<h:inputText value="#{loginBean.birthDate}"
    <f:convertDateTime binding="#{loginBean.convertDate}" />
</h:inputText>
\end{verbatim}

The \texttt{convertDate} property would look something like this:

\begin{verbatim}
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
    ... return convertDate;
}
public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
\end{verbatim}

See “Writing Properties Bound to Converters, Listeners, or Validators” on page 199 for more information on writing managed bean properties for converter, listener, and validator implementations.
Configuring JavaServer Faces Applications

The process of building and deploying simple JavaServer Faces applications is described in earlier chapters of this tutorial. When you create large and complex applications, however, various additional configuration tasks are required. These tasks include the following:

- Registering managed beans with the application so that all parts of the application have access to them
- Configuring managed beans and model beans so that they are instantiated with the proper values when a page makes reference to them
- Defining navigation rules for each of the pages in the application so that the application has a smooth page flow, if non-default navigation is needed
- Packaging the application to include all the pages, resources, and other files so that the application can be deployed on any compliant container

The following topics are addressed here:

- “Using Annotations to Configure Managed Beans” on page 296
- “Application Configuration Resource File” on page 297
- “Configuring Managed Beans” on page 300
- “Registering Application Messages” on page 308
- “Using Default Validators” on page 311
- “Registering a Custom Validator” on page 312
- “Registering a Custom Converter” on page 313
- “Configuring Navigation Rules” on page 313
- “Registering a Custom Renderer with a Render Kit” on page 317
- “Registering a Custom Component” on page 319
- “Basic Requirements of a JavaServer Faces Application” on page 320
Using Annotationsto Configure Managed Beans

JavaServer Faces support for bean annotations is introduced in Chapter 4, “JavaServer Faces Technology.” Bean annotations can be used for configuring JavaServer Faces applications.

The `@ManagedBean` (`javax.faces.bean.ManagedBean`) annotation in a class automatically registers that class as a resource with the JavaServer Faces implementation. Such a registered managed bean does not need managed-bean configuration entries in the application configuration resource file.

An example of using the `@ManagedBean` annotation in a class is as follows:

```java
@ManagedBean
@SessionScoped
public class DukesBday{
    ...
}
```

The above code snippet shows a bean that is managed by the JavaServer Faces implementation and is available for the length of the session. You do not need to configure the managed bean instance in the `faces-config.xml` file. In effect, this is an alternative to the application configuration resource file approach and reduces the task of configuring managed beans.

You can also define the scope of the managed bean within the class file, as shown in the above example. You can annotate beans with request, session, application, or view scope.

All classes will be scanned for annotations at startup unless the `faces-config` element in the `faces-config.xml` file has the `metadata-complete` attribute set to `true`.

Annotations are also available for other artifacts, such as components, converters, validators, and renderers, to be used in place of application configuration resource file entries. These are discussed, along with registration of custom listeners, custom validators, and custom converters, in Chapter 13, “Creating Custom UI Components and Other Custom Objects.”

Using Managed Bean Scopes

You can use annotations to define the scope in which the bean will be stored. You can specify one of the following scopes for a bean class:

- **Application** (`@ApplicationScoped`): Application scope persists across all users’ interactions with a web application.
- **Session** (`@SessionScoped`): Session scope persists across multiple HTTP requests in a web application.
- **View** (`@ViewScoped`): View scope persists during a user’s interaction with a single page (view) of a web application.
- **Request** (`@RequestScoped`): Request scope persists during a single HTTP request in a web application.
None (@NoneScoped): Indicates a scope is not defined for the application.

Custom (@CustomScoped): A user-defined, nonstandard scope. Its value must be configured as a java.util.Map. Custom scopes are used infrequently.

You may want to use @NoneScoped when a managed bean references another managed bean. The second bean should not be in a scope (@NoneScoped) if it is supposed to be created only when it is referenced. If you define a bean as @NoneScoped, the bean is instantiated anew each time it is referenced, so it does not get saved in any scope.

If your managed bean is referenced by the binding attribute of a component tag, you should define the bean with a request scope. If you placed the bean in session or application scope instead, the bean would need to take precautions to ensure thread safety, because javax.faces.component.UIComponent instances each depend on running inside of a single thread.

If you are configuring a bean that allows attributes to be associated with the view, you can use the view scope. The attributes persist until the user has navigated to the next view.

**Eager Application-Scoped Beans**

Managed beans are lazily instantiated. That is, they are instantiated when a request is made from the application.

To force an application-scoped bean to be instantiated and placed in the application scope as soon as the application is started and before any request is made, the eager attribute of the managed bean should be set to true as shown in the following example:

```java
@ManagedBean(eager=true)
@ApplicationScoped
```

**Application Configuration Resource File**

JavaServer Faces technology provides a portable configuration format (as an XML document) for configuring application resources. One or more XML documents, called *application configuration resource files*, may use this format to register and configure objects and resources, and to define navigation rules for applications. An application configuration resource file is usually named *faces-config.xml*.

You need an application configuration resource file in the following cases:

- To specify configuration elements for your application that are not available through managed bean annotations, such as localized messages and navigation rules
- To override managed bean annotations when the application is deployed

The application configuration resource file must be valid against the XML schema located at [http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd](http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd).
In addition, each file must include the following information, in the following order:

- The XML version number, usually with an encoding attribute:
  ```xml
  <?xml version="1.0" encoding='UTF-8'?>
  ```
- A `faces-config` tag enclosing all the other declarations:
  ```xml
  <faces-config version="2.0" xmlns="http://java.sun.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                         http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
  ...
  </faces-config>
  ```

You can have more than one application configuration resource file for an application. The JavaServer Faces implementation finds the configuration file or files by looking for the following:

- A resource named `/META-INF/faces-config.xml` in any of the JAR files in the web application's `/WEB-INF/lib/` directory and in parent class loaders. If a resource with this name exists, it is loaded as a configuration resource. This method is practical for a packaged library containing some components and renderers. In addition, any file with a name that ends in `faces-config.xml` is also considered a configuration resource and is loaded as such.
- A context initialization parameter, `javax.faces.application.CONFIG_FILES`, in your web deployment descriptor file that specifies one or more (comma-delimited) paths to multiple configuration files for your web application. This method is most often used for enterprise-scale applications that delegate to separate groups the responsibility for maintaining the file for each portion of a big application.
- A resource named `faces-config.xml` in the `/WEB-INF/` directory of your application. Simple web applications make their configuration files available in this way.

To access the resources registered with the application, an application developer can use an instance of the `javax.faces.application.Application` class, which is automatically created for each application. The `Application` instance acts as a centralized factory for resources that are defined in the XML file.

When an application starts up, the JavaServer Faces implementation creates a single instance of the `Application` class and configures it with the information you provided in the application configuration resource file.

## Ordering of Application Configuration Resource Files

Because JavaServer Faces technology allows the use of multiple application configuration resource files stored in different locations, the order in which they are loaded by the implementation becomes important in certain situations (for example, when using application-level objects). This order can be defined through an `ordering` element and its subelements in the application configuration resource file itself. The ordering of application configuration resource files can be absolute or relative.
Absolute ordering is defined by an `absolute-ordering` element in the file. With absolute ordering, the user specifies the order in which application configuration resource files will be loaded. The following example shows an entry for absolute ordering:

File `my-faces-config.xml`:

```xml
<faces-config>
    <name>myJSF</name>
    <absolute-ordering>
        <name>A</name>
        <name>B</name>
        <name>C</name>
    </absolute-ordering>
</faces-config>
```

In this example, A, B, and C are different application configuration resource files and are to be loaded in the listed order.

If there is an `absolute-ordering` element in the file, only the files listed by the subelement `name` are processed. To process any other application configuration resource files, an `others` subelement is required. In the absence of the `others` subelement, all other unlisted files will be ignored at load time.

Relative ordering is defined by an `ordering` element and its subelements `before` and `after`. With relative ordering, the order in which application configuration resource files will be loaded is calculated by considering ordering entries from the different files. The following example shows some of these considerations. In the following example, `config-A`, `config-B`, and `config-C` are different application configuration resource files.

File `config-A` contains the following elements:

```xml
<faces-config>
    <name>config-A</name>
    <ordering>
        <before>
            <name>config-B</name>
        </before>
    </ordering>
</faces-config>
```

File `config-B` (not shown here) does not contain any `ordering` elements.

File `config-C` contains the following elements:

```xml
<faces-config>
    <name>config-C</name>
    <ordering>
        <after>
            <name>config-B</name>
        </after>
    </ordering>
</faces-config>
```
Based on the before subelement entry, file config-A will be loaded before the config-B file. Based on the after subelement entry, file config-C will be loaded after the config-B file.

In addition, a subelement `others` can also be nested within the before and after subelements. If the others element is present, the specified file may receive highest or lowest preference among both listed and unlisted configuration files.

If an ordering element is not present in an application configuration file, then that file will be loaded after all the files that contain ordering elements.

### Configuring Managed Beans

When a page references a managed bean for the first time, the JavaServer Faces implementation initializes it based on a `@ManagedBean` annotation in the bean class (or a `@Named` annotation for CDI managed beans) or according to its configuration in the application configuration resource file. For information on using annotations to initialize beans, see “Using Annotations to Configure Managed Beans” on page 296.

You can use either annotations or the application configuration resource file to instantiate managed beans that are used in a JavaServer Faces application and to store them in scope. The managed bean creation facility is configured in the application configuration resource file using `managed-bean` XML elements to define each bean. This file is processed at application startup time. For information on using this facility, see “Using the `managed-bean` Element” on page 300.

With the managed bean creation facility, you can:

- Create beans in one centralized file that is available to the entire application, rather than conditionally instantiate beans throughout the application
- Customize a bean’s properties without any additional code
- Customize a bean’s property values directly from within the configuration file so that it is initialized with these values when it is created
- Using `value` elements, set a property of one managed bean to be the result of evaluating another value expression

This section shows you how to initialize beans using the managed bean creation facility. See “Writing Bean Properties” on page 192 and “Writing Managed Bean Methods” on page 200 for information on programming managed beans.

### Using the `managed-bean` Element

A managed bean is initiated in the application configuration resource file using a `managed-bean` element, which represents an instance of a bean class that must exist in the application. At runtime, the JavaServer Faces implementation processes the `managed-bean` element. If a page
references the bean, and if no bean instance exists, the JavaServer Faces implementation instantiates the bean as specified by the element configuration.

Here is an example managed bean configuration from the Duke’s Bookstore case study:

```xml
<managed-bean eager="true">
    <managed-bean-name> Book201 </managed-bean-name>
    <managed-bean-class> dukesbookstore.model.ImageArea </managed-bean-class>
    <managed-bean-scope> application </managed-bean-scope>
    <managed-property>
        <property-name> shape </property-name>
        <value> rect </value>
    </managed-property>
    <managed-property>
        <property-name> alt </property-name>
        <value> Duke </value>
    </managed-property>
    <managed-property>
        <property-name> coords </property-name>
        <value> 67, 23, 212, 268 </value>
    </managed-property>
</managed-bean>
```

Using NetBeans IDE, you can add a managed bean declaration by doing the following:

1. After opening your project in NetBeans IDE, expand the project node in the Projects pane.
2. Expand the Web Pages and WEB-INF nodes of the project node.
3. If there is no `faces-config.xml` in the project, create one as follows:
   a. From the File menu, choose New File.
   b. In the New File wizard, select the JavaServer Faces category, then select JSF Faces Configuration and click Next.
   c. On the Name and Location page, change the name and location of the file if necessary. The default file name is `faces-config.xml`.
   d. Click Finish.
4. Double-click `faces-config.xml` if the file is not already open.
5. After `faces-config.xml` opens in the editor pane, select XML from the sub-tab panel options.
6. Right-click in the editor pane.
7. From the Insert menu, choose Managed Bean.
8. In the Add Managed Bean dialog box:
   a. Type the display name of the bean in the Bean Name field.
   b. Click Browse to locate the bean's class.
9. In the Browse Class dialog box:
   a. Start typing the name of the class you are looking for in the Class Name field. While you are typing, the dialog will show the matching classes.
b. Select the class from the Matching Classes box.
   c. Click OK.

10. In the Add Managed Bean dialog box:
   a. Select the bean’s scope from the Scope menu.
   b. Click Add.

The preceding steps will add the managed-bean element and three elements inside of that element: a managed-bean-name element, a managed-bean-class element, and a managed-bean-scope element. You will need to edit the XML of the configuration file directly to further configure this managed bean.

The managed-bean-name element defines the key under which the bean will be stored in a scope. For a component’s value to map to this bean, the component tag’s value attribute must match the managed-bean-name up to the first period.

The managed-bean-class element defines the fully qualified name of the JavaBeans component class used to instantiate the bean.

The managed-bean element can contain zero or more managed-property elements, each corresponding to a property defined in the bean class. These elements are used to initialize the values of the bean properties. If you don’t want a particular property initialized with a value when the bean is instantiated, do not include a managed-property definition for it in your application configuration resource file.

If a managed-bean element does not contain other managed-bean elements, it can contain one map-entries element or list-entries element. The map-entries element configures a set of beans that are instances of Map. The list-entries element configures a set of beans that are instances of List.

In the following example, the newsletters managed bean, representing a UISelectItems component, is configured as an ArrayList that represents a set of SelectItem objects. Each SelectItem object is in turn configured as a managed bean with properties:

```xml
<managed-bean>
  <managed-bean-name>newsletters</managed-bean-name>
  <managed-bean-class>java.util.ArrayList</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <list-entries>
    <value-class>javax.faces.model.SelectItem</value-class>
    <value>#{newsletter0}</value>
    <value>#{newsletter1}</value>
    <value>#{newsletter2}</value>
    <value>#{newsletter3}</value>
  </list-entries>
</managed-bean>
<managed-bean>
  <managed-bean-name>newsletter0</managed-bean-name>
  <managed-bean-class>javax.faces.model.SelectItem</managed-bean-class>
```

Configuring Managed Beans

The JavaEE6Tutorial • January 2013

302
This approach may be useful for quick-and-dirty creation of selection item lists, before a development team has had time to create such lists from the database. Note that each of the individual newsletter beans has a managed-bean-scope setting of none, so that they will not themselves be placed into any scope.

See "Initializing Array and List Properties" on page 306 for more information on configuring collections as beans.

To map to a property defined by a managed-property element, you must ensure that the part of a component tag’s value expression after the period matches the managed-property element’s property-name element. In the earlier example, the maximum property is initialized with the value 10. The following section, "Initializing Properties Using the managed-property Element" on page 303, explains in more detail how to use the managed-property element. See "Initializing Managed Bean Properties" on page 307 for an example of initializing a managed bean property.

### Initializing Properties Using the managed-property Element

A managed-property element must contain a property-name element, which must match the name of the corresponding property in the bean. A managed-property element must also contain one of a set of elements that defines the value of the property. This value must be of the same type as that defined for the property in the corresponding bean. Which element you use to define the value depends on the type of the property defined in the bean. Table 14–1 lists all the elements that are used to initialize a value.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value It Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td>list-entries</td>
<td>Defines the values in a list</td>
</tr>
<tr>
<td>map-entries</td>
<td>Defines the values of a map</td>
</tr>
<tr>
<td>null-value</td>
<td>Explicitly sets the property to null</td>
</tr>
<tr>
<td>value</td>
<td>Defines a single value, such as a String, int, or JavaServer Faces EL expression</td>
</tr>
</tbody>
</table>

Chapter 14 • Configuring JavaServer Faces Applications 303
Using the managed-bean Element on page 300 includes an example of initializing an int property (a primitive type) using the value subelement. You also use the value subelement to initialize String and other reference types. The rest of this section describes how to use the value subelement and other subelements to initialize properties of Java Enum types, Map, array, and Collection, as well as initialization parameters.

Referencing a Java Enum Type

A managed bean property can also be a Java Enum type (see http://docs.oracle.com/javase/6/docs/api/java/lang/Enum.html). In this case, the value element of the managed-property element must be a String that matches one of the String constants of the Enum. In other words, the String must be one of the valid values that can be returned if you were to call valueOf(Class, String) on enum, where Class is the Enum class and String is the contents of the value subelement. For example, suppose the managed bean property is the following:

```java
public enum Suit { Hearts, Spades, Diamonds, Clubs }

public Suit getSuit() { ... return Suit.Hearts; }
```

Assuming you want to configure this property in the application configuration resource file, the corresponding managed-property element looks like this:

```xml
<managed-property>
  <property-name>Suit</property-name>
  <value>Hearts</value>
</managed-property>
```

When the system encounters this property, it iterates over each of the members of the enum and calls toString() on each member until it finds one that is exactly equal to the value from the value element.

Referencing a Context Initialization Parameter

Another powerful feature of the managed bean creation facility is the ability to reference implicit objects from a managed bean property.

Suppose you have a page that accepts data from a customer, including the customer's address. Suppose also that most of your customers live in a particular area code. You can make the area code component render this area code by saving it in an implicit object and referencing it when the page is rendered.

You can save the area code as an initial default value in the context initParam implicit object by adding a context parameter to your web application and setting its value in the deployment descriptor. For example, to set a context parameter called defaultAreaCode to 650, add a context-param element to the deployment descriptor, and give the parameter the name defaultAreaCode and the value 650.

Next, you write a managed-bean declaration that configures a property that references the parameter:
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>CustomerBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>areaCode</property-name>
    <value>#{initParam.defaultAreaCode}</value>
  </managed-property>
  ...
</managed-bean>

To access the area code at the time the page is rendered, refer to the property from the area component tag's value attribute:

```html
<h:inputText id=area value="#{customer.areaCode}"/>
```

Values are retrieved from other implicit objects in a similar way.

**Initializing Map Properties**

The map-entries element is used to initialize the values of a bean property with a type of java.util.Map if the map-entries element is used within a managed-property element. A map-entries element contains an optional key-class element, an optional value-class element, and zero or more map-entry elements.

Each of the map-entry elements must contain a key element and either a null-value or value element. Here is an example that uses the map-entries element:

```xml
<managed-bean>
  ...
  <managed-property>
    <property-name>prices</property-name>
    <map-entries>
      <map-entry>
        <key>My Early Years: Growing Up on *7</key>
        <value>30.75</value>
      </map-entry>
      <map-entry>
        <key>Web Servers for Fun and Profit</key>
        <value>40.75</value>
      </map-entry>
    </map-entries>
  </managed-property>
</managed-bean>
```

The map created from this map-entries tag contains two entries. By default, all the keys and values are converted to String. If you want to specify a different type for the keys in the map, embed the key-class element just inside the map-entries element:

```xml
<map-entries>
  <key-class>java.math.BigDecimal</key-class>
  ...
</map-entries>
```
This declaration will convert all the keys into java.math.BigDecimal. Of course, you must make sure the keys can be converted to the type you specify. The key from the example in this section cannot be converted to a BigDecimal, because it is a String.

If you want to specify a different type for all the values in the map, include the value-class element after the key-class element:

```xml
<map-entries>
  <key-class>int</key-class>
  <value-class>java.math.BigDecimal</value-class>
  ...
</map-entries>
```

Note that this tag sets only the type of all the value subelements.

Each map-entry in the preceding example includes a value subelement. The value subelement defines a single value, which will be converted to the type specified in the bean.

Instead of using a map-entries element, it is also possible to assign the entire map using a value element that specifies a map-typed expression.

### Initializing Array and List Properties

The list-entries element is used to initialize the values of an array or List property. Each individual value of the array or List is initialized using a value or null-value element. Here is an example:

```xml
<managed-bean>
  ...
  <managed-property>
    <property-name>books</property-name>
    <list-entries>
      <value-class>java.lang.String</value-class>
      <value>Web Servers for Fun and Profit</value>
      <value>${myBooks.bookId[3]}</value>
      <null-value/>
    </list-entries>
  </managed-property>
</managed-bean>
```

This example initializes an array or a List. The type of the corresponding property in the bean determines which data structure is created. The list-entries element defines the list of values in the array or List. The value element specifies a single value in the array or List and can reference a property in another bean. The null-value element will cause the setBooks method to be called with an argument of null. A null property cannot be specified for a property whose data type is a Java primitive, such as int or boolean.
Initializing Managed Bean Properties

Sometimes you might want to create a bean that also references other managed beans so you can construct a graph or a tree of beans. For example, suppose you want to create a bean representing a customer's information, including the mailing address and street address, each of which is also a bean. The following managed-bean declarations create a CustomerBean instance that has two AddressBean properties: one representing the mailing address, and the other representing the street address. This declaration results in a tree of beans with CustomerBean as its root and the two AddressBean objects as children.

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>com.example.mybeans.CustomerBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>mailingAddress</property-name>
    <value>#{addressBean}</value>
  </managed-property>
  <managed-property>
    <property-name>streetAddress</property-name>
    <value>#{addressBean}</value>
  </managed-property>
  <managed-property>
    <property-name>customerType</property-name>
    <value>New</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>addressBean</managed-bean-name>
  <managed-bean-class>com.example.mybeans.AddressBean</managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
  <managed-property>
    <property-name>street</property-name>
    <null-value/>
  </managed-property>
  ...
</managed-bean>
```

The first CustomerBean declaration (with the managed-bean-name of customer) creates a CustomerBean in request scope. This bean has two properties, mailingAddress and streetAddress. These properties use the value element to reference a bean named AddressBean.

The second managed bean declaration defines an AddressBean, but does not create it, because its managed-bean-scope element defines a scope of none. Recall that a scope of none means that the bean is created only when something else references it. Because both the mailingAddress and the streetAddress properties reference addressBean using the value element, two instances of AddressBean are created when CustomerBean is created.
When you create an object that points to other objects, do not try to point to an object with a shorter life span, because it might be impossible to recover that scope’s resources when it goes away. A session-scoped object, for example, cannot point to a request-scoped object. And objects with none scope have no effective life span managed by the framework, so they can point only to other none-scoped objects. Table 14–2 outlines all of the allowed connections.

### TABLE 14–2 Allowable Connections Between Scoped Objects

<table>
<thead>
<tr>
<th>An Object of This Scope</th>
<th>May Point to an Object of This Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>application</td>
<td>none, application</td>
</tr>
<tr>
<td>session</td>
<td>none, application, session</td>
</tr>
<tr>
<td>request</td>
<td>none, application, session, request, view</td>
</tr>
<tr>
<td>view</td>
<td>none, application, session, view</td>
</tr>
</tbody>
</table>

Be sure not to allow cyclical references between objects. For example, neither of the AddressBean objects in the preceding example should point back to the CustomerBean object, because CustomerBean already points to the AddressBean objects.

**Initializing Maps and Lists**

In addition to configuring `Map` and `List` properties, you can also configure a `Map` and a `List` directly so that you can reference them from a tag rather than referencing a property that wraps a `Map` or a `List`.

**Registering Application Messages**

Application messages can include any strings displayed to the user, as well as custom error messages (which are displayed by the `message` and `messages` tags) for your custom converters or validators. To make messages available at application startup time, do one of the following:

- Queue an individual message onto the `javax.faces.context.FacesContext` instance programmatically, as described in “Using `FacesMessage` to Create a Message” on page 310
- Register all the messages with your application using the application configuration resource file
Here is the section of the `faces-config.xml` file that registers the messages for the Duke's Bookstore case study application:

```xml
<application>
  <resource-bundle>
    <base-name>dukesbookstore.web.messages.Messages</base-name>
  </resource-bundle>
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>es</supported-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>fr</supported-locale>
  </locale-config>
</application>
```

This set of elements causes the application to be populated with the messages that are contained in the specified resource bundle.

The `resource-bundle` element represents a set of localized messages. It must contain the fully qualified path to the resource bundle containing the localized messages (in this case, `dukestutoring.web.messages.Messages`). The `var` element defines the EL name by which page authors refer to the resource bundle.

The `locale-config` element lists the default locale and the other supported locales. The `locale-config` element enables the system to find the correct locale based on the browser’s language settings.


To access the localized message, the application developer merely references the key of the message from the resource bundle.

You can pull localized text into an `alt` tag for a graphic image, as in the following example:

```xml
<h:graphicImage id="mapImage"
  name="book_all.jpg"
  library="images"
  alt="#{bundle.ChooseBook}"
  usemap="#bookMap"/>
```

The `alt` attribute can accept value expressions. In this case, the `alt` attribute refers to localized text that will be included in the alternative text of the image rendered by this tag.
Using FacesMessage to Create a Message

Instead of registering messages in the application configuration resource file, you can access the java.util.ResourceBundle directly from managed bean code. The code snippet below locates an email error message:

```java
String message = "";
...
message = ExampleBean.loadErrorMessage(context,
    ExampleBean.EX_RESOURCE_BUNDLE_NAME,
    "EMailError");
context.addMessage(toValidate.getClientId(context),
    new FacesMessage(message));
```

These lines call the bean’s loadErrorMessage method to get the message from the ResourceBundle. Here is the loadErrorMessage method:

```java
public static String loadErrorMessage(FacesContext context,
    String basename, String key) {
    if ( bundle == null ) {
        try {
            bundle = ResourceBundle.getBundle(basename,
                context.getViewRoot().getLocale());
        } catch (Exception e) {
            return null;
        }
    }
    return bundle.getString(key);
}
```

Referencing Error Messages

A JavaServer Faces page uses the message or messages tags to access error messages, as explained in “Displaying Error Messages with the h:message and h:messages Tags” on page 164.

The error messages these tags access include:

- The standard error messages that accompany the standard converters and validators that ship with the API. See Section 2.5.2.4 of the JavaServer Faces specification for a complete list of standard error messages.
- Custom error messages contained in resource bundles registered with the application by the application architect using the resource-bundle element in the configuration file.

When a converter or validator is registered on an input component, the appropriate error message is automatically queued on the component.
A page author can override the error messages queued on a component by using the following attributes of the component's tag:

- converterMessage: References the error message to display when the data on the enclosing component cannot be converted by the converter registered on this component.
- requiredMessage: References the error message to display when no value has been entered into the enclosing component.
- validatorMessage: References the error message to display when the data on the enclosing component cannot be validated by the validator registered on this component.

All three attributes are enabled to take literal values and value expressions. If an attribute uses a value expression, this expression references the error message in a resource bundle. This resource bundle must be made available to the application in one of the following ways:

- By the application architect using the resource-bundle element in the configuration file
- By the page author using the f:loadBundle tag

Conversely, the resource-bundle element must be used to make available to the application those resource bundles containing custom error messages that are queued on the component as a result of a custom converter or validator being registered on the component.

The following tags show how to specify the requiredMessage attribute using a value expression to reference an error message:

```xml
<h:inputText id="ccno" size="19"
    required="true"
    requiredMessage="#{customMessages.ReqMessage}"
>
    ...
</h:inputText>
<h:message styleClass="error-message" for="ccno"/>
```

The value expression used by requiredMessage in this example references the error message with the ReqMessage key in the resource bundle, customMessages.

This message replaces the corresponding message queued on the component and will display wherever the message or messages tag is placed on the page.

### Using Default Validators

In addition to the validators you declare on the components, you can also specify zero or more default validators in the application configuration resource file. The default validator applies to all `javax.faces.component.UIInput` instances in a view or component tree and is appended after the local defined validators. Here is an example of a default validator registered in the application configuration resource file:

```xml
<faces-config>
    <application>
```

Chapter 14 • Configuring JavaServer Faces Applications
Registering a Custom Validator

If the application developer provides an implementation of the `javax.faces.validator.Validator` interface to perform validation, you must register this custom validator either by using the `@FacesValidator` annotation, as described in "Implementing the Validator Interface" on page 284, or by using the validator XML element in the application configuration resource file:

```xml
<validator>
  ...
  <validator-id>FormatValidator</validator-id>
  <validator-class>
    myapplication.validators.FormatValidator
  </validator-class>
  <attribute>
    ...
    <attribute-name>formatPatterns</attribute-name>
    <attribute-class>java.lang.String</attribute-class>
  </attribute>
</validator>
```

Attributes specified in a validator tag override any settings in the `@FacesValidator` annotation.

The `validator-id` and `validator-class` elements are required subelements. The `validator-id` element represents the identifier under which the `Validator` class should be registered. This ID is used by the tag class corresponding to the custom validator tag.

The `validator-class` element represents the fully qualified class name of the `Validator` class.

The `attribute` element identifies an attribute associated with the `Validator` implementation. It has required `attribute-name` and `attribute-class` subelements. The `attribute-name` element refers to the name of the attribute as it appears in the validator tag. The `attribute-class` element identifies the Java type of the value associated with the attribute.

"Creating and Using a Custom Validator" on page 283 explains how to implement the `Validator` interface.

"Using a Custom Validator" on page 287 explains how to reference the validator from the page.
Registering a Custom Converter

As is the case with a custom validator, if the application developer creates a custom converter, you must register it with the application either by using the @FacesConverter annotation, as described in “Creating a Custom Converter” on page 279, or by using the converter XML element in the application configuration resource file. Here is a hypothetical converter configuration for CreditCardConverter from the Duke’s Bookstore case study:

```xml
<converter>
  <description>
    Converter for credit card numbers that normalizes the input to a standard format
  </description>
  <converter-id>CreditCardConverter</converter-id>
  <converter-class>
    dukesbookstore.converters.CreditCardConverter
  </converter-class>
</converter>
```

Attributes specified in a converter tag override any settings in the @FacesConverter annotation.

The converter element represents a javax.faces.convert.Converter implementation and contains required converter-id and converter-class elements.

The converter-id element identifies an ID that is used by the converter attribute of a UI component tag to apply the converter to the component’s data. “Using a Custom Converter” on page 282 includes an example of referencing the custom converter from a component tag.

The converter-class element identifies the Converter implementation.

“Creating and Using a Custom Converter” on page 279 explains how to create a custom converter.

Configuring Navigation Rules

Navigation between different pages of a JavaServer Faces application, such as choosing the next page to be displayed after a button or hyperlink component is clicked, is defined by a set of rules. Navigation rules can be implicit, or they can be explicitly defined in the application configuration resource file. For more information on implicit navigation rules, see "Implicit Navigation Rules" on page 316.

Each navigation rule specifies how to navigate from one page to another page or set of pages. The JavaServer Faces implementation chooses the proper navigation rule according to which page is currently displayed.
After the proper navigation rule is selected, the choice of which page to access next from the current page depends on two factors:

- The action method invoked when the component was clicked
- The logical outcome referenced by the component’s tag or returned from the action method

The outcome can be anything the developer chooses, but Table 14–3 lists some outcomes commonly used in web applications.

<table>
<thead>
<tr>
<th>Outcome</th>
<th>What It Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Everything worked. Go on to the next page.</td>
</tr>
<tr>
<td>failure</td>
<td>Something is wrong. Go on to an error page.</td>
</tr>
<tr>
<td>login</td>
<td>The user needs to log in first. Go on to the login page.</td>
</tr>
<tr>
<td>no results</td>
<td>The search did not find anything. Go to the search page again.</td>
</tr>
</tbody>
</table>

Usually, the action method performs some processing on the form data of the current page. For example, the method might check whether the user name and password entered in the form match the user name and password on file. If they match, the method returns the outcome success. Otherwise, it returns the outcome failure. As this example demonstrates, both the method used to process the action and the outcome returned are necessary to determine the correct page to access.

Here is a navigation rule that could be used with the example just described:

```xml
<navigation-rule>
  <from-view-id>/login.xhtml</from-view-id>
  <navigation-case>
    <from-action>#{LoginForm.login}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/storefront.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-action>#{LoginForm.logon}</from-action>
    <from-outcome>failure</from-outcome>
    <to-view-id>/logon.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

This navigation rule defines the possible ways to navigate from login.xhtml. Each navigation-case element defines one possible navigation path from login.xhtml. The first navigation-case says that if LoginForm.login returns an outcome of success, then storefront.xhtml will be accessed. The second navigation-case says that login.xhtml will be re-rendered if LoginForm.login returns failure.
The configuration of an application's page flow consists of a set of navigation rules. Each rule is defined by the navigation-rule element in the faces-config.xml file.

Each navigation-rule element corresponds to one component tree identifier defined by the optional from-view-id element. This means that each rule defines all the possible ways to navigate from one particular page in the application. If there is no from-view-id element, the navigation rules defined in the navigation-rule element apply to all the pages in the application. The from-view-id element also allows wildcard matching patterns. For example, this from-view-id element says that the navigation rule applies to all the pages in the books directory:

<from-view-id>/books/*</from-view-id>

A navigation-rule element can contain zero or more navigation-case elements. The navigation-case element defines a set of matching criteria. When these criteria are satisfied, the application will navigate to the page defined by the to-view-id element contained in the same navigation-case element.

The navigation criteria are defined by optional from-outcome and from-action elements. The from-outcome element defines a logical outcome, such as success. The from-action element uses a method expression to refer to an action method that returns a String, which is the logical outcome. The method performs some logic to determine the outcome and returns the outcome.

The navigation-case elements are checked against the outcome and the method expression in this order:

1. Cases specifying both a from-outcome value and a from-action value. Both of these elements can be used if the action method returns different outcomes depending on the result of the processing it performs.
2. Cases specifying only a from-outcome value. The from-outcome element must match either the outcome defined by the action attribute of the javax.faces.component.UICommand component or the outcome returned by the method referred to by the UICommand component.
3. Cases specifying only a from-action value. This value must match the action expression specified by the component tag.

When any of these cases is matched, the component tree defined by the to-view-id element will be selected for rendering.

▼ To Configure a Navigation Rule

Using NetBeans IDE, you can configure a navigation rule by doing the following.

1. **After opening your project in NetBeans IDE, expand the project node in the Projects pane.**

2. **Expand the Web Pages and WEB-INF nodes of the project node.**
Double-click faces-config.xml.

After faces-config.xml opens in the editor pane, right-click in the editor pane.

From the Insert menu, choose Navigation Rule.

In the Add Navigation Rule dialog:

a. Enter or browse for the page that represents the starting view for this navigation rule.

b. Click Add.

Right-click again in the editor pane.

From the Insert menu, choose Navigation Case.

In the Add Navigation Case dialog box:

a. From the From View menu, choose the page that represents the starting view for the navigation rule (from Step 6a).

b. (optional) In the From Action field, type the action method invoked when the component that triggered navigation is activated.

c. (optional) In the From Outcome field, enter the logical outcome string that the activated component references from its action attribute.

d. From the To View menu, choose or browse for the page that will be opened if this navigation case is selected by the navigation system.

e. Click Add.

See Also

"Referencing a Method That Performs Navigation" on page 187 explains how to use a component tag’s action attribute to point to an action method. "Writing a Method to Handle Navigation" on page 201 explains how to write an action method.

Implicit Navigation Rules

JavaServer Faces technology supports implicit navigation rules for Facelets applications. Implicit navigation applies when navigation-rules are not configured in the application configuration resource files.
When you add a component such as a `commandButton` in a page, and assign another page as the value for its `action` property, the default navigation handler will try to match a suitable page within the application implicitly.

```xml
<h:commandButton value="submit" action="response"/>
```

In the above example, the default navigation handler will try to locate a page named `response.xhtml` within the application and navigate to it.

### Registering a Custom Renderer with a Render Kit

When the application developer creates a custom renderer, as described in "Delegating Rendering to a Renderer" on page 270, you must register it using the appropriate render kit. Because the image map application implements an HTML image map, the `AreaRenderer` and `MapRenderer` classes in the Duke's Bookstore case study should be registered using the HTML render kit.

You register the renderer either by using the `@FacesRenderer` annotation, as described in "Creating the Renderer Class" on page 270, or by using the `render-kit` element of the application configuration resource file. Here is a hypothetical configuration of `AreaRenderer`:

```xml
<render-kit>
  <renderer>
    <component-family>Area</component-family>
    <renderer-type>DemoArea</renderer-type>
    <renderer-class>dukesbookstore.renderers.AreaRenderer</renderer-class>
    <attribute>
      <attribute-name>onmouseout</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    <attribute>
      <attribute-name>onmouseover</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    <attribute>
      <attribute-name>styleClass</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
  </renderer>
  ...
</render-kit>
```

Attributes specified in a renderer tag override any settings in the `@FacesRenderer` annotation.

The `render-kit` element represents a `javax.faces.render.RenderKit` implementation. If no `render-kit-id` is specified, the default HTML render kit is assumed. The `renderer` element represents a `javax.faces.render.Renderer` implementation. By nesting the `renderer` element inside the `render-kit` element, you are registering the renderer with the `RenderKit` implementation associated with the `render-kit` element.
The `renderer-class` is the fully qualified class name of the `Renderer`.

The `component-family` and `renderer-type` elements are used by a component to find renderers that can render it. The `component-family` identifier must match that returned by the component class's `getFamily` method. The component family represents a component or set of components that a particular renderer can render. The `renderer-type` must match that returned by the `getRendererType` method of the tag handler class.

By using the component family and renderer type to look up renderers for components, the JavaServer Faces implementation allows a component to be rendered by multiple renderers and allows a renderer to render multiple components.

Each of the `attribute` tags specifies a render-dependent attribute and its type. The `attribute` element doesn't affect the runtime execution of your application. Rather, it provides information to tools about the attributes the `Renderer` supports.

The object responsible for rendering a component (be it the component itself or a renderer to which the component delegates the rendering) can use facets to aid in the rendering process. These facets allow the custom component developer to control some aspects of rendering the component. Consider this custom component tag example:

```xml
<d:dataScroller>
  <f:facet name="header">
    <h:panelGroup>
      <h:outputText value="Account Id"/>
      <h:outputText value="Customer Name"/>
      <h:outputText value="Total Sales"/>
    </h:panelGroup>
  </f:facet>
  <f:facet name="next">
    <h:panelGroup>
      <h:outputText value="Next"/>
      <h:graphicImage url="/images/arrow-right.gif" />
    </h:panelGroup>
  </f:facet>
  ...
</d:dataScroller>
```

The `dataScroller` component tag includes a component that will render the header and a component that will render the Next button. If the renderer associated with this component renders the facets, you can include the following facet elements in the `renderer` element:

```xml
<facet>
  <description>This facet renders as the header of the table. It should be a panelGroup with the same number of columns as the data</description>
  <display-name>header</display-name>
  <facet-name>header</facet-name>
</facet>
<facet>
  <description>This facet renders as the content of the "next" button in the scroller. It should be a panelGroup that includes an outputText</description>
  <display-name>next</display-name>
  <facet-name>next</facet-name>
</facet>
```
tag that has the text "Next" and a right arrow icon.
</description>
<display-name>Next</display-name>
<facet-name>next</facet-name>
</facet>

If a component that supports facets provides its own rendering and you want to include facet elements in the application configuration resource file, you need to put them in the component's configuration rather than the renderer's configuration.

Registering a Custom Component

In addition to registering custom renderers (as explained in the preceding section), you also must register the custom components that are usually associated with the custom renderers. You use either a @FacesComponent annotation, as described in "Creating Custom Component Classes" on page 262, or the component element of the application configuration resource file.

Here is a hypothetical component element from the application configuration resource file that registers AreaComponent:

```xml
<component>
  <component-type>DemoArea</component-type>
  <component-class>
    dukesbookstore.components.AreaComponent
  </component-class>
  <property>
    <property-name>alt</property-name>
    <property-class>java.lang.String</property-class>
  </property>
  <property>
    <property-name>coords</property-name>
    <property-class>java.lang.String</property-class>
  </property>
  <property>
    <property-name>shape</property-name>
    <property-class>java.lang.String</property-class>
  </property>
</component>
```

Attributes specified in a component tag override any settings in the @FacesComponent annotation.

The component-type element indicates the name under which the component should be registered. Other objects referring to this component use this name. For example, the component-type element in the configuration for AreaComponent defines a value of DemoArea, which matches the value returned by the AreaTag class's getComponentType method.

The component-class element indicates the fully qualified class name of the component. The property elements specify the component properties and their types.
If the custom component can include facets, you can configure the facets in the component configuration using `facet` elements, which are allowed after the `component-class` elements. See "Registering a Custom Renderer with a Render Kit" on page 317 for further details on configuring facets.

Basic Requirements of a JavaServer Faces Application

In addition to configuring your application, you must satisfy other requirements of JavaServer Faces applications, including properly packaging all the necessary files and providing a deployment descriptor. This section describes how to perform these administrative tasks.

JavaServer Faces applications can be packaged in a WAR file, which must conform to specific requirements to execute across different containers. At a minimum, a WAR file for a JavaServer Faces application must contain the following:

- A web application deployment descriptor, called `web.xml`, to configure resources required by a web application
- A specific set of JAR files containing essential classes
- A set of application classes, JavaServer Faces pages, and other required resources, such as image files

A WAR file may also contain:

- An application configuration resource file, which configures application resources
- A set of tag library descriptor files

For example, a Java Server Faces web application WAR file using Facelets typically has the following directory structure:

```
$PROJECT_DIR
[Web Pages]
+- /[xhtml documents]
+- /resources
+- /WEB-INF
  +- /classes
  +- /lib
  +- /web.xml
  +- /faces-config.xml (optional)
  +- /*.taglib.xml (optional)
  +- /glassfish-web.xml
```

The `web.xml` file (or web deployment descriptor), the set of JAR files, and the set of application files must be contained in the `WEB-INF` directory of the WAR file.
Configuring an Application with a Web Deployment Descriptor

Web applications are commonly configured using elements contained in the web application deployment descriptor, `web.xml`. The deployment descriptor for a JavaServer Faces application must specify certain configurations, including the following:

- The servlet used to process JavaServer Faces requests
- The servlet mapping for the processing servlet
- The path to the configuration resource file, if it exists and is not located in a default location

The deployment descriptor can also include other, optional configurations, such as:

- Specifying where component state is saved
- Encrypting state saved on the client
- Compressing state saved on the client
- Restricting access to pages containing JavaServer Faces tags
- Turning on XML validation
- Specifying the Project Stage
- Verifying custom objects

This section gives more details on these configurations. Where appropriate, it also describes how you can make these configurations using NetBeans IDE.

Identifying the Servlet for Lifecycle Processing

A requirement of a JavaServer Faces application is that all requests to the application that reference previously saved JavaServer Faces components must go through `javax.faces.webapp.FacesServlet`. A FacesServlet instance manages the request processing lifecycle for web applications and initializes the resources required by JavaServer Faces technology.

Before a JavaServer Faces application can launch its first web page, the web container must invoke the FacesServlet instance in order for the application lifecycle process to start. See “The Lifecycle of a JavaServer Faces Application” on page 210 for more information.

The following example shows the default configuration of the FacesServlet:

```
<servlet>
  <servlet-name>FacesServlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>
```

You provide a mapping configuration entry to make sure the FacesServlet instance is invoked. The mapping to FacesServlet can be a prefix mapping, such as `/faces/*`, or an
extension mapping, such as *.xhtml. The mapping is used to identify a page as having
JavaServer Faces content. Because of this, the URL to the first page of the application must
include the URL pattern mapping.

The following elements, commonly used in the tutorial examples, specify a prefix mapping:

```xml
<servlet-mapping>
  <servlet-name>FacesServlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

...<welcome-file-list>
  <welcome-file>faces/greeting.xhtml</welcome-file>
</welcome-file-list>

The following elements, also commonly used in the tutorial examples, specify an extension
mapping:

```xml
<servlet-mapping>
  <servlet-name>FacesServlet</servlet-name>
  <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

...<welcome-file-list>
  <welcome-file>index.xhtml</welcome-file>
</welcome-file-list>

When you use this mechanism, users access the application as shown in the following example:

```
http://localhost:8080/guessNumber
```

In the case of extension mapping, if a request comes to the server for a page with an .xhtml
extension, the container will send the request to the FacesServlet instance, which will expect
a corresponding page of the same name containing the content to exist.

If you are using NetBeans IDE to create your application, a web deployment descriptor is
automatically created for you with default configurations. If you created your application
without an IDE, you can create a web deployment descriptor.

▼ **To Specify a Path to an Application Configuration Resource File**

As explained in "Application Configuration Resource File" on page 297, an application can have
multiple application configuration resource files. If these files are not located in the directories
that the implementation searches by default or the files are not named faces-config.xml, you
need to specify paths to these files.

To specify these paths using NetBeans IDE, do the following.

1. Expand the node of your project in the Projects pane.

2. Expand the Web Pages and WEB-INF nodes that are under the project node.
Double-click `web.xml`.

After the `web.xml` file appears in the editor pane, click General at the top of the editor pane.

Expand the Context Parameters node.

Click Add.

In the Add Context Parameter dialog:

a. Type `javax.faces.CONFIG_FILES` in the Param Name field.

b. Type the path to your configuration file in the Param Value field.

c. Click OK.

Repeat steps 1 through 7 for each configuration file.

To Specify Where State Is Saved

For all the components in a web application, you can specify in your deployment descriptor where you want the state to be saved, on either client or server. You do this by setting a context parameter in your deployment descriptor. By default, state is saved on the server, so you need to specify this context parameter only if you want to save state on the client. See “Saving and Restoring State” on page 269 for information on the advantages and disadvantages of each location.

To specify where state is saved using NetBeans IDE, do the following.

Expand the node of your project in the Projects pane.

Expand the Web Pages and WEB-INF nodes under the project node.

Double-click `web.xml`.

After the `web.xml` file appears in the editor pane, click General at the top of the editor pane.

Expand the Context Parameters node.

In the Add Context Parameter dialog:

a. Type `javax.faces.STATE_SAVING_METHOD` in the Param Name field.

b. Type `client` or `server` in the Param Value field.
c. Click OK.

More Information

Implementation of State Saving

If state is saved on the client, the state of the entire view is rendered to a hidden field on the page. The JavaServer Faces implementation saves the state on the server by default. Duke's Forest saves its state on the client.

Configuring Project Stage

Project Stage is a context parameter identifying the status of a JavaServer Faces application in the software lifecycle. The stage of an application can affect the behavior of the application. For example, error messages can be displayed during the Development stage but suppressed during the Production stage.

The possible Project Stage values are as follows:

- Development
- UnitTest
- SystemTest
- Production

Project Stage is configured through a context parameter in the web deployment descriptor file. Here is an example:

```xml
<context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```

If no Project Stage is defined, the default stage is Development. You can also add custom stages according to your requirements.

Including the Classes, Pages, and Other Resources

When packaging web applications using the included build scripts, you’ll notice that the scripts package resources in the following ways:

- All web pages are placed at the top level of the WAR file.
- The `faces-config.xml` file and the `web.xml` file are packaged in the WEB-INF directory.
- All packages are stored in the WEB-INF/classes/ directory.
- All application JAR files are packaged in the WEB-INF/lib/ directory.
- All resource files are either under the root of the web application/resources directory, or in the web application's classpath, META-INF/resources/resourceIdentifier directory. For more information on resources, see "Web Resources" on page 123.
When packaging your own applications, you can use NetBeans IDE or you can use the build scripts such as those created for Ant. You can modify the build scripts to fit your situation. However, you can continue to package your WAR files by using the directory structure described in this section, because this technique complies with the commonly accepted practice for packaging web applications.
Shortly after the Web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for the same purpose. Initially, Common Gateway Interface (CGI) server-side scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology had many shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

The following topics are addressed here:

- “What Is a Servlet?” on page 328
- “Servlet Lifecycle” on page 328
- “Sharing Information” on page 330
- “Creating and Initializing a Servlet” on page 331
- “Writing Service Methods” on page 332
- “Filtering Requests and Responses” on page 334
- “Invoking Other Web Resources” on page 338
- “Accessing the Web Context” on page 339
- “Maintaining Client State” on page 340
- “Finalizing a Servlet” on page 342
- “The mood Example Application” on page 344
- “Further Information about Java Servlet Technology” on page 346
What Is a Servlet?

A servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The `javax.servlet` and `javax.servlet.http` packages provide interfaces and classes for writing servlets. All servlets must implement the `Servlet` interface, which defines lifecycle methods. When implementing a generic service, you can use or extend the `GenericServlet` class provided with the Java Servlet API. The `HttpServlet` class provides methods, such as `doGet` and `doPost`, for handling HTTP-specific services.

Servlet Lifecycle

The lifecycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the web container
   a. Loads the servlet class.
   b. Creates an instance of the servlet class.
   c. Initializes the servlet instance by calling the `init` method. Initialization is covered in “Creating and Initializing a Servlet” on page 331.
2. Invokes the `service` method, passing request and response objects. Service methods are discussed in “Writing Service Methods” on page 332.

If it needs to remove the servlet, the container finalizes the servlet by calling the servlet’s `destroy` method. For more information, see “Finalizing a Servlet” on page 342.

Handling Servlet Lifecycle Events

You can monitor and react to events in a servlet’s lifecycle by defining listener objects whose methods get invoked when lifecycle events occur. To use these listener objects, you must define and specify the listener class.

Defining the Listener Class

You define a listener class as an implementation of a listener interface. Table 15–1 lists the events that can be monitored and the corresponding interface that must be implemented. When a listener method is invoked, it is passed an event that contains information appropriate to the event. For example, the methods in the `HttpSessionListener` interface are passed an `HttpSessionEvent`, which contains an `HttpSession`.

---

What Is a Servlet?

The JavaEE6Tutorial • January2013

328
<table>
<thead>
<tr>
<th>Object</th>
<th>Event</th>
<th>Listener Interface and Event Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web context</td>
<td>Initialization and destruction</td>
<td>javax.servlet.ServletContextListener and ServletContextEvent</td>
</tr>
<tr>
<td>Web context</td>
<td>Attribute added, removed, or replaced</td>
<td>javax.servlet.ServletContextAttributeListener and ServletContextAttributeEvent</td>
</tr>
<tr>
<td>Session</td>
<td>Creation, invalidation, activation, passivation, and timeout</td>
<td>javax.servlet.http.HttpSessionListener, javax.servlet.http.HttpSessionActivationListener, and HttpSessionEvent</td>
</tr>
<tr>
<td>Session</td>
<td>Attribute added, removed, or replaced</td>
<td>javax.servlet.http.HttpSessionAttributeListener and HttpSessionBindingEvent</td>
</tr>
<tr>
<td>Request</td>
<td>A servlet request has started being processed by web components</td>
<td>javax.servlet.ServletRequestListener and ServletRequestEvent</td>
</tr>
<tr>
<td>Request</td>
<td>Attribute added, removed, or replaced</td>
<td>javax.servlet.ServletRequestAttributeListener and ServletRequestAttributeEvent</td>
</tr>
</tbody>
</table>

Use the @WebListener annotation to define a listener to get events for various operations on the particular web application context. Classes annotated with @WebListener must implement one of the following interfaces:

- javax.servlet.ServletContextListener
- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener

For example, the following code snippet defines a listener that implements two of these interfaces:

```java
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebServlet;

@WebServlet()
public class SimpleServletListener implements ServletContextListener,
      ServletContextAttributeListener {
    ...
```
Handling Servlet Errors

Any number of exceptions can occur when a servlet executes. When an exception occurs, the web container generates a default page containing the following message:

A Servlet Exception Has Occurred

But you can also specify that the container should return a specific error page for a given exception.

Sharing Information

Web components, like most objects, usually work with other objects to accomplish their tasks. Web components can do so by

■ Using private helper objects (for example, JavaBeans components).
■ Sharing objects that are attributes of a public scope.
■ Using a database.
■ Invoking other web resources. The Java Servlet technology mechanisms that allow a web component to invoke other web resources are described in "Invoking Other Web Resources" on page 338.

Using Scope Objects

Collaborating web components share information by means of objects that are maintained as attributes of four scope objects. You access these attributes by using the `getAttribute` and `setAttribute` methods of the class representing the scope. Table 15–2 lists the scope objects.

<table>
<thead>
<tr>
<th>Scope Object</th>
<th>Class</th>
<th>Accessible from</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session</td>
<td>javax.servlet.http.HttpSession</td>
<td>Web components handling a request that belongs to the session. See &quot;Maintaining Client State&quot; on page 340.</td>
</tr>
<tr>
<td>Request</td>
<td>Subtype of javax.servlet.ServletRequest</td>
<td>Web components handling the request.</td>
</tr>
<tr>
<td>Page</td>
<td>javax.servlet.jsp.JspContext</td>
<td>The JSP page that creates the object.</td>
</tr>
</tbody>
</table>
Controlling Concurrent Access to Shared Resources

In a multithreaded server, shared resources can be accessed concurrently. In addition to scope object attributes, shared resources include in-memory data, such as instance or class variables, and external objects, such as files, database connections, and network connections.

Concurrent access can arise in several situations:

- Multiple web components accessing objects stored in the web context.
- Multiple web components accessing objects stored in a session.
- Multiple threads within a web component accessing instance variables. A web container will typically create a thread to handle each request. To ensure that a servlet instance handles only one request at a time, a servlet can implement the SingleThreadModel interface. If a servlet implements this interface, no two threads will execute concurrently in the servlet's service method. A web container can implement this guarantee by synchronizing access to a single instance of the servlet or by maintaining a pool of web component instances and dispatching each new request to a free instance. This interface does not prevent synchronization problems that result from web components' accessing shared resources, such as static class variables or external objects.

When resources can be accessed concurrently, they can be used in an inconsistent fashion. You prevent this by controlling the access using the synchronization techniques described in the Threads lesson at [http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html](http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html) in The Java Tutorial, Fourth Edition, by Sharon Zakhour et al. (Addison-Wesley, 2006).

Creating and Initializing a Servlet

Use the @WebServlet annotation to define a servlet component in a web application. This annotation is specified on a class and contains metadata about the servlet being declared. The annotated servlet must specify at least one URL pattern. This is done by using the urlPatterns or value attribute on the annotation. All other attributes are optional, with default settings. Use the value attribute when the only attribute on the annotation is the URL pattern; otherwise use the urlPatterns attribute when other attributes are also used.

Classes annotated with @WebServlet must extend the javax.servlet.http.HttpServlet class. For example, the following code snippet defines a servlet with the URL pattern /report:

```java
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;

@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...
```
The web container initializes a servlet after loading and instantiating the servlet class and before delivering requests from clients. To customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities, you can either override the `init` method of the `Servlet` interface or specify the `initParams` attribute of the `@WebServlet` annotation. The `initParams` attribute contains a `@WebInitParam` annotation. If it cannot complete its initialization process, a servlet throws an `UnavailableException`.

Use an initialization parameter to provide data needed by a particular servlet. By contrast, a context parameter provides data that is available to all components of a web application.

**Writing Service Methods**

The service provided by a servlet is implemented in the `service` method of a `GenericServlet`, in the `doMethod` methods (where `Method` can take the value `Get`, `Delete`, `Options`, `Post`, `Put`, or `Trace`) of an `HttpServlet` object, or in any other protocol-specific methods defined by a class that implements the `Servlet` interface. The term `service method` is used for any method in a servlet class that provides a service to a client.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response, based on that information. For HTTP servlets, the correct procedure for populating the response is to do the following:

1. Retrieve an output stream from the response.
2. Fill in the response headers.
3. Write any body content to the output stream.

Response headers must always be set before the response has been committed. The web container will ignore any attempt to set or add headers after the response has been committed. The next two sections describe how to get information from requests and generate responses.

**Getting Information from Requests**

A request contains data passed between a client and the servlet. All requests implement the `ServletRequest` interface. This interface defines methods for accessing the following information:

- Parameters, which are typically used to convey information between clients and servlets
- Object-valued attributes, which are typically used to pass information between the web container and a servlet or between collaborating servlets
- Information about the protocol used to communicate the request and about the client and server involved in the request
- Information relevant to localization
You can also retrieve an input stream from the request and manually parse the data. To read character data, use the `BufferedReader` object returned by the request's `getReader` method. To read binary data, use the `ServletInputStream` returned by `getInputStream`.

HTTP servlets are passed an HTTP request object, `HttpServletRequest`, which contains the request URL, HTTP headers, query string, and so on. An HTTP request URL contains the following parts:

```
http://[host]:[port][request-path]?[query-string]
```

The request path is further composed of the following elements:

- **Context path**: A concatenation of a forward slash (`/`) with the context root of the servlet’s web application.
- **Servlet path**: The path section that corresponds to the component alias that activated this request. This path starts with a forward slash (`/`).
- **Path info**: The part of the request path that is not part of the context path or the servlet path.

You can use the `getContextPath`, `getServletPath`, and `getPathInfo` methods of the `HttpServletRequest` interface to access this information. Except for URL encoding differences between the request URI and the path parts, the request URI is always comprised of the context path plus the servlet path plus the path info.

Query strings are composed of a set of parameters and values. Individual parameters are retrieved from a request by using the `getParameter` method. There are two ways to generate query strings.

- A query string can explicitly appear in a web page.
- A query string is appended to a URL when a form with a GET HTTP method is submitted.

**Constructing Responses**

A response contains data passed between a server and the client. All responses implement the `ServletResponse` interface. This interface defines methods that allow you to

- Retrieve an output stream to use to send data to the client. To send character data, use the `PrintWriter` returned by the response's `getWriter` method. To send binary data in a Multipurpose Internet Mail Extensions (MIME) body response, use the `ServletOutputStream` returned by `getOutputStream`. To mix binary and text data, as in a multipart response, use a `ServletOutputStream` and manage the character sections manually.
- Indicate the content type (for example, `text/html`) being returned by the response with the `setContentType(String)` method. This method must be called before the response is committed. A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at [http://www.iana.org/assignments/media-types/](http://www.iana.org/assignments/media-types/).
Filtering Requests and Responses

- Indicate whether to buffer output with the `setBufferSize(int)` method. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another web resource. The method must be called before any content is written or before the response is committed.

- Set localization information, such as locale and character encoding. See Chapter 17, “Internationalizing and Localizing Web Applications,” for details.

HTTP response objects, `javax.servlet.http.HttpServletResponse`, have fields representing HTTP headers, such as the following:

- Status codes, which are used to indicate the reason a request is not satisfied or that a request has been redirected.

- Cookies, which are used to store application-specific information at the client. Sometimes, cookies are used to maintain an identifier for tracking a user’s session (see “Session Tracking” on page 341).

Filtering Requests and Responses

A filter is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be “attached” to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way, it can be composed with more than one type of web resource.

The main tasks that a filter can perform are as follows:

- Query the request and act accordingly.

- Block the request-and-response pair from passing any further.

- Modify the request headers and data. You do this by providing a customized version of the request.

- Modify the response headers and data. You do this by providing a customized version of the response.

- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.

You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.
Programming Filters

The filtering API is defined by the Filter, FilterChain, and FilterConfig interfaces in the javax.servlet package. You define a filter by implementing the Filter interface.

Use the @WebFilter annotation to define a filter in a web application. This annotation is specified on a class and contains metadata about the filter being declared. The annotated filter must specify at least one URL pattern. This is done by using the urlPatterns or value attribute on the annotation. All other attributes are optional, with default settings. Use the value attribute when the only attribute on the annotation is the URL pattern; use the urlPatterns attribute when other attributes are also used.

Classes annotated with the @WebFilter annotation must implement the javax.servlet.Filter interface.

To add configuration data to the filter, specify the initParams attribute of the @WebFilter annotation. The initParams attribute contains a @WebInitParam annotation. The following code snippet defines a filter, specifying an initialization parameter:

```java
import javax.servlet.Filter;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;

@WebFilter(filterName = "TimeOfDayFilter",
urlPatterns = {"/**"},
initParams = {
    @WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ....
}
```

The most important method in the Filter interface is doFilter, which is passed request, response, and filter chain objects. This method can perform the following actions:

- Examine the request headers.
- Customize the request object if the filter wishes to modify request headers or data.
- Customize the response object if the filter wishes to modify response headers or data.
- Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. The filter invokes the next entity by calling the doFilter method on the chain object, passing in the request and response it was called with or the wrapped versions it may have created. Alternatively, the filter can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
- Examine response headers after invoking the next filter in the chain.
- Throw an exception to indicate an error in processing.
In addition to doFilter, you must implement the init and destroy methods. The init method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the FilterConfig object passed to init.

**Programming Customized Requests and Responses**

There are many ways for a filter to modify a request or a response. For example, a filter can add an attribute to the request or can insert data in the response.

A filter that modifies a response must usually capture the response before it is returned to the client. To do this, you pass a stand-in stream to the servlet that generates the response. The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet’s response.

To pass this stand-in stream to the servlet, the filter creates a response wrapper that overrides the getWriter or getOutputStream method to return this stand-in stream. The wrapper is passed to the doFilter method of the filter chain. Wrapper methods default to calling through to the wrapped request or response object.

To override request methods, you wrap the request in an object that extends either ServletRequestWrapper or HttpServletRequestWrapper. To override response methods, you wrap the response in an object that extends either ServletExceptionWrapper or HttpServletResponseWrapper.

**Specifying Filter Mappings**

A web container uses filter mappings to decide how to apply filters to web resources. A filter mapping matches a filter to a web component by name or to web resources by URL pattern. The filters are invoked in the order in which filter mappings appear in the filter mapping list of a WAR. You specify a filter mapping list for a WAR in its deployment descriptor by either using NetBeans IDE or coding the list by hand with XML.

If you want to log every request to a web application, you map the hit counter filter to the URL pattern `/*`.

You can map a filter to one or more web resources, and you can map more than one filter to a web resource. This is illustrated in Figure 15–1, where filter F1 is mapped to servlets S1, S2, and S3; filter F2 is mapped to servlet S2; and filter F3 is mapped to servlets S1 and S2.
Recall that a filter chain is one of the objects passed to the `doFilter` method of a filter. This chain is formed indirectly by means of filter mappings. The order of the filters in the chain is the same as the order in which filter mappings appear in the web application deployment descriptor.

When a filter is mapped to servlet `S1`, the web container invokes the `doFilter` method of `F1`. The `doFilter` method of each filter in `S1`’s filter chain is invoked by the preceding filter in the chain by means of the chain `doFilter` method. Because `S1`’s filter chain contains filters `F1` and `F3`, `F1`’s call to chain `doFilter` invokes the `doFilter` method of filter `F3`. When `F3`’s `doFilter` method completes, control returns to `F1`’s `doFilter` method.

\[ \text{To Specify Filter Mappings Using NetBeans IDE} \]

1. Expand the application’s project node in the Project pane.
2. Expand the Web Pages and WEB-INF nodes under the project node.
3. Double-click `web.xml`.
4. Click Filters at the top of the editor pane.
5. Expand the Servlet Filters node in the editor pane.
6. Click Add Filter Element to map the filter to a web resource by name or by URL pattern.
7. In the Add Servlet Filter dialog, enter the name of the filter in the Filter Name field.
8. Click Browse to locate the servlet class to which the filter applies.
   You can include wildcard characters so that you can apply the filter to more than one servlet.
Invoking Other Web Resources

Web components can invoke other web resources both indirectly and directly. A web component indirectly invokes another web resource by embedding a URL that points to another web component in content returned to a client. While it is executing, a web component directly invokes another resource by either including the content of another resource or forwarding a request to another resource.

To invoke a resource available on the server that is running a web component, you must first obtain a RequestDispatcher object by using the getRequestDispatcher("URL") method. You can get a RequestDispatcher object from either a request or the web context; however, the two methods have slightly different behavior. The method takes the path to the requested resource as an argument. A request can take a relative path (that is, one that does not begin with a /), but the web context requires an absolute path. If the resource is not available or if the server has not implemented a RequestDispatcher object for that type of resource, getRequestDispatcher will return null. Your servlet should be prepared to deal with this condition.
Including Other Resources in the Response

It is often useful to include another web resource, such as banner content or copyright information) in the response returned from a web component. To include another resource, invoke the `include` method of a `RequestDispatcher` object:

```java
include(request, response);
```

If the resource is static, the `include` method enables programmatic server-side includes. If the resource is a web component, the effect of the method is to send the request to the included web component, execute the web component, and then include the result of the execution in the response from the containing servlet. An included web component has access to the request object but is limited in what it can do with the response object.

- It can write to the body of the response and commit a response.
- It cannot set headers or call any method, such as `setCookie`, that affects the headers of the response.

Transferring Control to Another Web Component

In some applications, you might want to have one web component do preliminary processing of a request and have another component generate the response. For example, you might want to partially process a request and then transfer to another component, depending on the nature of the request.

To transfer control to another web component, you invoke the `forward` method of a `RequestDispatcher`. When a request is forwarded, the request URL is set to the path of the forwarded page. The original URL and its constituent parts are saved as request attributes `javax.servlet.forward.[request-uri|context-path|servlet-path|path-info|query-string]`.

The `forward` method should be used to give another resource responsibility for replying to the user. If you have already accessed a `ServletOutputStream` or `PrintWriter` object within the servlet, you cannot use this method; doing so throws an `IllegalStateException`.

Accessing the Web Context

The context in which web components execute is an object that implements the `ServletContext` interface. You retrieve the web context by using the `getServletContext` method. The web context provides methods for accessing

- Initialization parameters
- Resources associated with the web context
- Object-valued attributes
- Logging capabilities
The counter’s access methods are synchronized to prevent incompatible operations by servlets that are running concurrently. A filter retrieves the counter object by using the context’s `getAttribute` method. The incremented value of the counter is recorded in the log.

### Maintaining Client State

Many applications require that a series of requests from a client be associated with one another. For example, a web application can save the state of a user’s shopping cart across requests. Web-based applications are responsible for maintaining such state, called a session, because HTTP is stateless. To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

### Accessing a Session

Sessions are represented by an `HttpSession` object. You access a session by calling the `getSession` method of a request object. This method returns the current session associated with this request; or, if the request does not have a session, this method creates one.

### Associating Objects with a Session

You can associate object-valued attributes with a session by name. Such attributes are accessible by any web component that belongs to the same web context and is handling a request that is part of the same session.

Recall that your application can notify web context and session listener objects of servlet lifecycle events (“Handling Servlet Lifecycle Events” on page 328). You can also notify objects of certain events related to their association with a session such as the following:

- When the object is added to or removed from a session. To receive this notification, your object must implement the `javax.servlet.http.HttpSessionBindingListener` interface.
- When the session to which the object is attached will be passivated or activated. A session will be passivated or activated when it is moved between virtual machines or saved to and restored from persistent storage. To receive this notification, your object must implement the `javax.servlet.http.HttpSessionActivationListener` interface.
Session Management

Because an HTTP client has no way to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed. The timeout period can be accessed by using a session’s `getMaxInactiveInterval` and `setMaxInactiveInterval` methods.

- To ensure that an active session is not timed out, you should periodically access the session by using service methods because this resets the session’s time-to-live counter.
- When a particular client interaction is finished, you use the session’s `invalidate` method to invalidate a session on the server side and remove any session data.

To Set the Timeout Period Using NetBeans IDE

To set the timeout period in the deployment descriptor using NetBeans IDE, follow these steps.

1. Open the project if you haven’t already.
2. Expand the project’s node in the Projects pane.
3. Expand the Web Pages node and then the WEB-INF node.
5. Click General at the top of the editor.
6. In the Session Timeout field, type an integer value.
   The integer value represents the number of minutes of inactivity that must pass before the session times out.

Session Tracking

To associate a session with a user, a web container can use several methods, all of which involve passing an identifier between the client and the server. The identifier can be maintained on the client as a cookie, or the web component can include the identifier in every URL that is returned to the client.

If your application uses session objects, you must ensure that session tracking is enabled by having the application rewrite URLs whenever the client turns off cookies. You do this by calling the response’s `encodeURL(URL)` method on all URLs returned by a servlet. This method includes the session ID in the URL only if cookies are disabled; otherwise, the method returns the URL unchanged.
Finalizing a Servlet

The web container may determine that a servlet should be removed from service (for example, when a container wants to reclaim memory resources or when it is being shutdown). In such a case, the container calls the destroy method of the Servlet interface. In this method, you release any resources the servlet is using and save any persistent state. The destroy method releases the database object created in the init method.

A servlet’s service methods should all be complete when a servlet is removed. The server tries to ensure this by calling the destroy method only after all service requests have returned or after a server-specific grace period, whichever comes first. If your servlet has operations that may run longer than the server’s grace period, the operations could still be running when destroy is called. You must make sure that any threads still handling client requests complete.

The remainder of this section explains how to do the following:

- Keep track of how many threads are currently running the service method.
- Provide a clean shutdown by having the destroy method notify long-running threads of the shutdown and wait for them to complete.
- Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up, and return.

Tracking Service Requests

To track service requests, include in your servlet class a field that counts the number of service methods that are running. The field should have synchronized access methods to increment, decrement, and return its value:

```java
public class ShutdownExample extends HttpServlet {
    private int serviceCounter = 0;
    // Access methods for serviceCounter
    protected synchronized void enteringServiceMethod() {
        serviceCounter++;
    }
    protected synchronized void leavingServiceMethod() {
        serviceCounter--;
    }
    protected synchronized int numServices() {
        return serviceCounter;
    }
}
```

The service method should increment the service counter each time the method is entered and should decrement the counter each time the method returns. This is one of the few times that your HttpServlet subclass should override the service method. The new method should call super.service to preserve the functionality of the original service method.
protected void service(HttpServletRequest req,
HttpServletRequest resp)
    throws ServletException, IOException {
    enteringServiceMethod();
    try {
        super.service(req, resp);
    } finally {
        leavingServiceMethod();
    }
}

Notifying Methods to Shut Down

To ensure a clean shutdown, your destroy method should not release any shared resources until all the service requests have completed. One part of doing this is to check the service counter. Another part is to notify the long-running methods that it is time to shut down. For this notification, another field is required. The field should have the usual access methods:

public class ShutdownExample extends HttpServlet {
    private boolean shuttingDown;

    // Access methods for shuttingDown
    protected synchronized void setShuttingDown(boolean flag) {
        shuttingDown = flag;
    }

    protected synchronized boolean isShuttingDown() {
        return shuttingDown;
    }
}

Here is an example of the destroy method using these fields to provide a clean shutdown:

public void destroy() {
    /* Check to see whether there are still service methods */
    /* running, and if there are, tell them to stop. */
    if (numServices() > 0) {
        setShuttingDown(true);
    }

    /* Wait for the service methods to stop. */
    while(numServices() > 0) {
        try {
            Thread.sleep(interval);
        } catch (InterruptedException e) {
        }
    }
}

Creating Polite Long-Running Methods

The final step in providing a clean shutdown is to make any long-running methods behave politely. Methods that might run for a long time should check the value of the field that notifies them of shutdowns and should interrupt their work, if necessary:
public void doPost(...) {
    ...
    for(i = 0; ((i < lotsOfStuffToDo) &&
        !isShuttingDown()); i++) {
        try {
            partOfLongRunningOperation(i);
        } catch (InterruptedException e) {
            ...
        }
    }
}

The mood Example Application

The mood example application, located in the tut-install/examples/web/mood/ directory, is a simple example that displays Duke's moods at different times during the day. The example shows how to develop a simple application by using the @WebServlet, @WebFilter, and @WebListener annotations to create a servlet, a listener, and a filter.

Components of the mood Example Application


MoodServlet, the presentation layer of the application, displays Duke's mood in a graphic, based on the time of day. The @WebServlet annotation specifies the URL pattern:

@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...

TimeOfDayFilter sets an initialization parameter indicating that Duke is awake:

@WebFilter(filterName = "TimeOfDayFilter",
            urlPatterns = {"/*"},
            initParams = {
                @WebInitParam(name = "mood", value = "awake")})
public class TimeOfDayFilter implements Filter {
    ...

    The filter calls the doFilter method, which contains a switch statement that sets Duke's mood based on the current time.

SimpleServletListener logs changes in the servlet's lifecycle. The log entries appear in the server log.

Running the mood Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the mood example.
To Run the mood Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   `tut-install/examples/web/`
3. Select the mood folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the mood project and select Build.
7. Right-click the project and select Deploy.
   The URL specifies the context root, followed by the URL pattern specified for the servlet.
   A web page appears with the title “Servlet MoodServlet at /mood” a text string describing Duke’s mood, and an illustrative graphic.

To Run the mood Example Using Ant

1. In a terminal window, go to:
   `tut-install/examples/web/mood/`
2. Type the following command:
   `ant`
   This target builds the WAR file and copies it to the `tut-install/examples/web/mood/dist/` directory.
3. Type `ant deploy`.
   Ignore the URL shown in the deploy target output.
   The URL specifies the context root, followed by the URL pattern.
   A web page appears with the title “Servlet MoodServlet at /mood” a text string describing Duke’s mood, and an illustrative graphic.
Further Information about Java Servlet Technology

For more information on Java Servlet technology, see

- Java Servlet 3.0 specification:
- Java Servlet web site:
  http://www.oracle.com/technetwork/java/index-jsp-135475.html
Supporting file uploads is a very basic and common requirement for many web applications. Prior to Servlet 3.0, implementing file upload required the use of external libraries or complex input processing. Version 3.0 of the Java Servlet specification helps to provide a viable solution to the problem in a generic and portable way. The Servlet 3.0 specification supports file upload out of the box, so any web container that implements the specification can parse multipart requests and make mime attachments available through the HttpServletRequest object.

A new annotation, javax.servlet.annotation.MultipartConfig, is used to indicate that the servlet on which it is declared expects requests to made using the multipart/form-data MIME type. Servlets that are annotated with @MultipartConfig can retrieve the Part components of a given multipart/form-data request by calling the request.getPart(String name) or request.getParts() method.

The following topics are addressed here:

- “The @MultipartConfig Annotation” on page 347
- “The getParts and getPart Methods” on page 348
- “The fileupload Example Application” on page 349

The @MultipartConfig Annotation

The @MultipartConfig annotation supports the following optional attributes:

- location: An absolute path to a directory on the file system. The location attribute does not support a path relative to the application context. This location is used to store files temporarily while the parts are processed or when the size of the file exceeds the specified fileSizeThreshold setting. The default location is "".
- fileSizeThreshold: The file size in bytes after which the file will be temporarily stored on disk. The default size is 0 bytes.
MaxFileSize: The maximum size allowed for uploaded files, in bytes. If the size of any uploaded file is greater than this size, the web container will throw an exception (IllegalStateException). The default size is unlimited.

maxRequestSize: The maximum size allowed for a multipart/form-data request, in bytes. The web container will throw an exception if the overall size of all uploaded files exceeds this threshold. The default size is unlimited.

For example, the @MultipartConfig annotation could be constructed as follows:

```java
@MultipartConfig(location="/tmp", fileSizeThreshold=1024*1024,
maxFileSize=1024*1024*5, maxRequestSize=1024*1024*5*5)
```

Instead of using the @MultipartConfig annotation to hard-code these attributes in your file upload servlet, you could add the following as a child element of the `servlet` configuration element in the `web.xml` file.

```xml
<multipart-config>
  <location>/tmp</location>
  <max-file-size>20848820</max-file-size>
  <max-request-size>418018841</max-request-size>
  <file-size-threshold>1048576</file-size-threshold>
</multipart-config>
```

The `getParts` and `getPart` Methods

Servlet 3.0 supports two additional `HttpServletRequest` methods:

- Collection<Part> getParts()
- Part getPart(String name)

The `request.getParts()` method returns collections of all `Part` objects. If you have more than one input of type `file`, multiple `Part` objects are returned. Since `Part` objects are named, the `getPart(String name)` method can be used to access a particular `Part`. Alternatively, the `getParts()` method, which returns an `Iterable<Part>`, can be used to get an `Iterator` over all the `Part` objects.

The `javax.servlet.http.Part` interface is a simple one, providing methods that allow introspection of each `Part`. The methods do the following:

- Retrieve the name, size, and content-type of the `Part`
- Query the headers submitted with a `Part`
- Delete a `Part`
- Write a `Part` out to disk

For example, the `Part` interface provides the `write(String filename)` method to write the file with the specified name. The file can then be saved in the directory specified with the `location` attribute of the `@MultipartConfig` annotation or, in the case of the file upload example, in the location specified by the `Destination` field in the form.
The fileupload Example Application

The fileupload example illustrates how to implement and use the file upload feature.

The Duke's Forest case study provides a more complex example that uploads an image file and stores its content in a database.

Architecture of the fileupload Example Application

The fileupload example application consists of a single servlet and an HTML form that makes a file upload request to the servlet.

This example includes a very simple HTML form with two fields, File and Destination. The input type, file, enables a user to browse the local file system to select the file. When the file is selected, it is sent to the server as part of a POST request. During this process two mandatory restrictions are applied to the form with input type file:

- The enctype attribute must be set to a value of multipart/form-data.
- Its method must be POST.

When the form is specified in this manner, the entire request is sent to the server in encoded form. The servlet then handles the request to process the incoming file data and to extract a file from the stream. The destination is the path to the location where the file will be saved on your computer. Pressing the Upload button at the bottom of the form posts the data to the servlet, which saves the file in the specified destination.

The HTML form in tut-install/examples/web/fileupload/web/index.html is as follows:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>File Upload</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <form method="POST" action="upload" enctype="multipart/form-data">
      File:
      <input type="file" name="file" id="file" />
      <br/>
      Destination:
      <input type="text" value="/tmp" name="destination"/>
      <br/>
      <input type="submit" value="Upload" name="upload" id="upload" />
    </form>
  </body>
</html>
```

A POST request method is used when the client needs to send data to the server as part of the request, such as when uploading a file or submitting a completed form. In contrast, a GET request method sends a URL and headers only to the server, whereas POST requests also
include a message body. This allows arbitrary-length data of any type to be sent to the server. A header field in the POST request usually indicates the message body's Internet media type.

When submitting a form, the browser streams the content in, combining all parts, with each part representing a field of a form. Parts are named after the input elements and are separated from each other with string delimiters named boundary.

This is what submitted data from the fileupload form looks like, after selecting sample.txt as the file that will be uploaded to the tmp directory on the local file system:

```
POST /fileupload/upload HTTP/1.1
Host: localhost:8080
Content-Type: multipart/form-data; boundary=---------------------------263081694432439
Content-Length: 441
----------------------------263081694432439
Content-Disposition: form-data; name="file"; filename="sample.txt"
Content-Type: text/plain
Data from sample file
----------------------------263081694432439
Content-Disposition: form-data; name="destination"
/tmp
----------------------------263081694432439
Content-Disposition: form-data; name="upload"
Upload
----------------------------263081694432439--
```

The servlet FileUploadServlet.java can be found in the tut-install/examples/web/fileupload/src/java/fileupload/ directory. The servlet begins as follows:

```java
@WebServlet(name = "FileUploadServlet", urlPatterns = {"/upload"})
@MultipartConfig
public class FileUploadServlet extends HttpServlet {
    private final static Logger LOGGER =
            Logger.getLogger(FileUploadServlet.class.getCanonicalName());
```

The @WebServlet annotation uses the urlPatterns property to define servlet mappings.

The @MultipartConfig annotation indicates that the servlet expects requests to made using the multipart/form-data MIME type.

The processRequest method retrieves the destination and file part from the request, then calls the getFileName method to retrieve the file name from the file part. The method then creates a FileOutputStream and copies the file to the specified destination. The error-handling section of the method catches and handles some of the most common reasons why a file would not be found. The processRequest and getFileName methods look like this:
protected void processRequest(HttpServletRequest request, 
   HttpServletResponse response) 
   throws ServletException, IOException {
   response.setContentType("text/html;charset=UTF-8");

   // Create path components to save the file
   final String path = request.getParameter("destination");
   final Part filePart = request.getPart("file");
   final String fileName = getFileName(filePart);

   OutputStream out = null;
   InputStream filecontent = null;
   final PrintWriter writer = response.getWriter();
   try {
      out = new FileOutputStream(new File(path + File.separator + fileName));
      filecontent = filePart.getInputStream();
      int read = 0;
      final byte[] bytes = new byte[1024];
      while ((read = filecontent.read(bytes)) != -1) {
         out.write(bytes, 0, read);
      }
      writer.println("New file \" + fileName + \" created at \" + path + ");
      LOGGER.log(Level.INFO, "File\{0\} being uploaded to \{1\}", new Object[]{fileName, path});
   } catch (FileNotFoundException fne) {
      writer.println("You either did not specify a file to upload or are " + "trying to upload a file to a protected or nonexistent " + "location.");
      writer.println(<br/>ERROR: " + fne.getMessage());
      LOGGER.log(Level.SEVERE, "Problems during file upload. Error: {0}", new Object[]{fne.getMessage()});
   } finally {
      if (out != null) {
         out.close();
      }
      if (filecontent != null) {
         filecontent.close();
      }
      if (writer != null) {
         writer.close();
      }
   }

private String getFileName(final Part part) {
   final String partHeader = part.getHeader("content-disposition");
   LOGGER.log(Level.INFO, "Part Header = \{0\}", partHeader);
   for (String content : part.getHeader("content-disposition").split(";")) {
      if (content.trim().startsWith("filename")) {
         return content.substring(content.indexOf("='\") + 1).trim().replace("\\", ");
      }
   }
   return null;
}
Running the fileupload Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the fileupload example.

▼ To Build, Package, and Deploy the fileupload Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   
   tut-install/examples/web/

3. Select the fileupload folder.
4. Select the Open as Main Project checkbox.
5. Click Open Project.
6. In the Projects tab, right-click fileupload and select Deploy.

▼ To Build, Package, and Deploy the fileupload Example Using Ant

1. In a terminal window, go to:
   
   tut-install/examples/web/fileupload/

2. Type the following command:
   
   ant

3. Type the following command:
   
   ant deploy

▼ To Run the fileupload Example

1. In a web browser, type the following URL:
   
   http://localhost:8080/fileupload/

   The File Upload page opens.

2. Click Browse to display a file browser window.
3 Select a file to upload and click Open.
The name of the file you selected is displayed in the File field. If you do not select a file, an exception will be thrown.

4 In the Destination field, type a directory name.
The directory must have already been created and must also be writable. If you do not enter a directory name, or if you enter the name of a nonexistent or protected directory, an exception will be thrown.

5 Click Upload to upload the file you selected to the directory you specified in the Destination field.
A message reports that the file was created in the directory you specified.

6 Go to the directory you specified in the Destination field and verify that the uploaded file is present.
Internationalizing and Localizing Web Applications

The process of preparing an application to support more than one language and data format is called internationalization. Localization is the process of adapting an internationalized application to support a specific region or locale. Examples of locale-dependent information include messages and user interface labels, character sets and encoding, and date and currency formats. Although all client user interfaces should be internationalized and localized, these processes are particularly important for web applications because of the global nature of the web.

The following topics are addressed here:

■ “Java Platform Localization Classes” on page 355
■ “Providing Localized Messages and Labels” on page 356
■ “Date and Number Formatting” on page 359
■ “Character Sets and Encodings” on page 359

Java Platform Localization Classes

In the Java platform, java.util.Locale (http://docs.oracle.com/javase/6/docs/api/java/util/Locale.html) represents a specific geographical, political, or cultural region. The string representation of a locale consists of the international standard two-character abbreviation for language and country and an optional variant, separated by underscore (_) characters. Examples of locale strings include fr (French), de_CH (Swiss German), and en_US_POSIX (English on a POSIX-compliant platform).

Locale-sensitive data is stored in a java.util.ResourceBundle (http://docs.oracle.com/javase/6/docs/api/java/util/ResourceBundle.html). A resource bundle contains key-value pairs, where the key uniquely identifies a locale-specific object in the bundle. A resource bundle can be backed by a text file (properties resource bundle) or a class (list resource bundle) containing the pairs. You construct a resource bundle instance by appending a locale string representation to a base name.
The Duke's Tutoring application contains resource bundles with the base name `messages.properties` for the locales `pt` (Portuguese), `de` (German), `es` (Spanish), and `zh` (Chinese). The default locale, `en` (English), which is specified in the `faces-config.xml` file, uses the resource bundle with the base name `messages.properties`.

For more details on internationalization and localization in the Java platform, see (http://docs.oracle.com/javase/tutorial/i18n/index.html).

**Providing Localized Messages and Labels**

Messages and labels should be tailored according to the conventions of a user's language and region. There are two approaches to providing localized messages and labels in a web application:

- Provide a version of the web page in each of the target locales and have a controller servlet dispatch the request to the appropriate page depending on the requested locale. This approach is useful if large amounts of data on a page or an entire web application need to be internationalized.

- Isolate any locale-sensitive data on a page into resource bundles, and access the data so that the corresponding translated message is fetched automatically and inserted into the page. Thus, instead of creating strings directly in your code, you create a resource bundle that contains translations and read the translations from that bundle using the corresponding key.

The Duke's Tutoring application follows the second approach. Here are a few lines from the default resource bundle `messages.properties`:

- `nav.main=Main page`
- `nav.status=View status`
- `nav.current_session=View current tutoring session`
- `nav.park=View students at the park`
- `nav.admin=Administration`

- `admin.nav.main=Administration main page`
- `admin.nav.create_student=Create new student`
- `admin.nav.edit_student=Edit student`
- `admin.nav.create.guardian=Create new guardian`
- `admin.nav.edit.guardian=Edit guardian`
- `admin.nav.create.address=Create new address`
- `admin.nav.edit.address=Edit address`
- `admin.nav.activate.student=Activate student`

**Establishing the Locale**

To get the correct strings for a given user, a web application either retrieves the locale (set by a browser language preference) from the request using the `getLocale` method, or allows the user to explicitly select the locale.
A component can explicitly set the locale by using the `fmt:setLocale` tag.

The `locale-config` element in the configuration file registers the default locale and other supported locales. This element in Duke's Tutoring registers English as the default locale and indicates that German, Spanish, Portuguese, and Chinese are supported locales.

```xml
<locale-config>
  <default-locale>en</default-locale>
  <supported-locale>de</supported-locale>
  <supported-locale>es</supported-locale>
  <supported-locale>pt</supported-locale>
  <supported-locale>zh</supported-locale>
</locale-config>
```

The Status Manager in the Duke's Tutoring application uses the `getLocale` method to retrieve the locale and a `toString` method to return a localized translation of a student's status based on the locale.

```java
public class StatusManager {
    private FacesContext ctx = FacesContext.getCurrentInstance();
    private Locale locale;

    /** Creates a new instance of StatusManager */
    public StatusManager() {
        locale = ctx.getViewRoot().getLocale();
    }

    public String getLocalizedStatus(StatusType status) {
        return status.toString(locale);
    }
}
```

### Setting the Resource Bundle

The resource bundle is set with the `resource-bundle` element in the configuration file. The setting for Duke’s Tutoring looks like this:

```xml
<resource-bundle>
    <base-name>dukestutoring.web.messages.Messages</base-name>
    <var>bundle</var>
</resource-bundle>
```

After the locale is set, the controller of a web application could retrieve the resource bundle for that locale and save it as a session attribute (see “Associating Objects with a Session” on page 340) for use by other components or simply to return a text string appropriate for the selected locale:

```java
public String toString(Locale locale) {
    ResourceBundle res = ResourceBundle.getBundle(
```
Providing Localized Messages and Labels

```
"dukestutoring.web.messages.Messages", locale);
return res.getString(name() + ".string");
```

Alternatively, an application could use the `<f:loadBundle>` tag to set the resource bundle. This tag loads the correct resource bundle according to the locale stored in `FacesContext`.

```
<f:loadBundle basename="dukestutoring.web.messages.Messages"
    var="bundle"/>
```

Resource bundles containing messages that are explicitly referenced from a JavaServer Faces tag attribute using a value expression must be registered using the `resource-bundle` element of the configuration file.

For more information on using this element, see “Registering Application Messages” on page 308.

Retrieving Localized Messages

A web component written in the Java programming language retrieves the resource bundle from the session:

```
ResourceBundle messages = (ResourceBundle)session.getAttribute("messages");
```

Then it looks up the string associated with the key `person.lastName` as follows:

```
messages.getString("person.lastName");
```

You can only use a `message` or `messages` tag to display messages that are queued onto a component as a result of a converter or validator being registered on the component. The following example shows a `message` tag that displays the error message queued on the `userNo` input component if the validator registered on the component fails to validate the value the user enters into the component.

```
<h:inputText id="userNo" value="#{UserNumberBean.userNumber}">
    <f:validateLongRange minimum="0" maximum="10"/>
    ...
    <h:message
        style="color: red; text-decoration: underline" id="errors1" for="userNo"/>
```

For more information on using the `message` or `messages` tags, see “Displaying Error Messages with the `<h:message>` and `<h:messages>` Tags” on page 164.

Messages that are not queued on a component and are therefore not loaded automatically are referenced using a value expression. You can reference a localized message from almost any JavaServer Faces tag attribute.
The value expression that references a message has the same notation whether you loaded the resource bundle with the *f:loadBundle* tag or registered it with the *resource-bundle* element in the configuration file.

The value expression notation is `var.message`, in which `var` matches the `var` attribute of the *f:loadBundle* tag or the `var` element defined in the *resource-bundle* element of the configuration file, and `message` matches the key of the message contained in the resource bundle, referred to by the `var` attribute.

Here is an example from `editAddress.xhtml` in Duke's Tutoring:

```xml
<h:outputLabel for="country" value="#{bundle['address.country']}"/>
```

Notice that `bundle` matches the `var` element from the configuration file and that `country` matches the key in the resource bundle.

## Date and Number Formatting

Java programs use the `DateFormat.getDateInstance(int, locale)` to parse and format dates in a locale-sensitive manner. Java programs use the `NumberFormat.getXXXInstance(locale)` method, where `XXX` can be `Currency`, `Number`, or `Percent`, to parse and format numerical values in a locale-sensitive manner.

An application can use date/time and number converters to format dates and numbers in a locale-sensitive manner. For example, a shipping date could be converted as follows:

```xml
<h:outputText value="#{cashier.shipDate}"
   <f:convertDateTime dateStyle="full"/>
</h:outputText>
```

For information on JavaServer Faces converters, see "Using the Standard Converters" on page 175.

## Character Sets and Encodings

The following sections describe character sets and character encodings.

### Character Sets

A *character set* is a set of textual and graphic symbols, each of which is mapped to a set of nonnegative integers.

The first character set used in computing was US-ASCII. It is limited in that it can represent only American English. US-ASCII contains uppercase and lowercase Latin letters, numerals, punctuation, control codes, and a few miscellaneous symbols.
Unicode defines a standardized, universal character set that can be extended to accommodate additions. When the Java program source file encoding doesn’t support Unicode, you can represent Unicode characters as escape sequences by using the notation \uXXXX, where XXXX is the character’s 16-bit representation in hexadecimal. For example, the Spanish version of the Duke’s Tutoring message file uses Unicode for non-ASCII characters:

nav.main=Página Principal
nav.status=Mirar el estado
nav.current_session=Ver sesión actual del tutorial
nav.park=Ver estudiantes en el Parque
nav.admin=Administración

admin.nav.main=Página principal de administración
admin.nav.create_student=Crear un nuevo estudiante
admin.nav.edit_student=Editar información del estudiante
admin.nav.create_guardian=Crear un nuevo guardia
admin.nav.edit_guardian=Editar guardia
admin.nav.create_address=Crear una nueva dirección
admin.nav.edit_address=Editar dirección
admin.nav.activate_student=Activar estudiante

Character Encoding

A character encoding maps a character set to units of a specific width and defines byte serialization and ordering rules. Many character sets have more than one encoding. For example, Java programs can represent Japanese character sets using the EUC-JP or Shift-JIS encodings, among others. Each encoding has rules for representing and serializing a character set.

The ISO 8859 series defines 13 character encodings that can represent texts in dozens of languages. Each ISO 8859 character encoding can have up to 256 characters. ISO-8859-1 (Latin-1) comprises the ASCII character set, characters with diacritics (accents, diaereses, cedillas, circumflexes, and so on), and additional symbols.

UTF-8 (Unicode Transformation Format, 8-bit form) is a variable-width character encoding that encodes 16-bit Unicode characters as one to four bytes. A byte in UTF-8 is equivalent to 7-bit ASCII if its high-order bit is zero; otherwise, the character comprises a variable number of bytes.

UTF-8 is compatible with the majority of existing web content and provides access to the Unicode character set. Current versions of browsers and email clients support UTF-8. In addition, many new web standards specify UTF-8 as their character encoding. For example, UTF-8 is one of the two required encodings for XML documents (the other is UTF-16).

Web components usually use PrintWriter to produce responses; PrintWriter automatically encodes using ISO-8859-1. Servlets can also output binary data using OutputStream classes, which perform no encoding. An application that uses a character set that cannot use the default encoding must explicitly set a different encoding.
PART III

Web Services

Part III explores web services. This part contains the following chapters:

- Chapter 18, "Introduction to Web Services"
- Chapter 19, "Building Web Services with JAX-WS"
- Chapter 20, "Building RESTful Web Services with JAX-RS"
- Chapter 21, "JAX-RS: Advanced Topics and Example"
Introduction to Web Services

Part III of the tutorial discusses Java EE 6 web services technologies. For this book, these technologies include Java API for XML Web Services (JAX-WS) and Java API for RESTful Web Services (JAX-RS).

The following topics are addressed here:
- “What Are Web Services?” on page 363
- “Types of Web Services” on page 363
- “Deciding Which Type of Web Service to Use” on page 366

What Are Web Services?

Web services are client and server applications that communicate over the World Wide Web’s (WWW) HyperText Transfer Protocol (HTTP). As described by the World Wide Web Consortium (W3C), web services provide a standard means of interoperating between software applications running on a variety of platforms and frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions, thanks to the use of XML. Web services can be combined in a loosely coupled way to achieve complex operations. Programs providing simple services can interact with each other to deliver sophisticated added-value services.

Types of Web Services

On the conceptual level, a service is a software component provided through a network-accessible endpoint. The service consumer and provider use messages to exchange invocation request and response information in the form of self-containing documents that make very few assumptions about the technological capabilities of the receiver.
On a technical level, web services can be implemented in various ways. The two types of web services discussed in this section can be distinguished as “big” web services and “RESTful” web services.

“Big” Web Services

In Java EE 6, JAX-WS provides the functionality for “big” web services, which are described in Chapter 19, “Building Web Services with JAX-WS.” Big web services use XML messages that follow the Simple Object Access Protocol (SOAP) standard, an XML language defining a message architecture and message formats. Such systems often contain a machine-readable description of the operations offered by the service, written in the Web Services Description Language (WSDL), an XML language for defining interfaces syntactically.

The SOAP message format and the WSDL interface definition language have gained widespread adoption. Many development tools, such as NetBeans IDE, can reduce the complexity of developing web service applications.

A SOAP-based design must include the following elements.

- A formal contract must be established to describe the interface that the web service offers. WSDL can be used to describe the details of the contract, which may include messages, operations, bindings, and the location of the web service. You may also process SOAP messages in a JAX-WS service without publishing a WSDL.

- The architecture must address complex nonfunctional requirements. Many web service specifications address such requirements and establish a common vocabulary for them. Examples include transactions, security, addressing, trust, coordination, and so on.

- The architecture needs to handle asynchronous processing and invocation. In such cases, the infrastructure provided by standards, such as Web Services Reliable Messaging (WSRM), and APIs, such as JAX-WS, with their client-side asynchronous invocation support, can be leveraged out of the box.

RESTful Web Services

In Java EE 6, JAX-RS provides the functionality for Representational State Transfer (RESTful) web services. REST is well suited for basic, ad hoc integration scenarios. RESTful web services, often better integrated with HTTP than SOAP-based services are, do not require XML messages or WSDL service–API definitions.

Project Jersey is the production-ready reference implementation for the JAX-RS specification. Jersey implements support for the annotations defined in the JAX-RS specification, making it easy for developers to build RESTful web services with Java and the Java Virtual Machine (JVM).
Because RESTful web services use existing well-known W3C and Internet Engineering Task Force (IETF) standards (HTTP, XML, URI, MIME) and have a lightweight infrastructure that allows services to be built with minimal tooling, developing RESTful web services is inexpensive and thus has a very low barrier for adoption. You can use a development tool such as NetBeans IDE to further reduce the complexity of developing RESTful web services.

A RESTful design may be appropriate when the following conditions are met.

- The web services are completely stateless. A good test is to consider whether the interaction can survive a restart of the server.
- A caching infrastructure can be leveraged for performance. If the data that the web service returns is not dynamically generated and can be cached, the caching infrastructure that web servers and other intermediaries inherently provide can be leveraged to improve performance. However, the developer must take care because such caches are limited to the HTTP GET method for most servers.
- The service producer and service consumer have a mutual understanding of the context and content being passed along. Because there is no formal way to describe the web services interface, both parties must agree out of band on the schemas that describe the data being exchanged and on ways to process it meaningfully. In the real world, most commercial applications that expose services as RESTful implementations also distribute so-called value-added toolkits that describe the interfaces to developers in popular programming languages.
- Bandwidth is particularly important and needs to be limited. REST is particularly useful for limited-profile devices, such as PDAs and mobile phones, for which the overhead of headers and additional layers of SOAP elements on the XML payload must be restricted.
- Web service delivery or aggregation into existing web sites can be enabled easily with a RESTful style. Developers can use such technologies as JAX-RS and Asynchronous JavaScript with XML (AJAX) and such toolkits as Direct Web Remoting (DWR) to consume the services in their web applications. Rather than starting from scratch, services can be exposed with XML and consumed by HTML pages without significantly refactoring the existing web site architecture. Existing developers will be more productive because they are adding to something they are already familiar with rather than having to start from scratch with new technology.

RESTful web services are discussed in Chapter 20, “Building RESTful Web Services with JAX-RS.” This chapter contains information about generating the skeleton of a RESTful web service using both NetBeans IDE and the Maven project management tool.
Deciding Which Type of Web Service to Use

Basically, you would want to use RESTful web services for integration over the web and use big web services in enterprise application integration scenarios that have advanced quality of service (QoS) requirements.

- **JAX-WS**: addresses advanced QoS requirements commonly occurring in enterprise computing. When compared to JAX-RS, JAX-WS makes it easier to support the WS-* set of protocols, which provide standards for security and reliability, among other things, and interoperate with other WS-* conforming clients and servers.

- **JAX-RS**: makes it easier to write web applications that apply some or all of the constraints of the REST style to induce desirable properties in the application, such as loose coupling (evolving the server is easier without breaking existing clients), scalability (start small and grow), and architectural simplicity (use off-the-shelf components, such as proxies or HTTP routers). You would choose to use JAX-RS for your web application because it is easier for many types of clients to consume RESTful web services while enabling the server side to evolve and scale. Clients can choose to consume some or all aspects of the service and mash it up with other web-based services.

Java API for XML Web Services (JAX-WS) is a technology for building web services and clients that communicate using XML. JAX-WS allows developers to write message-oriented as well as Remote Procedure Call-oriented (RPC-oriented) web services.

In JAX-WS, a web service operation invocation is represented by an XML-based protocol, such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Although SOAP messages are complex, the JAX-WS API hides this complexity from the application developer. On the server side, the developer specifies the web service operations by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy. With JAX-WS, the developer does not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-WS, clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-WS is not restrictive: A JAX-WS client can access a web service that is not running on the Java platform, and vice versa. This flexibility is possible because JAX-WS uses technologies defined by the W3C: HTTP, SOAP, and WSDL. WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

Note – Several files in the JAX-WS examples depend on the port that you specified when you installed the GlassFish Server. These tutorial examples assume that the server runs on the default port, 8080. They do not run with a nondefault port setting.
The following topics are addressed here:

- “Creating a Simple Web Service and Clients with JAX-WS” on page 368
- “Types Supported by JAX-WS” on page 377
- “Web Services Interoperability and JAX-WS” on page 379
- “Further Information about JAX-WS” on page 379

Creating a Simple Web Service and Clients with JAX-WS

This section shows how to build and deploy a simple web service and two clients: an application client and a web client. The source code for the service is in the `tut-install/examples/jaxws/helloservice/` directory, and the clients are in the `tut-install/examples/jaxws/appclient/` and `tut-install/examples/jaxws/webclient/` directories.

**Figure 19–1** illustrates how JAX-WS technology manages communication between a web service and a client.

The starting point for developing a JAX-WS web service is a Java class annotated with the `javax.jws.WebService` annotation. The `@WebService` annotation defines the class as a web service endpoint.

A *service endpoint interface* or *service endpoint implementation* (SEI) is a Java interface or class, respectively, that declares the methods that a client can invoke on the service. An interface is not required when building a JAX-WS endpoint. The web service implementation class implicitly defines an SEI.

You may specify an explicit interface by adding the `endpointInterface` element to the `@WebService` annotation in the implementation class. You must then provide an interface that defines the public methods made available in the endpoint implementation class.

The basic steps for creating a web service and client are as follows:

1. Code the implementation class.
2. Compile the implementation class.
3. Package the files into a WAR file.
4. Deploy the WAR file. The web service artifacts, which are used to communicate with clients, are generated by the GlassFish Server during deployment.

5. Code the client class.

6. Use a wsimport Ant task to generate and compile the web service artifacts needed to connect to the service.

7. Compile the client class.

8. Run the client.

If you use NetBeans IDE to create a service and client, the IDE performs the wsimport task for you.

The sections that follow cover these steps in greater detail.

### Requirements of a JAX-WS Endpoint

JAX-WS endpoints must follow these requirements.

- The implementing class must be annotated with either the javax.jws.WebService or the javax.jws.WebServiceProvider annotation.
- The implementing class may explicitly reference an SEI through the endpointInterface element of the @WebService annotation but is not required to do so. If no endpointInterface is specified in @WebService, an SEI is implicitly defined for the implementing class.
- The business methods of the implementing class must be public and must not be declared static or final.
- Business methods that are exposed to web service clients must be annotated with javax.jws.WebMethod.
- Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types. See the list of JAXB default data type bindings at [http://docs.oracle.com/javaee/5/tutorial/doc/bnazq.html#bnazs](http://docs.oracle.com/javaee/5/tutorial/doc/bnazq.html#bnazs).
- The implementing class must not be declared final and must not be abstract.
- The implementing class must have a default public constructor.
- The implementing class must not define the finalize method.
- The implementing class may use the javax.annotation.PostConstruct or the javax.annotation.PreDestroy annotations on its methods for lifecycle event callbacks.

  - The @PostConstruct method is called by the container before the implementing class begins responding to web service clients.
  - The @PreDestroy method is called by the container before the endpoint is removed from operation.
Coding the Service Endpoint Implementation Class

In this example, the implementation class, `Hello`, is annotated as a web service endpoint using the `@WebService` annotation. `Hello` declares a single method named `sayHello`, annotated with the `@WebMethod` annotation, which exposes the annotated method to web service clients. The `sayHello` method returns a greeting to the client, using the name passed to it to compose the greeting. The implementation class also must define a default, public, no-argument constructor.

```java
package helloservice.endpoint;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Hello {
    private String message = new String("Hello, ");

    public void Hello() {
    }

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Building, Packaging, and Deploying the Service

You can use either NetBeans IDE or Ant to build, package, and deploy the `helloservice` application.

▼ To Build, Package, and Deploy the Service Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/jaxws/`

3. Select the `helloservice` folder.

4. Select the Open as Main Project check box.

5. Click Open Project.
In the Projects tab, right-click the helloservice project and select Deploy. This command builds and packages the application into helloservice.war, located in tut-install/examples/jaxws/helloservice/dist/, and deploys this WAR file to the GlassFish Server.

You can view the WSDL file of the deployed service by requesting the URL http://localhost:8080/helloservice/HelloService?wsdl in a web browser. Now you are ready to create a client that accesses this service.

▼ To Build, Package, and Deploy the Service Using Ant

1 In a terminal window, go to:
   tut-install/examples/jaxws/helloservice/

2 Type the following command:
   ant
   This command calls the default target, which builds and packages the application into a WAR file, helloservice.war, located in the dist directory.

3 Make sure that the GlassFish Server is started.

4 Type the following:
   ant deploy
   You can view the WSDL file of the deployed service by requesting the URL http://localhost:8080/helloservice/HelloService?wsdl in a web browser. Now you are ready to create a client that accesses this service.

Testing the Methods of a Web Service Endpoint

GlassFish Server allows you to test the methods of a web service endpoint.

▼ To Test the Service without a Client

To test the sayHello method of HelloService, follow these steps.

1 Open the web service test interface by typing the following URL in a web browser:
   http://localhost:8080/helloservice/HelloService?Tester

2 Under Methods, type a name as the parameter to the sayHello method.
3  **Click the sayHello button.**

This takes you to the sayHello Method invocation page.

Under Method returned, you’ll see the response from the endpoint.

## A Simple JAX-WS Application Client

The HelloAppClient class is a stand-alone application client that accesses the sayHello method of HelloService. This call is made through a port, a local object that acts as a proxy for the remote service. The port is created at development time by the wsimport task, which generates JAX-WS portable artifacts based on a WSDL file.

### Coding the Application Client

When invoking the remote methods on the port, the client performs these steps:

1. Uses the generated helloservice.endpoint.HelloService class, which represents the service at the URI of the deployed service’s WSDL file:

   ```java
   import helloservice.endpoint.HelloService;
   import javax.xml.ws.WebServiceRef;

   public class HelloAppClient {
     @WebServiceRef(wsdlLocation =
                   "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
     private static HelloService service;
   }
   ``

2. Retrieves a proxy to the service, also known as a port, by invoking getHelloPort on the service:

   ```java
   helloservice.endpoint.Hello port = service.getHelloPort();
   ``

   The port implements the SEI defined by the service.

3. Invokes the port’s sayHello method, passing a string to the service:

   ```java
   return port.sayHello(arg0);
   ``

Here is the full source of HelloAppClient, which is located in the following directory:

```
tut-install/examples/jaxws/appclient/src/appclient/
```

```java
package appclient;

import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
                   "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private static HelloService service;
}"
```
/**
 * @param args the command line arguments
 */

public static void main(String[] args) {
    System.out.println(sayHello("world"));
}

private static String sayHello(java.lang.String arg0) {
    helloservice.endpoint.Hello port = service.getHelloPort();
    return port.sayHello(arg0);
}

Running the Application Client
You can use either NetBeans IDE or Ant to build, package, deploy, and run the appclient application. To build the client, you must first have deployed helloservice, as described in "Building, Packaging, and Deploying the Service" on page 370.

▼ To Run the Application Client Using NetBeans IDE

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to:
   tut-install/examples/jaxws/

3 Select the appclient folder.

4 Select the Open as Main Project check box.

5 Click Open Project.

6 In the Projects tab, right-click the appclient project and select Run.
   You will see the output of the application client in the Output pane.

▼ To Run the Application Client Using Ant

1 In a terminal window, go to:
   tut-install/examples/jaxws/appclient/

2 Type the following command:
   ant
   This command calls the default target, which runs the wsimport task and builds and packages the application into a JAR file, appclient.jar, located in the dist directory.
3. Type the following command:
   
   ```java
   ant getclient
   ```
   
   This command deploys the app-client.jar file and retrieves the client stubs.

4. To run the client, type the following command:
   
   ```java
   ant run
   ```

A Simple JAX-WS Web Client

HelloServlet is a servlet that, like the Java client, calls the `sayHello` method of the web service. Like the application client, it makes this call through a port.

Coding the Servlet

To invoke the method on the port, the client performs these steps:

1. Imports the `HelloService` endpoint and the `WebServiceRef` annotation:

   ```java
   import helloservice.endpoint.HelloService;
   ...
   import javax.xml.ws.WebServiceRef;
   ```

2. Defines a reference to the web service by specifying the WSDL location:

   ```java
   @WebServiceRef(wsdlLocation =
                   "WEB-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
   ```

3. Declares the web service, then defines a private method that calls the `sayHello` method on the port:

   ```java
   private HelloService service;
   ...
   private String sayHello(java.lang.String arg0) {
       helloservice.endpoint.Hello port = service.getHelloPort();
       return port.sayHello(arg0);
   }
   ```

4. In the servlet, calls this private method:

   ```java
   out.println("<p>" + sayHello("world") + "</p>");
   ```

The significant parts of the `HelloServlet` code follow. The code is located in the `tut-install/examples/jaxws/src/java/webclient/` directory.

```java
package webclient;

import helloservice.endpoint.HelloService;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;

@WebServlet(name="HelloServlet", urlPatterns={"/HelloServlet"})
public class HelloServlet extends HttpServlet {
    @WebServiceRef(wsdlLocation =
        "WEB-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private HelloService service;

    /**
     * Processes requests for both HTTP <code>GET</code> and
     * <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html lang="en">");
            out.println("<head>");
            out.println("<title>Servlet HelloServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet HelloServlet at " +
                request.getContextPath () + "</h1>");
            out.println("<p>" + sayHello("world") + "</p>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    // doGet and doPost methods, which call processRequest, and
    // getServletInfo method

    private String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}

**Running the Web Client**

You can use either NetBeans IDE or Ant to build, package, deploy, and run the webclient application. To build the client, you must first have deployed helloservice, as described in "Building, Packaging, and Deploying the Service" on page 370.
To Run the Web Client Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   tut-install/examples/jaxws/

3. Select the webclient folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the webclient project and select Deploy.
   This task runs the wsimport tasks, builds and packages the application into a WAR file,
   webclient.war, located in the dist directory, and deploys it to the server.

7. In a web browser, navigate to the following URL:
   http://localhost:8080/webclient/HelloServlet
   The output of the sayHello method appears in the window.

To Run the Web Client Using Ant

1. In a terminal window, go to:
   tut-install/examples/jaxws/webclient/

2. Type the following command:
   ant
   This command calls the default target, which runs the wsimport tasks, then builds and
   packages the application into a WAR file, webclient.war, located in the dist directory.

3. Type the following command:
   ant deploy
   This task deploys the WAR file to the server.

4. In a web browser, navigate to the following URL:
   http://localhost:8080/webclient/HelloServlet
   The output of the sayHello method appears in the window.
Types Supported by JAX-WS

JAX-WS delegates the mapping of Java programming language types to and from XML definitions to JAXB. Application developers don’t need to know the details of these mappings but should be aware that not every class in the Java language can be used as a method parameter or return type in JAX-WS.

The following sections explain the default schema-to-Java and Java-to-schema data type bindings.

Schema-to-Java Mapping

The Java language provides a richer set of data type than XML schema. Table 19–1 lists the mapping of XML data types to Java data types in JAXB.

<table>
<thead>
<tr>
<th>XML Schema Type</th>
<th>Java Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:string</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>xsd:integer</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>xsd:int</td>
<td>int</td>
</tr>
<tr>
<td>xsd:long</td>
<td>long</td>
</tr>
<tr>
<td>xsd:short</td>
<td>short</td>
</tr>
<tr>
<td>xsd:decimal</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>xsd:float</td>
<td>float</td>
</tr>
<tr>
<td>xsd:double</td>
<td>double</td>
</tr>
<tr>
<td>xsd:boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>xsd:byte</td>
<td>byte</td>
</tr>
<tr>
<td>xsd:QName</td>
<td>javax.xml.namespace.QName</td>
</tr>
<tr>
<td>xsd:dateTime</td>
<td>javax.xml.datatype.XMLGregorianCalendar</td>
</tr>
<tr>
<td>xsd:base64Binary</td>
<td>byte[]</td>
</tr>
<tr>
<td>xsd:hexBinary</td>
<td>byte[]</td>
</tr>
<tr>
<td>xsd:unsignedInt</td>
<td>long</td>
</tr>
<tr>
<td>xsd:unsignedShort</td>
<td>int</td>
</tr>
<tr>
<td>xsd:unsignedByte</td>
<td>short</td>
</tr>
</tbody>
</table>
### Java-to-Schema Mapping

Table 19–2 shows the default mapping of Java classes to XML data types.

<table>
<thead>
<tr>
<th>Java Class</th>
<th>XML Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>xs:string</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>xs:integer</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>xs:decimal</td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>xs:dateTime</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>xs:dateTime</td>
</tr>
<tr>
<td>javax.xml.namespace.QName</td>
<td>xs:QName</td>
</tr>
<tr>
<td>java.net.URI</td>
<td>xs:string</td>
</tr>
<tr>
<td>javax.xml.datatype.XMLGregorianCalendar</td>
<td></td>
</tr>
<tr>
<td>xs:dateTime</td>
<td></td>
</tr>
<tr>
<td>javax.xml.datatype.Duration</td>
<td>xs:duration</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>xs:anyType</td>
</tr>
<tr>
<td>java.awt.Image</td>
<td>xs:base64Binary</td>
</tr>
<tr>
<td>javax.activation.DataHandler</td>
<td>xs:base64Binary</td>
</tr>
<tr>
<td>javax.xml.transform.Source</td>
<td>xs:base64Binary</td>
</tr>
<tr>
<td>java.util.UUID</td>
<td>xs:string</td>
</tr>
</tbody>
</table>
Web Services Interoperability and JAX-WS

JAX-WS supports the Web Services Interoperability (WS-I) Basic Profile Version 1.1. The WS-I Basic Profile is a document that clarifies the SOAP 1.1 and WSDL 1.1 specifications to promote SOAP interoperability. For links related to WS-I, see “Further Information about JAX-WS” on page 379.

To support WS-I Basic Profile Version 1.1, the JAX-WS runtime supports doc/literal and rpc/literal encodings for services, static ports, dynamic proxies, and the Dynamic Invocation Interface (DII).

Further Information about JAX-WS

For more information about JAX-WS and related technologies, see

- Java API for XML Web Services 2.2 specification:
  http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index4.html
- JAX-WS home:
  http://jax-ws.java.net/
- Simple Object Access Protocol (SOAP) 1.2 W3C Note:
  http://www.w3.org/TR/soap/
- Web Services Description Language (WSDL) 1.1 W3C Note:
  http://www.w3.org/TR/wsd1
- WS-I Basic Profile 1.1:
  http://www.ws-i.org
Building RESTful Web Services with JAX-RS

This chapter describes the REST architecture, RESTful web services, and the Java API for RESTful Web Services (JAX-RS, defined in JSR 311).

Jersey, the reference implementation of JAX-RS, implements support for the annotations defined in JSR 311, making it easy for developers to build RESTful web services by using the Java programming language.

If you are developing with GlassFish Server, you can install the Jersey samples and documentation by using the Update Tool. Instructions for using the Update Tool can be found in "Java EE 6 Tutorial Component" on page 70. The Jersey samples and documentation are provided in the Available Add-ons area of the Update Tool.

The following topics are addressed here:

■ "What Are RESTful Web Services?" on page 381
■ "Creating a RESTful Root Resource Class" on page 382
■ "Example Applications for JAX-RS" on page 396
■ "Further Information about JAX-RS" on page 401

What Are RESTful Web Services?

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.
The following principles encourage RESTful applications to be simple, lightweight, and fast:

- **Resource identification through URI**: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See “The @Path Annotation and URI Path Templates” on page 385 for more information.

- **Uniform interface**: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can then be deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See “Responding to HTTP Methods and Requests” on page 387 for more information.

- **Self-descriptive messages**: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See “Responding to HTTP Methods and Requests” on page 387 and “Using Entity Providers to Map HTTP Response and Request Entity Bodies” on page 389 for more information.

- **Stateful interactions through hyperlinks**: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See “Using Entity Providers to Map HTTP Response and Request Entity Bodies” on page 389 and “Building URIs” in the JAX-RS Overview document for more information.

## Creating a RESTful Root Resource Class

**Root resource classes** are POJOs that are either annotated with @Path or have at least one method annotated with @Path or a request method designator, such as @GET, @PUT, @POST, or @DELETE. **Resource methods** are methods of a resource class annotated with a request method designator.

This section explains how to use JAX-RS to annotate Java classes to create RESTful web services.

## Developing RESTful Web Services with JAX-RS

JAX-RS is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with JAX-RS annotations to define resources and the actions that can be performed on those resources.

JAX-RS annotations are runtime annotations; therefore, runtime reflection will generate the
helper classes and artifacts for the resource. A Java EE application archive containing JAX-RS resource classes will have the resources configured, the helper classes and artifacts generated, and the resource exposed to clients by deploying the archive to a Java EE server.

Table 20–1 lists some of the Java programming annotations that are defined by JAX-RS, with a brief description of how each is used. Further information on the JAX-RS APIs can be viewed at http://docs.oracle.com/javaee/6/api/.

### Table 20–1 Summary of JAX-RS Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Path</td>
<td>The @Path annotation’s value is a relative URI path indicating where the Java class will be hosted: for example, /helloworld. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: /helloworld/{username}.</td>
</tr>
<tr>
<td>@GET</td>
<td>The @GET annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.</td>
</tr>
<tr>
<td>@POST</td>
<td>The @POST annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.</td>
</tr>
<tr>
<td>@PUT</td>
<td>The @PUT annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.</td>
</tr>
<tr>
<td>@DELETE</td>
<td>The @DELETE annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.</td>
</tr>
<tr>
<td>@HEAD</td>
<td>The @HEAD annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.</td>
</tr>
<tr>
<td>@PathParam</td>
<td>The @PathParam annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Path class-level annotation.</td>
</tr>
<tr>
<td>@QueryParam</td>
<td>The @QueryParam annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.</td>
</tr>
</tbody>
</table>
TABLE 20–1 Summary of JAX-RS Annotations  (Continued)

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Consumes</td>
<td>The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.</td>
</tr>
<tr>
<td>@Produces</td>
<td>The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, &quot;text/plain&quot;.</td>
</tr>
<tr>
<td>@Provider</td>
<td>The @Provider annotation is used for anything that is of interest to the JAX-RS runtime, such as MessageBodyReader and MessageBodyWriter. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity and that can be built using Response.ResponseBuilder.</td>
</tr>
</tbody>
</table>

Overview of a JAX-RS Application

The following code sample is a very simple example of a root resource class that uses JAX-RS annotations:

```java
text
package com.sun.jersey.samples.helloworld.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;

// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        // Return some cliched textual content
        return "Hello World";
    }
}
```
The following sections describe the annotations used in this example.

- The `@Path` annotation's value is a relative URI path. In the preceding example, the Java class will be hosted at the URI path `/helloworld`. This is an extremely simple use of the `@Path` annotation, with a static URI path. Variables can be embedded in the URIs. *URI path templates* are URIs with variables embedded within the URI syntax.

- The `@GET` annotation is a request method designator, along with `@POST`, `@PUT`, `@DELETE`, and `@HEAD`, defined by JAX-RS and corresponding to the similarly named HTTP methods. In the example, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.

- The `@Produces` annotation is used to specify the MIME media types a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "text/plain".

- The `@Consumes` annotation is used to specify the MIME media types a resource can consume that were sent by the client. The example could be modified to set the message returned by the `getClichedMessage` method, as shown in this code example:

```java
@POST
@Consumes("text/plain")
public void postClichedMessage(String message) {
    // Store the message
}
```

**The @Path Annotation and URI Path Templates**

The `@Path` annotation identifies the URI path template to which the resource responds and is specified at the class or method level of a resource. The `@Path` annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the application, and the URL pattern to which the JAX-RS runtime responds.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by braces (`{ and `}`). For example, look at the following `@Path` annotation:

```java
@Path("/users/{username}")
```

In this kind of example, a user is prompted to type his or her name, and then a JAX-RS web service configured to respond to requests to this URI path template responds. For example, if the user types the username “Galileo,” the web service responds to the following URL:

http://example.com/users/Galileo

To obtain the value of the user name, the `@PathParam` annotation may be used on the method parameter of a request method, as shown in the following code example:
@Path("/users/{username}\")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {

        ...
    }
}

By default, the URI variable must match the regular expression "[^/]+?". This variable may be customized by specifying a different regular expression after the variable name. For example, if a user name must consist only of lowercase and uppercase alphanumeric characters, override the default regular expression in the variable definition:

@Path("/users/{username: [a-zA-Z][a-zA-Z_0-9]*}\")

In this example the username variable will match only user names that begin with one uppercase or lowercase letter and zero or more alphanumeric characters and the underscore character. If a user name does not match that template, a 404 (Not Found) response will be sent to the client.

A @Path value isn’t required to have leading or trailing slashes (/). The JAX-RS runtime parses URI path templates the same whether or not they have leading or trailing spaces.

A URI path template has one or more variables, with each variable name surrounded by braces: { to begin the variable name and } to end it. In the preceding example, username is the variable name. At runtime, a resource configured to respond to the preceding URI path template will attempt to process the URI data that corresponds to the location of {username} in the URI as the variable data for username.

For example, if you want to deploy a resource that responds to the URI path template http://example.com/myContextRoot/resources/{name1}/{name2}/, you must deploy the application to a Java EE server that responds to requests to the http://example.com/myContextRoot URI and then decorate your resource with the following @Path annotation:

@Path("/\{name1\}/\{name2\}/\")
public class SomeResource {

    ...
}

In this example, the URL pattern for the JAX-RS helper servlet, specified in web.xml, is the default:

<url-pattern>/resources/</url-pattern>

A variable name can be used more than once in the URI path template.
If a character in the value of a variable would conflict with the reserved characters of a URI, the conflicting character should be substituted with percent encoding. For example, spaces in the value of a variable should be substituted with `%20`.

When defining URI path templates, be careful that the resulting URI after substitution is valid.

Table 20–2 lists some examples of URI path template variables and how the URIs are resolved after substitution. The following variable names and values are used in the examples:

- `name1`: james
- `name2`: gatz
- `name3`:
- `location`: Main%20Street
- `question`: why

Note – The value of the `name3` variable is an empty string.

<table>
<thead>
<tr>
<th>URI Path Template</th>
<th>URI After Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://example.com/%7Bname1%7D/%7Bname2%7D/">http://example.com/{name1}/{name2}/</a></td>
<td><a href="http://example.com/james/gatz/">http://example.com/james/gatz/</a></td>
</tr>
<tr>
<td>(question)/(question)/</td>
<td></td>
</tr>
<tr>
<td><a href="http://example.com/maps/%7Blocation%7D">http://example.com/maps/{location}</a></td>
<td><a href="http://example.com/maps/Main%20Street">http://example.com/maps/Main%20Street</a></td>
</tr>
<tr>
<td><a href="http://example.com/%7Bname3%7D/home/">http://example.com/{name3}/home/</a></td>
<td><a href="http://example.com//home/">http://example.com//home/</a></td>
</tr>
</tbody>
</table>

**Responding to HTTP Methods and Requests**

The behavior of a resource is determined by the HTTP methods (typically, `GET`, `POST`, `PUT`, `DELETE`) to which the resource is responding.

**The Request Method Designator Annotations**

Request method designator annotations are runtime annotations, defined by JAX-RS, that correspond to the similarly named HTTP methods. Within a resource class file, HTTP methods are mapped to Java programming language methods by using the request method designator annotations. The behavior of a resource is determined by which HTTP method the resource is responding to. JAX-RS defines a set of request method designators for the common HTTP methods `@GET`, `@POST`, `@PUT`, `@DELETE`, and `@HEAD`; you can also create your own custom request method designators. Creating custom request method designators is outside the scope of this document.
The following example, an extract from the storage service sample, shows the use of the PUT method to create or update a storage container:

```java
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);
    URI uri = uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());
    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }
    MemoryStore.MS.createContainer(c);
    return r;
}
```

By default, the JAX-RS runtime will automatically support the methods HEAD and OPTIONS if not explicitly implemented. For HEAD, the runtime will invoke the implemented GET method, if present, and ignore the response entity, if set. For OPTIONS, the Allow response header will be set to the set of HTTP methods supported by the resource. In addition, the JAX-RS runtime will return a Web Application Definition Language (WADL) document describing the resource; see [http://www.w3.org/Submission/wadl/](http://www.w3.org/Submission/wadl/) for more information.

Methods decorated with request method designators must return void, a Java programming language type, or a javax.ws.rs.core.Response object. Multiple parameters may be extracted from the URI by using the @PathParam or @QueryParam annotations as described in "Extracting Request Parameters" on page 392. Conversion between Java types and an entity body is the responsibility of an entity provider, such as MessageBodyReader or MessageBodyWriter. Methods that need to provide additional metadata with a response should return an instance of the Response class. The ResponseBuilder class provides a convenient way to create a Response instance using a builder pattern. The HTTP PUT and POST methods expect an HTTP request body, so you should use a MessageBodyReader for methods that respond to PUT and POST requests.

Both @PUT and @POST can be used to create or update a resource. POST can mean anything, so when using POST, it is up to the application to define the semantics. PUT has well-defined semantics. When using PUT for creation, the client declares the URI for the newly created resource.

PUT has very clear semantics for creating and updating a resource. The representation the client sends must be the same representation that is received using a GET, given the same media type. PUT does not allow a resource to be partially updated, a common mistake when attempting to use the PUT method. A common application pattern is to use POST to create a resource and return a 201 response with a location header whose value is the URI to the newly created resource. In this pattern, the web service declares the URI for the newly created resource.
Using Entity Providers to Map HTTP Response and Request Entity Bodies

*Entity providers* supply mapping services between representations and their associated Java types. The two types of entity providers are `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built by using `Response.ResponseBuilder`.

Table 20–3 shows the standard types that are supported automatically for HTTP request and response entity bodies. You need to write an entity provider only if you are not choosing one of these standard types.

**Table 20–3 Types Supported for HTTP Request and Response Entity Bodies**

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Supported Media Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byte[]</code></td>
<td>All media types (<em>/</em>)</td>
</tr>
<tr>
<td><code>java.lang.String</code></td>
<td>All text media types (text/*)</td>
</tr>
<tr>
<td><code>java.io.InputStream</code></td>
<td>All media types (<em>/</em>)</td>
</tr>
<tr>
<td><code>java.io.Reader</code></td>
<td>All media types (<em>/</em>)</td>
</tr>
<tr>
<td><code>java.io.File</code></td>
<td>All media types (<em>/</em>)</td>
</tr>
<tr>
<td><code>javax.activation.DataSource</code></td>
<td>All media types (<em>/</em>)</td>
</tr>
<tr>
<td><code>javax.xml.transform.Source</code></td>
<td>XML media types (text/xml, application/xml, and application/*+xml)</td>
</tr>
<tr>
<td><code>javax.xml.bind.JAXBElement</code> and application-supplied JAXB classes</td>
<td>XML media types (text/xml, application/xml, and application/*+xml)</td>
</tr>
<tr>
<td><code>MultivaluedMap&lt;String, String&gt;</code></td>
<td>Form content (application/x-www-form-urlencoded)</td>
</tr>
<tr>
<td><code>StreamingOutput</code></td>
<td>All media types (<em>/</em>), <code>MessageBodyWriter</code> only</td>
</tr>
</tbody>
</table>

The following example shows how to use `MessageBodyReader` with the `@Consumes` and `@Provider` annotations:

```java
@Consumes("application/x-www-form-urlencoded")
@Provider
public class FormReader implements MessageBodyReader<NameValuePair> {
```

The following example shows how to use `MessageBodyWriter` with the `@Produces` and `@Provider` annotations:
@Produces("text/html")
@Provider
public class FormWriter implements MessageBodyWriter<Hashtable<String, String>> {

The following example shows how to use ResponseBuilder:

@GET
public Response getItem() {
    System.out.println("GET ITEM" + container + "" + item);
    Item i = MemoryStore.MS.getItem(container, item);
    if (i == null)
        throw new NotFoundException("Item not found");
    Date lastModified = i.getLastModified().getTime();
    EntityTag et = new EntityTag(i.getDigest());
    ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);
    if (rb != null)
        return rb.build();
    byte[] b = MemoryStore.MS.getItemData(container, item);
    return Response.ok(b, i.getMimeType()).
        lastModified(lastModified).tag(et).build();
}

Using @Consumes and @Produces to Customize Requests and Responses

The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response. You can specify which MIME media types of representations a resource can respond to or produce by using the following annotations:

- javax.ws.rs.Consumes
- javax.ws.rs.Produces

By default, a resource class can respond to and produce all MIME media types of representations specified in the HTTP request and response headers.

The @Produces Annotation

The @Produces annotation is used to specify the MIME media types or representations a resource can produce and send back to the client. If @Produces is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If applied at the method level, the annotation overrides any @Produces annotations applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the JAX-RS runtime sends back an HTTP "406 Not Acceptable" error.

The value of @Produces is an array of String of MIME types. For example:
@Produces({"image/jpeg,image/png"})

The following example shows how to apply @Produces at both the class and method levels:

@Path("/myResource")
@Produces("text/plain")
public class SomeResource {

    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}

The doGetAsPlainText method defaults to the MIME media type of the @Produces annotation at the class level. The doGetAsHtml method's @Produces annotation overrides the class-level @Produces setting and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing more than one MIME media type, the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically, the Accept header of the HTTP request declares what is most acceptable. For example, if the Accept header is Accept: text/plain, the doGetAsPlainText method will be invoked. Alternatively, if the Accept header is Accept: text/plain;q=0.9, text/html, which declares that the client can accept media types of text/plain and text/html but prefers the latter, the doGetAsHtml method will be invoked.

More than one media type may be declared in the same @Produces declaration. The following code example shows how this is done:

@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}

The doGetAsXmlOrJson method will get invoked if either of the media types application/xml and application/json is acceptable. If both are equally acceptable, the former will be chosen because it occurs first. The preceding examples refer explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors. For more information, see the constant field values of MediaType at [http://jsr311.java.net/nonav/releases/1.0/javax/ws/rs/core/MediaType.html](http://jsr311.java.net/nonav/releases/1.0/javax/ws/rs/core/MediaType.html).

The @Consumes Annotation

The @Consumes annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. If @Consumes is applied at the class level, all the response methods accept the specified MIME types by default. If applied at the method level, @Consumes overrides any @Consumes annotations applied at the class level.
If a resource is unable to consume the MIME type of a client request, the JAX-RS runtime sends back an HTTP 415 ("Unsupported Media Type") error.

The value of @Consumes is an array of String of acceptable MIME types. For example:

```java
@Consumes({"text/plain,text/html"})
```

The following example shows how to apply @Consumes at both the class and method levels:

```java
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
    @POST
    public String doPost(MimeMultipart mimeMultipartData) {
        ...
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String doPost2(FormURLEncodedProperties formData) {
        ...
    }
}
```

The doPost method defaults to the MIME media type of the @Consumes annotation at the class level. The doPost2 method overrides the class level @Consumes annotation to specify that it can accept URL-encoded form data.

If no resource methods can respond to the requested MIME type, an HTTP 415 ("Unsupported MIME Type") error is returned to the client.

The HelloWorld example discussed previously in this section can be modified to set the message by using @Consumes, as shown in the following code example:

```java
@POST
@Consumes("text/plain")
public void postClichedMessage(String message) {
    // Store the message
}
```

In this example, the Java method will consume representations identified by the MIME media type text/plain. Note that the resource method returns void. This means that no representation is returned and that a response with a status code of HTTP 204 ("No Content") will be returned.

### Extracting Request Parameters

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use of the @PathParam parameter to extract a path parameter from the path component of the request URL that matched the path declared in @Path.
You can extract the following types of parameters for use in your resource class:

- Query
- URI path
- Form
- Cookie
- Header
- Matrix

Query parameters are extracted from the request URI query parameters and are specified by using the javax.ws.rs.QueryParam annotation in the method parameter arguments. The following example, from the sparklines sample application, demonstrates using @QueryParam to extract query parameters from the Query component of the request URL:

```java
@Path("smooth")
@GET
public Response smooth(
    @DefaultValue("2") @QueryParam("step") int step,
    @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
    @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
    @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
    @DefaultValue("true") @QueryParam("min-color") ColorParam minColor,
    @DefaultValue("true") @QueryParam("max-color") ColorParam maxColor,
    @DefaultValue("true") @QueryParam("last-color") ColorParam lastColor
) { ... }
```

If the query parameter `step` exists in the query component of the request URI, the value of `step` will be extracted and parsed as a 32-bit signed integer and assigned to the `step` method parameter. If `step` does not exist, a default value of 2, as declared in the @DefaultValue annotation, will be assigned to the `step` method parameter. If the `step` value cannot be parsed as a 32-bit signed integer, an HTTP 400 (“Client Error”) response is returned.

User-defined Java programming language types may be used as query parameters. The following code example shows the ColorParam class used in the preceding query parameter example:

```java
public class ColorParam extends Color {
    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        } else {
            try {
                Field f = Color.class.getField(s);
```
The constructor for ColorParam takes a single String parameter.

Both @QueryParam and @PathParam can be used only on the following Java types:

- All primitive types except char
- All wrapper classes of primitive types except Character
- Any class with a constructor that accepts a single String argument
- Any class with the static method named valueOf(String) that accepts a single String argument
- List<T>, Set<T>, or SortedSet<T>, where T matches the already listed criteria. Sometimes, parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values.

If @DefaultValue is not used in conjunction with @QueryParam, and the query parameter is not present in the request, the value will be an empty collection for List, Set, or SortedSet; null for other object types; and the default for primitive types.

URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Path class-level annotation. URI parameters are specified using the javax.ws.rsPathParam annotation in the method parameter arguments. The following example shows how to use @Path variables and the @PathParam annotation in a method:

```java
@Path("/{username}")
public class MyResourceBean {  
    
    @GET  
    public String printUsername(@PathParam("username") String userId) {  
        ...  
    }  
}
```

In the preceding snippet, the URI path template variable name username is specified as a parameter to the printUsername method. The @PathParam annotation is set to the variable name username. At runtime, before printUsername is called, the value of username is extracted from the URI and cast to a String. The resulting String is then available to the method as the userId variable.

If the URI path template variable cannot be cast to the specified type, the JAX-RS runtime returns an HTTP 400 ("Bad Request") error to the client. If the @PathParam annotation cannot be cast to the specified type, the JAX-RS runtime returns an HTTP 404 ("Not Found") error to the client.
The `@PathParam` parameter and the other parameter-based annotations (`@MatrixParam`, `@HeaderParam`, `@CookieParam`, and `@FormParam`) obey the same rules as `@QueryParam`.

Cookie parameters, indicated by decorating the parameter with `javax.ws.rs.CookieParam`, extract information from the cookies declared in cookie-related HTTP headers. Header parameters, indicated by decorating the parameter with `javax.ws.rs.HeaderParam`, extract information from the HTTP headers. Matrix parameters, indicated by decorating the parameter with `javax.ws.rs.MatrixParam`, extract information from URL path segments.

Form parameters, indicated by decorating the parameter with `javax.ws.rs.FormParam`, extract information from a request representation that is of the MIME media type `application/x-www-form-urlencoded` and conforms to the encoding specified by HTML forms, as described in [http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1](http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1). This parameter is very useful for extracting information sent by POST in HTML forms.

The following example extracts the `name` form parameter from the POST form data:

```java
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    // Store the message
}
```

To obtain a general map of parameter names and values for query and path parameters, use the following code:

```java
@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

The following method extracts header and cookie parameter names and values into a map:

```java
@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    Map<String, Cookie> pathParams = hh.getCookies();
}
```

In general, `@Context` can be used to obtain contextual Java types related to the request or response.

For form parameters, it is possible to do the following:

```java
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Store the message
}
```
Example Applications for JAX-RS

This section provides an introduction to creating, deploying, and running your own JAX-RS applications. This section demonstrates the steps that are needed to create, build, deploy, and test a very simple web application that uses JAX-RS annotations.

A RESTful Web Service

This section explains how to use NetBeans IDE to create a RESTful web service. NetBeans IDE generates a skeleton for the application, and you simply need to implement the appropriate methods. If you do not use an IDE, try using one of the example applications that ship with Jersey as a template to modify.

You can find a version of this application at tut-install/examples/jaxrs/HelloWorldApplication/.

▼ To Create a RESTful Web Service Using NetBeans IDE

1 In NetBeans IDE, create a simple web application. This example creates a very simple “Hello, World” web application.
   a. From the File menu, choose New Project.
   b. From Categories, select Java Web. From Projects, select Web Application. Click Next.

   Note – For this step, you could also create a RESTful web service in a Maven web project by selecting Maven as the category and Maven Web Project as the project. The remaining steps would be the same.

   c. Type a project name, HelloWorldApplication, and click Next.
   d. Make sure that the Server is GlassFish Server (or similar wording).
   e. Click Finish.

   The project is created. The file index.jsp appears in the Source pane.

2 Right-click the project and select New; then select RESTful Web Services from Patterns.
   a. Select Simple Root Resource and click Next.
   b. Type a Resource Package name, such as helloWorld.
c. Type `helloworld` in the Path field. Type `HelloWorld` in the Class Name field. For MIME Type, select `text/html`.

d. Click Finish.

The REST Resources Configuration page appears.

e. Click OK.

A new resource, `HelloWorld.java`, is added to the project and appears in the Source pane. This file provides a template for creating a RESTful web service.

3 In `HelloWorld.java`, find the `getHtml()` method. Replace the `//TODO` comment and the exception with the following text, so that the finished product resembles the following method.

   ```java
   /**
    * Retrieves representation of an instance of helloWorld.HelloWorld
    * @return an instance of java.lang.String
    */
   @GET
   @Produces("text/html")
   public String getHtml() {
       return "<html lang="en">
       <body><h1>Hello, World!!</h1></body></html>";
   }
   ```

Note – Because the MIME type produced is HTML, you can use HTML tags in your return statement.

4 Test the web service. To do this, right-click the project node and click Test RESTful Web Services.

This step deploys the application and brings up a test client in the browser.

5 When the test client appears, select the `helloworld` resource in the left pane, and click the Test button in the right pane.

The words `Hello, World!!` appear in the Response window below.

6 Set the Run Properties:

   a. Right-click the project node and select Properties.

   b. In the dialog, select the Run category.

   c. Set the Relative URL to the location of the RESTful web service relative to the Context Path, which for this example is `resources/helloworld`. 

...
Tip – You can find the value for the Relative URL in the Test RESTful Web Services browser window. In the top of the right pane, after Resource, is the URL for the RESTful web service being tested. The part following the Context Path (http://localhost:8080/HelloWorldApp) is the Relative URL that needs to be entered here.

If you don’t set this property, the file index.jsp will appear by default when the application is run. As this file also contains Hello World as its default value, you might not notice that your RESTful web service isn’t running, so just be aware of this default and the need to set this property, or update index.jsp to provide a link to the RESTful web service.

7 Right-click the project and select Deploy.

8 Right-click the project and select Run.
A browser window opens and displays the return value of Hello, World!!

See Also For other sample applications that demonstrate deploying and running JAX-RS applications using NetBeans IDE, see “The rsvp Example Application” on page 398 and Your First Cup: An Introduction to the Java EE Platform at http://docs.oracle.com/javaee/6/firstcup/doc/. You may also look at the tutorials on the NetBeans IDE tutorial site, such as the one titled “Getting Started with RESTful Web Services” at http://www.netbeans.org/kb/docs/websvc/rest.html. This tutorial includes a section on creating a CRUD application from a database. Create, read, update, and delete (CRUD) are the four basic functions of persistent storage and relational databases.

The rsvp Example Application

The rsvp example application, located in the tut-install/examples/jaxrs/rsvp/ directory, allows invitees to an event to indicate whether they will attend. The events, people invited to the event, and the responses to the invite are stored in a Java DB database using the Java Persistence API. The JAX-RS resources in rsvp are exposed in a stateless session enterprise bean.

Components of the rsvp Example Application

The three enterprise beans in the rsvp example application are rsvp.ejb.ConfigBean, rsvp.ejb.StatusBean, and rsvp.ejb.ResponseBean.

ConfigBean is a singleton session bean that initializes the data in the database.

StatusBean exposes a JAX-RS resource for displaying the current status of all invitees to an event. The URI path template is declared as follows:

@Path("/status/{eventId}")
The URI path variable eventId is a @PathParam variable in the getResponse method, which responds to HTTP GET requests and has been annotated with @GET. The eventId variable is used to look up all the current responses in the database for that particular event.

ResponseBean exposes a JAX-RS resource for setting an invitee's response to a particular event. The URI path template for ResponseBean is declared as follows:

```java
@Path("/{eventId}/{inviteId}")
```

Two URI path variables are declared in the path template: eventId and inviteId. As in StatusBean, eventId is the unique ID for a particular event. Each invitee to that event has a unique ID for the invitation, and that is the inviteId. Both of these path variables are used in two JAX-RS methods in ResponseBean: getResponse and putResponse. The getResponse method responds to HTTP GET requests and displays the invitee's current response and a form to change the response.

An invitee who wants to change his or her response selects the new response and submits the form data, which is processed as an HTTP PUT request by the putResponse method. One of the parameters to the putResponse method, the userResponse string, is annotated with @FormParam("attendeeResponse"). The HTML form created by getResponse stores the changed response in the select list with an ID of attendeeResponse. The annotation @FormParam("attendeeResponse") indicates that the value of the select response is extracted from the HTTP PUT request and stored as the userResponse string. The putResponse method uses userResponse, eventId, and inviteId to update the invitee's response in the database.

The events, people, and responses in rsvp are encapsulated in Java Persistence API entities. The rsvp.entity.Event, rsvp.entity.Person, and rsvp.entity.Response entities respectively represent events, invitees, and responses to an event.

The rsvp.util.ResponseEnum class declares an enumerated type that represents all the possible response statuses an invitee may have.

**Running the rsvp Example Application**

Both NetBeans IDE and Ant can be used to deploy and run the rsvp example application.

**To Run the rsvp Example Application in NetBeans IDE**

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   
   `tut-install/examples/jaxrs/`

3. Select the rsvp folder.
4. Select the Open as Main Project check box.
5  Click Open Project.

6  Right-click the rsvp project in the left pane and select Run.
   The project will be compiled, assembled, and deployed to GlassFish Server. A web browser
   window will open to http://localhost:8080/rsvp.

7  In the web browser window, click the Event Status link for the Duke's Birthday event.
   You'll see the current invitees and their responses.

8  Click on the name of one of the invitees, select a response, and click Submit response; then click
   Back to event page.
   The invitee's new status should now be displayed in the table of invitees and their response
   statuses.

▼  To Run the rsvp Example Application Using Ant

Before You Begin  You must have started the Java DB database before running rsvp.

1  In a terminal window, go to:
   tut-install/examples/jaxrs/rsvp/

2  Type the following command:
   ant all
   This command builds, assembles, and deploys rsvp to GlassFish Server.

3  Open a web browser window to http://localhost:8080/rsvp.

4  In the web browser window, click the Event Status link for the Duke's Birthday event.
   You'll see the current invitees and their responses.

5  Click on the name of one of the invitees, select a response, and click Submit response, then click
   Back to event page.
   The invitee's new status should now be displayed in the table of invitees and their response
   statuses.

Real-World Examples

Most blog sites use RESTful web services. These sites involve downloading XML files, in RSS or
Atom format, that contain lists of links to other resources. Other web sites and web applications
that use REST-like developer interfaces to data include Twitter and Amazon S3 (Simple Storage
Service). With Amazon S3, buckets and objects can be created, listed, and retrieved using either
a REST-style HTTP interface or a SOAP interface. The examples that ship with Jersey include a
storage service example with a RESTful interface. The tutorial at http://netbeans.org/kb/docs/websvc/twitter-swing.html uses NetBeans IDE to create a simple, graphical, REST-based client that displays Twitter public timeline messages and lets you view and update your Twitter status.

Further Information about JAX-RS

For more information about RESTful web services and JAX-RS, see

- "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision":
- "Fielding Dissertation: Chapter 5: Representational State Transfer (REST)"
- JSR 311: JAX-RS: The Java API for RESTful Web Services:
  http://jcp.org/en/jsr/detail?id=311
- JAX-RS project:
  http://jsr311.java.net/
- Jersey project:
  http://jersey.java.net/
JAX-RS: Advanced Topics and Example

The Java API for RESTful Web Services (JAX-RS, defined in JSR 311) is designed to make it easy to develop applications that use the REST architecture. This chapter describes advanced features of JAX-RS. If you are new to JAX-RS, see Chapter 20, “Building RESTful Web Services with JAX-RS,” before you proceed with this chapter.

JAX-RS is part of the Java EE 6 full profile. JAX-RS is integrated with Contexts and Dependency Injection for the Java EE Platform (CDI), Enterprise JavaBeans (EJB) technology, and Java Servlet technology.

The following topics are addressed here:
- “Annotations for Field and Bean Properties of Resource Classes” on page 403
- “Subresources and Runtime Resource Resolution” on page 407
- “Integrating JAX-RS with EJB Technology and CDI” on page 408
- “Conditional HTTP Requests” on page 409
- “Runtime Content Negotiation” on page 410
- “Using JAX-RS With JAXB” on page 412
- “The customer Example Application” on page 418

Annotations for Field and Bean Properties of Resource Classes

JAX-RS annotations for resource classes let you extract specific parts or values from a Uniform Resource Identifier (URI) or request header.

JAX-RS provides the annotations listed in Table 21–1.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Context</td>
<td>Injects information into a class field, bean property, or method parameter</td>
</tr>
</tbody>
</table>
Annotations for Field and Bean Properties of Resource Classes

TABLE 21–1

Advanced JAX-RS Annotations

(Continued)

Annotation

Description

@CookieParam

Extracts information from cookies declared in the cookie request header

@FormParam

Extracts information from a request representation whose content type is
application/x-www-form-urlencoded

@HeaderParam

Extracts the value of a header

@MatrixParam

Extracts the value of a URI matrix parameter

@PathParam

Extracts the value of a URI template parameter

@QueryParam

Extracts the value of a URI query parameter

Extracting Path Parameters
URI path templates are URIs with variables embedded within the URI syntax. The @PathParam
annotation lets you use variable URI path fragments when you call a method.
The following code snippet shows how to extract the last name of an employee when the
employee’s email address is provided:
@Path(/employees/"{firstname}.{lastname}@{domain}.com")
public class EmpResource {
@GET
@Produces("text/xml")
public String getEmployeelastname(@PathParam("lastname") String lastName) {
...
}
}

In this example, the @Path annotation defines the URI variables (or path parameters)
{firstname} , {lastname}, and {domain}. The @PathParam in the method parameter of the
request method extracts the last name from the email address.
If your HTTP request is GET /employees/john.doe@example.com, the value “doe” is injected
into {lastname}.
You can specify several path parameters in one URI.

404

The Java EE 6 Tutorial • January 2013


You can declare a regular expression with a URI variable. For example, if it is required that the last name must consist only of lower and upper case characters, you can declare the following regular expression:

@Path(/employees/{"firstname"}.{lastname[a-zA-Z]*}@{domain}.com)

If the last name does not match the regular expression, a 404 response is returned.

Extracting Query Parameters

Use the @QueryParam annotation to extract query parameters from the query component of the request URI.

For instance, to query all employees who have joined within a specific range of years, use a method signature like the following:

@Path(/employees/"
@GET
public Response getEmployees(
    @DefaultValue("2002") @QueryParam("minyear") int minyear,
    @DefaultValue("2010") @QueryParam("maxyear") int maxyear)
    
This code snippet defines two query parameters, minyear and maxyear. The following HTTP request would query for all employees who have joined between 1999 and 2009:

GET /employees?maxyear=2009&minyear=1999

The @DefaultValue annotation defines a default value, which is to be used if no values are provided for the query parameters. By default, JAX-RS assigns a null value for object values and zero for primitive data types. You can use the @DefaultValue annotation to eliminate null or zero values and define your own default values for a parameter.

Extracting Form Data

Use the @FormParam annotation to extract form parameters from HTML forms. For example, the following form accepts the name, address, and manager's name of an employee:

<form action="http://example.com/employees/" method="post">
  
  <p>
    Employee name: <input type="text" name="empname" tabindex="1">
  
  Employee address: <input type="text" name="empaddress" tabindex="2">
  
  Manager name: <input type="text" name="managername" tabindex="3">
  
  </fieldset>
  
  </form>
Use the following code snippet to extract the manager name from this HTML form:

```java
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("managername") String managername) {
    // Store the value
    ...
}
```

To obtain a map of form parameter names to values, use a code snippet like the following:

```java
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Store the message
}
```

### Extracting the Java Type of a Request or Response

The `javax.ws.rs.core.Context` annotation retrieves the Java types related to a request or response.

The `javax.ws.rs.core.UriInfo` interface provides information about the components of a request URL. The following code snippet shows how to obtain a map of query and path parameter names to values:

```java
@GET
public String getParams(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

The `javax.ws.rs.core.HttpHeaders` interface provides information about request headers and cookies. The following code snippet shows how to obtain a map of header and cookie parameter names to values:

```java
@GET
public String getHeaders(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    MultivaluedMap<String, Cookie> pathParams = hh.getCookies();
}
```
Subresources and Runtime Resource Resolution

You can use a resource class to process only a part of the URI request. A root resource can then implement subresources that can process the remainder of the URI path.

A resource class method that is annotated with @Path is either a subresource method or a subresource locator:

- A subresource method is used to handle requests on a subresource of the corresponding resource.
- A subresource locator is used to locate subresources of the corresponding resource.

Subresource Methods

A subresource method handles an HTTP request directly. The method must be annotated with a request method designator such as @GET or @POST, in addition to @Path. The method is invoked for request URIs that match a URI template created by concatenating the URI template of the resource class with the URI template of the method.

The following code snippet shows how a subresource method can be used to extract the last name of an employee when the employee’s email address is provided:

```java
@Path("/employeeinfo")
public class EmployeeInfo {
    public EmployeeInfo() {} 

    @GET
    @Path("/employees/{firstname}.{lastname}@{domain}.com")
    @Produces("text/xml")
    public String getEmployeeLastName(@PathParam("lastname") String lastName) {
        ...
    }
}
```

The `getEmployeeLastName` method returns `doe` for the following GET request:

```
GET /employeeinfo/employees/john.doe@example.com
```

Subresource Locators

A subresource locator returns an object that will handle an HTTP request. The method must not be annotated with a request method designator. You must declare a subresource locator within a subresource class, and only subresource locators are used for runtime resource resolution.

The following code snippet shows a subresource locator:
// Root resource class
@Path("/employeeinfo")
public class EmployeeInfo {

    // Subresource locator: obtains the subresource Employee
    // from the path /employeeinfo/employees/{empid}
    @Path("/employees/{empid}")
    public Employee getEmployee(@PathParam("empid") String id) {
        // Find the Employee based on the id path parameter
        Employee emp = ...;
        ...
        return emp;
    }
}

// Subresource class
public class Employee {

    // Subresource method: returns the employee's last name
    @GET
    @Path("/lastname")
    public String getEmployeeLastName() {
        ...
        return lastName
    }
}

In this code snippet, the getEmployee method is the subresource locator that provides the Employee object, which services requests for lastname.

If your HTTP request is GET /employeeinfo/employees/as209/, the getEmployee method returns an Employee object whose id is as209. At runtime, JAX-RS sends a GET /employeeinfo/employees/as209/lastname request to the getEmployeeLastName method. The getEmployeeLastName method retrieves and returns the last name of the employee whose id is as209.

**Integrating JAX-RS with EJB Technology and CDI**

JAX-RS works with Enterprise JavaBeans technology (enterprise beans) and Contexts and Dependency Injection for the Java EE Platform (CDI).

In general, for JAX-RS to work with enterprise beans, you need to annotate the class of a bean with @Path to convert it to a root resource class. You can use the @Path annotation with stateless session beans and singleton POJO beans.

The following code snippet shows a stateless session bean and a singleton bean that have been converted to JAX-RS root resource classes.

@Stateless
@Path("stateless-bean")
public class StatelessResource {...}
@Singleton
@Path("singleton-bean")
public class SingletonResource {...}

Session beans can also be used for subresources.

JAX-RS and CDI have slightly different component models. By default, JAX-RS root resource classes are managed in the request scope, and no annotations are required for specifying the scope. CDI managed beans annotated with @RequestScoped or @ApplicationScoped can be converted to JAX-RS resource classes.

The following code snippet shows a JAX-RS resource class.

@Path("/employee/{id}")
public class Employee {
  public Employee(@PathParam("id") String id) {...}
}
@Path("{lastname}")
public final class EmpDetails {...}

The following code snippet shows this JAX-RS resource class converted to a CDI bean. The beans must be proxyable, so the Employee class requires a non-private constructor with no parameters, and the EmpDetails class must not be final.

@Path("/employee/{id}")
@RequestScoped
public class Employee {
  public Employee() {...}
  @Inject
  public Employee(@PathParam("id") String id) {...}
}
@Path("{lastname}")
@RequestScoped
public class EmpDetails {...}

### Conditional HTTP Requests

JAX-RS provides support for conditional GET and PUT HTTP requests. Conditional GET requests help save bandwidth by improving the efficiency of client processing.

A GET request can return a Not Modified (304) response if the representation has not changed since the previous request. For example, a web site can return 304 responses for all its static images that have not changed since the previous request.

A PUT request can return a Precondition Failed (412) response if the representation has been modified since the last request. The conditional PUT can help avoid the lost update problem.

Conditional HTTP requests can be used with the Last-Modified and ETag headers. The Last-Modified header can represent dates with granularity of one second.
Runtime Content Negotiation

In this code snippet, the constructor of the Employee class computes the entity tag from the request URI and calls the request.evaluatePreconditions method with that tag. If a client request returns an If-none-match header with a value that has the same entity tag that was computed, evaluate.Preconditions returns a pre-filled-out response with a 304 status code and an entity tag set that may be built and returned.

Runtime Content Negotiation

The @Produces and @Consumes annotations handle static content negotiation in JAX-RS. These annotations specify the content preferences of the server. HTTP headers such as Accept, Content-Type, and Accept-Language define the content negotiation preferences of the client.

For more details on the HTTP headers for content negotiation, see HTTP/1.1 - Content Negotiation (http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html).

The following code snippet shows the server content preferences:

```java
@Path("/employee/{joiningdate}")
public class Employee {
    Date joiningdate;

    @GET
    @Produces("application/xml")
    public Employee(@PathParam("joiningdate") Date joiningdate,
                    @Context Request req,
                    @Context UriInfo ui) {
        this.joiningdate = joiningdate;
        this.tag = computeEntityTag(ui.getRequestUri());
        if (req.getMethod().equals("GET")) {
            Response.ResponseBuilder rb = req.evaluatePreconditions(tag);
            if (rb != null) {
                throw new WebApplicationException(rb.build());
            }
        }
    }
}
```

The getEmployeeAddressText method is called for an HTTP request that looks as follows:
GET /employee
Accept: text/plain

This will produce the following response:

500 Oracle Parkway, Redwood Shores, CA

The getEmployeeAddressXml method is called for an HTTP request that looks as follows:

GET /employee
Accept: text/xml

This will produce the following response:

<address street="500 Oracle Parkway, Redwood Shores, CA" country="USA"/>

With static content negotiation, you can also define multiple content and media types for the client and server.

@Produces("text/plain", "text/xml")

In addition to supporting static content negotiation, JAX-RS also supports runtime content negotiation using the javax.ws.rs.core.Variant class and Request objects. The Variant class specifies the resource representation of content negotiation. Each instance of the Variant class may contain a media type, a language, and an encoding. The Variant object defines the resource representation that is supported by the server. The Variant.VariantListBuilder class is used to build a list of representation variants.

The following code snippet shows how to create a list of resource representation variants:

```java
List<Variant> vs =
    Variant.mediatypes("application/xml", "application/json")
    .languages("en", "fr").build();
```

This code snippet calls the build method of the VariantListBuilder class. The VariantListBuilder class is invoked when you call the mediatypes, languages, or encodings methods. The build method builds a series of resource representations. The Variant list created by the build method has all possible combinations of items specified in the mediatypes, languages, and encodings methods.
In this example, the size of the vs object as defined in this code snippet is 4, and the contents are as follows:

```java
["application/xml","en"],
["application/json","en"],
["application/xml","fr"],["application/json","fr"]
```

The `javax.ws.rs.core.Request.selectVariant` method accepts a list of `Variant` objects and chooses the `Variant` object that matches the HTTP request. This method compares its list of `Variant` objects with the `Accept`, `Accept-Encoding`, `Accept-Language`, and `Accept-Charset` headers of the HTTP request.

The following code snippet shows how to use the `selectVariant` method to select the most acceptable `Variant` from the values in the client request.

```java
@GET
public Response get(@Context Request r) {
    List<Variant> vs = ...;
    Variant v = r.selectVariant(vs);
    if (v == null) {
        return Response.notAcceptable(vs).build();
    } else {
        Object rep = selectRepresentation(v);
        return Response.ok(rep, v);
    }
}
```

The `selectVariant` method returns the `Variant` object that matches the request, or null if no matches are found. In this code snippet, if the method returns null, a `Response` object for a non-acceptable response is built. Otherwise, a `Response` object with an OK status and containing a representation in the form of an `Object` entity and a `Variant` is returned.

### Using JAX-RS With JAXB

Java Architecture for XML Binding (JAXB) is an XML-to-Java binding technology that simplifies the development of web services by enabling transformations between schema and Java objects and between XML instance documents and Java object instances. An XML schema defines the data elements and structure of an XML document. You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. JAXB technology provides the tools that enable you to convert your XML documents to and from Java objects.

By using JAXB, you can manipulate data objects in the following ways:

- You can start with an XML schema definition (XSD) and use `xjc`, the JAXB schema compiler tool, to create a set of JAXB-annotated Java classes that map to the elements and types defined in the XSD schema.
- You can start with a set of Java classes and use `schemagen`, the JAXB schema generator tool, to generate an XML schema.
Once a mapping between the XML schema and the Java classes exists, you can use the JAXB binding runtime to marshal and unmarshal your XML documents to and from Java objects and use the resulting Java classes to assemble a web services application.

XML is a common media format that RESTful services consume and produce. To deserialize and serialize XML, you can represent requests and responses by JAXB annotated objects. Your JAX-RS application can use the JAXB objects to manipulate XML data. JAXB objects can be used as request entity parameters and response entities. The JAX-RS runtime environment includes standard MessageBodyReader and MessageBodyWriter provider interfaces for reading and writing JAXB objects as entities.

With JAX-RS, you enable access to your services by publishing resources. Resources are just simple Java classes with some additional JAX-RS annotations. These annotations express the following:

- The path of the resource (the URL you use to access it)
- The HTTP method you use to call a certain method (for example, the GET or POST method)
- The MIME type with which a method accepts or responds

As you define the resources for your application, consider the type of data you want to expose. You may already have a relational database that contains information you want to expose to users, or you may have static content that does not reside in a database but does need to be distributed as resources. Using JAX-RS, you can distribute content from multiple sources.

RESTful web services can use various types of input/output formats for request and response. For example, the "customer example" described in "The customer Example Application" on page 418, uses XML.

JAX-RS also maps Java types to and from resource representations using entity providers. A MessageBodyReader entity provider reads a request entity and deserializes the request entity into a Java type. A MessageBodyWriter entity provider serializes from a Java type into a response entity. For example, if a String value is used as the request entity parameter, the MessageBodyReader entity provider deserializes the request body into a new String. If a JAXB type is used as the return type on a resource method, the MessageBodyWriter serializes the JAXB object into a response body.
By default, the JAX-RS runtime environment attempts to create and use a default JAXBContext class for JAXB classes. However, if the default JAXBContext class is not suitable, then you can supply a JAXBContext class for the application using a JAX-RS ContextResolver provider interface.

The following sections explain how to use JAXB with JAX-RS resource methods.

**Using Java Objects to Model Your Data**

If you do not have an XML schema definition for the data you want to expose, you can model your data as Java classes, add JAXB annotations to these classes, and use JAXB to generate an XML schema for your data. For example, if the data you want to expose is a collection of products and each product has an ID, a name, a description, and a price, you can model it as a Java class as follows:

```java
@XmlElement(name="product")
@XmlElementWrapper(useWrapper=true)
public class Product {
    @XmlElementWrapper(name="products")
    private List<Product> products;

    public Product() {
    }

    // Getter and setter methods
}
```

Run the JAXB schema generator on the command line to generate the corresponding XML schema definition:

```
schemagen Product.java
```

This command produces the XML schema as an .xsd file:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="product" type="product"/>

    <xs:complexType name="product">
        <xs:sequence>
            <xs:element name="id" type="xs:int"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>

</xs:schema>
```
Once you have this mapping, you can create Product objects in your application, return them, and use them as parameters in JAX-RS resource methods. The JAX-RS runtime uses JAXB to convert the XML data from the request into a Product object and to convert a Product object into XML data for the response. The following resource class provides a simple example:

```java
@Path("/product")
public class ProductService {

    @GET
    @Path("/get")
    @Produces("application/xml")
    public Product getProduct() {
        Product prod = new Product();
        prod.setId(1);
        prod.setName("Mattress");
        prod.setDescription("Queen size mattress");
        prod.setPrice(500);
        return prod;
    }

    @POST
    @Path("/create")
    @Consumes("application/xml")
    public Response createProduct(Product prod) {
        // Process or store the product and return a response
        // ...
    }
}
```

Some IDEs, such as NetBeans IDE, will run the schema generator tool automatically during the build process if you add Java classes that have JAXB annotations to your project. For a detailed example, see “The customer Example Application” on page 418. The customer example contains a more complex relationship between the Java classes that model the data, which results in a more hierarchical XML representation.

### Starting from an Existing XML Schema Definition

If you already have an XML schema definition in an .xsd file for the data you want to expose, use the JAXB schema compiler tool. Consider this simple example of an .xsd file:

```xml
<?xml version="1.0"?>
<xs:schema targetNamespace="http://xml.product"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    xmlns:myco="http://xml.product">

    <xs:element name="product" type="myco:Product"/>
</xs:schema>
```
<xs:complexType name="Product">
  <xs:sequence>
    <xs:element name="id" type="xs:int"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="price" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>

Run the schema compiler tool on the command line as follows:

xjc Product.xsd

This command generates the source code for Java classes that correspond to the types defined in the .xsd file. The schema compiler tool generates a Java class for each complexType defined in the .xsd file. The fields of each generated Java class are the same as the elements inside the corresponding complexType, and the class contains getter and setter methods for these fields.

In this case the schema compiler tool generates the classes product.xml.Product and product.xml.ObjectFactory. The Product class contains JAXB annotations, and its fields correspond to those in the .xsd definition:

```java
@XmlElementDecl(namespace = "http://xml.product", name = "product")
public JAXBElement<Product> createProduct(Product value) {
  return new JAXBElement<>(_Product_QNAME, Product.class, null, value);
}
```

The following code shows how to use the generated classes to return a JAXB element as XML in a JAX-RS resource method:

```java
@XmlElementDecl(namespace = "http://xml.product", name = "product")
public JAXBElement<Product> createProduct(Product value) {
  return new JAXBElement<>(_Product_QNAME, Product.class, null, value);
}
```
@Path("/product")
public class ProductService {
    @GET
    @Path("/get")
    @Produces("application/xml")
    public JAXBElement<Product> getProduct() {
        Product prod = new Product();
        prod.setId(1);
        prod.setName("Mattress");
        prod.setDescription("Queen size mattress");
        prod.setPrice(500);
        return new ObjectFactory().createProduct(prod);
    }
}

For @POST and @PUT resource methods, you can use a Product object directly as a parameter. JAX-RS maps the XML data from the request into a Product object.

@Path("/product")
public class ProductService {
    // ...
    @POST
    @Path("/create")
    @Consumes("application/xml")
    public Response createProduct(Product prod) {
        // Process or store the product and return a response
        // ...
    }
}

Some IDEs, such as NetBeans IDE, will run the schema compiler tool automatically during the build process if you add an .xsd file to your project sources. For a detailed example, see "Modifying the Example to Generate Entity Classes from an Existing Schema" on page 426. The modified customer example contains a more hierarchical XML schema definition, which results in a more complex relationship between the Java classes that model the data.

Using JSON with JAX-RS and JAXB

JAX-RS can automatically read and write XML using JAXB, but it can also work with JSON data. JSON is a simple text-based format for data exchange derived from JavaScript. For the examples above, the XML representation of a product is:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<product>
    <id>1</id>
    <name>Mattress</name>
    <description>Queen size mattress</description>
    <price>500</price>
</product>
```

The equivalent JSON representation is:
The customer Example Application

You can add the format application/json to the @Produces annotation in resource methods to produce responses with JSON data:

@GET
@Path("/get")
@Produces(["application/xml","application/json"])
public Product getProduct() { ... }

In this example the default response is XML, but the response is a JSON object if the client makes a GET request that includes this header:

Accept: application/json

The resource methods can also accept JSON data for JAXB annotated classes:

@POST
@Path("/create")
@Consumes(["application/xml","application/json"])
public Response createProduct(Product prod) { ... }

The client should include the following header when submitting JSON data with a POST request:

Content-Type: application/json

The customer Example Application

This section describes how to build and run the customer sample application. This example application is a RESTful web service that uses JAXB to perform the Create, Read, Update, Delete (CRUD) operations for a specific entity.

The customer sample application is in the tut-install/examples/jaxrs/customer/ directory. See Chapter 2, “Using the Tutorial Examples,” for basic information on building and running sample applications.
Overview of the customer Example Application

The source files of this application are at `tut-install/examples/jaxrs/customer/src/java/`. The application has three parts:

- The Customer and Address entity classes. These classes model the data of the application and contain JAXB annotations. See “The Customer and Address Entity Classes” on page 419 for details.
- The CustomerService resource class. This class contains JAX-RS resource methods that perform operations on Customer instances represented as XML or JSON data using JAXB. See “The CustomerService Class” on page 422 for details.
- The CustomerClientXML and CustomerClientJSON client classes. These classes test the resource methods of the web service using XML and JSON representations of Customer instances. See “The CustomerClientXML and CustomerClientJSON Classes” on page 424 for details.

The customer sample application shows you how to model your data entities as Java classes with JAXB annotations. The JAXB schema generator produces an equivalent XML schema definition file (.xsd) for your entity classes. The resulting schema is used to automatically marshal and unmarshal entity instances to and from XML or JSON in the JAX-RS resource methods.

In some cases you may already have an XML schema definition for your entities. See “Modifying the Example to Generate Entity Classes from an Existing Schema” on page 426 for instructions on how to modify the customer example to model your data starting from an .xsd file and using JAXB to generate the equivalent Java classes.

The Customer and Address Entity Classes

The following class represents a customer’s address:

```java
@XmlElement(name="address")
@XmlAccessorType(XmlAccessType.FIELD)
public class Address {
    @XmlElement(required=true)
    protected int number;

    @XmlElement(required=true)
    protected String street;

    @XmlElement(required=true)
    protected String city;

    @XmlElement(required=true)
    protected String state;
```
@XmlElement(required=true)
protected String zip;

@XmlElement(required=true)
protected String country;

public Address() {
    
    // Getter and setter methods
    // ...
}

The @XmlRootElement (name="address") annotation maps this class to the address XML element. The @XmlAccessorType (XmlAccessType.FIELD) annotation specifies that all the fields of this class are bound to XML by default. The @XmlElement (required=true) annotation specifies that an element must be present in the XML representation.

The following class represents a customer:

@XmlElement(name="customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    
    @XmlAttribute(required=true)
    protected int id;

    @XmlElement(required=true)
    protected String firstname;

    @XmlElement(required=true)
    protected String lastname;

    @XmlElement(required=true)
    protected Address address;

    @XmlElement(required=true)
    protected String email;

    @XmlElement(required=true)
    protected String phone;

    public Customer() {
        
        // Getter and setter methods
        // ...
    }

    The Customer class contains the same JAXB annotations as the previous class, except for the @XmlAttribute (required=true) annotation, which maps a property to an attribute of the XML element representing the class.

    The Customer class contains a property whose type is another entity, the Address class. This mechanism allows you to define in Java code the hierarchical relationships between entities without having to write an .xsd file yourself.
JAXB generates the following XML schema definition for the two classes above:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="address" type="address"/>
  <xs:element name="customer" type="customer"/>
  <xs:complexType name="address">
    <xs:sequence>
      <xs:element name="number" type="xs:int"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="state" type="xs:string"/>
      <xs:element name="zip" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element ref="address"/>
      <xs:element name="email" type="xs:string"/>
      <xs:element name="phone" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
  </xs:complexType>
</xs:schema>
```

The file `sample-input.xml` in the top-level directory of the project contains an example of an XML representation of a customer:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<customer id="1">
  <firstname>Duke</firstname>
  <lastname>OfJava</lastname>
  <address>
    <number>1</number>
    <street>Duke's Way</street>
    <city>JavaTown</city>
    <state>JA</state>
    <zip>12345</zip>
    <country>USA</country>
  </address>
  <email>duke@example.com</email>
  <phone>123-456-7890</phone>
</customer>
```

The file `sample-input.json` contains an example of a JSON representation of a customer:

```json
{
  "id": 1,
  "firstname": "Duke",
  "lastname": "OfJava",
```
The CustomerService Class

The CustomerService class has a createCustomer method that creates a customer resource based on the Customer class and returns a URI for the new resource. The persist method emulates the behavior of the JPA entity manager. This example uses a java.util.Properties file to store data. If you are using the default configuration of GlassFish Server, the properties file is at domain-dir/CustomerDATA.txt.

```java
@Path("/Customer")
public class CustomerService {
    public static final String DATA_STORE = "CustomerDATA.txt";
    public static final Logger logger = Logger.getLogger(CustomerService.class.getCanonicalName());
    ...

    @POST
    @Consumes( {"application/xml", "application/json"})
    public Response createCustomer(Customer customer) {
        try {
            long customerId = persist(customer);
            return Response.created(URI.create("/" + customerId)).build();
        } catch (Exception e) {
            throw new WebApplicationException(e,
                Response.Status.INTERNAL_SERVER_ERROR);
        }
    }

    private long persist(Customer customer) throws IOException {
        File dataFile = new File(DATA_STORE);
        if (!dataFile.exists()) {
            dataFile.createNewFile();
        }

        long customerId = customer.getId();
        Address address = customer.getAddress();
        Properties properties = new Properties();
        properties.load(new FileInputStream(dataFile));
```
properties.setProperty(String.valueOf(customerId),
    customer.getFirstName() + "",
    customer.getLastName() + "",
    address.getNumber() + "",
    address.getStreet() + "",
    address.getCity() + "",
    address.getState() + "",
    address.getZip() + "",
    address.getCountry() + "",
    customer.getEmail() + "",
    customer.getPhone());

properties.store(new FileOutputStream(DATA_STORE),null);
return customerId;

The response returned to the client has a URI to the newly created resource. The return type is an entity body mapped from the property of the response with the status code specified by the status property of the response. The WebApplicationException is a RuntimeException that is used to wrap the appropriate HTTP error status code, such as 404, 406, 415, or 500.

The @Consumes({"application/xml","application/json"}) and @Produces({"application/xml","application/json"}) annotations set the request and response media types to use the appropriate MIME client. These annotations can be applied to a resource method, a resource class, or even an entity provider. If you do not use these annotations, JAX-RS allows the use of any media type("*/*").

The following code snippet shows the implementation of the getCustomer and findById methods. The getCustomer method uses the @Produces annotation and returns a Customer object, which is converted to an XML or JSON representation depending on the Accept: header specified by the client.

@GET
@Path("{id}")
@Produces({"application/xml", "application/json"})
public Customer getCustomer(@PathParam("id") String customerId) {
    Customer customer = null;
    try {
        customer = findById(customerId);
    } catch (Exception ex) {
        logger.log(Level.SEVERE,
            "Error calling searchCustomer() for customerId {0}. {1}",
            new Object[]{customerId, ex.getMessage()});
    }
    return customer;
}

private Customer findById(String customerId) throws IOException {
    properties = new Properties();
    properties.load(new FileInputStream(DATA_STORE));
The customerId Example Application

```java
String rawData = properties.getProperty(customerId);
if (rawData != null) {
    final String[] field = rawData.split(",");
    Address address = new Address();
    Customer customer = new Customer();
customer.setId(Integer.parseInt(customerId));
customer.setAddress(address);
customer.setFirstname(field[0]);
customer.setLastname(field[1]);
address.setNumber(Integer.parseInt(field[2]));
address.setStreet(field[3]);
address.setCity(field[4]);
address.setState(field[5]);
address.setZip(field[6]);
address.setCountry(field[7]);
customer.setEmail(field[8]);
customer.setPhone(field[9]);
return customer;
} return null;
```

The `CustomerClientXML` and `CustomerClientJSON` Classes

Jersey is the reference implementation of JAX-RS (JSR 311). You can use the Jersey client API to write a test client for the customer example application. You can find the Jersey APIs at [http://jersey.java.net/nonav/apidocs/latest/jersey/](http://jersey.java.net/nonav/apidocs/latest/jersey/).

The `CustomerClientXML` class calls Jersey APIs to test the `CustomerService` web service:

```java
package customer.rest.client;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import customer.data.Address;
import customer.data.Customer;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ws.rs.core.MediaType;
public class CustomerClientXML {
    public static final Logger logger =
        Logger.getLogger(CustomerClientXML.class.getCanonicalName());
    public static void main(String[] args) {
        Client client = Client.create();
        // Define the URL for testing the example application
```

The Java EE 6 Tutorial • January 2013
WebResource webResource =
    client.resource("http://localhost:8080/customer/rest/Customer");

    // Test the POST method
    Customer customer = new Customer();
    Address address = new Address();
    customer.setAddress(address);
    customer.setId(1);
    customer.setFirstname("Duke");
    customer.setLastName("OfJava");
    address.setNumber(1);
    address.setStreet("Duke's Drive");
    address.setCity("JavaTown");
    address.setZip("1234");
    address.setState("JA");
    address.setCountry("USA");
    customer.setEmail("duke@java.net");
    customer.setPhone("12341234");
    ClientResponse response =
        webResource.type("application/xml").post(ClientResponse.class,
            customer);
    logger.info("POST status: {0} + response.getStatus());
    if (response.getStatus() == 201) {
        logger.info("POST succeeded");
    } else {
        logger.info("POST failed");
    }

    // Test the GET method using content negotiation
    response = webResource.path("1").accept(MediaType.APPLICATION_XML)
        .get(ClientResponse.class);
    Customer entity = response.getEntity(Customer.class);
    logger.log(Level.INFO,
        "GET status: {0} + response.getStatus());
    if (response.getStatus() == 200) {
        logger.log(Level.INFO, "GET succeeded, city is {0}",
            entity.getAddress().getCity());
    } else {
        logger.info("GET failed");
    }

    // Test the DELETE method
    response = webResource.path("1").delete(ClientResponse.class);
    logger.log(Level.INFO, "DELETE status: {0} + response.getStatus());
    if (response.getStatus() == 204) {
        logger.info("DELETE succeeded (no content)"/
    } else {
        logger.info("DELETE failed");
    }

    response = webResource.path("1").accept(MediaType.APPLICATION_XML)
        .get(ClientResponse.class);
    logger.log(Level.INFO, "GET status: {0} + response.getStatus());
    if (response.getStatus() == 204) {
        logger.info("After DELETE, the GET request returned no content.");
    }
This Jersey client tests the POST, GET, and DELETE methods using XML representations.

All of these HTTP status codes indicate success: 201 for POST, 200 for GET, and 204 for DELETE. For details about the meanings of HTTP status codes, see http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

The CustomerClientJSON class is similar to CustomerClientXML but it uses JSON representations to test the web service. In the CustomerClientJSON class "application/xml" is replaced by "application/json", and MediaType.APPLICATION_XML is replaced by MediaType.APPLICATION_JSON.

## Modifying the Example to Generate Entity Classes from an Existing Schema

This section describes how you can modify the customer example if you provide an XML schema definition file for your entities instead of providing Java classes. In this case JAXB generates the equivalent Java entity classes from the schema definition.

For the customer example you provide the following .xsd file:

```xml
<?xml version="1.0"?>
<xs:schema targetNamespace="http://xml.customer"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
  xmlns:ora="http://xml.customer">
    <xs:element name="customer" type="ora:Customer"/>
    <xs:complexType name="Address">
      <xs:sequence>
        <xs:element name="number" type="xs:int"/>
        <xs:element name="street" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="state" type="xs:string"/>
        <xs:element name="zip" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Customer">
      <xs:sequence>
        <xs:element name="firstname" type="xs:string"/>
        <xs:element name="lastname" type="xs:string"/>
        <xs:element name="address" type="ora:Address"/>
        <xs:element name="email" type="xs:string"/>
        <xs:element name="phone" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```
You can modify the customer example as follows:

▼ To Modify the customer Example to Generate Java Entity Classes from an Existing XML Schema Definition

1 Create a JAXB binding to generate the entity Java classes from the schema definition. For example, in NetBeans IDE, follow these steps:

a. Right click on the customer project and select New > Other...

b. Under the XML folder, select JAXB Binding and click Next.

c. In the Binding Name field, type CustomerBinding.

d. Click Browse and choose the .xsd file from your file system.

e. In the Package Name field, type customer.xml.

f. Click Finish.

This procedure creates the Customer class, the Address class, and some JAXB auxiliary classes in the package customer.xml.

2 Modify the CustomerService class as follows:

a. Replace the customer.data.* imports with customer.xml.* imports and import the JAXBElemnet and ObjectFactory classes:

import customer.xml.Customer;
import customer.xml.Address;
import customer.xml.ObjectFactory;
import javax.xml.bind.JAXBElement;

b. Replace the return type of the getCustomer method:

public JAXBElement<Customer> getCustomer(
    @PathParam("id") String customerId) {
    ...
    return new ObjectFactory().createCustomer(customer);
3 Modify the CustomerClientXML and CustomerClientJSON classes as follows:

   a. Replace the customer.data.* imports with customer.xml.* imports and import the JAXElement and ObjectFactory classes:
      
      import customer.xml.Address;
      import customer.xml.Customer;
      import customer.xml.ObjectFactory;
      import javax.xml.bind.JAXBElement;
      
      b. Create an ObjectFactory instance and a JAXBElement<Customer> instance at the beginning of the main method:

      public static void main(String[] args) {
         Client client = Client.create();
         ObjectFactory factory = new ObjectFactory();
         WebResource webResource = ...
         ...
         customer.setPhone("12341234");
         JAXBElement<Customer> customerJAXB = factory.createCustomer(customer);
         ClientResponse response = webResource.type("application/xml")
         .post(ClientResponse.class, customerJAXB);
         ...
      }
      
      c. Modify the GET request after testing the DELETE method:

      response = webResource.path("1").accept(MediaType.APPLICATION_XML)
      .get(ClientResponse.class);
      entity = response.getEntity(Customer.class);
      logger.log(Level.INFO, "GET status: {0}", response.getStatus());
      try {
         logger.info(entity.getAddress().getCity());
      } catch (NullPointerException ne) {
         // null after deleting the only customer
         logger.log(Level.INFO, "After DELETE, city is: {0}", ne.getCause());
      }
      
      The instructions for building, deploying, and running the example are the same for the original customer example and for the modified version using this procedure.

Running the customer Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the customer application.

▼ To Build, Package, and Deploy the customer Example Using NetBeans IDE

This procedure builds the application into the tut-install/examples/jax-rs/customer/build/web/ directory. The contents of this directory are deployed to the GlassFish Server.
From the File menu, choose Open Project.

In the Open Project dialog, navigate to:

tut-install/examples/jaxrs/

Select the customer folder.

Select the Open as Main Project check box.

Click Open Project.

It may appear that there are errors in the source files, because the files refer to JAXB classes that will be generated when you build the application. You can ignore these errors.

In the Projects tab, right-click the customer project and select Deploy.

To Build, Package, and Deploy the customer Example Using Ant

In a terminal window, go to:

tut-install/examples/jaxrs/customer/

Type the following command:

ant

This command calls the default target, which builds and packages the application into a WAR file, customer.war, located in the dist directory.

Type the following command:

ant deploy

Typing this command deploys customer.war to the GlassFish Server.

To Run the customer Example Using the Jersey Client

In NetBeans IDE, expand the Source Packages node.

Expand the customer.rest.client node.

Right-click the CustomerClientXML.java file and select Run File.

The output of the client looks like this:

```
run:
INFO: POST status: 201
INFO: POST succeeded
```
To Run the customer Example Using the Web Services Tester

1 In NetBeans IDE, right-click the customer node and select Test RESTful Web Services.

Note – The Web Services Tester works only with the modified version of the customer example.

2 In the Configure REST Test Client dialog, select Web Test Client in Project and click Browse.

3 In the Select Project dialog, choose the customer project and click OK.

4 In the Configure REST Test Client dialog, click OK.

5 When the test client appears in the browser, select the Customer resource node in the left pane.

6 Paste the following XML code into the Content text area, replacing “Insert content here”:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<customer id="1">
    <firstname>Duke</firstname>
    <lastname>OfJava</lastname>
    <address>
        <number>1</number>
        <street>Duke's Way</street>
        <city>JavaTown</city>
        <state>JA</state>
```
You can find the code in the file `customer/sample-input.xml`.

7 **Click Test.**
The following message appears in the window below:
Status: 201 (Created)

8 **Expand the Customer node and click `{id}`.**

9 **Type 1 in the id field and click Test to test the GET method.**
The following status message appears:
Status: 200 (OK)
The XML output for the resource appears in the Response window:

```
<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="http://xml.customer" id="1">
  <firstname>Duke</firstname>
  <lastname>OfJava</lastname>
  <address>
    <number>1</number>
    <street>Duke's Way</street>
    <city>JavaTown</city>
    <state>JA</state>
    <zip>12345</zip>
    <country>USA</country>
  </address>
  <email>duke@example.com</email>
  <phone>123-456-7890</phone>
</customer>
```

A GET for a nonexistent ID also returns a 200 (OK) status, but the output in the Response window shows no content:

```
<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="http://xml.customer"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:nil="true"/>
```

You can test other methods as follows:

- **Select PUT**, type the input for an existing customer, modify any content except the `id` value, and click Test to update the customer fields. A successful update returns the following status message:
  Status: 303 (See Other)

- **Select DELETE**, type the ID for an existing customer, and click Test to remove the customer. A successful delete returns the following status message:
Using Curl to Run the customer Example Application

Curl is a command-line tool you can use to run the customer application on UNIX platforms. You can download Curl from http://curl.haxx.se or add it to a Cygwin installation.

Run the following commands in the directory tut-install/examples/jaxrs/customer/ after deploying the application.

To add a new customer and test the POST method using XML data, use the following command:

```
curl -i --data @sample-input.xml \
-<header Content-type:application/xml \
 http://localhost:8080/customer/rest/Customer
```

To add a new customer using JSON data instead, use the following command:

```
curl -i --data @sample-input.json \
-<header Content-type:application/json \
 http://localhost:8080/customer/rest/Customer
```

A successful POST returns HTTP Status: 201 (Created).

To retrieve the details of the customer whose ID is 1, use the following command:

```
curl -i -X GET http://localhost:8080/customer/rest/Customer/1
```

To retrieve the details of the same customer represented as JSON data, use the following command:

```
curl -i --header Accept:application/json 
 -X GET http://localhost:8080/customer/rest/Customer/1
```

A successful GET returns HTTP Status: 200 (OK).

To delete a customer record, use the following command:

```
curl -i -X DELETE http://localhost:8080/customer/rest/Customer/1
```

A successful DELETE returns HTTP Status: 204.

The customer example and the modified version respond differently to a GET request for a customer ID that does not exist. The original customer example returns HTTP Status: 204 (No content), whereas the modified version returns HTTP Status: 200 (OK) with a response that contains the XML header but no customer data.
Enterprise Beans

Part IV explores Enterprise JavaBeans components. This part contains the following chapters:

- Chapter 22, "Enterprise Beans"
- Chapter 23, "Getting Started with Enterprise Beans"
- Chapter 24, "Running the Enterprise Bean Examples"
- Chapter 25, "A Message-Driven Bean Example"
- Chapter 26, "Using the Embedded Enterprise Bean Container"
- Chapter 27, "Using Asynchronous Method Invocation in Session Beans"
Enterprise beans are Java EE components that implement Enterprise JavaBeans (EJB) technology. Enterprise beans run in the EJB container, a runtime environment within the GlassFish Server (see “Container Types” on page 48). Although transparent to the application developer, the EJB container provides system-level services, such as transactions and security, to its enterprise beans. These services enable you to quickly build and deploy enterprise beans, which form the core of transactional Java EE applications.

The following topics are addressed here:
- “What Is an Enterprise Bean?” on page 435
- “What Is a Session Bean?” on page 437
- “What Is a Message-Driven Bean?” on page 439
- “Accessing Enterprise Beans” on page 441
- “The Contents of an Enterprise Bean” on page 447
- “Naming Conventions for Enterprise Beans” on page 449
- “The Lifecycles of Enterprise Beans” on page 450
- “Further Information about Enterprise Beans” on page 453

What Is an Enterprise Bean?

Written in the Java programming language, an enterprise bean is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called checkInventoryLevel and orderProduct. By invoking these methods, clients can access the inventory services provided by the application.
**Benefits of Enterprise Beans**

For several reasons, enterprise beans simplify the development of large, distributed applications. First, because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, rather than the bean developer, is responsible for system-level services, such as transaction management and security authorization.

Second, because the beans rather than the clients contain the application’s business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. Provided that they use the standard APIs, these applications can run on any compliant Java EE server.

**When to Use Enterprise Beans**

You should consider using enterprise beans if your application has any of the following requirements.

- The application must be scalable. To accommodate a growing number of users, you may need to distribute an application’s components across multiple machines. Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.

- Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.

- The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

**Types of Enterprise Beans**

Table 22–1 summarizes the two types of enterprise beans. The following sections discuss each type in more detail.
What Is a Session Bean?

A session bean encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean’s methods. The session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

A session bean is not persistent. (That is, its data is not saved to a database.)

For code samples, see Chapter 24, “Running the Enterprise Bean Examples.”

Types of Session Beans

Session beans are of three types: stateful, stateless, and singleton.

**Stateful Session Beans**

The state of an object consists of the values of its instance variables. In a stateful session bean, the instance variables represent the state of a unique client/bean session. Because the client interacts (“talks”) with its bean, this state is often called the conversational state.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. When the client terminates, its session bean appears to terminate and is no longer associated with the client.

The state is retained for the duration of the client/bean session. If the client removes the bean, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends, there is no need to retain the state.

**Stateless Session Beans**

A stateless session bean does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean’s instance variables may contain a state specific to that client but only for the duration of the invocation. When the method is finished, the
client-specific state should not be retained. Clients may, however, change the state of instance variables in pooled stateless beans, and this state is held over to the next invocation of the pooled stateless bean. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. That is, the state of a stateless session bean should apply across all clients.

Because they can support multiple clients, stateless session beans can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

A stateless session bean can implement a web service, but a stateful session bean cannot.

**Singleton Session Beans**

A singleton session bean is instantiated once per application and exists for the lifecycle of the application. Singleton session beans are designed for circumstances in which a single enterprise bean instance is shared across and concurrently accessed by clients.

Singleton session beans offer similar functionality to stateless session beans but differ from them in that there is only one singleton session bean per application, as opposed to a pool of stateless session beans, any of which may respond to a client request. Like stateless session beans, singleton session beans can implement web service endpoints.

Singleton session beans maintain their state between client invocations but are not required to maintain their state across server crashes or shutdowns.

Applications that use a singleton session bean may specify that the singleton should be instantiated upon application startup, which allows the singleton to perform initialization tasks for the application. The singleton may perform cleanup tasks on application shutdown as well, because the singleton will operate throughout the lifecycle of the application.

**When to Use Session Beans**

Stateful session beans are appropriate if any of the following conditions are true.

- The bean’s state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans.
To improve performance, you might choose a stateless session bean if it has any of these traits.

- The bean’s state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an email that confirms an online order.
- The bean implements a web service.

Singleton session beans are appropriate in the following circumstances.

- State needs to be shared across the application.
- A single enterprise bean needs to be accessed by multiple threads concurrently.
- The application needs an enterprise bean to perform tasks upon application startup and shutdown.
- The bean implements a web service.

What Is a Message-Driven Bean?

A message-driven bean is an enterprise bean that allows Java EE applications to process messages asynchronously. This type of bean normally acts as a JMS message listener, which is similar to an event listener but receives JMS messages instead of events. The messages can be sent by any Java EE component (an application client, another enterprise bean, or a web component) or by a JMS application or system that does not use Java EE technology. Message-driven beans can process JMS messages or other kinds of messages.

What Makes Message-Driven Beans Different from Session Beans?

The most visible difference between message-driven beans and session beans is that clients do not access message-driven beans through interfaces. Interfaces are described in the section “Accessing Enterprise Beans” on page 441. Unlike a session bean, a message-driven bean has only a bean class.
In several respects, a message-driven bean resembles a stateless session bean.

- A message-driven bean’s instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages, such as a JMS API connection, an open database connection, or an object reference to an enterprise bean object.

Client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through, for example, JMS by sending messages to the message destination for which the message-driven bean class is the MessageListener. You assign a message-driven bean’s destination during deployment by using GlassFish Server resources.

Message-driven beans have the following characteristics.

- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.
- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

When a message arrives, the container calls the message-driven bean’s onMessage method to process the message. The onMessage method normally casts the message to one of the five JMS message types and handles it in accordance with the application’s business logic. The onMessage method can call helper methods or can invoke a session bean to process the information in the message or to store it in a database.

A message can be delivered to a message-driven bean within a transaction context, so all operations within the onMessage method are part of a single transaction. If message processing is rolled back, the message will be redelivered. For more information, see Chapter 25, “A Message-Driven Bean Example,” and Chapter 44, “Transactions.”

When to Use Message-Driven Beans

Session beans allow you to send JMS messages and to receive them synchronously but not asynchronously. To avoid tying up server resources, do not to use blocking synchronous
receives in a server-side component; in general, JMS messages should not be sent or received synchronously. To receive messages asynchronously, use a message-driven bean.

### Accessing Enterprise Beans

**Note** – The material in this section applies only to session beans and not to message-driven beans. Because they have a different programming model, message-driven beans do not have interfaces or no-interface views that define client access.

Clients access enterprise beans either through a no-interface view or through a business interface. A no-interface view of an enterprise bean exposes the public methods of the enterprise bean implementation class to clients. Clients using the no-interface view of an enterprise bean may invoke any public methods in the enterprise bean implementation class or any superclasses of the implementation class. A business interface is a standard Java programming language interface that contains the business methods of the enterprise bean.

A client can access a session bean only through the methods defined in the bean’s business interface or through the public methods of an enterprise bean that has a no-interface view. The business interface or no-interface view defines the client’s view of an enterprise bean. All other aspects of the enterprise bean (method implementations and deployment settings) are hidden from the client.

Well-designed interfaces and no-interface views simplify the development and maintenance of Java EE applications. Not only do clean interfaces and no-interface views shield the clients from any complexities in the EJB tier, but they also allow the enterprise beans to change internally without affecting the clients. For example, if you change the implementation of a session bean business method, you won’t have to alter the client code. But if you were to change the method definitions in the interfaces, you might have to modify the client code as well. Therefore, it is important that you design the interfaces and no-interface views carefully to isolate your clients from possible changes in the enterprise beans.

Session beans can have more than one business interface. Session beans should, but are not required to, implement their business interface or interfaces.

### Using Enterprise Beans in Clients

The client of an enterprise bean obtains a reference to an instance of an enterprise bean through either dependency injection, using Java programming language annotations, or JNDI lookup, using the Java Naming and Directory Interface syntax to find the enterprise bean instance.
Dependency injection is the simplest way of obtaining an enterprise bean reference. Clients that run within a Java EE server-managed environment, JavaServer Faces web applications, JAX-RS web services, other enterprise beans, or Java EE application clients, support dependency injection using the `javax.ejb.EJB` annotation.

Applications that run outside a Java EE server-managed environment, such as Java SE applications, must perform an explicit lookup. JNDI supports a global syntax for identifying Java EE components to simplify this explicit lookup.

**Portable JNDI Syntax**

Three JNDI namespaces are used for portable JNDI lookups: `java:global`, `java:module`, and `java:app`.

- The `java:global` JNDI namespace is the portable way of finding remote enterprise beans using JNDI lookups. JNDI addresses are of the following form:

  ```
  java:global[/application name]/module/enterprise bean name[/interface name]
  ```

  Application name and module name default to the name of the application and module minus the file extension. Application names are required only if the application is packaged within an EAR. The interface name is required only if the enterprise bean implements more than one business interface.

- The `java:module` namespace is used to look up local enterprise beans within the same module. JNDI addresses using the `java:module` namespace are of the following form:

  ```
  java:module/enterprise bean name[/interface name]
  ```

  The interface name is required only if the enterprise bean implements more than one business interface.

- The `java:app` namespace is used to look up local enterprise beans packaged within the same application. That is, the enterprise bean is packaged within an EAR file containing multiple Java EE modules. JNDI addresses using the `java:app` namespace are of the following form:

  ```
  java:app[/module name]/enterprise bean name[/interface name]
  ```

  The module name is optional. The interface name is required only if the enterprise bean implements more than one business interface.

For example, if an enterprise bean, `MyBean`, is packaged within the web application archive `myApp.war`, the module name is `myApp`. The portable JNDI name is `java:module/MyBean`. An equivalent JNDI name using the `java:global` namespace is `java:global/myApp/MyBean`.

**Deciding on Remote or Local Access**

When you design a Java EE application, one of the first decisions you make is the type of client access allowed by the enterprise beans: remote, local, or web service.
Whether to allow local or remote access depends on the following factors.

- **Tight or loose coupling of related beans**: Tightly coupled beans depend on one another. For example, if a session bean that processes sales orders calls a session bean that emails a confirmation message to the customer, these beans are tightly coupled. Tightly coupled beans are good candidates for local access. Because they fit together as a logical unit, they typically call each other often and would benefit from the increased performance that is possible with local access.

- **Type of client**: If an enterprise bean is accessed by application clients, it should allow remote access. In a production environment, these clients almost always run on machines other than those on which the GlassFish Server is running. If an enterprise bean’s clients are web components or other enterprise beans, the type of access depends on how you want to distribute your components.

- **Component distribution**: Java EE applications are scalable because their server-side components can be distributed across multiple machines. In a distributed application, for example, the server that the web components run on may not be the one on which the enterprise beans they access are deployed. In this distributed scenario, the enterprise beans should allow remote access.

- **Performance**: Owing to such factors as network latency, remote calls may be slower than local calls. On the other hand, if you distribute components among different servers, you may improve the application’s overall performance. Both of these statements are generalizations; performance can vary in different operational environments. Nevertheless, you should keep in mind how your application design might affect performance.

If you aren’t sure which type of access an enterprise bean should have, choose remote access. This decision gives you more flexibility. In the future, you can distribute your components to accommodate the growing demands on your application.

Although it is uncommon, it is possible for an enterprise bean to allow both remote and local access. If this is the case, either the business interface of the bean must be explicitly designated as a business interface by being decorated with the `@Remote` or `@Local` annotations, or the bean class must explicitly designate the business interfaces by using the `@Remote` and `@Local` annotations. The same business interface cannot be both a local and a remote business interface.

**Local Clients**

A local client has these characteristics.

- It must run in the same application as the enterprise bean it accesses.
- It can be a web component or another enterprise bean.
- To the local client, the location of the enterprise bean it accesses is not transparent.
The no-interface view of an enterprise bean is a local view. The public methods of the enterprise bean implementation class are exposed to local clients that access the no-interface view of the enterprise bean. Enterprise beans that use the no-interface view do not implement a business interface.

The local business interface defines the bean’s business and lifecycle methods. If the bean’s business interface is not decorated with @Local or @Remote, and if the bean class does not specify the interface using @Local or @Remote, the business interface is by default a local interface.

To build an enterprise bean that allows only local access, you may, but are not required to, do one of the following:

- Create an enterprise bean implementation class that does not implement a business interface, indicating that the bean exposes a no-interface view to clients. For example:

  ```java
  @Session
  public class MyBean { ... }
  ```

- Annotate the business interface of the enterprise bean as a @Local interface. For example:

  ```java
  @Local
  public interface InterfaceName { ... }
  ```

- Specify the interface by decorating the bean class with @Local and specify the interface name. For example:

  ```java
  @Local(InterfaceName.class)
  public class BeanName implements InterfaceName { ... }
  ```

### Accessing Local Enterprise Beans Using the No-Interface View

Client access to an enterprise bean that exposes a local, no-interface view is accomplished through either dependency injection or JNDI lookup.

- To obtain a reference to the no-interface view of an enterprise bean through dependency injection, use the javax.ejb.EJB annotation and specify the enterprise bean’s implementation class:

  ```java
  @EJB
  ExampleBean exampleBean;
  ```

- To obtain a reference to the no-interface view of an enterprise bean through JNDI lookup, use the javax.naming.InitialContext interface’s lookup method:

  ```java
  ExampleBean exampleBean = (ExampleBean)
      InitialContext.lookup("java:module/ExampleBean");
  ```

Clients do not use the new operator to obtain a new instance of an enterprise bean that uses a no-interface view.
Accessing Local Enterprise Beans That Implement Business Interfaces

Client access to enterprise beans that implement local business interfaces is accomplished through either dependency injection or JNDI lookup.

- To obtain a reference to the local business interface of an enterprise bean through dependency injection, use the `javax.ejb.EJB` annotation and specify the enterprise bean’s local business interface name:

  ```
  @EJB
  Example example;
  ```

- To obtain a reference to a local business interface of an enterprise bean through JNDI lookup, use the `javax.naming.InitialContext` interface’s `lookup` method:

  ```java
  ExampleLocal example = (ExampleLocal)
      InitialContext.lookup("java:module/ExampleLocal");
  ```

Remote Clients

A remote client of an enterprise bean has the following traits.

- It can run on a different machine and a different JVM from the enterprise bean it accesses. (It is not required to run on a different JVM.)
- It can be a web component, an application client, or another enterprise bean.
- To a remote client, the location of the enterprise bean is transparent.
- The enterprise bean must implement a business interface. That is, remote clients may not access an enterprise bean through a no-interface view.

To create an enterprise bean that allows remote access, you must either

- Decorate the business interface of the enterprise bean with the `@Remote` annotation:

  ```
  @Remote
  public interface InterfaceName { ... }
  ```

- Decorate the bean class with `@Remote`, specifying the business interface or interfaces:

  ```
  @Remote(InterfaceName.class)
  public class BeanName implements InterfaceName { ... }
  ```

The remote interface defines the business and lifecycle methods that are specific to the bean. For example, the remote interface of a bean named `BankAccountBean` might have business methods named `deposit` and `credit`. Figure 22–1 shows how the interface controls the client’s view of an enterprise bean.
Client access to an enterprise bean that implements a remote business interface is accomplished through either dependency injection or JNDI lookup.

- To obtain a reference to the remote business interface of an enterprise bean through dependency injection, use the `javax.ejb.EJB` annotation and specify the enterprise bean's remote business interface name:

  ```java
  @EJB
  Example example;
  ```

- To obtain a reference to a remote business interface of an enterprise bean through JNDI lookup, use the `javax.naming.InitialContext` interface's `lookup` method:

  ```java
  ExampleRemote example = (ExampleRemote)
  InitialContext.lookup("java:global/myApp/ExampleRemote");
  ```

## Web Service Clients

A web service client can access a Java EE application in two ways. First, the client can access a web service created with JAX-WS. (For more information on JAX-WS, see Chapter 19, "Building Web Services with JAX-WS.") Second, a web service client can invoke the business methods of a stateless session bean. Message beans cannot be accessed by web service clients.

Provided that it uses the correct protocols (SOAP, HTTP, WSDL), any web service client can access a stateless session bean, whether or not the client is written in the Java programming language. The client doesn’t even “know” what technology implements the service: stateless session bean, JAX-WS, or some other technology. In addition, enterprise beans and web components can be clients of web services. This flexibility enables you to integrate Java EE applications with web services.

A web service client accesses a stateless session bean through the bean’s web service endpoint implementation class. By default, all public methods in the bean class are accessible to web service clients. The `@WebMethod` annotation may be used to customize the behavior of web service methods. If the `@WebMethod` annotation is used to decorate the bean class’s methods, only those methods decorated with `@WebMethod` are exposed to web service clients.

For a code sample, see “A Web Service Example: helloservice” on page 476.
Method Parameters and Access

The type of access affects the parameters of the bean methods that are called by clients. The following sections apply not only to method parameters but also to method return values.

Isolation

The parameters of remote calls are more isolated than those of local calls. With remote calls, the client and the bean operate on different copies of a parameter object. If the client changes the value of the object, the value of the copy in the bean does not change. This layer of isolation can help protect the bean if the client accidentally modifies the data.

In a local call, both the client and the bean can modify the same parameter object. In general, you should not rely on this side effect of local calls. Perhaps someday you will want to distribute your components, replacing the local calls with remote ones.

As with remote clients, web service clients operate on different copies of parameters than does the bean that implements the web service.

Granularity of Accessed Data

Because remote calls are likely to be slower than local calls, the parameters in remote methods should be relatively coarse-grained. A coarse-grained object contains more data than a fine-grained one, so fewer access calls are required. For the same reason, the parameters of the methods called by web service clients should also be coarse-grained.

The Contents of an Enterprise Bean

To develop an enterprise bean, you must provide the following files:

- **Enterprise bean class**: Implements the business methods of the enterprise bean and any lifecycle callback methods.
- **Business interfaces**: Define the business methods implemented by the enterprise bean class. A business interface is not required if the enterprise bean exposes a local, no-interface view.
- **Helper classes**: Other classes needed by the enterprise bean class, such as exception and utility classes.

Package the programming artifacts in the preceding list either into an EJB JAR file (a stand-alone module that stores the enterprise bean) or within a web application archive (WAR) module.

Packaging Enterprise Beans in EJB JAR Modules

An EJB JAR file is portable and can be used for various applications.
To assemble a Java EE application, package one or more modules, such as EJB JAR files, into an EAR file, the archive file that holds the application. When deploying the EAR file that contains the enterprise bean’s EJB JAR file, you also deploy the enterprise bean to the GlassFish Server. You can also deploy an EJB JAR that is not contained in an EAR file. Figure 22–2 shows the contents of an EJB JAR file.

FIGURE 22–2  Structure of an Enterprise Bean JAR

Packaging Enterprise Beans in WAR Modules

Enterprise beans often provide the business logic of a web application. In these cases, packaging the enterprise bean within the web application’s WAR module simplifies deployment and application organization. Enterprise beans may be packaged within a WAR module as Java programming language class files or within a JAR file that is bundled within the WAR module.

To include enterprise bean class files in a WAR module, the class files should be in the WEB-INF/classes directory.

To include a JAR file that contains enterprise beans in a WAR module, add the JAR to the WEB-INF/lib directory of the WAR module.

WAR modules that contain enterprise beans do not require an ejb-jar.xml deployment descriptor. If the application uses ejb-jar.xml, it must be located in the WAR module’s WEB-INF directory.

JAR files that contain enterprise bean classes packaged within a WAR module are not considered EJB JAR files, even if the bundled JAR file conforms to the format of an EJB JAR file.
The enterprise beans contained within the JAR file are semantically equivalent to enterprise beans located in the WAR module's WEB-INF/classes directory, and the environment namespace of all the enterprise beans are scoped to the WAR module.

For example, suppose that a web application consists of a shopping cart enterprise bean, a credit card processing enterprise bean, and a Java servlet front end. The shopping cart bean exposes a local, no-interface view and is defined as follows:

```java
package com.example.cart;
@Stateless
public class CartBean { ... }
```

The credit card processing bean is packaged within its own JAR file, cc.jar, exposes a local, no-interface view, and is defined as follows:

```java
package com.example.cc;
@Stateless
public class CreditCardBean { ... }
```

The servlet, com.example.web.StoreServlet, handles the web front end and uses both CartBean and CreditCardBean. The WAR module layout for this application looks as follows:

```
WEB-INF/classes/com/example/cart/CartBean.class
WEB-INF/classes/com/example/web/StoreServlet
WEB-INF/lib/cc.jar
WEB-INF/ejb-jar.xml
WEB-INF/web.xml
```

## Naming Conventions for Enterprise Beans

Because enterprise beans are composed of multiple parts, it's useful to follow a naming convention for your applications. Table 22–2 summarizes the conventions for the example beans in this tutorial.

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise bean name</td>
<td><code>nameBean</code></td>
<td>AccountBean</td>
</tr>
<tr>
<td>Enterprise bean class</td>
<td><code>nameBean</code></td>
<td>AccountBean</td>
</tr>
<tr>
<td>Business interface</td>
<td><code>name</code></td>
<td>Account</td>
</tr>
</tbody>
</table>
The Lifecycles of Enterprise Beans

An enterprise bean goes through various stages during its lifetime, or lifecycle. Each type of enterprise bean (stateful session, stateless session, singleton session, or message-driven) has a different lifecycle.

The descriptions that follow refer to methods that are explained along with the code examples in the next two chapters. If you are new to enterprise beans, you should skip this section and run the code examples first.

The Lifecycle of a Stateful Session Bean

Figure 22–3 illustrates the stages that a session bean passes through during its lifetime. The client initiates the lifecycle by obtaining a reference to a stateful session bean. The container performs any dependency injection and then invokes the method annotated with @PostConstruct, if any. The bean is now ready to have its business methods invoked by the client.

While in the ready stage, the EJB container may decide to deactivate, or passivate, the bean by moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the method annotated @PrePassivate, if any, immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean, calls the method annotated @PostActivate, if any, and then moves it to the ready stage.
At the end of the lifecycle, the client invokes a method annotated @Remove, and the EJB container calls the method annotated @PreDestroy, if any. The bean’s instance is then ready for garbage collection.

Your code controls the invocation of only one lifecycle method: the method annotated @Remove. All other methods in Figure 22–3 are invoked by the EJB container. See Chapter 45, “Resources and Resource Adapters,” for more information.

The Lifecycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its lifecycle has only two stages: nonexistent and ready for the invocation of business methods. Figure 22–4 illustrates the stages of a stateless session bean.

The EJB container typically creates and maintains a pool of stateless session beans, beginning the stateless session bean’s lifecycle. The container performs any dependency injection and then invokes the method annotated @PostConstruct, if it exists. The bean is now ready to have its business methods invoked by a client.

At the end of the lifecycle, the EJB container calls the method annotated @PreDestroy, if it exists. The bean’s instance is then ready for garbage collection.

The Lifecycle of a Singleton Session Bean

Like a stateless session bean, a singleton session bean is never passivated and has only two stages, nonexistent and ready for the invocation of business methods, as shown in Figure 22–5.
The EJB container initiates the singleton session bean lifecycle by creating the singleton instance. This occurs upon application deployment if the singleton is annotated with the @Startup annotation. The container performs any dependency injection and then invokes the method annotated @PostConstruct, if it exists. The singleton session bean is now ready to have its business methods invoked by the client.

At the end of the lifecycle, the EJB container calls the method annotated @PreDestroy, if it exists. The singleton session bean is now ready for garbage collection.

**The Lifecycle of a Message-Driven Bean**

Figure 22–6 illustrates the stages in the lifecycle of a message-driven bean.

The EJB container usually creates a pool of message-driven bean instances. For each instance, the EJB container performs these tasks.

1. If the message-driven bean uses dependency injection, the container injects these references before instantiating the instance.
2. The container calls the method annotated @PostConstruct, if any.
Like a stateless session bean, a message-driven bean is never passivated and has only two states: nonexistent and ready to receive messages.

At the end of the lifecycle, the container calls the method annotated @PreDestroy, if any. The bean’s instance is then ready for garbage collection.

Further Information about Enterprise Beans

For more information on Enterprise JavaBeans technology, see

- Enterprise JavaBeans 3.1 specification:
- Enterprise JavaBeans website:
  http://www.oracle.com/technetwork/java/ejb-141389.html
Getting Started with Enterprise Beans

This chapter shows how to develop, deploy, and run a simple Java EE application named converter. The purpose of converter is to calculate currency conversions between Japanese yen and Eurodollars. The converter application consists of an enterprise bean, which performs the calculations, and a web client.

Here’s an overview of the steps you’ll follow in this chapter:

1. Create the enterprise bean: ConverterBean.
2. Create the web client.
3. Deploy converter onto the server.
4. Using a browser, run the web client.

Before proceeding, make sure that you’ve done the following:

- Read Chapter 1, “Overview”
- Become familiar with enterprise beans (see Chapter 22, “Enterprise Beans”)
- Started the server (see “Starting and Stopping the GlassFish Server” on page 73)

The following topics are addressed here:

- “Creating the Enterprise Bean” on page 455
- “Modifying the Java EE Application” on page 458

Creating the Enterprise Bean

The enterprise bean in our example is a stateless session bean called ConverterBean. The source code for ConverterBean is in the tut-install/examples/ejb/converter/src/java/ directory.

Creating ConverterBean requires these steps:

1. Coding the bean’s implementation class (the source code is provided)
2. Compiling the source code
Coding the Enterprise Bean Class

The enterprise bean class for this example is called ConverterBean. This class implements two business methods: dollarToYen and yenToEuro. Because the enterprise bean class doesn’t implement a business interface, the enterprise bean exposes a local, no-interface view. The public methods in the enterprise bean class are available to clients that obtain a reference to ConverterBean. The source code for the ConverterBean class is as follows:

```java
package converter.ejb;
import java.math.BigDecimal;
import javax.ejb.*;

@Stateless
public class ConverterBean {
    private BigDecimal yenRate = new BigDecimal("83.0602");
    private BigDecimal euroRate = new BigDecimal("0.0093016");

    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }

    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

Note the @Stateless annotation decorating the enterprise bean class. This annotation lets the container know that ConverterBean is a stateless session bean.

Creating the converter Web Client

The web client is contained in the following servlet class:

```java
tut-install/examples.ejb/converter/src/java/converter/web/ConverterServlet.java
```

A Java servlet is a web component that responds to HTTP requests.

The ConverterServlet class uses dependency injection to obtain a reference to ConverterBean. The javax.ejb.EJB annotation is added to the declaration of the private member variable converterBean, which is of type ConverterBean. ConverterBean exposes a local, no-interface view, so the enterprise bean implementation class is the variable type:

```java
@WebServlet
public class ConverterServlet extends HttpServlet {
    @EJB
    ConverterBean converterBean;
    ...
}
```
When the user enters an amount to be converted to yen and euro, the amount is retrieved from the request parameters; then the ConverterBean.dollarToYen and the ConverterBean.yenToEuro methods are called:

```java
try {
    String amount = request.getParameter("amount");
    if (amount != null && amount.length() > 0) {
        // convert the amount to a BigDecimal from the request parameter
        BigDecimal d = new BigDecimal(amount);
        // call the ConverterBean.dollarToYen() method to get the amount
        // in Yen
        BigDecimal yenAmount = converter.dollarToYen(d);

        // call the ConverterBean.yenToEuro() method to get the amount
        // in Euros
        BigDecimal euroAmount = converter.yenToEuro(yenAmount);
    }
    ...
}
```

The results are displayed to the user.

**Running the converter Example**

Now you are ready to compile the enterprise bean class (ConverterBean.java) and the servlet class (ConverterServlet.java) and to package the compiled classes into a WAR file. You can use either NetBeans IDE or Ant to build, package, deploy, and run the converter example.

**To Build, Package, and Deploy the converter Example in NetBeans IDE**

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   `tut-install/examples/ebj/
3. Select the converter folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the converter project and select Deploy.
To Build, Package, and Deploy the converter Example Using Ant

1. In a terminal window, go to:
   tut-install/examples/ejb/converter/

2. Type the following command:
   `ant all`

   This command calls the default task, which compiles the source files for the enterprise bean and the servlet, placing the class files in the build subdirectory (not the src directory) of the project. The default task packages the project into a WAR module: converter.war. For more information about the Ant tool, see “Building the Examples” on page 75.

   **Note** – When compiling the code, the ant task includes the Java EE API JAR files in the classpath. These JARs reside in the modules directory of your GlassFish Server installation. If you plan to use other tools to compile the source code for Java EE components, make sure that the classpath includes the Java EE API JAR files.

To Run the converter Example

1. Open a web browser to the following URL:
   `http://localhost:8080/converter`

   The Servlet ConverterServlet page opens.

2. Type 100 in the input field and click Submit.

   A second page opens, showing the converted values.

Modifying the Java EE Application

The GlassFish Server supports iterative development. Whenever you make a change to a Java EE application, you must redeploy the application.

To Modify a Class File

To modify a class file in an enterprise bean, you change the source code, recompile it, and redeploy the application. For example, to update the exchange rate in the dollarToYen business method of the ConverterBean class, you would follow these steps.

To modify ConverterServlet, the procedure is the same.

1. Edit `ConverterBean.java` and save the file.
2 Recompile the source file.

- To recompile `ConverterBean.java` in NetBeans IDE, right-click the converter project and select Run.
  
  This recompiles the `ConverterBean.java` file, replaces the old class file in the build directory, and redeploy the application to GlassFish Server.

- Recompile `ConverterBean.java` using Ant:
  
  a. In a terminal window, go to the `tut-install/examples.ejb/converter/` directory.

  b. Type the following command:
     ```bash
     ant all
     ```
     
     This command repackages, deploys, and runs the application.
Running the Enterprise Bean Examples

Session beans provide a simple but powerful way to encapsulate business logic within an application. They can be accessed from remote Java clients, web service clients, and components running in the same server.

In Chapter 23, "Getting Started with Enterprise Beans," you built a stateless session bean named ConverterBean. This chapter examines the source code of four more session beans:

- CartBean: a stateful session bean that is accessed by a remote client
- CounterBean: a singleton session bean
- HelloServiceBean: a stateless session bean that implements a web service
- TimerSessionBean: a stateless session bean that sets a timer

The following topics are addressed here:

- "The cart Example" on page 461
- "A Singleton Session Bean Example: counter" on page 468
- "A Web Service Example: helloservice" on page 476
- "Using the Timer Service" on page 479
- "Handling Exceptions" on page 489

The cart Example

The cart example represents a shopping cart in an online bookstore and uses a stateful session bean to manage the operations of the shopping cart. The bean’s client can add a book to the cart, remove a book, or retrieve the cart’s contents. To assemble cart, you need the following code:

- Session bean class (CartBean)
- Remote business interface (Cart)

All session beans require a session bean class. All enterprise beans that permit remote access must have a remote business interface. To meet the needs of a specific application, an enterprise
bean may also need some helper classes. The CartBean session bean uses two helper classes, BookException and IdVerifier, which are discussed in the section “Helper Classes” on page 466.

The source code for this example is in the tut-install/examples/ejb/cart/ directory.

**The Business Interface**

The Cart business interface is a plain Java interface that defines all the business methods implemented in the bean class. If the bean class implements a single interface, that interface is assumed to be the business interface. The business interface is a local interface unless it is annotated with the javax.ejb.Remote annotation; the javax.ejb.Local annotation is optional in this case.

The bean class may implement more than one interface. In that case, the business interfaces must either be explicitly annotated @Local or @Remote or be specified by decorating the bean class with @Local or @Remote. However, the following interfaces are excluded when determining whether the bean class implements more than one interface:

- java.io.Serializable
- java.io.Externalizable
- Any of the interfaces defined by the javax.ejb package

The source code for the Cart business interface follows:

```java
package cart.ejb;

import cart.util.BookException;
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface Cart {
    public void initialize(String person) throws BookException;
    public void initialize(String person, String id) throws BookException;
    public void addBook(String title);
    public void removeBook(String title) throws BookException;
    public List<String> getContents();
    public void remove();
}
```
Session Bean Class

The session bean class for this example is called CartBean. Like any stateful session bean, the CartBean class must meet the following requirements.

- The class is annotated @Stateful.
- The class implements the business methods defined in the business interface.

Stateful session beans also may

- Implement the business interface, a plain Java interface. It is good practice to implement the bean’s business interface.
- Implement any optional lifecycle callback methods, annotated @PostConstruct, @PreDestroy, @PostActivate, and @PrePassivate.
- Implement any optional business methods annotated @Remove.

The source code for the CartBean class follows:

```java
package cart.ejb;
import cart.util.BookException;
import cart.util.IdVerifier;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;
@Stateful
public class CartBean implements Cart {
    String customerName;
    String customerId;
    List<String> contents;

    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
        customerId = "0";
        contents = new ArrayList<String>();
    }

    public void initialize(String person, String id) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
        IdVerifier idChecker = new IdVerifier();
    }
```

The cart Example
```java
if (idChecker.validate(id)) {
    customerId = id;
} else {
    throw new BookException("Invalid id: " + id);
}

customerId = id;
contents = new ArrayList<String>();
}

public void addBook(String title) {
    contents.add(title);
}

public void removeBook(String title) throws BookException {
    boolean result = contents.remove(title);
    if (result == false) {
        throw new BookException(title + " not in cart.");
    }
}

public List<String> getContents() {
    return contents;
}

@Remove
public void remove() {
    contents = null;
}

Lifecycle Callback Methods

A method in the bean class may be declared as a lifecycle callback method by annotating the method with the following annotations:

- `javax.annotation.PostConstruct`: Methods annotated with `@PostConstruct` are invoked by the container on newly constructed bean instances after all dependency injection has completed and before the first business method is invoked on the enterprise bean.

- `javax.annotation.PreDestroy`: Methods annotated with `@PreDestroy` are invoked after any method annotated `@Remove` has completed and before the container removes the enterprise bean instance.

- `javax.ejb.PostActivate`: Methods annotated with `@PostActivate` are invoked by the container after the container moves the bean from secondary storage to active status.

- `javax.ejb.PrePassivate`: Methods annotated with `@PrePassivate` are invoked by the container before it passivates the enterprise bean, meaning that the container temporarily removes the bean from the environment and saves it to secondary storage.

Lifecycle callback methods must return `void` and have no parameters.
Business Methods

The primary purpose of a session bean is to run business tasks for the client. The client invokes business methods on the object reference it gets from dependency injection or JNDI lookup. From the client's perspective, the business methods appear to run locally, although they run remotely in the session bean. The following code snippet shows how the CartClient program invokes the business methods:

```java
cart.create("Duke DeEarl", "123");
... cart.addBook("Bel Canto");
... List<String> bookList = cart.getContents();
... cart.removeBook("Gravity's Rainbow");
```

The CartBean class implements the business methods in the following code:

```java
public void addBook(String title) {
    contents.addElement(title);
}
public void removeBook(String title) throws BookException {
    boolean result = contents.remove(title);
    if (result == false) {
        throw new BookException(title + " not in cart.");
    }
}
public List<String> getContents() {
    return contents;
}
```

The signature of a business method must conform to these rules.

- The method name must not begin with `ejb`, to avoid conflicts with callback methods defined by the EJB architecture. For example, you cannot call a business method `ejbCreate` or `ejbActivate`.
- The access control modifier must be `public`.
- If the bean allows remote access through a remote business interface, the arguments and return types must be legal types for the Java Remote Method Invocation (RMI) API.
- If the bean is a web service endpoint, the arguments and return types for the methods annotated `@WebMethod` must be legal types for JAX-WS.
- The modifier must not be `static` or `final`.

The `throws` clause can include exceptions that you define for your application. The `removeBook` method, for example, throws a `BookException` if the book is not in the cart.

To indicate a system-level problem, such as the inability to connect to a database, a business method should throw a `javax.ejb.EJBException`. The container will not wrap application
exceptions, such as BookException. Because EJBException is a subclass of RuntimeException, you do not need to include it in the throws clause of the business method.

The @Remove Method

Business methods annotated with javax.ejb.Remove in the stateful session bean class can be invoked by enterprise bean clients to remove the bean instance. The container will remove the enterprise bean after a @Remove method completes, either normally or abnormally.

In CartBean, the remove method is a @Remove method:

```java
@Remove
public void remove() {
    contents = null;
}
```

Helper Classes

The CartBean session bean has two helper classes: BookException and IdVerifier. The BookException is thrown by the removeBook method, and the IdVerifier validates the customerId in one of the create methods. Helper classes may reside in an EJB JAR file that contains the enterprise bean class, a WAR file if the enterprise bean is packaged within a WAR, or in an EAR that contains an EJB JAR or a WAR file that contains an enterprise bean.

Running the cart Example

Now you are ready to compile the remote interface (Cart.java), the home interface (CartHome.java), the enterprise bean class (CartBean.java), the client class (CartClient.java), and the helper classes (BookException.java and IdVerifier.java). Follow these steps.

You can use either NetBeans IDE or Ant to build, package, deploy, and run the cart application.

To Run the cart Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   tut-install/examples/ejb/
3. Select the cart folder.
4 Select the Open as Main Project and Open Required Projects checkboxes.

5 Click Open Project.

6 In the Projects tab, right-click the cart project and select Deploy.
   This builds and packages the application into cart.ear, located in tut-install/examples/ejb/cart/dist/, and deploys this EAR file to your GlassFish Server instance.

7 From the Run menu, choose Run Main Project.
   You will see the output of the cart application client in the Output pane:

   ... 
   Retrieving book title from cart: Infinite Jest 
   Retrieving book title from cart: Bel Canto 
   Retrieving book title from cart: Kafka on the Shore 
   Removing "Gravity’s Rainbow" from cart.
   Caught a BookException: "Gravity’s Rainbow" not in cart.
   Java Result: 1
   run-cart-app-client:
   run-nb:
   BUILD SUCCESSFUL (total time: 14 seconds)

▼ To Run the cart Example Using Ant

1 In a terminal window, go to:
   tut-install/examples/ejb/cart/

2 Type the following command:
   ant
   This command calls the default target, which builds and packages the application into an EAR file, cart.ear, located in the dist directory.

3 Type the following command:
   ant deploy
   The cart.ear file is deployed to the GlassFish Server.

4 Type the following command:
   ant run
   This task retrieves the application client JAR, cartClient.jar, and runs the application client.
   The client JAR, cartClient.jar, contains the application client class, the helper class BookException, and the Cart business interface.
   This task is equivalent to running the following command:

   appclient -client cartClient.jar
When you run the client, the application client container injects any component references declared in the application client class, in this case the reference to the Cart enterprise bean.

**The all Task**
As a convenience, the all task will build, package, deploy, and run the application. To do this, enter the following command:

`ant all`

### A Singleton Session Bean Example: counter

The counter example demonstrates how to create a singleton session bean.

### Creating a Singleton Session Bean

The `javax.ejb.Singleton` annotation is used to specify that the enterprise bean implementation class is a singleton session bean:

```java
@Singleton
public class SingletonBean { ... }
```

### Initializing Singleton Session Beans

The EJB container is responsible for determining when to initialize a singleton session bean instance unless the singleton session bean implementation class is annotated with the `javax.ejb.Startup` annotation. In this case, sometimes called *eager initialization*, the EJB container must initialize the singleton session bean upon application startup. The singleton session bean is initialized before the EJB container delivers client requests to any enterprise beans in the application. This allows the singleton session bean to perform, for example, application startup tasks.

The following singleton session bean stores the status of an application and is eagerly initialized:

```java
@Startup
@Singleton
public class StatusBean {
    private String status;

    @PostConstruct
    void init {
        status = "Ready";
    }
    ...
}
```
Sometimes multiple singleton session beans are used to initialize data for an application and therefore must be initialized in a specific order. In these cases, use the `javax.ejb.DependsOn` annotation to declare the startup dependencies of the singleton session bean. The `@DependsOn` annotation’s `value` attribute is one or more strings that specify the name of the target singleton session bean. If more than one dependent singleton bean is specified in `@DependsOn`, the order in which they are listed is not necessarily the order in which the EJB container will initialize the target singleton session beans.

The following singleton session bean, `PrimaryBean`, should be started up first:

```java
@Singleton
public class PrimaryBean { ... }
```

SecondaryBean depends on `PrimaryBean`:

```java
@Singleton
@DependsOn("PrimaryBean")
public class SecondaryBean { ... }
```

This guarantees that the EJB container will initialize `PrimaryBean` before `SecondaryBean`.

The following singleton session bean, `TertiaryBean`, depends on `PrimaryBean` and `SecondaryBean`:

```java
@Singleton
@DependsOn({"PrimaryBean", "SecondaryBean"})
public class TertiaryBean { ... }
```

SecondaryBean explicitly requires `PrimaryBean` to be initialized before it is initialized, through its own `@DependsOn` annotation. In this case, the EJB container will first initialize `PrimaryBean`, then `SecondaryBean`, and finally `TertiaryBean`.

If, however, `SecondaryBean` did not explicitly depend on `PrimaryBean`, the EJB container may initialize either `PrimaryBean` or `SecondaryBean` first. That is, the EJB container could initialize the singletons in the following order: `SecondaryBean`, `PrimaryBean`, `TertiaryBean`.

### Managing Concurrent Access in a Singleton Session Bean

Singleton session beans are designed for concurrent access, situations in which many clients need to access a single instance of a session bean at the same time. A singleton’s client needs only a reference to a singleton in order to invoke any business methods exposed by the singleton and doesn’t need to worry about any other clients that may be simultaneously invoking business methods on the same singleton.

When creating a singleton session bean, concurrent access to the singleton’s business methods can be controlled in two ways: container-managed concurrency and bean-managed concurrency.
The `javax.ejb.ConcurrencyManagement` annotation is used to specify container-managed or bean-managed concurrency for the singleton. With `@ConcurrencyManagement`, a type attribute must be set to either `javax.ejb.ConcurrencyManagementType.CONTAINER` or `javax.ejb.ConcurrencyManagementType.BEAN`. If no `@ConcurrencyManagement` annotation is present on the singleton implementation class, the EJB container default of container-managed concurrency is used.

**Container-Managed Concurrency**

If a singleton uses *container-managed concurrency*, the EJB container controls client access to the business methods of the singleton. The `javax.ejb.Lock` annotation and a `javax.ejb.LockType` type are used to specify the access level of the singleton’s business methods or `@Timeout` methods. The `LockType` enumerated types are `READ` and `WRITE`.

Annotate a singleton’s business or timeout method with `@Lock(LockType.READ)` if the method can be concurrently accessed, or shared, with many clients. Annotate the business or timeout method with `@Lock(LockType.WRITE)` if the singleton session bean should be locked to other clients while a client is calling that method. Typically, the `@Lock(LockType.WRITE)` annotation is used when clients are modifying the state of the singleton.

Annotating a singleton class with `@Lock` specifies that all the business methods and any timeout methods of the singleton will use the specified lock type unless they explicitly set the lock type with a method-level `@Lock` annotation. If no `@Lock` annotation is present on the singleton class, the default lock type, `@Lock(LockType.WRITE)`, is applied to all business and timeout methods.

The following example shows how to use the `@ConcurrencyManagement`, `@Lock(LockType.READ)`, and `@Lock(LockType.WRITE)` annotations for a singleton that uses container-managed concurrency.

Although by default, singletons use container-managed concurrency, the `@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)` annotation may be added at the class level of the singleton to explicitly set the concurrency management type:

```java
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class ExampleSingletonBean {
    private String state;

    @Lock(LockType.READ)
    public String getState() {
        return state;
    }

    @Lock(LockType.WRITE)
    public void setState(String newState) {
        state = newState;
    }
}
```
The `getState` method can be accessed by many clients at the same time because it is annotated with `@Lock(LockType.READ)`. When the `setState` method is called, however, all the methods in `ExampleSingletonBean` will be locked to other clients because `setState` is annotated with `@Lock(LockType.WRITE)`. This prevents two clients from attempting to simultaneously change the state variable of `ExampleSingletonBean`.

The `getData` and `getStatus` methods in the following singleton are of type `READ`, and the `setStatus` method is of type `WRITE`:

```java
@Singleton
@Lock(LockType.READ)
public class SharedSingletonBean {
    private String data;
    private String status;

    public String getData() {
        return data;
    }

    public String getStatus() {
        return status;
    }

    @Lock(LockType.WRITE)
    public void setStatus(String newStatus) {
        status = newStatus;
    }
}
```

If a method is of locking type `WRITE`, client access to all the singleton’s methods is blocked until the current client finishes its method call or an access timeout occurs. When an access timeout occurs, the EJB container throws a `javax.ejb.ConcurrentAccessTimeoutException`. The `javax.ejb.AccessTimeout` annotation is used to specify the number of milliseconds before an access timeout occurs. If added at the class level of a singleton, `@AccessTimeout` specifies the access timeout value for all methods in the singleton unless a method explicitly overrides the default with its own `@AccessTimeout` annotation.

The `@AccessTimeout` annotation can be applied to both `@Lock(LockType.READ)` and `@Lock(LockType.WRITE)` methods. The `@AccessTimeout` annotation has one required element, `value`, and one optional element, `unit`. By default, the value is specified in milliseconds. To change the value unit, set `unit` to one of the `java.util.concurrent.TimeUnit` constants: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, or `SECONDS`.

The following singleton has a default access timeout value of 120,000 milliseconds, or 2 minutes. The `doTediousOperation` method overrides the default access timeout and sets the value to 360,000 milliseconds, or 6 minutes.

```java
@Singleton
@AccessTimeout(value=120000)
public class StatusSingletonBean {
    private String status;
    ASingletonSessionBeanExample:counter
}````
@Lock(LockType.WRITE)
public void setStatus(String new Status) {
    status = newStatus;
}

@Lock(LockType.WRITE)
@AccessTimeout(value=360000)
public void doTediousOperation {
    ...
}
}

The following singleton has a default access timeout value of 60 seconds, specified using the TimeUnit.SECONDS constant:

@Singleton
@AccessTimeout(value=60, timeUnit=SECONDS)
public class StatusSingletonBean {
    ...
}

**Bean-Managed Concurrency**

Singletons that use bean-managed concurrency allow full concurrent access to all the business and timeout methods in the singleton. The developer of the singleton is responsible for ensuring that the state of the singleton is synchronized across all clients. Developers who create singletons with bean-managed concurrency are allowed to use the Java programming language synchronization primitives, such as synchronization and volatile, to prevent errors during concurrent access.

Add a @ConcurrencyManagement annotation with the type set to ConcurrencyManagementType.BEAN at the class level of the singleton to specify bean-managed concurrency:

@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
@Singleton
public class AnotherSingletonBean {
    ...
}

**Handling Errors in a Singleton Session Bean**

If a singleton session bean encounters an error when initialized by the EJB container, that singleton instance will be destroyed.

Unlike other enterprise beans, once a singleton session bean instance is initialized, it is not destroyed if the singleton’s business or lifecycle methods cause system exceptions. This ensures that the same singleton instance is used throughout the application lifecycle.

**The Architecture of the counter Example**

The counter example consists of a singleton session bean, CounterBean, and a JavaServer Faces Facelets web front end.
CounterBean is a simple singleton with one method, getHits, that returns an integer representing the number of times a web page has been accessed. Here is the code of CounterBean:

```java
package counter.ejb;

import javax.ejb.Singleton;

/**
 * CounterBean is a simple singleton session bean that records the number
 * of hits to a web page.
 */
@Singleton
public class CounterBean {
    private int hits = 1;
    
    // Increment and return the number of hits
    public int getHits() {
        return hits++;
    }
}
```

The `@Singleton` annotation marks CounterBean as a singleton session bean. CounterBean uses a local, no-interface view.

CounterBean uses the EJB container’s default metadata values for singletons to simplify the coding of the singleton implementation class. There is no `@ConcurrencyManagement` annotation on the class, so the default of container-managed concurrency access is applied. There is no `@Lock` annotation on the class or business method, so the default of `@Lock(WRITE)` is applied to the only business method, getHits.

The following version of CounterBean is functionally equivalent to the preceding version:

```java
package counter.ejb;

import javax.ejb.Singleton;
import javax.ejb.ConcurrencyManagement;
import static javax.ejb.ConcurrencyManagementType.CONTAINER;
import javax.ejb.Lock;
import javax.ejb.LockType.WRITE;

/**
 * CounterBean is a simple singleton session bean that records the number
 * of hits to a web page.
 */
@Singleton
@ConcurrencyManagement(CONTAINER)
public class CounterBean {
    private int hits = 1;

    // Increment and return the number of hits
    @Lock(WRITE)
    public int getHits() {
        return hits++;
    }
}
```
The web front end of counter consists of a JavaServer Faces managed bean, Count.java, that is used by the Facelets XHTML files template.xhtml and template-client.xhtml. The Count JavaServer Faces managed bean obtains a reference to CounterBean through dependency injection. Count defines a hitCount JavaBeans property. When the getHitCount getter method is called from the XHTML files, CounterBean’s getHits method is called to return the current number of page hits.

Here's the Count managed bean class:

```java
@ManagedBean
@SessionScoped
public class Count {
    @EJB
    private CounterBean counterBean;
    private int hitCount;
    public Count() {
        this.hitCount = 0;
    }
    public int getHitCount() {
        hitCount = counterBean.getHits();
        return hitCount;
    }
    public void setHitCount(int newHits) {
        this.hitCount = newHits;
    }
}
```

The template.xhtml and template-client.xhtml files are used to render a Facelets view that displays the number of hits to that view. The template-client.xhtml file uses an expression language statement, #{count.hitCount}, to access the hitCount property of the Count managed bean. Here is the content of template-client.xhtml:

```xml
<ui:define name="title">
    This page has been accessed #{count.hitCount} time(s).
</ui:define>
```
Running the counter Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the counter example.

▼ To Run the counter Example Using NetBeans IDE

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to:
   tut-install/examples/ejb/

3 Select the counter folder.

4 Select the Open as Main Project check box.

5 Click Open Project.

6 In the Projects tab, right-click the counter project and select Run.
   A web browser will open the URL http://localhost:8080/counter, which displays the number of hits.

7 Click the browser’s Refresh button to see the hit count increment.

▼ To Run the counter Example Using Ant

1 In a terminal window, go to:
   tut-install/examples/ejb/counter/

2 Type the following command:
   ant all
This will build and deploy counter to your GlassFish Server instance.

3 In a web browser, type the following URL:
   http://localhost:8080/counter

4 Click the browser’s Refresh button to see the hit count increment.

A Web Service Example: helloservice

This example demonstrates a simple web service that generates a response based on information received from the client. HelloServiceBean is a stateless session bean that implements a single method: sayHello. This method matches the sayHello method invoked by the client described in “A Simple JAX-WS Application Client” on page 372.

The Web Service Endpoint Implementation Class

HelloServiceBean is the endpoint implementation class, typically the primary programming artifact for enterprise bean web service endpoints. The web service endpoint implementation class has the following requirements.

- The class must be annotated with either the javax.jws.WebService or the javax.jws.WebServiceProvider annotation.
- The implementing class may explicitly reference an SEI through the endpointInterface element of the @WebService annotation but is not required to do so. If no endpointInterface is specified in @WebService, an SEI is implicitly defined for the implementing class.
- The business methods of the implementing class must be public and must not be declared static or final.
- Business methods that are exposed to web service clients must be annotated with javax.jws.WebMethod.
- Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types. See the list of JAXB default data type bindings at “Types Supported by JAX-WS” on page 377.
- The implementing class must not be declared final and must not be abstract.
- The implementing class must have a default public constructor.
- The endpoint class must be annotated @Stateless.
- The implementing class must not define the finalize method.
- The implementing class may use the javax.annotation.PostConstruct or javax.annotation.PreDestroy annotations on its methods for lifecycle event callbacks.
The `@PostConstruct` method is called by the container before the implementing class
begins responding to web service clients.

The `@PreDestroy` method is called by the container before the endpoint is removed from
operation.

**Stateless Session Bean Implementation Class**

The `HelloServiceBean` class implements the `sayHello` method, which is annotated
`@WebMethod`. The source code for the `HelloServiceBean` class follows:

```java
package com.sun.tutorial.javaee.ejb;
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;
@Stateless
@WebService
public class HelloServiceBean {
    private String message = "Hello, ";
    public void HelloServiceBean() {}
    @WebMethod
    public String sayHello(String name) {
        return message + name + ";
    }
}
```

**Running the helloservice Example**

You can use either NetBeans IDE or Ant to build, package, and deploy the `helloservice`
example. You can then use the Administration Console to test the web service endpoint
methods.

▼ **To Build, Package, and Deploy the helloservice Example Using
NetBeans IDE**

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   `tut-install/examples.ejb/
3. Select the `helloservice` folder.
4. Select the Open as Main Project check box.
Click Open Project.

In the Projects tab, right-click the helloservice project and select Deploy.

This builds and packages the application into helloservice.ear, located in tut-install/examples/ejb/helloservice/dist/, and deploys this EAR file to the GlassFish Server.

To Build, Package, and Deploy the helloservice Example Using Ant

1. In a terminal window, go to:
   tut-install/examples/ejb/helloservice/

2. Type the following command:
   ant
   This runs the default task, which compiles the source files and packages the application into a JAR file located at tut-install/examples/ejb/helloservice/dist/helloservice.jar.

3. To deploy helloservice, type the following command:
   ant deploy
   Upon deployment, the GlassFish Server generates additional artifacts required for web service invocation, including the WSDL file.

To Test the Service without a Client

The GlassFish Server Administration Console allows you to test the methods of a web service endpoint. To test the sayHello method of HelloServiceBean, follow these steps.

1. Open the Administration Console by opening the following URL in a web browser:
   http://localhost:4848/

2. In the left pane of the Administration Console, select the Applications node.

3. In the Applications table, click helloservice.

4. In the Modules and Components table, click View Endpoint.

5. On the Web Service Endpoint Information page, click the Tester link:
   /HelloServiceBeanService/HelloServiceBean?Tester
   A Web Service Test Links page opens.

6. On the Web Service Test Links page, click the non-secure link (the one that specifies port 8080).
   A HelloServiceBeanService Web Service Tester page opens.
7 Under Methods, type a name as the parameter to the sayHello method.

8 Click the sayHello button.

The sayHello Method invocation page opens. Under Method returned, you’ll see the response from the endpoint.

Using the Timer Service

Applications that model business work flows often rely on timed notifications. The timer service of the enterprise bean container enables you to schedule timed notifications for all types of enterprise beans except for stateful session beans. You can schedule a timed notification to occur according to a calendar schedule, at a specific time, after a duration of time, or at timed intervals. For example, you could set timers to go off at 10:30 a.m. on May 23, in 30 days, or every 12 hours.

Enterprise bean timers are either programmatic timers or automatic timers. Programmatic timers are set by explicitly calling one of the timer creation methods of the TimerService interface. Automatic timers are created upon the successful deployment of an enterprise bean that contains a method annotated with the java.ejb.Schedule or java.ejb.Schedules annotations.

Creating Calendar-Based Timer Expressions

Timers can be set according to a calendar-based schedule, expressed using a syntax similar to the UNIX cron utility. Both programmatic and automatic timers can use calendar-based timer expressions. Table 24–1 shows the calendar-based timer attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Default Value</th>
<th>Allowable Values and Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>second</td>
<td>One or more seconds within a minute</td>
<td>0</td>
<td>0 to 59. For example: second=&quot;30&quot;.</td>
</tr>
<tr>
<td>minute</td>
<td>One or more minutes within an hour</td>
<td>0</td>
<td>0 to 59. For example: minute=&quot;15&quot;.</td>
</tr>
<tr>
<td>hour</td>
<td>One or more hours within a day</td>
<td>0</td>
<td>0 to 23. For example: hour=&quot;13&quot;.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>One or more days within a week</td>
<td>*</td>
<td>0 to 7 (both 0 and 7 refer to Sunday). For example: dayOfWeek=&quot;3&quot;. Sun, Mon, Tue, Wed, Thu, Fri, Sat. For example: dayOfWeek=&quot;Mon&quot;.</td>
</tr>
</tbody>
</table>
### Specifying Multiple Values in Calendar Expressions

You can specify multiple values in calendar expressions, as described in the following sections.

#### Using Wildcards in Calendar Expressions

Setting an attribute to an asterisk symbol (*) represents all allowable values for the attribute.

The following expression represents every minute:

`minute="*"`

The following expression represents every day of the week:

`dayOfWeek="*"`

#### Specifying a List of Values

To specify two or more values for an attribute, use a comma (,) to separate the values. A range of values is allowed as part of a list. Wildcards and intervals, however, are not allowed.

Duplicates within a list are ignored.

The following expression sets the day of the week to Tuesday and Thursday:

`dayOfWeek="Tue, Thu"`

The following expression represents 4:00 a.m., every hour from 9:00 a.m. to 5:00 p.m. using a range, and 10:00 p.m.
hour="4, 9–17, 22"

### Specifying a Range of Values

Use a dash character (–) to specify an inclusive range of values for an attribute. Members of a range cannot be wildcards, lists, or intervals. A range of the form $x–x$, is equivalent to the single-valued expression $x$. A range of the form $x–y$ where $x$ is greater than $y$ is equivalent to the expression $x–maximum\ value, minimum\ value–y$. That is, the expression begins at $x$, rolls over to the beginning of the allowable values, and continues up to $y$.

The following expression represents 9:00 a.m. to 5:00 p.m.:

hour="9–17"

The following expression represents Friday through Monday:

dayOfWeek="5–1"

The following expression represents the twenty-fifth day of the month to the end of the month, and the beginning of the month to the fifth day of the month:

dayOfMonth="25–5"

It is equivalent to the following expression:

dayOfMonth="25–Last, 1–5"

### Specifying Intervals

The forward slash (/) constrains an attribute to a starting point and an interval and is used to specify every $N$ seconds, minutes, or hours within the minute, hour, or day. For an expression of the form $x/y$, $x$ represents the starting point and $y$ represents the interval. The wildcard character may be used in the $x$ position of an interval and is equivalent to setting $x$ to 0.

Intervals may be set only for second, minute, and hour attributes.

The following expression represents every 10 minutes within the hour:

minute="*/10"

It is equivalent to:

minute="0, 10, 20, 30, 40, 50"

The following expression represents every 2 hours starting at noon:

hour="12/2"
Programmatic Timers

When a programmatic timer expires (goes off), the container calls the method annotated @Timeout in the bean’s implementation class. The @Timeout method contains the business logic that handles the timed event.

The @Timeout Method

Methods annotated @Timeout in the enterprise bean class must return void and optionally take a javax.ejb.Timer object as the only parameter. They may not throw application exceptions.

```java
@Timeout
public void timeout(Timer timer) {
    System.out.println("TimerBean: timeout occurred");
}
```

Creating Programmatic Timers

To create a timer, the bean invokes one of the create methods of the TimerService interface. These methods allow single-action, interval, or calendar-based timers to be created.

For single-action or interval timers, the expiration of the timer can be expressed as either a duration or an absolute time. The duration is expressed as a the number of milliseconds before a timeout event is triggered. To specify an absolute time, create a java.util.Date object and pass it to the TimerService.createSingleActionTimer or the TimerService.createTimer method.

The following code sets a programmatic timer that will expire in 1 minute (6,000 milliseconds):

```java
long duration = 6000;
Timer timer =
    timerService.createSingleActionTimer(duration, new TimerConfig());
```

The following code sets a programmatic timer that will expire at 12:05 p.m. on May 1, 2010, specified as a java.util.Date:

```java
SimpleDateFormat formatter =
    new SimpleDateFormat("MM/dd/yyyy 'at' HH:mm");
Date date = formatter.parse("05/01/2010 at 12:05");
Timer timer = timerService.createSingleActionTimer(date, new TimerConfig());
```

For calendar-based timers, the expiration of the timer is expressed as a javax.ejb.ScheduleExpression object, passed as a parameter to the TimerService.createCalendarTimer method. The ScheduleExpression class represents calendar-based timer expressions and has methods that correspond to the attributes described in "Creating Calendar-Based Timer Expressions" on page 479.

The following code creates a programmatic timer using the ScheduleExpression helper class:
ScheduleExpression schedule = new ScheduleExpression();
schedule.dayOfWeek("Mon");
schedule.hour("12-17, 23");
Timer timer = timerService.createCalendarTimer(schedule);

For details on the method signatures, see the TimerService API documentation at http://docs.oracle.com/javaee/6/api/javax/ejb/TimerService.html.

The bean described in “The timer session Example” on page 485 creates a timer as follows:

Timer timer = timerService.createTimer(intervalDuration,
"Created new programmatic timer");

In the timer session example, createTimer is invoked in a business method, which is called by a client.

Timers are persistent by default. If the server is shut down or crashes, persistent timers are saved and will become active again when the server is restarted. If a persistent timer expires while the server is down, the container will call the @Timeout method when the server is restarted.

Nonpersistent programmatic timers are created by calling TimerConfig.setPersistent(false) and passing the TimerConfig object to one of the timer-creation methods.

The Date and long parameters of the createTimer methods represent time with the resolution of milliseconds. However, because the timer service is not intended for real-time applications, a callback to the @Timeout method might not occur with millisecond precision. The timer service is for business applications, which typically measure time in hours, days, or longer durations.

**Automatic Timers**

Automatic timers are created by the EJB container when an enterprise bean that contains methods annotated with the @Schedule or @Schedules annotations is deployed. An enterprise bean can have multiple automatic timeout methods, unlike a programmatic timer, which allows only one method annotated with the @Timeout annotation in the enterprise bean class.

Automatic timers can be configured through annotations or through the ejb-jar.xml deployment descriptor.

Adding a @Schedule annotation on an enterprise bean marks that method as a timeout method according to the calendar schedule specified in the attributes of @Schedule.

The @Schedule annotation has elements that correspond to the calendar expressions detailed in "Creating Calendar-Based Timer Expressions" on page 479 and the persistent, info, and timezone elements.

The optional persistent element takes a Boolean value and is used to specify whether the automatic timer should survive a server restart or crash. By default, all automatic timers are persistent.
The optional timezone element is used to specify that the automatic timer is associated with a particular time zone. If set, this element will evaluate all timer expressions in relation to the specified time zone, regardless of the time zone in which the EJB container is running. By default, all automatic timers set are in relation to the default time zone of the server.

The optional info element is used to set an informational description of the timer. A timer’s information can be retrieved later by using Timer.getInfo.

The following timeout method uses @Schedule to set a timer that will expire every Sunday at midnight:

```java
@Schedule(dayOfWeek="Sun", hour="0")
public void cleanupWeekData() { ... }
```

The @Schedules annotation is used to specify multiple calendar-based timer expressions for a given timeout method.

The following timeout method uses the @Schedules annotation to set multiple calendar-based timer expressions. The first expression sets a timer to expire on the last day of every month. The second expression sets a timer to expire every Friday at 11:00 p.m.

```java
@Schedules ( {
    @Schedule(dayOfMonth="Last"),
    @Schedule(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup() { ... }
```

## Canceling and Saving Timers

Timers can be cancelled by the following events.

- When a single-event timer expires, the EJB container calls the associated timeout method and then cancels the timer.
- When the bean invokes the cancel method of the Timer interface, the container cancels the timer.

If a method is invoked on a cancelled timer, the container throws the javax.ejb.NoSuchObjectLocalException.

To save a Timer object for future reference, invoke its getHandle method and store the TimerHandle object in a database. (A TimerHandle object is serializable.) To reinstantiate the Timer object, retrieve the handle from the database and invoke getTimer on the handle. A TimerHandle object cannot be passed as an argument of a method defined in a remote or web service interface. In other words, remote clients and web service clients cannot access a bean’s TimerHandle object. Local clients, however, do not have this restriction.
Getting Timer Information

In addition to defining the cancel and getHandle methods, the Timer interface defines methods for obtaining information about timers:

public long getTimeRemaining();
public java.util.Date getNextTimeout();
public java.io.Serializable getInfo();

The getInfo method returns the object that was the last parameter of the createTimer invocation. For example, in the createTimer code snippet of the preceding section, this information parameter is a String object with the value created timer.

To retrieve all of a bean’s active timers, call the getTimers method of the TimerService interface. The getTimers method returns a collection of Timer objects.

Transactions and Timers

An enterprise bean usually creates a timer within a transaction. If this transaction is rolled back, the timer creation also is rolled back. Similarly, if a bean cancels a timer within a transaction that gets rolled back, the timer cancellation is rolled back. In this case, the timer’s duration is reset as if the cancellation had never occurred.

In beans that use container-managed transactions, the @Timeout method usually has the Required or RequiresNew transaction attribute to preserve transaction integrity. With these attributes, the EJB container begins the new transaction before calling the @Timeout method. If the transaction is rolled back, the container will call the @Timeout method at least one more time.

The timersession Example

The source code for this example is in the tut-install/examples.ejb/timersession/src/java/ directory.

TimerSessionBean is a singleton session bean that shows how to set both an automatic timer and a programmatic timer. In the source code listing of TimerSessionBean that follows, the setTimer and @Timeout methods are used to set a programmatic timer. A TimerService instance is injected by the container when the bean is created. Because it’s a business method, setTimer is exposed to the local, no-interface view of TimerSessionBean and can be invoked by the client. In this example, the client invokes setTimer with an interval duration of 30,000 milliseconds. The setTimer method creates a new timer by invoking the createTimer method of TimerService. Now that the timer is set, the EJB container will invoke the programmaticTimeout method of TimerSessionBean when the timer expires, in about 30 seconds.
public void setTimer(long intervalDuration) {
    logger.info("Setting a programmatic timeout for " + intervalDuration + " milliseconds from now.");
    Timer timer = timerService.createTimer(intervalDuration, "Created new programmatic timer");
}

@Timeout
public void programmaticTimeout(Timer timer) {
    this.setLastProgrammaticTimeout(new Date());
    logger.info("Programmatic timeout occurred.");
}

TimerSessionBean also has an automatic timer and timeout method, automaticTimeout. The automatic timer is set to expire every 3 minutes and is set by using a calendar-based timer expression in the @Schedule annotation:

@Schedule(minute="*/3", hour="*")
public void automaticTimeout() {
    this.setLastAutomaticTimeout(new Date());
    logger.info("Automatic timeout occurred");
}

TimerSessionBean also has two business methods: getLastProgrammaticTimeout and getLastAutomaticTimeout. Clients call these methods to get the date and time of the last timeout for the programmatic timer and automatic timer, respectively.

Here's the source code for the TimerSessionBean class:

package timersession.ejb;
import java.util.Date;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.ejb.Schedule;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerService;

@Singleton
public class TimerSessionBean {
    @Resource
    TimerService timerService;

    private Date lastProgrammaticTimeout;
    private Date lastAutomaticTimeout;

    private Logger logger = Logger.getLogger(
        "com.sun.tutorial.javaee.ejb.timersession.TimerSessionBean");

    public void setTimer(long intervalDuration) {
        ...
logger.info("Setting a programmatic timeout for "+ intervalDuration + " milliseconds from now.");
Timer timer = timerService.createTimer(intervalDuration, "Created new programmatic timer");

@Timeout
public void programmaticTimeout(Timer timer) {
    this.setLastProgrammaticTimeout(new Date());
    logger.info("Programmatic timeout occurred.");
}

@Schedule(minute="*/3", hour="*")
public void automaticTimeout() {
    this.setLastAutomaticTimeout(new Date());
    logger.info("Automatic timeout occurred");
}

public String getLastProgrammaticTimeout() {
    if (lastProgrammaticTimeout != null) {
        return lastProgrammaticTimeout.toString();
    } else {
        return "never";
    }
}

public void setLastProgrammaticTimeout(Date lastTimeout) {
    this.lastProgrammaticTimeout = lastTimeout;
}

public String getLastAutomaticTimeout() {
    if (lastAutomaticTimeout != null) {
        return lastAutomaticTimeout.toString();
    } else {
        return "never";
    }
}

public void setLastAutomaticTimeout(Date lastAutomaticTimeout) {
    this.lastAutomaticTimeout = lastAutomaticTimeout;
}

Note – GlassFish Server has a default minimum timeout value of 1,000 milliseconds, or 1 second. If you need to set the timeout value lower than 1,000 milliseconds, change the value of the Minimum Delivery Interval setting in the Administration Console. To modify the minimum timeout value, in the Administration Console expand Configurations, then expand server-config, click EJB Container, and select the EJB Timer Service tab. Enter a new value under Minimum Delivery Interval and click Save. The lowest practical value is around 10 milliseconds, owing to virtual machine constraints.
Running the timersession Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the timersession example.

▼ To Run the timersession Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   `tut-install/examples/ejb/
3. Select the timersession folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. From the Run menu, choose Run Project.
   This builds and packages the application into `timersession.war`, located in `tut-install/examples/ejb/timersession/dist/`, deploys this WAR file to your GlassFish Server instance, and then runs the web client.

▼ To Build, Package, and Deploy the timersession Example Using Ant

1. In a terminal window, go to:
   `tut-install/examples/ejb/timersession/
2. Type the following command:
   `ant
   This runs the default task, which compiles the source files and packages the application into a WAR file located at `tut-install/examples/ejb/timersession/dist/timersession.war`.
3. To deploy the application, type the following command:
   `ant deploy`

▼ To Run the Web Client

2. Click the Set Timer button to set a programmatic timer.
3 Wait for a while and click the browser’s Refresh button.
You will see the date and time of the last programmatic and automatic timeouts.
To see the messages that are logged when a timeout occurs, open the server.log file located in domain-dir/server/logs/.

Handling Exceptions

The exceptions thrown by enterprise beans fall into two categories: system and application.

A system exception indicates a problem with the services that support an application. For example, a connection to an external resource cannot be obtained, or an injected resource cannot be found. If it encounters a system-level problem, your enterprise bean should throw a javax.ejb.EJBException. Because the EJBException is a subclass of the RuntimeException, you do not have to specify it in the throws clause of the method declaration. If a system exception is thrown, the EJB container might destroy the bean instance. Therefore, a system exception cannot be handled by the bean’s client program, but instead requires intervention by a system administrator.

An application exception signals an error in the business logic of an enterprise bean. Application exceptions are typically exceptions that you’ve coded yourself, such as the BookException thrown by the business methods of the CartBean example. When an enterprise bean throws an application exception, the container does not wrap it in another exception. The client should be able to handle any application exception it receives.

If a system exception occurs within a transaction, the EJB container rolls back the transaction. However, if an application exception is thrown within a transaction, the container does not roll back the transaction.
Message-driven beans can implement any messaging type. Most commonly, they implement the Java Message Service (JMS) technology. The example in this chapter uses JMS technology, so you should be familiar with basic JMS concepts such as queues and messages. To learn about these concepts, see Chapter 47, “Java Message Service Concepts.”

This chapter describes the source code of a simple message-driven bean example. Before proceeding, you should read the basic conceptual information in the section “What Is a Message-Driven Bean?” on page 439 as well as “Using Message-Driven Beans to Receive Messages Asynchronously” on page 853.

The following topics are addressed here:

- “Overview of the simplemessage Example” on page 491
- “The simplemessage Application Client” on page 492
- “The Message-Driven Bean Class” on page 493
- “Running the simplemessage Example” on page 495

Overview of the simplemessage Example

The simplemessage application has the following components:

- SimpleMessageClient: An application client that sends several messages to a queue
- SimpleMessageBean: A message-driven bean that asynchronously receives and processes the messages that are sent to the queue

Figure 25–1 illustrates the structure of this application. The application client sends messages to the queue, which was created administratively using the Administration Console. The JMS provider (in this case, the GlassFish Server) delivers the messages to the instances of the message-driven bean, which then processes the messages.
The source code for this application is in the `tut-install/examples/ejb/simplemessage/` directory.

The `simplemessage` Application Client

The `SimpleMessageClient` sends messages to the queue that the `SimpleMessageBean` listens to. The client starts by injecting the connection factory and queue resources:

```java
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(mappedName="jms/Queue")
private static Queue queue;
```

Next, the client creates the connection, session, and message producer:

```java
connection = connectionFactory.createConnection();
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
messageProducer = session.createProducer(queue);
```

Finally, the client sends several messages to the queue:

```java
message = session.createTextMessage();
for (int i = 0; i < NUM_MSGS; i++) {
    message.setText("This is message " + (i + 1));
    System.out.println("Sending message: " + message.getText());
    messageProducer.send(message);
}
```
The Message-Driven Bean Class

The code for the SimpleMessageBean class illustrates the requirements of a message-driven bean class:

- It must be annotated with the `@MessageDriven` annotation if it does not use a deployment descriptor.
- The class must be defined as `public`.
- The class cannot be defined as `abstract` or `final`.
- It must contain a public constructor with no arguments.
- It must not define the `finalize` method.

It is recommended, but not required, that a message-driven bean class implement the message listener interface for the message type it supports. A bean that supports the JMS API implements the `javax.jms.MessageListener` interface.

Unlike session beans and entities, message-driven beans do not have the remote or local interfaces that define client access. Client components do not locate message-driven beans and invoke methods on them. Although message-driven beans do not have business methods, they may contain helper methods that are invoked internally by the `onMessage` method.

For the GlassFish Server, the `@MessageDriven` annotation typically contains a `mappedName` element that specifies the JNDI name of the destination from which the bean will consume messages. For complex message-driven beans, there can also be an `activationConfig` element containing `@ActivationConfigProperty` annotations used by the bean.

A message-driven bean can also inject a `MessageDrivenContext` resource. Commonly you use this resource to call the `setRollbackOnly` method to handle exceptions for a bean that uses container-managed transactions.

Therefore, the first few lines of the `SimpleMessageBean` class look like this:

```java
@MessageDriven(mappedName="jms/Queue", activationConfig = {
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue")
})
public class SimpleMessageBean implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;
    ...
}
```

NetBeans IDE typically creates a message-driven bean with a default set of `@ActivationConfigProperty` settings. You can delete those you do not need, or add others. 

Table 25–1 lists commonly used properties.
The Message-Driven Bean Class

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acknowledgeMode</td>
<td>Acknowledgment mode; see &quot;Controlling Message Acknowledgment&quot; on page 843 for information</td>
</tr>
<tr>
<td>destinationType</td>
<td>Either javax.jms.Queue or javax.jms.Topic</td>
</tr>
<tr>
<td>subscriptionDurability</td>
<td>For durable subscribers, set to Durable; see &quot;Creating Durable Subscriptions&quot; on page 847 for information</td>
</tr>
<tr>
<td>clientId</td>
<td>For durable subscribers, the client ID for the connection</td>
</tr>
<tr>
<td>subscriptionName</td>
<td>For durable subscribers, the name of the subscription</td>
</tr>
<tr>
<td>messageSelector</td>
<td>A string that filters messages; see &quot;JMS Message Selectors&quot; on page 839 for information, and see &quot;An Application That Uses the JMS API with a Session Bean&quot; on page 897 for an example</td>
</tr>
<tr>
<td>addressList</td>
<td>Remote system or systems with which to communicate; see &quot;An Application Example That Consumes Messages from a Remote Server&quot; on page 910 for an example</td>
</tr>
</tbody>
</table>

The onMessage Method

When the queue receives a message, the EJB container invokes the message listener method or methods. For a bean that uses JMS, this is the onMessage method of the MessageListener interface.

A message listener method must follow these rules:

- The method must be declared as public.
- The method must not be declared as final or static.

The onMessage method is called by the bean’s container when a message has arrived for the bean to service. This method contains the business logic that handles the processing of the message. It is the message-driven bean’s responsibility to parse the message and perform the necessary business logic.

The onMessage method has a single argument: the incoming message.

The signature of the onMessage method must follow these rules:

- The return type must be void.
- The method must have a single argument of type javax.jms.Message.
In the SimpleMessageBean class, the onMessage method casts the incoming message to a TextMessage and displays the text:

```java
public void onMessage(Message inMessage) {
    TextMessage msg = null;
    try {
        if (inMessage instanceof TextMessage) {
            msg = (TextMessage) inMessage;
            logger.info("MESSAGE BEAN: Message received: "+msg.getText());
        } else {
            logger.warning("Message of wrong type: "+inMessage.getClass().getName());
        }
    } catch (JMSException e) {
        e.printStackTrace();
        mdc.setRollbackOnly();
    } catch (Throwable te) {
        te.printStackTrace();
    }
}
```

Running the simplemessage Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the simplemessage example.

Administered Objects for the simplemessage Example

This example requires the following:

- A JMS connection factory resource
- A JMS destination resource

If you have run the simple JMS examples in Chapter 47, "Java Message Service Concepts," and have not deleted the resources, you already have these resources. Otherwise, the resources will be created automatically when you deploy the application.

For more information on creating JMS resources, see "JMS Administered Objects for the Synchronous Receive Example" on page 863.
To Run the simplemessage Application Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   tut-install/examples/ejb/

3. Select the simplemessage folder.

4. Select the Open as Main Project check box and the Open Required Projects check box.

5. Click Open Project.

6. In the Projects tab, right-click the simplemessage project and choose Build.
   This task packages the application client and the message-driven bean, then creates a file named simplemessage.ear in the dist directory.

7. Right-click the project and choose Run.
   This command creates any needed resources, deploys the project, returns a JAR file named simplemessageClient.jar, and then executes it.

   The output of the application client in the Output pane looks like this (preceded by application client container output):
   Sending message: This is message 1
   Sending message: This is message 2
   Sending message: This is message 3
   To see if the bean received the messages, check <install_dir>/domains/domain1/logs/server.log.

   The output from the message-driven bean appears in the server log (domain-dir/logs/server.log), wrapped in logging information.

   MESSAGE BEAN: Message received: This is message 1
   MESSAGE BEAN: Message received: This is message 2
   MESSAGE BEAN: Message received: This is message 3

   The received messages may appear in a different order from the order in which they were sent.

To Run the simplemessage Application Using Ant

1. In a terminal window, go to:
   tut-install/examples/ejb/simplemessage/
2 To compile the source files and package the application, use the following command:

```
ant
```
This target packages the application client and the message-driven bean, then creates a file named `simplemessage.ear` in the `dist` directory.

By using resource injection and annotations, you avoid having to create deployment descriptor files for the message-driven bean and application client. You need to use deployment descriptors only if you want to override the values specified in the annotated source files.

3 To create any needed resources, deploy the application, and run the client using Ant, use the following command:

```
ant run
```
Ignore the message that states that the application is deployed at a URL.

The output in the terminal window looks like this (preceded by application client container output):

```
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
To see if the bean received the messages, check `<install_dir>/domains/domain1/logs/server.log`.
```

In the server log file, the following lines appear, wrapped in logging information:

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

The received messages may appear in a different order from the order in which they were sent.

**Removing the Administered Objects for the simplemessage Example**

After you run the example, you can use NetBeans IDE to remove the connection factory and queue, as described in “To Delete JMS Resources Using NetBeans IDE” on page 864. If you are not using NetBeans IDE, you can use the `asadmin list-jms-resources` command to list the resources, and the `asadmin delete-jms-resource` command to remove each one.
Using the Embedded Enterprise Bean Container

This chapter demonstrates how to use the embedded enterprise bean container to run enterprise bean applications in the Java SE environment, outside of a Java EE server.

The following topics are addressed here:
- “Overview of the Embedded Enterprise Bean Container” on page 499
- “Developing Embeddable Enterprise Bean Applications” on page 499
- “The standalone Example Application” on page 502

Overview of the Embedded Enterprise Bean Container

The embedded enterprise bean container is used to access enterprise bean components from client code executed in a Java SE environment. The container and the client code are executed within the same virtual machine. The embedded enterprise bean container is typically used for testing enterprise beans without having to deploy them to a server.

Most of the services present in the enterprise bean container in a Java EE server are available in the embedded enterprise bean container, including injection, container-managed transactions, and security. Enterprise bean components execute similarly in both embedded and Java EE environments, and therefore the same enterprise bean can be easily reused in both standalone and networked applications.

Developing Embeddable Enterprise Bean Applications

All embeddable enterprise bean containers support the features listed in Table 26–1.
TABLE 26–1  Required Enterprise Bean Features in the Embeddable Container

<table>
<thead>
<tr>
<th>Enterprise Bean Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local session beans</td>
<td>Local and no-interface view stateless, stateful, and singleton session beans. All method access is synchronous. Session beans must not be web service endpoints.</td>
</tr>
<tr>
<td>Transactions</td>
<td>Container-managed and bean-managed transactions.</td>
</tr>
<tr>
<td>Security</td>
<td>Declarative and programmatic security.</td>
</tr>
<tr>
<td>Interceptors</td>
<td>Class-level and method-level interceptors for session beans.</td>
</tr>
<tr>
<td>Deployment descriptor</td>
<td>The optional ejb-jar.xml deployment descriptor, with the same overriding rules for the enterprise bean container in Java EE servers.</td>
</tr>
</tbody>
</table>

Container providers are allowed to support the full set of features in enterprise beans, but applications that use the embedded container will not be portable if they use enterprise bean features not listed in Table 26–1, such as the timer service, session beans as web service endpoints, or remote business interfaces.

Running Embedded Applications

The embedded container, the enterprise bean components, and the client all are executed in the same virtual machine using the same classpath. As a result, developers can run an application that uses the embedded container just like a typical Java SE application as follows:

```java
java -classpath mySessionBean.jar:containerProviderRuntime.jar:myClient.jar
com.example.ejb.client.Main
```

In the above example, mySessionBean.jar is an EJB JAR containing a local stateless session bean, containerProviderRuntime.jar is a JAR file supplied by the enterprise bean provider that contains the needed runtime classes for the embedded container, and myClient.jar is a JAR file containing a Java SE application that calls the business methods in the session bean through the embedded container.

Creating the Enterprise Bean Container

The java.ejb.embedded.EJBContainer abstract class represents an instance of the enterprise bean container and includes factory methods for creating a container instance. The EJBContainer.createEJBContainer method is used to create and initialize an embedded container instance.

The following code snippet shows how to create an embedded container that is initialized with the container provider’s default settings:
EJBContainer ec = EJBContainer.createEJBContainer();

By default, the embedded container will search the virtual machine classpath for enterprise bean modules: directories containing a META-INF/ejb-jar.xml deployment descriptor, directories containing a class file with one of the enterprise bean component annotations (such as @Stateless), or JAR files containing an ejb-jar.xml deployment descriptor or class file with an enterprise bean annotation. Any matching entries are considered enterprise bean modules within the same application. Once all the valid enterprise bean modules have been found in the classpath, the container will begin initializing the modules. When the createEJBContainer method successfully returns, the client application can obtain references to the client view of any enterprise bean module found by the embedded container.

An alternate version of the EJBContainer.createEJBContainer method takes a Map of properties and settings for customizing the embeddable container instance:

Properties props = new Properties();
props.setProperty(...);
...
EJBContainer ec = EJBContainer.createEJBContainer(props);

Explicitly Specifying Enterprise Bean Modules to be Initialized

Developers can specify exactly which enterprise bean modules the embedded container will initialize. To explicitly specify the enterprise bean modules initialized by the embedded container, set the EJBContainer.MODULES property.

The modules may be located either in the virtual machine classpath in which the embedded container and client code run, or alternately outside the virtual machine classpath.

To specify modules in the virtual machine classpath, set EJBContainer.MODULES to a String to specify a single module name, or a String array containing the module names. The embedded container searches the virtual machine classpath for enterprise bean modules matching the specified names.

Properties props = new Properties();
props.setProperty(EJBContainer.MODULES, "mySessionBean");
EJBContainer ec = EJBContainer.createEJBContainer(props);

To specify enterprise bean modules outside the virtual machine classpath, set EJBContainer.MODULES to a java.io.File object or an array of File objects. Each File object refers to an EJB JAR file, or a directory containing an expanded EJB JAR.

Properties props = new Properties();
File ejbJarFile = new File(...);
props.setProperty(EJBContainer.MODULES, ejbJarFile);
EJBContainer ec = EJBContainer.createEJBContainer(props);
Looking Up Session Bean References

To look up session bean references in an application using the embedded container, use an instance of EJBContainer to retrieve a javax.naming.Context object. Call the EJBContainer.getContext method to retrieve the Context object.

```java
EJBContainer ec = EJBContainer.createEJBContainer();
Context ctx = ec.getContext();
```

References to session beans can then be obtained using the portable JNDI syntax detailed in "Portable JNDI Syntax" on page 442. For example, to obtain a reference to MySessionBean, a local session bean with a no-interface view, use the following code:

```java
MySessionBean msb = (MySessionBean)
    ctx.lookup("java:global/mySessionBean/MySessionBean");
```

Shutting Down the Enterprise Bean Container

From the client, call the close method of the instance of EJBContainer to shut down the embedded container:

```java
EJBContainer ec = EJBContainer.createEJBContainer();
...
ec.close();
```

While clients are not required to shut down EJBContainer instances, doing so frees resources consumed by the embedded container. This is particularly important when the virtual machine under which the client application is running has a longer lifetime than the client application.

The standalone Example Application

The standalone example application demonstrates how to create an instance of the embedded enterprise bean container in a JUnit test class and call a session bean business method. Testing the business methods of an enterprise bean in a unit test allows developers to exercise the business logic of an application separately from the other application layers, such as the presentation layer, and without having to deploy the application to a Java EE server.

The standalone example has two main components: StandaloneBean, a stateless session bean, and StandaloneBeanTest, a JUnit test class that acts as a client to StandaloneBean using the embedded container.

StandaloneBean is a simple session bean exposing a local, no-interface view with a single business method, returnMessage, which returns "Greetings!" as a String.

```java
@Stateless
public class StandaloneBean {
```
private static final String message = "Greetings!";

public String returnMessage() {
    return message;
}

StandaloneBeanTest calls StandaloneBean{returnMessage} and tests that the returned message is correct. First, an embedded container instance and initial context are created within the setUp method, which is annotated with org.junit.Before to indicate that the method should be executed before the test methods.

@Before
public void setUp() {
    ec = EJBContainer.createEJBContainer();
    ctx = ec.getContext();
}

The testReturnMessage method, annotated with org.junit.Test to indicate that the method includes a unit test, obtains a reference to StandaloneBean through the Context instance, and calls StandaloneBean{returnMessage}. The result is compared with the expected result using a JUnit assertion, assertEquals. If the string returned from StandaloneBean{returnMessage} is equal to "Greetings!" the test passes.

@Test
public void testReturnMessage() throws Exception {
    logger.info("Testing standalone.ejb.StandalonBean.returnMessage()");
    StandaloneBean instance = (StandaloneBean)
    ctx.lookup("java:global/classes/StandaloneBean");
    String expResult = "Greetings!";
    String result = instance.returnMessage();
    assertEquals(expResult, result);
}

Finally, the tearDown method, annotated with org.junit.After to indicate that the method should be executed after all the unit tests have run, closes the embedded container instance.

@After
public void tearDown() {
    if (ec != null) {
        ec.close();
    }
}

To Run the standalone Example Application

Before You Begin
You must run the standalone example application within NetBeans IDE.

1 From the File menu, choose Open Project.
The standalone Example Application

2 In the Open Project dialog, navigate to:
   tut-install/examples.ejb/

3 Select the standalone folder and click Open Project.

4 In the Projects tab, right-click standalone and select Test.
   This will execute the JUnit test class StandaloneBeanTest. The Output tab shows the progress
   of the test and the output log.
Using Asynchronous Method Invocation in Session Beans

This chapter discusses how to implement asynchronous business methods in session beans and call them from enterprise bean clients.

The following topics are addressed here:

■ “Asynchronous Method Invocation” on page 505
■ “The async Example Application” on page 508

Asynchronous Method Invocation

Session beans can implement asynchronous methods, business methods where control is returned to the client by the enterprise bean container before the method is invoked on the session bean instance. Clients may then use the Java SE concurrency API to retrieve the result, cancel the invocation, and check for exceptions. Asynchronous methods are typically used for long-running operations, for processor-intensive tasks, for background tasks, to increase application throughput, or to improve application response time if the method invocation result isn’t required immediately.

When a session bean client invokes a typical non-asynchronous business method, control is not returned to the client until the method has completed. Clients calling asynchronous methods, however, immediately have control returned to them by the enterprise bean container. This allows the client to perform other tasks while the method invocation completes. If the method returns a result, the result is an implementation of the java.util.concurrent.Future<V> interface, where “V” is the result value type. The Future<V> interface defines methods the client may use to check whether the computation is completed, wait for the invocation to complete, retrieve the final result, and cancel the invocation.
Creating an Asynchronous Business Method

Annotate a business method with `javax.ejb.Asynchronous` to mark that method as an asynchronous method, or apply `@Asynchronous` at the class level to mark all the business methods of the session bean as asynchronous methods. Session bean methods that expose web services can’t be asynchronous.

Asynchronous methods must return either `void` or an implementation of the `Future<V>` interface. Asynchronous methods that return `void` can’t declare application exceptions, but if they return `Future<V>`, they may declare application exceptions. For example:

```java
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
}
```

This method will attempt to process the payment of an order, and return the status as a `String`. Even if the payment processor takes a long time, the client can continue working, and display the result when the processing finally completes.

The `javax.ejbAsyncResult<V>` class is a concrete implementation of the `Future<V>` interface provided as a helper class for returning asynchronous results. `AsyncResult` has a constructor with the result as a parameter, making it easy to create `Future<V>` implementations. For example, the `processPayment` method would use `AsyncResult` to return the status as a `String`:

```java
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    String status = ...;
    return new AsyncResult<String>(status);
}
```

The result is returned to the enterprise bean container, not directly to the client, and the enterprise bean container makes the result available to the client. The session bean can check whether the client requested that the invocation be cancelled by calling the `javax.ejb.SessionContext.wasCancelled` method. For example:

```java
@Asynchronous
public Future<String> processPayment(Order order) throws PaymentException {
    ...
    if (SessionContext.wasCancelled()) {
        // clean up
    } else {
        // process the payment
    }
    ...
}
```
Calling Asynchronous Methods from Enterprise Bean Clients

Session bean clients call asynchronous methods just like non-asynchronous business methods. If the asynchronous method returns a result, the client receives a `Future<V>` instance as soon as the method is invoked. This instance can be used to retrieve the final result, cancel the invocation, check whether the invocation has completed, check whether any exceptions were thrown during processing, and check whether the invocation was cancelled.

Retrieving the Final Result from an Asynchronous Method Invocation

The client may retrieve the result using one of the `Future<V>.get` methods. If processing hasn’t been completed by the session bean handling the invocation, calling one of the get methods will result in the client halting execution until the invocation completes. Use the `Future<V>.isDone` method to determine whether processing has completed before calling one of the get methods.

The `get()` method returns the result as the type specified in the type value of the `Future<V>` instance. For example, calling `Future<String>.get()` will return a `String` object. If the method invocation was cancelled, calls to `get()` result in a `java.util.concurrent.CancellationException` being thrown. If the invocation resulted in an exception during processing by the session bean, calls to `get()` result in a `java.util.concurrent.ExecutionException` being thrown. The cause of the `ExecutionException` may be retrieved by calling the `ExecutionException.getCause` method.

The `get(long timeout, java.util.concurrent.TimeUnit unit)` method is similar to the `get()` method, but allows the client to set a timeout value. If the timeout value is exceeded, a `java.util.concurrent.TimeoutException` is thrown. See the Javadoc for the `TimeUnit` class for the available units of time to specify the timeout value.

Cancelling an Asynchronous Method Invocation

Call the `cancel(boolean mayInterruptIfRunning)` method on the `Future<V>` instance to attempt to cancel the method invocation. The cancel method returns `true` if the cancellation was successful, and `false` if the method invocation cannot be cancelled.

When the invocation cannot be cancelled, the `mayInterruptIfRunning` parameter is used to alert the session bean instance on which the method invocation is running that the client attempted to cancel the invocation. If `mayInterruptIfRunning` is set to `true`, calls to `SessionContext.wasCancelled` by the session bean instance will return `true`. If `mayInterruptIfRunning` is set to `false`, calls to `SessionContext.wasCancelled` by the session bean instance will return `false`.

The `Future<V>.isCancelled` method is used to check whether the method invocation was cancelled before the asynchronous method invocation completed by calling `Future<V>.cancel`. The `isCancelled` method returns `true` if the invocation was cancelled.
Checking the Status of an Asynchronous Method Invocation

The `Future<V>.isDone` method returns true if the session bean instance completed processing the method invocation. The `isDone` method returns true if the asynchronous method invocation completed normally, was cancelled, or resulted in an exception. That is, `isDone` indicates only whether the session bean has completed processing the invocation.

The async Example Application

The async example demonstrates how to define an asynchronous business method on a session bean and call it from a web client. The `MailerBean` stateless session bean defines an asynchronous method, `sendMessage`, which uses the JavaMail API to send an email to a specified email address.

```
Note – This example needs to be configured for your environment before it runs correctly, and requires access to an SMTPS server.
```

Architecture of the async Example Application

The async application consists of a single stateless session bean, `MailerBean`, and a JavaServer Faces web application front end that uses Facelets tags in XHTML files to display a form for users to enter the email address for the recipient of an email. The status of the email is updated when the email is finally sent.

The `MailerBean` session bean injects a JavaMail resource used to send an email message to an address specified by the user. The message is created, modified, and sent using the JavaMail API. The injected JavaMail resource is configured through the GlassFish Server Administration Console, or through a resource configuration file packaged with the application. The resource configuration can be modified at runtime by the GlassFish Server administrator to use a different mail server or transport protocol.

```
@Asynchronous
public Future<String> sendMessage(String email) {
    String status;
    try {
        Message message = new MimeMessage(session);
        message.setFrom();
        message.setRecipients(Message.RecipientType.TO,
                                InternetAddress.parse(email, false));
        message.setSubject("Test message from async example");
        message.setHeader("X-Mailer", "JavaMail");
        DateFormat dateFormatter = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.SHORT);
        Date timeStamp = new Date();
        String messageBody = "This is a test message from the async example 
```
of the Java EE Tutorial. It was sent on "
+ dateFormatter.format(timeStamp)
+ ";
message.setText(messageBody);
message.setSentDate(timeStamp);
Transport.send(message);
status = "Sent";
logger.log(Level.INFO, "Mail sent to {0}", email);
)
catch (MessagingException ex) {
logger.severe("Error in sending message.");
status = "Encountered an error";
logger.severe(ex.getMessage() + ex.getNextException().getMessage());
logger.severe(ex.getCause().getMessage());
}
return new AsyncResult<String>(status);
}
The web client consists of a Facelets template, template.xhtml, two Facelets clients, index.xhtml and response.xhtml, and a JavaServer Faces managed bean, MailerManagedBean. The index.xhtml file contains a form for the target email address. When the user submits the form, the MailerManagedBean.send method is called. This method uses an injected instance of the MailerBean session bean to call MailerBean.sendMessage. The result is sent to the response.xhtml Facelets view.

Running the async Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the async example. First, however, you must configure the keystore and truststore.

▼ To Configure the Keystore and Truststore in GlassFish Server

The GlassFish Server domain needs to be configured with the server’s master password to access the keystore and truststore used to initiate secure communications using the SMTPS transport protocol.

1 Open the GlassFish Server Administration Console in a web browser at http://localhost:4848.

2 Expand Configurations, then expand server-config, then click JVM Settings.

3 Click JVM Options, then click Add JVM Option and enter
-Djavax.net.ssl.keyStorePassword=master-password, replacing master-password with the keystore master password. The default master password is changeit.

4 Click Add JVM Option and enter -Djavax.net.ssl.trustStorePassword=master-password, replacing master-password with the truststore master password. The default master password is changeit.

5 Click Save, then restart GlassFish Server.
Before You Begin

To Run the async Example Using NetBeans IDE

Before running this example, you must configure your GlassFish Server instance to access the keystore and truststore used by GlassFish Server to create a secure connection to the target SMTPS server.

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   tut-install/examples/ejb/

3. Select the async folder and click Open Project.

4. Under async in the project pane, expand the Server Resources node and double-click glassfish-resources.xml.

5. Enter the configuration settings for your SMTPS server in glassfish-resources.xml.
   The SMTPS server host name is set in the host attribute; the email address from which you want the message sent is set in the from attribute; and the SMTPS user name is set in the user attribute. Set the mail-smtps-password property value to the password for the SMTPS server user. The following code snippet shows an example resource configuration. Lines in bold need to be modified.

   ```xml
   <resources>
     <mail-resource debug="false"
         enabled="true"
         from="user@example.com"
         host="smtp.example.com"
         jndi-name="mail/myExampleSession"
         object-type="user"
         store-protocol="imap"
         store-protocol-class="com.sun.mail.imap.IMAPStore"
         transport-protocol="smtps"
         transport-protocol-class="com.sun.mail.smtp.SMTPSSLTransport"
         user="user@example.com">
       <description/>
       <property name="mail-smtps-auth" value="true"/>
       <property name="mail-smtps-password" value="mypassword"/>
     </mail-resource>
   </resources>
   ```

6. Right-click async in the project pane and select Run.
   This will compile, assemble, and deploy the application, and start a web browser at the following URL: http://localhost:8080/async.

7. In the web browser window, enter the email to which you want the test message sent and click Send email.
   If your configuration settings are correct, a test email will be sent, and the status message will read Sent in the web client. The test message should appear momentarily in the inbox of the recipient.
If an error occurs, the status will read Encountered an error. Check the server.log file for your domain to find the cause of the error.

**To Run the async Example Using Ant**

1. In a terminal window, navigate to `tut-install/examples.ejb/async/`.

2. In a text editor, open `setup/glassfish-resources.xml` and enter the configuration settings for your SMTPS server.
   
   The SMTPS server host name is set in the `host` attribute, email address from which you want the message sent is the `from` attribute, the SMTPS user name is the `user` attribute. Set the `mail-smtps-password` property value to the password for the SMTPS server user. The following code snippet shows an example resource configuration. Lines in bold need to be modified.

   ```xml
   <resources>
   <mail-resource debug="false"
      enabled="true"
      from="user@example.com"
      host="smtp.example.com"
      jndi-name="mail/myExampleSession"
      object-type="user"
      store-protocol="imap"
      store-protocol-class="com.sun.mail.imap.IMAPStore"
      transport-protocol="smtps"
      transport-protocol-class="com.sun.mail.smtp.SMTPSSLTransport"
      user="user@example.com">
      <description/>
      <property name="mail-smtps-auth" value="true"/>
      <property name="mail-smtps-password" value="mypassword"/>
   </mail-resource>
   </resources>
   
3. Enter the following command:

   ```
   ant all
   ```

   This will compile, assemble, and deploy the application, and start a web browser at the following URL: `http://localhost:8080/async`.

   **Note** - If your build system isn’t configured to automatically open a web browser, open the above URL in a browser window.

4. In the web browser window, enter the email to which you want the test message sent and click Send email.
   
   If your configuration settings are correct, a test email will be sent, and the status message will read Sent in the web client. The test message should appear momentarily in the inbox of the recipient.

   If an error occurs, the status will read Encountered an error. Check the server.log file for your domain to find the cause of the error.
PART V

Contexts and Dependency Injection for the Java EE Platform

Part V explores Contexts and Dependency Injection for the Java EE Platform. This part contains the following chapters:

- Chapter 28, “Introduction to Contexts and Dependency Injection for the Java EE Platform”
- Chapter 29, “Running the Basic Contexts and Dependency Injection Examples”
- Chapter 30, “Contexts and Dependency Injection for the Java EE Platform: Advanced Topics”
- Chapter 31, “Running the Advanced Contexts and Dependency Injection Examples”
Introduction to Contexts and Dependency Injection for the Java EE Platform

Contexts and Dependency Injection (CDI) for the Java EE platform is one of several Java EE 6 features that help to knit together the web tier and the transactional tier of the Java EE platform. CDI is a set of services that, used together, make it easy for developers to use enterprise beans along with JavaServer Faces technology in web applications. Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate various kinds of components in a loosely coupled but typesafe way.

CDI is specified by JSR 299, formerly known as Web Beans. Related specifications that CDI uses include the following:

- JSR 330, Dependency Injection for Java
- The Managed Beans specification, which is an offshoot of the Java EE 6 platform specification (JSR 316)

The following topics are addressed here:

- "Overview of CDI" on page 516
- “About Beans” on page 517
- “About CDI Managed Beans” on page 517
- “Beans as Injectable Objects” on page 518
- “Using Qualifiers” on page 519
- “Injecting Beans” on page 520
- “Using Scopes” on page 520
- “Overriding the Scope of a Bean at the Point of Injection” on page 522
- “Giving Beans EL Names” on page 522
- “Adding Setter and Getter Methods” on page 523
- “Using a Managed Bean in a Facelets Page” on page 524
- “Injecting Objects by Using Producer Methods” on page 524
- “Configuring a CDI Application” on page 525
- “Using the @PostConstruct and @PreDestroy Annotations With CDI Managed Bean Classes” on page 525
- “Further Information about CDI” on page 526
Overview of CDI

The most fundamental services provided by CDI are as follows:

- **Contexts**: The ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts

- **Dependency injection**: The ability to inject components into an application in a typesafe way, including the ability to choose at deployment time which implementation of a particular interface to inject

In addition, CDI provides the following services:

- Integration with the Expression Language (EL), which allows any component to be used directly within a JavaServer Faces page or a JavaServer Pages page
- The ability to decorate injected components
- The ability to associate interceptors with components using typesafe interceptor bindings
- An event-notification model
- A web conversation scope in addition to the three standard scopes (request, session, and application) defined by the Java Servlet specification
- A complete Service Provider Interface (SPI) that allows third-party frameworks to integrate cleanly in the Java EE 6 environment

A major theme of CDI is loose coupling. CDI does the following:

- Decouples the server and the client by means of well-defined types and qualifiers, so that the server implementation may vary
- Decouples the lifecycles of collaborating components by doing the following:
  - Making components contextual, with automatic lifecycle management
  - Allowing stateful components to interact like services, purely by message passing
  - Completely decouples message producers from consumers, by means of events
  - Decouples orthogonal concerns by means of Java EE interceptors

Along with loose coupling, CDI provides strong typing by

- Eliminating lookup using string-based names for wiring and correlations, so that the compiler will detect typing errors
- Allowing the use of declarative Java annotations to specify everything, largely eliminating the need for XML deployment descriptors, and making it easy to provide tools that introspect the code and understand the dependency structure at development time
About Beans

CDI redefines the concept of a bean beyond its use in other Java technologies, such as the JavaBeans and Enterprise JavaBeans (EJB) technologies. In CDI, a bean is a source of contextual objects that define application state and/or logic. A Java EE component is a bean if the lifecycle of its instances may be managed by the container according to the lifecycle context model defined in the CDI specification.

More specifically, a bean has the following attributes:

- A (nonempty) set of bean types
- A (nonempty) set of qualifiers (see “Using Qualifiers” on page 519)
- A scope (see “Using Scopes” on page 520)
- Optionally, a bean EL name (see “Giving Beans EL Names” on page 522)
- A set of interceptor bindings
- A bean implementation

A bean type defines a client-visible type of the bean. Almost any Java type may be a bean type of a bean.

- A bean type may be an interface, a concrete class, or an abstract class and may be declared final or have final methods.
- A bean type may be a parameterized type with type parameters and type variables.
- A bean type may be an array type. Two array types are considered identical only if the element type is identical.
- A bean type may be a primitive type. Primitive types are considered to be identical to their corresponding wrapper types in java.lang.
- A bean type may be a raw type.

About CDI Managed Beans

A managed bean is implemented by a Java class, which is called its bean class. A top-level Java class is a managed bean if it is defined to be a managed bean by any other Java EE technology specification, such as the JavaServer Faces technology specification, or if it meets all the following conditions:

- It is not a nonstatic inner class.
- It is a concrete class or is annotated @Decorator.
- It is not annotated with an EJB component-defining annotation or declared as an EJB bean class in ejb-jar.xml.
It has an appropriate constructor. That is, one of the following is the case:

- The class has a constructor with no parameters.
- The class declares a constructor annotated @Inject.

No special declaration, such as an annotation, is required to define a managed bean.

### Beans as Injectable Objects

The concept of injection has been part of Java technology for some time. Since the Java EE 5 platform was introduced, annotations have made it possible to inject resources and some other kinds of objects into container-managed objects. CDI makes it possible to inject more kinds of objects and to inject them into objects that are not container-managed.

The following kinds of objects can be injected:

- (Almost) any Java class
- Session beans
- Java EE resources: data sources, Java Message Service topics, queues, connection factories, and the like
- Persistence contexts (JPA EntityManager objects)
- Producer fields
- Objects returned by producer methods
- Web service references
- Remote enterprise bean references

For example, suppose that you create a simple Java class with a method that returns a string:

```java
package greetings;

public class Greeting {
    public String greet(String name) {
        return "Hello," + name + ".";
    }
}
```

This class becomes a bean that you can then inject into another class. This bean is not exposed to the EL in this form. "Giving Beans EL Names" on page 522 explains how you can make a bean accessible to the EL.
Using Qualifiers

You can use qualifiers to provide various implementations of a particular bean type. A qualifier is an annotation that you apply to a bean. A qualifier type is a Java annotation defined as @Target({METHOD, FIELD, PARAMETER, TYPE}) and @Retention(RUNTIME).

For example, you could declare an @Informal qualifier type and apply it to another class that extends the Greeting class. To declare this qualifier type, you would use the following code:

```java
package greetings;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}
```

You can then define a bean class that extends the Greeting class and uses this qualifier:

```java
package greetings;

@Informal
public class InformalGreeting extends Greeting {
    public String greet(String name) {
        return "Hi, " + name + "!";
    }
}
```

Both implementations of the bean can now be used in the application.

If you define a bean with no qualifier, the bean automatically has the qualifier @Default. The unannotated Greeting class could be declared as follows:

```java
package greetings;

import javax.enterprise.inject.Default;

@Default
public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```
Injecting Beans

In order to use the beans you create, you inject them into yet another bean that can then be used by an application, such as a JavaServer Faces application. For example, you might create a bean called Printer into which you would inject one of the Greeting beans:

```java
import javax.inject.Inject;
public class Printer {
    @Inject Greeting greeting;
    ...

    This code injects the @Default Greeting implementation into the bean. The following code injects the @Informal implementation:

    import javax.inject.Inject;
    public class Printer {
        @Inject @Informal Greeting greeting;
        ...
```

More is needed for the complete picture of this bean. Its use of scope needs to be understood. In addition, for a JavaServer Faces application, the bean needs to be accessible through the EL.

Using Scopes

For a web application to use a bean that injects another bean class, the bean needs to be able to hold state over the duration of the user’s interaction with the application. The way to define this state is to give the bean a scope. You can give an object any of the scopes described in Table 28–1, depending on how you are using it.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Annotation</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request</td>
<td>@RequestScoped</td>
<td>A user’s interaction with a web application in a single HTTP request.</td>
</tr>
<tr>
<td>Session</td>
<td>@SessionScoped</td>
<td>A user’s interaction with a web application across multiple HTTP requests.</td>
</tr>
<tr>
<td>Application</td>
<td>@ApplicationScoped</td>
<td>Shared state across all users’ interactions with a web application.</td>
</tr>
<tr>
<td>Dependent</td>
<td>@Dependent</td>
<td>The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).</td>
</tr>
</tbody>
</table>
### Using Scopes

The first three scopes are defined by both JSR 299 and the JavaServer Faces API. The last two are defined by JSR 299.

All predefined scopes except `@Dependent` are contextual scopes. CDI places beans of contextual scope in the context whose lifecycle is defined by the Java EE specifications. For example, a session context and its beans exist during the lifetime of an HTTP session. Injected references to the beans are contextually aware. The references always apply to the bean that is associated with the context for the thread that is making the reference. The CDI container ensures that the objects are created and injected at the correct time as determined by the scope that is specified for these objects.

You can also define and implement custom scopes, but that is an advanced topic. Custom scopes are likely to be used by those who implement and extend the CDI specification.

A scope gives an object a well-defined lifecycle context. A scoped object can be automatically created when it is needed and automatically destroyed when the context in which it was created ends. Moreover, its state is automatically shared by any clients that execute in the same context.

Java EE components, such as servlets and enterprise beans, and JavaBeans components do not by definition have a well-defined scope. These components are one of the following:

- **Singletons**, such as Enterprise JavaBeans singleton beans, whose state is shared among all clients
- **Stateless objects**, such as servlets and stateless session beans, which do not contain client-visible state
- **Objects that must be explicitly created and destroyed by their client**, such as JavaBeans components and stateful session beans, whose state is shared by explicit reference passing between clients

If, however, you create a Java EE component that is a managed bean, it becomes a scoped object, which exists in a well-defined lifecycle context.

### TABLE 28–1 Scopes (Continued)

<table>
<thead>
<tr>
<th>Scope</th>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversation</td>
<td><code>@ConversationScoped</code></td>
<td>A user’s interaction with a JavaServer Faces application, within explicit developer-controlled boundaries that extend the scope across multiple invocations of the JavaServer Faces lifecycle. All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.</td>
</tr>
</tbody>
</table>

---

Chapter 28 • Introduction to Contexts and Dependency Injection for the Java EE Platform 521
The web application for the Printer bean will use a simple request and response mechanism, so the managed bean can be annotated as follows:

```java
import javax.inject.Inject;
import javax.enterprise.context.RequestScoped;

@RequestScoped
public class Printer {
    @Inject @Informal Greeting greeting;
    ...
}
```

Beans that use session, application, or conversation scope must be serializable, but beans that use request scope do not have to be serializable.

### Overriding the Scope of a Bean at the Point of Injection

Overriding the scope of a bean at the point of injection enables an application to request a new instance of the bean with the default scope `@Dependent`. The `@Dependent` scope specifies that the bean's lifecycle is the lifecycle of the object into which the bean is injected. The CDI container provides no other lifecycle management for the instance.

**Note** – The effects of overriding the scope of a bean may be unpredictable or undesirable, especially if the overridden scope is `@Request` or `@Session`.

To override the scope of a bean at the point of injection, use the `javax.enterprise.inject.New` annotation instead of the `@Inject` annotation. For more information on the `@Inject` annotation, see "Injecting Beans" on page 520.

### Giving Beans EL Names

To make a bean accessible through the EL, use the `@Named` built-in qualifier:

```java
import javax.inject.Inject;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class Printer {
    @Inject @Informal Greeting greeting;
    ...
}
```

The `@Named` qualifier allows you to access the bean by using the bean name, with the first letter in lowercase. For example, a Facelets page would refer to the bean as `printer`. 
You can specify an argument to the @Named qualifier to use a nondefault name:

@Named("MyPrinter")

With this annotation, the Facelets page would refer to the bean as MyPrinter.

Adding Setter and Getter Methods

To make the state of the managed bean accessible, you need to add setter and getter methods for that state. The createSalutation method calls the bean’s greet method, and the getSalutation method retrieves the result.

Once the setter and getter methods have been added, the bean is complete. The final code looks like this:

```java
package greetings;

import javax.inject.Inject;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class Printer {
    @Inject @Informal Greeting greeting;

    private String name;
    private String salutation;

    public void createSalutation() {
        this.salutation = greeting.greet(name);
    }

    public String getSalutation() {
        return salutation;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```
Using a Managed Bean in a Facelets Page

To use the managed bean in a Facelets page, you typically create a form that uses user interface elements to call its methods and display their results. This example provides a button that asks the user to type a name, retrieves the salutation, and then displays the text in a paragraph below the button:

```html
<h:form id="greetme">
  <p>
    <h:outputLabel value="Enter your name: " for="name"/>
    <h:inputText id="name" value="#{printer.name}"/>
  </p>
  <p>
    <h:commandButton value="Say Hello"
      action="#{printer.createSalutation}"/>
  </p>
  <p>
    <h:outputText value="#{printer.salutation}"/>
  </p>
</h:form>
```

Injecting Objects by Using Producer Methods

Producer methods provide a way to inject objects that are not beans, objects whose values may vary at runtime, and objects that require custom initialization. For example, if you want to initialize a numeric value defined by a qualifier named @MaxNumber, you can define the value in a managed bean and then define a producer method, getMaxNumber, for it:

```java
private int maxNumber = 100;
...
@Produces @MaxNumber int getMaxNumber() {
    return maxNumber;
}
```

When you inject the object in another managed bean, the container automatically invokes the producer method, initializing the value to 100:

```java
@Inject @MaxNumber private int maxNumber;
```

If the value can vary at runtime, the process is slightly different. For example, the following code defines a producer method that generates a random number defined by a qualifier called @Random:

```java
private java.util.Random random =
  new java.util.Random( System.currentTimeMillis() );
java.util.Random getRandom() {
    return random;
}
@Produces @Random int next() {
    return getRandom().nextInt(maxNumber);
}
```

When you inject this object in another managed bean, you declare a contextual instance of the object:
You then call the `get` method of the `Instance`:

```java
this.number = randomInt.get();
```

### Configuring a CDI Application

An application that uses CDI must have a file named `beans.xml`. The file can be completely empty (it has content only in certain limited situations), but it must be present. For a web application, the `beans.xml` file must be in the `WEB-INF` directory. For EJB modules or JAR files, the `beans.xml` file must be in the `META-INF` directory.

### Using the `@PostConstruct` and `@PreDestroy` Annotations With CDI Managed Bean Classes

CDI managed bean classes and their superclasses support the annotations for initializing and for preparing for the destruction of a bean. These annotations are defined in the [JSR 250: Common Annotations for the Java platform](http://jcp.org/en/jsr/detail?id=250).

#### To Initialize a Managed Bean Using the `@PostConstruct` Annotation

Initializing a managed bean specifies the lifecycle callback method that the CDI framework should call after dependency injection but before the class is put into service.

1. In the managed bean class or any of its superclasses, define a method that performs the initialization that you require.

2. Annotate the declaration of the method with the `javax.annotation.PostConstruct` annotation.

   When the managed bean is injected into a component, CDI calls the method after all injection has occurred and after all initializers have been called.

   **Note** – As mandated in JSR 250, if the annotated method is declared in a superclass, the method is called unless a subclass of the declaring class overrides the method.
The UserNumberBean managed bean in "The guessnumber CDI Example" on page 531 uses @PostConstruct to annotate a method that resets all bean fields:

```java
@PostConstruct
public void reset () {
    this.minimum = 0;
    this.userNumber = 0;
    this.remainingGuesses = 0;
    this.maximum = maxNumber;
    this.number = randomInt.get();
}
```

▼ To Prepare for the Destruction of a Managed Bean Using the @PreDestroy Annotation

Preparing for the destruction of a managed bean specifies the lifecycle callback method that signals that an application component is about to be destroyed by the container.

1 In the managed bean class or any of its superclasses, prepare for the destruction of the managed bean.
   In this method, perform any cleanup that is required before the bean is destroyed, such as releasing a resource that the bean has been holding.

2 Annotate the declaration of the method with the javax.annotation.PreDestroy annotation. CDI calls this method before starting to destroy the bean.

Further Information about CDI

For more information about CDI for the Java EE platform, see

- Contexts and Dependency Injection for the Java EE platform specification:
- An introduction to Contexts and Dependency Injection for the Java EE platform:
- Dependency Injection for Java specification:
- Managed Beans specification, which is part of the Java Platform, Enterprise Edition 6 (Java EE 6) Specification:
This chapter describes in detail how to build and run simple examples that use CDI. The examples are in the `tut-install/examples/cdi/` directory:

To build and run the examples, you will do the following:
1. Use NetBeans IDE or the Ant tool to compile and package the example.
2. Use NetBeans IDE or the Ant tool to deploy the example.
3. Run the example in a web browser.

Each example has a `build.xml` file that refers to files in the `tut-install/examples/bp-project/` directory.

See Chapter 2, “Using the Tutorial Examples,” for basic information on installing, building, and running the examples.

The following topics are addressed here:
- “The simplegreeting CDI Example” on page 527
- “The guessnumber CDI Example” on page 531

**The simplegreeting CDI Example**

The simplegreeting example illustrates some of the most basic features of CDI: scopes, qualifiers, bean injection, and accessing a managed bean in a JavaServer Faces application. When you run the example, you click a button that presents either a formal or an informal greeting, depending on how you edited one of the classes. The example includes four source files, a Facelets page and template, and configuration files.
The simplegreeting CDI Example

The simplegreeting Source Files
The four source files for the simplegreeting example are
■

The default Greeting class, shown in “Beans as Injectable Objects” on page 518

■

The @Informal qualifier interface definition and the InformalGreeting class that
implements the interface, both shown in “Using Qualifiers” on page 519

■

The Printer managed bean class, which injects one of the two interfaces, shown in full in
“Adding Setter and Getter Methods” on page 523

The source files are located in the
tut-install/examples/cdi/simplegreeting/src/java/greetings/ directory.

The Facelets Template and Page
To use the managed bean in a simple Facelets application, you can use a very simple template
file and index.xhtml page. The template page, template.xhtml, looks like this:
<?xml version=’1.0’ encoding=’UTF-8’ ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<h:outputStylesheet library="css" name="default.css"/>
<title>
<ui:insert name="title">Default Title</ui:insert>
</title>
</h:head>
<body>
<div id="container">
<div id="header">
<h2><ui:insert name="head">Head</ui:insert></h2>
</div>
<div id="space">
<p></p>
</div>
<div id="content">
<ui:insert name="content"/>
</div>
</div>
</body>
</html>

To create the Facelets page, you can redefine the title and head, then add a small form to the
content:
528

The Java EE 6 Tutorial • January 2013


The form asks the user to type a name. The button is labeled Say Hello, and the action defined for it is to call the createSalutation method of the Printer managed bean. This method in turn calls the greet method of the defined Greeting class.

The output text for the form is the value of the greeting returned by the setter method. Depending on whether the default or the @Informal version of the greeting is injected, this is one of the following, where name is the name typed by the user:

Hello, name.

Hi, name!

The Facelets page and template are located in the tut-install/examples/cdi/simplegreeting/web/ directory.

The simple CSS file that is used by the Facelets page is in the following location:

tut-install/examples/cdi/simplegreeting/web/resources/css/default.css

**Configuration Files**

You must create an empty beans.xml file to indicate to GlassFish Server that your application is a CDI application. This file can have content in some situations, but not in simple applications like this one.

Your application also needs the basic web application deployment descriptors web.xml and glassfish-web.xml. These configuration files are located in the tut-install/examples/cdi/simplegreeting/web/WEB-INF/ directory.
Running the simplegreeting Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the simplegreeting application.

▼ To Build, Package, and Deploy the simplegreeting Example Using NetBeans IDE

This procedure builds the application into the `tut-install/examples/cdi/simplegreeting/build/web/` directory. The contents of this directory are deployed to the GlassFish Server.

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   `tut-install/examples/cdi/`
3. Select the simplegreeting folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. (Optional) To modify the `Printer.java` file, perform these steps:
   a. Expand the Source Packages node.
   b. Expand the `greetings` node.
   c. Double-click the `Printer.java` file.
   d. In the edit pane, comment out the `@Informal` annotation:
      ```java
      @Inject
      //@Informal
      Greeting greeting;
      ```
   e. Save the file.
7. In the Projects tab, right-click the simplegreeting project and select Deploy.

▼ To Build, Package, and Deploy the simplegreeting Example Using Ant

1. In a terminal window, go to:
   `tut-install/examples/cdi/simplegreeting/`
2 Type the following command:
   ant
   This command calls the default target, which builds and packages the application into a WAR file, simplegreeting.war, located in the dist directory.

3 Type the following command:
   ant deploy
   Typing this command deploys simplegreeting.war to the GlassFish Server.

▼ To Run the simplegreeting Example

1 In a web browser, type the following URL:
   http://localhost:8080/simplegreeting
   The Simple Greeting page opens.

2 Type a name in the text field.
   For example, suppose that you type Duke.

3 Click the Say Hello button.
   If you did not modify the Printer.java file, the following text string appears below the button:
   Hi, Duke!
   If you commented out the @Informal annotation in the Printer.java file, the following text string appears below the button:
   Hello, Duke.

The guessnumber CDI Example

The guessnumber example, somewhat more complex than the simplegreeting example, illustrates the use of producer methods and of session and application scope. The example is a game in which you try to guess a number in fewer than ten attempts. It is similar to the guessnumber example described in Chapter 5, “Introduction to Facelets,” except that you can keep guessing until you get the right answer or until you use up your ten attempts.

The example includes four source files, a Facelets page and template, and configuration files. The configuration files and the template are the same as those used for the simplegreeting example.
The guessnumber Source Files

The four source files for the guessnumber example are

- The @MaxNumber qualifier interface
- The @Random qualifier interface
- The Generator managed bean, which defines producer methods
- The UserNumberBean managed bean

The source files are located in the 
tut-install/examples/cdi/guessnumber/src/java/guessnumber/ directory.

The @MaxNumber and @Random Qualifier Interfaces

The @MaxNumber qualifier interface is defined as follows:

```java
package guessnumber;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface MaxNumber {
}
```

The @Random qualifier interface is defined as follows:

```java
package guessnumber;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;
```
@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@Qualifier
public @interface Random {

}

The Generator Managed Bean
The Generator managed bean contains the two producer methods for the application. The bean has the @ApplicationScoped annotation to specify that its context extends for the duration of the user’s interaction with the application:

package guessnumber;
import java.io.Serializable;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;
@ApplicationScoped
public class Generator implements Serializable {
    private static final long serialVersionUID = -7213673465118041882L;
    private java.util.Random random = new java.util.Random( System.currentTimeMillis() );
    private int maxNumber = 100;
    java.util.Random getRandom() {
        return random;
    }
    @Produces @Random int next() {
        return getRandom().nextInt(maxNumber);
    }
    @Produces @MaxNumber int getMaxNumber() {
        return maxNumber;
    }
}

The UserNumberBean Managed Bean
The UserNumberBean managed bean, the managed bean for the JavaServer Faces application, provides the basic logic for the game. This bean does the following:
- Implements setter and getter methods for the bean fields
- Injects the two qualifier objects
- Provides a reset method that allows you to begin a new game after you complete one
- Provides a check method that determines whether the user has guessed the number
Provides a `validateNumberRange` method that determines whether the user's input is correct.

The bean is defined as follows:

```java
package guessnumber;

import java.io.Serializable;
import javax.annotation.PostConstruct;
import javax.enterprise.context.SessionScoped;
import javax.enterprise.inject.Instance;
import javax.inject.Inject;
import javax.inject.Named;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;

@Named
@SessionScoped
public class UserNumberBean implements Serializable {
    private static final long serialVersionUID = -7698506329160109476L;
    private int number;
    private Integer userNumber;
    private int minimum;
    private int remainingGuesses;

    @MaxNumber
    @Inject
    private int maxNumber;

    @Random
    @Inject
    Instance<Integer> randomInt;

    public UserNumberBean() {
    }

    public int getNumber() {
        return number;
    }

    public void setUserNumber(Integer user_number) {
        userNumber = user_number;
    }

    public Integer getUserNumber() {
        return userNumber;
    }

    public int getMaximum() {
        return (this.maximum);
    }
}
```
public void setMaximum(int maximum) {
    this.maximum = maximum;
}

public int getMinimum() {
    return (this.minimum);
}

public void setMinimum(int minimum) {
    this.minimum = minimum;
}

public int getRemainingGuesses() {
    return remainingGuesses;
}

public String check() throws InterruptedException {
    if (userNumber > number) {
        maximum = userNumber - 1;
    }
    if (userNumber < number) {
        minimum = userNumber + 1;
    }
    if (userNumber == number) {
        FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage("Correct!"));
    }
    remainingGuesses--; return null;
}

@PostConstruct
public void reset() {
    this.minimum = 0;
    this.userNumber = 0;
    this.remainingGuesses = 10;
    this.maximum = maxNumber;
    this.number = randomInt.get();
}

public void validateNumberRange(FacesContext context,
    UIComponent toValidate,
    Object value) {
    if (remainingGuesses <= 0) {
        FacesMessage message = new FacesMessage("No guesses left!");
        context.addMessage(toValidate.getClientId(context), message);
        ((UIInput) toValidate).setValid(false);
        return;
    }
    int input = (Integer) value;
    if (input < minimum || input > maximum) {
        ((UIInput) toValidate).setValid(false);
        FacesMessage message = new FacesMessage("Invalid guess");
        context.addMessage(toValidate.getClientId(context), message);
    }
}
The guessnumber CDI Example

The Facelets Page

This example uses the same template that the simplegreeting example uses. The index.xhtml file, however, is more complex.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en"
 xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html">
 <ui:composition template="/template.xhtml">
   <ui:define name="title">Guess My Number</ui:define>
   <ui:define name="head">Guess My Number</ui:define>
   <ui:define name="content">
     <h:form id="GuessMain">
       <div style="color: black; font-size: 24px;">
         <p>I'm thinking of a number from
           <span style="color: blue">#{userNumberBean.minimum}</span>
           to
           <span style="color: blue">#{userNumberBean.maximum}</span>.
           You have
           <span style="color: blue">#{userNumberBean.remainingGuesses}</span> guesses.</p>
       </div>
       <h:panelGrid border="0" columns="5" style="font-size: 18px;">
         <h:outputLabel for="inputGuess">Number:</h:outputLabel>
         <h:inputText id="inputGuess" value="#{userNumberBean.userNumber}"
           required="true" size="3"
           disabled="#{userNumberBean.number eq userNumberBean.userNumber}"
           validator="#{userNumberBean.validateNumberRange}"
         />
         <h:commandButton id="GuessButton" value="Guess"
           action="#{userNumberBean.check}"*
           disabled="#{userNumberBean.number eq userNumberBean.userNumber}"/>
         <h:commandButton id="RestartButton" value="Reset"
           action="#{userNumberBean.reset}"
           immediate="true" />
         <h:outputText id="Higher" value="Higher!"
           rendered="#{userNumberBean.number gt userNumberBean.userNumber and userNumberBean.userNumber ne 0}"*
           style="color: #d20005"/>
         <h:outputText id="Lower" value="Lower!"
           rendered="#{userNumberBean.number lt userNumberBean.userNumber and userNumberBean.userNumber ne 0}"*
           style="color: #d20005"/>
       </h:panelGrid>
     </h:form>
   </ui:define>
 </ui:composition>
</html>
```
The Facelets page presents the user with the minimum and maximum values and the number of guesses remaining. The user's interaction with the game takes place within the panelGrid table, which contains an input field, Guess and Reset buttons, and a text field that appears if the guess is higher or lower than the correct number. Every time the user clicks the Guess button, the userNumberBean.check method is called to reset the maximum or minimum value or, if the guess is correct, to generate a FacesMessage to that effect. The method that determines whether each guess is valid is userNumberBean.validateNumberRange.

Running the guessnumber Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the guessnumber application.

▼ To Build, Package, and Deploy the guessnumber Example Using NetBeans IDE

This procedure builds the application into the tut-install/examples/cdi/guessnumber/build/web/ directory. The contents of this directory are deployed to the GlassFish Server.

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to:
   tut-install/examples/cdi/

3 Select the guessnumber folder.

4 Select the Open as Main Project check box.

5 Click Open Project.

6 In the Projects tab, right-click the guessnumber project and select Deploy.

▼ To Build, Package, and Deploy the guessnumber Example Using Ant

1 In a terminal window, go to:
   tut-install/examples/cdi/guessnumber/

2 Type the following command:
   ant
   This command calls the default target, which builds and packages the application into a WAR file, guessnumber.war, located in the dist directory.
3 Type the following command:
    `ant deploy`
    The `guessnumber.war` file will be deployed to the GlassFish Server.

**To Run the guessnumber Example**

1 In a web browser, type the following URL:
    `http://localhost:8080/guessnumber`
    The Guess My Number page opens.

2 Type a number in the Number text field and click Guess.
    The minimum and maximum values are modified, along with the remaining number of
    guesses.

3 Keep guessing numbers until you get the right answer or run out of guesses.
    If you get the right answer, the input field and Guess button are grayed out.

4 Click the Reset button to play the game again with a new random number.
This chapter describes more advanced features of Contexts and Dependency Injection for the Java EE Platform. Specifically, it covers additional features CDI provides to enable loose coupling of components with strong typing, in addition to those described in “Overview of CDI” on page 516.

The following topics are addressed here:

- “Using Alternatives in CDI Applications” on page 539
- “Using Producer Methods, Producer Fields, and Disposer Methods in CDI Applications” on page 541
- “Using Predefined Beans in CDI Applications” on page 544
- “Using Events in CDI Applications” on page 545
- “Using Interceptors in CDI Applications” on page 547
- “Using Decorators in CDI Applications” on page 549
- “Using Stereotypes in CDI Applications” on page 550

Using Alternatives in CDI Applications

When you have more than one version of a bean you use for different purposes, you can choose between them during the development phase by injecting one qualifier or another, as shown in “The simplegreeting CDI Example” on page 527.

Instead of having to change the source code of your application, however, you can make the choice at deployment time by using alternatives.

Alternatives are commonly used for purposes like the following:

- To handle client-specific business logic that is determined at runtime
- To specify beans that are valid for a particular deployment scenario (for example, when country-specific sales tax laws require country-specific sales tax business logic)
- To create dummy (mock) versions of beans to be used for testing
To make a bean available for lookup, injection, or EL resolution using this mechanism, give it a `javax.enterprise.inject.Alternative` annotation and then use the alternative element to specify it in the beans.xml file.

For example, you might want to create a full version of a bean and also a simpler version that you use only for certain kinds of testing. The example described in "The encoder Example: Using Alternatives" on page 553 contains two such beans, `CoderImpl` and `TestCoderImpl`. The test bean is annotated as follows:

```java
@Alternative
public class TestCoderImpl implements Coder { ... }
```

The full version is not annotated:

```java
public class CoderImpl implements Coder { ... }
```

The managed bean injects an instance of the `Coder` interface:

```java
@Inject
Coder coder;
```

The alternative version of the bean is used by the application only if that version is declared as follows in the beans.xml file:

```xml
<beans ...>
  <alternatives>
    <class>encoder.TestCoderImpl</class>
  </alternatives>
</beans>
```

If the `alternatives` element is commented out in the beans.xml file, the `CoderImpl` class is used.

You can also have several beans that implement the same interface, all annotated `@Alternative`. In this case, you must specify in the beans.xml file which of these alternative beans you want to use. If `CoderImpl` were also annotated `@Alternative`, one of the two beans would always have to be specified in the beans.xml file.

### Using Specialization

Specialization has a function similar to that of alternatives, in that it allows you to substitute one bean for another. However, you might want to make one bean override the other in all cases. Suppose you defined the following two beans:

```java
@Default @Asynchronous
public class AsynchronousService implements Service { ... }
```

```java
@Alternative
public class MockAsynchronousService extends AsynchronousService { ... }
```
If you then declared MockAsynchronousService as an alternative in your beans.xml file, the following injection point would resolve to MockAsynchronousService:

```java
@Inject Service service;
```

The following, however, would resolve to AsynchronousService rather than MockAsynchronousService, because MockAsynchronousService does not have the @Asynchronous qualifier:

```java
@Inject @Asynchronous Service service;
```

To make sure MockAsynchronousService was always injected, you would have to implement all bean types and bean qualifiers of AsynchronousService. However, if AsynchronousService declared a producer method or observer method, even this cumbersome mechanism would not ensure that the other bean was never invoked. Specialization provides a simpler mechanism.

**Specialization** happens at development time as well as at runtime. If you declare that one bean specializes another, it extends the other bean class, and at runtime the specialized bean completely replaces the other bean. If the first bean is produced by means of a producer method, you must also override the producer method.

You specialize a bean by giving it the `javax.enterprise.inject.Specializes` annotation. For example, you might declare a bean as follows:

```java
@Specializes
public class MockAsynchronousService extends AsynchronousService { ... }
```

In this case, the MockAsynchronousService class will always be invoked instead of the AsynchronousService class.

Usually, a bean marked with the @Specializes annotation is also an alternative and is declared as an alternative in the beans.xml file. Such a bean is meant to stand in as a replacement for the default implementation, and the alternative implementation automatically inherits all qualifiers of the default implementation as well as its EL name, if it has one.

**Using Producer Methods, Producer Fields, and Disposer Methods in CDI Applications**

A producer method generates an object that can then be injected. Typically, you use producer methods in the following situations:

- When you want to inject an object that is not itself a bean
- When the concrete type of the object to be injected may vary at runtime
- When the object requires some custom initialization that the bean constructor does not perform
For more information on producer methods, see "Injecting Objects by Using Producer Methods" on page 524.

A producer field is a simpler alternative to a producer method; it is a field of a bean that generates an object. It can be used instead of a simple getter method. Producer fields are particularly useful for declaring Java EE resources such as data sources, JMS resources, and web service references.

A producer method or field is annotated with the javax.enterprise.inject.Produces annotation.

Using Producer Methods

A producer method can allow you to select a bean implementation at runtime, instead of at development time or deployment time. For example, in the example described in “The producer methods Example: Using a Producer Method To Choose a Bean Implementation” on page 558, the managed bean defines the following producer method:

```java
@Produces
@Chosen
@RequestScoped
public Coder getCoder(@New TestCoderImpl tci,
                      @New CoderImpl ci) {
    switch (coderType) {
        case TEST:
            return tci;
        case SHIFT:
            return ci;
        default:
            return null;
    }
}
```

The javax.enterprise.inject.New qualifier instructs the CDI runtime to instantiate both of the coder implementations and provide them as arguments to the producer method. Here, getCoder becomes in effect a getter method, and when the coder property is injected with the same qualifier and other annotations as the method, the selected version of the interface is used.

```java
@Inject
@Chosen
@RequestScoped
Coder coder;
```

Specifying the qualifier is essential: It tells CDI which Coder to inject. Without it, the CDI implementation would not be able to choose between CoderImpl, TestCoderImpl, and the one returned by getCoder, and would cancel deployment, informing the user of the ambiguous dependency.
Using Producer Fields to Generate Resources

A common use of a producer field is to generate an object such as a JDBC DataSource or a Java Persistence API EntityManager. The object can then be managed by the container. For example, you could create a @UserDatabase qualifier and then declare a producer field for an entity manager as follows:

```java
@Produces
@UserDatabase
@PersistenceContext
private EntityManager em;
```

The @UserDatabase qualifier can be used when you inject the object into another bean, RequestBean, elsewhere in the application:

```java
@Inject
@UserDatabase
EntityManager em;
...
```

"The producer fields Example: Using Producer Fields to Generate Resources" on page 561 shows how to use producer fields to generate an entity manager. You can use a similar mechanism to inject @Resource, @EJB, or @WebServiceRef objects.

To minimize the reliance on resource injection, specify the producer field for the resource in one place in the application, then inject the object wherever in the application you need it.

Using a Disposer Method

You can use a producer method to generate an object that needs to be removed when its work is completed. If you do, you need a corresponding disposer method, annotated with a @Disposes annotation. For example, if you used a producer method instead of a producer field to create the entity manager, you would create and close it as follows:

```java
@PersistenceContext
private EntityManager em;

@Produces
@UserDatabase
public EntityManager create() {
    return em;
}

public void close(@Disposes @UserDatabase EntityManager em) {
    em.close();
}
```

The disposer method is called automatically when the context ends (in this case, at the end of the conversation, because RequestBean has conversation scope), and the parameter in the close method receives the object produced by the producer method, create.
Using Predefined Beans in CDI Applications

CDI provides predefined beans that implement the following interfaces:

- `javax.transaction.UserTransaction`
  A Java Transaction API (JTA) user transaction.

- `java.security.Principal`
  The abstract notion of a principal, which represents any entity, such as an individual, a corporation, or a login ID. Whenever the injected principal is accessed, it always represents the identity of the current caller. For example, a principal is injected into a field at initialization. Later, a method that uses the injected principal is called on the object into which the principal was injected. In this situation, the injected principal represents the identity of the current caller when the method is run.

- `javax.validation.Validator`
  A validator for bean instances. The bean that implements this interface enables a `Validator` object for the default bean validation `ValidatorFactory` object to be injected.

- `javax.validation.ValidatorFactory`
  A factory class for returning initialized `Validator` instances. The bean that implements this interface enables the default bean validation object `ValidatorFactory` to be injected.

To inject a predefined bean, create an injection point by using the `javax.annotation.Resource` annotation to obtain an instance of the bean. For the bean type, specify the class name of the interface the bean implements.

Predefined beans are injected with dependent scope and the predefined default qualifier `@Default`.

For more information about injecting resources, see “Resource Injection” on page 807.

The following code snippet shows how to use the `@Resource` annotation to inject a predefined bean. This code snippet injects a user transaction into the servlet class `TransactionServlet`. The user transaction is an instance of the predefined bean that implements the `javax.transaction.UserTransaction` interface.

```java
import javax.annotation.Resource;
import javax.servlet.http.HttpServlet;
import javax.transaction.UserTransaction;
...
public class TransactionServlet extends HttpServlet {
    @Resource UserTransaction transaction;
    ...
}
```
Using Events in CDI Applications

Events allow beans to communicate without any compile-time dependency. One bean can define an event, another bean can fire the event, and yet another bean can handle the event. The beans can be in separate packages and even in separate tiers of the application.

Defining Events

An event consists of the following:

- The event object, a Java object
- Zero or more qualifier types, the event qualifiers

For example, in the bill payment example described in "The bill payment Example: Using Events and Interceptors" on page 568, a PaymentEvent bean defines an event using three properties, which have setter and getter methods:

```java
public String paymentType;
public BigDecimal value;
public Date datetime;

public PaymentEvent() {
}
```

The example also defines qualifiers that distinguish between two kinds of PaymentEvent. Every event also has the default qualifier @Any.

Using Observer Methods to Handle Events

An event handler uses an observer method to consume events.

Each observer method takes as a parameter an event of a specific event type that is annotated with the @Observes annotation and with any qualifiers for that event type. The observer method is notified of an event if the event object matches the event type and if all the qualifiers of the event match the observer method event qualifiers.

The observer method can take other parameters in addition to the event parameter. The additional parameters are injection points and can declare qualifiers.

The event handler for the bill payment example, PaymentHandler, defines two observer methods, one for each type of PaymentEvent:

```java
public void creditPayment(@Observes @Credit PaymentEvent event) {
  ...
}
```
public void debitPayment(@Observes @DebitPaymentEvent event) {
...
}

Observer methods can also be conditional or transactional:

■ A conditional observer method is notified of an event only if an instance of the bean that defines the observer method already exists in the current context. To declare a conditional observer method, specify notifyObserver=IF_EXISTS as an argument to @Observes:

   @Observes(notifyObserver=IF_EXISTS)

   To obtain the default unconditional behavior, you can specify @Observes(notifyObserver=ALWAYS).

■ A transactional observer method is notified of an event during the before-completion or after-completion phase of the transaction in which the event was fired. You can also specify that the notification is to occur only after the transaction has completed successfully or unsuccessfully. To specify a transactional observer method, use any of the following arguments to @Observes:

   @Observes(during=BEFORE_COMPLETION)
   @Observes(during=AFTER_COMPLETION)
   @Observes(during=AFTER_SUCCESS)
   @Observes(during=AFTER_FAILURE)

   To obtain the default non-transactional behavior, specify @Observes(during=IN_PROGRESS).

   An observer method that is called before completion of a transaction may call the setRollbackOnly method on the transaction instance to force a transaction rollback.

Observer methods may throw exceptions. If a transactional observer method throws an exception, the exception is caught by the container. If the observer method is non-transactional, the exception terminates processing of the event, and no other observer methods for the event are called.

### Firing Events

To activate an event, call the javax.enterprise.event.Event.fire method. This method fires an event and notifies any observer methods.

In the bill payment example, a managed bean called PaymentBean fires the appropriate event by using information it receives from the user interface. There are actually four event beans, two for the event object and two for the payload. The managed bean injects the two event beans. The pay method uses a switch statement to choose which event to fire, using new to create the payload.
@Inject
@Credit
Event<PaymentEvent> creditEvent;

@Inject
@Debit
Event<PaymentEvent> debitEvent;

private static final int DEBIT = 1;
private static final int CREDIT = 2;
private int paymentOption = DEBIT;
...

@Logged
public String pay() {
    ...
    switch (paymentOption) {
        case DEBIT:
            PaymentEvent debitPayload = new PaymentEvent();
            // populate payload ...
            debitEvent.fire(debitPayload);
            break;
        case CREDIT:
            PaymentEvent creditPayload = new PaymentEvent();
            // populate payload ...
            creditEvent.fire(creditPayload);
            break;
        default:
            logger.severe("Invalid payment option!");
    }
    ...
}

The argument to the fire method is a PaymentEvent that contains the payload. The fired event is then consumed by the observer methods.

Using Interceptors in CDI Applications

An interceptor is a class used to interpose in method invocations or lifecycle events that occur in an associated target class. The interceptor performs tasks, such as logging or auditing, that are separate from the business logic of the application and are repeated often within an application. Such tasks are often called cross-cutting tasks. Interceptors allow you to specify the code for these tasks in one place for easy maintenance. When interceptors were first introduced to the Java EE platform, they were specific to enterprise beans. On the Java EE 6 platform you can use them with Java EE managed objects of all kinds, including managed beans.

For information on Java EE interceptors, see Chapter 50, “Using Java EE Interceptors.”

An interceptor class often contains a method annotated @AroundInvoke, which specifies the tasks the interceptor will perform when intercepted methods are invoked. It can also contain a method annotated @PostConstruct, @PreDestroy, @PrePassivate, or @PostActivate, to specify lifecycle callback interceptors, and a method annotated @AroundTimeout, to specify EJB...
timeout interceptors. An interceptor class can contain more than one interceptor method, but it must have no more than one method of each type.

Along with an interceptor, an application defines one or more interceptor binding types, which are annotations that associate an interceptor with target beans or methods. For example, the bill payment example contains an interceptor binding type named @Logged and an interceptor named LoggedInterceptor.

The interceptor binding type declaration looks something like a qualifier declaration, but it is annotated with javax.interceptor.InterceptorBinding:

```java
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logged {
}
```

An interceptor binding also has the java.lang.annotation.Inherited annotation, to specify that the annotation can be inherited from superclasses. The @Inherited annotation also applies to custom scopes (not discussed in this tutorial), but does not apply to qualifiers.

An interceptor binding type may declare other interceptor bindings.

The interceptor class is annotated with the interceptor binding as well as with the @Interceptor annotation. For an example, see "The LoggedInterceptor Interceptor Class" on page 572.

Every @AroundInvoke method takes a javax.interceptor.InvocationContext argument, returns a java.lang.Object, and throws an Exception. It can call InvocationContext methods. The @AroundInvoke method must call the proceed method, which causes the target class method to be invoked.

Once an interceptor and binding type are defined, you can annotate beans and individual methods with the binding type to specify that the interceptor is to be invoked either on all methods of the bean or on specific methods. For example, in the bill payment example, the PaymentHandler bean is annotated @Logged, which means that any invocation of its business methods will cause the interceptor's @AroundInvoke method to be invoked:

```java
@Logged
@SessionScoped
public class PaymentHandler implements Serializable {...}
```

However, in the PaymentBean bean, only the pay and reset methods have the @Logged annotation, so the interceptor is invoked only when these methods are invoked:

```java
@Logged
public String pay() {...}

@Logged
public void reset() {...}
```
In order for an interceptor to be invoked in a CDI application, it must, like an alternative, be specified in the beans.xml file. For example, the LoggedInterceptor class is specified as follows:

```xml
<interceptors>
  <class>billpayment.interceptors.LoggedInterceptor</class>
</interceptors>
```

If an application uses more than one interceptor, the interceptors are invoked in the order specified in the beans.xml file.

### Using Decorators in CDI Applications

A **decorator** is a Java class that is annotated `javax.decorator.Decorator` and that has a corresponding `decorators` element in the beans.xml file.

A decorator bean class must also have a delegate injection point, which is annotated `javax.decorator.Delegate`. This injection point can be a field, a constructor parameter, or an initializer method parameter of the decorator class.

Decorators are outwardly similar to interceptors. However, they actually perform tasks complementary to those performed by interceptors. Interceptors perform cross-cutting tasks associated with method invocation and with the lifecycles of beans, but cannot perform any business logic. Decorators, on the other hand, do perform business logic by intercepting business methods of beans. This means that instead of being reusable for different kinds of applications as are interceptors, their logic is specific to a particular application.

For example, instead of using an alternative `TestCoderImpl` class for the encoder example, you could create a decorator as follows:

```java
@Decorator
public abstract class CoderDecorator implements Coder {

    @Inject
    @Delegate
    @Any
    Coder coder;

    public String codeString(String s, int tval) {
        int len = s.length();

        return "\" + s + "\" becomes " + "\" + coder.codeString(s, tval)
            + "\", " + len + " characters in length";
    }
}
```

See “The decorators Example: Decorating a Bean” on page 574 for an example that uses this decorator.
This simple decorator returns more detailed output than the encoded string returned by the CoderImpl.codeString method. A more complex decorator could store information in a database or perform some other business logic.

A decorator can be declared as an abstract class, so that it does not have to implement all the business methods of the interface.

In order for a decorator to be invoked in a CDI application, it must, like an interceptor or an alternative, be specified in the beans.xml file. For example, the CoderDecorator class is specified as follows:

```xml
<decorators>
  <class>decorators.CoderDecorator</class>
</decorators>
```

If an application uses more than one decorator, the decorators are invoked in the order in which they are specified in the beans.xml file.

If an application has both interceptors and decorators, the interceptors are invoked first. This means, in effect, that you cannot intercept a decorator.

**Using Stereotypes in CDI Applications**

A stereotype is a kind of annotation, applied to a bean, that incorporates other annotations. Stereotypes can be particularly useful in large applications where you have a number of beans that perform similar functions. A stereotype is a kind of annotation that specifies the following:

- A default scope
- Zero or more interceptor bindings
- Optionally, a @Named annotation, guaranteeing default EL naming
- Optionally, an @Alternative annotation, specifying that all beans with this stereotype are alternatives

A bean annotated with a particular stereotype will always use the specified annotations, so you do not have to apply the same annotations to many beans.

For example, you might create a stereotype named Action, using the javax.enterprise.inject.Stereotype annotation:

```java
@RequestScoped
@Secure
@Transaction
@Named
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
All beans annotated @Action will have request scope, use default EL naming, and have the interceptor bindings @Transactional and @Secure.

You could also create a stereotype named Mock:

```java
@Alternative
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Mock {}  
```

All beans with this annotation are alternatives.

It is possible to apply multiple stereotypes to the same bean, so you can annotate a bean as follows:

```java
@Action
@Mock
public class MockLoginAction extends LoginAction { ... }
```

It is also possible to override the scope specified by a stereotype, simply by specifying a different scope for the bean. The following declaration gives the MockLoginAction bean session scope instead of request scope:

```java
@SessionScoped
@Action
@Mock
public class MockLoginAction extends LoginAction { ... }
```

CDI makes available a built-in stereotype called Model, which is intended for use with beans that define the model layer of a model-view-controller application architecture. This stereotype specifies that a bean is both @Named and @RequestScoped:

```java
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {}  
```
This chapter describes in detail how to build and run several advanced examples that use CDI. The examples are in the `tut-install/examples/cdi` directory.

To build and run the examples, you will do the following:

1. Use NetBeans IDE or the Ant tool to compile, package, and deploy the example.
2. Run the example in a web browser.

Each example has a `build.xml` file that refers to files in the `tut-install/examples/bp-project/` directory.

See Chapter 2, “Using the Tutorial Examples,” for basic information on installing, building, and running the examples.

The following topics are addressed here:

- “The encoder Example: Using Alternatives” on page 553
- “The producermethods Example: Using a Producer Method To Choose a Bean Implementation” on page 558
- “The producerfields Example: Using Producer Fields to Generate Resources” on page 561
- “The billpayment Example: Using Events and Interceptors” on page 568
- “The decorators Example: Decorating a Bean” on page 574

The encoder Example: Using Alternatives

The encoder example shows how to use alternatives to choose between two beans at deployment time, as described in “Using Alternatives in CDI Applications” on page 539. The example includes an interface and two implementations of it, a managed bean, a Facelets page, and configuration files.
The Coder Interface and Implementations

The Coder interface contains just one method, `codeString`, that takes two arguments: a string, and an integer value that specifies how the letters in the string should be transposed.

```java
public interface Coder {
    public String codeString(String s, int tval);
}
```

The interface has two implementation classes, `CoderImpl` and `TestCoderImpl`. The implementation of `codeString` in `CoderImpl` shifts the string argument forward in the alphabet by the number of letters specified in the second argument; any characters that are not letters are left unchanged. (This simple shift code is known as a Caesar cipher, for Julius Caesar, who reportedly used it to communicate with his generals.) The implementation in `TestCoderImpl` merely displays the values of the arguments. The `TestCoderImpl` implementation is annotated `@Alternative`:

```java
import javax.enterprise.inject.Alternative;
@Alternative
public class TestCoderImpl implements Coder {
    public String codeString(String s, int tval) {
        return "input string is " + s + "", shift value is " + tval;"
    }
}
```

The beans.xml file for the encoder example contains an `alternatives` element for the `TestCoderImpl` class, but by default the element is commented out:

```xml
<beans ... >
    <!-<alternatives>
    <class>encoder.TestCoderImpl</class>
    </alternatives>->
</beans>
```

This means that by default, the `TestCoderImpl` class, annotated `@Alternative`, will not be used. Instead, the `CoderImpl` class will be used.

The encoder Facelets Page and Managed Bean

The simple Facelets page for the encoder example, `index.xhtml`, asks the user to type the string and integer values and passes them to the managed bean, `CoderBean`, as `coderBean.inputString` and `coderBean.transVal`:

```xml
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
```
When the user clicks the Encode button, the page invokes the managed bean’s encodeString method and displays the result, coderBean.codedString, in blue. The page also has a Reset button that clears the fields.

The managed bean, CoderBean, is a @RequestScoped bean that declares its input and output properties. The transVal property has three Bean Validation constraints that enforce limits on the integer value, so that if the user types an invalid value, a default error message appears on the Facelets page. The bean also injects an instance of the Coder interface:

@Named
@RequestScoped
public class CoderBean {
    private String inputString;
    private String codedString;
    @Max(26)
    @Min(0)
    @NotNull
    private int transVal;

    @Inject
    Coder coder;
    ...
In addition to simple getter and setter methods for the three properties, the bean defines the `encodeString` action method called by the Facelets page. This method sets the `codedString` property to the value returned by a call to the `codeString` method of the `Coder` implementation:

```java
public void encodeString() {
    setCodedString(coder.codeString(inputString, transVal));
}
```

Finally, the bean defines the `reset` method to empty the fields of the Facelets page:

```java
public void reset(){
    setInputString("");
    setTransVal(0);
}
```

---

**Running the encoder Example**

You can use either NetBeans IDE or Ant to build, package, deploy, and run the encoder application.

**▼ To Build, Package, and Deploy the encoder Example Using NetBeans IDE**

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   
   `tut-install/examples/cdi/

3. Select the encoder folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the encoder project and select Deploy.

**▼ To Run the encoder Example Using NetBeans IDE**

1. In a web browser, type the following URL:
   
   `http://localhost:8080/encoder`

   The String Encoder page opens.
2 Type a string and the number of letters to shift by, then click Encode. The encoded string appears in blue on the Result line. For example, if you type Java and 4, the result is Neze.

3 Now, edit the beans.xml file to enable the alternative implementation of Coder.
   a. In the Projects tab, under the encoder project, expand the Web Pages node, then the WEB-INF node.
   b. Double-click the beans.xml file to open it.
   c. Remove the comment characters that surround the alternatives element, so that it looks like this:
      ```xml
      <alternatives>
      <class>encoder.TestCoderImpl</class>
      </alternatives>
      ```
   d. Save the file.

4 Right-click the encoder project and select Deploy.

5 In the web browser, retype the URL to show the String Encoder page for the redeployed project:
   http://localhost:8080/encoder/

6 Type a string and the number of letters to shift by, then click Encode.
   This time, the Result line displays your arguments. For example, if you type Java and 4, the result is:
   Result: input string is Java, shift value is 4

▼ To Build, Package, and Deploy the encoder Example Using Ant

1 In a terminal window, go to:
   tut-install/examples/cdi/encoder/

2 Type the following command:
   `ant`
   This command calls the default target, which builds and packages the application into a WAR file, encoder.war, located in the dist directory.

3 Type the following command:
   `ant deploy`
To Run the encoder Example Using Ant

1. In a web browser, type the following URL:
   http://localhost:8080/encoder/
   The String Encoder page opens.

2. Type a string and the number of letters to shift by, then click Encode.
   The encoded string appears in blue on the Result line. For example, if you type Java and 4, the result is Neze.

3. Now, edit the beans.xml file to enable the alternative implementation of Coder.
   a. In a text editor, open the following file:
      tut-install/examples/cdi/encoder/web/WEB-INF/beans.xml
   b. Remove the comment characters that surround the alternatives element, so that it looks like this:
      <alternatives>
       <class>encoder.TestCoderImpl</class>
      </alternatives>
   c. Save and close the file.

4. Type the following commands:
   ant undeploy
   ant
   ant deploy

5. In the web browser, retype the URL to show the String Encoder page for the redeployed project:
   http://localhost:8080/encoder

6. Type a string and the number of letters to shift by, then click Encode.
   This time, the Result line displays your arguments. For example, if you type Java and 4, the result is:
   Result: input string is Java, shift value is 4

The producermethods Example: Using a Producer Method To Choose a Bean Implementation

The producermethods example shows how to use a producer method to choose between two beans at runtime, as described in “Using Producer Methods, Producer Fields, and Disposer Methods in CDI Applications” on page 541. It is very similar to the encoder example described
Components of the producer methods Example

The components of producer methods are very much like those for encoder, with some significant differences.

Neither implementation of the Coder bean is annotated @Alternative, and the beans.xml file does not contain an alternatives element.

The Facelets page and the managed bean, CoderBean, have an additional property, coderType, that allows the user to specify at runtime which implementation to use. In addition, the managed bean has a producer method that selects the implementation using a qualifier type, @Chosen.

The bean declares two constants that specify whether the coder type is the test implementation or the implementation that actually shifts letters:

```java
private final static int TEST = 1;
private final static int SHIFT = 2;
private int coderType = SHIFT; // default value
```

The producer method, annotated with @Produces and @Chosen as well as @RequestScoped (so that it lasts only for the duration of a single request and response), takes both implementations as arguments, then returns one or the other, based on the coderType supplied by the user.

```java
@Produces
@Chosen
@RequestScoped
public Coder getCoder(@New TestCoderImpl tci,
@New CoderImpl ci) {
    switch (coderType) {
    case TEST:
        return tci;
    case SHIFT:
        return ci;
    default:
        return null;
    }
}
```

Finally, the managed bean injects the chosen implementation, specifying the same qualifier as that returned by the producer method to resolve ambiguities:

```java
@Inject
@Chosen
```
@RequestScoped
coder;

The Facelets page contains modified instructions and a pair of radio buttons whose selected value is assigned to the property coderBean.coderType:

```xml
<h:form id="encodeit">
  <h:selectOneRadio id="coderType" required=true value="#{coderBean.coderType}">
    <f:selectItem itemValue="1" itemLabel="Test"/>
    <f:selectItem itemValue="2" itemLabel="Shift Letters"/>
  </h:selectOneRadio>
  ...  
</h:form>
```

**Running the producermethods Example**

You can use either NetBeans IDE or Ant to build, package, deploy, and run the producermethods application.

**To Build, Package, and Deploy the producermethods Example Using NetBeans IDE**

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   
   `tut-install/examples/cdi/`

3. Select the producermethods folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the producermethods project and select Deploy.
To Build, Package, and Deploy the producermethods Example Using Ant

1. In a terminal window, go to:
   
tut-install/examples/cdi/producermethods/

2. Type the following command:
   
ant
   This command calls the default target, which builds and packages the application into a WAR file, producermethods.war, located in the dist directory.

3. Type the following command:
   
ant deploy

To Run the producermethods Example

1. In a web browser, type the following URL:
   
http://localhost:8080/producermethods
   
The String Encoder page opens.

2. Select either the Test or Shift Letters radio button, type a string and the number of letters to shift by, then click Encode.
   
Depending on your selection, the Result line displays either the encoded string or the input values you specified.

The producerfields Example: Using Producer Fields to Generate Resources

The producerfields example, which allows you to create a to-do list, shows how to use a producer field to generate objects that can then be managed by the container. This example generates an EntityManager object, but resources such as JDBC connections and datasources can also be generated this way.

The producerfields example is the simplest possible entity example. It also contains a qualifier and a class that generates the entity manager. It also contains a single entity, a stateful session bean, a Facelet page, and a managed bean.

The Producer Field for the producerfields Example

The most important component of the producerfields example is the smallest, the db.UserDatabaseEntityManager class, which isolates the generation of the EntityManager
The producer fields Example: Using Producer Fields to Generate Resources

object so it can easily be used by other components in the application. The class uses a producer field to inject an EntityManager annotated with the @UserDatabase qualifier, also defined in the db package:

```
@Singleton
public class UserDatabaseEntityManager {
  @PersistenceContext
  @UserDatabase
  private EntityManager em;
  ...
}
```

The class does not explicitly produce a persistence unit field, but the application has a persistence.xml file that specifies a persistence unit. The class is annotated javax.inject.Singleton to specify that the injector should instantiate it only once.

The db.UserDatabaseEntityManager class also contains commented-out code that uses create and close methods to generate and remove the producer field:

```
/* @PersistenceContext
    private EntityManager em;
    @Produces
    @UserDatabase
    public EntityManager create() {
      return em;
    } */

    public void close(@Disposes @UserDatabase EntityManager em) {
      em.close();
    }
```

You can remove the comment indicators from this code and place them around the field declaration to test how the methods work. The behavior of the application is the same with either mechanism.

The advantage of producing the EntityManager in a separate class rather than simply injecting it into an enterprise bean is that the object can easily be reused in a typesafe way. Also, a more complex application can create multiple entity managers using multiple persistence units, and this mechanism isolates this code for easy maintenance, as in the following example:

```
@Singleton
public class JPAResourceProducer {
  @Produces
  @PersistenceUnit(unitName="pu3")
  @TestDatabase
  EntityManagerFactory customerDatabasePersistenceUnit;

  @Produces
  @PersistenceContext(unitName="pu3")
  @TestDatabase
```
The producer fields example contains a simple entity class, entity. ToDo, and a stateful session bean, ejb.RequestBean, that uses it.

The entity class contains three fields: an autogenerated id field, a string specifying the task, and a timestamp. The timestamp field, timeCreated, is annotated with @Temporal, which is required for persistent Date fields.

```java
@Entity
public class ToDo implements Serializable {
...
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
protected String taskText;
@Temporal(TIMESTAMP)
protected Date timeCreated;
public ToDo() {
}
public ToDo(Long id, String taskText, Date timeCreated) {
    this.id = id;
    this.taskText = taskText;
    this.timeCreated = timeCreated;
}
...}
```

The remainder of the ToDo class contains the usual getters, setters, and other entity methods.

The RequestBean class injects the EntityManager generated by the producer method, annotated with the @UserDatabase qualifier:

```java
@ConversationScoped
@Stateful
public class RequestBean {
...
```
The producerfields Example: Using Producer Fields to Generate Resources

```java
@Inject
@UserDatabase
EntityManager em;

It then defines two methods, one that creates and persists a single ToDo list item, and another
that retrieves all the ToDo items created so far by creating a query:

```java
public ToDo createToDo(String inputString) {
    ToDo todo;
    Date currentTime = Calendar.getInstance().getTime();
    try {
        todo = new ToDo();
        todo.setTaskText(inputString);
        todo.setTimeCreated(currentTime);
        em.persist(todo);
        return todo;
    } catch (Exception e) {
        throw new EJBException(e.getMessage());
    }
}

public List<ToDo> getToDos() {
    try {
        List<ToDo> toDos =
            (List<ToDo>) em.createQuery(
                "SELECT t FROM ToDo t ORDER BY t.timeCreated"
            ).getResultList();
        return toDos;
    } catch (Exception e) {
        throw new EJBException(e.getMessage());
    }
}
```

The producerfields Example: Facelets Pages and Managed Bean

The producerfields example has two Facelets pages, index.xhtml and todolist.xhtml. The
simple form on the index.xhtml page asks the user only for the task. When the user clicks the
Submit button, the listBean.createTask method is called. When the user clicks the Show
Items button, the action specifies that the todolist.xhtml file should be displayed:

```xml
<h:body>
    <h2>To Do List</h2>
    <p>Type a task to be completed.</p>
    <h:form id="todolist">
        <p><h:outputLabel value="Type a string: " for="inputString"/></p>
        <h:inputText id="inputString" value="#{listBean.inputString}"/>
        <p><h:commandButton value="Submit" action="#{listBean.createTask()}"></p>
```
The managed bean, web.ListBean, injects the ejb.RequestBean session bean. It declares the entity ToDo entity and a list of the entity, along with the input string that it passes to the session bean. The inputString is annotated with the @NotNull Bean Validation constraint, so an attempt to submit an empty string results in an error.

@Named  
@ConversationScoped  
public class ListBean implements Serializable {

  ...  
  @EJB  
  private RequestBean request;  
  @NotNull  
  private String inputString;  
  private ToDo todo;  
  private List<ToDo> todos;

The createTask method called by the Submit button calls the createToDo method of RequestBean:

  public void createTask() {  
    this.todo = request.createToDo(inputString);  
  }

The getToDos method, which is called by the todolist.xhtml page, calls the getToDos method of RequestBean:

  public List<ToDo> getToDos() {  
    return request.getToDos();  
  }

To force the Facelets page to recognize an empty string as a null value and return an error, the web.xml file sets the context parameter javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL to true:

<context-param>  
  <param-name>javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-name>  
  <param-value>true</param-value>  
</context-param>

The todolist.xhtml page is a little more complicated than the index.html page. It contains a dataTable element that displays the contents of the ToDo list. The body of the page looks like this:
The value of the dataTable is listBean.toDos, the list returned by the managed bean’s getToDos method, which in turn calls the session bean’s getToDos method. Each row of the table displays the timeCreated and taskText fields of the individual task. Finally, a Back button returns the user to the index.xhtml page.

**Running the producerfields Example**

You can use either NetBeans IDE or Ant to build, package, deploy, and run the producerfields application.

**To Build, Package, and Deploy the producerfields Example Using NetBeans IDE**

1. If the database server is not already running, start it by following the instructions in “Starting and Stopping the Java DB Server” on page 75.

2. From the File menu, choose Open Project.

3. In the Open Project dialog, navigate to:
  
tut-install/examples/cdi/

4. Select the producerfields folder.

5. Select the Open as Main Project check box.
6 Click Open Project.

7 In the Projects tab, right-click the producerfields project and select Deploy.

▼ **To Build, Package, and Deploy the producerfields Example Using Ant**

1 If the database server is not already running, start it by following the instructions in “Starting and Stopping the Java DB Server” on page 75.

2 In a terminal window, go to:
   
   tut-install/examples/cdi/producerfields/

3 Type the following command:
   
   ant
   
   This command calls the default target, which builds and packages the application into a WAR file, producerfields.war, located in the dist directory.

4 Type the following command:
   
   ant deploy

▼ **To Run the producerfields Example**

1 In a web browser, type the following URL:
   
   http://localhost:8080/producerfields
   
   The Create To Do List page opens.

2 Type a string in the text field and click Submit.
   
   You can type additional strings and click Submit to create a task list with multiple items.

3 Click the Show Items button.
   
   The To Do List page opens, showing the timestamp and text for each item you created.

4 Click the Back button to return to the Create To Do List page.
   
   On this page, you can enter more items in the list.
The billpayment Example: Using Events and Interceptors

The billpayment example shows how to use both events and interceptors.

The example simulates paying an amount using a debit card or credit card. When the user chooses a payment method, the managed bean creates an appropriate event, supplies its payload, and fires it. A simple event listener handles the event using observer methods.

The example also defines an interceptor that is set on a class and on two methods of another class.

The PaymentEvent Event Class

The event class, event.PaymentEvent, is a simple bean class that contains a no-argument constructor. It also has a toString method and getter and setter methods for the payload components: a String for the payment type, a BigDecimal for the payment amount, and a Date for the timestamp.

```java
public class PaymentEvent implements Serializable {
    public String paymentType;
    public BigDecimal value;
    public Date datetime;

    public PaymentEvent() {
    }

    @Override
    public String toString() {
        return this.paymentType
            + " = " + this.value.toString()
            + " at " + this.datetime.toString();
    }
}
```

The event class is a simple bean that is instantiated by the managed bean using new and then populated. For this reason, the CDI container cannot intercept the creation of the bean, and hence it cannot allow interception of its getter and setter methods.

The PaymentHandler Event Listener

The event listener, listener.PaymentHandler, contains two observer methods, one for each of the two event types:

```java
@Logged
@SessionScoped
public class PaymentHandler implements Serializable {
    // event listener methods...
}
```
public void creditPayment(@Observes @CreditPaymentEvent event) {
    logger.log(Level.INFO, "PaymentHandler - Credit Handler: {0}",
            event.toString());
    // call a specific Credit handler class...
}

public void debitPayment(@Observes @DebitPaymentEvent event) {
    logger.log(Level.INFO, "PaymentHandler - Debit Handler: {0}",
            event.toString());
    // call a specific Debit handler class...
}

Each observer method takes as an argument the event, annotated with @Observes and with the qualifier for the type of payment. In a real application, the observer methods would pass the event information on to another component that would perform business logic on the payment.

The qualifiers are defined in the payment package, described in “The billpayment Facelets Pages and Managed Bean” on page 569.

Like PaymentEvent, the PaymentHandler bean is annotated @Logged, so that all its methods can be intercepted.

The billpayment Facelets Pages and Managed Bean

The billpayment example contains two Facelets pages, index.xhtml and the very simple response.xhtml. The body of index.xhtml looks like this:

```xml
<h:body>
    <h:form>
        <h:outputLabel value="Amount: $" for="amt"/>
        <h:inputText id="amt" value="#{paymentBean.value}" required="true" requiredMessage="An amount is required." maxlength="15"/>
    </h:form>
    <h:outputLabel value="Options:" for="opt"/>
    <h:selectOneRadio id="opt" value="#{paymentBean.paymentOption}"
        <f:selectItem id="debit" itemLabel="Debit Card" itemValue="1"/>
        <f:selectItem id="credit" itemLabel="Credit Card" itemValue="2"/>
    </h:selectOneRadio>
    <p><h:commandButton id="submit" value="Pay" action="#{paymentBean.pay}" /></p>
</h:body>
```
The input text field takes a payment amount, passed to paymentBean.value. Two radio buttons ask the user to select a Debit Card or Credit Card payment, passing the integer value to paymentBean.paymentOption. Finally, the Pay command button's action is set to the method paymentBean.pay, while the Reset button's action is set to the paymentBean.reset method.

The payment.Bean managed bean uses qualifiers to differentiate between the two kinds of payment event:

@Named
@SessionScoped
public class PaymentBean implements Serializable {

    @Inject
    @Credit
    Event<PaymentEvent> creditEvent;

    @Inject
    @Debit
    Event<PaymentEvent> debitEvent;

    The qualifiers, @Credit and @Debit, are defined in the payment package along with PaymentBean.

    Next, the PaymentBean defines the properties it obtains from the Facelets page and will pass on to the event:

    public static final int DEBIT = 1;
    public static final int CREDIT = 2;
    private int paymentOption = DEBIT;
    @Digits(integer = 10, fraction = 2, message = "Invalid value")
    private BigDecimal value;
    private Date datetime;

    The paymentOption value is an integer passed in from the radio button component; the default value is DEBIT. The value is a BigDecimal with a Bean Validation constraint that enforces a currency value with a maximum number of digits. The timestamp for the event, datetime, is a Date object initialized when the pay method is called.

    The pay method of the bean first sets the timestamp for this payment event. It then creates and populates the event payload, using the constructor for the PaymentEvent and calling the event's setter methods using the bean properties as arguments. It then fires the event.

    @Logged
    public String pay() {
        this.setDatetime(Calendar.getInstance().getTime());
switch (paymentOption) {
    case DEBIT:
        PaymentEvent debitPayload = new PaymentEvent();
        debitPayload.setPaymentType("Debit");
        debitPayload.setValue(value);
        debitPayload.setDatetime(datetime);
        debitEvent.fire(debitPayload);
        break;
    case CREDIT:
        PaymentEvent creditPayload = new PaymentEvent();
        creditPayload.setPaymentType("Credit");
        creditPayload.setValue(value);
        creditPayload.setDatetime(datetime);
        creditEvent.fire(creditPayload);
        break;
    default:
        logger.severe("Invalid payment option!");
    }
    return "response.xhtml";
}

The pay method returns the page to which the action is redirected, response.xhtml.

The PaymentBean class also contains a reset method that empties the value field on the index.xhtml page and sets the payment option to the default:

@Logged
public void reset() {
    setPaymentOption(DEBIT);
    setValue(BigDecimal.ZERO);
}

In this bean, only the pay and reset methods are intercepted.

The response.xhtml page displays the amount paid. It uses a rendered expression to display the payment method:

<h:body>
    <h:form>
        <h2>Bill Payment: Result</h2>
        <h3>Amount Paid with
            <h:outputText id="debit" value="Debit Card: ">
                rendered="#{paymentBean.paymentOption eq 1}" />
            <h:outputText id="credit" value="Credit Card: ">
                rendered="#{paymentBean.paymentOption eq 2}" />
            <h:outputText id="result" value="#{paymentBean.value}" >
                <f:convertNumber type="currency" />
            </h:outputText>
        </h3>
        <p><h:commandButton id="back" value="Back" action="index" /></p>
    </h:form>
</h:body>
The LoggedInterceptor Interceptor Class

The interceptor class, LoggedInterceptor, and its interceptor binding, Logged, are both defined in the interceptor package. The Logged interceptor binding is defined as follows:

```java
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logged {
}
```

The LoggedInterceptor class looks like this:

```java
@Logged
@Interceptor
public class LoggedInterceptor implements Serializable {

    ... public LoggedInterceptor() {
    }

    @AroundInvoke
    public Object logMethodEntry(InvocationContext invocationContext)
        throws Exception {
        System.out.println("Entering method: "
            + invocationContext.getMethod().getName() + " in class "
            + invocationContext.getMethod().getDeclaringClass().getName());
        return invocationContext.proceed();
    }

    }
```

The class is annotated with both the @Logged and the @Interceptor annotations. The @AroundInvoke method, logMethodEntry, takes the required InvocationContext argument, and calls the required proceed method. When a method is intercepted, logMethodEntry displays the name of the method being invoked as well as its class.

To enable the interceptor, the beans.xml file defines it as follows:

```xml
<interceptors>
    <class>billpayment.interceptor.LoggedInterceptor</class>
</interceptors>
```

In this application, the PaymentEvent and PaymentHandler classes are annotated @Logged, so all their methods are intercepted. In PaymentBean, only the pay and reset methods are annotated @Logged, so only those methods are intercepted.
Running the billpayment Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the billpayment application.

▼ To Build, Package, and Deploy the billpayment Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   tut-install/examples/cdi/
3. Select the billpayment folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the billpayment project and select Deploy.

▼ To Build, Package, and Deploy the billpayment Example Using Ant

1. In a terminal window, go to:
   tut-install/examples/cdi/billpayment/
2. Type the following command:
   ant
   This command calls the default target, which builds and packages the application into a WAR file, billpayment.war, located in the dist directory.
3. Type the following command:
   ant deploy

▼ To Run the billpayment Example

1. In a web browser, type the following URL:
   http://localhost:8080/billpayment
   The Bill Payment Options page opens.
2 Type a value in the Amount field.
The amount can contain up to 10 digits and include up to 2 decimal places. For example: 9876.54

3 Select Debit Card or Credit Card and click Pay.
The Bill Payment: Result page opens, displaying the amount paid and the method of payment:
Amount Paid with Credit Card: $9,876.34

4 (Optional) Click Back to return to the Bill Payment Options page.
You can also click Reset to return to the initial page values.

5 Examine the server log output.
In NetBeans IDE, the output is visible in the GlassFish Server 3+ output window. Otherwise, view domain-dir/logs/server.log.
The output from each interceptor appears in the log, followed by the additional logger output defined by the constructor and methods.

The decorators Example: Decorating a Bean

The decorators example, which is yet another variation on the encoder example, shows how to use a decorator to implement additional business logic for a bean. Instead of having the user choose between two alternative implementations of an interface at deployment time or runtime, a decorator adds some additional logic to a single implementation of the interface.

The example includes an interface, an implementation of it, a decorator, an interceptor, a managed bean, a Facelets page, and configuration files.

Components of the decorators Example

The decorators example is very similar to the encoder example described in “The encoder Example: Using Alternatives” on page 553. Instead of providing two implementations of the Coder interface, however, this example provides only the CoderImpl class. The decorator class, CoderDecorator, rather than simply return the coded string, displays the input and output strings’ values and length.

The CoderDecorator class, like CoderImpl, implements the business method of the Coder interface, codeString:

```java
@Decorator
public abstract class CoderDecorator implements Coder {
    @Inject
```
```java
@Delegate
@Any
Coder coder;

@Override
public String codeString(String s, int tval) {
    int len = s.length();
    return "\" + s + " becomes " + "\n" + coder.codeString(s, tval) + "\", " + len + " characters in length";
}
}

The decorator's codeString method calls the delegate object's codeString method to perform the actual encoding.

The decorators example includes the Logged interceptor binding and LoggedInterceptor class from the billpayment example. For this example, the interceptor is set on the CoderBean.encodeString method and the CoderImpl.codeString method. The interceptor code is unchanged; interceptors are usually reusable for different applications.

Except for the interceptor annotations, the CoderBean and CoderImpl classes are identical to the versions in the encoder example.

The beans.xml file specifies both the decorator and the interceptor:

```
<decorators>
    <class>decorators.CoderDecorator</class>
</decorators>
```

```
<interceptors>
    <class>decorators.LoggedInterceptor</class>
</interceptors>
```

### Running the decorators Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the decorators application.

**To Build, Package, and Deploy the decorators Example Using NetBeans IDE**

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   `tut-install/examples/cedi/
3. Select the decorators folder.
4. Select the Open as Main Project check box.

The decorators Example: Decorating a Bean

Chapter 31 • Running the Advanced Contexts and Dependency Injection Examples
5 Click Open Project.

6 In the Projects tab, right-click the decorators project and select Deploy.

▼ To Build, Package, and Deploy the decorators Example Using Ant

1 In a terminal window, go to:
   `tut-install/examples/cdi/decorators/`

2 Type the following command:
   ```
   ant
   ```
   This command calls the default target, which builds and packages the application into a WAR file, `decorators.war`, located in the `dist` directory.

3 Type the following command:
   ```
   ant deploy
   ```

▼ To Run the decorators Example

1 In a web browser, type the following URL:
   `http://localhost:8080/decorators`
   The Decorated String Encoder page opens.

2 Type a string and the number of letters to shift by, then click Encode.
   The output from the decorator method appears in blue on the Result line. For example, if you typed Java and 4, you would see the following:
   "Java" becomes "Neze", 4 characters in length

3 Examine the server log output.
   In NetBeans IDE, the output is visible in the GlassFish Server 3+ output window. Otherwise, view `domain-dir/logs/server.log`.
   The output from the interceptors appears:
   ```
   INFO: Entering method: encodeString in class decorators.CoderBean
   INFO: Entering method: codeString in class decorators.CoderImpl
   ```
Persistence

Part VI explores the Java Persistence API. This part contains the following chapters:

- Chapter 32, “Introduction to the Java Persistence API”
- Chapter 33, “Running the Persistence Examples”
- Chapter 34, “The Java Persistence Query Language”
- Chapter 35, “Using the Criteria API to Create Queries”
- Chapter 36, “Creating and Using String-Based Criteria Queries”
- Chapter 37, “Controlling Concurrent Access to Entity Data with Locking”
- Chapter 38, “Using a Second-Level Cache with Java Persistence API Applications”
Introduction to the Java Persistence API

The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications. Java Persistence consists of four areas:

- The Java Persistence API
- The query language
- The Java Persistence Criteria API
- Object/relational mapping metadata

The following topics are addressed here:

- “Entities” on page 579
- "Entity Inheritance” on page 591
- “Managing Entities” on page 595
- "Querying Entities” on page 600
- “Further Information about Persistence” on page 601

Entities

An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented through either persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.
Requirements for Entity Classes

An entity class must follow these requirements.

■ The class must be annotated with the `javax.persistence.Entity` annotation.
■ The class must have a public or protected, no-argument constructor. The class may have other constructors.
■ The class must not be declared `final`. No methods or persistent instance variables must be declared `final`.
■ If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the `Serializable` interface.
■ Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
■ Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

Persistent Fields and Properties in Entity Classes

The persistent state of an entity can be accessed through either the entity's instance variables or properties. The fields or properties must be of the following Java language types:

■ Java primitive types
■ `java.lang.String`
■ Other serializable types, including:
  ■ Wrappers of Java primitive types
  ■ `java.math.BigInteger`
  ■ `java.math.BigDecimal`
  ■ `java.util.Date`
  ■ `java.util.Calendar`
  ■ `java.sql.Date`
  ■ `java.sql.Time`
  ■ `java.sql.TimeStamp`
  ■ User-defined serializable types
  ■ `byte[]`
  ■ `Byte[]`
  ■ `char[]`
  ■ `Character[]`
■ Enumerated types
Entities

- Other entities and/or collections of entities
- Embeddable classes

Entities may use persistent fields, persistent properties, or a combination of both. If the mapping annotations are applied to the entity’s instance variables, the entity uses persistent fields. If the mapping annotations are applied to the entity’s getter methods for JavaBeans-style properties, the entity uses persistent properties.

**Persistent Fields**

If the entity class uses persistent fields, the Persistence runtime accesses entity-class instance variables directly. All fields not annotated `javax.persistence.Transient` or not marked as `Java transient` will be persisted to the data store. The object/relational mapping annotations must be applied to the instance variables.

**Persistent Properties**

If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components. JavaBeans-style properties use getter and setter methods that are typically named after the entity class’s instance variable names. For every persistent property `property` of type `Type` of the entity, there is a getter method `getProperty` and setter method `setProperty`. If the property is a Boolean, you may use `isProperty` instead of `getProperty`. For example, if a customer entity uses persistent properties and has a private instance variable called `firstName`, the class defines a `getFirstName` and `setFirstName` method for retrieving and setting the state of the `firstName` instance variable.

The method signature for single-valued persistent properties are as follows:

```java
Type getProperty()
void setProperty(Type type)
```

The object/relational mapping annotations for persistent properties must be applied to the getter methods. Mapping annotations cannot be applied to fields or properties annotated `@Transient` or marked `transient`.

**Using Collections in Entity Fields and Properties**

Collection-valued persistent fields and properties must use the supported Java collection interfaces regardless of whether the entity uses persistent fields or properties. The following collection interfaces may be used:

- `java.util.Collection`
- `java.util.Set`
If the entity class uses persistent fields, the type in the preceding method signatures must be one of these collection types. Generic variants of these collection types may also be used. For example, if it has a persistent property that contains a set of phone numbers, the `Customer` entity would have the following methods:

```java
Set<PhoneNumber> getPhoneNumbers() { ... }
void setPhoneNumbers(Set<PhoneNumber>) { ... }
```

If a field or property of an entity consists of a collection of basic types or embeddable classes, use the `javax.persistence.ElementCollection` annotation on the field or property.

The two attributes of `@ElementCollection` are `targetClass` and `fetch`. The `targetClass` attribute specifies the class name of the basic or embeddable class and is optional if the field or property is defined using Java programming language generics. The optional `fetch` attribute is used to specify whether the collection should be retrieved lazily or eagerly, using the `javax.persistence FetchType` constants of either `LAZY` or `EAGER`, respectively. By default, the collection will be fetched lazily.

The following entity, `Person`, has a persistent field, `nicknames`, which is a collection of `String` classes that will be fetched eagerly. The `targetClass` element is not required, because it uses generics to define the field.

```java
@Entity
public class Person {
    ...
    @ElementCollection(fetch=EAGER)
    protected Set<String> nickname = new HashSet();
    ...
}
```

Collections of entity elements and relationships may be represented by `java.util.Map` collections. A `Map` consists of a key and a value.

When using `Map` elements or relationships, the following rules apply.

- The `Map` key or value may be a basic Java programming language type, an embeddable class, or an entity.
- When the `Map` value is an embeddable class or basic type, use the `@ElementCollection` annotation.
- When the `Map` value is an entity, use the `@OneToMany` or `@ManyToMany` annotation.
- Use the `Map` type on only one side of a bidirectional relationship.

If the key type of a `Map` is a Java programming language basic type, use the annotation `javax.persistence.MapKeyColumn` to set the column mapping for the key. By default, the name
attribute of @MapKeyColumn is of the form `RELATIONSHIP-FIELD/PROPERTY-NAME_KEY`. For example, if the referencing relationship field name is `image`, the default name attribute is `IMAGE_KEY`.

If the key type of a Map is an entity, use the `javax.persistence.MapKeyJoinColumn` annotation. If the multiple columns are needed to set the mapping, use the annotation `javax.persistence.MapKeyJoinColumns` to include multiple `@MapKeyJoinColumn` annotations. If no `@MapKeyJoinColumn` is present, the mapping column name is by default set to `RELATIONSHIP-FIELD/PROPERTY-NAME_KEY`. For example, if the relationship field name is `employee`, the default name attribute is `EMPLOYEE_KEY`.

If Java programming language generic types are not used in the relationship field or property, the key class must be explicitly set using the `javax.persistence.MapKeyClass` annotation.

If the Map key is the primary key or a persistent field or property of the entity that is the Map value, use the `javax.persistence.MapKey` annotation. The `@MapKeyClass` and `@MapKey` annotations cannot be used on the same field or property.

If the Map value is a Java programming language basic type or an embeddable class, it will be mapped as a collection table in the underlying database. If generic types are not used, the `@ElementCollection` annotation’s `targetClass` attribute must be set to the type of the Map value.

If the Map value is an entity and part of a many-to-many or one-to-many unidirectional relationship, it will be mapped as a join table in the underlying database. A unidirectional one-to-many relationship that uses a Map may also be mapped using the `@JoinColumn` annotation.

If the entity is part of a one-to-many/many-to-one bidirectional relationship, it will be mapped in the table of the entity that represents the value of the Map. If generic types are not used, the `targetEntity` attribute of the `@OneToMany` and `@ManyToMany` annotations must be set to the type of the Map value.

**Validating Persistent Fields and Properties**

The Java API for JavaBeans Validation (Bean Validation) provides a mechanism for validating application data. Bean Validation is integrated into the Java EE containers, allowing the same validation logic to be used in any of the tiers of an enterprise application.

Bean Validation constraints may be applied to persistent entity classes, embeddable classes, and mapped superclasses. By default, the Persistence provider will automatically perform validation on entities with persistent fields or properties annotated with Bean Validation constraints immediately after the `PrePersist`, `PreUpdate`, and `PreRemove` lifecycle events.

Bean Validation constraints are annotations applied to the fields or properties of Java programming language classes. Bean Validation provides a set of constraints as well as an API for defining custom constraints. Custom constraints can be specific combinations of the default
constraints, or new constraints that don’t use the default constraints. Each constraint is associated with at least one validator class that validates the value of the constrained field or property. Custom constraint developers must also provide a validator class for the constraint.

Bean Validation constraints are applied to the persistent fields or properties of persistent classes. When adding Bean Validation constraints, use the same access strategy as the persistent class. That is, if the persistent class uses field access, apply the Bean Validation constraint annotations on the class’s fields. If the class uses property access, apply the constraints on the getter methods.

Table 9–2 lists Bean Validation’s built-in constraints, defined in the javax.validation.constraints package.

All the built-in constraints listed in Table 9–2 have a corresponding annotation, ConstraintName.List, for grouping multiple constraints of the same type on the same field or property. For example, the following persistent field has two @Pattern constraints:

```java
@Pattern.List({
    @Pattern(regex="..."),
    @Pattern(regex="...")
})
```

The following entity class, Contact, has Bean Validation constraints applied to its persistent fields.

```java
@Entity
public class Contact implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp="[a-z0-9!#$%&'*+/=?^_`{|}~\-]+(?:\.|\[a-z0-9!#$%&'*+/=?^_`{|}~\-]+)*@[a-z0-9!#$%&'*+/=?^_`{|}~\-]+(?:\[a-z0-9\](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?",
    message="{invalid.email}")
    protected String email;
    @Pattern(regexp="^[\(]?[\(]?\d{3}[\)]\)?[- ]?\d{3}[\- ]?\d{4}$
        message="{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp="^[\(]?[\(]?\d{3}[\)]\)?[- ]?\d{3}[\- ]?\d{4}$
        message="{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(javax.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
```

The @NotNull annotation on the firstName and lastName fields specifies that those fields are now required. If a new Contact instance is created where firstName or lastName have not been
initialized, Bean Validation will throw a validation error. Similarly, if a previously created instance of Contact has been modified so that firstName or lastName are null, a validation error will be thrown.

The email field has a @Pattern constraint applied to it, with a complicated regular expression that matches most valid email addresses. If the value of email doesn’t match this regular expression, a validation error will be thrown.

The homePhone and mobilePhone fields have the same @Pattern constraints. The regular expression matches 10 digit telephone numbers in the United States and Canada of the form (xxx) xxx–xxxx.

The birthday field is annotated with the @Past constraint, which ensures that the value of birthday must be in the past.

### Primary Keys in Entities

Each entity has a unique object identifier. A customer entity, for example, might be identified by a customer number. The unique identifier, or primary key, enables clients to locate a particular entity instance. Every entity must have a primary key. An entity may have either a simple or a composite primary key.

Simple primary keys use the javax.persistence.Id annotation to denote the primary key property or field.

Composite primary keys are used when a primary key consists of more than one attribute, which corresponds to a set of single persistent properties or fields. Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the javax.persistence.EmbeddedId and javax.persistence.IdClass annotations.

The primary key, or the property or field of a composite primary key, must be one of the following Java language types:

- Java primitive types
- Java primitive wrapper types
- java.lang.String
- java.util.Date (the temporal type should be DATE)
- java.sql.Date
- java.math.BigDecimal
- java.math.BigInteger

Floating-point types should never be used in primary keys. If you use a generated primary key, only integral types will be portable.
A primary key class must meet these requirements.

- The access control modifier of the class must be `public`.
- The properties of the primary key class must be `public` or `protected` if property-based access is used.
- The class must have a public default constructor.
- The class must implement the `hashCode()` and `equals(Object other)` methods.
- The class must be serializable.
- A composite primary key must be represented and mapped to multiple fields or properties of the entity class or must be represented and mapped as an embeddable class.
- If the class is mapped to multiple fields or properties of the entity class, the names and types of the primary key fields or properties in the primary key class must match those of the entity class.

The following primary key class is a composite key, and the `orderId` and `itemId` fields together uniquely identify an entity:

```java
public final class LineItemKey implements Serializable {
    public Integer orderId;
    public int itemId;

    public LineItemKey() {}
    public LineItemKey(Integer orderId, int itemId) {
        this.orderId = orderId;
        this.itemId = itemId;
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return (
            (orderId==null?other.orderId==null:orderId.equals(other.orderId))
            &&
            (itemId == other.itemId)
        );
    }

    public int hashCode() {
        return (
            (orderId==null?0:orderId.hashCode())
            ^
            ((int) itemId)
        );
    }
}
```
public String toString() {
    return "\n    \n    public String toString() {
    return " + orderId + "," + itemId;
    }
    }
related object. If an entity has a related field, the entity is said to “know” about its related object. For example, if Order knows what LineItem instances it has and if LineItem knows what Order it belongs to, they have a bidirectional relationship.

Bidirectional relationships must follow these rules.

- The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation. The mappedBy element designates the property or field in the entity that is the owner of the relationship.
- The many side of many-to-one bidirectional relationships must not define the mappedBy element. The many side is always the owning side of the relationship.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships, either side may be the owning side.

Unidirectional Relationships

In a unidirectional relationship, only one entity has a relationship field or property that refers to the other. For example, LineItem would have a relationship field that identifies Product, but Product would not have a relationship field or property for LineItem. In other words, LineItem knows about Product, but Product doesn’t know which LineItem instances refer to it.

Queries and Relationship Direction

Java Persistence query language and Criteria API queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one entity to another. For example, a query can navigate from LineItem to Product but cannot navigate in the opposite direction. For Order and LineItem, a query could navigate in both directions because these two entities have a bidirectional relationship.

Cascade Operations and Relationships

Entities that use relationships often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order; if the order is deleted, the line item also should be deleted. This is called a cascade delete relationship.

The javax.persistence.CascadeType enumerated type defines the cascade operations that are applied in the cascade element of the relationship annotations. Table 32–1 lists the cascade operations for entities.
TABLE 32–1  Cascade Operations for Entities

<table>
<thead>
<tr>
<th>Cascade Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>All cascade operations will be applied to the parent entity's related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}</td>
</tr>
<tr>
<td>DETACH</td>
<td>If the parent entity is detached from the persistence context, the related entity will also be detached.</td>
</tr>
<tr>
<td>MERGE</td>
<td>If the parent entity is merged into the persistence context, the related entity will also be merged.</td>
</tr>
<tr>
<td>PERSIST</td>
<td>If the parent entity is persisted into the persistence context, the related entity will also be persisted.</td>
</tr>
<tr>
<td>REFRESH</td>
<td>If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.</td>
</tr>
<tr>
<td>REMOVE</td>
<td>If the parent entity is removed from the current persistence context, the related entity will also be removed.</td>
</tr>
</tbody>
</table>

Cascade delete relationships are specified using the cascade=REMOVE element specification for @OneToOne and @OneToMany relationships. For example:

```java
@OneToMany(mappedBy="customer", cascade=REMOVE)
public Set<Order> getOrders() { return orders; }
```

**Orphan Removal in Relationships**

When a target entity in one-to-one or one-to-many relationship is removed from the relationship, it is often desirable to cascade the remove operation to the target entity. Such target entities are considered "orphans," and the orphanRemoval attribute can be used to specify that orphaned entities should be removed. For example, if an order has many line items and one of them is removed from the order, the removed line item is considered an orphan. If orphanRemoval is set to true, the line item entity will be deleted when the line item is removed from the order.

The orphanRemoval attribute in @OneToMany and @OneToOne takes a Boolean value and is by default false.

The following example will cascade the remove operation to the orphaned order entity when the customer entity is deleted:

```java
@OneToOne(mappedBy="customer", orphanRemoval=true)
public List<Order> getOrders() { ... }
```
Embeddable Classes in Entities

Embeddable classes are used to represent the state of an entity but don’t have a persistent identity of their own, unlike entity classes. Instances of an embeddable class share the identity of the entity that owns it. Embeddable classes exist only as the state of another entity. An entity may have single-valued or collection-valued embeddable class attributes.

Embeddable classes have the same rules as entity classes but are annotated with the javax.persistence.Embeddable annotation instead of @Entity.

The following embeddable class, ZipCode, has the fields zip and plusFour:

```java
@Embeddable
public class ZipCode {
    String zip;
    String plusFour;
    ...
}
```

This embeddable class is used by the Address entity:

```java
@Entity
public class Address {
    @Id
    protected long id
    String street1;
    String street2;
    String city;
    String province;
    @Embedded
    ZipCode zipCode;
    String country;
    ...
}
```

Entities that own embeddable classes as part of their persistent state may annotate the field or property with the javax.persistence.Embedded annotation but are not required to do so.

Embeddable classes may themselves use other embeddable classes to represent their state. They may also contain collections of basic Java programming language types or other embeddable classes. Embeddable classes may also contain relationships to other entities or collections of entities. If the embeddable class has such a relationship, the relationship is from the target entity or collection of entities to the entity that owns the embeddable class.
Entity Inheritance

Entities support class inheritance, polymorphic associations, and polymorphic queries. Entity classes can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.

The roster example application demonstrates entity inheritance, as described in “Entity Inheritance in the roster Application” on page 617.

Abstract Entities

An abstract class may be declared an entity by decorating the class with @Entity. Abstract entities are like concrete entities but cannot be instantiated.

Abstract entities can be queried just like concrete entities. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity:

```java
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information but are not entities. That is, the superclass is not decorated with the @Entity annotation and is not mapped as an entity by the Java Persistence provider. These superclasses are most often used when you have state and mapping information common to multiple entity classes.

Mapped superclasses are specified by decorating the class with the annotation javax.persistence.MappedSuperclass:

```java
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
}
```
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}

@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}

Mapped superclasses cannot be queried and can’t be used in EntityManager or Query operations. You must use entity subclasses of the mapped superclass in EntityManager or Query operations. Mapped superclasses can’t be targets of entity relationships. Mapped superclasses can be abstract or concrete.

Mapped superclasses do not have any corresponding tables in the underlying data store. Entities that inherit from the mapped superclass define the table mappings. For instance, in the preceding code sample, the underlying tables would be FULLTIMEEMPLOYEE and PARTTIMEEMPLOYEE, but there is no EMPLOYEE table.

Non-Entity Superclasses

Entities may have non-entity superclasses, and these superclasses can be either abstract or concrete. The state of non-entity superclasses is nonpERSISTent, and any state inherited from the non-entity superclass by an entity class is nonpERSISTent. Non-entity superclasses may not be used in EntityManager or Query operations. Any mapping or relationship annotations in non-entity superclasses are ignored.

Entity Inheritance Mapping Strategies

You can configure how the Java Persistence provider maps inherited entities to the underlying data store by decorating the root class of the hierarchy with the annotation javax.persistence.Inheritance. The following mapping strategies are used to map the entity data to the underlying database:

- A single table per class hierarchy
- A table per concrete entity class
- A “join” strategy, whereby fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class

The strategy is configured by setting the strategy element of @Inheritance to one of the options defined in the javax.persistence.InheritanceType enumerated type.
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};

The default strategy, InheritanceType.SINGLE_TABLE, is used if the @Inheritance annotation is not specified on the root class of the entity hierarchy.

The Single Table per Class Hierarchy Strategy

With this strategy, which corresponds to the default InheritanceType.SINGLE_TABLE, all classes in the hierarchy are mapped to a single table in the database. This table has a discriminator column containing a value that identifies the subclass to which the instance represented by the row belongs.

The discriminator column, whose elements are shown in Table 32–2, can be specified by using the javax.persistence.DiscriminatorColumn annotation on the root of the entity class hierarchy.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>name</td>
<td>The name of the column to be used as the discriminator column. The default is DTYPE. This element is optional.</td>
</tr>
<tr>
<td>DiscriminatorType</td>
<td>discriminatorType</td>
<td>The type of the column to be used as a discriminator column. The default is DiscriminatorType.STRING. This element is optional.</td>
</tr>
<tr>
<td>String</td>
<td>columnDefinition</td>
<td>The SQL fragment to use when creating the discriminator column. The default is generated by the Persistence provider and is implementation-specific. This element is optional.</td>
</tr>
<tr>
<td>String</td>
<td>length</td>
<td>The column length for String-based discriminator types. This element is ignored for non-String discriminator types. The default is 31. This element is optional.</td>
</tr>
</tbody>
</table>

The javax.persistence.DiscriminatorType enumerated type is used to set the type of the discriminator column in the database by setting the discriminatorType element of @DiscriminatorColumn to one of the defined types. DiscriminatorType is defined as:

public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
If `@DiscriminatorColumn` is not specified on the root of the entity hierarchy and a discriminator column is required, the Persistence provider assumes a default column name of `DTYPE` and column type of `DiscriminatorType.STRING`.

The `javax.persistence.DiscriminatorValue` annotation may be used to set the value entered into the discriminator column for each entity in a class hierarchy. You may decorate only concrete entity classes with `@DiscriminatorValue`.

If `@DiscriminatorValue` is not specified on an entity in a class hierarchy that uses a discriminator column, the Persistence provider will provide a default, implementation-specific value. If the `discriminatorType` element of `@DiscriminatorColumn` is `DiscriminatorType.STRING`, the default value is the name of the entity.

This strategy provides good support for polymorphic relationships between entities and queries that cover the entire entity class hierarchy. However, this strategy requires the columns that contain the state of subclasses to be nullable.

**The Table per Concrete Class Strategy**

In this strategy, which corresponds to `InheritanceType.TABLE_PER_CLASS`, each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the class's table in the database.

This strategy provides poor support for polymorphic relationships and usually requires either SQL UNION queries or separate SQL queries for each subclass for queries that cover the entire entity class hierarchy.

Support for this strategy is optional and may not be supported by all Java Persistence API providers. The default Java Persistence API provider in the GlassFish Server does not support this strategy.

**The Joined Subclass Strategy**

In this strategy, which corresponds to `InheritanceType.JOINED`, the root of the class hierarchy is represented by a single table, and each subclass has a separate table that contains only those fields specific to that subclass. That is, the subclass table does not contain columns for inherited fields or properties. The subclass table also has a column or columns that represent its primary key, which is a foreign key to the primary key of the superclass table.

This strategy provides good support for polymorphic relationships but requires one or more join operations to be performed when instantiating entity subclasses. This may result in poor performance for extensive class hierarchies. Similarly, queries that cover the entire class hierarchy require join operations between the subclass tables, resulting in decreased performance.

Some Java Persistence API providers, including the default provider in the GlassFish Server, require a discriminator column that corresponds to the root entity when using the joined subclass strategy. If you are not using automatic table creation in your application, make sure
that the database table is set up correctly for the discriminator column defaults, or use the 
@DiscriminatorColumn annotation to match your database schema. For information on 
discriminator columns, see “The Single Table per Class Hierarchy Strategy” on page 593.

Managing Entities

Entities are managed by the entity manager, which is represented by 
javax.persistence.EntityManager instances. Each EntityManager instance is associated 
with a persistence context: a set of managed entity instances that exist in a particular data store. 
A persistence context defines the scope under which particular entity instances are created, 
persisted, and removed. The EntityManager interface defines the methods that are used to 
interact with the persistence context.

The EntityManager Interface

The EntityManager API creates and removes persistent entity instances, finds entities by the 
entity’s primary key, and allows queries to be run on entities.

Container-Managed Entity Managers

With a container-managed entity manager, an EntityManager instance’s persistence context is 
automatically propagated by the container to all application components that use the 
EntityManager instance within a single Java Transaction API (JTA) transaction.

JTA transactions usually involve calls across application components. To complete a JTA 
transaction, these components usually need access to a single persistence context. This occurs 
when an EntityManager is injected into the application components by means of the 
javax.persistence.PersistenceContext annotation. The persistence context is 
automatically propagated with the current JTA transaction, and EntityManager references that 
are mapped to the same persistence unit provide access to the persistence context within that 
transaction. By automatically propagating the persistence context, application components 
don’t need to pass references to EntityManager instances to each other in order to make 
changes within a single transaction. The Java EE container manages the lifecycle of 
container-managed entity managers.

To obtain an EntityManager instance, inject the entity manager into the application 
component:

@PersistenceContext 
EntityManager em;
Application-Managed Entity Managers

With an application-managed entity manager, on the other hand, the persistence context is not propagated to application components, and the lifecycle of EntityManager instances is managed by the application.

Application-managed entity managers are used when applications need to access a persistence context that is not propagated with the JTA transaction across EntityManager instances in a particular persistence unit. In this case, each EntityManager creates a new, isolated persistence context. The EntityManager and its associated persistence context are created and destroyed explicitly by the application. They are also used when directly injecting EntityManager instances can't be done because EntityManager instances are not thread-safe. EntityManagerFactory instances are thread-safe.

Applications create EntityManager instances in this case by using the createEntityManager method of javax.persistence.EntityManagerFactory.

To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the javax.persistence.PersistenceUnit annotation:

```java
@PersistenceUnit
EntityManagerFactory emf;

Then obtain an EntityManager from the EntityManagerFactory instance:

```java
EntityManager em = emf.createEntityManager();
```

Application-managed entity managers don't automatically propagate the JTA transaction context. Such applications need to manually gain access to the JTA transaction manager and add transaction demarcation information when performing entity operations. The javax.transaction.UserTransaction interface defines methods to begin, commit, and rollback transactions. Inject an instance of UserTransaction by creating an instance variable annotated with @Resource:

```java
@Resource
UserTransaction utx;
```

To begin a transaction, call the UserTransaction.begin method. When all the entity operations are complete, call the UserTransaction.commit method to commit the transaction. The UserTransaction.rollback method is used to rollback the current transaction.

The following example shows how to manage transactions in an application that uses an application-managed entity manager:

```java
@PersistenceContext
EntityManagerFactory emf;
EntityManager em;
```
Finding Entities Using the EntityManager

The EntityManager.find method is used to look up entities in the data store by the entity’s primary key:

```java
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
    utx.begin();
    em.persist(SomeEntity);
    em.merge(AnotherEntity);
    em.remove(ThirdEntity);
    utx.commit();
} catch (Exception e) {
    utx.rollback();
}
```

Managing an Entity Instance’s Lifecycle

You manage entity instances by invoking operations on the entity by means of an EntityManager instance. Entity instances are in one of four states: new, managed, detached, or removed.

- New entity instances have no persistent identity and are not yet associated with a persistence context.
- Managed entity instances have a persistent identity and are associated with a persistence context.
- Detached entity instances have a persistent identity and are not currently associated with a persistence context.
- Removed entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

Persisting Entity Instances

New entity instances become managed and persistent either by invoking the persist method or by a cascading persist operation invoked from related entities that have the cascade=PERSIST or cascade=ALL elements set in the relationship annotation. This means that the entity’s data is stored to the database when the transaction associated with the persist operation is completed. If the entity is already managed, the persist operation is ignored.
although the persist operation will cascade to related entities that have the cascade element set to PERSIST or ALL in the relationship annotation. If persist is called on a removed entity instance, the entity becomes managed. If the entity is detached, either persist will throw an IllegalArgumentException, or the transaction commit will fail.

```java
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order order, Product product, int quantity) {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}
```

The persist operation is propagated to all entities related to the calling entity that have the cascade element set to ALL or PERSIST in the relationship annotation:

```java
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

**Removing Entity Instances**

Managed entity instances are removed by invoking the remove method or by a cascading remove operation invoked from related entities that have the cascade=REMOVE or cascade=ALL elements set in the relationship annotation. If the remove method is invoked on a new entity, the remove operation is ignored, although remove will cascade to related entities that have the cascade element set to REMOVE or ALL in the relationship annotation. If remove is invoked on a detached entity, either remove will throw an IllegalArgumentException, or the transaction commit will fail. If invoked on an already removed entity, remove will be ignored. The entity's data will be removed from the data store when the transaction is completed or as a result of the flush operation.

```java
public void removeOrder(Integer orderId) {
    try {
        Order order = em.find(Order.class, orderId);
        em.remove(order);
    }...
}
```

In this example, all LineItem entities associated with the order are also removed, as Order.getLineItems has cascade=ALL set in the relationship annotation.

**Synchronizing Entity Data to the Database**

The state of persistent entities is synchronized to the database when the transaction with which the entity is associated commits. If a managed entity is in a bidirectional relationship with another managed entity, the data will be persisted, based on the owning side of the relationship.
To force synchronization of the managed entity to the data store, invoke the `flush` method of the `EntityManager` instance. If the entity is related to another entity and the relationship annotation has the `cascade` element set to `PERSIST` or `ALL`, the related entity's data will be synchronized with the data store when `flush` is called.

If the entity is removed, calling `flush` will remove the entity data from the data store.

**Persistence Units**

A persistence unit defines a set of all entity classes that are managed by `EntityManager` instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the `persistence.xml` configuration file. The following is an example `persistence.xml` file:

```xml
<persistence>
  <persistence-unit name="OrderManagement">
    <description>This unit manages orders and customers. It does not rely on any vendor-specific features and can therefore be deployed to any persistence provider.</description>
    <jta-data-source>jdbc/MyOrderDB</jta-data-source>
    <jar-file>MyOrderApp.jar</jar-file>
    <class>com.widgets.Order</class>
    <class>com.widgets.Customer</class>
  </persistence-unit>
</persistence>
```

This file defines a persistence unit named `OrderManagement`, which uses a JTA-aware data source: `jdbc/MyOrderDB`. The `jar-file` and `class` elements specify managed persistence classes: entity classes, embeddable classes, and mapped superclasses. The `jar-file` element specifies JAR files that are visible to the packaged persistence unit that contain managed persistence classes, whereas the `class` element explicitly names managed persistence classes.

The `jta-data-source` (for JTA-aware data sources) and `non-jta-data-source` (for non-JTA-aware data sources) elements specify the global JNDI name of the data source to be used by the container.

The JAR file or directory whose `META-INF` directory contains `persistence.xml` is called the root of the persistence unit. The scope of the persistence unit is determined by the persistence unit’s root. Each persistence unit must be identified with a name that is unique to the persistence unit’s scope.
Persistent units can be packaged as part of a WAR or EJB JAR file or can be packaged as a JAR file that can then be included in a WAR or EAR file.

- If you package the persistent unit as a set of classes in an EJB JAR file, persistence.xml should be put in the EJB JAR's META-INF directory.
- If you package the persistence unit as a set of classes in a WAR file, persistence.xml should be located in the WAR file's WEB-INF/classes/META-INF directory.
- If you package the persistence unit in a JAR file that will be included in a WAR or EAR file, the JAR file should be located in either
  - The WEB-INF/lib directory of a WAR
  - The EAR file's library directory

Note – In the Java Persistence API 1.0, JAR files could be located at the root of an EAR file as the root of the persistence unit. This is no longer supported. Portable applications should use the EAR file's library directory as the root of the persistence unit.

**Querying Entities**

The Java Persistence API provides the following methods for querying entities.

- The Java Persistence query language (JPQL) is a simple, string-based language similar to SQL used to query entities and their relationships. See Chapter 34, “The Java Persistence Query Language,” for more information.
- The Criteria API is used to create typesafe queries using Java programming language APIs to query for entities and their relationships. See Chapter 35, “Using the Criteria API to Create Queries,” for more information.

Both JPQL and the Criteria API have advantages and disadvantages.

Just a few lines long, JPQL queries are typically more concise and more readable than Criteria queries. Developers familiar with SQL will find it easy to learn the syntax of JPQL. JPQL named queries can be defined in the entity class using a Java programming language annotation or in the application’s deployment descriptor. JPQL queries are not typesafe, however, and require a cast when retrieving the query result from the entity manager. This means that type-casting errors may not be caught at compile time. JPQL queries don’t support open-ended parameters.

Criteria queries allow you to define the query in the business tier of the application. Although this is also possible using JPQL dynamic queries, Criteria queries provide better performance because JPQL dynamic queries must be parsed each time they are called. Criteria queries are typesafe and therefore don’t require casting, as JPQL queries do. The Criteria API is just another Java programming language API and doesn’t require developers to learn the syntax of another
query language. Criteria queries are typically more verbose than JPQL queries and require the developer to create several objects and perform operations on those objects before submitting the query to the entity manager.

Further Information about Persistence

For more information about the Java Persistence API, see

- Java Persistence 2.0 API specification:  
- EclipseLink, the Java Persistence API implementation in the GlassFish Server:  
  http://www.eclipse.org/eclipselink/jpa.php
- EclipseLink team blog:  
  http://eclipselink.blogspot.com/
- EclipseLink wiki documentation:  
  http://wiki.eclipse.org/EclipseLink
Running the Persistence Examples

This chapter explains how to use the Java Persistence API. The material here focuses on the source code and settings of three examples. The first example, order, is an application that uses a stateful session bean to manage entities related to an ordering system. The second example, roster, is an application that manages a community sports system. The third example, address-book, is a web application that stores contact data. This chapter assumes that you are familiar with the concepts detailed in Chapter 32, “Introduction to the Java Persistence API.”

The following topics are addressed here:

- “The order Application” on page 603
- “The roster Application” on page 615
- “The address-book Application” on page 623

The order Application

The order application is a simple inventory and ordering application for maintaining a catalog of parts and placing an itemized order of those parts. The application has entities that represent parts, vendors, orders, and line items. These entities are accessed using a stateful session bean that holds the business logic of the application. A simple singleton session bean creates the initial entities on application deployment. A Facelets web application manipulates the data and displays data from the catalog.

The information contained in an order can be divided into elements. What is the order number? What parts are included in the order? What parts make up that part? Who makes the part? What are the specifications for the part? Are there any schematics for the part? The order application is a simplified version of an ordering system that has all these elements.

The order application consists of a single WAR module that includes the enterprise bean classes, the entities, the support classes, and the Facelets XHTML and class files.

The database schema in the Java DB database for order is shown in Figure 33–1.
Entity Relationships in the order Application

The order application demonstrates several types of entity relationships: self-referential, one-to-one, one-to-many, many-to-one, and unidirectional relationships.

Note – In this diagram, for simplicity, the PERSISTENCE_ORDER_ prefix is omitted from the table names.
Self-Referential Relationships

A self-referential relationship occurs between relationship fields in the same entity. Part has a field, bomPart, which has a one-to-many relationship with the field parts, which is also in Part. That is, a part can be made up of many parts, and each of those parts has exactly one bill-of-material part.

The primary key for Part is a compound primary key, a combination of the partNumber and revision fields. This key is mapped to the PARTNUMBER and REVISION columns in the EJB_ORDER_PART table:

```java
@ManyToOne
@JoinColumns({
    @JoinColumn(name="BOMPARTNUMBER",
               referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="BOMREVISION",
               referencedColumnName="REVISION")
})
public Part getBomPart() {
    return bomPart;
}

@OneToMany(mappedBy="bomPart")
public Collection<Part> getParts() {
    return parts;
}
```

One-to-One Relationships

Part has a field, vendorPart, that has a one-to-one relationship with VendorPart's part field. That is, each part has exactly one vendor part, and vice versa.

Here is the relationship mapping in Part:

```java
@OneToOne(mappedBy="part")
public VendorPart getVendorPart() {
    return vendorPart;
}
```

Here is the relationship mapping in VendorPart:

```java
@OneToOne
@JoinColumn({
    @JoinColumn(name="PARTNUMBER",
               referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="PARTREVISION",
               referencedColumnName="REVISION")
})
public Part getPart() {
    return part;
}
```
Note that, because Part uses a compound primary key, the @JoinColumns annotation is used to map the columns in the PERSISTENCE_ORDER_VENDOR_PART table to the columns in PERSISTENCE_ORDER_PART. The PERSISTENCE_ORDER_VENDOR_PART table’s PARTREVISION column refers to PERSISTENCE_ORDER_PART’s REVISION column.

**One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys**

Order has a field, lineItems, that has a one-to-many relationship with LineItem’s field order. That is, each order has one or more line item.

LineItem uses a compound primary key that is made up of the orderId and itemId fields. This compound primary key maps to the ORDERID and ITEMID columns in the PERSISTENCE_ORDER_LINEITEM table. ORDERID is a foreign key to the ORDERID column in the PERSISTENCE_ORDER_ORDER table. This means that the ORDERID column is mapped twice: once as a primary key field, orderId; and again as a relationship field, order.

Here is the relationship mapping in Order:

```java
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Here is the relationship mapping in LineItem:

```java
@ManyToOne
public Order getOrder() {
    return order;
}
```

**Unidirectional Relationships**

LineItem has a field, vendorPart, that has a unidirectional many-to-one relationship with VendorPart. That is, there is no field in the target entity in this relationship:

```java
@ManyToOne
public VendorPart getVendorPart() {
    return vendorPart;
}
```

**Primary Keys in the order Application**

The order application uses several types of primary keys: single-valued primary keys, compound primary keys, and generated primary keys.
Generated Primary Keys

VendorPart uses a generated primary key value. That is, the application does not assign primary key values for the entities but instead relies on the persistence provider to generate the primary key values. The @GeneratedValue annotation is used to specify that an entity will use a generated primary key.

In VendorPart, the following code specifies the settings for generating primary key values:

```java
@TableGenerator(
    name="vendorPartGen",
    table="PERSISTENCE_ORDER_SEQUENCE_GENERATOR",
    pkColumnName="GEN_KEY",
    valueColumnName="GEN_VALUE",
    pkColumnValue="VENDOR_PART_ID",
    allocationSize=10)
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
generator="vendorPartGen")
public Long getVendorPartNumber() {
    return vendorPartNumber;
}
```

The @TableGenerator annotation is used in conjunction with @GeneratedValue's strategy=TABLE element. That is, the strategy used to generate the primary keys is to use a table in the database. The @TableGenerator annotation is used to configure the settings for the generator table. The name element sets the name of the generator, which is vendorPartGen in VendorPart.

The EJB_ORDER_SEQUENCE_GENERATOR table, whose two columns are GEN_KEY and GEN_VALUE, will store the generated primary key values. This table could be used to generate other entity's primary keys, so the pkColumnValue element is set to VENDOR_PART_ID to distinguish this entity's generated primary keys from other entity's generated primary keys. The allocationSize element specifies the amount to increment when allocating primary key values. In this case, each VendorPart's primary key will increment by 10.

The primary key field vendorPartNumber is of type Long, as the generated primary key's field must be an integral type.

Compound Primary Keys

A compound primary key is made up of multiple fields and follows the requirements described in “Primary Keys in Entities” on page 585. To use a compound primary key, you must create a wrapper class.
In order, two entities use compound primary keys: Part and LineItem.

- Part uses the PartKey wrapper class. Part’s primary key is a combination of the part number and the revision number. PartKey encapsulates this primary key.
- LineItem uses the LineItemKey class. LineItem’s primary key is a combination of the order number and the item number. LineItemKey encapsulates this primary key.

This is the LineItemKey compound primary key wrapper class:

```java
package order.entity;

public final class LineItemKey implements java.io.Serializable {
    private Integer orderId;
    private int itemId;

    public int hashCode() {
        return ((this.getOrderId()==null ?0:this.getOrderId().hashCode())
            ^ ((int) this.getItemId()));
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getOrderId()==null
                        ?other.orderId==null:this.getOrderId().equals
                        (other.orderId)) && (this.getItemId ==
                        other.itemId));
    }

    public String toString() {
        return "" + orderId + "." + itemId;
    }
}
```

The @IdClass annotation is used to specify the primary key class in the entity class. In LineItem, @IdClass is used as follows:

```java
@IdClass(order.entity.LineItemKey.class)
@Entity
... public class LineItem {
...
```

The two fields in LineItem are tagged with the @Id annotation to mark those fields as part of the compound primary key:
For orderId, you also use the @Column annotation to specify the column name in the table and that this column should not be inserted or updated, as it is an overlapping foreign key pointing at the PERSISTENCE_ORDER_ORDER table's ORDERID column (see "One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys" on page 606). That is, orderId will be set by the Order entity.

In LineItem's constructor, the line item number (LineItem.itemId) is set using the Order.getNextId method:

```java
public LineItem(Order order, int quantity, VendorPart vendorPart) {
    this.order = order;
    this.itemId = order.getNextId();
    this.orderId = order.getOrderId();
    this.quantity = quantity;
    this.vendorPart = vendorPart;
}
```

Order.getNextId counts the number of current line items, adds 1, and returns that number:

```java
public int getNextId() {
    return this.lineItems.size() + 1;
}
```

Part doesn't require the @Column annotation on the two fields that comprise Part's compound primary key, because Part's compound primary key is not an overlapping primary key/foreign key:

```java
@IdClass(order.entity.PartKey.class)
@Entity
... public class Part {
    ... @Id
    public String getPartNumber() {
        return partNumber;
    }
    ... @Id
    public int getRevision() {
        return revision;
    }
```
Entity Mapped to More Than One Database Table

Part's fields map to more than one database table: PERSISTENCE_ORDER_PART and PERSISTENCE_ORDER_PART_DETAIL. The PERSISTENCE_ORDER_PART_DETAIL table holds the specification and schematics for the part. The @SecondaryTable annotation is used to specify the secondary table.

```java
@Entity
@Table(name="PERSISTENCE_ORDER_PART")
@SecondaryTable(name="PERSISTENCE_ORDER_PART_DETAIL", pkJoinColumns={
    @PrimaryKeyJoinColumn(name="PARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @PrimaryKeyJoinColumn(name="REVISION",
        referencedColumnName="REVISION")
})
public class Part {
    ...}
```

PERSISTENCE_ORDER_PART_DETAIL and PERSISTENCE_ORDER_PART share the same primary key values. The pkJoinColumns element of @SecondaryTable is used to specify that PERSISTENCE_ORDER_PART_DETAIL’s primary key columns are foreign keys to PERSISTENCE_ORDER_PART. The @PrimaryKeyJoinColumn annotation sets the primary key column names and specifies which column in the primary table the column refers to. In this case, the primary key column names for both PERSISTENCE_ORDER_PART_DETAIL and PERSISTENCE_ORDER_PART are the same: PARTNUMBER and REVISION, respectively.

Cascade Operations in the order Application

Entities that have relationships to other entities often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order; if the order is deleted, then the line item also should be deleted. This is called a cascade delete relationship.

In order, there are two cascade delete dependencies in the entity relationships. If the Order to which a LineItem is related is deleted, the LineItem also should be deleted. If the Vendor to which a VendorPart is related is deleted, the VendorPart also should be deleted.

You specify the cascade operations for entity relationships by setting the cascade element in the inverse (nonowning) side of the relationship. The cascade element is set to ALL in the case of Order . LineItems. This means that all persistence operations (deletes, updates, and so on) are cascaded from orders to line items.

Here is the relationship mapping in Order:
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}

Here is the relationship mapping in LineItem:

@ManyToOne
    public Order getOrder() {
        return order;
    }

BLOB and CLOB Database Types in the order Application

The PARTDETAIL table in the database has a column, DRAWING, of type BLOB. BLOB stands for binary large objects, which are used for storing binary data, such as an image. The DRAWING column is mapped to the field Part.drawing of type java.io.Serializable. The @Lob annotation is used to denote that the field is large object.

@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public Serializable getDrawing() {
    return drawing;
}

PERSISTENCE_ORDER_PART_DETAIL also has a column, SPECIFICATION, of type CLOB. CLOB stands for character large objects, which are used to store string data too large to be stored in a VARCHAR column. SPECIFICATION is mapped to the field Part.specification of type java.lang.String. The @Lob annotation is also used here to denote that the field is a large object.

@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public String getSpecification() {
    return specification;
}

Both of these fields use the @Column annotation and set the table element to the secondary table.
Temporal Types in the order Application

The Order.lastUpdate persistent property, which is of type java.util.Date, is mapped to the PERSISTENCE.ORDER.ORDER.LASTUPDATE database field, which is of the SQL type TIMESTAMP. To ensure the proper mapping between these types, you must use the @Temporal annotation with the proper temporal type specified in @Temporal's element. @Temporal's elements are of type javax.persistence.TemporalType. The possible values are:

- DATE, which maps to java.sql.Date
- TIME, which maps to java.sql.Time
- TIMESTAMP, which maps to java.sql.Timestamp

Here is the relevant section of Order:

```java
@Temporal(TIMESTAMP)
public Date getLastUpdate() {
    return lastUpdate;
}
```

Managing the order Application’s Entities

The RequestBean stateful session bean contains the business logic and manages the entities of order. RequestBean uses the @PersistenceContext annotation to retrieve an entity manager instance, which is used to manage order’s entities in RequestBean’s business methods:

```java
@PersistenceContext
private EntityManager em;
```

This EntityManager instance is a container-managed entity manager, so the container takes care of all the transactions involved in the managing order’s entities.

Creating Entities

The RequestBean.createPart business method creates a new Part entity. The EntityManager.persist method is used to persist the newly created entity to the database.

```java
Part part = new Part(partNumber,
        revision,
        description,
        revisionDate,
        specification,
        drawing);
em.persist(part);
```

The ConfigBean singleton session bean is used to initialize the data in order. ConfigBean is annotated with @Startup, which indicates that the EJB container should create ConfigBean when order is deployed. The createData method is annotated with @PostConstruct and creates the initial entities used by order by calling RequestBean’s business methods.
Finding Entities

The RequestBean.getOrderPrice business method returns the price of a given order, based on the orderId. The EntityManager.find method is used to retrieve the entity from the database.

```
Order order = em.find(Order.class, orderId);
```

The first argument of EntityManager.find is the entity class, and the second is the primary key.

Setting Entity Relationships

The RequestBean.createVendorPart business method creates a VendorPart associated with a particular Vendor. The EntityManager.persist method is used to persist the newly created VendorPart entity to the database, and the VendorPart.setVendor and Vendor.setVendorPart methods are used to associate the VendorPart with the Vendor.

```
PartKey pkey = new PartKey();
pkey.partNumber = partNumber;
pkey.revision = revision;

Part part = em.find(Part.class, pkey);
VendorPart vendorPart = new VendorPart(description, price, part);
em.persist(vendorPart);
Vendor vendor = em.find(Vendor.class, vendorId);
vendor.addVendorPart(vendorPart);
vendorPart.setVendor(vendor);
```

Using Queries

The RequestBean.adjustOrderDiscount business method updates the discount applied to all orders. This method uses the findAllOrders named query, defined in Order:

```
@NamedQuery(
    name="findAllOrders",
    query="SELECT o FROM Order o"
)
```

The EntityManager.createNamedQuery method is used to run the query. Because the query returns a List of all the orders, the Query.getResultList method is used.

```
List orders = em.createNamedQuery("findAllOrders")
    .getResultList();
```

The RequestBean.getTotalPricePerVendor business method returns the total price of all the parts for a particular vendor. This method uses a named parameter, id, defined in the named query findTotalVendorPartPricePerVendor defined in VendorPart.
@NamedQuery(
    name="findTotalVendorPartPricePerVendor",
    query="SELECT SUM(vp.price) " + 
    "FROM VendorPart vp " + 
    "WHERE vp.vendor.vendorId = :id"
)

When running the query, the Query.setParameter method is used to set the named parameter `id` to the value of `vendorId`, the parameter to RequestBean.getTotalPricePerVendor:

```java
return (Double) em.createNamedQuery("findTotalVendorPartPricePerVendor")
 .setParameter("id", vendorId)
 .getSingleResult();
```

The Query.getSingleResult method is used for this query because the query returns a single value.

### Removing Entities

The RequestBean.removeOrder business method deletes a given order from the database. This method uses the EntityManager.remove method to delete the entity from the database.

```java
Order order = em.find(Order.class, orderId);
em.remove(order);
```

### Running the order Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the order application. First, you will create the database tables in the Java DB server.

▶ **To Run the order Example Using NetBeans IDE**

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/persistence/`

3. Select the order folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the order project and select Run.
To Run the order Example Using Ant

1. In a terminal window, go to:
   `tut-install/examples/persistence/order/`

2. Type the following command:
   ```
   ant
   ```
   This runs the `default` task, which compiles the source files and packages the application into a
   WAR file located at `tut-install/examples/persistence/order/dist/order.war`.

3. To deploy the WAR, make sure that the GlassFish Server is started, then type the following
   command:
   ```
   ant deploy
   ```

4. Open a web browser to `http://localhost:8080/order/` to create and update the order data.

The all Task

As a convenience, the `all` task will build, package, deploy, and run the application. To do this,

   ```
   ant all
   ```

The roster Application

The roster application maintains the team rosters for players in recreational sports leagues. The
application has four components: Java Persistence API entities (`Player`, `Team`, and `League`),
a stateful session bean (`RequestBean`), an application client (`RosterClient`), and three helper
classes (`PlayerDetails`, `TeamDetails`, and `LeagueDetails`).

Functionally, roster is similar to the order application, with three new features that order
does not have: many-to-many relationships, entity inheritance, and automatic table creation at
deployment time.

The database schema in the underlying Java DB database for roster is shown in Figure 33-2.
Note – In this diagram, for simplicity, the PERSISTENCE_ROSTER_ prefix is omitted from the table names.

Relationships in the roster Application

A recreational sports system has the following relationships:

- A player can be on many teams.
- A team can have many players.
- A team is in exactly one league.
- A league has many teams.

In roster this system is reflected by the following relationships between the Player, Team, and League entities:

- There is a many-to-many relationship between Player and Team.
- There is a many-to-one relationship between Team and League.
The Many-To-Many Relationship in roster

The many-to-many relationship between Player and Team is specified by using the @ManyToMany annotation. In Team.java, the @ManyToMany annotation decorates the getPlayers method:

```java
@ManyToMany
@JoinTable(
    name="EJB_ROSTER_TEAM_PLAYER",
    joinColumns=@JoinColumn(name="TEAM_ID", referencedColumnName="ID"),
    inverseJoinColumns=@JoinColumn(name="PLAYER_ID", referencedColumnName="ID"))
public Collection<Player> getPlayers() {
    return players;
}
```

The @JoinTable annotation is used to specify a database table that will associate player IDs with team IDs. The entity that specifies the @JoinTable is the owner of the relationship, so the Team entity is the owner of the relationship with the Player entity. Because roster uses automatic table creation at deployment time, the container will create a joint table named EJB_ROSTER_TEAM_PLAYER.

Player is the inverse, or nonowning, side of the relationship with Team. As one-to-one and many-to-one relationships, the nonowning side is marked by the mappedBy element in the relationship annotation. Because the relationship between Player and Team is bidirectional, the choice of which entity is the owner of the relationship is arbitrary.

In Player.java, the @ManyToMany annotation decorates the getTeams method:

```java
@ManyToMany(mappedBy="players")
public Collection<Team> getTeams() {
    return teams;
}
```

Entity Inheritance in the roster Application

The roster application shows how to use entity inheritance, as described in “Entity Inheritance” on page 591.

The League entity in roster is an abstract entity with two concrete subclasses: SummerLeague and WinterLeague. Because League is an abstract class, it cannot be instantiated:

```java
...  
@Entity  
@Table(name = "EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...  
}  
```
Instead, when creating a league, clients use `SummerLeague` or `WinterLeague`. `SummerLeague` and `WinterLeague` inherit the persistent properties defined in `League` and add only a constructor that verifies that the sport parameter matches the type of sport allowed in that seasonal league. For example, here is the `SummerLeague` entity:

```java
@Entity
public class SummerLeague extends League
    implements java.io.Serializable {

    /** Creates a new instance of SummerLeague */
    public SummerLeague() {
    }

    public SummerLeague(String id, String name,
        String sport) throws IncorrectSportException {
        this.id = id;
        this.name = name;
        if (sport.equalsIgnoreCase("swimming") ||
            sport.equalsIgnoreCase("soccer") ||
            sport.equalsIgnoreCase("basketball") ||
            sport.equalsIgnoreCase("baseball")) {
            this.sport = sport;
        } else {
            throw new IncorrectSportException(
                "Sport is not a summer sport.");
        }
    }
}
```

The roster application uses the default mapping strategy of `InheritanceType.SINGLE_TABLE`, so the `@Inheritance` annotation is not required. If you want to use a different mapping strategy, decorate `League` with `@Inheritance` and specify the mapping strategy in the `strategy` element:

```java
@Entity
@Inheritance(strategy=JOINED)
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}
```

The roster application uses the default discriminator column name, so the `@DiscriminatorColumn` annotation is not required. Because you are using automatic table generation in roster, the Persistence provider will create a discriminator column called `DTYPE` in the `EJB_ROSTER_LEAGUE` table, which will store the name of the inherited entity used to create the league. If you want to use a different name for the discriminator column, decorate `League` with `@DiscriminatorColumn` and set the `name` element:

```java
@Entity
@DiscriminatorColumn(name="DISCRIMINATOR")
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}
```
Criteria Queries in the roster Application

The roster application uses Criteria API queries, as opposed to the JPQL queries used in order. Criteria queries are Java programming language, typesafe queries defined in the business tier of roster, in the RequestBean stateful session bean.

Metamodel Classes in the roster Application

Metamodel classes model an entity's attributes and are used by Criteria queries to navigate to an entity's attributes. Each entity class in roster has a corresponding metamodel class, generated at compile time, with the same package name as the entity and appended with an underscore character (_). For example, the roster.entity.Player entity has a corresponding metamodel class, roster.entity.Player_.

Each persistent field or property in the entity class has a corresponding attribute in the entity's metamodel class. For the Player entity, the corresponding metamodel class is:

```java
@StaticMetamodel(Player.class)
public class Player_ {
    public static volatile SingularAttribute<Player, String> id;
    public static volatile SingularAttribute<Player, String> name;
    public static volatile SingularAttribute<Player, String> position;
    public static volatile SingularAttribute<Player, Double> salary;
    public static volatile CollectionAttribute<Player, Team> teams;
}
```

Obtaining a CriteriaBuilder Instance in RequestBean

The CriteriaBuilder interface defines methods to create criteria query objects and create expressions for modifying those query objects. RequestBean creates an instance of CriteriaBuilder by using a @PostConstruct method, init:

```java
@PersistenceContext
private EntityManager em;
private CriteriaBuilder cb;

@PostConstruct
private void init() {
    cb = em.getCriteriaBuilder();
}
```

The EntityManager instance is injected at runtime, and then that EntityManager object is used to create the CriteriaBuilder instance by calling getCriteriaBuilder. The CriteriaBuilder instance is created in a @PostConstruct method to ensure that the EntityManager instance has been injected by the enterprise bean container.

Creating Criteria Queries in RequestBean’s Business Methods

Many of the business methods in RequestBean define Criteria queries. One business method, getPlayersByPosition, returns a list of players who play a particular position on a team:
public List<PlayerDetails> getPlayersByPosition(String position) {
    logger.info("getPlayersByPosition");
    List<Player> players = null;
    try {
        CriteriaQuery<Player> cq = cb.createQuery(Player.class);
        if (cq != null) {
            Root<Player> player = cq.from(Player.class);
            // set the where clause
            cq.where(cb.equal(player.get(Player_.position), position));
            cq.select(player);
            TypedQuery<Player> q = em.createQuery(cq);
            players = q.getResultList();
        }
        return copyPlayersToDetails(players);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

A query object is created by calling the CriteriaBuilder object's createQuery method, with the type set to Player because the query will return a list of players.

The query root, the base entity from which the query will navigate to find the entity's attributes and related entities, is created by calling the from method of the query object. This sets the FROM clause of the query.

The WHERE clause, set by calling the where method on the query object, restricts the results of the query according to the conditions of an expression. The CriteriaBuilder.equal method compares the two expressions. In getPlayersByPosition, the position attribute of the Player_ metamodel class, accessed by calling the get method of the query root, is compared to the position parameter passed to getPlayersByPosition.

The SELECT clause of the query is set by calling the select method of the query object. The query will return Player entities, so the query root object is passed as a parameter to select.

The query object is prepared for execution by calling EntityManager.createQuery, which returns a TypedQuery<T> object with the type of the query, in this case Player. This typed query object is used to execute the query, which occurs when the getResultList method is called, and a List<Player> collection is returned.

Automatic Table Generation in the roster Application

At deployment time, the GlassFish Server will automatically drop and create the database tables used by roster. This is done by setting the eclipselink.ddl-generation property to drop-and-create-tables in persistence.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
```
This feature is specific to the Java Persistence API provider used by the GlassFish Server and is not portable across Java EE servers. Automatic table creation is useful for development purposes, however, and the <i>eclipselink.ddl-generation</i> property may be removed from <i>persistence.xml</i> when preparing the application for production use, when deploying to other Java EE servers, or when using other persistence providers.

## Running the roster Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the roster application.

### To Run the roster Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to: 
   
   ```
   tut-install/examples/persistence/
   ```
3. Select the roster folder.
4. Select the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project.
6. In the Projects tab, right-click the roster project and select Run.

   You will see the following partial output from the application client in the Output tab:

   ```
   List all players in team T2:
P6 Ian Carlyle goalkeeper 555.0
P7 Rebecca Struthers midfielder 777.0
P8 Anne Anderson forward 65.0
P9 Jan Wesley defender 100.0
P10 Terry Smithson midfielder 100.0

   List all teams in league L1:
T1 Honey Bees Visalia
   ```
The roster Application

T2 Gophers Manteca
T5 Crows Orland

List all defenders:
P2 Alice Smith defender 505.0
P5 Barney Bold defender 100.0
P9 Jan Wesley defender 100.0
P22 Janice Walker defender 857.0
P25 Frank Fletcher defender 399.0
...

▼ To Run the roster Example Using Ant

1 In a terminal window, go to:
tut-install/examples/persistence/roster/

2 Type the following command:
ant
This runs the default task, which compiles the source files and packages the application into an EAR file located at tut-install/examples/persistence/roster/dist/roster.ear.

3 To deploy the EAR, make sure that the GlassFish Server is started; then type the following command:
ant deploy
The build system will check whether the Java DB database server is running and start it if it is not running, then deploy roster.ear. The GlassFish Server will then drop and create the database tables during deployment, as specified in persistence.xml.

After roster.ear is deployed, a client JAR, rosterClient.jar, is retrieved. This contains the application client.

4 To run the application client, type the following command:
ant run
You will see the output, which begins:

[echo] running application client container.
[exec] List all players in team T2:
[exec] P6 Ian Carlyle goalkeeper 555.0
[exec] P7 Rebecca Struthers midfielder 777.0
[exec] P8 Anne Anderson forward 65.0
[exec] P9 Jan Wesley defender 100.0
[exec] P10 Terry Smithson midfielder 100.0

[exec] List all teams in league L1:
[exec] T1 Honey Bees Visalia
[exec] T2 Gophers Manteca
[exec] T5 Crows Orland
[exec] List all defenders:
[exec] P2 Alice Smith defender 505.0
[exec] P5 Barney Bold defender 100.0
[exec] P9 Jan Wesley defender 100.0
[exec] P22 Janice Walker defender 857.0
[exec] P25 Frank Fletcher defender 399.0
...

The all Task
As a convenience, the all task will build, package, deploy, and run the application. To do this, type the following command:

ant all

The address-book Application

The address-book example application is a simple web application that stores contact data. It uses a single entity class, Contact, that uses the Java API for JavaBeans Validation (Bean Validation) to validate the data stored in the persistent attributes of the entity, as described in “Validating Persistent Fields and Properties” on page 583.

Bean Validation Constraints in address-book

The Contact entity uses the @NotNull, @Pattern, and @Past constraints on the persistent attributes.

The @NotNull constraint marks the attribute as a required field. The attribute must be set to a non-null value before the entity can be persisted or modified. Bean Validation will throw a validation error if the attribute is null when the entity is persisted or modified.

The @Pattern constraint defines a regular expression that the value of the attribute must match before the entity can be persisted or modified. This constraint has two different uses in address-book:

- The regular expression declared in the @Pattern annotation on the email field matches email addresses of the form name@domain.name.top.level.domain, allowing only valid characters for email addresses. For example, username@example.com will pass validation, as will firstName.lastName@mail.example.com. However, firstName.lastName@example.com, which contains an illegal comma character in the local name, will fail validation.

- The mobilePhone and homePhone fields are annotated with a @Pattern constraint that defines a regular expression to match phone numbers of the form (xxx) xxx-xxxx.

The @Past constraint is applied to the birthday field, which must be a java.util.Date in the past.
Here are the relevant parts of the Contact entity class:

```java
@Entity
public class Contact implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp="[a-z0-9!#$%&'*+/=?^_`{|}~-.]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-.]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?",
    message="{invalid.email}")
    protected String email;
    @Pattern(regexp="^\(?(\d{3})\)?[- ]?(\d{3})[- ]?(\d{4})$",
    message="{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp="^\(?(\d{3})\)?[- ]?(\d{3})[- ]?(\d{4})$",
    message="{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(javax.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

### Specifying Error Messages for Constraints in address-book

Some of the constraints in the Contact entity specify an optional message:

```java
@Pattern(regexp="^\(?(\d{3})\)?[- ]?(\d{3})[- ]?(\d{4})$",
    message="{invalid.phonenumber}")
protected String mobilePhone;
```

The optional message element in the @Pattern constraint overrides the default validation message. The message can be specified directly:

```java
@Pattern(regexp="^\(?(\d{3})\)?[- ]?(\d{3})[- ]?(\d{4})$",
    message="Invalid phone number!")
protected String homePhone;
```

The constraints in Contact, however, are strings in the resource bundle `tut-install/examples/persistence/address-book/src/java/ValidationMessages.properties`. This allows the validation messages to be located in one single properties file and the messages to be easily localized. Overridden Bean Validation messages must be placed in a resource bundle properties file named `ValidationMessages.properties` in the default package, with localized resource bundles.
taking the form ValidationMessages_locale-prefix.properties. For example, ValidationMessages_es.properties is the resource bundle used in Spanish speaking locales.

Validating Contact Input from a JavaServer Faces Application

The address-book application uses a JavaServer Faces web front end to allow users to enter contacts. While JavaServer Faces has a form input validation mechanism using tags in Facelets XHTML files, address-book doesn’t use these validation tags. Bean Validation constraints in JavaServer Faces managed beans, in this case in the Contact entity, automatically trigger validation when the forms are submitted.

The following code snippet from the Create.xhtml Facelets file shows some of the input form for creating new Contact instances:

```xml
<h:form columns="3" role="presentation">
    <tr>
        <td><h:outputLabel value="#{bundle.CreateContactLabel_firstName}" for="firstName" /></td>
        <td><h:inputText id="firstName" value="#{contactController.selected.firstName}" title="#{bundle.CreateContactTitle_firstName}" /></td>
        <td><h:message for="firstName" /></td>
    </tr>
    <tr>
        <td><h:outputLabel value="#{bundle.CreateContactLabel_lastName}" for="lastName" /></td>
        <td><h:inputText id="lastName" value="#{contactController.selected.lastName}" title="#{bundle.CreateContactTitle_lastName}" /></td>
        <td><h:message for="lastName" /></td>
    </tr>
    ...
</table>
</h:form>
```

The <h:inputText> tags firstName and lastName are bound to the attributes in the Contact entity instance selected in the ContactController stateless session bean. Each <h:inputText> tag has an associated <h:message> tag that will display validation error messages. The form doesn’t require any JavaServer Faces validation tags, however.

Running the address-book Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the address-book application.
To Run the address-book Example Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/persistence/`

3. Select the `address-book` folder.

4. Select the Open as Main Project and Open Required Projects check boxes.

5. Click Open Project.

6. In the Projects tab, right-click the `address-book` project and select Run.
   
   After the application has been deployed, a web browser window appears at the following URL:
   `http://localhost:8080/address-book/`

7. Click Show All Contact Items, then Create New Contact. Type values in the form fields; then click 
   Save.
   
   If any of the values entered violate the constraints in Contact, an error message will appear in 
   red beside the form field with the incorrect values.

To Run the address-book Example Using Ant

1. In a terminal window, go to:
   `tut-install/examples/persistence/address-book/`

2. Type the following command:
   
   `ant`
   
   This will compile and assemble the `address-book` application.

3. Type the following command:
   
   `ant deploy`
   
   This will deploy the application to GlassFish Server.

4. Open a web browser window and type the following URL:
   `http://localhost:8080/address-book/`
Tip – As a convenience, the all task will build, package, deploy, and run the application. To do this, type the following command:

```
ant all
```

5. Click Show All Contact Items, then Create New Contact. Type values in the form fields; then click Save.

If any of the values entered violate the constraints in Contact, an error message will appear in red beside the form field with the incorrect values.
The Java Persistence Query Language

The Java Persistence query language defines queries for entities and their persistent state. The query language allows you to write portable queries that work regardless of the underlying data store.

The query language uses the abstract persistence schemas of entities, including their relationships, for its data model and defines operators and expressions based on this data model. The scope of a query spans the abstract schemas of related entities that are packaged in the same persistence unit. The query language uses an SQL-like syntax to select objects or values based on entity abstract schema types and relationships among them.

This chapter relies on the material presented in earlier chapters. For conceptual information, see Chapter 32, "Introduction to the Java Persistence API." For code examples, see Chapter 33, "Running the Persistence Examples."

The following topics are addressed here:

- "Query Language Terminology" on page 630
- "Creating Queries Using the Java Persistence Query Language" on page 630
- "Simplified Query Language Syntax" on page 632
- "Example Queries" on page 633
- "Full Query Language Syntax" on page 637
Query Language Terminology

The following list defines some of the terms referred to in this chapter:

- **Abstract schema**: The persistent schema abstraction (persistent entities, their state, and their relationships) over which queries operate. The query language translates queries over this persistent schema abstraction into queries that are executed over the database schema to which entities are mapped.

- **Abstract schema type**: The type to which the persistent property of an entity evaluates in the abstract schema. That is, each persistent field or property in an entity has a corresponding state field of the same type in the abstract schema. The abstract schema type of an entity is derived from the entity class and the metadata information provided by Java language annotations.

- **Backus-Naur Form (BNF)**: A notation that describes the syntax of high-level languages. The syntax diagrams in this chapter are in BNF notation.

- **Navigation**: The traversal of relationships in a query language expression. The navigation operator is a period.

- **Path expression**: An expression that navigates to an entity's state or relationship field.

- **State field**: A persistent field of an entity.

- **Relationship field**: A persistent field of an entity whose type is the abstract schema type of the related entity.

Creating Queries Using the Java Persistence Query Language

The `EntityManager.createQuery` and `EntityManager.createNamedQuery` methods are used to query the datastore by using Java Persistence query language queries.

The `createQuery` method is used to create *dynamic queries*, which are queries defined directly within an application’s business logic:

```java
public List findWithName(String name) {
    return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE :custName",
                        .setParameter("custName", name)
                        .setMaxResults(10)
                        .getResultList();
}
```

The `createNamedQuery` method is used to create *static queries*, or queries that are defined in metadata by using the `javax.persistence.NamedQuery` annotation. The `name` element of `@NamedQuery` specifies the name of the query that will be used with the `createNamedQuery` method. The `query` element of `@NamedQuery` is the query:

```java
@NamedQuery(
    name="findAllCustomersWithName",
```
Here's an example of `createNamedQuery`, which uses the `@NamedQuery`:

```java
@PersistenceContext
public EntityManager em;
...

customers = em.createNamedQuery("findAllCustomersWithName")
  .setParameter("custName", "Smith")
  .getResultList();
```

### Named Parameters in Queries

Named parameters are query parameters that are prefixed with a colon (:). Named parameters in a query are bound to an argument by the following method:

```java
javax.persistence.Query.setParameter(String name, Object value)
```

In the following example, the `name` argument to the `findWithName` business method is bound to the `:custName` named parameter in the query by calling `Query.setParameter`:

```java
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName"
    )
      .setParameter("custName", name)
      .getResultList();
}
```

Named parameters are case-sensitive and may be used by both dynamic and static queries.

### Positional Parameters in Queries

You may use positional parameters instead of named parameters in queries. Positional parameters are prefixed with a question mark (?) followed the numeric position of the parameter in the query. The `Query.setParameter(integer position, Object value)` method is used to set the parameter values.

In the following example, the `findWithName` business method is rewritten to use input parameters:

```java
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE ?1"
    )
      .setParameter(1, name)
      .getResultList();
}
```

Input parameters are numbered starting from 1. Input parameters are case-sensitive, and may be used by both dynamic and static queries.
Simplified Query Language Syntax

This section briefly describes the syntax of the query language so that you can quickly move on to “Example Queries” on page 633. When you are ready to learn about the syntax in more detail, see “Full Query Language Syntax” on page 637.

Select Statements

A select query has six clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY. The SELECT and FROM clauses are required, but the WHERE, GROUP BY, HAVING, and ORDER BY clauses are optional. Here is the high-level BNF syntax of a query language select query:

\[
\text{QL statement} ::= \text{select clause from clause} \[\text{where clause}]\[\text{groupby clause}]\[\text{having clause}]\[\text{orderby clause}]
\]

- The SELECT clause defines the types of the objects or values returned by the query.
- The FROM clause defines the scope of the query by declaring one or more identification variables, which can be referenced in the SELECT and WHERE clauses. An identification variable represents one of the following elements:
  - The abstract schema name of an entity
  - An element of a collection relationship
  - An element of a single-valued relationship
  - A member of a collection that is the multiple side of a one-to-many relationship
- The WHERE clause is a conditional expression that restricts the objects or values retrieved by the query. Although the clause is optional, most queries have a WHERE clause.
- The GROUP BY clause groups query results according to a set of properties.
- The HAVING clause is used with the GROUP BY clause to further restrict the query results according to a conditional expression.
- The ORDER BY clause sorts the objects or values returned by the query into a specified order.

Update and Delete Statements

Update and delete statements provide bulk operations over sets of entities. These statements have the following syntax:

\[
\begin{align*}
\text{update statement} & ::= \text{update clause [where clause]} \\
\text{delete statement} & ::= \text{delete clause [where clause]}
\end{align*}
\]

The update and delete clauses determine the type of the entities to be updated or deleted. The WHERE clause may be used to restrict the scope of the update or delete operation.
Example Queries

The following queries are from the Player entity of the roster application, which is documented in “The roster Application” on page 615.

Simple Queries

If you are unfamiliar with the query language, these simple queries are a good place to start.

A Basic Select Query

```
SELECT p
FROM Player p
```

- **Data retrieved**: All players.
- **Description**: The FROM clause declares an identification variable named p, omitting the optional keyword AS. If the AS keyword were included, the clause would be written as follows:

```
FROM Player AS p
```

The Player element is the abstract schema name of the Player entity.

- **See also**: “Identification Variables” on page 643.

Eliminating Duplicate Values

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = ?1
```

- **Data retrieved**: The players with the position specified by the query’s parameter.
- **Description**: The DISTINCT keyword eliminates duplicate values.

The WHERE clause restricts the players retrieved by checking their position, a persistent field of the Player entity. The ?1 element denotes the input parameter of the query.

- **See also**: “Input Parameters” on page 648 and “The DISTINCT Keyword” on page 658.

Using Named Parameters

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

- **Data retrieved**: The players having the specified positions and names.
**Description:** The position and name elements are persistent fields of the Player entity. The `WHERE` clause compares the values of these fields with the named parameters of the query, set using the `Query.setNamedParameter` method. The query language denotes a named input parameter using a colon (:) followed by an identifier. The first input parameter is `:position`, the second is `:name`.

**Queries That Navigate to Related Entitites**

In the query language, an expression can traverse, or navigate, to related entities. These expressions are the primary difference between the Java Persistence query language and SQL. Queries navigates to related entities, whereas SQL joins tables.

**A Simple Query with Relationships**

```
SELECT DISTINCT p
FROM Player p, IN(p.teams) t
```

**Data retrieved:** All players who belong to a team.

**Description:** The `FROM` clause declares two identification variables: `p` and `t`. The `p` variable represents the Player entity, and the `t` variable represents the related Team entity. The declaration for `t` references the previously declared `p` variable. The `IN` keyword signifies that `teams` is a collection of related entities. The `p.teams` expression navigates from a Player to its related Team. The period in the `p.teams` expression is the navigation operator.

You may also use the `JOIN` statement to write the same query:

```
SELECT DISTINCT p
FROM Player p JOIN p.teams t
```

This query could also be rewritten as:

```
SELECT DISTINCT p
FROM Player p
WHERE p.team IS NOT EMPTY
```

**Navigating to Single-Valued Relationship Fields**

Use the `JOIN` clause statement to navigate to a single-valued relationship field:

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = 'soccer' OR l.sport = 'football'
```

In this example, the query will return all teams that are in either soccer or football leagues.

**Traversing Relationships with an Input Parameter**

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```
Data retrieved: The players whose teams belong to the specified city.

Description: This query is similar to the previous example but adds an input parameter. The AS keyword in the FROM clause is optional. In the WHERE clause, the period preceding the persistent variable city is a delimiter, not a navigation operator. Strictly speaking, expressions can navigate to relationship fields (related entities) but not to persistent fields. To access a persistent field, an expression uses the period as a delimiter.

Expressions cannot navigate beyond (or further qualify) relationship fields that are collections. In the syntax of an expression, a collection-valued field is a terminal symbol. Because the teams field is a collection, the WHERE clause cannot specify p.teams.city (an illegal expression).

See also: “Path Expressions” on page 646.

Traversing Multiple Relationships

SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league

Data retrieved: The players who belong to the specified league.

Description: The expressions in this query navigate over two relationships. The p.teams expression navigates the Player-Team relationship, and the t.league expression navigates the Team-League relationship.

In the other examples, the input parameters are String objects; in this example, the parameter is an object whose type is a League. This type matches the league relationship field in the comparison expression of the WHERE clause.

Navigating According to Related Fields

SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport

Data retrieved: The players who participate in the specified sport.

Description: The sport persistent field belongs to the League entity. To reach the sport field, the query must first navigate from the Player entity to Team (p.teams) and then from Team to the League entity (t.league). Because it is not a collection, the league relationship field can be followed by the sport persistent field.

Queries with Other Conditional Expressions

Every WHERE clause must specify a conditional expression, of which there are several kinds. In the previous examples, the conditional expressions are comparison expressions that test for equality. The following examples demonstrate some of the other kinds of conditional expressions. For descriptions of all conditional expressions, see “WHERE Clause” on page 647.
The LIKE Expression

```
SELECT p
FROM Player p
WHERE p.name LIKE 'Mich%'
```

- **Data retrieved:** All players whose names begin with “Mich.”
- **Description:** The LIKE expression uses wildcard characters to search for strings that match the wildcard pattern. In this case, the query uses the LIKE expression and the % wildcard to find all players whose names begin with the string “Mich.” For example, “Michael” and “Michelle” both match the wildcard pattern.
- **See also:** "LIKE Expressions” on page 650.

The IS NULL Expression

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

- **Data retrieved:** All teams not associated with a league.
- **Description:** The IS NULL expression can be used to check whether a relationship has been set between two entities. In this case, the query checks whether the teams are associated with any leagues and returns the teams that do not have a league.
- **See also:** "NULL Comparison Expressions” on page 650 and "NULL Values” on page 655.

The IS EMPTY Expression

```
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

- **Data retrieved:** All players who do not belong to a team.
- **Description:** The teams relationship field of the Player entity is a collection. If a player does not belong to a team, the teams collection is empty, and the conditional expression is TRUE.
- **See also:** "Empty Collection Comparison Expressions” on page 651.

The BETWEEN Expression

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

- **Data retrieved:** The players whose salaries fall within the range of the specified salaries.
- **Description:** This BETWEEN expression has three arithmetic expressions: a persistent field (p.salary) and the two input parameters (:lowerSalary and :higherSalary). The following expression is equivalent to the BETWEEN expression:
  
  ```
  p.salary >= :lowerSalary AND p.salary <= :higherSalary
  ```
- **See also:** "BETWEEN Expressions” on page 649.
Comparison Operators

```
SELECT DISTINCT p1
FROM Player p1, Player p2
WHERE p1.salary > p2.salary AND p2.name = :name
```

- **Data retrieved:** All players whose salaries are higher than the salary of the player with the specified name.
- **Description:** The `FROM` clause declares two identification variables (p1 and p2) of the same type (Player). Two identification variables are needed because the `WHERE` clause compares the salary of one player (p2) with that of the other players (p1).
- **See also:** “Identification Variables” on page 643.

Bulk Updates and Deletes

The following examples show how to use the `UPDATE` and `DELETE` expressions in queries. `UPDATE` and `DELETE` operate on multiple entities according to the condition or conditions set in the `WHERE` clause. The `WHERE` clause in `UPDATE` and `DELETE` queries follows the same rules as `SELECT` queries.

**Update Queries**

```
UPDATE Player p
SET p.status = 'inactive'
WHERE p.lastPlayed < :inactiveThresholdDate
```

- **Description:** This query sets the status of a set of players to inactive if the player's last game was longer than the date specified in `inactiveThresholdDate`.

**Delete Queries**

```
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

- **Description:** This query deletes all inactive players who are not on a team.

Full Query Language Syntax

This section discusses the query language syntax, as defined in the Java Persistence API 2.0 specification available at [http://jcp.org/en/jsr/detail?id=317](http://jcp.org/en/jsr/detail?id=317). Much of the following material paraphrases or directly quotes the specification.

BNF Symbols

Table 34-1 describes the BNF symbols used in this chapter.
TABLE 34–1 BNF Symbol Summary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>The element to the left of the symbol is defined by the constructs on the right.</td>
</tr>
<tr>
<td>*</td>
<td>The preceding construct may occur zero or more times.</td>
</tr>
<tr>
<td>{...}</td>
<td>The constructs within the braces are grouped together.</td>
</tr>
<tr>
<td>[... ]</td>
<td>The constructs within the brackets are optional.</td>
</tr>
<tr>
<td></td>
<td>An exclusive OR.</td>
</tr>
<tr>
<td>BOLDFACE</td>
<td>A keyword; although capitalized in the BNF diagram, keywords are not case-sensitive.</td>
</tr>
<tr>
<td>White space</td>
<td>A whitespace character can be a space, a horizontal tab, or a line feed.</td>
</tr>
</tbody>
</table>

**BNF Grammar of the Java Persistence Query Language**

Here is the entire BNF diagram for the query language:

```bnf
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::= FROM [identification_variable_declaration | collection_member_declaration]|
identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= abstract_schema_name [AS] identification_variable
join ::= join_spec join_association_path_expression [AS] identification_variable
fetch_join ::= join_specFETCH join_association_path_expression
association_path_expression ::= collection_valued_path_expression | single_valued_path_expression
join_spec ::= [LEFT | OUTER | INNER] JOIN
join_collection_valued_path_expression ::= join_single_valued_association_path_expression
join_single_valued_association_path_expression ::= identification_variable.collection_valued_association_field
join_collection_valued_path_expression ::= IN (collection_valued_path_expression) [AS] identification_variable
collection_member_declaration ::= IN (collection_valued_path_expression) [AS] identification_variable
```
collection_member_declaration
simple_select_clause ::= SELECT [DISTINCT]
simple_select_expression
simple_select_expression ::= single_valued_path_expression |
aggregate_expression |
identification_variable
conditional_expression ::= conditional_term |
conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::= comparison_expression |
between_expression |
in_expression |
null_comparison_expression |
empty_collection_comparison_expression |
collection_member_expression |
exists_expression
between_expression ::= arithmetic_expression [NOT] BETWEEN
arithmetic_expression AND arithmetic_expression |
string_expression [NOT] BETWEEN string_expression AND
string_expression |
datetime_expression [NOT] BETWEEN
datetime_expression AND datetime_expression
in_expression ::= state_field_path_expression [NOT] IN (in_item {, in_item}*
| subquery)
in_item ::= literal | input_parameter
like_expression ::= string_expression [NOT] LIKE pattern_value [ESCAPE
escape_character]
null_comparison_expression ::= {single_valued_path_expression | input_parameter} IS [NOT] NULL
empty_collection_comparison_expression ::= collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression |
[NOT] MEMBER [OF] collection_valued_path_expression
exists_expression ::= [NOT] EXISTS {subquery}
all_or_any_expression ::= {ALL | ANY | SOME} {subquery}
comparison_expression ::= string_expression comparison_operator {string_expression |
all_or_any_expression} |
boolean_expression (= |<> ) {boolean_expression |
all_or_any_expression} |
enum_expression (= |<> ) {enum_expression |
all_or_any_expression} |
datetime_expression comparison_operator
{datetime_expression | all_or_any_expression} |
entity_expression (= |<> ) {entity_expression |
all_or_any_expression} |
arithmetic_expression comparison_operator
{arithmetic_expression | all_or_any_expression}
comparison_operator ::= = | > | >= | < | <= | <>

arithmetic_expression ::= simple_arithmetic_expression | (subquery)

simple_arithmetic_expression ::= arithmetic_term | simple_arithmetic_expression {+ | - }

arithmetic_term ::= arithmetic_factor | arithmetic_term {* | / }

arithmetic_factor ::= [+ | - ] arithmetic_primary

arithmetic_primary ::= state_field_path_expression | numeric_literal | (simple_arithmetic_expression) | input_parameter | functions_returning_numerics | aggregate_expression

string_expression ::= string_primary | (subquery)

string_primary ::= state_field_path_expression | string_literal | input_parameter | functions_returning_strings | aggregate_expression

datetime_expression ::= datetime_primary | (subquery)

datetime_primary ::= state_field_path_expression | input_parameter | functions_returning_datetime | aggregate_expression

boolean_expression ::= boolean_primary | (subquery)

boolean_primary ::= state_field_path_expression | boolean_literal | input_parameter

enum_expression ::= enum_primary | (subquery)

enum_primary ::= state_field_path_expression | enum_literal | input_parameter

daentity_expression ::= single_valued_association_path_expression | simple_entity_expression

simple_entity_expression ::= identification_variable | input_parameter

functions_returning_numerics ::= LENGTH(string_primary) | LOCATE(string_primary, string_primary[, simple_arithmetic_expression]) | ABS(simple_arithmetic_expression) | SQRT(simple_arithmetic_expression) | MOD(simple_arithmetic_expression, simple_arithmetic_expression) | SIZE(collection_valued_path_expression)

functions_returning_datetime ::= CURRENT_DATE | CURRENT_TIME | CURRENT_TIMESTAMP

functions_returning_strings ::=
CONCAT(string_primary, string_primary) | SUBSTRING(string_primary, simple_arithmetic_expression, simple_arithmetic_expression) | TRIM([[trim_specification] [trim_character] FROM] string_primary) | LOWER(string_primary) | UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH

FROM Clause

The FROM clause defines the domain of the query by declaring identification variables.

Identifiers

An identifier is a sequence of one or more characters. The first character must be a valid first character (letter, $, _) in an identifier of the Java programming language, hereafter in this chapter called simply "Java". Each subsequent character in the sequence must be a valid nonfirst character (letter, digit, $, _) in a Java identifier. (For details, see the Java SE API documentation of the isJavaIdentifierStart and isJavaIdentifierPart methods of the Character class.)

The question mark (?) is a reserved character in the query language and cannot be used in an identifier.

A query language identifier is case-sensitive, with two exceptions:

- Keywords
- Identification variables

An identifier cannot be the same as a query language keyword. Here is a list of query language keywords:

<table>
<thead>
<tr>
<th>ABS</th>
<th>ALL</th>
<th>AND</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>ASC</td>
<td>AVG</td>
<td>BETWEEN</td>
</tr>
<tr>
<td>BIT_LENGTH</td>
<td>BOTH</td>
<td>BY</td>
<td>CASE</td>
</tr>
<tr>
<td>CHAR_LENGTH</td>
<td>CHARACTER_LENGTH</td>
<td>CLASS</td>
<td>COALESCE</td>
</tr>
<tr>
<td>CONCAT</td>
<td>COUNT</td>
<td>CURRENT_DATE</td>
<td>CURRENT_TIMESTAMP</td>
</tr>
<tr>
<td>DELETE</td>
<td>DESC</td>
<td>DISTINCT</td>
<td>ELSE</td>
</tr>
<tr>
<td>EMPTY</td>
<td>END</td>
<td>ENTRY</td>
<td>ESCAPE</td>
</tr>
<tr>
<td>EXISTS</td>
<td>FALSE</td>
<td>FETCH</td>
<td>FROM</td>
</tr>
<tr>
<td>GROUP</td>
<td>HAVING</td>
<td>IN</td>
<td>INDEX</td>
</tr>
<tr>
<td>INNER</td>
<td>IS</td>
<td>JOIN</td>
<td>KEY</td>
</tr>
</tbody>
</table>
It is not recommended that you use an SQL keyword as an identifier, because the list of keywords may expand to include other reserved SQL words in the future.

**Identification Variables**

An *identification variable* is an identifier declared in the `FROM` clause. Although they can reference identification variables, the `SELECT` and `WHERE` clauses cannot declare them. All identification variables must be declared in the `FROM` clause.

Because it is an identifier, an identification variable has the same naming conventions and restrictions as an identifier, with the exception that an identification variable is case-insensitive. For example, an identification variable cannot be the same as a query language keyword. (See the preceding section for more naming rules.) Also, within a given persistence unit, an identification variable name must not match the name of any entity or abstract schema.

The `FROM` clause can contain multiple declarations, separated by commas. A declaration can reference another identification variable that has been previously declared (to the left). In the following `FROM` clause, the variable `t` references the previously declared variable `p`:

```
FROM Player p, IN (p.teams) AS t
```

Even if it is not used in the `WHERE` clause, an identification variable's declaration can affect the results of the query. For example, compare the next two queries. The following query returns all players, whether or not they belong to a team:

```
SELECT p
FROM Player p
```

In contrast, because it declares the `t` identification variable, the next query fetches all players who belong to a team:

```
FROM Player p
```

```
WHERE t
```
The following query returns the same results as the preceding query, but the WHERE clause makes it easier to read:

```
SELECT p
FROM Player p
WHERE p.teams IS NOT EMPTY
```

An identification variable always designates a reference to a single value whose type is that of the expression used in the declaration. There are two kinds of declarations: range variable and collection member.

**Range Variable Declarations**

To declare an identification variable as an abstract schema type, you specify a range variable declaration. In other words, an identification variable can range over the abstract schema type of an entity. In the following example, an identification variable named p represents the abstract schema named Player:

```
FROM Player p
```

A range variable declaration can include the optional AS operator:

```
FROM Player AS p
```

To obtain objects, a query usually uses path expressions to navigate through the relationships. But for those objects that cannot be obtained by navigation, you can use a range variable declaration to designate a starting point, or root.

If the query compares multiple values of the same abstract schema type, the FROM clause must declare multiple identification variables for the abstract schema:

```
FROM Player p1, Player p2
```

For an example of such a query, see “Comparison Operators” on page 637.

**Collection Member Declarations**

In a one-to-many relationship, the multiple side consists of a collection of entities. An identification variable can represent a member of this collection. To access a collection member, the path expression in the variable's declaration navigates through the relationships in the abstract schema. (For more information on path expressions, see “Path Expressions” on page 646.) Because a path expression can be based on another path expression, the navigation can traverse several relationships. See “Traversing Multiple Relationships” on page 635.

A collection member declaration must include the IN operator but can omit the optional AS operator.
In the following example, the entity represented by the abstract schema named Player has a relationship field called teams. The identification variable called t represents a single member of the teams collection.

FROM Player p, IN (p.teams) t

**Joins**

The JOIN operator is used to traverse over relationships between entities and is functionally similar to the IN operator.

In the following example, the query joins over the relationship between customers and orders:

SELECT c
FROM Customer c JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000

The INNER keyword is optional:

SELECT c
FROM Customer c INNER JOIN c.orders o
WHERE c.status = 1 AND o.totalPrice > 10000

These examples are equivalent to the following query, which uses the IN operator:

SELECT c
FROM Customer c, IN(c.orders) o
WHERE c.status = 1 AND o.totalPrice > 10000

You can also join a single-valued relationship:

SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = :sport

A LEFT JOIN or LEFT OUTER JOIN retrieves a set of entities where matching values in the join condition may be absent. The OUTER keyword is optional.

SELECT c.name, o.totalPrice
FROM Order o LEFT JOIN o.customer c

A FETCH JOIN is a join operation that returns associated entities as a side effect of running the query. In the following example, the query returns a set of departments and, as a side effect, the associated employees of the departments, even though the employees were not explicitly retrieved by the SELECT clause.

SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
Path Expressions

Path expressions are important constructs in the syntax of the query language, for several reasons. First, path expressions define navigation paths through the relationships in the abstract schema. These path definitions affect both the scope and the results of a query. Second, path expressions can appear in any of the main clauses of a query (SELECT, DELETE, HAVING, UPDATE, WHERE, FROM, GROUP BY, ORDER BY). Finally, although much of the query language is a subset of SQL, path expressions are extensions not found in SQL.

Examples of Path Expressions

Here, the WHERE clause contains a single_valued_path_expression; the p is an identification variable, and salary is a persistent field of Player:

```sql
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

Here, the WHERE clause also contains a single_valued_path_expression; t is an identification variable, league is a single-valued relationship field, and sport is a persistent field of league:

```sql
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

Here, the WHERE clause contains a collection_valued_path_expression; p is an identification variable, and teams designates a collection-valued relationship field:

```sql
SELECT DISTINCT p
FROM Player p
WHERE p.teams IS EMPTY
```

Expression Types

The type of a path expression is the type of the object represented by the ending element, which can be one of the following:

- Persistent field
- Single-valued relationship field
- Collection-valued relationship field

For example, the type of the expression `p.salary` is `double` because the terminating persistent field (`salary`) is a `double`.

In the expression `p.teams`, the terminating element is a collection-valued relationship field (`teams`). This expression’s type is a collection of the abstract schema type named Team. Because Team is the abstract schema name for the Team entity, this type maps to the entity. For more information on the type mapping of abstract schemas, see "Return Types" on page 657.
Navigation

A path expression enables the query to navigate to related entities. The terminating elements of an expression determine whether navigation is allowed. If an expression contains a single-valued relationship field, the navigation can continue to an object that is related to the field. However, an expression cannot navigate beyond a persistent field or a collection-valued relationship field. For example, the expression `p.teams.league.sport` is illegal because `teams` is a collection-valued relationship field. To reach the `sport` field, the FROM clause could define an identification variable named `t` for the `teams` field:

```
FROM Player AS p, IN (p.teams) t
WHERE t.league.sport = 'soccer'
```

WHERE Clause

The WHERE clause specifies a conditional expression that limits the values returned by the query. The query returns all corresponding values in the data store for which the conditional expression is TRUE. Although usually specified, the WHERE clause is optional. If the WHERE clause is omitted, the query returns all values. The high-level syntax for the WHERE clause follows:

```
where_clause ::= WHERE conditional_expression
```

Literals

There are four kinds of literals: string, numeric, Boolean, and enum.

- **String literals**: A string literal is enclosed in single quotes:

  `'Duke'`

  If a string literal contains a single quote, you indicate the quote by using two single quotes:

  `'Duke''s`

  Like a Java String, a string literal in the query language uses the Unicode character encoding.

- **Numeric literals**: There are two types of numeric literals: exact and approximate.

  An exact numeric literal is a numeric value without a decimal point, such as 65, -233, and +12. Using the Java integer syntax, exact numeric literals support numbers in the range of a Java long.

  An approximate numeric literal is a numeric value in scientific notation, such as 57., -85.7, and +2.1. Using the syntax of the Java floating-point literal, approximate numeric literals support numbers in the range of a Java double.

- **Boolean literals**: A Boolean literal is either TRUE or FALSE. These keywords are not case-sensitive.
**Enum literals**: The Java Persistence query language supports the use of enum literals using the Java enum literal syntax. The enum class name must be specified as a fully qualified class name:

```
SELECT e
FROM Employee e
WHERE e.status = com.xyz.EmployeeStatus.FULL_TIME
```

---

### Input Parameters

An input parameter can be either a named parameter or a positional parameter.

- A named input parameter is designated by a colon (:) followed by a string; for example, `:name`.
- A positional input parameter is designated by a question mark (?) followed by an integer. For example, the first input parameter is `?1`, the second is `?2`, and so forth.

The following rules apply to input parameters.

- They can be used only in a `WHERE` or `HAVING` clause.
- Positional parameters must be numbered, starting with the integer 1.
- Named parameters and positional parameters may not be mixed in a single query.
- Named parameters are case-sensitive.

---

### Conditional Expressions

A `WHERE` clause consists of a conditional expression, which is evaluated from left to right within a precedence level. You can change the order of evaluation by using parentheses.

---

### Operators and Their Precedence

**Table 34–2** lists the query language operators in order of decreasing precedence.

<table>
<thead>
<tr>
<th>Type</th>
<th>Precedence Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigation</td>
<td>. (a period)</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+ – (unary)</td>
</tr>
<tr>
<td></td>
<td>* / (multiplication and division)</td>
</tr>
<tr>
<td></td>
<td>+ – (addition and subtraction)</td>
</tr>
</tbody>
</table>
### BETWEEN Expressions

A BETWEEN expression determines whether an arithmetic expression falls within a range of values.

These two expressions are equivalent:

\[ p\_age \ \text{BETWEEN} \ 15 \ \text{AND} \ 19 \]
\[ p\_age \ \geq 15 \ \text{AND} \ p\_age \ \leq 19 \]

The following two expressions also are equivalent:

\[ p\_age \ \text{NOT BETWEEN} \ 15 \ \text{AND} \ 19 \]
\[ p\_age \ < 15 \ \text{OR} \ p\_age \ > 19 \]

If an arithmetic expression has a NULL value, the value of the BETWEEN expression is unknown.

### IN Expressions

An IN expression determines whether a string belongs to a set of string literals or whether a number belongs to a set of number values.
The path expression must have a string or numeric value. If the path expression has a NULL value, the value of the IN expression is unknown.

In the following example, the expression is TRUE if the country is UK, but FALSE if the country is Peru.

```
o.country IN ('UK', 'US', 'France')
```

You may also use input parameters:

```
o.country IN ('UK', 'US', 'France', :country)
```

**LIKE Expressions**

A LIKE expression determines whether a wildcard pattern matches a string.

The path expression must have a string or numeric value. If this value is NULL, the value of the LIKE expression is unknown. The pattern value is a string literal that can contain wildcard characters. The underscore (_) wildcard character represents any single character. The percent (%) wildcard character represents zero or more characters. The ESCAPE clause specifies an escape character for the wildcard characters in the pattern value. Table 34–3 shows some sample LIKE expressions.

**Table 34–3** LIKE Expression Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>TRUE</th>
<th>FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>address.phone LIKE '12%3'</td>
<td>'123'</td>
<td>'12993'</td>
</tr>
<tr>
<td></td>
<td>'12993'</td>
<td></td>
</tr>
<tr>
<td>asentence.word LIKE '_se'</td>
<td>'lose'</td>
<td>'loose'</td>
</tr>
<tr>
<td>aword.underscored LIKE '_%' ESCAPE ''</td>
<td>'_foo'</td>
<td>'bar'</td>
</tr>
<tr>
<td>address.phone NOT LIKE '12%3'</td>
<td>'1234'</td>
<td>'123'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'12993'</td>
</tr>
</tbody>
</table>

**NULL Comparison Expressions**

A NULL comparison expression tests whether a single-valued path expression or an input parameter has a NULL value. Usually, the NULL comparison expression is used to test whether a single-valued relationship has been set:

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```
This query selects all teams where the league relationship is not set. Note that the following query is not equivalent:

```sql
SELECT t
FROM Team t
WHERE t.league = NULL
```

The comparison with `NULL` using the equals operator (`=`) always returns an unknown value, even if the relationship is not set. The second query will always return an empty result.

### Empty Collection Comparison Expressions

The `IS [NOT] EMPTY` comparison expression tests whether a collection-valued path expression has no elements. In other words, it tests whether a collection-valued relationship has been set.

If the collection-valued path expression is `NULL`, the empty collection comparison expression has a `NULL` value.

Here is an example that finds all orders that do not have any line items:

```sql
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

### Collection Member Expressions

The `[NOT] MEMBER [OF]` collection member expression determines whether a value is a member of a collection. The value and the collection members must have the same type.

If either the collection-valued or single-valued path expression is unknown, the collection member expression is unknown. If the collection-valued path expression designates an empty collection, the collection member expression is `FALSE`.

The `OF` keyword is optional.

The following example tests whether a line item is part of an order:

```sql
SELECT o
FROM Order o
WHERE :lineItem MEMBER OF o.lineItems
```

### Subqueries

Subqueries may be used in the `WHERE` or `HAVING` clause of a query. Subqueries must be surrounded by parentheses.

The following example finds all customers who have placed more than ten orders:

```sql
SELECT c
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```
Subqueries may contain EXISTS, ALL, and ANY expressions.

- **EXISTS expressions**: The [NOT] EXISTS expression is used with a subquery and is true only if the result of the subquery consists of one or more values and is false otherwise.

  The following example finds all employees whose spouses are also employees:
  
  ```sql
  SELECT DISTINCT emp
  FROM Employee emp
  WHERE EXISTS (SELECT spouseEmp
                 FROM Employee spouseEmp
                 WHERE spouseEmp = emp.spouse)
  ```

- **ALL and ANY expressions**: The ALL expression is used with a subquery and is true if all the values returned by the subquery are true or if the subquery is empty.
  
  The ANY expression is used with a subquery and is true if some of the values returned by the subquery are true. An ANY expression is false if the subquery result is empty or if all the values returned are false. The SOME keyword is synonymous with ANY.
  
  The ALL and ANY expressions are used with the =, <, <=, >, >=, and <> comparison operators.
  
  The following example finds all employees whose salaries are higher than the salaries of the managers in the employee’s department:
  
  ```sql
  SELECT emp
  FROM Employee emp
  WHERE emp.salary > ALL (SELECT m.salary
                           FROM Manager m
                           WHERE m.department = emp.department)
  ```

**Functional Expressions**

The query language includes several string, arithmetic, and date/time functions that may be used in the SELECT, WHERE, or HAVING clause of a query. The functions are listed in Table 34-4, Table 34-5, and Table 34-6.

In Table 34-4, the start and length arguments are of type int and designate positions in the String argument. The first position in a string is designated by 1.
### String Expressions

<table>
<thead>
<tr>
<th>Function Syntax</th>
<th>ReturnType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT(String, String)</td>
<td>String</td>
</tr>
<tr>
<td>LENGTH(String)</td>
<td>int</td>
</tr>
<tr>
<td>LOCATE(String, String [, start])</td>
<td>int</td>
</tr>
<tr>
<td>SUBSTRING(String, start, length)</td>
<td>String</td>
</tr>
<tr>
<td>TRIM([LEADING</td>
<td>TRAILING</td>
</tr>
<tr>
<td>LOWER(String)</td>
<td>String</td>
</tr>
<tr>
<td>UPPER(String)</td>
<td>String</td>
</tr>
</tbody>
</table>

The **CONCAT** function concatenates two strings into one string.

The **LENGTH** function returns the length of a string in characters as an integer.

The **LOCATE** function returns the position of a given string within a string. This function returns the first position at which the string was found as an integer. The first argument is the string to be located. The second argument is the string to be searched. The optional third argument is an integer that represents the starting string position. By default, **LOCATE** starts at the beginning of the string. The starting position of a string is 1. If the string cannot be located, **LOCATE** returns 0.

The **SUBSTRING** function returns a string that is a substring of the first argument based on the starting position and length.

The **TRIM** function trims the specified character from the beginning and/or end of a string. If no character is specified, **TRIM** removes spaces or blanks from the string. If the optional **LEADING** specification is used, **TRIM** removes only the leading characters from the string. If the optional **TRAILING** specification is used, **TRIM** removes only the trailing characters from the string. The default is **BOTH**, which removes the leading and trailing characters from the string.

The **LOWER** and **UPPER** functions convert a string to lowercase or uppercase, respectively.

In **Table 34–5**, the number argument can be an **int**, a **float**, or a **double**.
TABLE 34–5  Arithmetic Expressions

<table>
<thead>
<tr>
<th>Function Syntax</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(number)</td>
<td>int, float, or double</td>
</tr>
<tr>
<td>MOD(int, int)</td>
<td>int</td>
</tr>
<tr>
<td>SQRT(double)</td>
<td>double</td>
</tr>
<tr>
<td>SIZE(Collection)</td>
<td>int</td>
</tr>
</tbody>
</table>

The ABS function takes a numeric expression and returns a number of the same type as the argument.

The MOD function returns the remainder of the first argument divided by the second.

The SQRT function returns the square root of a number.

The SIZE function returns an integer of the number of elements in the given collection.

In Table 34–6, the date/time functions return the date, time, or timestamp on the database server.

TABLE 34–6  Date/Time Expressions

<table>
<thead>
<tr>
<th>Function Syntax</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>CURRENT_TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>java.sql.Timestamp</td>
</tr>
</tbody>
</table>

Case Expressions

Case expressions change based on a condition, similar to the case keyword of the Java programming language. The CASE keyword indicates the start of a case expression, and the expression is terminated by the END keyword. The WHEN and THEN keywords define individual conditions, and the ELSE keyword defines the default condition should none of the other conditions be satisfied.

The following query selects the name of a person and a conditional string, depending on the subtype of the Person entity. If the subtype is Student, the string kid is returned. If the subtype is Guardian or Staff, the string adult is returned. If the entity is some other subtype of Person, the string unknown is returned.

```sql
SELECT p.name
CASE TYPE(p)
  WHEN Student THEN 'kid'
```

The Java EE 6 Tutorial • January 2013
WHEN Guardian THEN 'adult'
WHEN Staff THEN 'adult'
ELSE 'unknown'
END
FROM Person p

The following query sets a discount for various types of customers. Gold-level customers get a 20% discount, silver-level customers get a 15% discount, bronze-level customers get a 10% discount, and everyone else gets a 5% discount.

UPDATE Customer c
SET c.discount =
  CASE c.level
      WHEN 'Gold' THEN 20
      WHEN 'SILVER' THEN 15
      WHEN 'Bronze' THEN 10
      ELSE 5
  END

NULL Values

If the target of a reference is not in the persistent store, the target is NULL. For conditional expressions containing NULL, the query language uses the semantics defined by SQL92. Briefly, these semantics are as follows.

- If a comparison or arithmetic operation has an unknown value, it yields a NULL value.
- Two NULL values are not equal. Comparing two NULL values yields an unknown value.
- The IS NULL test converts a NULL persistent field or a single-valued relationship field to TRUE. The IS NOT NULL test converts them to FALSE.
- Boolean operators and conditional tests use the three-valued logic defined by Table 34–7 and Table 34–8. (In these tables, T stands for TRUE, F for FALSE, and U for unknown.)

<table>
<thead>
<tr>
<th>AND</th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>F</td>
<td>U</td>
</tr>
</tbody>
</table>
Equality Semantics

In the query language, only values of the same type can be compared. However, this rule has one exception: Exact and approximate numeric values can be compared. In such a comparison, the required type conversion adheres to the rules of Java numeric promotion.

The query language treats compared values as if they were Java types and not as if they represented types in the underlying data store. For example, a persistent field that could be either an integer or a NULL must be designated as an Integer object and not as an int primitive. This designation is required because a Java object can be NULL, but a primitive cannot.

Two strings are equal only if they contain the same sequence of characters. Trailing blanks are significant; for example, the strings ‘abc’ and ‘abc ’ are not equal.

Two entities of the same abstract schema type are equal only if their primary keys have the same value. Table 34–9 shows the operator logic of a negation, and Table 34–10 shows the truth values of conditional tests.

<table>
<thead>
<tr>
<th>TABLE 34–8</th>
<th>OR Operator Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 34–9</th>
<th>NOT Operator Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT Value</td>
<td>Value</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 34–10</th>
<th>Conditional Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional Test</td>
<td>T</td>
</tr>
<tr>
<td>Expression IS TRUE</td>
<td>T</td>
</tr>
<tr>
<td>Expression IS FALSE</td>
<td>F</td>
</tr>
<tr>
<td>Expression is unknown</td>
<td>F</td>
</tr>
</tbody>
</table>
SELECT Clause

The SELECT clause defines the types of the objects or values returned by the query.

Return Types

The return type of the SELECT clause is defined by the result types of the select expressions contained within it. If multiple expressions are used, the result of the query is an Object[], and the elements in the array correspond to the order of the expressions in the SELECT clause and in type to the result types of each expression.

A SELECT clause cannot specify a collection-valued expression. For example, the SELECT clause p.teams is invalid because teams is a collection. However, the clause in the following query is valid because the t is a single element of the teams collection:

```
SELECT t
FROM Player p, IN (p.teams) t
```

The following query is an example of a query with multiple expressions in the SELECT clause:

```
SELECT c.name, c.country.name
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

This query returns a list of Object[] elements; the first array element is a string denoting the customer name, and the second array element is a string denoting the name of the customer’s country.

The result of a query may be the result of an aggregate function, listed in Table 34–11.

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>Double</td>
<td>Returns the mean average of the fields</td>
</tr>
<tr>
<td>COUNT</td>
<td>Long</td>
<td>Returns the total number of results</td>
</tr>
<tr>
<td>MAX</td>
<td>The type of the field</td>
<td>Returns the highest value in the result set</td>
</tr>
<tr>
<td>MIN</td>
<td>The type of the field</td>
<td>Returns the lowest value in the result set</td>
</tr>
<tr>
<td>SUM</td>
<td>Long (for integral fields)</td>
<td>Returns the sum of all the values in the result set</td>
</tr>
<tr>
<td></td>
<td>Double (for floating-point fields)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BigInteger (for BigInteger fields)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BigDecimal (for BigDecimal fields)</td>
<td></td>
</tr>
</tbody>
</table>
For select method queries with an aggregate function (AVG, COUNT, MAX, MIN, or SUM) in the SELECT clause, the following rules apply:

- The AVG, MAX, MIN, and SUM functions return null if there are no values to which the function can be applied.
- The COUNT function returns 0 if there are no values to which the function can be applied.

The following example returns the average order quantity:

```sql
SELECT AVG(o.quantity)
FROM Order o
```

The following example returns the total cost of the items ordered by Roxane Coss:

```sql
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

The following example returns the total number of orders:

```sql
SELECT COUNT(o)
FROM Order o
```

The following example returns the total number of items that have prices in Hal Incandenza's order:

```sql
SELECT COUNT(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Incandenza' AND c.firstname = 'Hal'
```

**The DISTINCT Keyword**

The DISTINCT keyword eliminates duplicate return values. If a query returns a java.util.Collection, which allows duplicates, you must specify the DISTINCT keyword to eliminate duplicates.

**Constructor Expressions**

Constructor expressions allow you to return Java instances that store a query result element instead of an Object[].

The following query creates a CustomerDetail instance per Customer matching the WHERE clause. A CustomerDetail stores the customer name and customer’s country name. So the query returns a List of CustomerDetail instances:

```sql
SELECT NEW com.xyz.CustomerDetail(c.name, c.country.name)
FROM customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```
**ORDER BY Clause**

As its name suggests, the ORDER BY clause orders the values or objects returned by the query.

If the ORDER BY clause contains multiple elements, the left-to-right sequence of the elements determines the high-to-low precedence.

The ASC keyword specifies ascending order, the default, and the DESC keyword indicates descending order.

When using the ORDER BY clause, the SELECT clause must return an orderable set of objects or values. You cannot order the values or objects for values or objects not returned by the SELECT clause. For example, the following query is valid because the ORDER BY clause uses the objects returned by the SELECT clause:

```sql
SELECT o
FROM Customer c JOIN c.orders o JOIN o.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, o.totalcost
```

The following example is not valid, because the ORDER BY clause uses a value not returned by the SELECT clause:

```sql
SELECT p.product_name
FROM Order o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = 'Faehmel' AND c.firstname = 'Robert'
ORDER BY o.quantity
```

**GROUP BY and HAVING Clauses**

The GROUP BY clause allows you to group values according to a set of properties.

The following query groups the customers by their country and returns the number of customers per country:

```sql
SELECT c.country, COUNT(c)
FROM Customer c GROUP BY c.country
```

The HAVING clause is used with the GROUP BY clause to further restrict the returned result of a query.

The following query groups orders by the status of their customer and returns the customer status plus the average totalPrice for all orders where the corresponding customers has the same status. In addition, it considers only customers with status 1, 2, or 3, so orders of other customers are not taken into account:

```sql
SELECT c.status, AVG(o.totalPrice)
FROM Order o JOIN o.customer c
GROUP BY c.status HAVING c.status IN (1, 2, 3)
```
Using the Criteria API to Create Queries

The Criteria API is used to define queries for entities and their persistent state by creating query-defining objects. Criteria queries are written using Java programming language APIs, are typesafe, and are portable. Such queries work regardless of the underlying data store.

The following topics are addressed here:
- “Overview of the Criteria and Metamodel APIs” on page 661
- “Using the Metamodel API to Model Entity Classes” on page 663
- “Using the Criteria API and Metamodel API to Create Basic Typesafe Queries” on page 664

Overview of the Criteria and Metamodel APIs

Similar to JPQL, the Criteria API is based on the abstract schema of persistent entities, their relationships, and embedded objects. The Criteria API operates on this abstract schema to allow developers to find, modify, and delete persistent entities by invoking Java Persistence API entity operations. The Metamodel API works in concert with the Criteria API to model persistent entity classes for Criteria queries.

The Criteria API and JPQL are closely related and are designed to allow similar operations in their queries. Developers familiar with JPQL syntax will find equivalent object-level operations in the Criteria API.

The following simple Criteria query returns all instances of the Pet entity in the data source:

```java
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> allPets = q resultList();
```

The equivalent JPQL query is:
SELECT p
FROM Pet p

This query demonstrates the basic steps to create a Criteria query:

1. Use an EntityManager instance to create a CriteriaBuilder object.
2. Create a query object by creating an instance of the CriteriaQuery interface. This query object's attributes will be modified with the details of the query.
3. Set the query root by calling the from method on the CriteriaQuery object.
4. Specify what the type of the query result will be by calling the select method of the CriteriaQuery object.
5. Prepare the query for execution by creating a TypedQuery<T> instance, specifying the type of the query result.
6. Execute the query by calling the getResultList method on the TypedQuery<T> object. Because this query returns a collection of entities, the result is stored in a List.

The tasks associated with each step are discussed in detail in this chapter.

To create a CriteriaBuilder instance, call the getCriteriaBuilder method on the EntityManager instance:

CriteriaBuilder cb = em.getCriteriaBuilder();

The query object is created by using the CriteriaBuilder instance:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

The query will return instances of the Pet entity, so the type of the query is specified when the CriteriaQuery object is created to create a typesafe query.

The FROM clause of the query is set, and the root of the query specified, by calling the from method of the query object:

Root<Pet> pet = cq.from(Pet.class);

The SELECT clause of the query is set by calling the select method of the query object and passing in the query root:

cq.select(pet);

The query object is now used to create a TypedQuery<T> object that can be executed against the data source. The modifications to the query object are captured to create a ready-to-execute query:

TypedQuery<Pet> q = em.createQuery(cq);
This typed query object is executed by calling its getResultList method, because this query will return multiple entity instances. The results are stored in a List<Pet> collection-valued object.

```java
List<Pet> allPets = q.getResultList();
```

## Using the Metamodel API to Model Entity Classes

The Metamodel API is used to create a metamodel of the managed entities in a particular persistence unit. For each entity class in a particular package, a metamodel class is created with a trailing underscore and with attributes that correspond to the persistent fields or properties of the entity class.

The following entity class, `com.example.Pet`, has four persistent fields: id, name, color, and owners:

```java
package com.example;
...
@Entity
public class Pet {
    @Id
    protected Long id;
    protected String name;
    protected String color;
    @ManyToOne
    protected Set<Owner> owners;
    ...
}
```

The corresponding Metamodel class is:

```java
package com.example;
...
@Static Metamodel(Pet.class)
public class Pet_ {
    public static volatile SingularAttribute<Pet, Long> id;
    public static volatile SingularAttribute<Pet, String> name;
    public static volatile SingularAttribute<Pet, String> color;
    public static volatile SetAttribute<Pet, Owner> owners;
}
```

The metamodel class and its attributes are used in Criteria queries to refer to the managed entity classes and their persistent state and relationships.
Using Metamodel Classes

Metamodel classes that correspond to entity classes are of the following type:

```java
javax.persistence.metamodel.EntityType<T>
```

Metamodel classes are typically generated by annotation processors either at development time or at runtime. Developers of applications that use Criteria queries may generate static metamodel classes by using the persistence provider’s annotation processor or may obtain the metamodel class by either calling the `getModel` method on the query root object or first obtaining an instance of the `Metamodel` interface and then passing the entity type to the instance’s `entity` method.

The following code snippet shows how to obtain the Pet entity’s metamodel class by calling `Root<T>.getModel`:

```java
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
EntityType<Pet> Pet_ = pet.getModel();
```

The following code snippet shows how to obtain the Pet entity’s metamodel class by first obtaining a metamodel instance by using `EntityManager.getMetamodel` and then calling `entity` on the metamodel instance:

```java
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
```

**Note** – The most common use case is to generate type-safe static metamodel classes at development time. Obtaining the metamodel classes dynamically, by calling `Root<T>.getModel` or `EntityManager.getMetamodel` and then the `entity` method, doesn’t allow for type-safety and doesn’t allow the application to call persistent field or property names on the metamodel class.

Using the Criteria API and Metamodel API to Create Basic Typesafe Queries

The basic semantics of a Criteria query consists of a `SELECT` clause, a `FROM` clause, and an optional `WHERE` clause, similar to a JPQL query. Criteria queries set these clauses by using Java programming language objects, so the query can be created in a typesafe manner.
Creating a Criteria Query

The `javax.persistence.criteria.CriteriaBuilder` interface is used to construct

- Criteria queries
- Selections
- Expressions
- Predicates
- Ordering

To obtain an instance of the `CriteriaBuilder` interface, call the `getCriteriaBuilder` method on either an `EntityManager` or an `EntityManagerFactory` instance.

The following code shows how to obtain a `CriteriaBuilder` instance by using the `EntityManager.getCriteriaBuilder` method.

```java
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Criteria queries are constructed by obtaining an instance of the following interface:

```java
javax.persistence.criteria.CriteriaQuery
```

`CriteriaQuery` objects define a particular query that will navigate over one or more entities. Obtain `CriteriaQuery` instances by calling one of the `CriteriaBuilder.createQuery` methods. For creating typesafe queries, call the `CriteriaBuilder.createQuery` method as follows:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
```

The `CriteriaQuery` object's type should be set to the expected result type of the query. In the preceding code, the object's type is set to `CriteriaQuery<Pet>` for a query that will find instances of the `Pet` entity.

In the following code snippet, a `CriteriaQuery` object is created for a query that returns a `String`:

```java
CriteriaQuery<String> cq = cb.createQuery(String.class);
```

Query Roots

For a particular `CriteriaQuery` object, the root entity of the query, from which all navigation originates, is called the *query root*. It is similar to the `FROM` clause in a JPQL query.

Create the query root by calling the `from` method on the `CriteriaQuery` instance. The argument to the `from` method is either the entity class or an `EntityType<T>` instance for the entity.
The following code sets the query root to the Pet entity:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

The following code sets the query root to the Pet class by using an EntityType<T> instance:

```java
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet_);
```

Criteria queries may have more than one query root. This usually occurs when the query navigates from several entities.

The following code has two Root instances:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet1 = cq.from(Pet.class);
Root<Pet> pet2 = cq.from(Pet.class);
```

## Querying Relationships Using Joins

For queries that navigate to related entity classes, the query must define a join to the related entity by calling one of the `From.join` methods on the query root object or another join object. The `join` methods are similar to the `JOIN` keyword in JPQL.

The target of the join uses the Metamodel class of type `EntityType<T>` to specify the persistent field or property of the joined entity.

The `join` methods return an object of type `Join<X, Y>`, where X is the source entity and Y is the target of the join. In the following code snippet, Pet is the source entity, Owner is the target, and Pet_ is a statically generated metamodel class:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = pet.join(Pet_.owners);
```

Joins can be chained together to navigate to related entities of the target entity without having to create a `Join<X, Y>` instance for each join:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = cq.join(Pet_.owners).join(Owner_.addresses);
```
Path Navigation in Criteria Queries

Path objects are used in the SELECT and WHERE clauses of a Criteria query and can be query root entities, join entities, or other Path objects. The Path.get method is used to navigate to attributes of the entities of a query.

The argument to the get method is the corresponding attribute of the entity’s Metamodel class. The attribute can either be a single-valued attribute, specified by @SingularAttribute in the Metamodel class, or a collection-valued attribute, specified by one of @CollectionAttribute, @SetAttribute, @ListAttribute, or @MapAttribute.

The following query returns the names of all the pets in the data store. The get method is called on the query root, pet, with the name attribute of the Pet entity’s Metamodel class, Pet_.name, as the argument:

```java
CriteriaQuery<String> cq = cb.createQuery(String.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet.get(Pet_.name));
```

Restricting Criteria Query Results

The results of a query can be restricted on the CriteriaQuery object according to conditions set by calling the CriteriaQuery.where method. Calling the where method is analogous to setting the WHERE clause in a JPQL query.

The where method evaluates instances of the Expression interface to restrict the results according to the conditions of the expressions. Expression instances are created by using methods defined in the Expression and CriteriaBuilder interfaces.

The Expression Interface Methods

An Expression object is used in a query’s SELECT, WHERE, or HAVING clause. Table 35–1 shows conditional methods you can use with Expression objects.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isNull</td>
<td>Tests whether an expression is null</td>
</tr>
<tr>
<td>isNotNull</td>
<td>Tests whether an expression is not null</td>
</tr>
<tr>
<td>in</td>
<td>Tests whether an expression is within a list of values</td>
</tr>
</tbody>
</table>

The following query uses the Expression.isNotNull method to find all pets where the color attribute is null:

```java
CriteriaQuery<String> cq = cb.createQuery(String.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet.get(Pet_.name));
cq.where(pet.get(Pet_.color).isNotNull());
```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).isNull());

The following query uses the Expression.in method to find all brown and black pets:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).in("brown", "black"));

The in method also can check whether an attribute is a member of a collection.

**Expression Methods in the CriteriaBuilder Interface**

The CriteriaBuilder interface defines additional methods for creating expressions. These methods correspond to the arithmetic, string, date, time, and case operators and functions of JPQL. Table 35–2 shows conditional methods you can use with CriteriaBuilder objects.

<table>
<thead>
<tr>
<th>Conditional Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>Tests whether two expressions are equal</td>
</tr>
<tr>
<td>notEqual</td>
<td>Tests whether two expressions are not equal</td>
</tr>
<tr>
<td>gt</td>
<td>Tests whether the first numeric expression is greater than the second numeric expression</td>
</tr>
<tr>
<td>ge</td>
<td>Tests whether the first numeric expression is greater than or equal to the second numeric expression</td>
</tr>
<tr>
<td>lt</td>
<td>Tests whether the first numeric expression is less than the second numeric expression</td>
</tr>
<tr>
<td>le</td>
<td>Tests whether the first numeric expression is less than or equal to the second numeric expression</td>
</tr>
<tr>
<td>between</td>
<td>Tests whether the first expression is between the second and third expression in value</td>
</tr>
<tr>
<td>like</td>
<td>Tests whether the expression matches a given pattern</td>
</tr>
</tbody>
</table>

The following code uses the CriteriaBuilder.equal method:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
...

The following code uses the CriteriaBuilder.gt method:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date someDate = new Date(...);
cq.where(cb.gt(pet.get(Pet_.birthday), date));
The following code uses the `CriteriaBuilder.between` method:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate));
```

The following code uses the `CriteriaBuilder.like` method:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.like(pet.get(Pet_.name), "*do"));
```

Multiple conditional predicates can be specified by using the compound predicate methods of the `CriteriaBuilder` interface, as shown in Table 35–3.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>A logical conjunction of two Boolean expressions</td>
</tr>
<tr>
<td>or</td>
<td>A logical disjunction of two Boolean expressions</td>
</tr>
<tr>
<td>not</td>
<td>A logical negation of the given Boolean expression</td>
</tr>
</tbody>
</table>

The following code shows the use of compound predicates in queries:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido")
    .and(cb.equal(pet.get(Pet_.color), "brown"));
```

### Managing Criteria Query Results

For queries that return more than one result, it's often helpful to organize those results. The `CriteriaQuery` interface defines the `orderBy` method to order query results according to attributes of an entity. The `CriteriaQuery` interface also defines the `groupBy` method to group the results of a query together according to attributes of an entity, and the `having` method to restrict those groups according to a condition.

### Ordering Results

The order of the results of a query can be set by calling the `CriteriaQuery.orderBy` method and passing in an `Order` object. `Order` objects are created by calling either the `CriteriaBuilder.asc` or the `CriteriaBuilder.desc` method. The `asc` method is used to
order the results by ascending value of the passed expression parameter. The desc method is used to order the results by descending value of the passed expression parameter. The following query shows the use of the desc method:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get(Pet_.birthday)));

In this query, the results will be ordered by the pet’s birthday from highest to lowest. That is, pets born in December will appear before pets born in May.

The following query shows the use of the asc method:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = cq.join(Pet_.owners).join(Owner_.address);
cq.select(pet);
cq.orderBy(cb.asc(address.get(Address_.postalCode)));

In this query, the results will be ordered by the pet owner’s postal code from lowest to highest. That is, pets whose owner lives in the 10001 zip code will appear before pets whose owner lives in the 91000 zip code.

If more than one Order object is passed to orderBy, the precedence is determined by the order in which they appear in the argument list of orderBy. The first Order object has precedence.

The following code orders results by multiple criteria:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner = cq.join(Pet_.owners);
cq.select(pet);
cq.orderBy(cb.asc(owner.get(Owner_.lastName), owner.get(Owner_.firstName)));

The results of this query will be ordered alphabetically by the pet owner’s last name, then first name.

**Grouping Results**

The CriteriaQuery.groupBy method partitions the query results into groups. These groups are set by passing an expression to groupBy:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));

This query returns all Pet entities and groups the results by the pet’s color.

The CriteriaQuery.having method is used in conjunction with groupBy to filter over the groups. The having method takes a conditional expression as a parameter. By calling the having method, the query result is restricted according to the conditional expression:
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
cq.having(cb.in(pet.get(Pet_.color)).value("brown").value("blonde"));

In this example, the query groups the returned Pet entities by color, as in the preceding example. However, the only returned groups will be Pet entities where the color attribute is set to brown or blonde. That is, no gray-colored pets will be returned in this query.

**Executing Queries**

To prepare a query for execution, create a TypedQuery<T> object with the type of the query result by passing the CriteriaQuery object to EntityManager.createQuery.

Queries are executed by calling either getSingleResult or getResultList on the TypedQuery<T> object.

**Single-Valued Query Results**

The TypedQuery<T>.getSingleResult method is used for executing queries that return a single result:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
Pet result = q.getSingleResult();

**Collection-Valued Query Results**

The TypedQuery<T>.getResultList method is used for executing queries that return a collection of objects:

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
...
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
Creating and Using String-Based Criteria Queries

This chapter describes how to create weakly-typed string-based Criteria API queries.

The following topics are addressed here:

- “Overview of String-Based Criteria API Queries” on page 673
- “Creating String-Based Queries” on page 674
- “Executing String-Based Queries” on page 675

Overview of String-Based Criteria API Queries

String-based Criteria API queries ("string-based queries") are Java programming language queries that use strings rather than strongly-typed metamodel objects to specify entity attributes when traversing a data hierarchy. String-based queries are constructed similarly to metamodel queries, can be static or dynamic, and can express the same kind of queries and operations as strongly-typed metamodel queries.

Strongly-typed metamodel queries are the preferred method of constructing Criteria API queries. The main advantage of string-based queries over metamodel queries is the ability to construct Criteria queries at development time without the need to generate static metamodel classes or otherwise access dynamically generated metamodel classes. The main disadvantage to string-based queries is their lack of type safety, which may lead to runtime errors due to type mismatches that would be caught at development time when using strongly-typed metamodel queries.

For information on constructing criteria queries, see Chapter 35, “Using the Criteria API to Create Queries.”
Creating String-Based Queries

To create a string-based query, specify the attribute names of entity classes directly as strings, rather than the attributes of the metamodel class. For example, this query finds all Pet entities where the value of the name attribute is Fido:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get("name"), "Fido"));
```

The name of the attribute is specified as a string. This query is the equivalent of the following metamodel query:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido"));
```

**Note** – Type mismatch errors in string-based queries won’t appear until the code is executed at runtime, unlike in the above metamodel query, where type mismatches will be caught at compile time.

Joins are specified in the same way:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address = pet.join("owners").join("addresses");
```

All the conditional expressions, method expressions, path navigation methods, and result restriction methods used in metamodel queries can be used in string-based queries. In each case, the attributes are specified using strings. For example, here is a string-based query that uses the in expression:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get("color").in("brown", "black"));
```

Here is a string-based query that orders the results in descending order by date:

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy(cb.desc(pet.get("birthday")));
Executing String-Based Queries

String-based queries are executed similarly to strongly-typed Criteria queries. First create a `javax.persistence.TypedQuery` object by passing the criteria query object to the `EntityManager.createQuery` method and then call either `getSingleResult` or `getResultList` on the query object to execute the query.

```java
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
 cq.where(cb.equal(pet.get("name"), "Fido"));
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> results = q.getResultList();
```
Controlling Concurrent Access to Entity Data with Locking

This chapter details how to handle concurrent access to entity data, and the locking strategies available to Java Persistence API application developers.

The following topics are addressed here:

- “Overview of Entity Locking and Concurrency” on page 677
- “Lock Modes” on page 679

Overview of Entity Locking and Concurrency

Entity data is concurrently accessed if the data in a data source is accessed at the same time by multiple applications. Special care must be taken to ensure that the underlying data’s integrity is preserved when accessed concurrently.

When data is updated in the database tables in a transaction, the persistence provider assumes the database management system will hold short-term read locks and long-term write locks to maintain data integrity. Most persistence providers will delay database writes until the end of the transaction, except when the application explicitly calls for a flush (that is, the application calls the EntityManager.flush method or executes queries with the flush mode set to AUTO).

By default, persistence providers use optimistic locking, where, before committing changes to the data, the persistence provider checks that no other transaction has modified or deleted the data since the data was read. This is accomplished by a version column in the database table, with a corresponding version attribute in the entity class. When a row is modified, the version value is incremented. The original transaction checks the version attribute, and if the data has been modified by another transaction, a javax.persistence.OptimisticLockException will be thrown, and the original transaction will be rolled back. When the application specifies optimistic lock modes, the persistence provider verifies that a particular entity has not changed since it was read from the database even if the entity data was not modified.

Pessimistic locking goes further than optimistic locking. With pessimistic locking, the persistence provider creates a transaction that obtains a long-term lock on the data until the
transaction is completed, which prevents other transactions from modifying or deleting the
data until the lock has ended. Pessimistic locking is a better strategy than optimistic locking
when the underlying data is frequently accessed and modified by many transactions.

**Note** – Using pessimistic locks on entities that are not subject to frequent modification may
result in decreased application performance.

### Using Optimistic Locking

The `javax.persistence.Version` annotation is used to mark a persistent field or property as a
version attribute of an entity. By adding a version attribute, the entity is enabled for optimistic
concurrency control. The version attribute is read and updated by the persistence provider
when an entity instance is modified during a transaction. The application may read the version
attribute, but *must not* modify the value.

**Note** – Although some persistence providers may support optimistic locking for entities that do
not have version attributes, portable applications should always use entities with version
attributes when using optimistic locking. If the application attempts to lock an entity without a
version attribute, and the persistence provider doesn’t support optimistic locking for
non-versioned entities, a `PersistenceException` will be thrown.

The `@Version` annotation has the following requirements:

- Only a single `@Version` attribute may be defined per entity.
- The `@Version` attribute must be in the primary table for an entity mapped to multiple tables.
- The type of the `@Version` attribute must be one of the following: `int`, `Integer`, `long`, `Long`,
  `short`, `Short`, or `java.sql.Timestamp`.

The following code snippet shows how to define a version attribute in an entity with persistent
fields:

```java
@Version
protected int version;
```
The following code snippet shows how to define a version attribute in an entity with persistent properties:

```java
@Version
protected Short getVersion() { ... }
```

## Lock Modes

The application may increase the level of locking for an entity by specifying the use of lock modes. Lock modes may be specified to increase the level of optimistic locking or to request the use of pessimistic locks.

The use of optimistic lock modes causes the persistence provider to check the version attributes for entities that were read (but not modified) during a transaction as well as for entities that were updated.

The use of pessimistic lock modes specifies that the persistence provider is to immediately acquire long-term read or write locks for the database data corresponding to entity state.

The lock mode for an entity operation may be set by specifying one of the lock modes defined in the `javax.persistence.LockModeType` enumerated type, listed in Table 37–1.

### Table 37–1  Lock Modes for Concurrent Entity Access

<table>
<thead>
<tr>
<th>Lock Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIMISTIC</td>
<td>Obtain an optimistic read lock for all entities with version attributes.</td>
</tr>
<tr>
<td>OPTIMISTIC_FORCE_INCREMENT</td>
<td>Obtain an optimistic read lock for all entities with version attributes,</td>
</tr>
<tr>
<td></td>
<td>and increment the version attribute value.</td>
</tr>
<tr>
<td>PESSIMISTIC_READ</td>
<td>Immediately obtain a long-term read lock on the data to prevent the data</td>
</tr>
<tr>
<td></td>
<td>from being modified or deleted. Other transactions may read the data while</td>
</tr>
<tr>
<td></td>
<td>the lock is maintained, but may not modify or delete the data.</td>
</tr>
<tr>
<td></td>
<td>The persistence provider is permitted to obtain a database write lock</td>
</tr>
<tr>
<td></td>
<td>when a read lock was requested, but not vice versa.</td>
</tr>
<tr>
<td>PESSIMISTIC_WRITE</td>
<td>Immediately obtain a long-term write lock on the data to prevent the data</td>
</tr>
<tr>
<td></td>
<td>from being read, modified, or deleted.</td>
</tr>
<tr>
<td>PESSIMISTIC_FORCE_INCREMENT</td>
<td>Immediately obtain a long-term lock on the data to prevent the data</td>
</tr>
<tr>
<td></td>
<td>from being modified or deleted, and increment the version attribute</td>
</tr>
<tr>
<td></td>
<td>of versioned entities.</td>
</tr>
<tr>
<td>READ</td>
<td>A synonym for OPTIMISTIC. Use of LockModeType.OPTIMISTIC is to be</td>
</tr>
<tr>
<td></td>
<td>preferred for new applications.</td>
</tr>
</tbody>
</table>
### Lock Modes for Concurrent Entity Access (Continued)

<table>
<thead>
<tr>
<th>Lock Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE</td>
<td>A synonym for OPTIMISTIC_FORCE_INCREMENT. Use of LockModeType.OPTIMISTIC_FORCE_INCREMENT is to be preferred for new applications.</td>
</tr>
<tr>
<td>NONE</td>
<td>No additional locking will occur on the data in the database.</td>
</tr>
</tbody>
</table>

---

### Setting the Lock Mode

The lock mode may be specified by one of the following techniques:

- Calling the `EntityManager.lock` and passing in one of the lock modes:
  ```java
  EntityManager em = ...;
  Person person = ...;
  em.lock(person, LockModeType.OPTIMISTIC);
  ```

- Calling one of the `EntityManager.find` methods that takes the lock mode as a parameter:
  ```java
  EntityManager em = ...;
  String personPK = ...;
  Person person = em.find(Person.class, personPK, LockModeType.PESSIMISTIC_WRITE);
  ```

- Calling one of the `EntityManager.refresh` methods that takes the lock mode as a parameter:
  ```java
  EntityManager em = ...;
  String personPK = ...;
  Person person = em.find(Person.class, personPK);
  ...
  em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
  ```

- Calling the `Query.setLockMode` or `TypedQuery.setLockMode` method, passing the lock mode as the parameter:
  ```java
  Query q = em.createQuery(...);
  q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
  ```

- Adding a `lockMode` element to the `@NamedQuery` annotation:
  ```java
  @NamedQuery(name="lockPersonQuery",
  query="SELECT p FROM Person p WHERE p.name LIKE :name",
  lockMode=PESSIMISTIC_READ)
  ```

---

### Using Pessimistic Locking

Versioned entities as well as entities that do not have version attributes can be locked pessimistically.

To lock entities pessimistically, set the lock mode to `PESSIMISTIC_READ`, `PESSIMISTIC_WRITE`, or `PESSIMISTIC_FORCE_INCREMENT`.
If a pessimistic lock cannot be obtained on the database rows, and the failure to lock the data results in a transaction rollback, a PessimisticLockException is thrown. If a pessimistic lock cannot be obtained, but the locking failure doesn’t result in a transaction rollback, a LockTimeoutException is thrown.

Pessimistically locking a version entity with PESSIMISTIC_FORCE_INCREMENT results in the version attribute being incremented even if the entity data is unmodified. When pessimistically locking a versioned entity, the persistence provider will perform the version checks that occur during optimistic locking, and if the version check fails, an OptimisticLockException will be thrown. Attempting to lock a non-versioned entity with PESSIMISTIC_FORCE_INCREMENT is not portable and may result in a PersistenceException if the persistence provider doesn’t support optimistic locks for non-versioned entities. Locking a versioned entity with PESSIMISTIC_WRITE results in the version attribute being incremented if the transaction was successfully committed.

**Pessimistic Locking Timeouts**

The length of time in milliseconds the persistence provider should wait to obtain a lock on the database tables may be specified using the javax.persistence.lock.timeout property. If the time it takes to obtain a lock exceeds the value of this property, a LockTimeoutException will be thrown, but the current transaction will not be marked for rollback. If this property is set to 0, the persistence provider should throw a LockTimeoutException if it cannot immediately obtain a lock.

*Note* – Portable applications should not rely on the setting of javax.persistence.lock.timeout, as the locking strategy and underlying database may mean that the timeout value cannot be used. The value of javax.persistence.lock.timeout is a hint, not a contract.

This property may be set programmatically by passing it to the EntityManager methods that allow lock modes to be specified, the Query.setLockMode and TypedQuery.setLockMode methods, the @NamedQuery annotation, and as a property to the Persistence.createEntityManagerFactory method. It may also be set as a property in the persistence.xml deployment descriptor.

If javax.persistence.lock.timeout is set in multiple places, the value will be determined in the following order:

1. The argument to one of the EntityManager or Query methods.
2. The setting in the @NamedQuery annotation.
3. The argument to the Persistence.createEntityManagerFactory method.
4. The value in the persistence.xml deployment descriptor.
Using a Second-Level Cache with Java Persistence API Applications

This chapter explains how to modify the second-level cache mode settings to improve the performance of applications that use the Java Persistence API.

The following topics are addressed here:
- “Overview of the Second-Level Cache” on page 683
- “Specifying the Cache Mode Settings to Improve Performance” on page 685

Overview of the Second-Level Cache

A second-level cache is a local store of entity data managed by the persistence provider to improve application performance. A second-level cache helps improve performance by avoiding expensive database calls, keeping the entity data local to the application. A second-level cache is typically transparent to the application, as it is managed by the persistence provider and underlies the persistence context of an application. That is, the application reads and commits data through the normal entity manager operations without knowing about the cache.

Note – Persistence providers are not required to support a second-level cache. Portable applications should not rely on support by persistence providers for a second-level cache.

The second-level cache for a persistence unit may be configured to one of several second-level cache modes. The following cache mode settings are defined by the Java Persistence API.

<table>
<thead>
<tr>
<th>Cache Mode Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>All entity data is stored in the second-level cache for this persistence unit.</td>
</tr>
</tbody>
</table>
One consequence of using a second-level cache in an application is that the underlying data may have changed in the database tables, while the value in the cache has not, a circumstance called a stale read. Stale reads may be avoided by changing the second-level cache to one of the cache mode settings, controlling which entities may be cached (described in “Controlling Whether Entities May Be Cached” on page 684), or changing the cache’s retrieval or store modes (described in “Setting the Cache Retrieval and Store Modes” on page 686). Which strategies best avoid stale reads are application dependent.

### Controlling Whether Entities May Be Cached

The `javax.persistence.Cacheable` annotation is used to specify that an entity class, and any subclasses, may be cached when using the `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE` cache modes. Subclasses may override the `@Cacheable` setting by adding a `@Cacheable` annotation and changing the value.

To specify that an entity may be cached, add a `@Cacheable` annotation at the class level:

```java
@Cacheable
@Entity
public class Person { ... }
```

By default, the `@Cacheable` annotation is `true`. The following example is equivalent:

```java
@Cacheable(true)
@Entity
public class Person { ... }
```
To specify that an entity must not be cached, add a `@Cacheable` annotation and set it to `false`:

```java
@Cacheable(false)
@Entity
public class OrderStatus { ... }
```

When the `ENABLE_SELECTIVE` cache mode is set, the persistence provider will cache any entities that have the `@Cacheable(true)` annotation and any subclasses of that entity that have not been overridden. The persistence provider will not cache entities that have `@Cacheable(false)` or have no `@Cacheable` annotation. That is, the `ENABLE_SELECTIVE` mode will cache only entities that have been explicitly marked for the cache using the `@Cacheable` annotation.

When the `DISABLE_SELECTIVE` cache mode is set, the persistence provider will cache any entities that do not have the `@Cacheable(false)` annotation. Entities that do not have `@Cacheable` annotations, and entities with the `@Cacheable(true)` annotation will be cached. That is, the `DISABLE_SELECTIVE` mode will cache all entities that have not been explicitly prevented from being cached.

If the cache mode is set to `UNDEFINED`, or is left unset, the behavior of entities annotated with `@Cacheable` is undefined. If the cache mode is set to `ALL` or `NONE`, the value of the `@Cacheable` annotation is ignored by the persistence provider.

---

**Specifying the Cache Mode Settings to Improve Performance**

To adjust the cache mode settings for a persistence unit, specify one of the cache modes as the value of the `shared-cache-mode` element in the `persistence.xml` deployment descriptor (shown in **bold**):

```xml
<persistence-unit name="examplePU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/__default</jta-data-source>
    <shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
```

**Note** – Because support for a second-level cache is not required by the Java Persistence API specification, setting the second-level cache mode in `persistence.xml` will have no effect when using a persistence provider that does not implement a second-level cache.
Alternatively, the shared cache mode may be specified by setting the `javax.persistence.sharedCache.mode` property to one of the shared cache mode settings:

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myExamplePU", new Properties().add("javax.persistence.sharedCache.mode", "ENABLE_SELECTIVE"));
```

## Setting the Cache Retrieval and Store Modes

If the second-level cache has been enabled for a persistence unit by setting the shared cache mode, the behavior of the second-level cache can be further modified by setting the `javax.persistence.cache.retrieveMode` and `javax.persistence.cache.storeMode` properties. These properties may be set at the persistence context level by passing the property name and value to the `EntityManager.setProperty` method, or may be set on a per-`EntityManager` operation (`EntityManager.find` or `EntityManager.refresh`) or per-query level.

### Cache Retrieval Mode

The cache retrieval mode, set by the `javax.persistence.retrieveMode` property, controls how data is read from the cache for calls to the `EntityManager.find` method and from queries.

The `retrieveMode` property can be set to one of the constants defined by the `javax.persistence.CacheRetrieveMode` enumerated type, either `USE` (the default) or `BYPASS`. When it is set to `USE`, data is retrieved from the second-level cache, if available. If the data is not in the cache, the persistence provider will read it from the database. When it is set to `BYPASS`, the second-level cache is bypassed and a call to the database is made to retrieve the data.

### Cache Store Mode

The cache store mode, set by the `javax.persistence.storeMode` property, controls how data is stored in the cache.

The `storeMode` property can be set to one of the constants defined by the `javax.persistence.CacheStoreMode` enumerated type, either `USE` (the default), `BYPASS`, or `REFRESH`. When set to `USE` the cache data is created or updated when data is read from or committed to the database. If data is already in the cache, setting the store mode to `USE` will not force a refresh when data is read from the database.

When the store mode is set to `BYPASS`, data read from or committed to the database is not inserted or updated in the cache. That is, the cache is unchanged.

When the store mode is set to `REFRESH`, the cache data is created or updated when data is read from or committed to the database, and a refresh is forced on data in the cache upon database reads.
**Setting the Cache Retrieval or Store Mode**

To set the cache retrieval or store mode for the persistence context, call the `EntityManager.setProperty` method with the property name and value pair:

```java
EntityManager em = ...;
em.setProperty("javax.persistence.cache.storeMode", "BYPASS");
```

To set the cache retrieval or store mode when calling the `EntityManager.find` or `EntityManager.refresh` methods, first create a `Map<String, Object>` instance and add a name/value pair as follows:

```java
EntityManager em = ...;
Map<String, Object> props = new HashMap<String, Object>();
props.put("javax.persistence.cache.retrieveMode", "BYPASS");
String personPK = ...;
Person person = em.find(Person.class, personPK, props);
```

**Note** – The cache retrieve mode is ignored when calling the `EntityManager.refresh` method, as calls to `refresh` always result in data being read from the database, not the cache.

To set the retrieval or store mode when using queries, call the `Query.setHint` or `TypedQuery.setHint` methods, depending on the type of query:

```java
EntityManager em = ...;
CriteriaQuery<Person> cq = ...;
TypedQuery<Person> q = em.createQuery(cq);
q.setHint("javax.persistence.cache.storeMode", "REFRESH");
...
```

Setting the store or retrieve mode in a query or when calling the `EntityManager.find` or `EntityManager.refresh` method overrides the setting of the entity manager.

**Controlling the Second-Level Cache Programmatically**

The `javax.persistence.Cache` interface defines methods for interacting with the second-level cache programmatically. The `Cache` interface defines methods to check whether a particular entity has cached data, to remove a particular entity from the cache, to remove all instances (and instances of subclasses) of an entity class from the cache, and to clear the cache of all entity data.

**Note** – If the second-level cache has been disabled, calls to the `Cache` interface’s methods have no effect, except for `contains`, which will always return `false`. 
Checking Whether an Entity’s Data Is Cached

Call the Cache.contains method to find out whether a given entity is currently in the second-level cache. The contains method returns true if the entity’s data is cached, and false if the data is not in the cache.

```java
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...
if (cache.contains(Person.class, personPK)) {
    // the data is cached
} else {
    // the data is NOT cached
}
```

Removing an Entity from the Cache

Call one of the Cache.evict methods to remove a particular entity or all entities of a given type from the second-level cache. To remove a particular entity from the cache, call the evict method and pass in the entity class and the primary key of the entity:

```java
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
String personPK = ...
cache.evict(Person.class, personPK);
```

To remove all instances of a particular entity class, including subclasses, call the evict method and specify the entity class:

```java
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evict(Person.class);
```

All instances of the Person entity class will be removed from the cache. If the Person entity has a subclass, Student, calls to the above method will remove all instances of Student from the cache as well.

Removing All Data from the Cache

Call the Cache.evictAll method to completely clear the second-level cache:

```java
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();
cache.evictAll();
```
Security

Part VII explores security concepts and examples. This part contains the following chapters:

- Chapter 39, “Introduction to Security in the Java EE Platform”
- Chapter 40, “Getting Started Securing Web Applications”
- Chapter 41, “Getting Started Securing Enterprise Applications”
- Chapter 42, “Java EE Security: Advanced Topics”
Every enterprise that has either sensitive resources that can be accessed by many users or resources that traverse unprotected, open, networks, such as the Internet, needs to be protected. This chapter introduces basic security concepts and security mechanisms. More information on these concepts and mechanisms can be found in the chapter on security in the Java EE 6 specification. This document is available for download online at http://www.jcp.org/en/jsr/detail?id=316.

Other chapters in Part VII discuss security requirements in web tier and enterprise tier applications:

- Chapter 40, “Getting Started Securing Web Applications,” explains how to add security to web components, such as servlets.
- Chapter 41, “Getting Started Securing Enterprise Applications,” explains how to add security to Java EE components, such as enterprise beans and application clients.

Some of the material in this chapter assumes that you understand basic security concepts. To learn more about these concepts before you begin this chapter, you should explore the Java SE security web site at http://docs.oracle.com/javase/6/docs/technotes/guides/security/.

The following topics are addressed here:

- “Overview of Java EE Security” on page 692
- “Security Mechanisms” on page 697
- “Securing Containers” on page 700
- “Securing the GlassFish Server” on page 702
- “Working with Realms, Users, Groups, and Roles” on page 702
- “Establishing a Secure Connection Using SSL” on page 710
- “Further Information about Security” on page 711
Overview of Java EE Security

Enterprise tier and web tier applications are made up of components that are deployed into various containers. These components are combined to build a multitier enterprise application. Security for components is provided by their containers. A container provides two kinds of security: declarative and programmatic.

- **Declarative security** expresses an application component’s security requirements by using either deployment descriptors or annotations.
  
  A deployment descriptor is an XML file that is external to the application and that expresses an application’s security structure, including security roles, access control, and authentication requirements. For more information about deployment descriptors, read “Using Deployment Descriptors for Declarative Security” on page 701.
  
  Annotations, also called metadata, are used to specify information about security within a class file. When the application is deployed, this information can be either used by or overridden by the application deployment descriptor. Annotations save you from having to write declarative information inside XML descriptors. Instead, you simply put annotations on the code, and the required information gets generated. For this tutorial, annotations are used for securing applications wherever possible. For more information about annotations, see “Using Annotations to Specify Security Information” on page 700.

- **Programmatic security** is embedded in an application and is used to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application. For more information about programmatic security, read “Using Programmatic Security” on page 701.

A Simple Application Security Walkthrough

The security behavior of a Java EE environment may be better understood by examining what happens in a simple application with a web client, a user interface, and enterprise bean business logic.

In the following example, which is taken from the Java EE 6 Specification, the web client relies on the web server to act as its authentication proxy by collecting user authentication data from the client and using it to establish an authenticated session.

Step 1: Initial Request

In the first step of this example, the web client requests the main application URL. This action is shown in Figure 39–1.
Since the client has not yet authenticated itself to the application environment, the server responsible for delivering the web portion of the application, hereafter referred to as the web server, detects this and invokes the appropriate authentication mechanism for this resource. For more information on these mechanisms, see “Security Mechanisms” on page 697.

**Step 2: Initial Authentication**

The web server returns a form that the web client uses to collect authentication data, such as user name and password, from the user. The web client forwards the authentication data to the web server, where it is validated by the web server, as shown in Figure 39–2. The validation mechanism may be local to a server or may leverage the underlying security services. On the basis of the validation, the web server sets a credential for the user.

**Step 3: URL Authorization**

The credential is used for future determinations of whether the user is authorized to access restricted resources it may request. The web server consults the security policy associated with the web resource to determine the security roles that are permitted access to the resource. The security policy is derived from annotations or from the deployment descriptor. The web container then tests the user’s credential against each role to determine whether it can map the user to the role. *Figure 39–3* shows this process.
The web server's evaluation stops with an “is authorized” outcome when the web server is able to map the user to a role. A “not authorized” outcome is reached if the web server is unable to map the user to any of the permitted roles.

**Step 4: Fulfilling the Original Request**

If the user is authorized, the web server returns the result of the original URL request, as shown in Figure 39–4.

In our example, the response URL of a web page is returned, enabling the user to post form data that needs to be handled by the business-logic component of the application. See Chapter 40, “Getting Started Securing Web Applications,” for more information on protecting web applications.

**Step 5: Invoking Enterprise Bean Business Methods**

The web page performs the remote method call to the enterprise bean, using the user’s credential to establish a secure association between the web page and the enterprise bean, as shown in Figure 39–5. The association is implemented as two related security contexts: one in the web server and one in the EJB container.
The EJB container is responsible for enforcing access control on the enterprise bean method. The container consults the security policy associated with the enterprise bean to determine the security roles that are permitted access to the method. The security policy is derived from annotations or from the deployment descriptor. For each role, the EJB container determines whether it can map the caller to the role by using the security context associated with the call.

The container’s evaluation stops with an “is authorized” outcome when the container is able to map the caller’s credential to a role. A “not authorized” outcome is reached if the container is unable to map the caller to any of the permitted roles. A “not authorized” result causes an exception to be thrown by the container and propagated back to the calling web page.

If the call is authorized, the container dispatches control to the enterprise bean method. The result of the bean’s execution of the call is returned to the web page and ultimately to the user by the web server and the web client.

**Features of a Security Mechanism**

A properly implemented security mechanism will provide the following functionality:

- Prevent unauthorized access to application functions and business or personal data (authentication)
- Hold system users accountable for operations they perform (non-repudiation)
- Protect a system from service interruptions and other breaches that affect quality of service

Ideally, properly implemented security mechanisms will also be

- Easy to administer
- Transparent to system users
- Interoperable across application and enterprise boundaries
Characteristics of Application Security

Java EE applications consist of components that can contain both protected and unprotected resources. Often, you need to protect resources to ensure that only authorized users have access. Authorization provides controlled access to protected resources. Authorization is based on identification and authentication. Identification is a process that enables recognition of an entity by a system, and authentication is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system.

Authorization and authentication are not required for an entity to access unprotected resources. Accessing a resource without authentication is referred to as unauthenticated, or anonymous, access.

The characteristics of application security that, when properly addressed, help to minimize the security threats faced by an enterprise include the following:

- **Authentication**: The means by which communicating entities, such as client and server, prove to each other that they are acting on behalf of specific identities that are authorized for access. This ensures that users are who they say they are.

- **Authorization, or access control**: The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints. This ensures that users have permission to perform operations or access data.

- **Data integrity**: The means used to prove that information has not been modified by a third party, an entity other than the source of the information. For example, a recipient of data sent over an open network must be able to detect and discard messages that were modified after they were sent. This ensures that only authorized users can modify data.

- **Confidentiality, or data privacy**: The means used to ensure that information is made available only to users who are authorized to access it. This ensures that only authorized users can view sensitive data.

- **Non-repudiation**: The means used to prove that a user who performed some action cannot reasonably deny having done so. This ensures that transactions can be proved to have happened.
Quality of Service: The means used to provide better service to selected network traffic over various technologies.

Auditing: The means used to capture a tamper-resistant record of security-related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms. To enable this, the system maintains a record of transactions and security information.

Security Mechanisms

The characteristics of an application should be considered when deciding the layer and type of security to be provided for applications. The following sections discuss the characteristics of the common mechanisms that can be used to secure Java EE applications. Each of these mechanisms can be used individually or with others to provide protection layers based on the specific needs of your implementation.

Java SE Security Mechanisms

Java SE provides support for a variety of security features and mechanisms:

- **Java Authentication and Authorization Service (JAAS):** JAAS is a set of APIs that enable services to authenticate and enforce access controls upon users. JAAS provides a pluggable and extensible framework for programmatic user authentication and authorization. JAAS is a core Java SE API and is an underlying technology for Java EE security mechanisms.

- **Java Generic Security Services (Java GSS-API):** Java GSS-API is a token-based API used to securely exchange messages between communicating applications. The GSS-API offers application programmers uniform access to security services atop a variety of underlying security mechanisms, including Kerberos.

- **Java Cryptography Extension (JCE):** JCE provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. Block ciphers operate on groups of bytes; stream ciphers operate on one byte at a time. The software also supports secure streams and sealed objects.

- **Java Secure Sockets Extension (JSSE):** JSSE provides a framework and an implementation for a Java version of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication to enable secure Internet communications.

- **Simple Authentication and Security Layer (SASL):** SASL is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authentication data is to be exchanged but does not itself specify the contents of that data. SASL is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.
Java SE also provides a set of tools for managing keystores, certificates, and policy files; generating and verifying JAR signatures; and obtaining, listing, and managing Kerberos tickets.

For more information on Java SE security, visit [http://docs.oracle.com/javase/6/docs/technotes/guides/security/](http://docs.oracle.com/javase/6/docs/technotes/guides/security/).

## Java EE Security Mechanisms

Java EE security services are provided by the component container and can be implemented by using declarative or programmatic techniques (see “Securing Containers” on page 700). Java EE security services provide a robust and easily configured security mechanism for authenticating users and authorizing access to application functions and associated data at many different layers. Java EE security services are separate from the security mechanisms of the operating system.

### Application-Layer Security

In Java EE, component containers are responsible for providing application-layer security, security services for a specific application type tailored to the needs of the application. At the application layer, application firewalls can be used to enhance application protection by protecting the communication stream and all associated application resources from attacks.

Java EE security is easy to implement and configure and can offer fine-grained access control to application functions and data. However, as is inherent to security applied at the application layer, security properties are not transferable to applications running in other environments and protect data only while it is residing in the application environment. In the context of a traditional enterprise application, this is not necessarily a problem, but when applied to a web services application, in which data often travels across several intermediaries, you would need to use the Java EE security mechanisms along with transport-layer security and message-layer security for a complete security solution.

The advantages of using application-layer security include the following.

- Security is uniquely suited to the needs of the application.
- Security is fine grained, with application-specific settings.

The disadvantages of using application-layer security include the following.

- The application is dependent on security attributes that are not transferable between application types.
- Support for multiple protocols makes this type of security vulnerable.
- Data is close to or contained within the point of vulnerability.

For more information on providing security at the application layer, see “Securing Containers” on page 700.
Transport-Layer Security

Transport-layer security is provided by the transport mechanisms used to transmit information over the wire between clients and providers; thus, transport-layer security relies on secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). Transport security is a point-to-point security mechanism that can be used for authentication, message integrity, and confidentiality. When running over an SSL-protected session, the server and client can authenticate each other and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. Security is active from the time the data leaves the client until it arrives at its destination, or vice versa, even across intermediaries. The problem is that the data is not protected once it gets to the destination. One solution is to encrypt the message before sending.

Transport-layer security is performed in a series of phases, as follows.

- The client and server agree on an appropriate algorithm.
- A key is exchanged using public-key encryption and certificate-based authentication.
- A symmetric cipher is used during the information exchange.

Digital certificates are necessary when running HTTPS using SSL. The HTTPS service of most web servers will not run unless a digital certificate has been installed. Digital certificates have already been created for the GlassFish Server.

The advantages of using transport-layer security include the following.

- It is relatively simple, well-understood, standard technology.
- It applies to both a message body and its attachments.

The disadvantages of using transport-layer security include the following.

- It is tightly coupled with the transport-layer protocol.
- It represents an all-or-nothing approach to security. This implies that the security mechanism is unaware of message contents, so that you cannot selectively apply security to portions of the message as you can with message-layer security.
- Protection is transient. The message is protected only while in transit. Protection is removed automatically by the endpoint when it receives the message.
- It is not an end-to-end solution, simply point-to-point.

For more information on transport-layer security, see “Establishing a Secure Connection Using SSL” on page 710.

Message-Layer Security

In message-layer security, security information is contained within the SOAP message and/or SOAP message attachment, which allows security information to travel along with the message or attachment. For example, a portion of the message may be signed by a sender and encrypted
for a particular receiver. When sent from the initial sender, the message may pass through intermediate nodes before reaching its intended receiver. In this scenario, the encrypted portions continue to be opaque to any intermediate nodes and can be decrypted only by the intended receiver. For this reason, message-layer security is also sometimes referred to as end-to-end security.

The advantages of message-layer security include the following.

- Security stays with the message over all hops and after the message arrives at its destination.
- Security can be selectively applied to different portions of a message and, if using XML Web Services Security, to attachments.
- Message security can be used with intermediaries over multiple hops.
- Message security is independent of the application environment or transport protocol.

The disadvantage of using message-layer security is that it is relatively complex and adds some overhead to processing.

The GlassFish Server supports message security using Metro, a web services stack that uses Web Services Security (WSS) to secure messages. Because this message security is specific to Metro and is not a part of the Java EE platform, this tutorial does not discuss using WSS to secure messages. See the Metro User’s Guide at http://metro.java.net/guide/.

## Securing Containers

In Java EE, the component containers are responsible for providing application security. A container provides two types of security: declarative and programmatic.

### Using Annotations to Specify Security Information

Annotations enable a declarative style of programming and so encompass both the declarative and programmatic security concepts. Users can specify information about security within a class file by using annotations. The GlassFish Server uses this information when the application is deployed. Not all security information can be specified by using annotations, however. Some information must be specified in the application deployment descriptors.

Specific annotations that can be used to specify security information within an enterprise bean class file are described in “Securing an Enterprise Bean Using Declarative Security” on page 742. Chapter 40, “Getting Started Securing Web Applications,” describes how to use annotations to secure web applications where possible. Deployment descriptors are described only where necessary.

For more information on annotations, see “Further Information about Security” on page 711.
Using Deployment Descriptors for Declarative Security

Declarative security can express an application component’s security requirements by using deployment descriptors. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads the deployment descriptor and acts upon the corresponding application, module, or component accordingly. Deployment descriptors must provide certain structural information for each component if this information has not been provided in annotations or is not to be defaulted.

This part of the tutorial does not document how to create deployment descriptors; it describes only the elements of the deployment descriptor relevant to security. NetBeans IDE provides tools for creating and modifying deployment descriptors.

Different types of components use different formats, or schemas, for their deployment descriptors. The security elements of deployment descriptors discussed in this tutorial include the following.

- Web components may use a web application deployment descriptor named `web.xml`.
  The schema for web component deployment descriptors is provided in Chapter 14 of the Java Servlet 3.0 specification (JSR 315), which can be downloaded from [http://jcp.org/en/jsr/detail?id=315](http://jcp.org/en/jsr/detail?id=315).
- Enterprise JavaBeans components may use an EJB deployment descriptor named `META-INF/ejb-jar.xml`, contained in the EJB JAR file.
  The schema for enterprise bean deployment descriptors is provided in Chapter 19 of the EJB 3.1 specification (JSR 318), which can be downloaded from [http://jcp.org/en/jsr/detail?id=318](http://jcp.org/en/jsr/detail?id=318).

Using Programmatic Security

Programmatic security is embedded in an application and is used to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application. The API for programmatic security consists of methods of the EJBContext interface and the HttpServletRequest interface. These methods allow components to make business-logic decisions based on the security role of the caller or remote user.

Programmatic security is discussed in more detail in the following sections:

- “Using Programmatic Security with Web Applications” on page 724
- “Securing an Enterprise Bean Programmatically” on page 746
Securing the GlassFish Server

This tutorial describes deployment to the GlassFish Server, which provides highly secure, interoperable, and distributed component computing based on the Java EE security model. GlassFish Server supports the Java EE 6 security model. You can configure GlassFish Server for the following purposes:

- Adding, deleting, or modifying authorized users. For more information on this topic, see “Working with Realms, Users, Groups, and Roles” on page 702.
- Configuring secure HTTP and Internet Inter-Orb Protocol (IIOP) listeners.
- Configuring secure Java Management Extensions (JMX) connectors.
- Adding, deleting, or modifying existing or custom realms.
- Defining an interface for pluggable authorization providers using Java Authorization Contract for Containers (JACC). JACC defines security contracts between the GlassFish Server and authorization policy modules. These contracts specify how the authorization providers are installed, configured, and used in access decisions.
- Using pluggable audit modules.
- Customizing authentication mechanisms. All implementations of Java EE 6 compatible web containers are required to support the Servlet Profile of JSR 196, which offers an avenue for customizing the authentication mechanism applied by the web container on behalf of one or more applications.
- Setting and changing policy permissions for an application.

Working with Realms, Users, Groups, and Roles

You often need to protect resources to ensure that only authorized users have access. See “Characteristics of Application Security” on page 696 for an introduction to the concepts of authentication, identification, and authorization.

This section discusses setting up users so that they can be correctly identified and either given access to protected resources or denied access if they are not authorized to access the protected resources. To authenticate a user, you need to follow these basic steps.

1. The application developer writes code to prompt for a username and password. The various methods of authentication are discussed in “Specifying Authentication Mechanisms” on page 719.
2. The application developer communicates how to set up security for the deployed application by use of a metadata annotation or deployment descriptor. This step is discussed in “Setting Up Security Roles” on page 707.
3. The server administrator sets up authorized users and groups on the GlassFish Server. This is discussed in “Managing Users and Groups on the GlassFish Server” on page 706.
4. The application deployer maps the application’s security roles to users, groups, and principals defined on the GlassFish Server. This topic is discussed in “Mapping Roles to Users and Groups” on page 709.

What Are Realms, Users, Groups, and Roles?

A realm is a security policy domain defined for a web or application server. A realm contains a collection of users, who may or may not be assigned to a group. Managing users on the GlassFish Server is discussed in “Managing Users and Groups on the GlassFish Server” on page 706.

An application will often prompt for a user name and password before allowing access to a protected resource. After the user name and password have been entered, that information is passed to the server, which either authenticates the user and sends the protected resource or does not authenticate the user, in which case access to the protected resource is denied. This type of user authentication is discussed in “Specifying an Authentication Mechanism in the Deployment Descriptor” on page 722.

In some applications, authorized users are assigned to roles. In this situation, the role assigned to the user in the application must be mapped to a principal or group defined on the application server. Figure 39–6 shows this. More information on mapping roles to users and groups can be found in “Setting Up Security Roles” on page 707.

The following sections provide more information on realms, users, groups, and roles.
What Is a Realm?

The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database containing a collection of users and groups. A realm is a complete database of users and groups identified as valid users of one or more applications and controlled by the same authentication policy.

The Java EE server authentication service can govern users in multiple realms. The file, admin-realm, and certificate realms come preconfigured for the GlassFish Server.

In the file realm, the server stores user credentials locally in a file named keyfile. You can use the Administration Console to manage users in the file realm. When using the file realm, the server authentication service verifies user identity by checking the file realm. This realm is used for the authentication of all clients except for web browser clients that use HTTPS and certificates.

In the certificate realm, the server stores user credentials in a certificate database. When using the certificate realm, the server uses certificates with HTTPS to authenticate web clients. To verify the identity of a user in the certificate realm, the authentication service verifies an X.509 certificate. For step-by-step instructions for creating this type of certificate, see “Working with Digital Certificates” on page 759. The common name field of the X.509 certificate is used as the principal name.

![Diagram of mapping roles to users and groups](image-url)
The admin-realm is also a file realm and stores administrator user credentials locally in a file named admin-keyfile. You can use the Administration Console to manage users in this realm in the same way you manage users in the file realm. For more information, see "Managing Users and Groups on the GlassFish Server" on page 706.

**What Is a User?**

A *user* is an individual or application program identity that has been defined in the GlassFish Server. In a web application, a user can have associated with that identity a set of roles that entitle the user to access all resources protected by those roles. Users can be associated with a group.

A Java EE user is similar to an operating system user. Typically, both types of users represent people. However, these two types of users are not the same. The Java EE server authentication service has no knowledge of the username and password you provide when you log in to the operating system. The Java EE server authentication service is not connected to the security mechanism of the operating system. The two security services manage users that belong to different realms.

**What Is a Group?**

A *group* is a set of authenticated users, classified by common traits, defined in the GlassFish Server. A Java EE user of the file realm can belong to a group on the GlassFish Server. (A user in the certificate realm cannot.) A group on the GlassFish Server is a category of users classified by common traits, such as job title or customer profile. For example, most customers of an e-commerce application might belong to the CUSTOMER group, but the big spenders would belong to the PREFERRED group. Categorizing users into groups makes it easier to control the access of large numbers of users.

A group on the GlassFish Server has a different scope from a role. A group is designated for the entire GlassFish Server, whereas a role is associated only with a specific application in the GlassFish Server.

**What Is a Role?**

A *role* is an abstract name for the permission to access a particular set of resources in an application. A role can be compared to a key that can open a lock. Many people might have a copy of the key. The lock doesn’t care who you are, only that you have the right key.
Some Other Terminology

The following terminology is also used to describe the security requirements of the Java EE platform:

- **Principal**: An entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise. A principal is identified by using a principal name and authenticated by using authentication data.

- **Security policy domain**, also known as **security domain** or **realm**: A scope over which a common security policy is defined and enforced by the security administrator of the security service.

- **Security attributes**: A set of attributes associated with every principal. The security attributes have many uses: for example, access to protected resources and auditing of users. Security attributes can be associated with a principal by an authentication protocol.

- **Credential**: An object that contains or references security attributes used to authenticate a principal for Java EE services. A principal acquires a credential upon authentication or from another principal that allows its credential to be used.

Managing Users and Groups on the GlassFish Server

Follow these steps for managing users before you run the tutorial examples.

▼ **To Add Users to the GlassFish Server**

1. **Start the GlassFish Server, if you haven’t already done so.**
   Information on starting the GlassFish Server is available in "Starting and Stopping the GlassFish Server" on page 73.

2. **Start the Administration Console, if you haven’t already done so.**
   To start the Administration Console, open a web browser and specify the URL `http://localhost:4848/`. If you changed the default Admin port during installation, type the correct port number in place of 4848.

3. **In the navigation tree, expand the Configurations node, then expand the server-config node.**

4. **Expand the Security node.**

5. **Expand the Realms node.**
6 Select the realm to which you are adding users.

- Select the file realm to add users you want to access applications running in this realm.
  For the example security applications, select the file realm.
  The Edit Realm page opens.

- Select the admin-realm to add users you want to enable as system administrators of the GlassFish Server.
  The Edit Realm page opens.

You cannot add users to the certificate realm by using the Administration Console. In the certificate realm, you can add only certificates. For information on adding (importing) certificates to the certificate realm, see "Adding Users to the Certificate Realm" on page 762.

7 On the Edit Realm page, click the Manage Users button.
The File Users or Admin Users page opens.

8 On the File Users or Admin Users page, click New to add a new user to the realm.
The New File Realm User page opens.

9 Type values in the UserID, Group List, New Password, and Confirm New Password fields.
For the Admin Realm, the Group List field is read-only, and the group name is asadmin. Restart the GlassFish Server and Administration Console after you add a user to the Admin Realm.
For more information on these properties, see "Working with Realms, Users, Groups, and Roles" on page 702.

For the example security applications, specify a user with any name and password you like, but make sure that the user is assigned to the group TutorialUser. The user name and password are case-sensitive. Keep a record of the user name and password for working with the examples later in this tutorial.

10 Click OK to add this user to the realm, or click Cancel to quit without saving.

Setting Up Security Roles

When you design an enterprise bean or web component, you should always think about the kinds of users who will access the component. For example, a web application for a human resources department might have a different request URL for someone who has been assigned the role of DEPT_ADMIN than for someone who has been assigned the role of DIRECTOR. The DEPT_ADMIN role may let you view employee data, but the DIRECTOR role enables you to modify employee data, including salary data. Each of these security roles is an abstract logical grouping
of users that is defined by the person who assembles the application. When an application is deployed, the deployer will map the roles to security identities in the operational environment, as shown in Figure 39–6.

For Java EE components, you define security roles using the @DeclareRoles and @RolesAllowed metadata annotations.

The following is an example of an application in which the role of DEPT-ADMIN is authorized for methods that review employee payroll data, and the role of DIRECTOR is authorized for methods that change employee payroll data.

The enterprise bean would be annotated as shown in the following code:

```java
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
...
@DeclareRoles({"DEPT-ADMIN", "DIRECTOR"})
@Stateless public class PayrollBean implements Payroll {

@Resource SessionContext ctx;

@RolesAllowed("DEPT-ADMIN")
public void reviewEmployeeInfo(EmplInfo info) {
    oldInfo = ... read from database;
    // ...
}

@RolesAllowed("DIRECTOR")
public void updateEmployeeInfo(EmplInfo info) {
    newInfo = ... update database;
    // ...
}
...
}
```

For a servlet, you can use the @HttpConstraint annotation within the @ServletSecurity annotation to specify the roles that are allowed to access the servlet. For example, a servlet might be annotated as follows:

```java
@WebServlet(name = "PayrollServlet", urlPatterns = {"/payroll"})
@ServletSecurity(
    @HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
        rolesAllowed = {"DEPT-ADMIN", "DIRECTOR"}))
public class GreetingServlet extends HttpServlet {

These annotations are discussed in more detail in "Specifying Security for Basic Authentication Using Annotations" on page 731 and "Securing an Enterprise Bean Using Declarative Security" on page 742.

The Java EE 6 Tutorial • January 2013
After users have provided their login information and the application has declared what roles are authorized to access protected parts of an application, the next step is to map the security role to the name of a user, or principal.

**Mapping Roles to Users and Groups**

When you are developing a Java EE application, you don’t need to know what categories of users have been defined for the realm in which the application will be run. In the Java EE platform, the security architecture provides a mechanism for mapping the roles defined in the application to the users or groups defined in the runtime realm.

The role names used in the application are often the same as the group names defined on the GlassFish Server. Under these circumstances, you can enable a default principal-to-role mapping on the GlassFish Server by using the Administration Console. The task “To Set Up Your System for Running the Security Examples” on page 729 explains how to do this. All the tutorial security examples use default principal-to-role mapping.

If the role names used in an application are not the same as the group names defined on the server, use the runtime deployment descriptor to specify the mapping. The following example demonstrates how to do this mapping in the `glassfish-web.xml` file, which is the file used for web applications:

```xml
<glassfish-web-app>
  ...
  <security-role-mapping>
    <role-name>Mascot</role-name>
    <principal-name>Duke</principal-name>
  </security-role-mapping>

  <security-role-mapping>
    <role-name>Admin</role-name>
    <group-name>Director</group-name>
  </security-role-mapping>
  ...
</glassfish-web-app>
```

A role can be mapped to specific principals, specific groups, or both. The principal or group names must be valid principals or groups in the current default realm or in the realm specified in the `login-config` element. In this example, the role of Mascot used in the application is mapped to a principal, named Duke, that exists on the application server. Mapping a role to a specific principal is useful when the person occupying that role may change. For this application, you would need to modify only the runtime deployment descriptor rather than search and replace throughout the application for references to this principal.

Also in this example, the role of Admin is mapped to a group of users assigned the group name of Director. This is useful because the group of people authorized to access director-level administrative data has to be maintained only on the GlassFish Server. The application developer does not need to know who these people are, but only needs to define the group of people who will be given access to the information.
The role-name must match the role-name in the security-role element of the corresponding deployment descriptor or the role name defined in a @DeclareRoles annotation.

Establishing a Secure Connection Using SSL

Secure Sockets Layer (SSL) technology is security that is implemented at the transport layer (see "Transport-Layer Security" on page 699 for more information about transport-layer security). SSL allows web browsers and web servers to communicate over a secure connection. In this secure connection, the data is encrypted before being sent and then is decrypted upon receipt and before processing. Both the browser and the server encrypt all traffic before sending any data.

SSL addresses the following important security considerations:

- **Authentication**: During your initial attempt to communicate with a web server over a secure connection, that server will present your web browser with a set of credentials in the form of a server certificate (also called a public key certificate). The purpose of the certificate is to verify that the site is who and what it claims to be. In some cases, the server may request a certificate proving that the client is who and what it claims to be; this mechanism is known as client authentication.

- **Confidentiality**: When data is being passed between the client and the server on a network, third parties can view and intercept this data. SSL responses are encrypted so that the data cannot be deciphered by the third party and the data remains confidential.

- **Integrity**: When data is being passed between the client and the server on a network, third parties can view and intercept this data. SSL helps guarantee that the data will not be modified in transit by that third party.

The SSL protocol is designed to be as efficient as securely possible. However, encryption and decryption are computationally expensive processes from a performance standpoint. It is not strictly necessary to run an entire web application over SSL, and it is customary for a developer to decide which pages require a secure connection and which do not. Pages that might require a secure connection include those for login, personal information, shopping cart checkouts, or credit card information transmittal. Any page within an application can be requested over a secure socket by simply prefixing the address with https: instead of http:. Any pages that absolutely require a secure connection should check the protocol type associated with the page request and take the appropriate action if https: is not specified.

Using name-based virtual hosts on a secured connection can be problematic. This is a design limitation of the SSL protocol itself. The SSL handshake, whereby the client browser accepts the server certificate, must occur before the HTTP request is accessed. As a result, the request information containing the virtual host name cannot be determined before authentication, and it is therefore not possible to assign multiple certificates to a single IP address. If all virtual hosts on a single IP address need to authenticate against the same certificate, the addition of multiple virtual hosts should not interfere with normal SSL operations on the server. Be aware, however,
that most client browsers will compare the server’s domain name against the domain name listed in the certificate, if any; this is applicable primarily to official certificates signed by a certificate authority (CA). If the domain names do not match, these browsers will display a warning to the client. In general, only address-based virtual hosts are commonly used with SSL in a production environment.

**Verifying and Configuring SSL Support**

As a general rule, you must address the following issues to enable SSL for a server:

- There must be a Connector element for an SSL connector in the server deployment descriptor.
- There must be valid keystore and certificate files.
- The location of the keystore file and its password must be specified in the server deployment descriptor.

An SSL HTTPS connector is already enabled in the GlassFish Server.

For testing purposes and to verify that SSL support has been correctly installed, load the default introduction page with a URL that connects to the port defined in the server deployment descriptor:

https://localhost:8181/

The https in this URL indicates that the browser should be using the SSL protocol. The localhost in this example assumes that you are running the example on your local machine as part of the development process. The 8181 in this example is the secure port that was specified where the SSL connector was created. If you are using a different server or port, modify this value accordingly.

The first time that you load this application, the New Site Certificate or Security Alert dialog box appears. Select Next to move through the series of dialog boxes, and select Finish when you reach the last dialog box. The certificates will display only the first time. When you accept the certificates, subsequent hits to this site assume that you still trust the content.

**Further Information about Security**

For more information about security in Java EE applications, see:

- Java EE 6 specification:
- Enterprise JavaBeans 3.1 specification:
Further Information about Security

- Implementing Enterprise Web Services 1.3 specification:
- Java SE security information:
  http://docs.oracle.com/javase/6/docs/technotes/guides/security/
- Java Servlet 3.0 specification:
- Java Authorization Contract for Containers 1.4 specification:
The ways to implement security for Java EE web applications are discussed in a general way in “Securing Containers” on page 700. This chapter provides more detail and a few examples that explore these security services as they relate to web components.

A web application is accessed using a web browser over a network, such as the Internet or a company’s intranet. As discussed in “Distributed Multitiered Applications” on page 39, the Java EE platform uses a distributed multitiered application model, and web applications run in the web tier.

Web applications contain resources that can be accessed by many users. These resources often traverse unprotected, open networks, such as the Internet. In such an environment, a substantial number of web applications will require some type of security.

Securing applications and their clients in the business tier and the EIS tier is discussed in Chapter 41, “Getting Started Securing Enterprise Applications.”

The following topics are addressed here:

- “Overview of Web Application Security” on page 713
- “Securing Web Applications” on page 715
- “Using Programmatic Security with Web Applications” on page 724
- “Examples: Securing Web Applications” on page 729

**Overview of Web Application Security**

In the Java EE platform, web components provide the dynamic extension capabilities for a web server. Web components can be Java servlets or JavaServer Faces pages. The interaction between a web client and a web application is illustrated in Figure 40–1.
Certain aspects of web application security can be configured when the application is installed, or deployed, to the web container. Annotations and/or deployment descriptors are used to relay information to the deployer about security and other aspects of the application. Specifying this information in annotations or in the deployment descriptor helps the deployer set up the appropriate security policy for the web application. Any values explicitly specified in the deployment descriptor override any values specified in annotations.

Security for Java EE web applications can be implemented in the following ways.

- **Declarative security**: Can be implemented using either metadata annotations or an application's deployment descriptor. See “Overview of Java EE Security” on page 692 for more information.

  Declarative security for web applications is described in “Securing Web Applications” on page 715.

- **Programmatic security**: Is embedded in an application and can be used to make security decisions when declarative security alone is not sufficient to express the security model of an application. Declarative security alone may not be sufficient when conditional login in a particular work flow, instead of for all cases, is required in the middle of an application. See “Overview of Java EE Security” on page 692 for more information.

  Servlet 3.0 provides the authenticate, login, and logout methods of the HttpServletRequest interface. With the addition of the authenticate, login, and logout methods to the Servlet specification, an application deployment descriptor is no longer required for web applications but may still be used to further specify security requirements beyond the basic default values.
Programmatic security is discussed in “Using Programmatic Security with Web Applications” on page 724

- **Message Security**: Works with web services and incorporates security features, such as digital signatures and encryption, into the header of a SOAP message, working in the application layer, ensuring end-to-end security. Message security is not a component of Java EE 6 and is mentioned here for informational purposes only.

Some of the material in this chapter builds on material presented earlier in this tutorial. In particular, this chapter assumes that you are familiar with the information in the following chapters:

- Chapter 3, “Getting Started with Web Applications”
- Chapter 4, “JavaServer Faces Technology”
- Chapter 15, “Java Servlet Technology”
- Chapter 39, “Introduction to Security in the Java EE Platform”

### Securing Web Applications

Web applications are created by application developers who give, sell, or otherwise transfer the application to an application deployer for installation into a runtime environment. Application developers communicate how to set up security for the deployed application by using annotations or deployment descriptors. This information is passed on to the deployer, who uses it to define method permissions for security roles, set up user authentication, and set up the appropriate transport mechanism. If the application developer doesn’t define security requirements, the deployer will have to determine the security requirements independently.

Some elements necessary for security in a web application cannot be specified as annotations for all types of web applications. This chapter explains how to secure web applications using annotations wherever possible. It explains how to use deployment descriptors where annotations cannot be used.

### Specifying Security Constraints

A **security constraint** is used to define the access privileges to a collection of resources using their URL mapping.

If your web application uses a servlet, you can express the security constraint information by using annotations. Specifically, you use the `@HttpConstraint` and, optionally, the `@HttpMethodConstraint` annotations within the `@ServletSecurity` annotation to specify a security constraint.

If your web application does not use a servlet, however, you must specify a `security-constraint` element in the deployment descriptor file. The authentication mechanism cannot be expressed using annotations, so if you use any authentication method other than BASIC (the default), a deployment descriptor is required.
The following subelements can be part of a security-constraint:

- **Web resource collection** (web-resource-collection): A list of URL patterns (the part of a URL after the host name and port you want to constrain) and HTTP operations (the methods within the files that match the URL pattern you want to constrain) that describe a set of resources to be protected. Web resource collections are discussed in “Specifying a Web Resource Collection” on page 716.

- **Authorization constraint** (auth-constraint): Specifies whether authentication is to be used and names the roles authorized to perform the constrained requests. For more information about authorization constraints, see “Specifying an Authorization Constraint” on page 717.

- **User data constraint** (user-data-constraint): Specifies how data is protected when transported between a client and a server. User data constraints are discussed in “Specifying a Secure Connection” on page 717.

## Specifying a Web Resource Collection

A web resource collection consists of the following subelements:

- **web-resource-name** is the name you use for this resource. Its use is optional.

- **url-pattern** is used to list the request URI to be protected. Many applications have both unprotected and protected resources. To provide unrestricted access to a resource, do not configure a security constraint for that particular request URI.

The request URI is the part of a URL after the host name and port. For example, let’s say that you have an e-commerce site with a catalog that you would want anyone to be able to access and browse, and a shopping cart area for customers only. You could set up the paths for your web application so that the pattern `/cart/*` is protected but nothing else is protected. Assuming that the application is installed at context path `/myapp`, the following are true:


A user will be prompted to log in the first time he or she accesses a resource in the `cart` subdirectory.

- **http-method** or **http-method-omission** is used to specify which methods should be protected or which methods should be omitted from protection. An HTTP method is protected by a web-resource-collection under any of the following circumstances:
  - If no HTTP methods are named in the collection (which means that all are protected)
  - If the collection specifically names the HTTP method in an http-method subelement
  - If the collection contains one or more http-method-omission elements, none of which names the HTTP method
“Securing HTTP Resources” on page 775 describes how to create security constraints for HTTP methods that do not unintentionally leave your application open to cross-site tracing or HTTP verb tampering attacks.

Specifying an Authorization Constraint

An authorization constraint (auth-constraint) contains the role-name element. You can use as many role-name elements as needed here.

An authorization constraint establishes a requirement for authentication and names the roles authorized to access the URL patterns and HTTP methods declared by this security constraint. If there is no authorization constraint, the container must accept the request without requiring user authentication. If there is an authorization constraint but no roles are specified within it, the container will not allow access to constrained requests under any circumstances. Each role name specified here must either correspond to the role name of one of the security-role elements defined for this web application or be the specially reserved role name *, which indicates all roles in the web application. Role names are case sensitive. The roles defined for the application must be mapped to users and groups defined on the server, except when default principal-to-role mapping is used.

For more information about security roles, see “Declaring Security Roles” on page 723. For information on mapping security roles, see “Mapping Roles to Users and Groups” on page 709.

For a servlet, the @HttpConstraint and @HttpMethodConstraint annotations accept a rolesAllowed element that specifies the authorized roles.

Specifying a Secure Connection

A user data constraint (user-data-constraint in the deployment descriptor) contains the transport-guarantee subelement. A user data constraint can be used to require that a protected transport-layer connection, such as HTTPS, be used for all constrained URL patterns and HTTP methods specified in the security constraint. The choices for transport guarantee are CONFIDENTIAL, INTEGRAL, or NONE. If you specify CONFIDENTIAL or INTEGRAL as a security constraint, it generally means that the use of SSL is required and applies to all requests that match the URL patterns in the web resource collection, not just to the login dialog box.

The strength of the required protection is defined by the value of the transport guarantee.

- Specify CONFIDENTIAL when the application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission.
- Specify INTEGRAL when the application requires that the data be sent between client and server in such a way that it cannot be changed in transit.
- Specify NONE to indicate that the container must accept the constrained requests on any connection, including an unprotected one.
Note – In practice, Java EE servers treat the CONFIDENTIAL and INTEGRAL transport guarantee values identically.

The user data constraint is handy to use in conjunction with basic and form-based user authentication. When the login authentication method is set to BASIC or FORM, passwords are not protected, meaning that passwords sent between a client and a server on an unprotected session can be viewed and intercepted by third parties. Using a user data constraint with the user authentication mechanism can alleviate this concern. Configuring a user authentication mechanism is described in “Specifying an Authentication Mechanism in the Deployment Descriptor” on page 722.

To guarantee that data is transported over a secure connection, ensure that SSL support is configured for your server. SSL support is already configured for the GlassFish Server.

Note – After you switch to SSL for a session, you should never accept any non-SSL requests for the rest of that session. For example, a shopping site might not use SSL until the checkout page, and then it might switch to using SSL to accept your card number. After switching to SSL, you should stop listening to non-SSL requests for this session. The reason for this practice is that the session ID itself was not encrypted on the earlier communications. This is not so bad when you’re only doing your shopping, but after the credit card information is stored in the session, you don’t want anyone to use that information to fake the purchase transaction against your credit card. This practice could be easily implemented by using a filter.

Specifying Security Constraints for Resources

You can create security constraints for resources within your application. For example, you could allow users with the role of PARTNER full access to all resources at the URL pattern /company/wholesale/* and allow users with the role of CLIENT full access to all resources at the URL pattern /company/retail/*. This is the recommended way to protect resources if you do not want to protect some HTTP methods while leaving other HTTP methods unprotected. An example of a deployment descriptor that would demonstrate this functionality is the following:

```xml
<security-constraint>
  <web-resource-collection>
    <web-resource-name>wholesale</web-resource-name>
    <url-pattern>/company/wholesale/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>PARTNER</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```
Specifying Authentication Mechanisms

A user authentication mechanism specifies

■ The way a user gains access to web content
■ With basic authentication, the realm in which the user will be authenticated
■ With form-based authentication, additional attributes

When an authentication mechanism is specified, the user must be authenticated before access is granted to any resource that is constrained by a security constraint. There can be multiple security constraints applying to multiple resources, but the same authentication method will apply to all constrained resources in an application.

Before you can authenticate a user, you must have a database of user names, passwords, and roles configured on your web or application server. For information on setting up the user database, see “Managing Users and Groups on the GlassFish Server” on page 706.

The Java EE platform supports the following authentication mechanisms:

■ Basic authentication
■ Form-based authentication
■ Digest authentication
■ Client authentication
■ Mutual authentication

Basic, form-based, and digest authentication are discussed in this section. Client and mutual authentication are discussed in Chapter 42, "Java EE Security: Advanced Topics."

HTTP basic authentication and form-based authentication are not very secure authentication mechanisms. Basic authentication sends user names and passwords over the Internet as Base64-encoded text. Form-based authentication sends this data as plain text. In both cases, the target server is not authenticated. Therefore, these forms of authentication leave user data exposed and vulnerable. If someone can intercept the transmission, the user name and password information can easily be decoded.
However, when a secure transport mechanism, such as SSL, or security at the network level, such as the Internet Protocol Security (IPsec) protocol or virtual private network (VPN) strategies, is used in conjunction with basic or form-based authentication, some of these concerns can be alleviated. To specify a secure transport mechanism, use the elements described in “Specifying a Secure Connection” on page 717.

**HTTP Basic Authentication**

Specifying *HTTP basic authentication* requires that the server request a user name and password from the web client and verify that the user name and password are valid by comparing them against a database of authorized users in the specified or default realm.

Basic authentication is the default when you do not specify an authentication mechanism.

When basic authentication is used, the following actions occur:

1. A client requests access to a protected resource.
2. The web server returns a dialog box that requests the user name and password.
3. The client submits the user name and password to the server.
4. The server authenticates the user in the specified realm and, if successful, returns the requested resource.

*Figure 40–2* shows what happens when you specify HTTP basic authentication.

**Figure 40–2**  HTTP Basic Authentication

- **Client**
  - Requests a protected resource
  - Requests username:password
  - Sends username:password
  - Returns requested resource

- **Server**
  - Requests username:password
  - Sends username:password
  - Returns requested resource
Form-Based Authentication

Form-based authentication allows the developer to control the look and feel of the login authentication screens by customizing the login screen and error pages that an HTTP browser presents to the end user. When form-based authentication is declared, the following actions occur.

1. A client requests access to a protected resource.
2. If the client is unauthenticated, the server redirects the client to a login page.
3. The client submits the login form to the server.
4. The server attempts to authenticate the user.
   a. If authentication succeeds, the authenticated user’s principal is checked to ensure that it is in a role that is authorized to access the resource. If the user is authorized, the server redirects the client to the resource by using the stored URL path.
   b. If authentication fails, the client is forwarded or redirected to an error page.

Figure 40–3 shows what happens when you specify form-based authentication.

The section “The hello1_formauth Example: Form-Based Authentication with a JavaServer Faces Application” on page 734 is an example application that uses form-based authentication.
When you create a form-based login, be sure to maintain sessions using cookies or SSL session information.

For authentication to proceed appropriately, the action of the login form must always be \texttt{j\_security\_check}. This restriction is made so that the login form will work no matter which resource it is for and to avoid requiring the server to specify the action field of the outbound form. The following code snippet shows how the form should be coded into the HTML page:

```html
<form method="POST" action="j\_security\_check">
  <input type="text" name="j\_username">
  <input type="password" name="j\_password">
</form>
```

**Digest Authentication**

Like basic authentication, \textit{digest authentication} authenticates a user based on a user name and a password. However, unlike basic authentication, digest authentication does not send user passwords over the network. Instead, the client sends a one-way cryptographic hash of the password and additional data. Although passwords are not sent on the wire, digest authentication requires that clear-text password equivalents be available to the authenticating container so that it can validate received authenticators by calculating the expected digest.

**Specifying an Authentication Mechanism in the Deployment Descriptor**

To specify an authentication mechanism, use the \texttt{login-config} element. It can contain the following subelements.

- The \texttt{auth-method} subelement configures the authentication mechanism for the web application. The element content must be either \texttt{NONE}, \texttt{BASIC}, \texttt{DIGEST}, \texttt{FORM}, or \texttt{CLIENT-CERT}.
- The \texttt{realm-name} subelement indicates the realm name to use when the basic authentication scheme is chosen for the web application.
- The \texttt{form-login-config} subelement specifies the login and error pages that should be used when form-based login is specified.

\textbf{Note} – Another way to specify form-based authentication is to use the \texttt{authenticate}, \texttt{login}, and \texttt{logout} methods of \texttt{HttpServletRequest}, as discussed in "Authenticating Users Programmatically" on page 724.

When you try to access a web resource that is constrained by a \texttt{security-constraint} element, the web container activates the authentication mechanism that has been configured for that resource. The authentication mechanism specifies how the user will be prompted to log in. If
the login-config element is present and the auth-method element contains a value other than NONE, the user must be authenticated to access the resource. If you do not specify an authentication mechanism, authentication of the user is not required.

The following example shows how to declare form-based authentication in your deployment descriptor:

```xml
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>file</realm-name>
  <form-login-config>
    <form-login-page>/login.xhtml</form-login-page>
    <form-error-page>/error.xhtml</form-error-page>
  </form-login-config>
</login-config>
```

The login and error page locations are specified relative to the location of the deployment descriptor. Examples of login and error pages are shown in “Creating the Login Form and the Error Page” on page 734.

The following example shows how to declare digest authentication in your deployment descriptor:

```xml
<login-config>
  <auth-method>DIGEST</auth-method>
</login-config>
```

### Declaring Security Roles

You can declare security role names used in web applications by using the security-role element of the deployment descriptor. Use this element to list all the security roles that you have referenced in your application.

The following snippet of a deployment descriptor declares the roles that will be used in an application using the security-role element and specifies which of these roles is authorized to access protected resources using the auth-constraint element:

```xml
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Protected Area</web-resource-name>
    <url-pattern>/security/protected/*</url-pattern>
    <http-method>PUT</http-method>
    <http-method>DELETE</http-method>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>manager</role-name>
  </auth-constraint>
</security-constraint>
```
<!-- Security roles used by this web application -->
<security-role>
  <role-name>manager</role-name>
</security-role>
<security-role>
  <role-name>employee</role-name>
</security-role>

In this example, the security-role element lists all the security roles used in the application: manager and employee. This enables the deployer to map all the roles defined in the application to users and groups defined on the GlassFish Server.

The auth-constraint element specifies the role, manager, that can access the HTTP methods PUT, DELETE, GET, POST located in the directory specified by the url-pattern element (/jsp/security/protected/*).

The @ServletSecurity annotation cannot be used in this situation because its constraints apply to all URL patterns specified by the @WebServlet annotation.

Using Programmatic Security with Web Applications

Programmatic security is used by security-aware applications when declarative security alone is not sufficient to express the security model of the application.

Authenticating Users Programmatically

Servlet 3.0 specifies the following methods of the HttpServletRequest interface that enable you to authenticate users for a web application programmatically:

- authenticate, which allows an application to instigate authentication of the request caller by the container from within an unconstrained request context. A login dialog box displays and collects the user name and password for authentication purposes.
- login, which allows an application to collect user name and password information as an alternative to specifying form-based authentication in an application deployment descriptor.
- logout, which allows an application to reset the caller identity of a request.
The following example code shows how to use the `login` and `logout` methods:

```java
package test;
import java.io.IOException;
import java.io.PrintWriter;
import java.math.BigDecimal;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="TutorialServlet", urlPatterns={"/TutorialServlet"})
public class TutorialServlet extends HttpServlet {
    @EJB
    private ConverterBean converterBean;

    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>TutorialServlet</title>");
            out.println("</head>");
            out.println("<body>");
            request.login("TutorialUser", "TutorialUser");
            BigDecimal result = converterBean.dollarToYen(new BigDecimal("1.0"));
            out.println("<html>Servlet TutorialServlet result of dollarToYen = " + result + "</html>");
            out.println("<body>");
            out.println("</html>");
        } catch (Exception e) {
            throw new ServletException(e);
        } finally {
            request.logout();
            out.close();
        }
    }
}
```

The following example code shows how to use the `authenticate` method:
Checking Caller Identity Programmatically

In general, security management should be enforced by the container in a manner that is transparent to the web component. The security API described in this section should be used only in the less frequent situations in which the web component methods need to access the security context information.

Servlet 3.0 specifies the following methods that enable you to access security information about the component’s caller:

- **getRemoteUser**, which determines the user name with which the client authenticated. The `getRemoteUser` method returns the name of the remote user (the caller) associated by the container with the request. If no user has been authenticated, this method returns `null`.

- **isUserInRole**, which determines whether a remote user is in a specific security role. If no user has been authenticated, this method returns `false`. This method expects a `String` `role-name` parameter.

  The `security-role-ref` element should be declared in the deployment descriptor with a `role-name` subelement containing the role name to be passed to the method. Using security role references is discussed in “Declaring and Linking Role References” on page 728.

- **getUserPrincipal**, which determines the principal name of the current user and returns a `java.security.Principal` object. If no user has been authenticated, this method returns `null`. Calling the `getName` method on the `Principal` returned by `getUserPrincipal` returns the name of the remote user.

Your application can make business-logic decisions based on the information obtained using these APIs.
Example Code for Programmatic Security

The following code demonstrates the use of programmatic security for the purposes of programmatic login. This servlet does the following:

1. It displays information about the current user.
2. It prompts the user to log in.
3. It prints out the information again to demonstrate the effect of the login method.
4. It logs the user out.
5. It prints out the information again to demonstrate the effect of the logout method.

```java
package enterprise.programmatic_login;
import java.io.*;
import java.net.*;
import javax.annotation.security.DeclareRoles;
import javax.servlet.*;
import javax.servlet.http.*;

@DeclareRoles("javaee6user")
public class LoginServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP GET and POST methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            String userName = request.getParameter("txtUserName");
            String password = request.getParameter("txtPassword");
            out.println("Before Login<br><br>IsUserInRole?.." + request.isUserInRole("javaee6user") + <br>);
            out.println("getRemoteUser?.." + request.getRemoteUser() + <br>);
            out.println("getUserPrincipal?.." + request.getUserPrincipal() + <br>);
            out.println("getAuthType?.." + request.getAuthType() + <br>);
            try {
                request.login(userName, password);
            } catch(ServletException ex) {
                out.println("Login Failed with a ServletException.." + ex.getMessage());
                return;
            }
            out.println("After Login..." + <br>);
            out.println("IsUserInRole?.." + request.isUserInRole("javaee6user") + <br>);
        } finally {
            out.close();
        }
    }
}
```
out.println("getRemoteUser?.." + request.getRemoteUser() + "+<br>");
out.println("getUserPrincipal?.." + request.getUserPrincipal() + "+<br>");
out.println("getAuthType?.." + request.getAuthType() + "+<br>");
request.logout();
out.println("After Logout..." + "+<br>");
out.println("IsUserInRole?.." + request.isUserInRole("javaee6user") + "+<br>");
out.println("getRemoteUser?.." + request.getRemoteUser() + "+<br>");
out.println("getUserPrincipal?.." + request.getUserPrincipal() + "+<br>");
out.println("getAuthType?.." + request.getAuthType() + "+<br>");
}

finally {
  out.close();
}
...

**Declaring and Linking Role References**

A *security role reference* is a mapping between the name of a role that is called from a web component using `isUserInRole(String role)` and the name of a security role that has been defined for the application. If no `security-role-ref` element is declared in a deployment descriptor and the `isUserInRole` method is called, the container defaults to checking the provided role name against the list of all security roles defined for the web application. Using the default method instead of using the `security-role-ref` element limits your flexibility to change role names in an application without also recompiling the servlet making the call.

The `security-role-ref` element is used when an application uses `HttpServletRequest.isUserInRole(String role)`. The value passed to the `isUserInRole` method is a String representing the role name of the user. The value of the role-name element must be the String used as the parameter to the `HttpServletRequest.isUserInRole(String role)`. The role-link must contain the name of one of the security roles defined in the security-role elements. The container uses the mapping of `security-role-ref` to `security-role` when determining the return value of the call.

For example, to map the security role reference `cust` to the security role with role name `bankCustomer`, the syntax would be:

```xml
<servlet>
  ...
  <security-role-ref>
    <role-name>cust</role-name>
    <role-link>bankCustomer</role-link>
  </security-role-ref>
  ...
</servlet>
```

If the servlet method is called by a user in the `bankCustomer` security role, `isUserInRole("cust")` returns true.

The `role-link` element in the `security-role-ref` element must match a `role-name` defined in the `security-role` element of the same `web.xml` deployment descriptor, as shown here:

```
<security-role>
  <role-name>bankCustomer</role-name>
</security-role>
```

A security role reference, including the name defined by the reference, is scoped to the component whose deployment descriptor contains the `security-role-ref` deployment descriptor element.

### Examples: Securing Web Applications

Some basic setup is required before any of the example applications will run correctly. The examples use annotations, programmatic security, and/or declarative security to demonstrate adding security to existing web applications.

Here are some other locations where you will find examples of securing various types of applications:

- “The cart-secure Example: Securing an Enterprise Bean with Declarative Security” on page 750
- “The converter-secure Example: Securing an Enterprise Bean with Programmatic Security” on page 754
- GlassFish samples: [http://glassfish-samples.java.net/](http://glassfish-samples.java.net/)

### To Set Up Your System for Running the Security Examples

To set up your system for running the security examples, you need to configure a user database that the application can use for authenticating users. Before continuing, follow these steps.

1. Add an authorized user to the GlassFish Server. For the examples in this chapter and in Chapter 41, "Getting Started Securing Enterprise Applications," add a user to the file realm of the GlassFish Server, and assign the user to the group TutorialUser:
   
   a. From the Administration Console, expand the Configurations node, then expand the server-config node.
   
   b. Expand the Security node.
   
   c. Expand the Realms node.
   
   d. Select the File node.
e. On the Edit Realm page, click Manage Users.


g. In the User ID field, type a User ID.

h. In the Group List field, type TutorialUser.

i. In the New Password and Confirm New Password fields, type a password.

j. Click OK.

Be sure to write down the user name and password for the user you create so that you can use it for testing the example applications. Authentication is case sensitive for both the user name and password, so write down the user name and password exactly. This topic is discussed more in “Managing Users and Groups on the GlassFish Server” on page 706.

2 Set up Default Principal to Role Mapping on the GlassFish Server:

a. From the Administration Console, expand the Configurations node, then expand the server-config node.

b. Select the Security node.

c. Select the Default Principal to Role Mapping Enabled check box.

d. Click Save.

**The hello2_basicauth Example: Basic Authentication with a Servlet**

This example explains how to use basic authentication with a servlet. With basic authentication of a servlet, the web browser presents a standard login dialog that is not customizable. When a user submits his or her name and password, the server determines whether the user name and password are those of an authorized user and sends the requested web resource if the user is authorized to view it.
In general, the following steps are necessary for adding basic authentication to an unsecured servlet, such as the ones described in Chapter 3, “Getting Started with Web Applications.” In the example application included with this tutorial, many of these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application. The completed version of this example application can be found in the "tut-install/examples/security/hello2_basicauth/" directory.

1. Follow the steps in “To Set Up Your System for Running the Security Examples” on page 729.
2. Create a web module as described in Chapter 3, “Getting Started with Web Applications,” for the servlet example, hello2.
3. Add the appropriate security annotations to the servlet. The security annotations are described in “Specifying Security for Basic Authentication Using Annotations” on page 731.
4. Build, package, and deploy the web application by following the steps in “To Build, Package, and Deploy the hello2_basicauth Example Using NetBeans IDE” on page 732 or “To Build, Package, and Deploy the hello2_basicauth Example Using Ant” on page 732.
5. Run the web application by following the steps described in “To Run the hello2_basicauth Example” on page 733.

Specifying Security for Basic Authentication Using Annotations

The default authentication mechanism used by the GlassFish Server is basic authentication. With basic authentication, the GlassFish Server spawns a standard login dialog to collect user name and password data for a protected resource. Once the user is authenticated, access to the protected resource is permitted.

To specify security for a servlet, use the @ServletSecurity annotation. This annotation allows you to specify both specific constraints on HTTP methods and more general constraints that apply to all HTTP methods for which no specific constraint is specified. Within the @ServletSecurity annotation, you can specify the following annotations:

- The @HttpMethodConstraint annotation, which applies to a specific HTTP method
- The more general @HttpConstraint annotation, which applies to all HTTP methods for which there is no corresponding @HttpMethodConstraint annotation

Both the @HttpMethodConstraint and @HttpConstraint annotations within the @ServletSecurity annotation can specify the following:

- A transportGuarantee element that specifies the data protection requirements (that is, whether or not SSL/TLS is required) that must be satisfied by the connections on which requests arrive. Valid values for this element are NONE and CONFIDENTIAL.
- A rolesAllowed element that specifies the names of the authorized roles.

For the hello2_basicauth application, the GreetingServlet has the following annotations:
These annotations specify that the request URI /greeting can be accessed only by users who have been authorized to access this URL because they have been verified to be in the role TutorialUser. The data will be sent over a protected transport in order to keep the user name and password data from being read in transit.

If you use the @ServletSecurity annotation, you do not need to specify security settings in the deployment descriptor. Use the deployment descriptor to specify settings for nondefault authentication mechanisms, for which you cannot use the @ServletSecurity annotation.

▼ To Build, Package, and Deploy the hello2_basicauth Example Using NetBeans IDE

1 Follow the steps in “To Set Up Your System for Running the Security Examples” on page 729.

2 In NetBeans IDE, from the File menu, choose Open Project.

3 In the Open Project dialog, navigate to:
   tut-install/examples/security/

4 Select the hello2_basicauth folder.

5 Select the Open as Main Project check box.

6 Click Open Project.

7 Right-click hello2_basicauth in the Projects pane and select Deploy.
   This option builds and deploys the example application to your GlassFish Server instance.

▼ To Build, Package, and Deploy the hello2_basicauth Example Using Ant

1 Follow the steps in “To Set Up Your System for Running the Security Examples” on page 729.

2 In a terminal window, go to:
   tut-install/examples/security/hello2_basicauth/

3 Type the following command:
   ant
This command calls the default target, which builds and packages the application into a WAR file, hello2_basicauth.war, that is located in the dist directory.

4 Make sure that the GlassFish Server is started.

5 To deploy the application, type the following command:
   `ant deploy`

**To Run the hello2_basicauth Example**

1 In a web browser, navigate to the following URL:
   `https://localhost:8181/hello2_basicauth/greeting`
   You may be prompted to accept the security certificate for the server. If so, accept the security certificate. If the browser warns that the certificate is invalid because it is self-signed, add a security exception for the application.
   An Authentication Required dialog box appears. Its appearance varies, depending on the browser you use.

2 Type a user name and password combination that corresponds to a user who has already been created in the file realm of the GlassFish Server and has been assigned to the group of TutorialUser; then click OK.
   Basic authentication is case sensitive for both the user name and password, so type the user name and password exactly as defined for the GlassFish Server.
   The server returns the requested resource if all the following conditions are met.
   - A user with the user name you entered is defined for the GlassFish Server.
   - The user with the user name you entered has the password you entered.
   - The user name and password combination you entered is assigned to the group TutorialUser on the GlassFish Server.
   - The role of TutorialUser, as defined for the application, is mapped to the group TutorialUser, as defined for the GlassFish Server.

3 Type a name in the text field and click the Submit button.
   Because you have already been authorized, the name you enter in this step does not have any limitations. You have unlimited access to the application now.
   The application responds by saying “Hello” to the name you typed.

Next Steps For repetitive testing of this example, you may need to close and reopen your browser. You should also run the ant undeploy and ant clean targets or the NetBeans IDE Clean and Build option to get a fresh start.
The hello1_formauth Example: Form-Based Authentication with a JavaServer Faces Application

This example explains how to use form-based authentication with a JavaServer Faces application. With form-based authentication, you can customize the login screen and error pages that are presented to the web client for authentication of the user name and password. When a user submits his or her name and password, the server determines whether the user name and password are those of an authorized user and, if authorized, sends the requested web resource.

This example, hello1_formauth, adds security to the basic JavaServer Faces application shown in “Web Modules: The hello1 Example” on page 84.

In general, the steps necessary for adding form-based authentication to an unsecured JavaServer Faces application are similar to those described in “The hello2_basicauth Example: Basic Authentication with a Servlet” on page 730. The major difference is that you must use a deployment descriptor to specify the use of form-based authentication, as described in “Specifying Security for the Form-Based Authentication Example” on page 736. In addition, you must create a login form page and a login error page, as described in “Creating the Login Form and the Error Page” on page 734.

The completed version of this example application can be found in the tut-install/examples/security/hello1_formauth/ directory.

Creating the Login Form and the Error Page

When using form-based login mechanisms, you must specify a page that contains the form you want to use to obtain the user name and password, as well as a page to display if login authentication fails. This section discusses the login form and the error page used in this example. “Specifying Security for the Form-Based Authentication Example” on page 736 shows how you specify these pages in the deployment descriptor.

The login page can be an HTML page or a servlet, and it must return an HTML page containing a form that conforms to specific naming conventions (see the Java Servlet 3.0 specification for more information on these requirements). To do this, include the elements that accept user name and password information between <form></form> tags in your login page. The content of an HTML page or servlet for a login page should be coded as follows:

```
<form method="post" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
</form>
```

You can use Facelets tags instead of the HTML input tags, but you must use the HTML form tag, not a Facelets tag.
The full code for the login page used in this example can be found at `tut-install/examples/security/hello1_formauth/web/login.xhtml`. Here is the code for this page:

```html
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Login Form</title>
  </h:head>
  <h:body>
    <h2>Hello, please log in:</h2>
    <form method="post" action="/j_security_check">
      <table columns="2" role="presentation">
        <tr>
          <td><h:outputLabel for="j_username" value="Please type your user name:="/></td>
          <td><h:inputText id="j_username" autocomplete="off" size="20" /></td>
        </tr>
        <tr>
          <td><h:outputLabel for="j_password" value="Please type your password:="/></td>
          <td><h:inputSecret id="j_password" autocomplete="off" size="20" /></td>
        </tr>
      </table>
      <p>
        <h:commandButton type="submit" value="Submit" />
        &nbsp;
        <h:commandButton type="reset" value="Reset" />
      </p>
    </form>
  </h:body>
</html>

The login error page is displayed if the user enters a user name and password combination that is not authorized to access the protected URI. For this example, the login error page can be found at `tut-install/examples/security/hello1_formauth/web/error.xhtml`. For this example, the login error page explains the reason for receiving the error page and provides a link that will allow the user to try again. Here is the code for this page:

```html
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Login Error</title>
  </head>
  <body>
    <h2>Invalid user name or password.</h2>
    <p>Please enter a user name or password that is authorized to access this application. For this application, this means a user that has been created in the <code>file</code> realm and has been assigned to the <code>group</code> of <code>TutorialUser</code>.</p>
    <p><a href="/faces/index.xhtml">Return to login page</a></p>
</body>
</html>
```
Specifying Security for the Form-Based Authentication Example

This example takes a very simple servlet-based web application and adds form-based security. To specify form-based instead of basic authentication for a JavaServer Faces example, you must use the deployment descriptor.

The following sample code shows the security elements added to the deployment descriptor for this example, which can be found in `tut-install/examples/security/hello1_formauth/web/WEB-INF/web.xml`.

```xml
<security-constraint>
  <display-name>Constraint1</display-name>
  <web-resource-collection>
    <web-resource-name>wrcoll</web-resource-name>
    <description/>
    <url-pattern>/</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description/>
    <role-name>TutorialUser</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>file</realm-name>
  <form-login-config>
    <form-login-page>/faces/login.xhtml</form-login-page>
    <form-error-page>/faces/error.xhtml</form-error-page>
  </form-login-config>
</login-config>

<security-role>
  <description/>
  <role-name>TutorialUser</role-name>
</security-role>
```

To Build, Package, and Deploy the hello1_formauth Example Using NetBeans IDE

1. Follow the steps in “To Set Up Your System for Running the Security Examples” on page 729.

2. In NetBeans IDE, from the File menu, choose Open Project.

3. In the Open Project dialog, navigate to:
   `tut-install/examples/security/`

4. Select the hello1_formauth folder.
5 Select the Open as Main Project check box.

6 Click Open Project.

7 Right-click hello1_formauth in the Projects pane and select Deploy.

▼ To Build, Package, and Deploy the hello1_formauth Example Using Ant

1 Follow the steps in “To Set Up Your System for Running the Security Examples” on page 729.

2 In a terminal window, go to:
   
tut-install/examples/security/hello1_formauth/

3 Type the following command at the terminal window or command prompt:
   
ant

   This target will spawn any necessary compilations, copy files to the
tut-install/examples/security/hello1_formauth/build/ directory, create the WAR file, and
   copy it to the tut-install/examples/security/hello1_formauth/dist/ directory.

4 To deploy hello1_formauth.war to the GlassFish Server, type the following command:
   
ant deploy

▼ To Run the hello1_formauth Example

To run the web client for hello1_formauth, follow these steps.

1 Open a web browser to the following URL:
   
https://localhost:8181/hello1_formauth/
   
The login form opens in the browser.

2 Type a user name and password combination that corresponds to a user who has already been
   created in the file realm of the GlassFish Server and has been assigned to the group of TutorialUser.
   
   Form-based authentication is case sensitive for both the user name and password, so type the
   user name and password exactly as defined for the GlassFish Server.

3 Click the Submit button.

   If you entered My_Name as the name and My_Pwd for the password, the server returns the
   requested resource if all the following conditions are met:
   
   ▪ A user with the user name My_Name is defined for the GlassFish Server.
   ▪ The user with the user name My_Name has a password My_Pwd defined for the GlassFish Server.
The user My_Name with the password My_Pwd is assigned to the group TutorialUser on the GlassFish Server.

The role TutorialUser, as defined for the application, is mapped to the group TutorialUser, as defined for the GlassFish Server. When these conditions are met and the server has authenticated the user, the application appears.

4 **Type your name and click the Submit button.**

Because you have already been authorized, the name you enter in this step does not have any limitations. You have unlimited access to the application now.

The application responds by saying “Hello” to you.

**Next Steps** For additional testing and to see the login error page generated, close and reopen your browser, type the application URL, and type a user name and password that are not authorized.

**Note** – For repetitive testing of this example, you may need to close and reopen your browser. You should also run the ant clean and ant undeploy commands to ensure a fresh build if using the Ant tool, or select Clean and Build then Deploy if using NetBeans IDE.
Getting Started Securing Enterprise Applications

System administrators, application developers, bean providers, and deployers are responsible for administering security for enterprise applications:

- A system administrator is responsible for setting up a database of users and assigning them to the proper group. The system administrator is also responsible for setting GlassFish Server properties that enable the applications to run properly. Some security-related examples set up a default principal-to-role mapping, anonymous users, default users, and propagated identities. When needed for this tutorial, the steps for performing specific tasks are provided.

- An application developer or bean provider is responsible for annotating the classes and methods of the enterprise application in order to provide information to the deployer about which methods need to have restricted access. This tutorial describes the steps necessary to complete this task.

- A deployer is responsible for taking the security view provided by the application developer and implementing that security upon deployment. This document provides the information needed to accomplish this task for the tutorial example applications.

The following topics are addressed here:

- "Securing Enterprise Beans" on page 739
- "Examples: Securing Enterprise Beans" on page 749

Securing Enterprise Beans

Enterprise beans are Java EE components that implement EJB technology. Enterprise beans run in the EJB container, a runtime environment within the GlassFish Server. Although transparent to the application developer, the EJB container provides system-level services, such as transactions and security to its enterprise beans, which form the core of transactional Java EE applications.
Enterprise bean methods can be secured in either of the following ways:

- **Declarative security** (preferred): Expresses an application component's security requirements using either deployment descriptors or annotations. The presence of an annotation in the business method of an enterprise bean class that specifies method permissions is all that is needed for method protection and authentication in some situations. This section discusses this simple and efficient method of securing enterprise beans.

Because of some limitations to the simplified method of securing enterprise beans, you would want to continue to use the deployment descriptor to specify security information in some instances. An authentication mechanism must be configured on the server for the simple solution to work. Basic authentication is the GlassFish Server’s default authentication method.

This tutorial explains how to invoke user name/password authentication of authorized users by decorating the enterprise application’s business methods with annotations that specify method permissions.

To make the deployer’s task easier, the application developer can define security roles. A security role is a grouping of permissions that a given type of application users must have in order to successfully use the application. For example, in a payroll application, some users will want to view their own payroll information (*employee*), some will need to view others’ payroll information (*manager*), and some will need to be able to change others’ payroll information (*payrollDept*). The application developer would determine the potential users of the application and which methods would be accessible to which users. The application developer would then decorate classes or methods of the enterprise bean with annotations that specify the types of users authorized to access those methods. Using annotations to specify authorized users is described in “Specifying Authorized Users by Declaring Security Roles” on page 743.

When one of the annotations is used to define method permissions, the deployment system will automatically require user name/password authentication. In this type of authentication, a user is prompted to enter a user name and password, which will be compared against a database of known users. If the user is found and the password matches, the roles that the user is assigned will be compared against the roles that are authorized to access the method. If the user is authenticated and found to have a role that is authorized to access that method, the data will be returned to the user.

Using declarative security is discussed in “Securing an Enterprise Bean Using Declarative Security” on page 742.

- **Programmatic security**: For an enterprise bean, code embedded in a business method that is used to access a caller's identity programmatically and that uses this information to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application.

In general, security management should be enforced by the container in a manner that is transparent to the enterprise beans’ business methods. The programmatic security APIs described in this chapter should be used only in the less frequent situations in which the
enterprise bean business methods need to access the security-context information, such as when you want to grant access based on the time of day or other nontrivial condition checks for a particular role.

Programmatic security is discussed in “Securing an Enterprise Bean Programmatically” on page 746.


As mentioned earlier, enterprise beans run in the EJB container, a runtime environment within the GlassFish Server, as shown in Figure 41–1.
This section discusses securing a Java EE application where one or more modules, such as EJB JAR files, are packaged into an EAR file, the archive file that holds the application. Security annotations will be used in the Java programming class files to specify authorized users and basic, or user name/password, authentication.

Enterprise beans often provide the business logic of a web application. In these cases, packaging the enterprise bean within the web application’s WAR module simplifies deployment and application organization. Enterprise beans may be packaged within a WAR module as Java class files or within a JAR file that is bundled within the WAR module. When a servlet or JavaServer Faces page handles the web front end and the application is packaged into a WAR module as a Java class file, security for the application can be handled in the application’s web.xml file. The EJB in the WAR file can have its own deployment descriptor, ejb-jar.xml, if required. Securing web applications using web.xml is discussed in Chapter 40, “Getting Started Securing Web Applications.”

The following sections describe declarative and programmatic security mechanisms that can be used to protect enterprise bean resources. The protected resources include enterprise bean methods that are called from application clients, web components, or other enterprise beans.


**Securing an Enterprise Bean Using Declarative Security**

Declarative security enables the application developer to specify which users are authorized to access which methods of the enterprise beans and to authenticate these users with basic, or username-password, authentication. Frequently, the person who is developing an enterprise application is not the same person who is responsible for deploying the application. An application developer who uses declarative security to define method permissions and authentications mechanisms is passing along to the deployer a security view of the enterprise beans contained in the EJB JAR. When a security view is passed on to the deployer, he or she uses this information to define method permissions for security roles. If you don’t define a security view, the deployer will have to determine what each business method does to determine which users are authorized to call each method.

A security view consists of a set of security roles, a semantic grouping of permissions that a given type of users of an application must have to successfully access the application. Security roles are meant to be logical roles, representing a type of user. You can define method permissions for each security role. A method permission is a permission to invoke a specified group of methods of an enterprise bean’s business interface, home interface, component interface, and/or web service endpoints. After method permissions are defined, user name/password authentication will be used to verify the identity of the user.
It is important to keep in mind that security roles are used to define the logical security view of an application. They should not be confused with the user groups, users, principals, and other concepts that exist in the GlassFish Server. An additional step is required to map the roles defined in the application to users, groups, and principals that are the components of the user database in the file realm of the GlassFish Server. These steps are outlined in “Mapping Roles to Users and Groups” on page 709.

The following sections show how an application developer uses declarative security to either secure an application or to create a security view to pass along to the deployer.

**Specifying Authorized Users by Declaring Security Roles**

This section discusses how to use annotations to specify the method permissions for the methods of a bean class. For more information on these annotations, refer to the Common Annotations for the Java Platform specification at [http://jcp.org/en/jsr/detail?id=250](http://jcp.org/en/jsr/detail?id=250).

Method permissions can be specified on the class, the business methods of the class, or both. Method permissions can be specified on a method of the bean class to override the method permissions value specified on the entire bean class. The following annotations are used to specify method permissions:

- **@DeclareRoles**: Specifies all the roles that the application will use, including roles not specifically named in a @RolesAllowed annotation. The set of security roles the application uses is the total of the security roles defined in the @DeclareRoles and @RolesAllowed annotations.

  The @DeclareRoles annotation is specified on a bean class, where it serves to declare roles that can be tested (for example, by calling isCallerInRole) from within the methods of the annotated class. When declaring the name of a role used as a parameter to the isCallerInRole(String roleName) method, the declared name must be the same as the parameter value.

  The following example code demonstrates the use of the @DeclareRoles annotation:

  ```java
  @DeclareRoles("BusinessAdmin")
  public class Calculator {
    ...
  }
  ```

  The syntax for declaring more than one role is as shown in the following example:

  ```java
  @DeclareRoles({"Administrator", "Manager", "Employee"})
  ```

- **@RolesAllowed(list-of-roles)**: Specifies the security roles permitted to access methods in an application. This annotation can be specified on a class or on one or more methods. When specified at the class level, the annotation applies to all methods in the class. When specified on a method, the annotation applies to that method only and overrides any values specified at the class level.
To specify that no roles are authorized to access methods in an application, use the @DenyAll annotation. To specify that a user in any role is authorized to access the application, use the @PermitAll annotation.

When used in conjunction with the @DeclareRoles annotation, the combined set of security roles is used by the application.

The following example code demonstrates the use of the @RolesAllowed annotation:

```java
@DeclareRoles({"Administrator", "Manager", "Employee"})
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        //...
    }
}
```

- **@PermitAll**: Specifies that all security roles are permitted to execute the specified method or methods. The user is not checked against a database to ensure that he or she is authorized to access this application.

  This annotation can be specified on a class or on one or more methods. Specifying this annotation on the class means that it applies to all methods of the class. Specifying it at the method level means that it applies to only that method.

  The following example code demonstrates the use of the @PermitAll annotation:

  ```java
  import javax.annotation.security.*;
  @RolesAllowed("RestrictedUsers")
  public class Calculator {
      @RolesAllowed("Administrator")
      public void setNewRate(int rate) {
          //...
      }
      @PermitAll
      public long convertCurrency(long amount) {
          //...
      }
  }
  ```

- **@DenyAll**: Specifies that no security roles are permitted to execute the specified method or methods. This means that these methods are excluded from execution in the Java EE container.

  The following example code demonstrates the use of the @DenyAll annotation:

  ```java
  import javax.annotation.security.*;
  @RolesAllowed("Users")
  public class Calculator {
      @RolesAllowed("Administrator")
      public void setNewRate(int rate) {
          //...
      }
      @DenyAll
      public long convertCurrency(long amount) {
          //...
      }
  }
  ```
The following code snippet demonstrates the use of the @DeclareRoles annotation with the isCallerInRole method. In this example, the @DeclareRoles annotation declares a role that the enterprise bean PayrollBean uses to make the security check by using isCallerInRole("payroll") to verify that the caller is authorized to change salary data:

```java
@DeclareRoles("payroll")
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;
    public void updateEmployeeInfo(EmplInfo info) {
        oldInfo = ... read from database;
        // The salary field can be changed only by callers
        // who have the security role "payroll"
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (info.salary != oldInfo.salary && !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
        ...
    }
}
```

The following example code illustrates the use of the @RolesAllowed annotation:

```java
@RolesAllowed("admin")
public class SomeClass {
    public void aMethod () {...}
    public void bMethod () {...}
    ...
}
```

```java
@Stateless public class MyBean extends SomeClass implements A {
    @RolesAllowed("HR")
    public void aMethod () {...}
    
    public void cMethod () {...}
    ...
}
```

In this example, assuming that aMethod, bMethod, and cMethod are methods of business interface A, the method permissions values of methods aMethod and bMethod are @RolesAllowed("HR") and @RolesAllowed("admin"), respectively. The method permissions for method cMethod have not been specified.

To clarify, the annotations are not inherited by the subclass itself. Instead, the annotations apply to methods of the superclass that are inherited by the subclass.
Specifying an Authentication Mechanism and Secure Connection

When method permissions are specified, basic user name/password authentication will be invoked by the GlassFish Server.

To use a different type of authentication or to require a secure connection using SSL, specify this information in an application deployment descriptor.

Securing an Enterprise Bean Programmatically

Programmatic security, code that is embedded in a business method, is used to access a caller's identity programmatically and uses this information to make security decisions within the method itself.

Accessing an Enterprise Bean Caller’s Security Context

In general, security management should be enforced by the container in a manner that is transparent to the enterprise bean's business methods. The security API described in this section should be used only in the less frequent situations in which the enterprise bean business methods need to access the security context information, such as when you want to restrict access to a particular time of day.

The javax.ejb.EJBContext interface provides two methods that allow the bean provider to access security information about the enterprise bean’s caller:

- `getCallerPrincipal`, which allows the enterprise bean methods to obtain the current caller principal's name. The methods might, for example, use the name as a key to information in a database.

The following code sample illustrates the use of the `getCallerPrincipal` method:

```java
@Stateless public class EmployeeServiceBean implements EmployeeService {
    @Resource SessionContext ctx;
    @PersistenceContext EntityManager em;

    public void changePhoneNumber(...) {
        ...
        // obtain the caller principal.
        callerPrincipal = ctx.getCallerPrincipal();

        // obtain the caller principal's name.
        callerKey = callerPrincipal.getName();

        // use callerKey as primary key to find EmployeeRecord
        EmployeeRecord myEmployeeRecord =
            em.find(EmployeeRecord.class, callerKey);

        // update phone number
        myEmployeeRecord.setPhoneNumber(...);
        ...
    }
}
```
In this example, the enterprise bean obtains the principal name of the current caller and uses it as the primary key to locate an EmployeeRecord entity. This example assumes that the application has been deployed such that the current caller principal contains the primary key used for the identification of employees (for example, employee number).

- `isCallerInRole`, which the enterprise bean code can use to allow the bean provider/application developer to code the security checks that cannot be easily defined using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.

The enterprise bean code can use the `isCallerInRole` method to test whether the current caller has been assigned to a given security role. Security roles are defined by the bean provider or the application assembler and are assigned by the deployer to principals or principal groups that exist in the operational environment.

The following code sample illustrates the use of the `isCallerInRole` method:

```java
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;
    public void updateEmployeeInfo(EmplInfo info) {
        oldInfo = ... read from database;

        // The salary field can be changed only by callers
        // who have the security role "payroll"
        if (info.salary != oldInfo.salary &&
            !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
    }
}
```

You would use programmatic security in this way to dynamically control access to a method, for example, when you want to deny access except during a particular time of day. An example application that uses the `getCallerPrincipal` and `isCallerInRole` methods is described in “The converter-secure Example: Securing an Enterprise Bean with Programmatic Security” on page 754.

### Propagating a Security Identity (Run-As)

You can specify whether a caller’s security identity should be used for the execution of specified methods of an enterprise bean or whether a specific run-as identity should be used. Figure 41–2 illustrates this concept.
In this illustration, an application client is making a call to an enterprise bean method in one EJB container. This enterprise bean method, in turn, makes a call to an enterprise bean method in another container. The security identity during the first call is the identity of the caller. The security identity during the second call can be any of the following options.

- By default, the identity of the caller of the intermediate component is propagated to the target enterprise bean. This technique is used when the target container trusts the intermediate container.

- A specific identity is propagated to the target enterprise bean. This technique is used when the target container expects access using a specific identity.

To propagate an identity to the target enterprise bean, configure a run-as identity for the bean, as described in “Configuring a Component’s Propagated Security Identity” on page 748. Establishing a run-as identity for an enterprise bean does not affect the identities of its callers, which are the identities tested for permission to access the methods of the enterprise bean. The run-as identity establishes the identity that the enterprise bean will use when it makes calls.

The run-as identity applies to the enterprise bean as a whole, including all the methods of the enterprise bean’s business interface, local and remote interfaces, component interface, and web service endpoint interfaces, the message listener methods of a message-driven bean, the timeout method of an enterprise bean, and all internal methods of the bean that might be called in turn.

**Configuring a Component’s Propagated Security Identity**

You can configure an enterprise bean’s run-as, or propagated, security identity by using the @RunAs annotation, which defines the role of the application during execution in a Java EE container. The annotation can be specified on a class, allowing developers to execute an application under a particular role. The role must map to the user/group information in the container’s security realm. The @RunAs annotation specifies the name of a security role as its parameter.

Here is some example code that demonstrates the use of the @RunAs annotation.

```java
@RunAs("Admin")
public class Calculator {
```
You will have to map the run-as role name to a given principal defined on the GlassFish Server if the given roles are associated with more than one user principal.

**Trust between Containers**

When an enterprise bean is designed so that either the original caller identity or a designated identity is used to call a target bean, the target bean will receive the propagated identity only. The target bean will not receive any authentication data.

There is no way for the target container to authenticate the propagated security identity. However, because the security identity is used in authorization checks (for example, method permissions or with the `isCallerInRole` method), it is vitally important that the security identity be authentic. Because no authentication data is available to authenticate the propagated identity, the target must trust that the calling container has propagated an authenticated security identity.

By default, the GlassFish Server is configured to trust identities that are propagated from different containers. Therefore, you do not need to take any special steps to set up a trust relationship.

**Deploying Secure Enterprise Beans**

The deployer is responsible for ensuring that an assembled application is secure after it has been deployed in the target operational environment. If a security view has been provided to the deployer through the use of security annotations and/or a deployment descriptor, the security view is mapped to the mechanisms and policies used by the security domain in the target operational environment, which in this case is the GlassFish Server. If no security view is provided, the deployer must set up the appropriate security policy for the enterprise bean application.

Deployment information is specific to a web or application server.

**Examples: Securing Enterprise Beans**

The following examples show how to secure enterprise beans using declarative and programmatic security.
The cart-secure Example: Securing an Enterprise Bean with Declarative Security

This section discusses how to configure an enterprise bean for basic user name/password authentication. When a bean that is constrained in this way is requested, the server requests a user name and password from the client and verifies that the user name and password are valid by comparing them against a database of authorized users on the GlassFish Server.

If the topic of authentication is new to you, see "Specifying Authentication Mechanisms" on page 719.

This example demonstrates security by starting with the unsecured enterprise bean application, cart, which is found in the directory tut-install/examples/ejb/cart/ and is discussed in “The cart Example” on page 461.

In general, the following steps are necessary to add user name/password authentication to an existing application that contains an enterprise bean. In the example application included with this tutorial, these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application.

1. Create an application like the one in “The cart Example” on page 461. The example in this tutorial starts with this example and demonstrates adding basic authentication of the client to this application. The example application discussed in this section can be found at tut-install/examples/security/cart-secure/.

2. If you have not already done so, complete the steps in “To Set Up Your System for Running the Security Examples” on page 729 to configure your system for running the tutorial applications.

3. Modify the source code for the enterprise bean, CartBean.java, to specify which roles are authorized to access which protected methods. This step is discussed in “Annotating the Bean” on page 750.

4. Build, package, and deploy the enterprise bean; then build and run the client application by following the steps in “To Run the cart-secure Example Using NetBeans IDE” on page 752 or “To Run the cart-secure Example Using Ant” on page 753.

Annotating the Bean

The source code for the original cart application was modified as shown in the following code snippet (modifications in bold). The resulting file can be found in tut-install/examples/security/cart-secure/cart-secure-ejb/src/java/cart/ejb/CartBean.java.

The code snippet is as follows:

```java
package cart.ejb;
import cart.util.BookException;
```
import cart.util.IdVerifier;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;

@Stateful
@DeclareRoles("TutorialUser")
public class CartBean implements Cart {
    List<String> contents;
    String customerId;
    String customerName;

    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
        customerId = "0";
        contents = new ArrayList<String>();
    }

    public void initialize(String person, String id) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }
        IdVerifier idChecker = new IdVerifier();
        if (idChecker.validate(id)) {
            customerId = id;
        } else {
            throw new BookException("Invalid id: " + id);
        }
        contents = new ArrayList<String>();
    }

    @RolesAllowed("TutorialUser")
    public void addBook(String title) {
        contents.add(title);
    }

    @RolesAllowed("TutorialUser")
    public void removeBook(String title) throws BookException {
        boolean result = contents.remove(title);
        if (result == false) {
            throw new BookException("" + title + " not in cart.");
        }
    }
}

Examples: Securing Enterprise Beans
@RolesAllowed("TutorialUser")
public List<String> getContents() {
    return contents;
}

@Remove()
@RolesAllowed("TutorialUser")
public void remove() {
    contents = null;
}

The @RolesAllowed annotation is specified on methods for which you want to restrict access. In this example, only users in the role of TutorialUser will be allowed to add and remove books from the cart and to list the contents of the cart. A @RolesAllowed annotation implicitly declares a role that will be referenced in the application; therefore, no @DeclareRoles annotation is required. The presence of the @RolesAllowed annotation also implicitly declares that authentication will be required for a user to access these methods. If no authentication method is specified in the deployment descriptor, the type of authentication will be user name/password authentication.

▼ To Run the cart-secure Example Using NetBeans IDE

1 Follow the steps in “To Set Up Your System for Running the Security Examples” on page 729.

2 In NetBeans IDE, from the File menu, choose Open Project.

3 In the Open Project dialog, navigate to:
tut-install/examples/security/

4 Select the cart-secure folder.

5 Select the Open as Main Project and Open Required Projects check boxes.

6 Click Open Project.

7 In the Projects tab, right-click the cart-secure project and select Build.

8 In the Projects tab, right-click the cart-secure project and select Deploy.
   This step builds and packages the application into cart-secure.ear, located in the directory
tut-install/examples/security/cart-secure/dist/, and deploys this EAR file to your
GlassFish Server instance.

9 To run the application client, right-click the cart-secure project and select Run.
   A Login for user: dialog box appears.
In the dialog box, type the username and password of a file realm user created on the GlassFish Server and assigned to the group TutorialUser; then click OK.

If the user name and password you enter are authenticated, the output of the application client appears in the Output pane:

```
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
Java Result: 1
```

If the user name and password are not authenticated, the dialog box reappears until you type correct values.

### To Run the cart-secure Example Using Ant

1. Follow the steps in "To Set Up Your System for Running the Security Examples" on page 729.

2. In a terminal window, go to:
   
   `tut-install/examples/security/cart-secure/`

3. To build the application and package it into an EAR file, type the following command at the terminal window or command prompt:
   
   `ant`

4. To deploy the application to the GlassFish Server, type the following command:
   
   `ant deploy`

5. To run the application client, type the following command:
   
   `ant run`

   This task retrieves the application client JAR and runs the application client.

   A Login for user: dialog box appears.

6. In the dialog box, type the user name and password of a file realm user created on the GlassFish Server and assigned to the group TutorialUser; then click OK.

   If the user name and password are authenticated, the client displays the following output:

   ```
   [echo] running application client container.
   [exec] Retrieving book title from cart: Infinite Jest
   [exec] Retrieving book title from cart: Bel Canto
   [exec] Retrieving book title from cart: Kafka on the Shore
   [exec] Removing "Gravity's Rainbow" from cart.
   [exec] Caught a BookException: "Gravity's Rainbow" not in cart.
   [exec] Result: 1
   ```
If the username and password are not authenticated, the dialog box reappears until you type correct values.

**The converter-secure Example: Securing an Enterprise Bean with Programmatic Security**

This example demonstrates how to use the `getCallerPrincipal` and `isCallerInRole` methods with an enterprise bean. This example starts with a very simple EJB application, converter, and modifies the methods of the `ConverterBean` so that currency conversion will occur only when the requester is in the role of `TutorialUser`.

The completed version of this example can be found in the `tut-install/examples/security/converter-secure` directory. This example is based on the unsecured enterprise bean application, converter, which is discussed in Chapter 23, "Getting Started with Enterprise Beans," and is found in the `tut-install/examples/ejb/converter/` directory. This section builds on the example by adding the necessary elements to secure the application by using the `getCallerPrincipal` and `isCallerInRole` methods, which are discussed in more detail in "Accessing an Enterprise Bean Caller’s Security Context" on page 746.

In general, the following steps are necessary when using the `getCallerPrincipal` and `isCallerInRole` methods with an enterprise bean. In the example application included with this tutorial, many of these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application.

1. Create a simple enterprise bean application.
2. Set up a user on the GlassFish Server in the file realm, in the group `TutorialUser`, and set up default principal to role mapping. To do this, follow the steps in "To Set Up Your System for Running the Security Examples" on page 729.
3. Modify the bean to add the `getCallerPrincipal` and `isCallerInRole` methods.
4. If the application contains a web client that is a servlet, specify security for the servlet, as described in "Specifying Security for Basic Authentication Using Annotations" on page 731.
5. Build, package, deploy, and run the application.

**Modifying ConverterBean**

The source code for the original `ConverterBean` class was modified to add the `if..else` clause that tests whether the caller is in the role of `TutorialUser`. If the user is in the correct role, the currency conversion is computed and displayed. If the user is not in the correct role, the computation is not performed, and the application displays the result as 0. The code example can be found in `tut-install/examples/ejb/converter-secure/converter-secure-ejb/src/java/converter/ejb/ConverterBean.java`.

The code snippet (with modifications shown in bold) is as follows:
package converter.ejb;

import java.math.BigDecimal;
import javax.ejb.Stateless;
import java.security.Principal;
import javax.annotation.Resource;
import javax.ejb.SessionContext;
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;

@Stateless()
@DeclareRoles("TutorialUser")
public class ConverterBean{
    @Resource SessionContext ctx;
    private BigDecimal yenRate = new BigDecimal("89.5094");
    private BigDecimal euroRate = new BigDecimal("0.0081");

    @RolesAllowed("TutorialUser")
    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = new BigDecimal("0.0");
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (ctx.isCallerInRole("TutorialUser")) {
            result = dollars.multiply(yenRate);
            return result.setScale(2, BigDecimal.ROUND_UP);
        } else {
            return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }

    @RolesAllowed("TutorialUser")
    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = new BigDecimal("0.0");
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (ctx.isCallerInRole("TutorialUser")) {
            result = yen.multiply(euroRate);
            return result.setScale(2, BigDecimal.ROUND_UP);
        } else {
            return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }
}

Modifying ConverterServlet

The following annotations specify security for the converter web client, ConverterServlet:

@WebServlet(name = "ConverterServlet", urlPatterns = {"/"/})
@ServletSecurity(
    @HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL, rolesAllowed = {"TutorialUser"}))
To Build, Package, and Deploy the converter-secure Example Using NetBeans IDE

1. Follow the steps in “To Set Up Your System for Running the Security Examples” on page 729.
2. In NetBeans IDE, from the File menu, choose Open Project.
3. In the Open Project dialog, navigate to:
tut-install/examples/security/
4. Select the converter-secure folder.
5. Select the Open as Main Project check box.
6. Click Open Project.
7. Right-click the converter-secure project and select Build.
8. Right-click the converter-secure project and select Deploy.

To Build, Package, and Deploy the converter-secure Example Using Ant

1. Follow the steps in “To Set Up Your System for Running the Security Examples” on page 729.
2. In a terminal window, go to:
tut-install/examples/security/converter-secure/
3. Type the following command:
   ant all
   This command both builds and deploys the example.

To Run the converter-secure Example

1. Open a web browser to the following URL:
   http://localhost:8080/converter-secure
   An Authentication Required dialog box appears.
2. Type a user name and password combination that corresponds to a user who has already been created in the file realm of the GlassFish Server and has been assigned to the group of TutorialUser; then click OK.
3 Type 100 in the input field and click Submit.
   A second page appears, showing the converted values.
This chapter provides advanced information on securing Java EE applications.

The following topics are addressed here:

- “Working with Digital Certificates” on page 759
- “Authentication Mechanisms” on page 764
- “Using Form-Based Login in JavaServer Faces Web Applications” on page 769
- “Using the JDBC Realm for User Authentication” on page 771
- “Securing HTTP Resources” on page 775
- “Securing Application Clients” on page 778
- “Securing Enterprise Information Systems Applications” on page 780
- “Configuring Security Using Deployment Descriptors” on page 783
- “Further Information about Security” on page 785

### Working with Digital Certificates

Digital certificates for the GlassFish Server have already been generated and can be found in the directory `domain-dir/config/`. These digital certificates are self-signed and are intended for use in a development environment; they are not intended for production purposes. For production purposes, generate your own certificates and have them signed by a certificate authority (CA).

To use the Secure Sockets Layer (SSL), an application or web server must have an associated certificate for each external interface, or IP address, that accepts secure connections. The theory behind this design is that a server should provide some kind of reasonable assurance that its owner is who you think it is, particularly before receiving any sensitive information. It may be useful to think of a certificate as a “digital driver’s license” for an Internet address. The certificate states with which company the site is associated, along with some basic contact information about the site owner or administrator.
The digital certificate is cryptographically signed by its owner and is difficult for anyone else to forge. For sites involved in e-commerce or in any other business transaction in which authentication of identity is important, a certificate can be purchased from a well-known CA such as VeriSign or Thawte. If your server certificate is self-signed, you must install it in the GlassFish Server keystore file (keystore.jks). If your client certificate is self-signed, you should install it in the GlassFish Server truststore file (cacerts.jks).

Sometimes, authentication is not really a concern. For example, an administrator might simply want to ensure that data being transmitted and received by the server is private and cannot be snooped by anyone eavesdropping on the connection. In such cases, you can save the time and expense involved in obtaining a CA certificate and simply use a self-signed certificate.

SSL uses public-key cryptography, which is based on key pairs. Key pairs contain one public key and one private key. Data encrypted with one key can be decrypted only with the other key of the pair. This property is fundamental to establishing trust and privacy in transactions. For example, using SSL, the server computes a value and encrypts it by using its private key. The encrypted value is called a digital signature. The client decrypts the encrypted value by using the server’s public key and compares the value to its own computed value. If the two values match, the client can trust that the signature is authentic, because only the private key could have been used to produce such a signature.

Digital certificates are used with HTTPS to authenticate web clients. The HTTPS service of most web servers will not run unless a digital certificate has been installed. Use the procedure outlined in the next section, “Creating a Server Certificate” on page 760, to set up a digital certificate that can be used by your application or web server to enable SSL.

One tool that can be used to set up a digital certificate is keytool, a key and certificate management utility that ships with the JDK. This tool enables users to administer their own public/private key pairs and associated certificates for use in self-authentication, whereby the user authenticates himself or herself to other users or services, or data integrity and authentication services, using digital signatures. The tool also allows users to cache the public keys, in the form of certificates, of their communicating peers.

For a better understanding of keytool and public-key cryptography, see “Further Information about Security” on page 785 for a link to the keytool documentation.

Creating a Server Certificate

A server certificate has already been created for the GlassFish Server and can be found in the domain-dir/config/ directory. The server certificate is in keystore.jks. The cacerts.jks file contains all the trusted certificates, including client certificates.

If necessary, you can use keytool to generate certificates. The keytool utility stores the keys and certificates in a file termed a keystore, a repository of certificates used for identifying a client or a server. Typically, a keystore is a file that contains one client’s or one server’s identity. The keystore protects private keys by using a password.
If you don’t specify a directory when specifying the keystore file name, the keystores are created in the directory from which the keytool command is run. This can be the directory where the application resides, or it can be a directory common to many applications.

The general steps for creating a server certificate are as follows.
1. Create the keystore.
2. Export the certificate from the keystore.
3. Sign the certificate.
4. Import the certificate into a truststore: a repository of certificates from parties with which you expect to communicate or from Certificate Authorities that you trust to identify parties. The truststore is used by the client to verify the certificate that is sent by the server. A truststore typically contains more than one certificate.

"To Use keytool to Create a Server Certificate" on page 761 provides specific information on using the keytool utility to perform these steps.

**To Use keytool to Create a Server Certificate**

Run keytool to generate a new key pair in the default development keystore file, keystore.jks. This example uses the alias server-alias to generate a new public/private key pair and wrap the public key into a self-signed certificate inside keystore.jks. The key pair is generated by using an algorithm of type RSA, with a default password of changeit. For more information and other examples of creating and managing keystore files, read the keytool documentation.

Note – RSA is public-key encryption technology developed by RSA Data Security, Inc.

From the directory in which you want to create the key pair, run keytool as shown in the following steps.

1. **Generate the server certificate.**

   Type the keytool command all on one line:
   ```bash
   java-home/bin/keytool -genkey -alias server-alias -keyalg RSA -keypass changeit -storepass changeit -keystore keystore.jks
   ```
   
   When you press Enter, keytool prompts you to enter the server name, organizational unit, organization, locality, state, and country code.

   You must type the server name in response to keytool’s first prompt, in which it asks for first and last names. For testing purposes, this can be localhost.

   When you run the example applications, the host (server name) specified in the keystore must match the host identified in the javaee.server.name property specified in the tut-install/examples/bp-project/build.properties file (by default, this is localhost).
2 Export the generated server certificate in keystore.jks into the file server.cer.

Type the keytool command all on one line:

```
java-home/bin/keytool -export -alias server-alias -storepass changeit -file server.cer -keystore keystore.jks
```

3 If you want to have the certificate signed by a CA, read the example in the keytool documentation.

4 To add the server certificate to the truststore file, cacerts.jks, run keytool from the directory where you created the keystore and server certificate.

Use the following parameters:

```
java-home/bin/keytool -import -v -trustcacerts -alias server-alias -file server.cer -keystore cacerts.jks -keypass changeit -storepass changeit
```

Information on the certificate, such as that shown next, will appear:

- **Owner**: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
- **Issuer**: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
- **Serial number**: 3e932169
- **Certificate fingerprints**:
- **Signature algorithm name**: SHA256withRSA
- **Version**: 3

Trust this certificate? [no]:

5 Type yes, then press the Enter or Return key.

The following information appears:

- Certificate was added to keystore
  [Storing cacerts.jks]

### Adding Users to the Certificate Realm

In the certificate realm, user identity is set up in the GlassFish Server security context and populated with user data obtained from cryptographically verified client certificates. For step-by-step instructions for creating this type of certificate, see "Working with Digital Certificates" on page 759.
Using a Different Server Certificate with the GlassFish Server

Follow the steps in “Creating a Server Certificate” on page 760 to create your own server certificate, have it signed by a CA, and import the certificate into keystore.jks.

Make sure that when you create the certificate, you follow these rules:

- When you create the server certificate, keytool prompts you to enter your first and last name. In response to this prompt, you must enter the name of your server. For testing purposes, this can be localhost.
- The server/host specified in the keystore must match the host identified in the javaee.server.name property specified in the tut-install/examples/bp-project/build.properties file for running the example applications.
- Your key/certificate password in keystore.jks should match the password of your keystore, keystore.jks. This is a bug. If there is a mismatch, the Java SDK cannot read the certificate and you get a “tampered” message.
- If you want to replace the existing keystore.jks, you must either change your keystore’s password to the default password (changeit) or change the default password to your keystore’s password.

To Specify a Different Server Certificate

To specify that the GlassFish Server should use the new keystore for authentication and authorization decisions, you must set the JVM options for the GlassFish Server so that they recognize the new keystore. To use a different keystore from the one provided for development purposes, follow these steps.

1. Start the GlassFish Server if you haven’t already done so. Information on starting the GlassFish Server can be found in “Starting and Stopping the GlassFish Server” on page 73.
2. Open the GlassFish Server Administration Console in a web browser at http://localhost:4848.
3. Expand Configurations, then expand server-config, then click JVM Settings.
4. Select the JVM Options tab.
5. Change the following JVM options so that they point to the location and name of the new keystore. The current settings are shown below:

-Djavax.net.ssl.keyStore=${com.sun.aas.instanceRoot}/config/keystore.jks
-Djavax.net.ssl.trustStore=${com.sun.aas.instanceRoot}/config/cacerts.jks
If you’ve changed the keystore password from its default value, you need to add the password option as well:
-Djavax.net.ssl.keyStorePassword=your-new-password

Click Save, then restart GlassFish Server.

Authentication Mechanisms

This section discusses the client authentication and mutual authentication mechanisms.

Client Authentication

With client authentication, the web server authenticates the client by using the client’s public key certificate. Client authentication is a more secure method of authentication than either basic or form-based authentication. It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the client’s public key certificate. SSL technology provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a public key certificate as the digital equivalent of a passport. The certificate is issued by a trusted organization, a certificate authority (CA), and provides identification for the bearer.

Before using client authentication, make sure the client has a valid public key certificate. For more information on creating and using public key certificates, read “Working with Digital Certificates” on page 759.

The following example shows how to declare client authentication in your deployment descriptor:

```xml
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

Mutual Authentication

With mutual authentication, the server and the client authenticate each other. Mutual authentication is of two types:

- Certificate-based (see Figure 42–1)
- User name/password-based (see Figure 42–2)
When using certificate-based mutual authentication, the following actions occur.

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the server's certificate.
4. If successful, the client sends its certificate to the server.
5. The server verifies the client's credentials.
6. If successful, the server grants access to the protected resource requested by the client.

Figure 42–1 shows what occurs during certificate-based mutual authentication.

In user name/password-based mutual authentication, the following actions occur.

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the server's certificate.
4. If successful, the client sends its username and password to the server.
5. The server verifies the client’s credentials.
6. If the verification is successful, the server grants access to the protected resource requested by the client.

Figure 42–2 shows what occurs during user name/password-based mutual authentication.

**Enabling Mutual Authentication over SSL**

This section discusses setting up client-side authentication. Enabling both server-side and client-side authentication is called mutual, or two-way, authentication. In client authentication, clients are required to submit certificates issued by a certificate authority that you choose to accept.
There are at least two ways to enable mutual authentication over SSL:

- The preferred method is to set the method of authentication in the web.xml application deployment descriptor to CLIENT-CERT. This enforces mutual authentication by modifying the deployment descriptor of the given application. In this way, client authentication is enabled only for a specific resource controlled by the security constraint, and the check is performed only when the application requires client authentication.

- A less commonly used method is to set the clientAuth property in the certificate realm to true if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. A false value (which is the default) will not require a certificate chain unless the client requests a resource protected by a security constraint that uses CLIENT-CERT authentication. When you enable client authentication by setting the clientAuth property to true, client authentication will be required for all the requests going through the specified SSL port. If you turn clientAuth on, it is on all of the time, which can severely degrade performance.

When client authentication is enabled in both of these ways, client authentication will be performed twice.

**Creating a Client Certificate for Mutual Authentication**

If you have a certificate signed by a trusted Certificate Authority (CA) such as Verisign, and the GlassFish Server cacerts.jks file already contains a certificate verified by that CA, you do not need to complete this step. You need to install your certificate in the GlassFish Server certificate file only when your certificate is self-signed.

From the directory where you want to create the client certificate, run keytool as outlined here. When you press Enter, keytool prompts you to enter the server name, organizational unit, organization, locality, state, and country code.

You must enter the server name in response to keytool’s first prompt, in which it asks for first and last names. For testing purposes, this can be localhost. The host specified in the keystore must match the host identified in the javaee.server.host variable specified in your tut-install/examples/bp-project/build.properties file. If this example is to verify mutual authentication and you receive a runtime error stating that the HTTPS host name is wrong, re-create the client certificate, being sure to use the same host name you will use when running the example. For example, if your machine name is duke, then enter duke as the certificate CN or when prompted for first and last names. When accessing the application, enter a URL that points to the same location (for example, https://duke:8181/mutualauth/hello). This is necessary because during SSL handshake, the server verifies the client certificate by comparing the certificate name to the host name from which it originates.

\[\text{To Create a Client Certificate and Keystore}\]

To create a keystore named clientkeystore.jks that contains a client certificate named client.cer, follow these steps.
1  Create a backup copy of the server truststore file. To do this,

   a. Change to the directory containing the server's keystore and truststore files,
      \texttt{domain-dir/config}.

   b. Copy \texttt{cacerts.jks} to \texttt{cacerts.backup.jks}.

   c. Copy \texttt{keystore.jks} to \texttt{keystore.backup.jks}.

      Do not put client certificates in the \texttt{cacerts.jks} file. Any certificate you add to the \texttt{cacerts} file effectively can be a trusted root for any and all certificate chains. After you have completed development, delete the development version of the \texttt{cacerts} file and replace it with the original copy.

2  Generate the client certificate. Enter the following command from the directory where you want to generate the client certificate:

   \begin{verbatim}
   java-home\bin\keytool -genkey -alias client-alias -keyalg RSA
   -keypass changeit -storepass changeit -keystore client_keystore.jks
   \end{verbatim}

3  Export the generated client certificate into the file \texttt{client.cer}:

   \begin{verbatim}
   java-home\bin\keytool -export -alias client-alias -storepass changeit
   -file client.cer -keystore client_keystore.jks
   \end{verbatim}

4  Add the certificate to the truststore file \texttt{domain-dir/config/cacerts.jks}. Run keytool from the directory where you created the keystore and client certificate. Use the following parameters:

   \begin{verbatim}
   java-home\bin\keytool -import -v -trustcacerts -alias client-alias
   -file client.cer -keystore domain-dir/config/cacerts.jks
   -keypass changeit -storepass changeit
   \end{verbatim}

   The keytool utility returns a message like this one:

   Owner: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
   Issuer: CN=localhost, OU=My Company, O=Software, L=Santa Clara, ST=CA, C=US
   Serial number: 3e39e66a
   Certificate fingerprints:
   Signature algorithm name: SHA1withRSA
   Version: 3
   Trust this certificate? [no]: yes
   Certificate was added to keystore
   [Storing cacerts.jks]

5  Restart GlassFish Server.
Using Form-Based Login in JavaServer Faces Web Applications

This section describes strategies for implementing form-based login in JavaServer Faces applications.

Using j_security_check in JavaServer Faces Forms

The most common way of authenticating a user in web applications is through a login form. As described in "Form-Based Authentication" on page 721, Java EE security defines the j_security_check action for login forms. This allows the web container to authenticate users from many different web application resources. Facelets forms, using the h:form, h:inputText, and h:inputSecret tags, however, generate the action and input IDs automatically, which means developers are unable to specify j_security_check as the form action, nor can they set the user name and password input field IDs to j_username and j_password, respectively.

Using standard HTML form tags allows developers to specify the correct action and input IDs for the form.

```html
<form action="j_security_check" method="POST">
   <input type="text" name="j_username" />
   <input type="secret" name="j_password" />
   ...
</form>
```

This form, however, doesn't have access to the features of a JavaServer Faces application, such as automatic localization of strings and the use of templating to define the look and feel of the pages. A standard HTML form, in combination with Facelets and HTML tags, allows developers to use localized strings for the input field labels while still ensuring the form uses standard Java EE security:

```html
<form action="j_security_check" method="POST">
   <h:outputLabel for="j_username">#{bundle['login.username']}:</h:outputLabel>
   <h:inputText id="j_username" size="20" />

   <h:outputLabel for="j_password">#{bundle['login.password']}:</h:outputLabel>
   <h:inputSecret id="j_password" size="20" />

   <input type="submit" value="#{bundle['login.submit']}" />
</form>
```
Using a Managed Bean for Authentication in JavaServer Faces Applications

A managed bean can authenticate users of a JavaServer Faces application, which allows regular Facelets form tags to be used instead of a mix of standard HTML and Facelets tags. In this case, the managed bean defines login and logout methods, and Facelets forms call these methods in the action attribute. The managed bean's methods call the javax.servlet.http.HttpServletRequest.login and HttpServletRequest.logout methods to manage user authentication.

In the following managed bean, a stateless session bean uses the user credentials passed to the login method to authenticate the user and resets the caller identity of the request when the logout method is called.

```java
@Stateless
@Named
public class LoginBean {
    private String username;
    private String password;

    public String getUsername() {
        return this.username;
    }

    public void setUserName(String username) {
        this.username = username;
    }

    public String getPassword() {
        return this.password;
    }

    public void setPassword() {
        this.password = password;
    }

    ... public String login () {
        FacesContext context = FacesContext.getCurrentInstance();
        HttpServletRequest request = (HttpServletRequest)
                context.getExternalContext().getRequest();
        try {
            request.login(this.username, this.password);
        } catch (ServletException e) {
            context.addMessage(null, new FacesMessage("Login failed.");
            return "error";
        }
        return "admin/index";
    }

    public void logout() {
        FacesContext context = FacesContext.getCurrentInstance();
    }
```
HttpServletRequest request = (HttpServletRequest)
    context.getExternalContext().getRequest();
try {
    request.logout();
} catch (ServletException e) {
    ...
    context.addMessage(null, new FacesMessage("Logout failed."));
}
}

The Facelets form then calls these methods for user login and logout.

```xml
<form>
    <label for="usernameInput">#{bundle['login.username']}
        <input type="text" id="usernameInput" value="#{loginBean.username}" required="true" />
    </label>
    <br />
    <label for="passwordInput">#{bundle['login.password']}
        <input type="password" id="passwordInput" value="#{loginBean.password}" required="true" />
    </label>
    <br />
    <button value="#{bundle['login.submit']}" action="#{loginBean.login}" />
</form>
```

Using the JDBC Realm for User Authentication

An authentication realm, sometimes called a security policy domain or security domain, is a scope over which an application server defines and enforces a common security policy. A realm contains a collection of users, who may or may not be assigned to a group. GlassFish Server comes preconfigured with the file, certificate, and administration realms. An administrator can also set up LDAP, JDBC, digest, or custom realms.

An application can specify in its deployment descriptor which realm to use. If the application does not specify a realm, GlassFish Server uses its default realm, the file realm. If an application specifies that a JDBC realm is to be used for user authentication, GlassFish Server will retrieve user credentials from a database. The application server uses the database information and the enabled JDBC realm option in the configuration file.

A database provides an easy way to add, edit, or delete users at runtime and enables users to create their own accounts without any administrative assistance. Using a database also has an additional benefit, providing a place to securely store any extra user information. A realm can be thought of as a database of user names and passwords that identify valid users of a web application or set of web applications with an enumeration of the list of roles associated with
each valid user. Access to specific web application resources is granted to all users in a particular role, instead of enumerating a list of associated users. A user name can have any number of roles associated with it.

Two of the tutorial case studies, Chapter 52, "Duke's Tutoring Case Study Example," and Chapter 53, "Duke's Forest Case Study Example," use the JDBC realm for user authentication. Where appropriate, reference will be made to one or both of these examples.

## To Configure a JDBC Authentication Realm

GlassFish Server enables administrators to specify a user's credentials (user name and password) in the JDBC realm instead of in the connection pool. This prevents other applications from browsing the database tables for user credentials. By default, storing passwords as clear text is not supported in the JDBC realm. Under normal circumstances, passwords should not be stored as clear text.

1. **Create the database tables in which user credentials for the realm will be stored.**

How you create the database tables depends on the database you are using. Duke's Forest uses an Ant task, `create-tables`, in the `build.xml` file for the Entities project. The task executes an SQL script, `create.sql`, that creates the FOREST.PERSON, FOREST.GROUPS, and FOREST.PERSON_GROUPS database tables, as shown below:

```sql
CREATE TABLE "FOREST"."PERSON"
(
    ID int NOT NULL PRIMARY KEY GENERATED ALWAYS AS IDENTITY
    (START WITH 1, INCREMENT BY 1),
    FIRSTNAME varchar(50) NOT NULL,
    LASTNAME varchar(100) NOT NULL,
    EMAIL varchar(45) NOT NULL UNIQUE,
    ADDRESS varchar(45) NOT NULL,
    CITY varchar(45) NOT NULL,
    PASSWORD varchar(100),
    DTYPE varchar(31)
)
;
CREATE UNIQUE INDEX SQL_PERSON_EMAIL_INDEX ON "FOREST"."PERSON"(EMAIL);
CREATE UNIQUE INDEX SQL_PERSON_ID_INDEX ON "FOREST"."PERSON"(ID);
CREATE TABLE "FOREST"."GROUPS"
(
    ID int NOT NULL PRIMARY KEY GENERATED ALWAYS AS IDENTITY
    (START WITH 1, INCREMENT BY 1),
    NAME varchar(50) NOT NULL,
    DESCRIPTION varchar(300)
)
;
CREATE TABLE "FOREST"."PERSON_GROUPS"
(
    GROUPS_ID int NOT NULL,
    EMAIL varchar(45) NOT NULL
);```

Using the JDBC Realm for User Authentication
The Duke's Tutoring case study uses a singleton bean, ConfigBean, to create its database tables, instead of using SQL commands.

2 Add user credentials to the database tables you created.

How you add user credentials to the database tables depends on the database that you are using. Duke's Forest uses an Ant task. The create-tables Ant task for Duke's Forest adds the user credentials to the tables created in the previous step:

INSERT INTO "FOREST"."PERSON" (FIRSTNAME,LASTNAME,EMAIL,ADDRESS,CITY,PASSWORD,DTYPE) VALUES ('Robert','Exampler','robert@example.com','Example street','San Francisco','81dc9bdb52d04dc20036dbd8313ed055','Customer');
INSERT INTO "FOREST"."PERSON" (FIRSTNAME,LASTNAME,EMAIL,ADDRESS,CITY,PASSWORD,DTYPE) VALUES ('Admin','Admin','admin@example.com','Example street','San Francisco','81dc9bdb52d04dc20036dbd8313ed055','Administrator');
INSERT INTO "FOREST"."PERSON" (FIRSTNAME,LASTNAME,EMAIL,ADDRESS,CITY,PASSWORD,DTYPE) VALUES ('Jack','Frost','jack@example.com','Example Blvd','San Francisco','81dc9bdb52d04dc20036dbd8313ed055','Customer');
INSERT INTO "FOREST"."PERSON" (FIRSTNAME,LASTNAME,EMAIL,ADDRESS,CITY,PASSWORD,DTYPE) VALUES ('Payment','User','paymentUser@dukesforest.com',''-','-','58175e1df62779046a3a4e2483575937','Customer');

INSERT INTO "FOREST"."GROUPS" (NAME, DESCRIPTION) VALUES ('USERS', 'Users of the store');
INSERT INTO "FOREST"."GROUPS" (NAME, DESCRIPTION) VALUES ('ADMINS', 'Administrators of the store');

INSERT INTO "FOREST"."PERSON_GROUPS" (GROUPS_ID,EMAIL) VALUES (1,'robert@example.com');
INSERT INTO "FOREST"."PERSON_GROUPS" (GROUPS_ID,EMAIL) VALUES (2,'admin@example.com');
INSERT INTO "FOREST"."PERSON_GROUPS" (GROUPS_ID,EMAIL) VALUES (1,'jack@example.com');
INSERT INTO "FOREST"."PERSON_GROUPS" (GROUPS_ID,EMAIL) VALUES (1,'paymentUser@dukesforest.com');

The Duke's Tutoring case study uses a singleton bean, ConfigBean, to populate its database tables, instead of using SQL commands.
3 Create a JDBC connection pool for the database.
Duke's Forest uses an Ant task, create-forest-pool, to create the derby_net_forest_forestPool JDBC connection pool for the database:

```xml
<target name="create-forest-pool" description="create JDBC connection pool">
  <antcall target="create-jdbc-connection-pool">
    <param name="pool.name" value="derby_net_forest_forestPool"/>
  </antcall>
</target>
```

You can also use the Administration Console or the command line to create a connection pool.

4 Create a JDBC resource for the database.
Duke's Forest uses an Ant task, create-forest-resource, to create the jdbc/forest JDBC resource for the database:

```xml
<target name="create-forest-resource" depends="create-forest-pool" description="create JDBC resource">
  <antcall target="create-jdbc-resource">
    <param name="pool.name" value="derby_net_forest_forestPool"/>
    <param name="jdbc.resource.name" value="jdbc/forest"/>
  </antcall>
</target>
```

You can also use the Administration Console or the command line to create a JDBC resource.

5 Create a realm.
Duke's Forest uses an Ant task, create-forest-realm, to create jdbcRealm, the JDBC realm used for user authentication:

```xml
<target name="create-forest-realm" depends="create-forest-resource" description="create JDBC realm">
  <antcall target="create-jdbc-realm">
    <param name="jdbc.resource.name" value="jdbc/forest"/>
    <param name="jdbc.realm.name" value="jdbcRealm"/>
    <param name="user.table.name" value="forest.PERSON"/>
    <param name="user.name.column" value="email"/>
    <param name="password.column" value="password"/>
    <param name="group.table" value="forest.GROUPS"/>
    <param name="group.name.column" value="name"/>
    <param name="assign.groups" value="USERS,ADMINS"/>
    <param name="digest.algorithm" value="MD5"/>
  </antcall>
</target>
```

This task associates the resource with the realm, defines the tables and columns for users and groups used for authentication, and defines the digest algorithm that will be used for storing passwords in the database.

You can also use the Administration Console or the command line to create a realm.

6 Modify the deployment descriptor for your application to specify the JDBC realm:

- For an enterprise application in an EAR file, modify the glassfish-application.xml file.
For a web application in a WAR file, modify the web.xml file.

For an enterprise bean in an EJB JAR file, modify the glassfish-ejb-jar.xml file.

For example, for the Duke's Forest application, the web.xml file specifies the jdbcRealm realm:

```xml
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>jdbcRealm</realm-name>
  <form-login-config>
    <form-login-page>/login.xhtml</form-login-page>
    <form-error-page>/login.xhtml</form-error-page>
  </form-login-config>
</login-config>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Secure Pages</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ADMINS</role-name>
  </auth-constraint>
</security-constraint>
```

Form-based login is specified for all web pages under /admin. Access to those pages will be allowed only to users in the ADMINS role.

7 Assign security roles to users or groups of users in the realm.

To assign a security role to a group or to a user, add a security-role-mapping element to the application server-specific deployment descriptor, in this case glassfish-web.xml:

```xml
<security-role-mapping>
  <role-name>USERS</role-name>
  <group-name>USERS</group-name>
</security-role-mapping>

<security-role-mapping>
  <role-name>ADMINS</role-name>
  <group-name>ADMINS</group-name>
</security-role-mapping>
```

Since GlassFish Server users are assigned to groups during the user creation process, this is more efficient than mapping security roles to individual users.

**Securing HTTP Resources**

When a request URI is matched by multiple constrained URL patterns, the constraints that apply to the request are those that are associated with the best matching URL pattern. The servlet matching rules defined in Chapter 12, “Mapping Requests To Servlets,” in the Java Servlet 3.0 Specification, are used to determine the best matching URL pattern to the request URI. No protection requirements apply to a request URI that is not matched by a constrained URL pattern. The HTTP method of the request plays no role in selecting the best matching URL pattern for a request.
When HTTP methods are listed within a constraint definition, the protections defined by the constraint are applied to the listed methods only.

When HTTP methods are not listed within a constraint definition, the protections defined by the constraint apply to the complete set of HTTP methods, including HTTP extension methods.

When constraints with different protection requirements apply to the same combination of URL patterns and HTTP methods, the rules for combining the protection requirements are as defined in Section 13.8.1, "Combining Constraints," in the Java Servlet 3.0 Specification.

Follow these guidelines to properly secure a web application:

- Do not list HTTP methods within constraint definitions. This is the simplest way to ensure that you are not leaving HTTP methods unprotected. For example:

```xml
<security-constraint>
  <display-name>Do not enumerate Http Methods</display-name>
  <web-resource-collection>
    <url-pattern>/company/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>sales</role-name>
  </auth-constraint>
</security-constraint>
```

If you list methods in a constraint, all non-listed methods of the effectively infinite set of possible HTTP methods, including extension methods, will be unprotected. The following example shows a constraint that lists the GET method and thus defines no protection on any of the other possible HTTP methods. Do not use such a constraint unless you are certain that this is the protection scheme you intend to define.

```xml
<security-constraint>
  <display-name>Protect GET only, leave all other methods unprotected</display-name>
  <web-resource-collection>
    <url-pattern>/company/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>sales</role-name>
  </auth-constraint>
</security-constraint>
```

- If you need to apply specific types of protection to specific HTTP methods, make sure you define constraints to cover every method that you want to permit, with or without constraint, at the corresponding URL patterns. If there are any methods that you do not want to permit, you must also create a constraint that denies access to those methods at the same patterns; for an example, see security constraint #5 in the next bullet.

For example, to permit GET and POST, where POST requires authentication and GET is permitted without constraint, you could define the following constraints:
The simplest way to ensure that you deny all HTTP methods except those that you want to be permitted is to use http-method-omission elements to omit those HTTP methods from the security constraint, and also to define an auth-constraint that names no roles. The security constraint will apply to all methods except those that were named in the omissions, and the constraint will apply only to the resources matched by the patterns in the constraint.

For example, the following constraint excludes access to all methods except GET and POST at the resources matched by the pattern /company/*:

```xml
<security-constraint>
  <display-name>Deny all HTTP methods except GET and POST</display-name>
  <web-resource-collection>
    <url-pattern>/company/*</url-pattern>
    <http-method-omission>GET</http-method-omission>
    <http-method-omission>POST</http-method-omission>
  </web-resource-collection>
  <auth-constraint/>
</security-constraint>
```

If you want to extend these exclusions to the unconstrained parts of your application, also include the URL pattern / (forward slash):

```xml
<security-constraint>
  <display-name>Deny all HTTP methods except GET and POST</display-name>
  <web-resource-collection>
    <url-pattern>/company/*</url-pattern>
    <url-pattern>/</url-pattern>
    <http-method-omission>GET</http-method-omission>
    <http-method-omission>POST</http-method-omission>
  </web-resource-collection>
  <auth-constraint/>
</security-constraint>
```
If, for your web application, you do not want any resource to be accessible unless you explicitly define a constraint that permits access to it, you can define an auth-constraint that names no roles and associate it with the URL pattern `/`. The URL pattern `/` is the weakest matching pattern. Do not list any HTTP methods in this constraint.

```xml
<!-- SECURITY CONSTRAINT #7 -->
<security-constraint>
  <display-name>
    Switch from Constraint to Permission model
    (where everything is denied by default)
  </display-name>
  <web-resource-collection>
    <url-pattern>/</url-pattern>
  </web-resource-collection>
  <auth-constraint/>
</security-constraint>
```

### Securing Application Clients

The Java EE authentication requirements for application clients are the same as for other Java EE components, and the same authentication techniques can be used as for other Java EE application components. No authentication is necessary when accessing unprotected web resources.

When accessing protected web resources, the usual varieties of authentication can be used: HTTP basic authentication, HTTP login-form authentication, or SSL client authentication. "Specifying an Authentication Mechanism in the Deployment Descriptor" on page 722 describes how to specify HTTP basic authentication and HTTP login-form authentication. "Client Authentication" on page 764 describes how to specify SSL client authentication.

Authentication is required when accessing protected enterprise beans. The authentication mechanisms for enterprise beans are discussed in "Securing Enterprise Beans" on page 739.

An application client makes use of an authentication service provided by the application client container for authenticating its users. The container's service can be integrated with the native platform's authentication system so that a single sign-on capability is used. The container can authenticate the user either when the application is started or when a protected resource is accessed.

An application client can provide a class, called a login module, to gather authentication data. If so, the javax.security.auth.callback.CallbackHandler interface must be implemented, and the class name must be specified in its deployment descriptor. The application's callback handler must fully support Callback objects specified in the javax.security.auth.callback package.
Using Login Modules

An application client can use the Java Authentication and Authorization Service (JAAS) to create login modules for authentication. A JAAS-based application implements the `javax.security.auth.callback.CallbackHandler` interface so that it can interact with users to enter specific authentication data, such as user names or passwords, or to display error and warning messages.

Applications implement the `CallbackHandler` interface and pass it to the login context, which forwards it directly to the underlying login modules. A login module uses the callback handler both to gather input, such as a password or smart card PIN, from users and to supply information, such as status information, to users. Because the application specifies the callback handler, an underlying login module can remain independent of the various ways applications interact with users.

For example, the implementation of a callback handler for a GUI application might display a window to solicit user input, or the implementation of a callback handler for a command-line tool might simply prompt the user for input directly from the command line.

The login module passes an array of appropriate callbacks to the callback handler’s `handle` method, such as a `NameCallback` for the user name and a `PasswordCallback` for the password; the callback handler performs the requested user interaction and sets appropriate values in the callbacks. For example, to process a `NameCallback`, the `CallbackHandler` might prompt for a name, retrieve the value from the user, and call the `setName` method of the `NameCallback` to store the name.

For more information on using JAAS for authentication in login modules, refer to the documentation listed in “Further Information about Security” on page 785.

Using Programmatic Login

Programmatic login enables the client code to supply user credentials. If you are using an EJB client, you can use the `com.sun.appserv.security.ProgrammaticLogin` class with its convenient `login` and `logout` methods. Programmatic login is specific to a server.
Securing Enterprise Information Systems Applications

In Enterprise Information Systems (EIS) applications, components request a connection to an EIS resource. As part of this connection, the EIS can require a sign-on for the requester to access the resource. The application component provider has two choices for the design of the EIS sign-on:

- **Container-managed sign-on**: The application component lets the container take the responsibility of configuring and managing the EIS sign-on. The container determines the user name and password for establishing a connection to an EIS instance. For more information, see "Container-Managed Sign-On" on page 780.

- **Component-managed sign-on**: The application component code manages EIS sign-on by including code that performs the sign-on process to an EIS. For more information, see "Component-Managed Sign-On" on page 780.

You can also configure security for resource adapters. See "Configuring Resource Adapter Security" on page 781.

### Container-Managed Sign-On

In container-managed sign-on, an application component does not have to pass any sign-on security information to the `getConnection()` method. The security information is supplied by the container, as shown in the following example (the method call is highlighted in **bold**):

```java
// Business method in an application component
Context initctx = new InitialContext();
// Perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
  (javax.resource.cci.ConnectionFactory)initctx.lookup("java:comp/env/eis/MainframeCxFactory");
// Invoke factory to obtain a connection. The security information is not passed in the getConnection method
javax.resource.cci.Connection cx = cxf.getConnection();
...```

### Component-Managed Sign-On

In component-managed sign-on, an application component is responsible for passing the needed sign-on security information for the resource to the `getConnection` method. For example, security information might be a user name and password, as shown here (the method call is highlighted in **bold**):

```java
// Method in an application component
Context initctx = new InitialContext();
// Perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
  (javax.resource.cci.ConnectionFactory)initctx.lookup("java:comp/env/eis/MainframeCxFactory");
javax.resource.cci.Connection cx = cxf.getConnection();
...```
javax.resource.cci.ConnectionFactory cxf =
   (javax.resource.cci.ConnectionFactory)initctx.lookup(
   "java:comp/env/eis/MainframeCxFactory");

// Get a new ConnectionSpec
com.myeis.ConnectionSpecImpl properties = //..

// Invoke factory to obtain a connection
properties.setUserName("...");
properties.setPassword("...");
javax.resource.cci.Connection cx =
   cxf.getConnection(properties);
...

Configuring Resource Adapter Security

A resource adapter is a system-level software component that typically implements network connectivity to an external resource manager. A resource adapter can extend the functionality of the Java EE platform either by implementing one of the Java EE standard service APIs, such as a JDBC driver, or by defining and implementing a resource adapter for a connector to an external application system. Resource adapters can also provide services that are entirely local, perhaps interacting with native resources. Resource adapters interface with the Java EE platform through Java EE service provider interfaces (Java EE SPI). A resource adapter that uses Java EE SPIs to attach to the Java EE platform will be able to work with all Java EE products.

To configure the security settings for a resource adapter, you need to edit the resource adapter descriptor file, ra.xml. Here is an example of the part of an ra.xml file that configures security properties for a resource adapter:

<authentication-mechanism>
   <authentication-mechanism-type>
      BasicPassword
   </authentication-mechanism-type>
   <credential-interface>
      javax.resource.spi.security.PasswordCredential
   </credential-interface>
</authentication-mechanism>

You can find out more about the options for configuring resource adapter security by reviewing as-install/Lib/dtds/connector_1_0.dtd. You can configure the following elements in the resource adapter deployment descriptor file:

- **Authentication mechanisms**: Use the authentication-mechanism element to specify an authentication mechanism supported by the resource adapter. This support is for the resource adapter, not for the underlying EIS instance.
  
  There are two supported mechanism types:
  
  - BasicPassword, which supports the following interface:

javax.resource.spi.security.PasswordCredential
Kerbv5, which supports the following interface:

    javax.resource.spi.security.GenericCredential

The GlassFish Server does not currently support this mechanism type.

- **Reauthentication support**: Use the `reauthentication-support` element to specify whether the resource adapter implementation supports reauthentication of existing `Managed-Connection` instances. Options are `true` or `false`.

- **Security permissions**: Use the `security-permission` element to specify a security permission that is required by the resource adapter code. Support for security permissions is optional and is not supported in the current release of the GlassFish Server. You can, however, manually update the `server.policy` file to add the relevant permissions for the resource adapter.

  The security permissions listed in the deployment descriptor are different from those required by the default permission set as specified in the connector specification.

  For more information on the implementation of the security permission specification, see the security policy file documentation listed in “Further Information about Security” on page 785.

In addition to specifying resource adapter security in the `ra.xml` file, you can create a security map for a connector connection pool to map an application principal or a user group to a back-end EIS principal. The security map is usually used if one or more EIS back-end principals are used to execute operations (on the EIS) initiated by various principals or user groups in the application.

### To Map an Application Principal to EIS Principals

When using the GlassFish Server, you can use security maps to map the caller identity of the application (principal or user group) to a suitable EIS principal in container-managed transaction-based scenarios. When an application principal initiates a request to an EIS, the GlassFish Server first checks for an exact principal by using the security map defined for the connector connection pool to determine the mapped back-end EIS principal. If there is no exact match, the GlassFish Server uses the wildcard character specification, if any, to determine the mapped back-end EIS principal. Security maps are used when an application user needs to execute an EIS operation that requires execution as a specific identity in the EIS.

To work with security maps, use the Administration Console. From the Administration Console, follow these steps to get to the security maps page.

1. **In the navigation tree, expand the Resources node.**
2. **Expand the Connectors node.**
3. **Select the Connector Connection Pools node.**
4 On the Connector Connection Pools page, click the name of the connection pool for which you want to create a security map.

5 Click the Security Maps tab.

6 Click New to create a new security map for the connection pool.

7 Type a name by which you will refer to the security map, as well as the other required information.
   Click the Help button for more information on the individual options.

Configuring Security Using Deployment Descriptors

The recommended way to configure security in the Java EE 6 platform is with annotations. If you wish to override the security settings at deployment time, you can use security elements in the `web.xml` deployment descriptor to do so. This section describes how to use the deployment descriptor to specify basic authentication and to override default principal-to-role mapping.

Specifying Security for Basic Authentication in the Deployment Descriptor

The elements of the deployment descriptor that add basic authentication to an example tell the server or browser to perform the following tasks:

- Send a standard login dialog to collect user name and password data
- Verify that the user is authorized to access the application
- If authorized, display the servlet to the user

The following sample code shows the security elements for a deployment descriptor that could be used in the example of basic authentication found in the `tut-install/examples/security/hello2_basicauth/` directory:

```
<security-constraint>
    <display-name>SecurityConstraint</display-name>
    <web-resource-collection>
        <web-resource-name>WRCollection</web-resource-name>
        <url-pattern>/greeting</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>TutorialUser</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>file</realm-name>
</login-config>

<security-role>
  <role-name>TutorialUser</role-name>
</security-role>

This deployment descriptor specifies that the request URI /greeting can be accessed only by users who have entered their user names and passwords and have been authorized to access this URL because they have been verified to be in the role TutorialUser. The user name and password data will be sent over a protected transport in order to keep it from being read in transit.

**Specifying Non-Default Principal-to-Role Mapping in the Deployment Descriptor**

To map a role name permitted by the application or module to principals (users) and groups defined on the server, use the security-role-mapping element in the runtime deployment descriptor file (glassfish-application.xml, glassfish-web.xml, or glassfish-ejb-jar.xml). The entry needs to declare a mapping between a security role used in the application and one or more groups or principals defined for the applicable realm of the GlassFish Server. An example for the glassfish-web.xml file is shown below:

```
<glassfish-web-app>
  <security-role-mapping>
    <role-name>DIRECTOR</role-name>
    <principal-name>schwartz</principal-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>DEPT-ADMIN</role-name>
    <group-name>dept-admins</group-name>
  </security-role-mapping>
</glassfish-web-app>
```

The role name can be mapped to either a specific principal (user), a group, or both. The principal or group names referenced must be valid principals or groups in the current default realm of the GlassFish Server. The role-name in this example must exactly match the role-name in the security-role element of the corresponding web.xml file or the role name defined in the @DeclareRoles and/or @RolesAllowed annotations.
Further Information about Security

For more information about security, see:

- Documentation on the keytool command:
  http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html

- Java Authentication and Authorization Service (JAAS) Reference Guide
  http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html

- Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide
  http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASLMDevGuide.html

- Documentation on security policy file syntax:
  http://docs.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html#FileSyntax
Java EE Supporting Technologies

Part VIII explores several technologies that support the Java EE platform. This part contains the following chapters:

- Chapter 43, “Introduction to Java EE Supporting Technologies”
- Chapter 44, “Transactions”
- Chapter 45, “Resources and Resource Adapters”
- Chapter 46, “The Resource Adapter Example”
- Chapter 47, “Java Message Service Concepts”
- Chapter 48, “Java Message Service Examples”
- Chapter 49, “Bean Validation: Advanced Topics”
- Chapter 50, “Using Java EE Interceptors”
Introduction to Java EE Supporting Technologies

The Java EE platform includes several technologies and APIs that extend its functionality. These technologies allow applications to access a wide range of services in a uniform manner. These technologies are explained in greater detail in Chapter 44, “Transactions,” and Chapter 45, “Resources and Resource Adapters,” as well as Chapter 47, “Java Message Service Concepts,” Chapter 48, “Java Message Service Examples,” and Chapter 49, “Bean Validation: Advanced Topics.”

The following topics are addressed here:
- “Transactions in Java EE Applications” on page 789
- “Resources in Java EE Applications” on page 790

Transactions in Java EE Applications

In a Java EE application, a transaction is a series of actions that must all complete successfully, or else all the changes in each action are backed out. Transactions end in either a commit or a rollback.

The Java Transaction API (JTA) allows applications to access transactions in a manner that is independent of specific implementations. JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the transactional application, the Java EE server, and the manager that controls access to the shared resources affected by the transactions.

The JTA defines the UserTransaction interface that applications use to start, commit, or rollback transactions. Application components get a UserTransaction object through a JNDI lookup by using the name java:comp/UserTransaction or by requesting injection of a UserTransaction object. An application server uses a number of JTA-defined interfaces to communicate with a transaction manager; a transaction manager uses JTA-defined interfaces to interact with a resource manager.

Resources in Java EE Applications

A resource is a program object that provides connections to such systems as database servers and messaging systems.

The Java EE Connector Architecture and Resource Adapters

The Java EE Connector architecture enables Java EE components to interact with enterprise information systems (EISs) and EISs to interact with Java EE components. EIS software includes such kinds of systems as enterprise resource planning (ERP), mainframe transaction processing, and nonrelational databases. Connector architecture simplifies the integration of diverse EISs. Each EIS requires only one implementation of the Connector architecture. Because it adheres to the Connector specification, an implementation is portable across all compliant Java EE servers.

The specification defines the contracts for an application server as well as for resource adapters, which are system-level software drivers for specific EIS resources. These standard contracts provide pluggability between application servers and EISs. The Java EE Connector Architecture 1.6 specification defines new system contracts such as Generic Work Context and Security Inflow. The Java EE Connector Architecture 1.6 specification is available at http://jcp.org/en/jsr/detail?id=322.

A resource adapter is a Java EE component that implements the Connector architecture for a specific EIS. A resource adapter can choose to support the following levels of transactions:

- NoTransaction: No transaction support is provided.
- LocalTransaction: Resource manager local transactions are supported.
- XATransaction: The resource adapter supports the XA distributed transaction processing model and the JTA XATransaction interface.

See Chapter 45, “Resources and Resource Adapters,” for a more detailed explanation of resource adapters.

Java Database Connectivity Software

To store, organize, and retrieve data, most applications use relational databases. Java EE applications access relational databases through the JDBC API.
A JDBC resource, or data source, provides applications with a means of connecting to a database. Typically, a JDBC resource is created for each database accessed by the applications deployed in a domain. Transactional access to JDBC resources is available from servlets, JavaServer Faces pages, and enterprise beans. The connection pooling and distributed transaction features are intended for use by JDBC drivers to coordinate with an application server. For more information, see “DataSource Objects and Connection Pools” on page 806.

Java Message Service

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

The Java Message Service (JMS) API allows applications to create, send, receive, and read messages. It defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.
Transactions

A typical enterprise application accesses and stores information in one or more databases. Because this information is critical for business operations, it must be accurate, current, and reliable. Data integrity would be lost if multiple programs were allowed to update the same information simultaneously or if a system that failed while processing a business transaction were to leave the affected data only partially updated. By preventing both of these scenarios, software transactions ensure data integrity. Transactions control the concurrent access of data by multiple programs. In the event of a system failure, transactions make sure that after recovery, the data will be in a consistent state.

The following topics are addressed here:

- “What Is a Transaction?” on page 793
- “Container-Managed Transactions” on page 794
- “Bean-Managed Transactions” on page 800
- “Transaction Timeouts” on page 801
- “Updating Multiple Databases” on page 802
- “Transactions in Web Components” on page 803
- “Further Information about Transactions” on page 803

What Is a Transaction?

To emulate a business transaction, a program may need to perform several steps. A financial program, for example, might transfer funds from a checking account to a savings account by using the steps listed in the following pseudocode:

```java
begin transaction
    debit checking account
    credit savings account
    update history log
commit transaction
```
Either all or none of the three steps must complete. Otherwise, data integrity is lost. Because the steps within a transaction are a unified whole, a transaction is often defined as an indivisible unit of work.

A transaction can end in two ways: with a commit or with a rollback. When a transaction commits, the data modifications made by its statements are saved. If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction. In the pseudocode, for example, if a disk drive were to crash during the credit step, the transaction would roll back and undo the data modifications made by the debit statement. Although the transaction fails, data integrity would be intact because the accounts still balance.

In the preceding pseudocode, the begin and commit statements mark the boundaries of the transaction. When designing an enterprise bean, you determine how the boundaries are set by specifying either container-managed or bean-managed transactions.

**Container-Managed Transactions**

In an enterprise bean with container-managed transaction demarcation, the EJB container sets the boundaries of the transactions. You can use container-managed transactions with any type of enterprise bean: session or message-driven. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction. By default, if no transaction demarcation is specified, enterprise beans use container-managed transaction demarcation.

Typically, the container begins a transaction immediately before an enterprise bean method starts and commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method.

Container-managed transactions do not require all methods to be associated with transactions. When developing a bean, you can set the transaction attributes to specify which of the bean's methods are associated with transactions.

Enterprise beans that use container-managed transaction demarcation must not use any transaction-management methods that interfere with the container's transaction demarcation boundaries. Examples of such methods are the commit, setAutoCommit, and rollback methods of java.sql.Connection or the commit and rollback methods of javax.jms.Session. If you require control over the transaction demarcation, you must use application-managed transaction demarcation.

Enterprise beans that use container-managed transaction demarcation also must not use the javax.transaction.UserTransaction interface.
Transaction Attributes

A transaction attribute controls the scope of a transaction. Figure 44–1 illustrates why controlling the scope is important. In the diagram, method-A begins a transaction and then invokes method-B of Bean-2. When method-B executes, does it run within the scope of the transaction started by method-A, or does it execute with a new transaction? The answer depends on the transaction attribute of method-B.

A transaction attribute can have one of the following values:
- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

Required Attribute

If the client is running within a transaction and invokes the enterprise bean’s method, the method executes within the client’s transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

The Required attribute is the implicit transaction attribute for all enterprise bean methods running with container-managed transaction demarcation. You typically do not set the Required attribute unless you need to override another transaction attribute. Because transaction attributes are declarative, you can easily change them later.
**RequiresNew Attribute**

If the client is running within a transaction and invokes the enterprise bean’s method, the container takes the following steps:

1. Suspends the client's transaction
2. Starts a new transaction
3. Delegates the call to the method
4. Resumes the client's transaction after the method completes

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the RequiresNew attribute when you want to ensure that the method always runs within a new transaction.

**Mandatory Attribute**

If the client is running within a transaction and invokes the enterprise bean’s method, the method executes within the client’s transaction. If the client is not associated with a transaction, the container throws a TransactionRequiredException.

Use the Mandatory attribute if the enterprise bean’s method must use the transaction of the client.

**NotSupported Attribute**

If the client is running within a transaction and invokes the enterprise bean’s method, the container suspends the client’s transaction before invoking the method. After the method has completed, the container resumes the client’s transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the NotSupported attribute for methods that don’t need transactions. Because transactions involve overhead, this attribute may improve performance.

**Supports Attribute**

If the client is running within a transaction and invokes the enterprise bean’s method, the method executes within the client’s transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Because the transactional behavior of the method may vary, you should use the Supports attribute with caution.
Never Attribute

If the client is running within a transaction and invokes the enterprise bean’s method, the container throws a `RemoteException`. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Summary of Transaction Attributes

Table 44–1 summarizes the effects of the transaction attributes. Both the T1 and the T2 transactions are controlled by the container. A T1 transaction is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean. A T2 transaction is started by the container just before the method executes.

In the last column of Table 44–1, the word “None” means that the business method does not execute within a transaction controlled by the container. However, the database calls in such a business method might be controlled by the transaction manager of the database management system.

<table>
<thead>
<tr>
<th>Transaction Attribute</th>
<th>Client’s Transaction</th>
<th>Business Method’s Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>None</td>
<td>T2</td>
</tr>
<tr>
<td>Required</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>RequiresNew</td>
<td>None</td>
<td>T2</td>
</tr>
<tr>
<td>RequiresNew</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>Mandatory</td>
<td>None</td>
<td>Error</td>
</tr>
<tr>
<td>Mandatory</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>NotSupported</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>NotSupported</td>
<td>T1</td>
<td>None</td>
</tr>
<tr>
<td>Supports</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Supports</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>Never</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Never</td>
<td>T1</td>
<td>Error</td>
</tr>
</tbody>
</table>

Setting Transaction Attributes

Transaction attributes are specified by decorating the enterprise bean class or method with a `javax.ejb.TransactionAttribute` annotation and setting it to one of the `javax.ejb.TransactionAttributeType` constants.
If you decorate the enterprise bean class with @TransactionAttribute, the specified TransactionAttributeType is applied to all the business methods in the class. Decorating a business method with @TransactionAttribute applies the TransactionAttributeType only to that method. If a @TransactionAttribute annotation decorates both the class and the method, the method TransactionAttributeType overrides the class TransactionAttributeType.

The TransactionAttributeType constants shown in Table 44–2 encapsulate the transaction attributes described earlier in this section.

<table>
<thead>
<tr>
<th>Transaction Attribute</th>
<th>TransactionAttributeConstant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>TransactionAttributeType.REQUIRED</td>
</tr>
<tr>
<td>RequiresNew</td>
<td>TransactionAttributeType.REQUIRES_NEW</td>
</tr>
<tr>
<td>Mandatory</td>
<td>TransactionAttributeType.MANDATORY</td>
</tr>
<tr>
<td>NotSupported</td>
<td>TransactionAttributeType.NOT_SUPPORTED</td>
</tr>
<tr>
<td>Supports</td>
<td>TransactionAttributeType.SUPPORTS</td>
</tr>
<tr>
<td>Never</td>
<td>TransactionAttributeType.NEVER</td>
</tr>
</tbody>
</table>

The following code snippet demonstrates how to use the @TransactionAttribute annotation:

```java
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
    ...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}
    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}
    public void thirdMethod() {...}
    public void fourthMethod() {...}
}
```

In this example, the TransactionBean class’s transaction attribute has been set to NotSupported, firstMethod has been set to RequiresNew, and secondMethod has been set to Required. Because a @TransactionAttribute set on a method overrides the class @TransactionAttribute, calls to firstMethod will create a new transaction, and calls to secondMethod will either run in the current transaction or start a new transaction. Calls to thirdMethod or fourthMethod do not take place within a transaction.
Rolling Back a Container-Managed Transaction

There are two ways to roll back a container-managed transaction. First, if a system exception is thrown, the container will automatically roll back the transaction. Second, by invoking the `setRollbackOnly` method of the `EJBContext` interface, the bean method instructs the container to roll back the transaction. If the bean throws an application exception, the rollback is not automatic but can be initiated by a call to `setRollbackOnly`.

Synchronizing a Session Bean’s Instance Variables

The `SessionSynchronization` interface, which is optional, allows stateful session bean instances to receive transaction synchronization notifications. For example, you could synchronize the instance variables of an enterprise bean with their corresponding values in the database. The container invokes the `SessionSynchronization` methods (`afterBegin`, `beforeCompletion`, and `afterCompletion`) at each of the main stages of a transaction.

The `afterBegin` method informs the instance that a new transaction has begun. The container invokes `afterBegin` immediately before it invokes the business method.

The container invokes the `beforeCompletion` method after the business method has finished but just before the transaction commits. The `beforeCompletion` method is the last opportunity for the session bean to roll back the transaction (by calling `setRollbackOnly`).

The `afterCompletion` method indicates that the transaction has completed. This method has a single boolean parameter whose value is `true` if the transaction was committed and `false` if it was rolled back.

Methods Not Allowed in Container-Managed Transactions

You should not invoke any method that might interfere with the transaction boundaries set by the container. The list of prohibited methods follows:

- The `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
- The `getUserTransaction` method of `javax.ejb.EJBContext`
- Any method of `javax.transaction.UserTransaction`

You can, however, use these methods to set boundaries in application-managed transactions.
Bean-Managed Transactions

In bean-managed transaction demarcation, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. Although beans with container-managed transactions require less coding, they have one limitation: When a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation will make coding your bean difficult, you should consider using bean-managed transactions.

The following pseudocode illustrates the kind of fine-grained control you can obtain with application-managed transactions. By checking various conditions, the pseudocode decides whether to start or stop certain transactions within the business method:

```java
begin transaction
  ...
  update table-a
  ...
  if (condition-x)
    commit transaction
  else if (condition-y)
    update table-b
    commit transaction
  else
    rollback transaction
  begin transaction
  update table-c
  commit transaction
```

When coding an application-managed transaction for session or message-driven beans, you must decide whether to use Java Database Connectivity or JTA transactions. The sections that follow discuss both types of transactions.

JTA Transactions

JTA, or the Java Transaction API, allows you to demarcate transactions in a manner that is independent of the transaction manager implementation. GlassFish Server implements the transaction manager with the Java Transaction Service (JTS). However, your code doesn’t call the JTS methods directly but instead invokes the JTA methods, which then call the lower-level JTS routines.

A JTA transaction is controlled by the Java EE transaction manager. You may want to use a JTA transaction because it can span updates to multiple databases from different vendors. A particular DBMS’s transaction manager may not work with heterogeneous databases. However, the Java EE transaction manager does have one limitation: It does not support nested transactions. In other words, it cannot start a transaction for an instance until the preceding transaction has ended.

To demarcate a JTA transaction, you invoke the begin, commit, and rollback methods of the `javax.transaction.UserTransaction` interface.
Returning without Committing

In a stateless session bean with bean-managed transactions, a business method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction.

In a stateful session bean with a JTA transaction, the association between the bean instance and the transaction is retained across multiple client calls. Even if each business method called by the client opens and closes the database connection, the association is retained until the instance completes the transaction.

In a stateful session bean with a JDBC transaction, the JDBC connection retains the association between the bean instance and the transaction across multiple calls. If the connection is closed, the association is not retained.

Methods Not Allowed in Bean-Managed Transactions

Do not invoke the getRollbackOnly and setRollbackOnly methods of the EJBContext interface in bean-managed transactions. These methods should be used only in container-managed transactions. For bean-managed transactions, invoke the getStatus and rollback methods of the UserTransaction interface.

Transaction Timeouts

For container-managed transactions, you can use the Administration Console to configure the transaction timeout interval. See “Starting the Administration Console” on page 74.

For enterprise beans with bean-managed JTA transactions, you invoke the setTransactionTimeout method of the UserTransaction interface.

To Set a Transaction Timeout

1. In the Administration Console, expand the Configurations node, then expand the server-config node and select Transaction Service.

2. On the Transaction Service page, set the value of the Transaction Timeout field to the value of your choice (for example, 5).

   With this setting, if the transaction has not completed within 5 seconds, the EJB container rolls it back.

   The default value is 0, meaning that the transaction will not time out.
Updating Multiple Databases

The Java EE transaction manager controls all enterprise bean transactions except for bean-managed JDBC transactions. The Java EE transaction manager allows an enterprise bean to update multiple databases within a transaction. Figure 44–2 and Figure 44–3 show two scenarios for updating multiple databases in a single transaction.

In Figure 44–2, the client invokes a business method in Bean-A. The business method begins a transaction, updates Database X, updates Database Y, and invokes a business method in Bean-B. The second business method updates Database Z and returns control to the business method in Bean-A, which commits the transaction. All three database updates occur in the same transaction.

In Figure 44–3, the client calls a business method in Bean-A, which begins a transaction and updates Database X. Then Bean-A invokes a method in Bean-B, which resides in a remote Java EE server. The method in Bean-B updates Database Y. The transaction managers of the Java EE servers ensure that both databases are updated in the same transaction.

FIGURE 44–2 Updating Multiple Databases
Transactions in Web Components

You can demarcate a transaction in a web component by using either the `java.sql.Connection` or the `javax.transaction.UserTransaction` interface. These are the same interfaces that a session bean with bean-managed transactions can use. Transactions demarcated with the `UserTransaction` interface are discussed in “JTA Transactions” on page 800.

Further Information about Transactions

For more information about transactions, see

- Java Transaction API 1.1 specification:
Java EE components can access a wide variety of resources, including databases, mail sessions, Java Message Service objects, and URLs. The Java EE 6 platform provides mechanisms that allow you to access all these resources in a similar manner. This chapter explains how to get connections to several types of resources.

The following topics are addressed here:

- “Resources and JNDI Naming” on page 805
- “DataSource Objects and Connection Pools” on page 806
- “Resource Injection” on page 807
- “Resource Adapters and Contracts” on page 810
- “Metadata Annotations” on page 814
- “Common Client Interface” on page 815
- “Using Resource Adapters With Contexts and Dependency Injection for the Java EE Platform (CDI)” on page 816
- “Further Information about Resources” on page 817

Resources and JNDI Naming

In a distributed application, components need to access other components and resources, such as databases. For example, a servlet might invoke remote methods on an enterprise bean that retrieves information from a database. In the Java EE platform, the Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources.

A resource is a program object that provides connections to systems, such as database servers and messaging systems. (A Java Database Connectivity resource is sometimes referred to as a data source.) Each resource object is identified by a unique, people-friendly name, called the JNDI name. For example, the JNDI name of the JDBC resource for the Java DB database that is shipped with the GlassFish Server is jdbc/__default.
An administrator creates resources in a JNDI namespace. In the GlassFish Server, you can use either the Administration Console or the asadmin command to create resources. Applications then use annotations to inject the resources. If an application uses resource injection, the GlassFish Server invokes the JNDI API, and the application is not required to do so. However, it is also possible for an application to locate resources by making direct calls to the JNDI API.

A resource object and its JNDI name are bound together by the naming and directory service. To create a new resource, a new name/object binding is entered into the JNDI namespace. You inject resources by using the @Resource annotation in an application.

You can use a deployment descriptor to override the resource mapping that you specify in an annotation. Using a deployment descriptor allows you to change an application by repackaging it rather than by both recompiling the source files and repackaging. However, for most applications, a deployment descriptor is not necessary.

**DataSource Objects and Connection Pools**

To store, organize, and retrieve data, most applications use a relational database. Java EE 6 components may access relational databases through the JDBC API. For information on this API, see [http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html](http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html).

In the JDBC API, databases are accessed by using DataSource objects. A DataSource has a set of properties that identify and describe the real-world data source that it represents. These properties include such information as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on. In the GlassFish Server, a data source is called a JDBC resource.

Applications access a data source by using a connection, and a DataSource object can be thought of as a factory for connections to the particular data source that the DataSource instance represents. In a basic DataSource implementation, a call to the getConnection method returns a connection object that is a physical connection to the data source.

A DataSource object may be registered with a JNDI naming service. If so, an application can use the JNDI API to access that DataSource object, which can then be used to connect to the data source it represents.

DataSource objects that implement connection pooling also produce a connection to the particular data source that the DataSource class represents. The connection object that the getConnection method returns is a handle to a PooledConnection object rather than being a physical connection. An application uses the connection object in the same way that it uses a connection. Connection pooling has no effect on application code except that a pooled connection, like all connections, should always be explicitly closed. When an application closes a connection that is pooled, the connection is returned to a pool of reusable connections. The next time getConnection is called, a handle to one of these pooled connections will be returned
if one is available. Because connection pooling avoids creating a new physical connection every time one is requested, applications can run significantly faster.

A JDBC connection pool is a group of reusable connections for a particular database. Because creating each new physical connection is time consuming, the server maintains a pool of available connections to increase performance. When it requests a connection, an application obtains one from the pool. When an application closes a connection, the connection is returned to the pool.

Applications that use the Persistence API specify the DataSource object they are using in the jta-data-source element of the persistence.xml file:

```xml
<jta-data-source>jdbc/MyOrderDB</jta-data-source>
```

This is typically the only reference to a JDBC object for a persistence unit. The application code does not refer to any JDBC objects.

---

**Resource Injection**

The javax.annotation.Resource annotation is used to declare a reference to a resource; @Resource can decorate a class, a field, or a method. The container will inject the resource referred to by @Resource into the component either at runtime or when the component is initialized, depending on whether field/method injection or class injection is used. With field-based and method-based injection, the container will inject the resource when the application is initialized. For class-based injection, the resource is looked up by the application at runtime.

The @Resource annotation has the following elements:

- **name**: The JNDI name of the resource
- **type**: The Java language type of the resource
- **authenticationType**: The authentication type to use for the resource
- **shareable**: Indicates whether the resource can be shared
- **mappedName**: A nonportable, implementation-specific name to which the resource should be mapped
- **description**: The description of the resource

The name element is the JNDI name of the resource and is optional for field-based and method-based injection. For field-based injection, the default name is the field name qualified by the class name. For method-based injection, the default name is the JavaBeans property name, based on the method qualified by the class name. The name element must be specified for class-based injection.
The type of resource is determined by one of the following:

- The type of the field the @Resource annotation is decorating for field-based injection
- The type of the JavaBeans property the @Resource annotation is decorating for method-based injection
- The type element of @Resource

For class-based injection, the type element is required.

The authenticationType element is used only for connection factory resources, such as the resources of a connector, also called the resource adapter, or data source. This element can be set to one of the javax.annotation.Resource.AuthenticationType enumerated type values: CONTAINER, the default, and APPLICATION.

The shareable element is used only for Object Resource Broker (ORB) instance resources or connection factory resource. This element indicates whether the resource can be shared between this component and other components and may be set to true, the default, or false.

The mappedName element is a nonportable, implementation-specific name to which the resource should be mapped. Because the name element, when specified or defaulted, is local only to the application, many Java EE servers provide a way of referring to resources across the application server. This is done by setting the mappedName element. Use of the mappedName element is nonportable across Java EE server implementations.

The description element is the description of the resource, typically in the default language of the system on which the application is deployed. This element is used to help identify resources and to help application developers choose the correct resource.

### Field-Based Injection

To use field-based resource injection, declare a field and decorate it with the @Resource annotation. The container will infer the name and type of the resource if the name and type elements are not specified. If you do specify the type element, it must match the field’s type declaration.

In the following code, the container infers the name of the resource, based on the class name and the field name: com.example.SomeClass/myDB. The inferred type is javax.sql.DataSource.class:

```java
package com.example;

public class SomeClass {
    @Resource
    private javax.sql.DataSource myDB;
    ...
}
```
In the following code, the JNDI name is customerDB, and the inferred type is `javax.sql.DataSource.class`:

```java
package com.example;

public class SomeClass {
    @Resource(name="customerDB")
    private javax.sql.DataSource myDB;
    ...
}
```

### Method-Based Injection

To use method-based injection, declare a setter method and decorate it with the `@Resource` annotation. The container will infer the name and type of the resource if the name and type elements are not specified. The setter method must follow the JavaBeans conventions for property names: The method name must begin with `set`, have a `void` return type, and only one parameter. If you do specify the type element, it must match the field's type declaration.

In the following code, the container infers the name of the resource based on the class name and the field name: `com.example.SomeClass/myDB`. The inferred type is `javax.sql.DataSource.class`:

```java
package com.example;

public class SomeClass {
    private javax.sql.DataSource myDB;
    ...
    @Resource
    private void setMyDB(javax.sql.DataSource ds) {
        myDB = ds;
    }
    ...
}
```

In the following code, the JNDI name is customerDB, and the inferred type is `javax.sql.DataSource.class`:

```java
package com.example;

public class SomeClass {
    private javax.sql.DataSource myDB;
    ...
    @Resource(name="customerDB")
    private void setMyDB(javax.sql.DataSource ds) {
        myDB = ds;
    }
    ...
}
```
Class-Based Injection

To use class-based injection, decorate the class with a @Resource annotation, and set the required name and type elements:

```java
@Resource(name="myMessageQueue", type="javax.jms.ConnectionFactory")
public class SomeMessageBean {
    ...
}
```

The @Resources annotation is used to group together multiple @Resource declarations for class-based injection. The following code shows the @Resources annotation containing two @Resource declarations. One is a Java Message Service message queue, and the other is a JavaMail session:

```java
@Resources({
    @Resource(name="myMessageQueue", type="javax.jms.ConnectionFactory"),
    @Resource(name="myMailSession", type="javax.mail.Session")
})
public class SomeMessageBean {
    ...
}
```

Resource Adapters and Contracts

A resource adapter is a Java EE component that implements the Java EE Connector architecture for a specific EIS. Examples of EISs include enterprise resource planning, mainframe transaction processing, and database systems. In a Java EE server, the Java Message Server and JavaMail also act as EISs that you access using resource adapters. As illustrated in Figure 45–1, the resource adapter facilitates communication between a Java EE application and an EIS.
Stored in a Resource Adapter Archive (RAR) file, a resource adapter can be deployed on any Java EE server, much like a Java EE application. A RAR file may be contained in an Enterprise Archive (EAR) file, or it may exist as a separate file.

A resource adapter is analogous to a JDBC driver. Both provide a standard API through which an application can access a resource that is outside the Java EE server. For a resource adapter, the target system is an EIS; for a JDBC driver, it is a DBMS. Resource adapters and JDBC drivers are rarely created by application developers. In most cases, both types of software are built by vendors that sell tools, servers, or integration software.

The resource adapter mediates communication between the Java EE server and the EIS by means of contracts. The application contract defines the API through which a Java EE component, such as an enterprise bean, accesses the EIS. This API is the only view that the component has of the EIS. The system contracts link the resource adapter to important services that are managed by the Java EE server. The resource adapter itself and its system contracts are transparent to the Java EE component.

**Management Contracts**

The Java EE Connector Architecture defines system contracts that enable resource adapter lifecycle and thread management.
Lifecycle Management

The Connector Architecture specifies a lifecycle management contract that allows an application server to manage the lifecycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during the deployment or application server startup. This contract also provides a means for the application server to notify the resource adapter instance when it is undeployed or when an orderly shutdown of the application server takes place.

Work Management Contract

The Connector Architecture work management contract ensures that resource adapters use threads in the proper, recommended manner. This contract also enables an application server to manage threads for resource adapters.

Resource adapters that improperly use threads can jeopardize the entire application server environment. For example, a resource adapter might create too many threads or might not properly release threads it has created. Poor thread handling inhibits application server shutdown and impacts the application server’s performance because creating and destroying threads are expensive operations.

The work management contract establishes a means for the application server to pool and reuse threads, similar to pooling and reusing connections. By adhering to this contract, the resource adapter does not have to manage threads itself. Instead, the resource adapter has the application server create and provide needed threads. When it is finished with a given thread, the resource adapter returns the thread to the application server. The application server manages the thread, either returning it to a pool for later reuse or destroying it. Handling threads in this manner results in increased application server performance and more efficient use of resources.

In addition to moving thread management to the application server, the Connector Architecture provides a flexible model for a resource adapter that uses threads.

- The requesting thread can choose to block (stop its own execution) until the work thread completes.
- The requesting thread can block while it waits to get the work thread. When the application server provides a work thread, the requesting thread and the work thread execute in parallel.
- The resource adapter can opt to submit the work for the thread to a queue. The thread executes the work from the queue at some later point. The resource adapter continues its own execution from the point it submitted the work to the queue, no matter when the thread executes it.

With the latter two approaches, the submitting thread and the work thread may execute simultaneously or independently. For these approaches, the contract specifies a listener mechanism to notify the resource adapter that the thread has completed its operation. The resource adapter can also specify the execution context for the thread, and the work management contract controls the context in which the thread executes.
Generic Work Context Contract

The work management contract between the application server and a resource adapter enables a resource adapter to do a task, such as communicating with the EIS or delivering messages, by delivering Work instances for execution.

A generic work context contract enables a resource adapter to control the contexts in which the Work instances that it submits are executed by the application server’s WorkManager. A generic work context mechanism also enables an application server to support new message inflow and delivery schemes. It also provides a richer contextual Work execution environment to the resource adapter while still maintaining control over concurrent behavior in a managed environment.

The generic work context contract standardizes the transaction context and the security context.

Outbound and Inbound Contracts

The Connector Architecture defines the following outbound contracts, system-level contracts between an application server and an EIS that enable outbound connectivity to an EIS.

- The connection management contract supports connection pooling, a technique that enhances application performance and scalability. Connection pooling is transparent to the application, which simply obtains a connection to the EIS.
- The transaction management contract extends the connection management contract and provides support for management of both local and XA transactions. A local transaction is limited in scope to a single EIS system, and the EIS resource manager itself manages such transaction. An XA transaction or global transaction can span multiple resource managers. This form of transaction requires transaction coordination by an external transaction manager, typically bundled with an application server. A transaction manager uses a two-phase commit protocol to manage a transaction that spans multiple resource managers or EISs, and uses one-phase commit optimization if only one resource manager is participating in an XA transaction.
- The security management contract provides mechanisms for authentication, authorization, and secure communication between a Java EE server and an EIS to protect the information in the EIS. A work security map matches EIS identities to the application server domain’s identities.

Inbound contracts are system contracts between a Java EE server and an EIS that enable inbound connectivity from the EIS: pluggability contracts for message providers and contracts for importing transactions.
Java EE Connector Architecture 1.6 introduces a set of annotations to minimize the need for deployment descriptors.

- The @Connector annotation can be used by the resource adapter developer to specify that the JavaBeans component is a resource adapter JavaBeans component. This annotation is used for providing metadata about the capabilities of the resource adapter. Optionally, you can provide a JavaBeans component implementing the ResourceAdapter interface, as in the following example:

```java
@Connector(
    description = "Sample adapter using the JavaMail API",
    displayName = "InboundResourceAdapter",
    vendorName = "My Company, Inc.",
    eisType = "MAIL",
    version = "1.0"
)
public class ResourceAdapterImpl
    implements ResourceAdapter, java.io.Serializable {
    ...
    ...
}
```

- The @ConnectionDefinition annotation defines a set of connection interfaces and classes pertaining to a particular connection type, as in the following example:

```java
@ConnectionDefinition(
    connectionFactory = JavaMailConnectionFactory.class,
    connectionFactoryImpl = JavaMailConnectionFactoryImpl.class,
    connection = JavaMailConnection.class,
    connectionImpl = JavaMailConnectionImpl.class
)
public class ManagedConnectionFactoryImpl implements
    ManagedConnectionFactory, Serializable {
    ...
    ...
}
```

- The @AdministeredObject annotation designates a JavaBeans component as an administered object.

- The @Activation annotation contains configuration information pertaining to inbound connectivity from an EIS instance, as in the following example:

```java
@Activation(
    messageListeners = {JavaMailMessageListener.class}
)
public class ActivationSpecImpl
    implements ActivationSpec, Serializable {
    ...
    @ConfigProperty()
    // serverName property value
    private String serverName = "";
    
    @ConfigProperty()
    // userName property value
    private String userName = "";
```
The `@ConfigProperty` annotation can be used on JavaBeans components to provide additional configuration information that may be used by the deployer and resource adapter provider. The preceding example code shows several `@ConfigProperty` annotations.

The specification allows a resource adapter to be developed in mixed-mode form, that is the ability for a resource adapter developer to use both metadata annotations and deployment descriptors in applications. An application assembler or deployer may use the deployment descriptor to override the metadata annotations specified by the resource adapter developer.

The deployment descriptor for a resource adapter, if present, is named `ra.xml`. The `metadata-complete` attribute defines whether the deployment descriptor for the resource adapter module is complete or whether the class files available to the module and packaged with the resource adapter need to be examined for annotations that specify deployment information.

For the complete list of annotations and JavaBeans components introduced in the Java EE 6 platform, see the Java EE Connector architecture 1.6 specification.

## Common Client Interface

This section explains how components use the Connector Architecture Common Client Interface (CCI) API and a resource adapter to access data from an EIS. The CCI API defines a set of interfaces and classes whose methods allow a client to perform typical data access operations. The CCI interfaces and classes are as follows:

- **ConnectionFactory**: Provides an application component with a `Connection` instance to an EIS.
- **Connection**: Represents the connection to the underlying EIS.
- **ConnectionSpec**: Provides a means for an application component to pass connection-request-specific properties to the `ConnectionFactory` when making a connection request.
- **Interaction**: Provides a means for an application component to execute EIS functions, such as database stored procedures.
InteractionSpec: Holds properties pertaining to an application component's interaction with an EIS.

Record: The superinterface for the various kinds of record instances. Record instances can be MappedRecord, IndexedRecord, or ResultSet instances, all of which inherit from the Record interface.

RecordFactory: Provides an application component with a Record instance.

IndexedRecord: Represents an ordered collection of Record instances based on the java.util.List interface.

A client or application component that uses the CCI to interact with an underlying EIS does so in a prescribed manner. The component must establish a connection to the EIS's resource manager, and it does so using the ConnectionFactory. The Connection object represents the connection to the EIS and is used for subsequent interactions with the EIS.

The component performs its interactions with the EIS, such as accessing data from a specific table, using an Interaction object. The application component defines the Interaction object by using an InteractionSpec object. When it reads data from the EIS, such as from database tables, or writes to those tables, the application component does so by using a particular type of Record instance: a MappedRecord, an IndexedRecord, or a ResultSet instance.

Note, too, that a client application that relies on a CCI resource adapter is very much like any other Java EE client that uses enterprise bean methods.

Using Resource Adapters With Contexts and Dependency Injection for the Java EE Platform (CDI)

To enable a resource adapter for CDI, provide a beans.xml file in the META-INF directory of the packaged archive of the resource adapter. For more information about beans.xml, see “Configuring a CDI Application” on page 525.

All classes in the resource adapter are available for injection. All classes in the resource adapter can be CDI managed beans except for the following classes:

- **Resource adapter beans**: These beans are classes that are annotated with the javax.resource.spi.Connector annotation or are declared as corresponding elements in the resource adapter deployment descriptor, ra.xml.

- **Managed connection factory beans**: These beans are classes that are annotated with the javax.resource.spi.ConnectorDefinition annotation or the javax.resource.spi.ConnectorDefinitions annotation or are declared as corresponding elements in ra.xml.

- **Activation specification beans**: These beans are classes that are annotated with the javax.resource.spi.Activation annotation or are declared as corresponding elements in ra.xml.
Administered object beans: These beans are classes that are annotated with the javax.resource.spi.AdministeredObject annotation or are declared as corresponding elements in ra.xml.

Further Information about Resources

For more information about resources and annotations, see

- Java EE 6 Platform Specification (JSR 316):
- Java EE Connector architecture 1.6 specification:
- EJB 3.1 specification:
  http://jcp.org/en/jsr/detail?id=318
- Common Annotations for the Java Platform:
  http://www.jcp.org/en/jsr/detail?id=250
The Resource Adapter Example

The mailconnector example shows how you can use a resource adapter, a message-driven bean (MDB), and JavaServer Faces technology to create an application that can send email messages and browse for messages. This example uses a sample implementation of the JavaMail API called mock-javamail. The resource adapter is deployed separately, while the MDB and the web application are packaged in an EAR file.

The following topics are addressed here:
- “The Resource Adapter” on page 819
- “The Message-Driven Bean” on page 820
- “The Web Application” on page 820
- “Running the mailconnector Example” on page 820

The Resource Adapter

The mailconnector resource adapter enables the MDB to receive email messages that are delivered to a specific mailbox folder on a mail server. It also provides connection factory objects clients can use to obtain connection objects that allow them to synchronously query email servers for new messages in a specific mailbox folder.

In this example, the MDB activates the resource adapter, but it does not receive email messages. Instead, this example allows users to synchronously query an email server for new messages.

The components of the resource adapter are as follows:
- mailconnector.ra: Base class of the mailconnector resource adapter
- mailconnector.ra/inbound: Classes that implement the inbound resource adapter, which supports delivery of JavaMail messages to MDBs
- mailconnector.ra/outbound: Classes that implement the outbound resource adapter, which supports synchronous queries to email servers
mailconnector.api: Interfaces that are implemented by MDBs associated with this resource adapter and by the Connection andConnectionFactory interfaces provided by the outbound resource adapter

mailconnector.share: JavaBeans class that implements the ConnectionSpec interface, allowing properties to be passed to the outbound resource adapter

When the resource adapter is deployed, it uses the Work Management facilities available to resource adapters to start a thread that monitors mailbox folders for new messages. The polling thread of the resource adapter monitors the mailbox folders every 30 seconds for new messages.

The Message-Driven Bean

The mailconnector message-driven bean, JavaMailMessageBean, activates the resource adapter. When an MDB is deployed, the application server passes the MDB's activation config properties (commented out in this case) to the mailconnector resource adapter, which forwards it to the polling thread. When the MDB is undeployed, the application server notifies the resource adapter, which notifies the polling thread to stop monitoring the mail folder associated with the MDB being undeployed.

The MDB is packaged in an EJB JAR file.

The Web Application

The web application in the mailconnector example contains an HTML page (index.html), Facelets pages, and managed beans that let you log in, send email messages to a mailbox folder, and query for new messages in a mail folder using the connection interfaces provided by the mailconnector resource adapter.

The application protects the Facelets pages by using form-based authentication, specified through a security constraint in the web.xml file.

The web application is packaged in a WAR file.

Running the mailconnector Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the mailconnector example.

Before You Deploy the mailconnector Example

Before you deploy the mailconnector application, perform the following steps.
1. Download `mock-javamail-1.9.jar` from http://download.java.net/maven/2/org/jvnet/mock-javamail/mock-javamail/1.9/.

2. Copy this JAR file to the directory `as-install/lib`.

3. Restart GlassFish Server.


5. In the Administration Console, expand the Configurations node, then expand the `server-config` node.


7. Select the Default Principal to Role Mapping Enabled check box.

8. Click Save.

▼ To Build, Package, and Deploy the `mailconnector` Example Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to: `tut-install/examples/connectors/mailconnector/`.

3. Select the `mailconnector-ra` folder and click Open Project.

4. In the Projects tab, right-click the `mailconnector-ra` project and select Build.
   This command builds the resource adapter. It also places identical files named `mailconnector.rar` and `mailconnector.jar` in the `mailconnector` directory.

5. In the Projects tab, right-click the `mailconnector-ra` project and select Deploy.

6. From the File menu, choose Open Project.

7. In the Open Project dialog, navigate to: `tut-install/examples/connectors/mailconnector/`.

8. Select the `mailconnector-ear` folder.
9. Select the Open Required Projects check box and click Open Project.

10. In the Projects tab, right-click the mailconnector-ear project and select Build.

11. In a terminal window, navigate to:
   tut-install/examples/connectors/mailconnector/mailconnector-ear/

12. Enter the following command to create the resources and users:
   ```
   ant setup
   ```

13. In NetBeans IDE, in the Projects tab, right-click the mailconnector-ear project and select Deploy.

▼ To Build, Package, and Deploy the mailconnector Example Using Ant

1. In a terminal window, go to:
   ```
   tut-install/examples/connectors/mailconnector/mailconnector-ear/
   ```

2. Enter the following command:
   ```
   ant all
   ```
   This command builds and deploys the mailconnector-ra RAR file, sets up users and resources, then builds and deploys the mailconnector-ear EAR file. It also places identical files named mailconnector.rar and mailconnector.jar in the mailconnector directory.

▼ To Run the mailconnector Example

1. In a web browser, navigate to the following URL:
   ```
   http://localhost:8080/mailconnector-war/
   ```

2. Log in with a user name of either user1, user2, user3, or user4. The password is the same as the user name.

   You can send messages and browse for the messages you sent. The messages you sent are available 30 seconds after you sent them.

   For example, you can log in as user1 and send a message to user4, then log in as user4 and query for messages. In the form for browsing messages, verify that the fields are correct, then click Browse.

   View the server log to follow the flow of the application. Most classes and methods specify logging information that makes the sequence of events easy to follow.
3 Before you undeploy the application, in a terminal window, navigate to
	\texttt{tut-install/examples/connectors/mailconnector/mailconnector-ear/} and enter the
following command to remove the resources and users:
\texttt{ant takedown}

You cannot undeploy the resource adapter until you run this command.

Next Steps  When you clean the application, you can also remove the \texttt{mailconnector.rar} and
\texttt{mailconnector.jar} files from the \texttt{mailconnector} directory.

Remove the \texttt{mock-javamail-1.9.jar} file from the \texttt{as-install/lib} directory if you might run any
other applications that use the JavaMail API (for example, “The async Example Application” on
page 508).
This chapter provides an introduction to the Java Message Service (JMS) API, a Java API that allows applications to create, send, receive, and read messages using reliable, asynchronous, loosely coupled communication. It covers the following topics:

- “Overview of the JMS API” on page 825
- “Basic JMS API Concepts” on page 828
- “The JMS API Programming Model” on page 832
- “Creating Robust JMS Applications” on page 842
- “Using the JMS API in Java EE Applications” on page 851
- “Further Information about JMS” on page 858

Overview of the JMS API

This overview defines the concept of messaging, describes the JMS API and when it can be used, and explains how the JMS API works within the Java EE platform.

What Is Messaging?

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is loosely coupled. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which message format and which destination to use. In this respect, messaging differs from
tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application’s methods.

Messaging also differs from electronic mail (email), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

What Is the JMS API?

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.

The JMS API enables communication that is not only loosely coupled but also:

- **Asynchronous**: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
- **Reliable**: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The current version of the JMS specification is Version 1.1. You can download a copy of the specification from the JMS web site: [http://www.oracle.com/technetwork/java/index-jsp-142945.html](http://www.oracle.com/technetwork/java/index-jsp-142945.html).

When Can You Use the JMS API?

An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as a remote procedure call (RPC), under the following circumstances:

- The provider wants the components not to depend on information about other components’ interfaces, so components can be easily replaced.
- The provider wants the application to run whether or not all components are up and running simultaneously.
- The application business model allows a component to send information to another and to continue to operate without receiving an immediate response.
For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like these:

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so the factory can make more cars.
- The factory component can send a message to the parts components so the factory can assemble the parts it needs.
- The parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers.
- Both the factory and the parts components can send messages to the accounting component to update budget numbers.
- The business can publish updated catalog items to its sales force.

Using messaging for these tasks allows the various components to interact with one another efficiently, without tying up network or other resources. Figure 47–1 illustrates how this simple example might work.

Manufacturing is only one example of how an enterprise can use the JMS API. Retail applications, financial services applications, health services applications, and many others can make use of messaging.

**How Does the JMS API Work with the Java EE Platform?**

When the JMS API was introduced in 1998, its most important purpose was to allow Java applications to access existing messaging-oriented middleware (MOM) systems, such as MQSeries from IBM. Since that time, many vendors have adopted and implemented the JMS API, so a JMS product can now provide a complete messaging capability for an enterprise.
Beginning with the 1.3 release of the Java EE platform, the JMS API has been an integral part of the platform, and application developers have been able to use messaging with Java EE components.

The JMS API in the Java EE platform has the following features.

- Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously. (Applets, however, are not required to support the JMS API.)

- Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages. A JMS provider can optionally implement concurrent processing of messages by message-driven beans.

- Message send and receive operations can participate in distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.

The JMS API enhances the Java EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among Java EE components and legacy systems capable of messaging. A developer can easily add new behavior to a Java EE application that has existing business events by adding a new message-driven bean to operate on specific business events. The Java EE platform, moreover, enhances the JMS API by providing support for distributed transactions and allowing for the concurrent consumption of messages. For more information, see the Enterprise JavaBeans specification, v3.1.

The JMS provider can be integrated with the application server using the Java EE Connector architecture. You access the JMS provider through a resource adapter. This capability allows vendors to create JMS providers that can be plugged in to multiple application servers, and it allows application servers to support multiple JMS providers. For more information, see the Java EE Connector architecture specification, v1.6.

**Basic JMS API Concepts**

This section introduces the most basic JMS API concepts, the ones you must know to get started writing simple application clients that use the JMS API.

The next section introduces the JMS API programming model. Later sections cover more advanced concepts, including the ones you need in order to write applications that use message-driven beans.
JMS API Architecture

A JMS application is composed of the following parts.

- **A JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the Java EE platform includes a JMS provider.

- **JMS clients** are the programs or components, written in the Java programming language, that produce and consume messages. Any Java EE application component can act as a JMS client.

- **Messages** are the objects that communicate information between JMS clients.

- **Administered objects** are preconfigured JMS objects created by an administrator for the use of clients. The two kinds of JMS administered objects are destinations and connection factories, described in “JMS Administered Objects” on page 833.

Figure 47–2 illustrates the way these parts interact. Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

**FIGURE 47–2  JMS API Architecture**

![Diagram of JMS API Architecture]

Messing Domains

Before the JMS API existed, most messaging products supported either the point-to-point or the publish/subscribe approach to messaging. The JMS specification provides a separate domain for each approach and defines compliance for each domain. A stand-alone JMS provider can implement one or both domains. A Java EE provider must implement both domains.
In fact, most implementations of the JMS API support both the point-to-point and the publish/subscribe domains, and some JMS clients combine the use of both domains in a single application. In this way, the JMS API has extended the power and flexibility of messaging products.

The JMS specification goes one step further: It provides common interfaces that enable you to use the JMS API in a way that is not specific to either domain. The following subsections describe the two messaging domains and the use of the common interfaces.

**Point-to-Point Messaging Domain**

A *point-to-point* (PTP) product or application is built on the concept of message *queues*, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or expire.

PTP messaging, illustrated in Figure 47–3, has the following characteristics:

- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
- The receiver acknowledges the successful processing of a message.

![Point-to-Point Messaging](image)

Use PTP messaging when every message you send must be processed successfully by one consumer.

**Publish/Subscribe Messaging Domain**

In a *publish/subscribe* (pub/sub) product or application, clients address messages to a *topic*, which functions somewhat like a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system
takes care of distributing the messages arriving from a topic’s multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

Pub/sub messaging has the following characteristics.

- Each message can have multiple consumers.
- Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The JMS API relaxes this timing dependency to some extent by allowing subscribers to create **durable subscriptions**, which receive messages sent while the subscribers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients. For more information about durable subscriptions, see “Creating Durable Subscriptions” on page 847.

Use pub/sub messaging when each message can be processed by any number of consumers (or none). Figure 47–4 illustrates pub/sub messaging.

---

**Programming with the Common Interfaces**

Version 1.1 of the JMS API allows you to use the same code to send and receive messages under either the PTP or the pub/sub domain. The destinations you use remain domain-specific, and the behavior of the application will depend in part on whether you are using a queue or a topic. However, the code itself can be common to both domains, making your applications flexible and reusable. This tutorial describes and illustrates these common interfaces.
Message Consumption

Messaging products are inherently asynchronous: There is no fundamental timing dependency between the production and the consumption of a message. However, the JMS specification uses this term in a more precise sense. Messages can be consumed in either of two ways:

- **Synchronously**: A subscriber or a receiver explicitly fetches the message from the destination by calling the `receive` method. The `receive` method can block until a message arrives or can time out if a message does not arrive within a specified time limit.

- **Asynchronously**: A client can register a `message listener` with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage` method, which acts on the contents of the message.

The JMS API Programming Model

The basic building blocks of a JMS application are:

- Administered objects: connection factories and destinations
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

Figure 47–5 shows how all these objects fit together in a JMS client application.
This section describes all these objects briefly and provides sample commands and code snippets that show how to create and use the objects. The last subsection briefly describes JMS API exception handling.

Examples that show how to combine all these objects in applications appear in later sections. For more detail, see the JMS API documentation, part of the Java EE API documentation.

**JMS Administered Objects**

Two parts of a JMS application, destinations and connection factories, are best maintained administratively rather than programatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API. Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then access them by using resource injection.
With GlassFish Server, you can use the `asadmin create-jms-resource` command or the Administration Console to create JMS administered objects in the form of connector resources. You can also specify the resources in a file named `glassfish-resources.xml` that you can bundle with an application.

NetBeans IDE provides a wizard that allows you to create JMS resources for GlassFish Server. See “To Create JMS Resources Using NetBeans IDE” on page 863 for details.

**JMS Connection Factories**

A *connection factory* is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interface. To learn how to create connection factories, see “To Create JMS Resources Using NetBeans IDE” on page 863.

At the beginning of a JMS client program, you usually inject a connection factory resource into a `ConnectionFactory` object. For example, the following code fragment specifies a resource whose JNDI name is `jms/ConnectionFactory` and assigns it to a `ConnectionFactory` object:

```java
@Resource(lookup = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

In a Java EE application, JMS administered objects are normally placed in the `jms` naming subcontext.

**JMS Destinations**

A *destination* is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called queues. In the pub/sub messaging domain, destinations are called topics. A JMS application can use multiple queues or topics (or both). To learn how to create destination resources, see “To Create JMS Resources Using NetBeans IDE” on page 863.

To create a destination using the GlassFish Server, you create a JMS destination resource that specifies a JNDI name for the destination.

In the GlassFish Server implementation of JMS, each destination resource refers to a physical destination. You can create a physical destination explicitly, but if you do not, the Application Server creates it when it is needed and deletes it when you delete the destination resource.

In addition to injecting a connection factory resource into a client program, you usually inject a destination resource. Unlike connection factories, destinations are specific to one domain or the other. To create an application that allows you to use the same code for both topics and queues, you assign the destination to a `Destination` object.

The following code specifies two resources, a queue and a topic. The resource names are mapped to destination resources created in the JNDI namespace.
With the common interfaces, you can mix or match connection factories and destinations. That is, in addition to using theConnectionFactory interface, you can inject a QueueConnectionFactory resource and use it with a Topic, and you can inject a TopicConnectionFactory resource and use it with a Queue. The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

**JMS Connections**

A connection encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

**Note** – In the Java EE platform, the ability to create multiple sessions from a single connection is limited to application clients. In web and enterprise bean components, a connection can create no more than one session.

Connections implement the Connection interface. When you have a ConnectionFactory object, you can use it to create a Connection:

```java
Connection connection = connectionFactory.createConnection();
```

Before an application completes, you must close any connections you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

```java
connection.close();
```

Before your application can consume messages, you must call the connection's start method; for details, see "JMS Message Consumers" on page 837. If you want to stop message delivery temporarily without closing the connection, you call the stop method.

**JMS Sessions**

A session is a single-threaded context for producing and consuming messages. You use sessions to create the following:

- Message producers
- Message consumers
Messages
Queue browsers
Temporary queues and topics (see “Creating Temporary Destinations” on page 846)

Sessions serialize the execution of message listeners; for details, see “JMS Message Listeners” on page 838.

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work. For details, see “Using JMS API Local Transactions” on page 849.

Sessions implement the Session interface. After you create a Connection object, you use it to create a Session:

```java
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

The first argument means the session is not transacted; the second means the session automatically acknowledges messages when they have been received successfully. (For more information, see “Controlling Message Acknowledgment” on page 843.)

To create a transacted session, use the following code:

```java
Session session = connection.createSession(true, 0);
```

Here, the first argument means the session is transacted; the second indicates that message acknowledgment is not specified for transacted sessions. For more information on transactions, see “Using JMS API Local Transactions” on page 849. For information about the way JMS transactions work in Java EE applications, see “Using the JMS API in Java EE Applications” on page 851.

### JMS Message Producers

A message producer is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface.

You use a Session to create a MessageProducer for a destination. The following examples show that you can create a producer for a Destination object, a Queue object, or a Topic object.

```java
MessageProducer producer = session.createProducer(dest);
MessageProducer producer = session.createProducer(queue);
MessageProducer producer = session.createProducer(topic);
```

You can create an unidentified producer by specifying null as the argument to createProducer. With an unidentified producer, you do not specify a destination until you send a message.

After you have created a message producer, you can use it to send messages by using the send method:
producer.send(message);

You must first create the messages; see "JMS Messages" on page 839.

If you have created an unidentified producer, use an overloaded send method that specifies the destination as the first parameter. For example:

```java
MessageProducer anon_prod = session.createProducer(null);
anon_prod.send(dest, message);
```

### JMS Message Consumers

A *message consumer* is an object that is created by a session and used for receiving messages sent to a destination. It implements the `MessageConsumer` interface.

A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

For example, you could use a `Session` to create a `MessageConsumer` for a `Destination` object, a `Queue` object, or a `Topic` object:

```java
MessageConsumer consumer = session.createConsumer(dest);
MessageConsumer consumer = session.createConsumer(queue);
MessageConsumer consumer = session.createConsumer(topic);
```

You use the `Session.createDurableSubscriber` method to create a durable topic subscriber. This method is valid only if you are using a topic. For details, see “Creating Durable Subscriptions” on page 847.

After you have created a message consumer it becomes active, and you can use it to receive messages. You can use the `close` method for a `MessageConsumer` to make the message consumer inactive. Message delivery does not begin until you start the connection you created by calling its `start` method. (Remember always to call the `start` method; forgetting to start the connection is one of the most common JMS programming errors.)
You use the `receive` method to consume a message synchronously. You can use this method at any time after you call the `start` method:

```java
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000); // time out after a second
```

To consume a message asynchronously, you use a message listener, as described in the next section.

**JMS Message Listeners**

A message listener is an object that acts as an asynchronous event handler for messages. This object implements the `MessageListener` interface, which contains one method, `onMessage`. In the `onMessage` method, you define the actions to be taken when a message arrives.

You register the message listener with a specific `MessageConsumer` by using the `setMessageListener` method. For example, if you define a class named `Listener` that implements the `MessageListener` interface, you can register the message listener as follows:

```java
Listener myListener = newListener();
consumer.setMessageListener(myListener);
```

**Note** – In the Java EE platform, a `MessageListener` can be used only in an application client, not in a web component or enterprise bean.

After you register the message listener, you call the `start` method on the `Connection` to begin message delivery. (If you call `start` before you register the message listener, you are likely to miss messages.)

When message delivery begins, the JMS provider automatically calls the message listener’s `onMessage` method whenever a message is delivered. The `onMessage` method takes one argument of type `Message`, which your implementation of the method can cast to any of the other message types (see “Message Bodies” on page 840).

A message listener is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on the type of destination for which the message consumer was created. A message listener does, however, usually expect a specific message type and format.

Your `onMessage` method should handle all exceptions. It must not throw checked exceptions, and throwing a `RuntimeException` is considered a programming error.

The session used to create the message consumer serializes the execution of all message listeners registered with the session. At any time, only one of the session’s message listeners is running.
In the Java EE platform, a message-driven bean is a special kind of message listener. For details, see “Using Message-Driven Beans to Receive Messages Asynchronously” on page 853.

**JMS Message Selectors**

If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages that interest it. Message selectors assign the work of filtering messages to the JMS provider rather than to the application. For an example of an application that uses a message selector, see “An Application That Uses the JMS API with a Session Bean” on page 897.

A message selector is a String that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The message selector in the example selects any message that has a NewsType property that is set to the value 'Sports' or 'Opinion':

```
NewsType = 'Sports' OR NewsType = 'Opinion'
```

The createConsumer and createDurableSubscriber methods allow you to specify a message selector as an argument when you create a message consumer.

The message consumer then receives only messages whose headers and properties match the selector. (See “Message Headers” on page 839, and “Message Properties” on page 840.) A message selector cannot select messages on the basis of the content of the message body.

**JMS Messages**

The ultimate purpose of a JMS application is to produce and consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message can have three parts: a header, properties, and a body. Only the header is required. The following sections describe these parts.

For complete documentation of message headers, properties, and bodies, see the documentation of the Message interface in the API documentation.

**Message Headers**

A JMS message header contains a number of predefined fields that contain values used by both clients and providers to identify and route messages. Table 47–1 lists the JMS message header fields and indicates how their values are set. For example, every message has a unique identifier, which is represented in the header field JMSMessageID. The value of another header field, JMSDestination, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.
Each header field has associated setter and getter methods, which are documented in the description of the Message interface. Some header fields are intended to be set by a client, but many are set automatically by the send or the publish method, which overrides any client-set values.

**TABLE 47–1** How JMS Message Header Field Values Are Set

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Set By</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDestination</td>
<td>send or publish method</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>send or publish method</td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>send or publish method</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>send or publish method</td>
</tr>
<tr>
<td>JMSMessageID</td>
<td>send or publish method</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>send or publish method</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>Client</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>Client</td>
</tr>
<tr>
<td>JMSType</td>
<td>Client</td>
</tr>
<tr>
<td>JMSRedelivered</td>
<td>JMS provider</td>
</tr>
</tbody>
</table>

**Message Properties**

You can create and set properties for messages if you need values in addition to those provided by the header fields. You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors (see "JMS Message Selectors" on page 839). For an example of setting a property to be used as a message selector, see "An Application That Uses the JMS API with a Session Bean" on page 897.

The JMS API provides some predefined property names that a provider can support. The use of these predefined properties or of user-defined properties is optional.

**Message Bodies**

The JMS API defines five message body formats, also called message types, which allow you to send and receive data in many different forms and which provide compatibility with existing messaging formats. **Table 47–2** describes these message types.

**TABLE 47–2** JMS Message Types

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Body Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextMessage</td>
<td>A java.lang.String object (for example, the contents of an XML file).</td>
</tr>
</tbody>
</table>
TABLE 47–2  JMS Message Types  (Continued)

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Body Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapMessage</td>
<td>A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.</td>
</tr>
<tr>
<td>BytesMessage</td>
<td>A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.</td>
</tr>
<tr>
<td>StreamMessage</td>
<td>A stream of primitive values in the Java programming language, filled and read sequentially.</td>
</tr>
<tr>
<td>ObjectMessage</td>
<td>A Serializable object in the Java programming language.</td>
</tr>
<tr>
<td>Message</td>
<td>Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.</td>
</tr>
</tbody>
</table>

The JMS API provides methods for creating messages of each type and for filling in their contents. For example, to create and send a TextMessage, you might use the following statements:

```java
TextMessage message = session.createTextMessage();
message.setText(msg_text);  // msg_text is a String
producer.send(message);
```

At the consuming end, a message arrives as a generic Message object and must be cast to the appropriate message type. You can use one or more getter methods to extract the message contents. The following code fragment uses the getText method:

```java
Message m = consumer.receive();
if (m instanceof TextMessage) {
   TextMessage message = (TextMessage) m;
   System.out.println("Reading message: " + message.getText());
} else {
   // Handle error
}
```

**JMS Queue Browsers**

Messages sent to a queue remain in the queue until the message consumer for that queue consumes them. The JMS API provides a QueueBrowser object that allows you to browse the messages in the queue and display the header values for each message. To create a QueueBrowser object, use the Session.createBrowser method. For example:

```java
QueueBrowser browser = session.createBrowser(queue);
```

See “A Simple Example of Browsing Messages in a Queue” on page 875 for an example of using a QueueBrowser object.
The `createBrowser` method allows you to specify a message selector as a second argument when you create a `QueueBrowser`. For information on message selectors, see “JMS Message Selectors” on page 839.

The JMS API provides no mechanism for browsing a topic. Messages usually disappear from a topic as soon as they appear: If there are no message consumers to consume them, the JMS provider removes them. Although durable subscriptions allow messages to remain on a topic while the message consumer is not active, no facility exists for examining them.

### JMS Exception Handling

The root class for exceptions thrown by JMS API methods is `JMSException`. Catching `JMSException` provides a generic way of handling all exceptions related to the JMS API.

The `JMSException` class includes the following subclasses, described in the API documentation:

- `IllegalStateException`
- `InvalidClientIDException`
- `InvalidDestinationException`
- `InvalidSelectorException`
- `JMSSecurityException`
- `MessageEOFException`
- `MessageFormatException`
- `MessageNotReadableException`
- `MessageNotWriteableException`
- `ResourceAllocationException`
- `TransactionInProgressException`
- `TransactionRolledBackException`

All the examples in the tutorial catch and handle `JMSException` when it is appropriate to do so.

### Creating Robust JMS Applications

This section explains how to use features of the JMS API to achieve the level of reliability and performance your application requires. Many people choose to implement JMS applications because they cannot tolerate dropped or duplicate messages and because they require that every message be received once and only once. The JMS API provides this functionality.

The most reliable way to produce a message is to send a `PERSISTENT` message within a transaction. JMS messages are `PERSISTENT` by default. A `transaction` is a unit of work into which you can group a series of operations, such as message sends and receives, so that the operations either all succeed or all fail. For details, see “Specifying Message Persistence” on page 845 and “Using JMS API Local Transactions” on page 849.
The most reliable way to consume a message is to do so within a transaction, either from a queue or from a durable subscription to a topic. For details, see "Creating Temporary Destinations" on page 846, "Creating Durable Subscriptions" on page 847, and "Using JMS API Local Transactions" on page 849.

For other applications, a lower level of reliability can reduce overhead and improve performance. You can send messages with varying priority levels (see "Setting Message Priority Levels" on page 845) and you can set them to expire after a certain length of time (see "Allowing Messages to Expire" on page 846).

The JMS API provides several ways to achieve various kinds and degrees of reliability. This section divides them into two categories, basic and advanced.

The following sections describe these features as they apply to JMS clients. Some of the features work differently in Java EE applications; in these cases, the differences are noted here and are explained in detail in “Using the JMS API in Java EE Applications” on page 851.

### Using Basic Reliability Mechanisms

The basic mechanisms for achieving or affecting reliable message delivery are as follows:

- **Controlling message acknowledgment**: You can specify various levels of control over message acknowledgment.
- **Specifying message persistence**: You can specify that messages are persistent, meaning they must not be lost in the event of a provider failure.
- **Setting message priority levels**: You can set various priority levels for messages, which can affect the order in which the messages are delivered.
- **Allowing messages to expire**: You can specify an expiration time for messages so they will not be delivered if they are obsolete.
- **Creating temporary destinations**: You can create temporary destinations that last only for the duration of the connection in which they are created.

### Controlling Message Acknowledgment

Until a JMS message has been acknowledged, it is not considered to be successfully consumed. The successful consumption of a message ordinarily takes place in three stages.

1. The client receives the message.
2. The client processes the message.
3. The message is acknowledged. Acknowledgment is initiated either by the JMS provider or by the client, depending on the session acknowledgment mode.

In transacted sessions (see “Using JMS API Local Transactions” on page 849), acknowledgment happens automatically when a transaction is committed. If a transaction is rolled back, all consumed messages are redelivered.
In nontransacted sessions, when and how a message is acknowledged depend on the value specified as the second argument of the createSession method. The three possible argument values are as follows:

- **Session.AUTO_ACKNOWLEDGE**: The session automatically acknowledges a client’s receipt of a message either when the client has successfully returned from a call to receive or when the MessageListener it has called to process the message returns successfully.

  A synchronous receive in an AUTO_ACKNOWLEDGE session is the one exception to the rule that message consumption is a three-stage process as described earlier. In this case, the receipt and acknowledgment take place in one step, followed by the processing of the message.

- **Session.CLIENT_ACKNOWLEDGE**: A client acknowledges a message by calling the message’s acknowledge method. In this mode, acknowledgment takes place on the session level: Acknowledging a consumed message automatically acknowledges the receipt of all messages that have been consumed by its session. For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.

  **Note** – In the Java EE platform, a CLIENT_ACKNOWLEDGE session can be used only in an application client, not in a web component or enterprise bean.

- **Session.DUPS_OK_ACKNOWLEDGE**: This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should be used only by consumers that can tolerate duplicate messages. (If the JMS provider delivers a message, it must set the value of the JMSRedelivered message header to true.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates.

If messages have been received from a queue but not acknowledged when a session terminates, the JMS provider retains them and redelivers them when a consumer next accesses the queue. The provider also retains unacknowledged messages for a terminated session that has a durable TopicSubscriber. (See “Creating Durable Subscriptions” on page 847.) Unacknowledged messages for a nondurable TopicSubscriber are dropped when the session is closed.

If you use a queue or a durable subscription, you can use the Session.recover method to stop a nontransacted session and restart it with its first unacknowledged message. In effect, the session’s series of delivered messages is reset to the point after its last acknowledged message. The messages it now delivers may be different from those that were originally delivered, if messages have expired or if higher-priority messages have arrived. For a nondurable TopicSubscriber, the provider may drop unacknowledged messages when its session is recovered.

The sample program in “A Message Acknowledgment Example” on page 887 demonstrates two ways to ensure that a message will not be acknowledged until processing of the message is complete.
Specifying Message Persistence

The JMS API supports two delivery modes specifying whether messages are lost if the JMS provider fails. These delivery modes are fields of the `DeliveryMode` interface.

- The **PERSISTENT** delivery mode, the default, instructs the JMS provider to take extra care to ensure that a message is not lost in transit in case of a JMS provider failure. A message sent with this delivery mode is logged to stable storage when it is sent.
- The **NON_PERSISTENT** delivery mode does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails.

You can specify the delivery mode in either of two ways.

- You can use the `setDeliveryMode` method of the `MessageProducer` interface to set the delivery mode for all messages sent by that producer. For example, the following call sets the delivery mode to NON_PERSISTENT for a producer:

```java
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

- You can use the long form of the `send` or the `publish` method to set the delivery mode for a specific message. The second argument sets the delivery mode. For example, the following `send` call sets the delivery mode for message to NON_PERSISTENT:

```java
producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
```

The third and fourth arguments set the priority level and expiration time, which are described in the next two subsections.

If you do not specify a delivery mode, the default is PERSISTENT. Using the NON_PERSISTENT delivery mode may improve performance and reduce storage overhead, but you should use it only if your application can afford to miss messages.

Setting Message Priority Levels

You can use message priority levels to instruct the JMS provider to deliver urgent messages first. You can set the priority level in either of two ways.

- You can use the `setPriority` method of the `MessageProducer` interface to set the priority level for all messages sent by that producer. For example, the following call sets a priority level of 7 for a producer:

```java
producer.setPriority(7);
```

- You can use the long form of the `send` or the `publish` method to set the priority level for a specific message. The third argument sets the priority level. For example, the following `send` call sets the priority level for message to 3:

```java
producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
```

The ten levels of priority range from 0 (lowest) to 9 (highest). If you do not specify a priority level, the default level is 4. A JMS provider tries to deliver higher-priority messages before lower-priority ones but does not have to deliver messages in exact order of priority.
Allowing Messages to Expire

By default, a message never expires. If a message will become obsolete after a certain period, however, you may want to set an expiration time. You can do this in either of two ways.

- You can use the `setTimeToLive` method of the `MessageProducer` interface to set a default expiration time for all messages sent by that producer. For example, the following call sets a time to live of one minute for a producer:

  ```java
  producer.setTimeToLive(60000);
  ```

- You can use the long form of the `send` or the `publish` method to set an expiration time for a specific message. The fourth argument sets the expiration time in milliseconds. For example, the following `send` call sets a time to live of 10 seconds:

  ```java
  producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
  ```

If the specified `timeToLive` value is 0, the message never expires.

When the message is sent, the specified `timeToLive` is added to the current time to give the expiration time. Any message not delivered before the specified expiration time is destroyed. The destruction of obsolete messages conserves storage and computing resources.

Creating Temporary Destinations

Normally, you create JMS destinations (queues and topics) administratively rather than programmatically. Your JMS provider includes a tool to create and remove destinations, and it is common for destinations to be long-lasting.

The JMS API also enables you to create destinations (TemporaryQueue and TemporaryTopic objects) that last only for the duration of the connection in which they are created. You create these destinations dynamically using the `Session.createTemporaryQueue` and the `Session.createTemporaryTopic` methods.

The only message consumers that can consume from a temporary destination are those created by the same connection that created the destination. Any message producer can send to the temporary destination. If you close the connection to which a temporary destination belongs, the destination is closed and its contents are lost.

You can use temporary destinations to implement a simple request/reply mechanism. If you create a temporary destination and specify it as the value of the `JMSReplyTo` message header field when you send a message, then the consumer of the message can use the value of the `JMSReplyTo` field as the destination to which it sends a reply. The consumer can also reference the original request by setting the `JMSCorrelationID` header field of the reply message to the value of the `JMSMessageID` header field of the request. For example, an `onMessage` method can create a session so that it can send a reply to the message it receives. It can use code such as the following:
Creating Robust JMS Applications

### Using Advanced Reliability Mechanisms

The more advanced mechanisms for achieving reliable message delivery are the following:

- **Creating durable subscriptions**: You can create durable topic subscriptions, which receive messages published while the subscriber is not active. Durable subscriptions offer the reliability of queues to the publish/subscribe message domain.

- **Using local transactions**: You can use local transactions, which allow you to group a series of sends and receives into an atomic unit of work. Transactions are rolled back if they fail at any time.

### Creating Durable Subscriptions

To ensure that a pub/sub application receives all published messages, use PERSISTENT delivery mode for the publishers and durable subscriptions for the subscribers.

The `Session.createConsumer` method creates a nondurable subscriber if a topic is specified as the destination. A nondurable subscriber can receive only messages that are published while it is active.

At the cost of higher overhead, you can use the `Session.createDurableSubscriber` method to create a durable subscriber. A durable subscription can have only one active subscriber at a time.

A durable subscriber registers a durable subscription by specifying a unique identity that is retained by the JMS provider. Subsequent subscriber objects that have the same identity resume the subscription in the state in which it was left by the preceding subscriber. If a durable subscription has no active subscriber, the JMS provider retains the subscription's messages until they are received by the subscription or until they expire.

You establish the unique identity of a durable subscriber by setting the following:

- A client ID for the connection
- A topic and a subscription name for the subscriber

You set the client ID administratively for a client-specific connection factory using either the command line or the Administration Console.

After using this connection factory to create the connection and the session, you call the `createDurableSubscriber` method with two arguments: the topic and a string that specifies the name of the subscription:

```java
producer = session.createProducer(msg.getJMSReplyTo());
replyMsg = session.createTextMessage("Consumer " +
    "processed message: " + msg.getText());
replyMsg.setJMSCorrelationID(msg.getJMSMessageID());
producer.send(replyMsg);
```

For more examples, see Chapter 48, "Java Message Service Examples."
String subName = "MySub";
MessageConsumer topicSubscriber =
    session.createDurableSubscriber(myTopic, subName);

The subscriber becomes active after you start the Connection or TopicConnection. Later, you might close the subscriber:

topicSubscriber.close();

The JMS provider stores the messages sent or published to the topic, as it would store messages sent to a queue. If the program or another application calls createDurableSubscriber using the same connection factory and its client ID, the same topic, and the same subscription name, then the subscription is reactivated and the JMS provider delivers any messages that were published while the subscriber was inactive.

To delete a durable subscription, first close the subscriber, then use the unsubscribe method with the subscription name as the argument:

topicSubscriber.close();
session.unsubscribe("MySub");

The unsubscribe method deletes the state the provider maintains for the subscriber.

Figure 47–6 and Figure 47–7 show the difference between a nondurable and a durable subscriber. With an ordinary, nondurable subscriber, the subscriber and the subscription begin and end at the same point and are, in effect, identical. When a subscriber is closed, the subscription also ends. Here, create stands for a call to Session.createConsumer with a Topic argument, and close stands for a call to MessageConsumer.close. Any messages published to the topic between the time of the first close and the time of the second create are not consumed by the subscriber. In Figure 47–6, the subscriber consumes messages M1, M2, M5, and M6, but messages M3 and M4 are lost.

---

**FIGURE 47–6**  Nondurable Subscribers and Subscriptions

With a durable subscriber, the subscriber can be closed and re-created, but the subscription continues to exist and to hold messages until the application calls the unsubscribe method. In
Figure 47–7, create stands for a call to Session.createDurableSubscriber, close stands for a call to MessageConsumer.close, and unsubscribe stands for a call to Session.unsubscribe. Messages published while the subscriber is closed are received when the subscriber is created again, so even though messages M2, M4, and M5 arrive while the subscriber is closed, they are not lost.

See “A MessageAcknowledgment Example” on page 887, “A Durable Subscription Example” on page 889, and “An Application That Uses the JMS API with a Session Bean” on page 897 for examples of Java EE applications that use durable subscriptions.

Using JMS API Local Transactions

You can group a series of operations into an atomic unit of work called a transaction. If any one of the operations fails, the transaction can be rolled back, and the operations can be attempted again from the beginning. If all the operations succeed, the transaction can be committed.

In a JMS client, you can use local transactions to group message sends and receives. The JMS API Session interface provides commit and rollback methods you can use in a JMS client. A transaction commit means that all produced messages are sent and all consumed messages are acknowledged. A transaction rollback means that all produced messages are destroyed and all consumed messages are recovered and redelivered unless they have expired (see “Allowing Messages to Expire” on page 846).

A transacted session is always involved in a transaction. As soon as the commit or the rollback method is called, one transaction ends and another transaction begins. Closing a transacted session rolls back its transaction in progress, including any pending sends and receives.
In an Enterprise JavaBeans component, you cannot use the `Session.commit` and `Session.rollback` methods. Instead, you use distributed transactions, described in “Using the JMS API in Java EE Applications” on page 851.

You can combine several sends and receives in a single JMS API local transaction. If you do so, you need to be careful about the order of the operations. You will have no problems if the transaction consists of all sends or all receives, or if the receives all come before the sends. However, if you try to use a request/reply mechanism, in which you send a message and then try to receive a reply to that message in the same transaction, the program will hang, because the send cannot take place until the transaction is committed. The following code fragment illustrates the problem:

```java
// Don't do this!
outMsg.setJMSReplyTo(replyQueue);
producer.send(outQueue, outMsg);
consumer = session.createConsumer(replyQueue);
inMsg = consumer.receive();
session.commit();
```

Because a message sent during a transaction is not actually sent until the transaction is committed, the transaction cannot contain any receives that depend on that message's having been sent.

In addition, the production and the consumption of a message cannot both be part of the same transaction. The reason is that the transactions take place between the clients and the JMS provider, which intervenes between the production and the consumption of the message. **Figure 47–8** illustrates this interaction.

**FIGURE 47–8 Using JMS API Local Transactions**

The sending of one or more messages to one or more destinations by Client 1 can form a single transaction, because it forms a single set of interactions with the JMS provider using a single session. Similarly, the receiving of one or more messages from one or more destinations by Client 2 also forms a single transaction using a single session. But because the two clients have no direct interaction and are using two different sessions, no transactions can take place between them.
Another way of putting this is that the act of producing and/or consuming messages in a session can be transactional, but the act of producing and consuming a specific message across different sessions cannot be transactional.

This is the fundamental difference between messaging and synchronized processing. Instead of tightly coupling the sending and receiving of data, message producers and consumers use an alternative approach to reliability, built on a JMS provider’s ability to supply a once-and-only-once message delivery guarantee.

When you create a session, you specify whether it is transacted. The first argument to the `createSession` method is a boolean value. A value of `true` means the session is transacted; a value of `false` means it is not transacted. The second argument to this method is the acknowledgment mode, which is relevant only to nontransacted sessions (see “Controlling Message Acknowledgment” on page 843). If the session is transacted, the second argument is ignored, so it is a good idea to specify 0 to make the meaning of your code clear. For example:

```java
session = connection.createSession(true, 0);
```

The `commit` and the `rollback` methods for local transactions are associated with the session. You can combine queue and topic operations in a single transaction if you use the same session to perform the operations. For example, you can use the same session to receive a message from a queue and send a message to a topic in the same transaction.

You can pass a client program’s session to a message listener’s constructor function and use it to create a message producer. In this way, you can use the same session for receives and sends in asynchronous message consumers.

“A Local Transaction Example” on page 891 provides an example of using JMS API local transactions.

### Using the JMS API in Java EE Applications

This section describes how using the JMS API in enterprise bean applications or web applications differs from using it in application clients.

A general rule in the Java EE platform specification applies to all Java EE components that use the JMS API within EJB or web containers: Application components in the web and EJB containers must not attempt to create more than one active (not closed) `Session` object per connection.

This rule does not apply to application clients. The application client container supports the creation of multiple sessions for each connection.
Using @Resource Annotations in Enterprise Bean or Web Components

When you use the @Resource annotation in an application client component, you normally declare the JMS resource static:

```java
@Resource(lookup = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(lookup = "jms/Queue")
private static Queue queue;
```

However, when you use this annotation in a session bean, a message-driven bean, or a web component, do not declare the resource static:

```java
@Resource(lookup = "jms/ConnectionFactory")
private ConnectionFactory connectionFactory;
@Resource(lookup = "jms/Topic")
private Topic topic;
```

If you declare the resource static in these components, runtime errors will result.

Using Session Beans to Produce and to Synchronously Receive Messages

An application that produces messages or synchronously receives them can use a session bean to perform these operations. The example in "An Application That Uses the JMS API with a Session Bean" on page 897 uses a stateless session bean to publish messages to a topic.

Because a blocking synchronous receive ties up server resources, it is not a good programming practice to use such a receive call in an enterprise bean. Instead, use a timed synchronous receive, or use a message-driven bean to receive messages asynchronously. For details about blocking and timed synchronous receives, see "Writing the Clients for the Synchronous Receive Example" on page 860.

Using the JMS API in an enterprise bean is in many ways similar to using it in an application client. The main differences are the areas of resource management and transactions.
Managing JMS Resources in Session Beans

The JMS API resources are a JMS API connection and a JMS API session. In general, it is important to release JMS resources when they are no longer being used. Here are some useful practices to follow:

- If you wish to maintain a JMS API resource only for the life span of a business method, it is a good idea to close the resource in a `finally` block within the method.
- If you would like to maintain a JMS API resource for the life span of an enterprise bean instance, it is a good idea to use a `@PostConstruct` callback method to create the resource and to use a `@PreDestroy` callback method to close the resource. If you use a stateful session bean and you wish to maintain the JMS API resource in a cached state, you must close the resource in a `@PrePassivate` callback method and set its value to `null`, and you must create it again in a `@PostActivate` callback method.

Managing Transactions in Session Beans

Instead of using local transactions, you use container-managed transactions for bean methods that perform sends or receives, allowing the EJB container to handle transaction demarcation. Because container-managed transactions are the default, you do not have to use an annotation to specify them.

You can use bean-managed transactions and the `javax.transaction.UserTransaction` interface's transaction demarcation methods, but you should do so only if your application has special requirements and you are an expert in using transactions. Usually, container-managed transactions produce the most efficient and correct behavior. This tutorial does not provide any examples of bean-managed transactions.

Using Message-Driven Beans to Receive Messages Asynchronously

The sections “What Is a Message-Driven Bean?” on page 439 and “How Does the JMS API Work with the Java EE Platform?” on page 827 describe how the Java EE platform supports a special kind of enterprise bean, the message-driven bean, which allows Java EE applications to process JMS messages asynchronously. Session beans allow you to send messages and to receive them synchronously but not asynchronously.

A message-driven bean is a message listener that can reliably consume messages from a queue or a durable subscription. The messages can be sent by any Java EE component (from an application client, another enterprise bean, or a web component) or from an application or a system that does not use Java EE technology.
Like a message listener in an application client, a message-driven bean contains an `onMessage` method that is called automatically when a message arrives. Like a message listener, a message-driven bean class can implement helper methods invoked by the `onMessage` method to aid in message processing.

A message-driven bean, however, differs from an application client’s message listener in the following ways:

- Certain setup tasks are performed by the EJB container.
- The bean class uses the `@MessageDriven` annotation to specify properties for the bean or the connection factory, such as a destination type, a durable subscription, a message selector, or an acknowledgment mode. The examples in Chapter 48, “Java Message Service Examples,” show how the JMS resource adapter works in the GlassFish Server.

The EJB container automatically performs several setup tasks that a stand-alone client must perform:

- Creating a message consumer to receive the messages. Instead of creating a message consumer in your source code, you associate the message-driven bean with a destination and a connection factory at deployment time. If you want to specify a durable subscription or use a message selector, you do this at deployment time also.
- Registering the message listener. You must not call `setMessageListener`.
- Specifying a message acknowledgment mode. The default mode, `AUTO_ACKNOWLEDGE`, is used unless it is overridden by a property setting.

If JMS is integrated with the application server using a resource adapter, the JMS resource adapter handles these tasks for the EJB container.

Your message-driven bean class must implement the `javax.jms.MessageListener` interface and the `onMessage` method.

It may implement a `@PostConstruct` callback method to create a connection, and a `@PreDestroy` callback method to close the connection. Typically, it implements these methods if it produces messages or performs synchronous receives from another destination.

The bean class commonly injects a `MessageDrivenContext` resource, which provides some additional methods you can use for transaction management.

The main difference between a message-driven bean and a session bean is that a message-driven bean has no local or remote interface. Instead, it has only a bean class.

A message-driven bean is similar in some ways to a stateless session bean: Its instances are relatively short-lived and retain no state for a specific client. The instance variables of the message-driven bean instance can contain some state across the handling of client messages: for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object.
Like a stateless session bean, a message-driven bean can have many interchangeable instances running at the same time. The container can pool these instances to allow streams of messages to be processed concurrently. The container attempts to deliver messages in chronological order when that would not impair the concurrency of message processing, but no guarantees are made as to the exact order in which messages are delivered to the instances of the message-driven bean class. Because concurrency can affect the order in which messages are delivered, you should write your applications to handle messages that arrive out of sequence.

For example, your application could manage conversations by using application-level sequence numbers. An application-level conversation control mechanism with a persistent conversation state could cache later messages until earlier messages have been processed.

Another way to ensure order is to have each message or message group in a conversation require a confirmation message that the sender blocks on receipt of. This forces the responsibility for order back onto the sender and more tightly couples senders to the progress of message-driven beans.

To create a new instance of a message-driven bean, the container does the following:
- Instantiates the bean
- Performs any required resource injection
- Calls the @PostConstruct callback method, if it exists

To remove an instance of a message-driven bean, the container calls the @PreDestroy callback method.

Figure 47–9 shows the lifecycle of a message-driven bean.
Managing Distributed Transactions

JMS client applications use JMS API local transactions (described in “Using JMS API Local Transactions” on page 849), which allow the grouping of sends and receives within a specific JMS session. Java EE applications commonly use distributed transactions to ensure the integrity of accesses to external resources. For example, distributed transactions allow multiple applications to perform atomic updates on the same database, and they allow a single application to perform atomic updates on multiple databases.

In a Java EE application that uses the JMS API, you can use transactions to combine message sends or receives with database updates and other resource manager operations. You can access resources from multiple application components within a single transaction. For example, a servlet can start a transaction, access multiple databases, invoke an enterprise bean that sends a JMS message, invoke another enterprise bean that modifies an EIS system using the Connector architecture, and finally commit the transaction. Your application cannot, however, both send a JMS message and receive a reply to it within the same transaction; the restriction described in “Using JMS API Local Transactions” on page 849 still applies.

Distributed transactions within the EJB container can be either of two kinds:

- **Container-managed transactions**: The EJB container controls the integrity of your transactions without your having to call commit or rollback. Container-managed transactions are recommended for Java EE applications that use the JMS API. You can specify appropriate transaction attributes for your enterprise bean methods.
  
  Use the Required transaction attribute (the default) to ensure that a method is always part of a transaction. If a transaction is in progress when the method is called, the method will be part of that transaction; if not, a new transaction will be started before the method is called and will be committed when the method returns.

- **Bean-managed transactions**: You can use these in conjunction with the javax.transaction.UserTransaction interface, which provides its own commit and rollback methods you can use to delimit transaction boundaries. Bean-managed transactions are recommended only for those who are experienced in programming transactions.

You can use either container-managed transactions or bean-managed transactions with message-driven beans. To ensure that all messages are received and handled within the context of a transaction, use container-managed transactions and use the Required transaction attribute (the default) for the onMessage method. This means that if there is no transaction in progress, a new transaction will be started before the method is called and will be committed when the method returns.
When you use container-managed transactions, you can call the following `MessageDrivenContext` methods:

- `setRollbackOnly`: Use this method for error handling. If an exception occurs, `setRollbackOnly` marks the current transaction so that the only possible outcome of the transaction is a rollback.

- `getRollbackOnly`: Use this method to test whether the current transaction has been marked for rollback.

If you use bean-managed transactions, the delivery of a message to the `onMessage` method takes place outside the distributed transaction context. The transaction begins when you call `UserTransaction.begin` method within the `onMessage` method, and it ends when you call `UserTransaction.commit` or `UserTransaction.rollback`. Any call to the `Connection.createSession` method must take place within the transaction. If you call `UserTransaction.rollback`, the message is not redelivered, whereas calling `setRollbackOnly` for container-managed transactions does cause a message to be redelivered.

Neither the JMS API specification nor the Enterprise JavaBeans specification (available from [http://jcp.org/en/jsr/detail?id=318](http://jcp.org/en/jsr/detail?id=318)) specifies how to handle calls to JMS API methods outside transaction boundaries. The Enterprise JavaBeans specification does state that the EJB container is responsible for acknowledging a message that is successfully processed by the `onMessage` method of a message-driven bean that uses bean-managed transactions. Using bean-managed transactions allows you to process the message by using more than one transaction or to have some parts of the message processing take place outside a transaction context. In most cases, however, container-managed transactions provide greater reliability and are therefore preferable.

When you create a session in an enterprise bean, the container ignores the arguments you specify, because it manages all transactional properties for enterprise beans. It is still a good idea to specify arguments of `true` and `0` to the `createSession` method to make this situation clear:

```java
session = connection.createSession(true, 0);
```

When you use container-managed transactions, you normally use the `Required` transaction attribute (the default) for your enterprise bean’s business methods.

You do not specify a message acknowledgment mode when you create a message-driven bean that uses container-managed transactions. The container acknowledges the message automatically when it commits the transaction.

If a message-driven bean uses bean-managed transactions, the message receipt cannot be part of the bean-managed transaction, so the container acknowledges the message outside the transaction.

If the `onMessage` method throws a `RuntimeException`, the container does not acknowledge processing the message. In that case, the JMS provider will redeliver the unacknowledged message in the future.
Using the JMS API with Application Clients and Web Components

An application client in a Java EE application can use the JMS API in much the same way that a stand-alone client program does. It can produce messages, and it can consume messages by using either synchronous receives or message listeners. See Chapter 25, “A Message-Driven Bean Example,” for an example of an application client that produces messages. For an example of using an application client to produce and to consume messages, see “An Application Example That Deploys a Message-Driven Bean on Two Servers” on page 916.

The Java EE platform specification does not impose strict constraints on how web components should use the JMS API. In the GlassFish Server, a web component can send messages and consume them synchronously but cannot consume them asynchronously.

Because a blocking synchronous receive ties up server resources, it is not a good programming practice to use such a receive call in a web component. Instead, use a timed synchronous receive. For details about blocking and timed synchronous receives, see “Writing the Clients for the Synchronous Receive Example” on page 860.

Further Information about JMS

For more information about JMS, see:

- Java Message Service web site:
  
  http://www.oracle.com/technetwork/java/index-jsp-142945.html

- Java Message Service specification, version 1.1, available from:
  
  http://www.oracle.com/technetwork/java/docs-136352.html
This chapter provides examples that show how to use the JMS API in various kinds of Java EE applications. It covers the following topics:

- “Writing Simple JMS Applications” on page 860
- “Writing Robust JMS Applications” on page 886
- “An Application That Uses the JMS API with a Session Bean” on page 897
- “An Application That Uses the JMS API with an Entity” on page 902
- “An Application Example That Consumes Messages from a Remote Server” on page 910
- “An Application Example That Deploys a Message-Driven Bean on Two Servers” on page 916

The examples are in the `tut-install/examples/jms/` directory.

The steps to build and run each example are as follows:

1. Use NetBeans IDE or Ant to compile and package the example.
2. Use NetBeans IDE or Ant to deploy the example and create resources for it.
3. Use NetBeans IDE, the `appclient` command, or Ant to run the client.

Each example has a `build.xml` file that refers to files in the `tut-install/examples/bp-project/` directory.

Each example has a `setup/glassfish-resources.xml` file that is used to create resources for the example.

See Chapter 25, "A Message-Driven Bean Example," for a simpler example of a Java EE application that uses the JMS API.
Writing Simple JMS Applications

This section shows how to create, package, and run simple JMS clients that are packaged as application clients and deployed to a Java EE server. The clients demonstrate the basic tasks a JMS application must perform:

- Creating a connection and a session
- Creating message producers and consumers
- Sending and receiving messages

In a Java EE application, some of these tasks are performed, in whole or in part, by the container. If you learn about these tasks, you will have a good basis for understanding how a JMS application works on the Java EE platform.

Each example uses two clients: one that sends messages and one that receives them. You can run the clients in NetBeans IDE or in two terminal windows.

When you write a JMS client to run in an enterprise bean application, you use many of the same methods in much the same sequence as for an application client. However, there are some significant differences. "Using the JMS API in Java EE Applications" on page 851 describes these differences, and this chapter provides examples that illustrate them.

The examples for this section are in the `tut-install/examples/jms/simple/` directory, under the following four subdirectories:

- producer
- synchconsumer
- asynchconsumer
- messagebrowser

A Simple Example of Synchronous Message Receives

This section describes the sending and receiving clients in an example that uses the receive method to consume messages synchronously. This section then explains how to compile, package, and run the clients using the GlassFish Server.

The following subsections describe the steps in creating and running the example.

Writing the Clients for the Synchronous Receive Example

The sending client, `producer/src/java/Producer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic:

   ```java
   @Resource(lookup = "jms/ConnectionFactory")
   private static ConnectionFactory connectionFactory;
   @Resource(lookup = "jms/Queue")private static Queue queue;
   @Resource(lookup = "jms/Topic")private static Topic topic;
   ```
2. Retrieves and verifies command-line arguments that specify the destination type and the number of arguments:

```java
final int NUM_MSGS;
String destType = args[0];
System.out.println("Destination type is " + destType);
if (!(destType.equals("queue") || destType.equals("topic"))) {
    System.err.println("Argument must be \"queue\" or \"topic\"\n" + "topic\"");
    System.exit(1);
} else {
    if (args.length == 2) {
        NUM_MSGS = (new Integer(args[1])).intValue();
    } else {
        NUM_MSGS = 1;
    }
}
```

3. Assigns either the queue or the topic to a destination object, based on the specified destination type:

```java
Destination dest = null;
try {
    if (destType.equals("queue")) {
        dest = (Destination) queue;
    } else {
        dest = (Destination) topic;
    }
} catch (Exception e) {
    System.err.println("Error setting destination: " + e.toString());
    e.printStackTrace();
    System.exit(1);
}
```

4. Creates a Connection and a Session:

```java
Connection connection = connectionFactory.createConnection();
Session session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

5. Creates a MessageProducer and a TextMessage:

```java
MessageProducer producer = session.createProducer(dest);
TextMessage message = session.createTextMessage();
```

6. Sends one or more messages to the destination:

```java
for (int i = 0; i < NUM_MSGS; i++) {
    message.setText("This is message " + (i + 1) + " from producer");
    System.out.println("Sending message: " + message.getText());
    producer.send(message);
}
```

7. Sends an empty control message to indicate the end of the message stream:

```java
producer.send(session.createMessage());
```

Sending an empty message of no specified type is a convenient way to indicate to the consumer that the final message has arrived.
8. Closes the connection in a `finally` block, automatically closing the session and `MessageProducer`:

```java
} finally {
    if (connection != null) {
        try { connection.close(); }
        catch (JMSException e) { }
    }
}
```

The receiving client, `synchconsumer/src/java/SynchConsumer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic.
2. Assigns either the queue or the topic to a destination object, based on the specified destination type.
3. Creates a `Connection` and a `Session`.
4. Creates a `MessageConsumer`:

   ```java
   consumer = session.createConsumer(dest);
   ```

5. Starts the connection, causing message delivery to begin:

   ```java
   connection.start();
   ```

6. Receives the messages sent to the destination until the end-of-message-stream control message is received:

   ```java
   while (true) {
       Message m = consumer.receive(1);
       if (m != null) {
           if (m instanceof TextMessage) {
               message = (TextMessage) m;
               System.out.println("Reading message: " + message.getText());
           } else {
               break;
           }
       }
   }
   ```

   Because the control message is not a `TextMessage`, the receiving client terminates the `while` loop and stops receiving messages after the control message arrives.

7. Closes the connection in a `finally` block, automatically closing the session and `MessageConsumer`.

The `receive` method can be used in several ways to perform a synchronous receive. If you specify no arguments or an argument of 0, the method blocks indefinitely until a message arrives:

```java
Message m = consumer.receive();
Message m = consumer.receive(0);
```
For a simple client, this may not matter. But if you do not want your application to consume system resources unnecessarily, use a timed synchronous receive. Do one of the following:

- Call the `receive` method with a timeout argument greater than 0:
  ```java
  Message m = consumer.receive(1); // 1 millisecond
  ```
- Call the `receiveNoWait` method, which receives a message only if one is available:
  ```java
  Message m = consumer.receiveNoWait();
  ```

The `SynchConsumer` client uses an indefinite while loop to receive messages, calling `receive` with a timeout argument. Calling `receiveNoWait` would have the same effect.

**Starting the JMS Provider**

When you use the GlassFish Server, your JMS provider is the GlassFish Server. Start the server as described in "Starting and Stopping the GlassFish Server" on page 73.

**JMS Administered Objects for the Synchronous Receive Example**

This example uses the following JMS administered objects:

- A connection factory
- Two destination resources: a topic and a queue

NetBeans IDE and the Ant tasks for the JMS examples create needed JMS resources when you deploy the applications, using a file named setup/glassfish-resources.xml. This file is most easily created using NetBeans IDE, although you can create it by hand.

You can also use the `asadmin create-jms-resource` command to create resources, the `asadmin list-jms-resources` command to display their names, and the `asadmin delete-jms-resource` command to remove them.

▼ **To Create JMS Resources Using NetBeans IDE**

Follow these steps to create a JMS resource in GlassFish Server using NetBeans IDE. Repeat these steps for each resource you need.

The example applications in this chapter already have the resources, so you will need to follow these steps only when you create your own applications.

1. **Right-click the project for which you want to create resources and choose New, then choose Other.**
   The New File wizard opens.

2. **Under Categories, select GlassFish.**

3. **Under File Types, select JMS Resource.**
   The General Attributes - JMS Resource page opens.
4 In the JNDI Name field, type the name of the resource.
   By convention, JMS resource names begin with jms/.

5 Select the radio button for the resource type.
   Normally, this is either javax.jms.Queue, javax.jms.Topic, or
   javax.jms.ConnectionFactory.

6 Click Next.
   The JMS Properties page opens.

7 For a queue or topic, type a name for a physical queue in the Value field for the Name property.
   You can type any value for this required field.
   Connection factories have no required properties. In a few situations, discussed in later
   sections, you may need to specify a property.

8 Click Finish.
   A file named glassfish-resources.xml is created in your project, in a directory named setup.
   In the project pane, you can find it under the Server Resources node. If this file exists, resources
   are created automatically by NetBeans IDE when you deploy the project.

▼ To Delete JMS Resources Using NetBeans IDE

1 In the Services pane, expand the Servers node, then expand the GlassFish Server 3+ node.

2 Expand the Resources node, then expand the Connector Resources node.

3 Expand the Admin Object Resources node.

4 Right-click any destination you want to remove and select Unregister.

5 Expand the Connector Connection Pools node.

6 Right-click any connection factory you want to remove and select Unregister.
   Every connection factory has both a connector connection pool and an associated connector
   resource. When you remove the connector connection pool, the resource is removed
   automatically. You can verify the removal by expanding the Connector Resources node.

Running the Clients for the Synchronous Receive Example

To run these examples using the GlassFish Server, package each one in an application client JAR
file. The application client JAR file requires a manifest file, located in the src/conf directory for
each example, along with the .class file.
The build.xml file for each example contains Ant targets that compile, package, and deploy the example. The targets place the .class file for the example in the build/jar directory. Then the targets use the jar command to package the class file and the manifest file in an application client JAR file.

Because the examples use the common interfaces, you can run them using either a queue or a topic.

▼ To Build and Package the Clients for the Synchronous Receive Example Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog, navigate to:
   tut-install/examples/jms/simple/
3. Select the producer folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the project and select Build.
7. From the File menu, choose Open Project again.
8. Select the synchconsumer folder.
9. Select the Open as Main Project check box.
10. Click Open Project.
11. In the Projects tab, right-click the project and select Build.

▼ To Deploy and Run the Clients for the Synchronous Receive Example Using NetBeans IDE

1. Deploy and run the Producer example:
   a. Right-click the producer project and select Properties.
   b. Select Run from the Categories tree.
c. In the Arguments field, type the following:
   queue 3

d. Click OK.

e. Right-click the project and select Run.
   The output of the program looks like this (along with some additional output):
   Destination type is queue
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   The messages are now in the queue, waiting to be received.

   Note – When you run an application client, the command often takes a long time to complete.

2 Now deploy and run the SynchConsumer example:

a. Right-click the synchconsumer project and select Properties.

b. Select Run from the Categories tree.

c. In the Arguments field, type the following:
   queue

d. Click OK.

e. Right-click the project and select Run.
   The output of the program looks like this (along with some additional output):
   Destination type is queue
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer

3 Now try running the programs in the opposite order. Right-click the synchconsumer project and select Run.
   The Output pane displays the destination type and then appears to hang, waiting for messages.

4 Right-click the producer project and select Run.
   When the messages have been sent, the SynchConsumer client receives them and exits. The Output pane shows the output of both programs, in two different tabs.
Now run the Producer example using a topic instead of a queue.

a. Right-click the producer project and select Properties.

b. Select Run from the Categories tree.

c. In the Arguments field, type the following:
   topic 3

d. Click OK.

e. Right-click the project and select Run.
   The output looks like this (along with some additional output):
   Destination type is topic
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer

Now run the SynchConsumer example using the topic.

a. Right-click the synchconsumer project and select Properties.

b. Select Run from the Categories tree.

c. In the Arguments field, type the following:
   topic

d. Click OK.

e. Right-click the project and select Run.
   The result, however, is different. Because you are using a topic, messages that were sent
   before you started the consumer cannot be received. (See “Publish/Subscribe Messaging
   Domain” on page 830 for details.) Instead of receiving the messages, the program appears to
   hang.

Run the Producer example again. Right-click the producer project and select Run.

Now the SynchConsumer example receives the messages:
   Destination type is topic
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
To Build and Package the Clients for the Synchronous Receive Example Using Ant

1. In a terminal window, go to the producer directory:
   ```
   cd producer
   ```

2. Type the following command:
   ```
   ant
   ```

3. In a terminal window, go to the synchconsumer directory:
   ```
   cd ../synchconsumer
   ```

4. Type the following command:
   ```
   ant
   ```

   The targets place the application client JAR file in the dist directory for each example.

To Deploy and Run the Clients for the Synchronous Receive Example Using Ant and the appclient Command

You can run the clients using the appclient command. The build.xml file for each project includes a target that creates resources, deploys the client, and then retrieves the client stubs that the appclient command uses. Each of the clients takes one or more command-line arguments: a destination type and, for Producer, a number of messages.

To build, deploy, and run the Producer and SynchConsumer examples using Ant and the appclient command, follow these steps.

To run the clients, you need two terminal windows.

1. In a terminal window, go to the producer directory:
   ```
   cd ../producer
   ```

2. Create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:
   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

3. Run the Producer program, sending three messages to the queue:
   ```
   appclient -client client-jar/producerClient.jar queue 3
   ```

   The output of the program looks like this (along with some additional output):

   Destination type is queue
   Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer

The messages are now in the queue, waiting to be received.

Note – When you run an application client, the command often takes a long time to complete.

4 In the same window, go to the synchconsumer directory:

   cd ../synchconsumer

5 Deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:

   ant getclient

   Ignore the message that states that the application is deployed at a URL.

6 Run the SynchConsumer client, specifying the queue:

   appclient -client client-jar/synchconsumerClient.jar queue

   The output of the client looks like this (along with some additional output):

   Destination type is queue
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer

7 Now try running the clients in the opposite order. Run the SynchConsumer client:

   appclient -client client-jar/synchconsumerClient.jar queue

   The client displays the destination type and then appears to hang, waiting for messages.

8 In a different terminal window, run the Producer client.

   cd tut-install/examples/jms/simple/producer
   appclient -client client-jar/producerClient.jar queue 3

   When the messages have been sent, the SynchConsumer client receives them and exits.

9 Now run the Producer client using a topic instead of a queue:

   appclient -client client-jar/producerClient.jar topic 3

   The output of the client looks like this (along with some additional output):

   Destination type is topic
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer

10 Now run the SynchConsumer client using the topic:

   appclient -client client-jar/synchconsumerClient.jar topic
The result, however, is different. Because you are using a topic, messages that were sent before you started the consumer cannot be received. (See “Publish/Subscribe Messaging Domain” on page 830 for details.) Instead of receiving the messages, the client appears to hang.

11 Run the Producer client again.

Now the SynchConsumer client receives the messages (along with some additional output):

```
Destination type is topic
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
```

A Simple Example of Asynchronous Message Consumption

This section describes the receiving clients in an example that uses a message listener to consume messages asynchronously. This section then explains how to compile and run the clients using the GlassFish Server.

Writing the Clients for the Asynchronous Receive Example

The sending client is `producer/src/java/Producer.java`, the same client used in the example in “A Simple Example of Synchronous Message Receives” on page 860.

An asynchronous consumer normally runs indefinitely. This one runs until the user types the character q or Q to stop the client.

The receiving client, `asynchconsumer/src/java/AsynchConsumer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic.
2. Assigns either the queue or the topic to a destination object, based on the specified destination type.
3. Creates a Connection and a Session.
4. Creates a MessageConsumer.
5. Creates an instance of the TextListener class and registers it as the message listener for the MessageConsumer:

   ```java
   listener = new TextListener();consumer.setMessageListener(listener);
   ```

6. Starts the connection, causing message delivery to begin.
7. Listens for the messages published to the destination, stopping when the user types the character q or Q:

   ```java
   System.out.println("To end program, type Q or q, " + "then <return>");
   inputStreamReader = new InputStreamReader(System.in);
   while ( (!((answer == 'q') || (answer == 'Q'))) {   
```
try {
    answer = (char) inputStreamReader.read();
} catch (IOException e) {
    System.out.println("I/O exception: " + e.toString());
}
}

8. Closes the connection, which automatically closes the session and MessageConsumer.

The message listener, asyncconsumer/src/java/TextListener.java, follows these steps:

1. When a message arrives, the onMessage method is called automatically.
2. The onMessage method converts the incoming message to a TextMessage and displays its content. If the message is not a text message, it reports this fact:

   ```java
   public void onMessage(Message message) {
       TextMessage msg = null;
       try {
           if (message instanceof TextMessage) {
               msg = (TextMessage) message;
               System.out.println("Reading message: " + msg.getText());
           } else {
               System.out.println("Message is not a " + "TextMessage");
           }
       } catch (JMSException e) {
           System.out.println("JMSException in onMessage(): " + e.toString());
       } catch (Throwable t) {
           System.out.println("Exception in onMessage():" + t.getMessage());
       }
   }
   ```

For this example, you will use the connection factory and destinations you created for "A Simple Example of Synchronous Message Receives" on page 860.

▼ To Build and Package the AsyncConsumer Client Using NetBeans IDE

1 From the File menu, choose Open Project.
2 In the Open Project dialog, navigate to:
   tut-install/examples/jms/simple/
3 Select the asyncconsumer folder.
4 Select the Open as Main Project check box.
5 Click Open Project.
6 In the Projects tab, right-click the project and select Build.
To Deploy and Run the Clients for the Asynchronous Receive Example Using NetBeans IDE

1. Run the AsyncConsumer example:
   a. Right-click the asyncconsumer project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:
      topic
   d. Click OK.
   e. Right-click the project and select Run.
      The client displays the following lines and appears to hang:
      Destination type is topic
      To end program, type Q or q, then <return>

2. Now run the Producer example:
   a. Right-click the producer project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:
      topic 3
   d. Click OK.
   e. Right-click the project and select Run.
      The output of the client looks like this:
      Destination type is topic
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer
      In the other tab, the AsyncConsumer client displays the following:
      Destination type is topic
      To end program, type Q or q, then <return>
      Reading message: This is message 1 from producer
      Reading message: This is message 2 from producer
      Reading message: This is message 3 from producer
      Message is not a TextMessage
The last line appears because the client has received the non-text control message sent by the Producer client.

3 Type Q or q in the Output window and press Return to stop the client.

4 Now run the Producer client using a queue.
   In this case, as with the synchronous example, you can run the Producer client first, because there is no timing dependency between the sender and the receiver.
   a. Right-click the producer project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:
      queue 3
   d. Click OK.
   e. Right-click the project and select Run.
      The output of the client looks like this:
      Destination type is queue
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer

5 Run the AsynchConsumer client.
   a. Right-click the asynchconsumer project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:
      queue
   d. Click OK.
   e. Right-click the project and select Run.
      The output of the client looks like this:
      Destination type is queue
      To end program, type Q or q, then <return>
      Reading message: This is message 1 from producer
      Reading message: This is message 2 from producer
      Reading message: This is message 3 from producer
      Message is not a TextMessage

6 Type Q or q in the Output window and press Return to stop the client.
To Build and Package the AsynchConsumer Client Using Ant

1. In a terminal window, go to the asynchconsumer directory:
   cd ../asynchconsumer

2. Type the following command:
   ant
   The targets package both the main class and the message listener class in the JAR file and place the file in the dist directory for the example.

To Deploy and Run the Clients for the Asynchronous Receive Example Using Ant and the appclient Command

1. Deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:
   ant getclient
   Ignore the message that states that the application is deployed at a URL.

2. Run the AsynchConsumer client, specifying the topic destination type.
   appclient -client client-jar/asynchconsumerClient.jar topic
   The client displays the following lines (along with some additional output) and appears to hang:
   Destination type is topic
   To end program, type Q or q, then <return>

3. In the terminal window where you ran the Producer client previously, run the client again, sending three messages.
   appclient -client client-jar/producerClient.jar topic 3
   The output of the client looks like this (along with some additional output):
   Destination type is topic
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   In the other window, the AsynchConsumer client displays the following (along with some additional output):
   Destination type is topic
   To end program, type Q or q, then <return>
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
   Message is not a TextMessage
The last line appears because the client has received the non-text control message sent by the Producer client.

4 Type Q or q and press Return to stop the client.

5 Now run the clients using a queue.

In this case, as with the synchronous example, you can run the Producer client first, because there is no timing dependency between the sender and receiver:

```
appclient -client client-jar/producerClient.jar queue 3
```

The output of the client looks like this:

```
Destination type is queue
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
```

6 Run the AsynchConsumer client:

```
appclient -client client-jar/asynchconsumerClient.jar queue
```

The output of the client looks like this (along with some additional output):

```
Destination type is queue
To end program, type Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```

7 Type Q or q to stop the client.

**A Simple Example of Browsing Messages in a Queue**

This section describes an example that creates a QueueBrowser object to examine messages on a queue, as described in "JMS Queue Browsers" on page 841. This section then explains how to compile, package, and run the example using the GlassFish Server.

**Writing the Client for the QueueBrowser Example**

To create a QueueBrowser for a queue, you call the Session.createBrowser method with the queue as the argument. You obtain the messages in the queue as an Enumeration object. You can then iterate through the Enumeration object and display the contents of each message.

The messagebrowser/src/java/MessageBrowser.java client performs the following steps:

1. Injects resources for a connection factory and a queue.
2. Creates a Connection and a Session.
3. Creates a QueueBrowser:

```java
QueueBrowser browser = session.createBrowser(queue);
```

4. Retrieves the Enumeration that contains the messages:

```java
Enumeration msgs = browser.getEnumeration();
```

5. Verifies that the Enumeration contains messages, then displays the contents of the messages:

```java
if ( !msgs.hasMoreElements() ) {
    System.out.println("No messages in queue");
} else {
    while (msgs.hasMoreElements()) {
        Message tempMsg = (Message)msgs.nextElement();
        System.out.println("Message: " + tempMsg);
    }
}
```

6. Closes the connection, which automatically closes the session and the QueueBrowser.

The format in which the message contents appear is implementation-specific. In the GlassFish Server, the message format looks something like this:

```
Message contents:
Text: This is message 3 from producer
Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSTimestamp(): 1129061034355
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
```

For this example, you will use the connection factory and queue you created for “A Simple Example of Synchronous Message Receives” on page 860.

To Run the MessageBrowser Client Using NetBeans IDE

To build, package, deploy, and run the MessageBrowser example using NetBeans IDE, follow these steps.

You also need the Producer example to send the message to the queue, and one of the consumer clients to consume the messages after you inspect them. If you did not do so already, package these examples.

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/jms/simple/`
3 Select the messagebrowser folder.

4 Select the Open as Main Project check box.

5 Click Open Project.

6 In the Projects tab, right-click the project and select Build.

7 Run the Producer client, sending one message to the queue:
   a. Right-click the producer project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:
      queue
   d. Click OK.
   e. Right-click the project and select Run.
      The output of the client looks like this:
      Destination type is queue
      Sending message: This is message 1 from producer

8 Run the MessageBrowser client. Right-click the messagebrowser project and select Run.
   The output of the client looks something like this:
   Message:
   Text: This is message 1 from producer
   Class: com.sun.messaging.jms.client.TextMessageImpl
   getJMSMessageID(): ID:12-128.149.71.199(8c:34:4a:1a:1b:b8)-40883-125590521747
   getJMSTimestamp(): 1129062957611
   getJMSCorrelationID(): null
   JMSReplyTo: null
   JMSDestination: PhysicalQueue
   getJMSDeliveryMode(): PERSISTENT
   getJMSRedelivered(): false
   getJMSType(): null
   getJMSExpiration(): 0
   getJMSPriority(): 4
   Properties: null
   Message:
   Class: com.sun.messaging.jms.client.MessageImpl
   getJMSMessageID(): ID:13-128.149.71.199(8c:34:4a:1a:1b:b8)-40883-125590521747
   getJMSTimestamp(): 1129062957616
   getJMSCorrelationID(): null
   JMSReplyTo: null
   JMSDestination: PhysicalQueue
   getJMSDeliveryMode(): PERSISTENT
   getJMSRedelivered(): false
The first message is the TextMessage, and the second is the non-text control message.

9 Run the SynchConsumer client to consume the messages.
   a. Right-click the synchconsumer project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:
      queue
   d. Click OK.
   e. Right-click the project and select Run.
      The output of the client looks like this:
      Destination type is queue
      Reading message: This is message 1 from producer

▼ To Run the MessageBrowser Client Using Ant and the appclient Command

To build, package, deploy, and run the MessageBrowser example using Ant, follow these steps.

You also need the Producer example to send the message to the queue, and one of the consumer clients to consume the messages after you inspect them. If you did not do so already, package these examples.

To run the clients, you need two terminal windows.

1 In a terminal window, go to the messagebrowser directory.
   cd ../messagebrowser

2 Type the following command:
   ant
   The targets place the application client JAR file in the dist directory for the example.

3 In another terminal window, go to the producer directory.

4 Run the Producer client, sending one message to the queue:
   appclient -client client-jar/producerClient.jar queue
The output of the client looks like this (along with some additional output):

Destination type is queue
Sending message: This is message 1 from producer

5 Go to the messagebrowser directory.

6 Deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:

   ant getclient

   Ignore the message that states that the application is deployed at a URL.

7 Because this example takes no command-line arguments, you can run the MessageBrowser client using the following command:

   ant run

   Alternatively, you can type the following command:

   appclient -client client-jar/messagebrowserClient.jar

   The output of the client looks something like this (along with some additional output):

   Message:
   Text: This is message 1 from producer
   Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
   getJMSMessageID(): ID:12-128.149.71.199(8c:34:4a:1a:1b:b8)-40883-1255900521747
   getJMSTimestamp(): 1255900521747
   getJMSCorrelationID(): null
   JMSReplyTo: null
   JMSDestination: PhysicalQueue
   getJMSDeliveryMode(): PERSISTENT
   getJMSRedelivered(): false
   getJMSType(): null
   getJMSExpiration(): 0
   getJMSPriority(): 4
   Properties: null
   Message:
   Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
   getJMSMessageID(): ID:13-128.149.71.199(8c:34:4a:1a:1b:b8)-40883-1255900521767
   getJMSTimestamp(): 1255900521767
   getJMSCorrelationID(): null
   JMSReplyTo: null
   JMSDestination: PhysicalQueue
   getJMSDeliveryMode(): PERSISTENT
   getJMSRedelivered(): false
   getJMSType(): null
   getJMSExpiration(): 0
   getJMSPriority(): 4
   Properties: null

   The first message is the TextMessage, and the second is the non-text control message.

8 Go to the synchconsumer directory.
Run the SynchConsumer client to consume the messages:

```
appclient -client client-jar/synchconsumerClient.jar queue
```

The output of the client looks like this (along with some additional output):

```
Destination type is queue
Reading message: This is message 1 from producer
```

## Running JMS Clients on Multiple Systems

JMS clients that use the GlassFish Server can exchange messages with each other when they are running on different systems in a network. The systems must be visible to each other by name (the UNIX host name or the Microsoft Windows computer name) and must both be running the GlassFish Server.

**Note** – Any mechanism for exchanging messages between systems is specific to the Java EE server implementation. This tutorial describes how to use the GlassFish Server for this purpose.

Suppose you want to run the Producer client on one system, earth, and the SynchConsumer client on another system, jupiter. Before you can do so, you need to perform these tasks:

1. Create two new connection factories
2. Change the name of the default JMS host on one system
3. Edit the source code for the two examples
4. Recompile and repackagethe examples

**Note** – A limitation in the JMS provider in the GlassFish Server may cause a runtime failure to create a connection to systems that use the Dynamic Host Configuration Protocol (DHCP) to obtain an IP address. You can, however, create a connection from a system that uses DHCP to a system that does not use DHCP. In the examples in this tutorial, earth can be a system that uses DHCP, and jupiter can be a system that does not use DHCP.

When you run the clients, they will work as shown in Figure 48–1. The client run on earth needs the queue on earth only so the resource injection will succeed. The connection, session, and message producer are all created on jupiter using the connection factory that points to jupiter. The messages sent from earth will be received on jupiter.
For examples showing how to deploy more complex applications on two different systems, see "An Application Example That Consumes Messages from a Remote Server" on page 910 and "An Application Example That Deploys a Message-Driven Bean on Two Servers" on page 916.

▼ **To Create Administered Objects for Multiple Systems**

To run these clients, you must do the following:

- Create a new connection factory on both earth and jupiter
- Create a destination resource on both earth and jupiter

You do not have to install the tutorial examples on both systems, but you must be able to access the file system where it is installed. You may find it more convenient to install the tutorial examples on both systems if the two use different operating systems (for example, Windows and UNIX). Otherwise you will have to edit the `tut-install/examples/bp-project/build.properties` file and change the location of the `javaee.home` property each time you build or run a client on a different system.

1. **Start the GlassFish Server on earth.**
2. **Start the GlassFish Server on jupiter.**
To create a new connection factory on *jupiter*, follow these steps:

a. From a command shell on *jupiter*, go to the `tut-install/examples/jms/simple/producer/` directory.

b. Type the following command:

```
ant create-local-factory
```

The `create-local-factory` target, defined in the `build.xml` file for the Producer example, creates a connection factory named `jms/JupiterConnectionFactory`.

To create a new connection factory on *earth* that points to the connection factory on *jupiter*, follow these steps:

a. From a command shell on *earth*, go to the `tut-install/examples/jms/simple/producer/` directory.

b. Type the following command:

```
ant create-remote-factory -Dsys=remote-system-name
```

Replace `remote-system-name` with the actual name of the remote system.

The `create-remote-factory` target, defined in the `build.xml` file for the Producer example, also creates a connection factory named `jms/JupiterConnectionFactory`. In addition, it sets the `AddressList` property for this factory to the name of the remote system.

Additional resources will be created when you deploy the application, if they have not been created before.

The reason the `glassfish-resources.xml` file does not specify `jms/JupiterConnectionFactory` is that on *earth* the connection factory requires the `AddressList` property setting, whereas on *jupiter* it does not. You can examine the targets in the `build.xml` file for details.

### Changing the Default Host Name

By default, the default host name for the JMS service on the GlassFish Server is `localhost`. To access the JMS service from another system, however, you must change the host name. You can change it either to the actual host name or to `0.0.0.0`.

You can change the default host name using either the Administration Console or the `asadmin` command.

#### To Change the Default Host Name Using the Administration Console


2. In the Administration Console, go to `Server Configuration` > `Server: Default` > `System Properties`.

3. Search for `DefaultHostName` and click on it.

4. Change the value to the desired host name or `0.0.0.0`.

5. Click `OK` to save the changes.

6. Restart the GlassFish Server to apply the changes.
In the navigation tree, expand the Configurations node, then expand the server-config node.

Under the server-config node, expand the Java Message Service node.

Under the Java Message Service node, expand the JMS Hosts node.

Under the JMS Hosts node, select default_JMS_host.

The Edit JMS Host page opens.

In the Host field, type the name of the system, or type 0.0.0.0.

Click Save.

Restart the GlassFish Server.

To Change the Default Host Name Using the asadmin Command

Specify a command like one of the following:

asadmin set server-config.jms-service.jms-host.default_JMS_host.host="0.0.0.0"

asadmin set server-config.jms-service.jms-host.default_JMS_host.host="hostname"

Restart the GlassFish Server.

To Run the Clients Using NetBeans IDE

These steps assume you have the tutorial installed on both of the systems you are using and that you are able to access the file system of jupiter from earth or vice versa. You will edit the source files to specify the new connection factory. Then you will rebuild and run the clients.

To edit the source files, follow these steps:

a. On earth, open the following file in NetBeans IDE:
   tut-install/examples/jms/simple/producer/src/java/Producer.java

b. Find the following line:
   @Resource(lookup = "jms/ConnectionFactory")

c. Change the line to the following:
   @Resource(lookup = "jms/JupiterConnectionFactory")

d. Save the file.
e. On Jupiter, open the following file in NetBeans IDE:
   `tut-install/examples/jms/simple/synchconsumer/src/java/SynchConsumer.java`

f. Repeat Step b and Step c, then save the file.

2 To recompile and repackgage the Producer example on Earth, right-click the producer project and select Clean and Build.

3 To recompile and repackgage the SynchConsumer example on Jupiter, right-click the synchconsumer project and select Clean and Build.

4 On Earth, deploy and run Producer. Follow these steps:
   a. Right-click the producer project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:
      ```queue 3```
   d. Click OK.
   e. Right-click the project and select Run.
      The output looks like this (along with some additional output):
      ```
      Destination type is topic
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer
      ```

5 On Jupiter, run SynchConsumer. Follow these steps:
   a. Right-click the synchconsumer project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:
      ```queue```
   d. Click OK.
   e. Right-click the project and select Run.
      The output of the program looks like this (along with some additional output):
      ```
      Destination type is queue
      Reading message: This is message 1 from producer
      Reading message: This is message 2 from producer
      ```
To Run the Clients Using Ant and the appclient Command

These steps assume you have the tutorial installed on both of the systems you are using and that you are able to access the file system of jupiter from earth or vice versa. You will edit the source files to specify the new connection factory. Then you will rebuild and run the clients.

1 To edit the source files, follow these steps:

a. On earth, open the following file in a text editor:
   `tut-install/examples/jms/simple/producer/src/java/Producer.java`

b. Find the following line:
   ```
   @Resource(lookup = "jms/ConnectionFactory")
   ```

c. Change the line to the following:
   ```
   @Resource(lookup = "jms/JupiterConnectionFactory")
   ```

d. Save and close the file.

e. On jupiter, open the following file in a text editor:
   `tut-install/examples/jms/simple/synchconsumer/src/java/SynchConsumer.java`

f. Repeat Step b and Step c, then save and close the file.

2 To recompile and repackaged the Producer example on earth, type the following:
   `ant`

3 To recompile and repackaged the SynchConsumer example on jupiter, go to the synchconsumer directory and type the following:
   `ant`

4 On earth, deploy and run Producer. Follow these steps:

a. On earth, from the producer directory, create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:
   `ant getclient`
   Ignore the message that states that the application is deployed at a URL.

b. To run the client, type the following:
   ```
   appclient -client client-jar/producerClient.jar queue 3
   ```
The output looks like this (along with some additional output):

```
Destination type is topic
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
```

5 **On jupiter, run SynchConsumer. Follow these steps:**

a. From the synchconsumer directory, create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:

```
ant getclient
```

Ignore the message that states that the application is deployed at a URL.

b. To run the client, type the following:

```
appclient -client client-jar/synchconsumerClient.jar queue
```

The output of the program looks like this (along with some additional output):

```
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
```

**Undeploying and Cleaning the Simple JMS Examples**

After you finish running the examples, you can undeploy them and remove the build artifacts.

You can also use the asadmin delete-jms-resource command to delete the destinations and connection factories you created. However, it is recommended that you keep them, because they will be used in most of the examples later in this chapter. After you have created them, they will be available whenever you restart the GlassFish Server.

**Writing Robust JMS Applications**

The following examples show how to use some of the more advanced features of the JMS API.
A Message Acknowledgment Example

The AckEquivExample.java client shows how both of the following scenarios ensure that a message will not be acknowledged until processing of it is complete:

- Using an asynchronous message consumer (a message listener) in an AUTO_ACKNOWLEDGE session
- Using a synchronous receiver in a CLIENT_ACKNOWLEDGE session

**Note** – In the Java EE platform, message listeners and CLIENT_ACKNOWLEDGE sessions can be used only in application clients, as in this example.

With a message listener, the automatic acknowledgment happens when the onMessage method returns (that is, after message processing has finished). With a synchronous receiver, the client acknowledges the message after processing is complete. If you use AUTO_ACKNOWLEDGE with a synchronous receive, the acknowledgment happens immediately after the receive call; if any subsequent processing steps fail, the message cannot be redelivered.

The example is in the following directory:

`tut-install/examples/jms/advanced/ackequivexample/src/java/

The example contains an AsynchSubscriber class with a TextListener class, a MultiplePublisher class, a SynchReceiver class, a SynchSender class, a main method, and a method that runs the other classes’ threads.

The example uses the following objects:

- `jms/ConnectionFactory`, `jms/Queue`, and `jms/Topic`: resources that you created for “A Simple Example of Synchronous Message Receives” on page 860.
- `jms/ControlQueue`: an additional queue
- `jms/DurableConnectionFactory`: a connection factory with a client ID (see “Creating Durable Subscriptions” on page 847 for more information)

The new queue and connection factory are created at deployment time.

You can use either NetBeans IDE or Ant to build, package, deploy, and run ackequivexample.

▼ **To Run ackequivexample Using NetBeans IDE**

1 To build and package the client, follow these steps.
   a. From the File menu, choose Open Project.
b. In the Open Project dialog, navigate to:
   \texttt{tut-install/examples/jms/advanced/}

c. Select the \texttt{ackequivexample} folder.

d. Select the Open as Main Project check box.

e. Click Open Project.

f. In the Projects tab, right-click the project and select Build.

2 To run the client, right-click the \texttt{ackequivexample} project and select Run.

   The client output looks something like this (along with some additional output):
   
   Queue name is \texttt{jms/ControlQueue}
   Queue name is \texttt{jms/Queue}
   Topic name is \texttt{jms/Topic}
   Connection factory name is \texttt{jms/DurableConnectionFactory}
   SENDER: Created client-acknowledge session
   SENDER: Sending message: Here is a client-acknowledge message
   RECEIVER: Created client-acknowledge session
   RECEIVER: Processing message: Here is a client-acknowledge message
   RECEIVER: Now I’ll acknowledge the message
   SUBSCRIBER: Created auto-acknowledge session
   SUBSCRIBER: Sending synchronize message to control queue
   PUBLISHER: Created auto-acknowledge session
   PUBLISHER: Receiving synchronize messages from control queue; count = 1
   PUBLISHER: Received synchronize message; expect 0 more
   PUBLISHER: Publishing message: Here is an auto-acknowledge message 1
   PUBLISHER: Publishing message: Here is an auto-acknowledge message 2
   SUBSCRIBER: Processing message: Here is an auto-acknowledge message 1
   SUBSCRIBER: Processing message: Here is an auto-acknowledge message 2
   SUBSCRIBER: Processing message: Here is an auto-acknowledge message 3

3 After you run the client, you can delete the destination resource \texttt{jms/ControlQueue} by using
   the following command:
   
   \texttt{asadmin delete-jms-resource \texttt{jms/ControlQueue}}

   You will need the other resources for other examples.

\textbf{To Run ackequivexample Using Ant}

1 In a terminal window, go to the following directory:
   \texttt{tut-install/examples/jms/advanced/ackequivexample/}

2 To compile and package the client, type the following command:
   \texttt{ant}
3 To create needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs, type the following command:

```java
ant getclient
```

Ignore the message that states that the application is deployed at a URL.

4 Because this example takes no command-line arguments, you can run the client using the following command:

```java
ant run
```

Alternatively, you can type the following command:

```java
appclient -client client-jar/ackequivexampleClient.jar
```

The client output looks something like this (along with some additional output):

```
Queue name is jms/ControlQueue
Queue name is jms/Queue
Topic name is jms/Topic
Connection factory name is jms/DurableConnectionFactory
SENDER: Created client-acknowledge session
SENDER: Sending message: Here is a client-acknowledge message
RECEIVER: Created client-acknowledge session
RECEIVER: Processing message: Here is a client-acknowledge message
RECEIVER: Now I'll acknowledge the message
SUBSCRIBER: Created auto-acknowledge session
SUBSCRIBER: Sending synchronize message to control queue
PUBLISHER: Created auto-acknowledge session
PUBLISHER: Receiving synchronize messages from control queue; count = 1
PUBLISHER: Received synchronize message; expect 0 more
PUBLISHER: Publishing message: Here is an auto-acknowledge message 1
PUBLISHER: Publishing message: Here is an auto-acknowledge message 2
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 1
PUBLISHER: Publishing message: Here is an auto-acknowledge message 3
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 2
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 3
```

5 After you run the client, you can delete the destination resource jms/ControlQueue by using the following command:

```java
asadmin delete-jms-resource jms/ControlQueue
```

You will need the other resources for other examples.

## A Durable Subscription Example

DurableSubscriberExample.java shows how durable subscriptions work. It demonstrates that a durable subscription is active even when the subscriber is not active. The example contains a DurableSubscriber class, a MultiplePublisher class, a main method, and a method that instantiates the classes and calls their methods in sequence.
The example is in the `tut-install/examples/jms/advanced/durablesubscriberexample/src/java/` directory.

The example begins in the same way as any publish/subscribe client: The subscriber starts, the publisher publishes some messages, and the subscriber receives them. At this point, the subscriber closes itself. The publisher then publishes some messages while the subscriber is not active. The subscriber then restarts and receives those messages.

You can use either NetBeans IDE or Ant to build, package, deploy, and run `durablesubscriberexample`.

▼ To Run `durablesubscriberexample` Using NetBeans IDE

1. To compile and package the client, follow these steps:
   a. From the File menu, choose Open Project.
   b. In the Open Project dialog, navigate to:
      `tut-install/examples/jms/advanced/
   c. Select the `durablesubscriberexample` folder.
   d. Select the Open as Main Project check box.
   e. Click Open Project.
   f. In the Projects tab, right-click the project and select Build.

2. To run the client, right-click the `durablesubscriberexample` project and select Run.

   The output looks something like this (along with some additional output):

   ```
   Connection factory without client ID is jms/ConnectionFactory
   Connection factory with client ID is jms/DurableConnectionFactory
   Topic name is jms/Topic
   Starting subscriber
   PUBLISHER: Publishing message: Here is a message 1
   SUBSCRIBER: Reading message: Here is a message 1
   PUBLISHER: Publishing message: Here is a message 2
   SUBSCRIBER: Reading message: Here is a message 2
   PUBLISHER: Publishing message: Here is a message 3
   SUBSCRIBER: Reading message: Here is a message 3
   Closing subscriber
   PUBLISHER: Publishing message: Here is a message 4
   PUBLISHER: Publishing message: Here is a message 5
   PUBLISHER: Publishing message: Here is a message 6
   Starting subscriber
   SUBSCRIBER: Reading message: Here is a message 4
   SUBSCRIBER: Reading message: Here is a message 5
   SUBSCRIBER: Reading message: Here is a message 6
   ```
Closing subscriber
Unsubscribing from durable subscription

3 After you run the client, you can delete the connection factory
   jms/DurableConnectionFactory by using the following command:
   asadmin delete-jms-resource jms/DurableConnectionFactory

▼ To Run durablesubscriberexample Using Ant

1 In a terminal window, go to the following directory:
   tut-install/examples/jms/advanced/durablesubscriberexample/

2 To compile and package the client, type the following command:
   ant

3 To create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve
   the client stubs, type the following command:
   ant getclient
   Ignore the message that states that the application is deployed at a URL.

4 Because this example takes no command-line arguments, you can run the client using the
   following command:
   ant run
   Alternatively, you can type the following command:
   appclient -client client-jar/durablesubscriberexampleClient.jar

5 After you run the client, you can delete the connection factory
   jms/DurableConnectionFactory by using the following command:
   asadmin delete-jms-resource jms/DurableConnectionFactory

A Local Transaction Example

TransactedExample.java demonstrates the use of transactions in a JMS client application. The
example is in the tut-install/examples/jms/advanced/transactedexample/src/java/
directory.

This example shows how to use a queue and a topic in a single transaction as well as how to pass
a session to a message listener’s constructor function. The example represents a highly
simplified e-commerce application in which the following actions occur.

1. A retailer sends a MapMessage to the vendor order queue, ordering a quantity of computers,
   and waits for the vendor’s reply:
2. The vendor receives the retailer's order message and sends an order message to the supplier order topic in one transaction. This JMS transaction uses a single session, so you can combine a receive from a queue with a send to a topic. Here is the code that uses the same session to create a consumer for a queue and a producer for a topic:

```java
vendorOrderReceiver = session.createConsumer(vendorOrderQueue);
supplierOrderProducer = session.createProducer(supplierOrderTopic);
```

The following code receives the incoming message, sends an outgoing message, and commits the session. The message processing has been removed to keep the sequence simple:

```java
inMessage = vendorOrderReceiver.receive();
// Process the incoming message and format the outgoing message
... supplierOrderProducer.send(orderMessage);
... session.commit();
```

For simplicity, there are only two suppliers, one for CPUs and one for hard drives.

3. Each supplier receives the order from the order topic, checks its inventory, and then sends the items ordered to the queue named in the order message's JMSReplyTo field. If it does not have enough of the item in stock, the supplier sends what it has. The synchronous receive from the topic and the send to the queue take place in one JMS transaction.

```java
receiver = session.createConsumer(orderTopic);
...
inMessage = receiver.receive();
if (inMessage instanceof MapMessage) {
    orderMessage = (MapMessage) inMessage;
}
// Process message
MessageProducer producer =
    session.createProducer((Queue) orderMessage.getJMSReplyTo());
outMessage = session.createMapMessage();
// Add content to message
producer.send(outMessage);
// Display message contentssession.commit();
```

4. The vendor receives the suppliers' replies from its confirmation queue and updates the state of the order. Messages are processed by an asynchronous message listener; this step shows the use of JMS transactions with a message listener.

```java
MapMessage component = (MapMessage) message;
...
orderNumber = component.getInt("VendorOrderNumber");
Order order = Order.getOrder(orderNumber).processSubOrder(component);
session.commit();

5. When all outstanding replies are processed for a given order, the vendor message listener
sends a message notifying the retailer whether it can fulfill the order.

Queue replyQueue = (Queue) order.order.getJMSReplyTo();
MessageProducer producer = session.createProducer(replyQueue);
MapMessage retailerConfirmMessage = session.createMapMessage();
// Format the message
producer.send(retailerConfirmMessage);
session.commit();

6. The retailer receives the message from the vendor:

inMessage = (MapMessage) orderConfirmReceiver.receive();

Figure 48–2 illustrates these steps.

The example contains five classes: GenericSupplier, Order, Retailer, Vendor, and VendorMessageListener. The example also contains a main method and a method that runs
the threads of the Retailer, Vendor, and two supplier classes.
All the messages use the MapMessage message type. Synchronous receives are used for all message reception except when the vendor processes the replies of the suppliers. These replies are processed asynchronously and demonstrate how to use transactions within a message listener.

At random intervals, the Vendor class throws an exception to simulate a database problem and cause a rollback.

All classes except Retailer use transacted sessions.

The example uses three queues named jms/AQueue, jms/BQueue, and jms/CQueue, and one topic named jms/OTopic.

You can use either NetBeans IDE or Ant to build, package, deploy, and run transactedexample.

▼ To Run transactedexample Using NetBeans IDE

1  In a terminal window, go to the following directory:
   tut-install/examples/jms/advanced/transactedexample/

2  To compile and package the client, follow these steps:
   a. From the File menu, choose Open Project.
   b. In the Open Project dialog, navigate to:
      tut-install/examples/jms/advanced/
   c. Select the transactedexample folder.
   d. Select the Open as Main Project check box.
   e. Click Open Project.
   f. In the Projects tab, right-click the project and select Build.

3  To deploy and run the client, follow these steps:
   a. Right-click the transactedexample project and select Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type a number that specifies the number of computers to order:
d. Click OK.

e. Right-click the project and select Run.

The output looks something like this (along with some additional output):

```
Quantity to be ordered is 3
Retailer: ordered 3 computer(s)
Vendor: ordered 3 computer(s)
Vendor: ordered 3 CPU(s) and hard drive(s)
CPU Supplier: Vendor ordered 3 CPU(s)
CPU Supplier: sent 3 CPU(s)
CPU Supplier: committed transaction
Vendor: committed transaction 1
Hard Drive Supplier: Vendor ordered 3 Hard Drive(s)
Hard Drive Supplier: sent 1 Hard Drive(s)
Vendor: Completed processing for order 1
  Hard Drive Supplier: committed transaction
Vendor: unable to send 3 computer(s)
  Vendor: committed transaction 2
Retailer: Order not filled
Retailer: placing another order
Retailer: ordered 6 computer(s)
Vendor: JMSException occurred: javax.jms.JMSException:
  Simulated database concurrent access exception
  at TransactedExamplesVendor.run(Unknown Source)
  Vendor: rolled back transaction 1
Vendor: Retailer ordered 6 Computer(s)
Vendor: ordered 6 CPU(s) and hard drive(s)
CPU Supplier: Vendor ordered 6 CPU(s)
Hard Drive Supplier: Vendor ordered 6 Hard Drive(s)
CPU Supplier: sent 6 CPU(s)
  CPU Supplier: committed transaction
Hard Drive Supplier: sent 6 Hard Drive(s)
  Hard Drive Supplier: committed transaction
Vendor: committed transaction 1
Vendor: Completed processing for order 2
Vendor: sent 6 computer(s)
Retailer: Order filled
Vendor: committed transaction 2
```

4. After you run the client, you can delete the destination resources in NetBeans IDE or by using the following commands:

```
asadmin delete-jms-resource jms/AQueue
asadmin delete-jms-resource jms/BQueue
asadmin delete-jms-resource jms/CQueue
asadmin delete-jms-resource jms/OTopic
```

▼ To Run transactedexample Using Ant and the appclient Command

1. In a terminal window, go to the following directory:
   
   tut-install/examples/jms/advanced/transactedexample/
To build and package the client, type the following command:

ant

Create needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:

ant getclient

Ignore the message that states that the application is deployed at a URL.

Use a command like the following to run the client.
The argument specifies the number of computers to order.

appclient -client client-jar/transactedexampleClient.jar 3

The output looks something like this (along with some additional output):

Quantity to be ordered is 3
Retailer: ordered 3 computer(s)
Vendor: Retailer ordered 3 Computer(s)
Vendor: ordered 3 CPU(s) and hard drive(s)
CPU Supplier: Vendor ordered 3 CPU(s)
CPU Supplier: sent 3 CPU(s)
  CPU Supplier: committed transaction
Vendor: committed transaction 1
Hard Drive Supplier: Vendor ordered 3 Hard Drive(s)
Hard Drive Supplier: sent 1 Hard Drive(s)
Vendor: Completed processing for order 1
  Hard Drive Supplier: committed transaction
Vendor: unable to send 3 computer(s)
  Vendor: committed transaction 2
Retailer: Order not filled
Retailer: placing another order
Retailer: ordered 6 computer(s)
Vendor: JMSException occurred: javax.jms.JMSException:
  Simulated database concurrent access exception
  javax.jms.JMSException: Simulated database concurrent access exception
  at TransactedExample$Vendor.run(Unknown Source)
  Vendor: rolled back transaction 1
Vendor: Retailer ordered 6 Computer(s)
Vendor: ordered 6 CPU(s) and hard drive(s)
CPU Supplier: Vendor ordered 6 CPU(s)
Hard Drive Supplier: Vendor ordered 6 Hard Drive(s)
CPU Supplier: sent 6 CPU(s)
  CPU Supplier: committed transaction
Hard Drive Supplier: sent 6 Hard Drive(s)
  Hard Drive Supplier: committed transaction
Vendor: committed transaction 1
Vendor: Completed processing for order 2
Vendor: sent 6 computer(s)
Retailer: Order filled
  Vendor: committed transaction 2
After you run the client, you can delete the destination resources by using the following commands:

- `asadmin delete-jms-resource jms/AQueue`
- `asadmin delete-jms-resource jms/BQueue`
- `asadmin delete-jms-resource jms/CQueue`
- `asadmin delete-jms-resource jms/OTopic`

### An Application That Uses the JMS API with a Session Bean

This section explains how to write, compile, package, deploy, and run an application that uses the JMS API in conjunction with a session bean. The application contains the following components:

- An application client that invokes a session bean
- A session bean that publishes several messages to a topic
- A message-driven bean that receives and processes the messages using a durable topic subscriber and a message selector

You will find the source files for this section in the `tut-install/examples/jms/clientsessionmdb/` directory. Path names in this section are relative to this directory.

### Writing the Application Components for the clientsessionmdb Example

This application demonstrates how to send messages from an enterprise bean (in this case, a session bean) rather than from an application client, as in the example in Chapter 25, “A Message-Driven Bean Example.” Figure 48–3 illustrates the structure of this application.
The Publisher enterprise bean in this example is the enterprise-application equivalent of a wire-service news feed that categorizes news events into six news categories. The message-driven bean could represent a newsroom, where the sports desk, for example, would set up a subscription for all news events pertaining to sports.

The application client in the example injects the Publisher enterprise bean’s remote home interface and then calls the bean’s business method. The enterprise bean creates 18 text messages. For each message, it sets a String property randomly to one of six values representing the news categories and then publishes the message to a topic. The message-driven bean uses a message selector for the property to limit which of the published messages it receives.

**Coding the Application Client: MyAppClient.java**

The application client, `MyAppClient.java`, performs no JMS API operations and so is simpler than the client in Chapter 25, “A Message-Driven Bean Example.” The client uses dependency injection to obtain the Publisher enterprise bean’s business interface:

```java
@EJB(name="PublisherRemote")
static private PublisherRemote publisher;
```

The client then calls the bean’s business method twice.

**Coding the Publisher Session Bean**

The Publisher bean is a stateless session bean that has one business method. The Publisher bean uses a remote interface rather than a local interface because it is accessed from the application client.
The remote interface, `clientsessionmdb-ejb/src/java/sb/PublisherRemote.java`, declares a single business method, `publishNews`.

The bean class, `clientsessionmdb-ejb/src/java/sb/PublisherBean.java`, implements the `publishNews` method and its helper method `chooseType`. The bean class also injects `SessionContext`, `ConnectionFactory`, and `Topic` resources and implements `@PostConstruct` and `@PreDestroy` callback methods. The bean class begins as follows:

```java
@Stateless
@Remote({PublisherRemote.class})
public class PublisherBean implements PublisherRemote {
    @Resource
    private SessionContext sc;

    @Resource(lookup = "jms/ConnectionFactory")
    private ConnectionFactory connectionFactory;

    @Resource(lookup = "jms/Topic")
    private Topic topic;

    ...}
```

The `@PostConstruct` callback method of the bean class, `makeConnection`, creates the Connection used by the bean. The business method `publishNews` creates a Session and a MessageProducer and publishes the messages.

The `@PreDestroy` callback method, `endConnection`, deallocates the resources that were allocated by the `@PostConstruct` callback method. In this case, the method closes the Connection.

### Coding the Message-Driven Bean: `MessageBean.java`

The message-driven bean class, `clientsessionmdb-ejb/src/java/mdb/MessageBean.java`, is almost identical to the one in Chapter 25, "A Message-Driven Bean Example." However, the `@MessageDriven` annotation is different, because instead of a queue, the bean is using a topic with a durable subscription, and it is also using a message selector. Therefore, the annotation sets the activation config properties `messageSelector`, `subscriptionDurability`, `clientId`, and `subscriptionName`, as follows:

```java
@MessageDriven(mappedName = "jms/Topic", activationConfig = {
    @ActivationConfigProperty(propertyName = "messageSelector",
                                propertyValue = "NewsType = 'Sports' OR NewsType = 'Opinion'"),
    @ActivationConfigProperty(propertyName = "subscriptionDurability",
                                propertyValue = "Durable"),
    @ActivationConfigProperty(propertyName = "clientId",
                                propertyValue = "MyID"),
    @ActivationConfigProperty(propertyName = "subscriptionName",
                                propertyValue = "MySub")
})
```
Note – For a message-driven bean, the destination is specified with the mappedName element instead of the lookup element.

The JMS resource adapter uses these properties to create a connection factory for the message-driven bean that allows the bean to use a durable subscriber.

Creating Resources for the clientsessionmdb Example

This example uses the topic named jms/Topic and the connection factory jms/ConnectionFactory, which are also used in previous examples. If you deleted the connection factory or topic, they will be recreated when you deploy the example.

Running the clientsessionmdb Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the clientsessionmdb example.

▼ To Run the clientsessionmdb Example Using NetBeans IDE

1. To compile and package the project, follow these steps:
   a. From the File menu, choose Open Project.
   b. In the Open Project dialog, navigate to:
      tut-install/examples/jms/
   c. Select the clientsessionmdb folder.
   d. Select the Open as Main Project check box and the Open Required Projects check box.
   e. Click Open Project.
   f. In the Projects tab, right-click the clientsessionmdb project and select Build.

      This task creates the following:
      - An application client JAR file that contains the client class file and the session bean’s remote interface, along with a manifest file that specifies the main class and places the EJB JAR file in its classpath
      - An EJB JAR file that contains both the session bean and the message-driven bean
An application EAR file that contains the two JAR files

2 Right-click the project and select Run.

This command creates any needed resources, deploys the project, returns a JAR file named clientsessionmdbClient.jar, and then executes it.

The output of the application client in the Output pane looks like this (preceded by application client container output):

To view the bean output, check <install_dir>/domains/domain1/logs/server.log.

The output from the enterprise beans appears in the server log (domain-dir/logs/server.log), wrapped in logging information. The Publisher session bean sends two sets of 18 messages numbered 0 through 17. Because of the message selector, the message-driven bean receives only the messages whose NewsType property is Sports or Opinion.

▼ To Run the clientsessionmdb Example Using Ant

1 Go to the following directory:

\t\texttt{tut-install/examples/jms/clientsessionmdb/}

2 To compile the source files and package the application, use the following command:

\texttt{ant}

The ant command creates the following:

- An application client JAR file that contains the client class file and the session bean’s remote interface, along with a manifest file that specifies the main class and places the EJB JAR file in its classpath
- An EJB JAR file that contains both the session bean and the message-driven bean
- An application EAR file that contains the two JAR files

The clientsessionmdb.ear file is created in the dist directory.

3 To create any needed resources, deploy the application, and run the client, use the following command:

\texttt{ant run}

Ignore the message that states that the application is deployed at a URL.

The client displays these lines (preceded by application client container output):

To view the bean output, check <install_dir>/domains/domain1/logs/server.log.
The output from the enterprise beans appears in the server log file, wrapped in logging information. The Publisher session bean sends two sets of 18 messages numbered 0 through 17. Because of the message selector, the message-driven bean receives only the messages whose NewsType property is Sports or Opinion.

An Application That Uses the JMS API with an Entity

This section explains how to write, compile, package, deploy, and run an application that uses the JMS API with an entity. The application uses the following components:

- An application client that both sends and receives messages
- Two message-driven beans
- An entity class

You will find the source files for this section in the tut-install/examples/jms/clientmdbentity/ directory. Path names in this section are relative to this directory.

Overview of the clientmdbentity Example Application

This application simulates, in a simplified way, the work flow of a company’s human resources (HR) department when it processes a new hire. This application also demonstrates how to use the Java EE platform to accomplish a task that many JMS applications need to perform.

A JMS client must often wait for several messages from various sources. It then uses the information in all these messages to assemble a message that it then sends to another destination. The common term for this process is joining messages. Such a task must be transactional, with all the receives and the send as a single transaction. If not all the messages are received successfully, the transaction can be rolled back. For an application client example that illustrates this task, see "A Local Transaction Example" on page 891.

A message-driven bean can process only one message at a time in a transaction. To provide the ability to join messages, an application can have the message-driven bean store the interim information in an entity. The entity can then determine whether all the information has been received; when it has, the entity can report this back to one of the message-driven beans, which then creates and sends the message to the other destination. After it has completed its task, the entity can be removed.

The basic steps of the application are as follows.
1. The HR department's application client generates an employee ID for each new hire and then publishes a message (M1) containing the new hire's name, employee ID, and position. The client then creates a temporary queue, ReplyQueue, with a message listener that waits for a reply to the message. (See “Creating Temporary Destinations” on page 846 for more information.)

2. Two message-driven beans process each message: One bean, OfficeMDB, assigns the new hire's office number, and the other bean, EquipmentMDB, assigns the new hire's equipment. The first bean to process the message creates and persists an entity named SetupOffice, then calls a business method of the entity to store the information it has generated. The second bean locates the existing entity and calls another business method to add its information.

3. When both the office and the equipment have been assigned, the entity business method returns a value of true to the message-driven bean that called the method. The message-driven bean then sends to the reply queue a message (M2) describing the assignments. Then it removes the entity. The application client's message listener retrieves the information.

Figure 48–4 illustrates the structure of this application. Of course, an actual HR application would have more components; other beans could set up payroll and benefits records, schedule orientation, and so on.

Figure 48–4 assumes that OfficeMDB is the first message-driven bean to consume the message from the client. OfficeMDB then creates and persists the SetupOffice entity and stores the office information. EquipmentMDB then finds the entity, stores the equipment information, and learns that the entity has completed its work. EquipmentMDB then sends the message to the reply queue and removes the entity.
Writing the Application Components for the clientmdbentity Example

Writing the components of the application involves coding the application client, the message-driven beans, and the entity class.

Coding the Application Client: HumanResourceClient.java

The application client, clientmdbentity-app-client/src/java/HumanResourceClient.java, performs the following steps:

1. Injects ConnectionFactory and Topic resources
2. Creates a TemporaryQueue to receive notification of processing that occurs, based on new-hire events it has published
3. Creates a MessageConsumer for the TemporaryQueue, sets the MessageConsumer’s message listener, and starts the connection
4. Creates a MessageProducer and a MapMessage
5. Creates five new employees with randomly generated names, positions, and ID numbers (in sequence) and publishes five messages containing this information
The message listener, HRLListener, waits for messages that contain the assigned office and equipment for each employee. When a message arrives, the message listener displays the information received and determines whether all five messages have arrived. When they have, the message listener notifies the main method, which then exits.

**Coding the Message-Driven Beans for the clientmdbentity Example**

This example uses two message-driven beans:

- clientmdbentity-ejb/src/java/eb/EquipmentMDB.java
- clientmdbentity-ejb/src/java/eb/OfficeMDB.java

The beans take the following steps:

1. They inject MessageDrivenContext and ConnectionFactory resources.
2. The onMessage method retrieves the information in the message. The EquipmentMDB's onMessage method chooses equipment, based on the new hire’s position; the OfficeMDB's onMessage method randomly generates an office number.
3. After a slight delay to simulate real world processing hitches, the onMessage method calls a helper method, compose.
4. The compose method takes the following steps:
   a. It either creates and persists the SetupOffice entity or finds it by primary key.
   b. It uses the entity to store the equipment or the office information in the database, calling either the doEquipmentList or the doOfficeNumber business method.
   c. If the business method returns true, meaning that all of the information has been stored, it creates a connection and a session, retrieves the reply destination information from the message, creates a MessageProducer, and sends a reply message that contains the information stored in the entity.
   d. It removes the entity.

**Coding the Entity Class for the clientmdbentity Example**

The SetupOffice class, clientmdbentity-ejb/src/java/eb/SetupOffice.java, is an entity class. The entity and the message-driven beans are packaged together in an EJB JAR file. The entity class is declared as follows:

```java
@Entity
public class SetupOffice implements Serializable {

    // Constructor and methods
}
```

The class contains a no-argument constructor and a constructor that takes two arguments, the employee ID and name. It also contains getter and setter methods for the employee ID, name, office number, and equipment list. The getter method for the employee ID has the @Id annotation to indicate that this field is the primary key.
@Id
public String getEmployeeId() {
    return id;
}

The class also implements the two business methods, doEquipmentList and doOfficeNumber, and their helper method, checkIfSetupComplete.

The message-driven beans call the business methods and the getter methods.

The persistence.xml file for the entity specifies the most basic settings:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="clientmdbentity-ejbPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/__default</jta-data-source>
    <class>eb.SetupOffice</class>
    <properties>
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Creating Resources for the clientmdbentity Example

This example uses the connection factory jms/ConnectionFactory and the topic jms/Topic, both of which you used in “An Application That Uses the JMS API with a Session Bean” on page 897. It also uses the JDBC resource named jdbc/__default, which is enabled by default when you start the GlassFish Server.

If you deleted the connection factory or topic, they will be created when you deploy the example.

Running the clientmdbentity Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the clientmdbentity example.
To Run the clientmdbentity Example Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   tut-install/examples/jms/

3. Select the clientmdbentity folder.

4. Select the Open as Main Project check box and the Open Required Projects check box.

5. Click Open Project.

6. In the Projects tab, right-click the clientmdbentity project and select Build.
   
   This task creates the following:
   - An application client JAR file that contains the client class and listener class files, along with a manifest file that specifies the main class
   - An EJB JAR file that contains the message-driven beans and the entity class, along with the persistence.xml file
   - An application EAR file that contains the two JAR files along with an application.xml file

7. If the Java DB database is not already running, follow these steps:
   
   a. Click the Services tab.
   
   b. Expand the Databases node.
   
   c. Right-click the Java DB node and select Start Server.

8. In the Projects tab, right-click the project and select Run.

   This command creates any needed resources, deploys the project, returns a client JAR file named clientmdbentityClient.jar, and then executes it.

   The output of the application client in the Output pane looks something like this:

   PUBLISHER: Setting hire ID to 50, name Bill Tudor, position Programmer
   PUBLISHER: Setting hire ID to 51, name Carol Jones, position Senior Programmer
   PUBLISHER: Setting hire ID to 52, name Mark Wilson, position Manager
   PUBLISHER: Setting hire ID to 53, name Polly Wren, position Senior Programmer
   PUBLISHER: Setting hire ID to 54, name Joe Lawrence, position Director

   Waiting for 5 message(s)
   New hire event processed:
   Employee ID: 52
   Name: Mark Wilson
   Equipment: PDA
   Office number: 294
Waiting for 4 message(s)
New hire event processed:
  Employee ID: 53
  Name: Polly Wren
  Equipment: Laptop
  Office number: 186
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 54
  Name: Joe Lawrence
  Equipment: Java Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
  Name: Carol Jones
  Equipment: Laptop
  Office number: 262

The output from the message-driven beans and the entity class appears in the server log, wrapped in logging information.

For each employee, the application first creates the entity and then finds it. You may see runtime errors in the server log, and transaction rollbacks may occur. The errors occur if both of the message-driven beans discover at the same time that the entity does not yet exist, so they both try to create it. The first attempt succeeds, but the second fails because the bean already exists. After the rollback, the second message-driven bean tries again and succeeds in finding the entity. Container-managed transactions allow the application to run correctly, in spite of these errors, with no special programming.

▼ To Run the clientmdbentity Example Using Ant

1. Go to the following directory:
   tut-install/examples/jms/clientmdbentity/
2. To compile the source files and package the application, use the following command:
   ant

The ant command creates the following:
- An application client JAR file that contains the client class and listener class files, along with a manifest file that specifies the main class
- An EJB JAR file that contains the message-driven beans and the entity class, along with the persistence.xml file
- An application EAR file that contains the two JAR files along with an application.xml file
To create any needed resources, deploy the application, and run the client, use the following command:

```
ant run
```

This command starts the database server if it is not already running, then deploys and runs the application.

Ignore the message that states that the application is deployed at a URL.

The output in the terminal window looks something like this (preceded by application client container output):

```
running application client container.
PUBLISHER: Setting hire ID to 50, name Bill Tudor, position Programmer
PUBLISHER: Setting hire ID to 51, name Carol Jones, position Senior Programmer
PUBLISHER: Setting hire ID to 52, name Mark Wilson, position Manager
PUBLISHER: Setting hire ID to 53, name Polly Wren, position Senior Programmer
PUBLISHER: Setting hire ID to 54, name Joe Lawrence, position Director
```

Waiting for 5 message(s)

```
New hire event processed:
  Employee ID: 52
  Name: Mark Wilson
  Equipment: PDA
  Office number: 294
Waiting for 4 message(s)
New hire event processed:
  Employee ID: 53
  Name: Polly Wren
  Equipment: Laptop
  Office number: 186
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 54
  Name: Joe Lawrence
  Equipment: Java Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
  Name: Carol Jones
  Equipment: Laptop
  Office number: 262
```

The output from the message-driven beans and the entity class appears in the server log, wrapped in logging information.

For each employee, the application first creates the entity and then finds it. You may see runtime errors in the server log, and transaction rollbacks may occur. The errors occur if both of the message-driven beans discover at the same time that the entity does not yet exist, so they both try to create it. The first attempt succeeds, but the second fails because the bean already
exists. After the rollback, the second message-driven bean tries again and succeeds in finding the entity. Container-managed transactions allow the application to run correctly, in spite of these errors, with no special programming.

An Application Example That Consumes Messages from a Remote Server

This section and the following section explain how to write, compile, package, deploy, and run a pair of Java EE modules that run on two Java EE servers and that use the JMS API to interchange messages with each other. It is a common practice to deploy different components of an enterprise application on different systems within a company, and these examples illustrate on a small scale how to do this for an application that uses the JMS API.

The two examples work in slightly different ways. In the first example, the deployment information for a message-driven bean specifies the remote server from which it will consume messages. In the next example, described in “An Application Example That Deploys a Message-Driven Bean on Two Servers” on page 916, the same message-driven bean is deployed on two different servers, so it is the client module that specifies the servers (one local, one remote) to which it is sending messages.

This first example divides the example in Chapter 25, “A Message-Driven Bean Example,” into two modules: one containing the application client, and the other containing the message-driven bean.

You will find the source files for this section in the `tut-install/examples/jms/consumerremote` directory. Path names in this section are relative to this directory.

Overview of the consumerremote Example Modules

This example is very similar to the one in Chapter 25, “A Message-Driven Bean Example,” except for the fact that it is packaged as two separate modules:

- One module contains the application client, which runs on the remote system and sends three messages to a queue.
- The other module contains the message-driven bean, which is deployed on the local server and consumes the messages from the queue on the remote server.

The basic steps of the modules are as follows:

1. The administrator starts two Java EE servers, one on each system.
2. On the local server, the administrator deploys the message-driven bean module, which specifies the remote server where the client is deployed.
3. On the remote server, the administrator places the client JAR file.
4. The client module sends three messages to a queue.
5. The message-driven bean consumes the messages.

Figure 48–5 illustrates the structure of this application. You can see that it is almost identical to Figure 25–1 except that there are two Java EE servers. The queue used is the one on the remote server; the queue must also exist on the local server for resource injection to succeed.

**Writing the Module Components for the consumer remote Example**

Writing the components of the modules involves
- Coding the application client
- Coding the message-driven bean

The application client, jupiterclient/src/java/SimpleClient.java, is almost identical to the one in “The simple message Application Client” on page 492.

Similarly, the message-driven bean, earthmdb/src/java/MessageBean.java, is almost identical to the one in “The Message-Driven Bean Class” on page 493. The only significant difference is that the activation config properties include one property that specifies the name of the remote system. You need to edit the source file to specify the name of your system.

**Creating Resources for the consumer remote Example**

The application client can use any connection factory that exists on the remote server; in this example, it uses jms/ConnectionFactory. Both components use the queue named jms/Queue,

**FIGURE 48–5  A Java EE Application That Consumes Messages from a Remote Server**

<table>
<thead>
<tr>
<th>Java EE Server on Jupiter</th>
<th>Sends</th>
<th>Delivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Client</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java EE Server on Earth</th>
<th>EJB Container</th>
<th>MDB Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Msg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 48 • Java Message Service Examples
which you created for “A Simple Example of Synchronous Message Receives” on page 860. The message-driven bean does not need a previously created connection factory; the resource adapter creates one for it.

Any missing resources will be created when you deploy the example.

Using Two Application Servers for the consumerremote Example

As in “Running JMS Clients on Multiple Systems” on page 880, the two servers are referred to as earth and jupiter.

The GlassFish Server must be running on both systems.

Before you can run the example, you must change the default name of the JMS host on jupiter, as described in “To Change the Default Host Name Using the Administration Console” on page 882. If you have already performed this task, you do not have to repeat it.

Which system you use to package and deploy the modules and which system you use to run the client depend on your network configuration (specifically, which file system you can access remotely). These instructions assume you can access the file system of jupiter from earth but cannot access the file system of earth from jupiter. (You can use the same systems for jupiter and earth that you used in “Running JMS Clients on Multiple Systems” on page 880.)

You can package both modules on earth and deploy the message-driven bean there. The only action you perform on jupiter is running the client module.

Running the consumerremote Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the consumerremote example.

▼ To Run the consumerremote Example Using NetBeans IDE

To edit the message-driven bean source file and then package, deploy, and run the modules using NetBeans IDE, follow these steps.

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to:  
   tut-install/examples/jms/consumerremote/

3 Select the earthmdb folder.
4 Select the Open as Main Project check box.
5 Click Open Project.
6 Edit the MessageBean.java file as follows:
   a. In the Projects tab, expand the earthmdb, Source Packages, and mdb nodes, then double-click MessageBean.java.
   b. Find the following line within the @MessageDriven annotation:

      ```java
      @ActivationConfigProperty(propertyName = "addressList",
                               propertyValue = "remotesystem"),
      ```

   c. Replace remotesystem with the name of your remote system.
7 Right-click the earthmdb project and select Build.
   This command creates a JAR file that contains the bean class file.
8 From the File menu, choose Open Project.
9 Select the jupiterclient folder.
10 Select the Open as Main Project check box.
11 Click Open Project.
12 In the Projects tab, right-click the jupiterclient project and select Build.
   This command creates a JAR file that contains the client class file and a manifest file.
13 Right-click the earthmdb project and select Deploy.
14 To copy the jupiterclient module to the remote system, follow these steps:
   a. Change to the directory jupiterclient/dist:

      ```bash
      cd tut-install/examples/jms/consumerremote/jupiterclient/dist
      ```
   b. Type a command like the following:

      ```bash
      cp jupiterclient.jar F:/
      ```
   That is, copy the client JAR file to a location on the remote file system. You can use the file system graphical user interface on your system instead of the command line.
To run the application client, follow these steps:

a. If you did not previously create the queue and connection factory on the remote system (jupiter), go to the `<tut-install>/examples/jms/consumerremote/jupiterclient/` directory on the remote system and type the following command:

   `ant add-resources`

b. Go to the directory on the remote system (jupiter) where you copied the client JAR file.

c. To deploy the client module and retrieve the client stubs, use the following command:

   `asadmin deploy --retrieve . jupiterclient.jar`

   This command deploys the client JAR file and retrieves the client stubs in a file named `jupiterclientClient.jar`.

d. To run the client, use the following command:

   `appclient -client jupiterclientClient.jar`

   On `jupiter`, the output of the `appclient` command looks like this (preceded by application client container output):

   Sending message: This is message 1 from jupiterclient
   Sending message: This is message 2 from jupiterclient
   Sending message: This is message 3 from jupiterclient

   On `earth`, the output in the server log looks something like this (preceded by logging information):

   MESSAGE BEAN: Message received: This is message 1 from jupiterclient
   MESSAGE BEAN: Message received: This is message 2 from jupiterclient
   MESSAGE BEAN: Message received: This is message 3 from jupiterclient

e. To undeploy the client after you finish running it, use the following command:

   `asadmin undeploy jupiterclient`

To Run the consumerremote Example Using Ant

To edit the message-driven bean source file and then package, deploy, and run the modules using Ant, follow these steps.

1. Open the following file in an editor:

   `<tut-install>/examples/jms/consumerremote/earthmdb/src/java/mdb/MessageBean.java`

2. Find the following line within the `@MessageDriven` annotation:

   ```java
   @ActivationConfigProperty(propertyName = "addressList",
                           propertyValue = "remotesystem"),
   ```

3. Replace `remotesystem` with the name of your remote system, then save and close the file.
4 Go to the following directory:  
   tut-install/examples/jms/consumerremote/earthmdb/

5 Type the following command:  
   ant  
   This command creates a JAR file that contains the bean class file.

6 Type the following command:  
   ant deploy  

7 Go to the jupiterclient directory:  
   cd ../jupiterclient  

8 Type the following command:  
   ant  
   This command creates a JAR file that contains the client class file and a manifest file.

9 To copy the jupiterclient module to the remote system, follow these steps:  
   a. Change to the directory jupiterclient/dist:  
      cd ../jupiterclient/dist  
   b. Type a command like the following:  
      cp jupiterclient.jar F:/  
      That is, copy the client JAR file to a location on the remote file system.

10 To run the application client, follow these steps:  
   a. If you did not previously create the queue and connection factory on the remote system  
      (jupiter), go to the tut-install/examples/jms/consumerremote/jupiterclient/ directory  
      on the remote system and type the following command:  
      ant add-resources  
   b. Go to the directory on the remote system (jupiter) where you copied the client JAR file.  
   c. To deploy the client module and retrieve the client stubs, use the following command:  
      asadmin deploy --retrieve . jupiterclient.jar  
      This command deploys the client JAR file and retrieves the client stubs in a file named  
      jupiterclientClient.jar  
   d. To run the client, use the following command:  
      appclient -client jupiterclientClient.jar
On jupiter, the output of the appclient command looks like this (preceded by application client container output):

Sending message: This is message 1 from jupiterclient
Sending message: This is message 2 from jupiterclient
Sending message: This is message 3 from jupiterclient

On earth, the output in the server log looks something like this (preceded by logging information):

MESSAGE BEAN: Message received: This is message 1 from jupiterclient
MESSAGE BEAN: Message received: This is message 2 from jupiterclient
MESSAGE BEAN: Message received: This is message 3 from jupiterclient

To undeploy the client after you finish running it, use the following command:

asadmin undeploy jupiterclient

An Application Example That Deploys a Message-Driven Bean on Two Servers

This section, like the preceding one, explains how to write, compile, package, deploy, and run a pair of Java EE modules that use the JMS API and run on two Java EE servers. These modules are slightly more complex than the ones in the first example.

The modules use the following components:

- An application client that is deployed on the local server. It uses two connection factories, an ordinary one and one configured to communicate with the remote server, to create two publishers and two subscribers and to publish and consume messages.
- A message-driven bean that is deployed twice: once on the local server, and once on the remote one. It processes the messages and sends replies.

In this section, the term local server means the server on which both the application client and the message-driven bean are deployed (earth in the preceding example). The term remote server means the server on which only the message-driven bean is deployed (jupiter in the preceding example).

You will find the source files for this section in the tut-install/examples/jms/sendremote/directory. Path names in this section are relative to this directory.

Overview of the sendremote Example Modules

This pair of modules is somewhat similar to the modules in “An Application Example That Consumes Messages from a Remote Server” on page 910 in that the only components are a
client and a message-driven bean. However, the modules here use these components in more complex ways. One module consists of the application client. The other module contains only the message-driven bean and is deployed twice, once on each server.

The basic steps of the modules are as follows.

1. You start two Java EE servers, one on each system.

2. On the local server (earth), you create two connection factories: one local and one that communicates with the remote server (jupiter). On the remote server, you create a connection factory that has the same name as the one that communicates with the remote server.

3. The application client looks up the two connection factories (the local one and the one that communicates with the remote server) to create two connections, sessions, publishers, and subscribers. The subscribers use a message listener.

4. Each publisher publishes five messages.

5. Each of the local and the remote message-driven beans receives five messages and sends replies.

6. The client’s message listener consumes the replies.

Figure 48–6 illustrates the structure of this application. M1 represents the first message sent using the local connection factory, and RM1 represents the first reply message sent by the local MDB. M2 represents the first message sent using the remote connection factory, and RM2 represents the first reply message sent by the remote MDB.
Writing the Module Components for the sendremote Example

Writing the components of the modules involves coding the application client and the message-driven bean.

**Coding the Application Client: MultiAppServerClient.java**

The application client class, `multiclient/src/java/MultiAppServerClient.java`, does the following:

1. It injects resources for two connection factories and a topic.
2. For each connection factory, it creates a connection, a publisher session, a publisher, a subscriber session, a subscriber, and a temporary topic for replies.
3. Each subscriber sets its message listener, ReplyListener, and starts the connection.
4. Each publisher publishes five messages and creates a list of the messages the listener should expect.
5. When each reply arrives, the message listener displays its contents and removes it from the list of expected messages.
6. When all the messages have arrived, the client exits.

**Coding the Message-Driven Bean: ReplyMsgBean.java**

The message-driven bean class, replybean/src/ReplyMsgBean.java, does the following:

1. Uses the @MessageDriven annotation:
   ```java
   @MessageDriven(mappedName = "jms/Topic")
   ```
2. Injects resources for the MessageDrivenContext and for a connection factory. It does not need a destination resource because it uses the value of the incoming message’s JMSReplyTo header as the destination.
3. Uses a @PostConstruct callback method to create the connection, and a @PreDestroy callback method to close the connection.

The onMessage method of the message-driven bean class does the following:

1. Casts the incoming message to a TextMessage and displays the text
2. Creates a connection, a session, and a publisher for the reply message
3. Publishes the message to the reply topic
4. Closes the connection

On both servers, the bean will consume messages from the topic jms/Topic.

**Creating Resources for the send remote Example**

This example uses the connection factory named jms/ConnectionFactory and the topic named jms/Topic. These objects must exist on both the local and the remote servers.

This example uses an additional connection factory, jms/JupiterConnectionFactory, which communicates with the remote system; you created it in “To Create Administered Objects for Multiple Systems” on page 881. This connection factory must exist on the local server.

The build.xml file for the multiclient module contains targets you can use to create these resources if you deleted them previously.

To create the resource needed only on the local system, use the following command:

```
ant create-remote-factory -Dsys=remote-system-name
```

The other resources will be created when you deploy the application.
To Enable Deployment on the Remote System

GlassFish Server by default does not allow deployment on a remote system. You must create a password for the administrator on the remote system, then enable secure administration on that system. After that, you will be able to deploy the message-driven bean on the remote system.

2. In the navigation tree, expand the Configurations node, then expand the server-config node.
3. Expand the Security node.
4. Expand the Realms node.
5. Select the admin-realm node.
6. On the Edit Realm page, click Manage Users.
7. In the File Users table, click admin in the User ID column.
8. On the Edit File Realm Users page, type a password (for example, jmsadmin) in the New Password and Confirm New Password fields, then click Save.
9. In the navigation tree, click the Server (Admin Server) node.
11. Click Enable Secure Admin, accepting the default values for the alias and instance.
12. The server on **jupiter** will stop and restart automatically. Log in to the Administration Console with the admin user ID and the password you created and verify that the settings are correct.

To Use Two Application Servers for the send remote Example

If you are using NetBeans IDE, you need to add the remote server in order to deploy the message-driven bean there. To do so, follow these steps.

1. In NetBeans IDE, click the Services tab.
Right-click the Servers node and select Add Server. In the Add Server Instance dialog, follow these steps:

a. Select GlassFish Server 3+ from the Server list.

b. In the Name field, specify a name slightly different from that of the local server, such as GlassFish Server 3+ (2).

c. Click Next.

d. For the Server Location, browse to the location of the GlassFish Server on the remote system. This location must be visible from the local system.

e. Click Next.

f. Select the Register Remote Domain radio button.

g. In the Host Name field, type the name of the remote system.

h. Click Finish.

i. In the dialog that appears, enter the user name (admin) and the password you created.

Before you can run the example, you must change the default name of the JMS host on jupiter, as described in “To Change the Default Host Name Using the Administration Console” on page 882. If you have already performed this task, you do not have to repeat it.

**Running the sendremote Example**

You can use either NetBeans IDE or Ant to build, package, deploy, and run the sendremote example.

**To Run the sendremote Example Using NetBeans IDE**

To build the replybean module, follow these steps:

a. From the File menu, choose Open Project.

b. In the Open Project dialog, navigate to:
   
tut-install/examples/jms/sendremote/

   c. Select the replybean folder.
d. Select the Open as Main Project check box.

e. Click Open Project.

f. In the Projects tab, right-click the replybean project and select Build.
   This command creates a JAR file that contains the bean class file.

2 To build the multiclient module, follow these steps:

   a. From the File menu, choose Open Project.

   b. Select the multiclient folder.

   c. Select the Open as Main Project check box.

   d. Click Open Project.

   e. In the Projects tab, right-click the multiclient project and select Build.
      This command creates a JAR file that contains the client class file and a manifest file.

3 To create any needed resources and deploy the multiclient module on the local server, follow these steps:

   a. Right-click the multiclient project and select Properties.

   b. Select Run from the Categories tree.

   c. From the Server list, select GlassFish Server 3+ (the local server).

   d. Click OK.

   e. Right-click the multiclient project and select Deploy.
      You can use the Services tab to verify that multiclient is deployed as an App Client Module on the local server.

4 To deploy the replybean module on the local and remote servers, follow these steps:

   a. Right-click the replybean project and select Properties.

   b. Select Run from the Categories tree.

   c. From the Server list, select GlassFish Server 3+ (the local server).
d. Click OK.

e. Right-click the replybean project and select Deploy.

f. Right-click the replybean project again and select Properties.

g. Select Run from the Categories tree.

h. From the Server list, select GlassFish Server 3+ (2) (the remote server).

i. Click OK.

j. Right-click the replybean project and select Deploy.

You can use the Services tab to verify that replybean is deployed as an EJB Module on both servers.

5 To run the application client, right-click the multiclient project and select Run.

This command returns a JAR file named multiclientClient.jar and then executes it.

On the local system, the output of the appc1ient command looks something like this:

running application client container.
...
Sent message: text: id=1 to local app server
Sent message: text: id=2 to remote app server
ReplyListener: Received message: id=1, text=ReplyMsgBean processed message: text: id=1 to local app server
Sent message: text: id=3 to local app server
ReplyListener: Received message: id=3, text=ReplyMsgBean processed message: text: id=3 to local app server
ReplyListener: Received message: id=2, text=ReplyMsgBean processed message: text: id=2 to remote app server
Sent message: text: id=4 to remote app server
ReplyListener: Received message: id=4, text=ReplyMsgBean processed message: text: id=4 to remote app server
Sent message: text: id=5 to local app server
ReplyListener: Received message: id=5, text=ReplyMsgBean processed message: text: id=5 to local app server
Sent message: text: id=6 to remote app server
ReplyListener: Received message: id=6, text=ReplyMsgBean processed message: text: id=6 to remote app server
Sent message: text: id=7 to local app server
ReplyListener: Received message: id=7, text=ReplyMsgBean processed message: text: id=7 to local app server
Sent message: text: id=8 to remote app server
ReplyListener: Received message: id=8, text=ReplyMsgBean processed message: text: id=8 to remote app server
Sent message: text: id=9 to local app server
ReplyListener: Received message: id=9, text=ReplyMsgBean processed message: text: id=9 to local app server
Sent message: text: id=10 to remote app server
ReplyListener: Received message: id=10, text=ReplyMsgBean processed message: text: id=10 to remote app server
Waiting for 0 message(s) from local app server
Waiting for 0 message(s) from remote app server
Finished
Closing connection 1
Closing connection 2

On the local system, where the message-driven bean receives the odd-numbered messages, the output in the server log looks like this (wrapped in logging information):

ReplyMsgBean: Received message: text: id=1 to local app server
ReplyMsgBean: Received message: text: id=3 to local app server
ReplyMsgBean: Received message: text: id=5 to local app server
ReplyMsgBean: Received message: text: id=7 to local app server
ReplyMsgBean: Received message: text: id=9 to local app server

On the remote system, where the bean receives the even-numbered messages, the output in the server log looks like this (wrapped in logging information):

ReplyMsgBean: Received message: text: id=2 to remote app server
ReplyMsgBean: Received message: text: id=4 to remote app server
ReplyMsgBean: Received message: text: id=6 to remote app server
ReplyMsgBean: Received message: text: id=8 to remote app server
ReplyMsgBean: Received message: text: id=10 to remote app server

To Run the sendremote Example Using Ant

1. To package the modules, follow these steps:

   a. Go to the following directory:

      tut-install/examples/jms/sendremote/multiclient/

   b. Type the following command:

      ant

      This command creates a JAR file that contains the client class file and a manifest file.

   c. Change to the directory replybean:

      cd ../replybean

   d. Type the following command:

      ant

      This command creates a JAR file that contains the bean class file.

2. To deploy the replybean module on the local and remote servers, follow these steps:

   a. Verify that you are still in the directory replybean.
b. Type the following command:
   
   ```
   ant deploy
   ```
   Ignore the message that states that the application is deployed at a URL.

c. Type the following command:
   
   ```
   ant deploy-remote -Dsys=remote-system-name
   ```
   Replace `remote-system-name` with the actual name of the remote system.

3 To deploy the client, follow these steps:

a. Change to the directory `multiclient`:
   
   ```
   cd ../multiclient
   ```

b. Type the following command:
   
   ```
   ant getclient
   ```

4 To run the client, type the following command:
   
   ```
   ant run
   ```

On the local system, the output looks something like this:

```
running application client container.
...
Sent message: text: id=1 to local app server
Sent message: text: id=2 to remote app server
ReplyListener: Received message: id=1, text=ReplyMsgBean processed message: text: id=1 to local app server
Sent message: text: id=3 to local app server
ReplyListener: Received message: id=3, text=ReplyMsgBean processed message: text: id=3 to local app server
ReplyListener: Received message: id=2, text=ReplyMsgBean processed message: text: id=2 to remote app server
Sent message: text: id=4 to remote app server
ReplyListener: Received message: id=4, text=ReplyMsgBean processed message: text: id=4 to remote app server
Sent message: text: id=5 to local app server
ReplyListener: Received message: id=5, text=ReplyMsgBean processed message: text: id=5 to local app server
Sent message: text: id=6 to remote app server
ReplyListener: Received message: id=6, text=ReplyMsgBean processed message: text: id=6 to remote app server
Sent message: text: id=7 to local app server
ReplyListener: Received message: id=7, text=ReplyMsgBean processed message: text: id=7 to local app server
Sent message: text: id=8 to remote app server
ReplyListener: Received message: id=8, text=ReplyMsgBean processed message: text: id=8 to remote app server
Sent message: text: id=9 to local app server
ReplyListener: Received message: id=9, text=ReplyMsgBean processed message: text: id=9 to local app server
Sent message: text: id=10 to remote app server
```
On the local system, where the message-driven bean receives the odd-numbered messages, the output in the server log looks like this (wrapped in logging information):

ReplyMsgBean: Received message: text: id=1 to local app server
ReplyMsgBean: Received message: text: id=3 to local app server
ReplyMsgBean: Received message: text: id=5 to local app server
ReplyMsgBean: Received message: text: id=7 to local app server
ReplyMsgBean: Received message: text: id=9 to local app server

On the remote system, where the bean receives the even-numbered messages, the output in the server log looks like this (wrapped in logging information):

ReplyMsgBean: Received message: text: id=2 to remote app server
ReplyMsgBean: Received message: text: id=4 to remote app server
ReplyMsgBean: Received message: text: id=6 to remote app server
ReplyMsgBean: Received message: text: id=8 to remote app server
ReplyMsgBean: Received message: text: id=10 to remote app server

▼ To Disable Deployment on the Remote System

After running this example and undeploying the components, you should disable secure administration on the remote system (jupiter). In addition, you will probably want to return the GlassFish Server on jupiter to its previous state of not requiring a user name and password for administration, to make it easier to run subsequent examples there.

1 On the remote system (jupiter), start the Administration Console by opening a browser at http://localhost:4848/, if it is not already running.
   You will need to log in.

2 In the navigation tree, click the Server (Admin Server) node.

3 On the General Information page, click Secure Administration.

4 Click Disable Secure Admin.

5 The server will stop and restart automatically. Log in to the Administration Console again.

6 In the navigation tree, expand the Configurations node, then expand the server-config node.

7 Expand the Security node.
8 Expand the Realms node.
9 Select the admin-realm node.
10 On the Edit Realm page, click Manage Users.
11 In the File Users table, click admin in the User ID column.
12 On the Edit File Realm Users page, click Save.
13 In the dialog that asks you to confirm that you are setting an empty password for the specified user, click OK.

The next time you start the Administration Console or issue an asadmin command, you will not need to provide login credentials.

**Next Steps**

On earth, if you used NetBeans IDE to add the remote server, you may also want to remove the server.
Bean Validation: Advanced Topics

This chapter describes how to create custom constraints, custom validator messages, and constraint groups using the Java API for Java Beans Validation (Bean Validation).

The following topics are addressed here:
- “Creating Custom Constraints” on page 929
- “Customizing Validator Messages” on page 930
- “Grouping Constraints” on page 931

Creating Custom Constraints

Bean Validation defines annotations, interfaces, and classes to allow developers to create custom constraints.

Using the Built-In Constraints to Make a New Constraint

Bean Validation includes several built-in constraints that can be combined to create new, reusable constraints. This can simplify constraint definition by allowing developers to define a custom constraint made up of several built-in constraints that may then be applied to component attributes with a single annotation.

```java
@Pattern.List(
    @Pattern(regexp = "^[a-z0-9!#$%&'*+/=?^_\`~\-]+(?:\.[a-z0-9!#$%&'*+/=?^_\`~\-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\"),
    @Pattern(regexp = "^[a-z0-9!#$%&'*+/=?^_\`~\-]+\.[a-z0-9!#$%&'*+/=?^_\`~\-]+\.[a-z0-9]\")
)
@Constraint(validatedBy = {})
@Documented
@Target({ElementType.METHOD,
```
Customizing Validator Messages

Bean Validation includes a resource bundle of default messages for the built-in constraints. These messages can be customized, and localized for non-English speaking locales.

The ValidationMessages Resource Bundle

The ValidationMessages resource bundle and the locale variants of this resource bundle contain strings that override the default validation messages. The ValidationMessages resource bundle is typically a properties file, ValidationMessages.properties, in the default package of an application.

Localizing Validation Messages

Locale variants of ValidationMessages.properties are added by appending an underscore and the locale prefix to the base name of the file. For example, the Spanish locale variant resource bundle would be ValidationMessages_es.properties.
Grouping Constraints

Constraints may be added to one or more groups. Constraint groups are used to create subsets of constraints so only certain constraints will be validated for a particular object. By default, all constraints are included in the Default constraint group.

Constraint groups are represented by interfaces.

```java
public interface Employee {}
public interface Contractor {}
```

Constraint groups can inherit from other groups.

```java
public interface Manager extends Employee {}
```

When a constraint is added to an element, the constraint declares the groups to which that constraint belongs by specifying the class name of the group interface name in the `groups` element of the constraint.

```java
@NotNull(groups=Employee.class)
Phone workPhone;
```

Multiple groups can be declared by surrounding the groups with angle brackets (`{ and `}) and separating the groups’ class names with commas.

```java
@NotNull(groups={ Employee.class, Contractor.class })
Phone workPhone;
```

If a group inherits from another group, validating that group results in validating all constraints declared as part of the supergroup. For example, validating the `Manager` group results in the `workPhone` field being validated, because `Employee` is a superinterface of `Manager`.

Customizing Group Validation Order

By default, constraint groups are validated in no particular order. There are cases where some groups should be validated before others. For example, in a particular class, basic data should be validated before more advanced data.

To set the validation order for a group, add a `javax.validation.GroupSequence` annotation to the interface definition, listing the order in which the validation should occur.

```java
@GroupSequence({Default.class, ExpensiveValidationGroup.class})
public interface FullValidationGroup {}
```

When validating `FullValidationGroup`, first the `Default` group is validated. If all the data passes validation, then the `ExpensiveValidationGroup` group is validated. If a constraint is...
Grouping Constraints

part of both the Default and the ExpensiveValidationGroup groups, the constraint is validated as part of the Default group, and will not be validated on the subsequent ExpensiveValidationGroup pass.
Using Java EE Interceptors

This chapter discusses how to create interceptor classes and methods that interpose on method invocations or lifecycle events on a target class.

The following topics are addressed here:
- “Overview of Interceptors” on page 933
- “Using Interceptors” on page 935
- “The interceptor Example Application” on page 939

Overview of Interceptors

Interceptors are used in conjunction with Java EE managed classes to allow developers to invoke interceptor methods on an associated target class, in conjunction with method invocations or lifecycle events. Common uses of interceptors are logging, auditing, and profiling.


An interceptor can be defined within a target class as an interceptor method, or in an associated class called an interceptor class. Interceptor classes contain methods that are invoked in conjunction with the methods or lifecycle events of the target class.

Interceptor classes and methods are defined using metadata annotations, or in the deployment descriptor of the application containing the interceptors and target classes.

Note – Applications that use the deployment descriptor to define interceptors are not portable across Java EE servers.

Interceptor methods within the target class or in an interceptor class are annotated with one of the metadata annotations defined in Table 50–1.
### Interceptor Metadata Annotations

<table>
<thead>
<tr>
<th>Interceptor Metadata Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>javax.interceptor.AroundInvoke</code></td>
<td>Designates the method as an interceptor method.</td>
</tr>
<tr>
<td><code>javax.interceptor.AroundTimeout</code></td>
<td>Designates the method as a timeout interceptor, for interposing on timeout methods for enterprise bean timers.</td>
</tr>
<tr>
<td><code>javax.annotation.PostConstruct</code></td>
<td>Designates the method as an interceptor method for post-construct lifecycle events.</td>
</tr>
<tr>
<td><code>javax.annotation.PreDestroy</code></td>
<td>Designates the method as an interceptor method for pre-destroy lifecycle events.</td>
</tr>
</tbody>
</table>

### Interceptor Classes

Interceptor classes may be designated with the optional `javax.interceptor.Interceptor` annotation, but interceptor classes aren’t required to be so annotated. An interceptor class must have a public, no-argument constructor.

The target class can have any number of interceptor classes associated with it. The order in which the interceptor classes are invoked is determined by the order in which the interceptor classes are defined in the `javax.interceptor.Interceptors` annotation. However, this order can be overridden in the deployment descriptor.

Interceptor classes may be targets of dependency injection. Dependency injection occurs when the interceptor class instance is created, using the naming context of the associated target class, and before any `@PostConstruct` callbacks are invoked.

### Interceptor Lifecycle

Interceptor classes have the same lifecycle as their associated target class. When a target class instance is created, an interceptor class instance is also created for each declared interceptor class in the target class. That is, if the target class declares multiple interceptor classes, an instance of each class is created when the target class instance is created. The target class instance and all interceptor class instances are fully instantiated before any `@PostConstruct` callbacks are invoked, and any `@PreDestroy` callbacks are invoked before the target class and interceptor class instances are destroyed.
Interceptors and CDI

Contexts and Dependency Injection for the Java EE Platform (CDI) builds on the basic functionality of Java EE interceptors. For information on CDI interceptors, including a discussion of interceptor binding types, see “Using Interceptors in CDI Applications” on page 547.

Using Interceptors

An interceptor is defined using one of the interceptor metadata annotations listed in Table 50–1 within the target class, or in a separate interceptor class. The following code declares an @AroundTimeout interceptor method within a target class.

```java
@Stateless
public class TimerBean {
    ...
    @Schedule(minute="*/1", hour="*")
    public void automaticTimerMethod() {
        ...
    }
    @AroundTimeout
    public void timeoutInterceptorMethod(InvocationContext ctx) {
        ...
    }
    ...
}
```

If interceptor classes are used, use the `javax.interceptor.Interceptors` annotation to declare one or more interceptors at the class or method level of the target class. The following code declares interceptors at the class level.

```java
@Stateless
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
public class OrderBean { ... }
```

The following code declares a method-level interceptor class.

```java
@Stateless
public class OrderBean { ...
    @Interceptors(OrderInterceptor.class)
    public void placeOrder(Order order) {
        ...
    }
    ...
}
```
Intercepting Method Invocations

The @AroundInvoke annotation is used to designate interceptor methods for managed object methods. Only one around-invoked interceptor method per class is allowed. Around-invoked interceptor methods have the following form:

```java
@AroundInvoke
visibility Object method-name(InvocationContext) throws Exception { ... }
```

For example:

```java
@AroundInvoke
public void interceptOrder(InvocationContext ctx) { ... }
```

Around-invoked interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

An around-invoked interceptor can call any component or resource callable by the target method on which it interposes, have the same security and transaction context as the target method, and run in the same Java virtual machine call-stack as the target method.

Around-invoked interceptors can throw any exception allowed by the throws clause of the target method. They may catch and suppress exceptions, and then recover by calling the InvocationContext.proceed method.

Using Multiple Method Interceptors

Use the @Interceptors annotation to declare multiple interceptors for a target method or class.

```java
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
               LastInterceptor.class})
public void updateInfo(String info) { ... }
```

The order of the interceptors in the @Interceptors annotation is the order in which the interceptors are invoked.

Multiple interceptors may also be defined in the deployment descriptor. The order of the interceptors in the deployment descriptor is the order in which the interceptors will be invoked.

```xml
<interceptor-binding>
  <target-name>myapp.OrderBean</target-name>
  <interceptor-class>myapp.PrimaryInterceptor.class</interceptor-class>
  <interceptor-class>myapp.SecondaryInterceptor.class</interceptor-class>
  <interceptor-class>myapp.LastInterceptor.class</interceptor-class>
  <method-name>updateInfo</method-name>
</interceptor-binding>
```

To explicitly pass control to the next interceptor in the chain, call the InvocationContext.proceed method.
Data can be shared across interceptors:

- The same InvocationContext instance is passed as an input parameter to each interceptor method in the interceptor chain for a particular target method. The InvocationContext instance’s contextData property is used to pass data across interceptor methods. The contextData property is a java.util.Map<String, Object> object. Data stored in contextData is accessible to interceptor methods further down the interceptor chain.
- The data stored in contextData is not sharable across separate target class method invocations. That is, a different InvocationContext object is created for each invocation of the method in the target class.

### Accessing Target Method Parameters From an Interceptor Class

The InvocationContext instance passed to each around-invoke method may be used to access and modify the parameters of the target method. The parameters property of InvocationContext is an array of Object instances that corresponds to the parameter order of the target method. For example, for the following target method, the parameters property, in the InvocationContext instance passed to the around-invoke interceptor method in PrimaryInterceptor, is an Object array containing two String objects (firstName and lastName) and a Date object (date):

```java
@Interceptors(PrimaryInterceptor.class)
public void updateInfo(String firstName, String lastName, Date date) { ... }
```

The parameters can be accessed and modified using the InvocationContext.getParameters and InvocationContext.setParameters methods, respectively.

### Intercepting Lifecycle Callback Events

Interceptors for lifecycle callback events (post-create and pre-destroy) may be defined in the target class or in interceptor classes. The @PostCreate annotation is used to designate a method as a post-create lifecycle event interceptor. The @PreDestroy annotation is used to designate a method as a pre-destroy lifecycle event interceptor.

Lifecycle event interceptors defined within the target class have the following form:

```java
void method-name() { ... }
```
For example:

```java
@PostCreate
void initialize() { ... }
```

Lifecycle event interceptors defined in an interceptor class have the following form:

```java
void <method-name>(InvocationContext) { ... }
```

For example:

```java
@PreDestroy
void cleanup(InvocationContext ctx) { ... }
```

Lifecycle interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

Lifecycle interceptor methods are called in an unspecified security and transaction context. That is, portable Java EE applications should not assume the lifecycle event interceptor method has access to a security or transaction context. Only one interceptor method for each lifecycle event (post-create and pre-destroy) is allowed per class.

### Using Multiple Lifecycle Callback Interceptors

Multiple lifecycle interceptors may be defined for a target class by specifying the interceptor classes in the `@Interceptors` annotation:

```java
@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class,
               LastInterceptor.class})
@Stateless
public class OrderBean { ... }
```

The order in which the interceptor classes are listed in the `@Interceptors` annotation defines the order in which the interceptors are invoked.

Data stored in the `contextData` property of `InvocationContext` is not sharable across different lifecycle events.

### Intercepting Timeout Events

Interceptors for EJB timer service timeout methods may be defined using the `@AroundTimeout` annotation on methods in the target class or in an interceptor class. Only one `@AroundTimeout` method per class is allowed.

Timeout interceptors have the following form:

```java
Object <method-name>(InvocationContext) throws Exception { ... }
```

For example:
@AroundTimeout
protected void timeoutInterceptorMethod(InvocationContext ctx) { ... }

Timeout interceptor methods can have public, private, protected, or package-level access, and
must not be declared static or final.

Timeout interceptors can call any component or resource callable by the target timeout
method, and are invoked in the same transaction and security context as the target method.

Timeout interceptors may access the timer object associated with the target timeout method
through the InvocationContext instance's getTimer method.

Using Multiple Timeout Interceptors

Multiple timeout interceptors may be defined for a given target class by specifying the
interceptor classes containing @AroundTimeout interceptor methods in an @Interceptors
annotation at the class level.

If a target class specifies timeout interceptors in an interceptor class, and also has a
@AroundTimeout interceptor method within the target class itself, the timeout interceptors in
the interceptor classes are called first, followed by the timeout interceptors defined in the target
class. For example, in the following example, assume that both the PrimaryInterceptor and
SecondaryInterceptor classes have timeout interceptor methods.

@Interceptors({PrimaryInterceptor.class, SecondaryInterceptor.class})
@Stateful
public class OrderBean {
  ...
  @AroundTimeout
  private void last(InvocationContext ctx) { ... }
  ...
}

The timeout interceptor in PrimaryInterceptor will be called first, followed by the timeout
interceptor in SecondaryInterceptor, and finally the last method defined in the target class.

The interceptor Example Application

The interceptor example demonstrates how to use an interceptor class, containing an
@AroundInvoke interceptor method, with a stateless session bean.

The HelloBean stateless session bean is a simple enterprise bean with two business methods,
getName and setName, to retrieve and modify a string. The setName business method has an
@Interceptors annotation that specifies an interceptor class, HelloInterceptor, for that
method.

@Interceptors(HelloInterceptor.class)
public void setName(String name) {
  this.name = name;
}
The HelloInterceptor class defines an @AroundInvoke interceptor method, modifyGreeting, that converts the string passed to HelloBean.setName to lowercase.

@AroundInvoke
public Object modifyGreeting(InvocationContext ctx) throws Exception {
    Object[] parameters = ctx.getParameters();
    String param = (String) parameters[0];
    param = param.toLowerCase();
    parameters[0] = param;
    ctx.setParameters(parameters);
    try {
        return ctx.proceed();
    } catch (Exception e) {
        logger.warning("Error calling ctx.proceed in modifyGreeting()");
        return null;
    }
}

The parameters to HelloBean.setName are retrieved and stored in an Object array by calling the InvocationContext.getParameters method. Because setName has only one parameter, it is the first and only element in the array. The string is set to lowercase and stored in the parameters array, then passed to InvocationContext.setParameters. To return control to the session bean, InvocationContext.proceed is called.

The user interface of interceptor is a JavaServer Faces web application that consists of two Facelets views: index.xhtml, which contains a form for entering the name, and response.xhtml, which displays the final name.

Running the interceptor Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the interceptor example.

▼ To Run the interceptor Example Using NetBeans IDE

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to tut-install/examples/eb/.

3 Select the interceptor folder and click Open Project.

4 In the Projects tab, right-click the interceptor project and select Run.

This will compile, deploy, and run the interceptor example, opening a web browser page to http://localhost:8080/interceptor/.
5 Type a name into the form and select Submit.  
The name will be converted to lowercase by the method interceptor defined in the HelloInterceptor class.

▼ To Run the interceptor Example Using Ant

1 Go to the following directory:  
tut-install/examples/ejb/interceptor/

2 To compile the source files and package the application, use the following command:  
ant  
This command calls the default target, which builds and packages the application into a WAR file, interceptor.war, located in the dist directory.

3 To deploy and run the application using Ant, use the following command:  
ant run  
This command deploys and runs the interceptor example, opening a web browser page to http://localhost:8080/interceptor/.

4 Type a name into the form and select Submit.  
The name will be converted to lowercase by the method interceptor defined in the HelloInterceptor class.
PART IX

Case Studies

Part IX presents case studies that use a variety of Java EE technologies. This part contains the following chapters:

- Chapter 51, “Duke’s Bookstore Case Study Example”
- Chapter 52, “Duke’s Tutoring Case Study Example”
- Chapter 53, “Duke’s Forest Case Study Example”
Duke’s Bookstore Case Study Example

The Duke’s Bookstore example is a simple e-commerce application that illustrates some of the more advanced features of JavaServer Faces technology in combination with Contexts and Dependency Injection for the Java EE Platform (CDI), enterprise beans, and the Java Persistence API. Users can select books from an image map, view the bookstore catalog, and purchase books. No security is used in this application.

The following topics are addressed here:

- “Design and Architecture of Duke’s Bookstore” on page 945
- “The Duke’s Bookstore Interface” on page 946
- “Running the Duke’s Bookstore Case Study Application” on page 951

Design and Architecture of Duke’s Bookstore

Duke’s Bookstore is a simple web application that uses many features of JavaServer Faces technology, in addition to other Java EE 6 features:

- JavaServer Faces technology, as well as Contexts and Dependency Injection for the Java EE Platform (CDI):
  - A set of Facelets pages, along with a template, provides the user interface to the application.
  - CDI managed beans are associated with each of the Facelets pages.
  - A custom image map component on the front page allows you to select a book to enter the store. Each area of the map is represented by a JavaServer Faces managed bean. Text hyperlinks are also provided for accessibility.
  - Action listeners are registered on the image map and the text hyperlinks. These listeners retrieve the ID value for the selected book and store it in the session map so it can be retrieved by the managed bean for the next page.
  - The h:dataTable tag is used to render the book catalog and shopping cart contents dynamically.
A custom converter is registered on the credit card field on the checkout page, `bookcashier.xhtml`, which also uses an `f:validateRegExp` tag to ensure that the input is correctly formatted.

A value-change listener is registered on the name field on `bookcashier.xhtml`. This listener saves the name in a parameter so the following page, `bookreceipt.xhtml`, can access it.

- Enterprise beans: Local, no-interface-view stateless session bean and singleton bean
- A Java Persistence API entity

The packages of the Duke's Bookstore application, located in the `tut-install/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/` directory, are as follows:

- components: Includes the custom UI component classes, `MapComponent` and `AreaComponent`
- converters: Includes the custom converter class, `CreditCardConverter`
- ejb: Includes two enterprise beans:
  - A singleton bean, `ConfigBean`, that initializes the data in the database
  - A stateless session bean, `BookRequestBean`, that contains the business logic to manage the entity
- entity: Includes the `Book` entity class
- exceptions: Includes three exception classes
- listeners: Includes the event handler and event listener classes
- model: Includes a model `JavaBeans` class
- renderers: Includes the custom renderers for the custom UI component classes
- web.managedbeans: Includes the managed beans for the Facelets pages
- web.messages: Includes the resource bundle files for localized messages

The Duke's Bookstore Interface

This section provides additional detail regarding the components of the Duke's Bookstore example and how they interact.

The Book Java Persistence API Entity

The `Book` entity, located in the `dukesbookstore.entity` package, encapsulates the book data stored by Duke's Bookstore.
The Book entity defines attributes used in the example:

- A book ID
- The author's first name
- The author's surname
- The title
- The price
- Whether the book is on sale
- The publication year
- A description of the book
- The number of copies in the inventory

The Book entity also defines a simple named query, `findBooks`.

**Enterprise Beans Used in Duke's Bookstore**

Two enterprise beans located in the `dukesbookstore.ejb` package provide the business logic for Duke's Bookstore.

`BookRequestBean` is a stateless session bean that contains the business methods for the application. The methods create, retrieve, and purchase books, and update the inventory for a book. To retrieve the books, the `getBooks` method calls the `findBooks` named query defined in the Book entity.

`ConfigBean` is a singleton session bean used to create the books in the catalog when the application is initially deployed. It calls the `createBook` method defined in `BookRequestBean`.

**Facelets Pages and Managed Beans Used in Duke's Bookstore**

The Duke's Bookstore application uses Facelets and its templating features to display the user interface. The Facelets pages interact with a set of CDI managed beans that provide the underlying properties and methods for the user interface. The front page also interacts with the custom components used by the application.

The application uses the following Facelets pages, which are located in the `tut-install/examples/case-studies/dukes-bookstore/web/` directory:

- `bookstoreTemplate.xhtml` - The template file, which specifies a header used on every page as well as the style sheet used by all the pages. The template also retrieves the language set in the web browser.

  Uses the `LocaleBean` managed bean.
index.xhtml
Landing page, which lays out the custom map and area components using managed beans configured in the faces-config.xml file, and allows the user to select a book and advance to the bookstore.xhtml page.

bookstore.xhtml
Page that allows the user to obtain details on the selected book or the featured book, to add either book to the shopping cart, and to advance to the bookcatalog.xhtml page.

Uses the BookstoreBean managed bean.

bookdetails.xhtml
Page that shows details on a book selected from bookstore.xhtml or other pages and allows the user to add the book to the cart and/or advance to the bookcatalog.xhtml.

Uses the BookDetailsBean managed bean.

bookcatalog.xhtml
Page that displays the books in the catalog and allows the user to add books to the shopping cart, view the details for any book, view the shopping cart, empty the shopping cart, or purchase the books in the shopping cart.

Uses the CatalogBean and ShoppingCart managed beans.

bookshowcart.xhtml
Page that displays the contents of the shopping cart and allows the user to remove items, view the details for an item, empty the shopping cart, purchase the books in the shopping cart, or return to the catalog.

Uses the ShowCartBean and ShoppingCart managed beans.

bookcashier.xhtml
Page that allows the user to purchase books, specify a shipping option, subscribe to newsletters, or join the Duke Fan Club with a purchase over a certain amount.

Uses the CashierBean and ShoppingCart managed beans.

bookreceipt.xhtml
Page that confirms the user's purchase and allows the user to return to the catalog page to continue shopping.

Uses the CashierBean managed bean.

In addition to the managed beans used by the Facelets template and pages, the application uses the following managed beans:

AbstractBean
Contains utility methods called by other managed beans.

ShoppingCartItem
Contains methods called by ShoppingCart, CatalogBean, and ShowCartBean.
Custom Components and Other Custom Objects Used in Duke's Bookstore

The map and area custom components for Duke's Bookstore, along with associated renderer, listener, and model classes, are defined in the following packages in the `tut-install/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/` directory:

- **components**
  Contains the `MapComponent` and `AreaComponent` classes. See “Creating Custom Component Classes” on page 262.

- **listeners**
  Contains the `AreaSelectedEvent` class, along with other listener classes. See “Handling Events for Custom Components” on page 275.

- **model**
  Contains the `ImageArea` class. See “Configuring Model Data” on page 259.

- **renderers**
  Contains the `MapRenderer` and `AreaRenderer` classes. See “Delegating Rendering to a Renderer” on page 270.

The `tut-install/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/` directory also contains a custom converter and other custom listeners not specifically tied to the custom components:

- **converters**
  Contains the `CreditCardConverter` class. See “Creating and Using a Custom Converter” on page 279.

- **listeners**
  Contains the `LinkBookChangeListener`, `MapBookChangeListener`, and `NameChanged` classes. See “Implementing an Event Listener” on page 273.

Properties Files Used in Duke's Bookstore

The strings used in the Duke's Bookstore application are encapsulated into resource bundles to allow the display of localized strings in multiple locales. The properties files, located in the `tut-install/examples/case-studies/dukes-bookstore/src/java/dukesbookstore/web/messages/` directory, consist of a default file containing English strings and three additional files for other locales. The files are as follows:

- **Messages.properties**
  Default file, containing English strings

- **Messages_de.properties**
  File containing German strings

- **Messages_es.properties**
  File containing Spanish strings

- **Messages_fr.properties**
  File containing French strings
The language setting in the user's web browser determines which locale is used. The \texttt{html} tag in \texttt{bookstoreTemplate.xhtml} retrieves the language setting from the \texttt{language} property of \texttt{LocaleBean}:

\begin{verbatim}
<html lang="#{localeBean.language}"
   ...
\end{verbatim}

For more information about resource bundles, see Chapter 17, "Internationalizing and Localizing Web Applications."

The resource bundle is configured as follows in the \texttt{faces-config.xml} file:

\begin{verbatim}
<application>
    <resource-bundle>
        <base-name>dukesbookstore.web.messages.Messages</base-name>
        <var>bundle</var>
    </resource-bundle>
    <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>fr</supported-locale>
        <supported-locale>es</supported-locale>
    </locale-config>
</application>
\end{verbatim}

This configuration means that in the Facelets pages, messages are retrieved using the prefix \texttt{bundle} with the key found in the \texttt{Messages\_locale\_messages.properties} file, as in the following example from the \texttt{index.xhtml} page:

\begin{verbatim}
<h:outputText style="font-weight:bold" value="#{bundle.ChooseBook}" />
\end{verbatim}

In \texttt{Messages\_properties}, the key string is defined as follows:

\begin{verbatim}
ChooseBook=Choose a Book from our Catalog
\end{verbatim}

**Deployment Descriptors Used in Duke's Bookstore**

The following deployment descriptors are used in Duke's Bookstore:

- \texttt{src/conf/persistence.xml} - The Java Persistence API configuration file
- \texttt{web/WEB-INF/beans.xml} - An empty deployment descriptor file used to enable the CDI runtime
- \texttt{web/WEB-INF/bookstore.taglib.xml} - The tag library descriptor file for the custom components
Running the Duke’s Bookstore Case Study Application

You can use either NetBeans IDE or Ant to build, package, deploy, and run the Duke’s Bookstore application.

▼ To Build and Deploy Duke’s Bookstore Using NetBeans IDE

Before You Begin
You must have already configured GlassFish Server as a Java EE server in NetBeans IDE, as described in “To Add GlassFish Server as a Server in NetBeans IDE” on page 72.

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to:
   tut-install/examples/case-studies/

3 Select the Open as Main Project check box.

4 Click Open Project.

5 Right-click dukes-bookstore in the project pane and select Deploy.
   This will build, package, and deploy Duke’s Bookstore to the GlassFish Server, starting the Java DB database and GlassFish Server if they have not already been started.

▼ To Build and Deploy Duke’s Bookstore Using Ant

Before You Begin
Make sure the GlassFish Server is started as described in “Starting and Stopping the GlassFish Server” on page 73 and the Java DB server is started as described in “Starting and Stopping the Java DB Server” on page 75.
Running the Duke's Bookstore Case Study Application

1. In a terminal window, go to:
   tut-install/examples/case-studies/dukes-bookstore/

2. Type the following command:
   
   ```
   ant all
   ```

   This command builds, packages, and deploys Duke's Bookstore to the GlassFish Server.

▼ To Run Duke's Bookstore

1. In a web browser, type the following URL:
   
   ```
   http://localhost:8080/dukesbookstore/
   ```

2. On the Duke's Bookstore main page, click a book in the graphic, or click one of the links at the bottom of the page.

3. Use the pages in the application to view and purchase books.
Duke’s Tutoring Case Study Example

The Duke’s Tutoring example application is a tracking system for a tutoring center for students. Students or their guardians can check in and out. The tutoring center can track attendance and status updates and can store contact information for guardians and students.

The following topics are addressed here:
- “Design and Architecture of Duke’s Tutoring” on page 953
- “Main Interface” on page 955
- “Administration Interface” on page 959
- “Running the Duke’s Tutoring Case Study Application” on page 960

Design and Architecture of Duke’s Tutoring

Duke’s Tutoring is a web application that incorporates several Java EE technologies. It exposes both a main interface (for students, guardians, and tutoring center staff) and an administration interface (for staff to maintain the system). The business logic for both interfaces is provided by enterprise beans. The enterprise beans use the Java Persistence API to create and store the application’s data in the database. Figure 52–1 illustrates the architecture of the application.
The Duke’s Tutoring application is organized into two main projects, the dukes-tutoring-common library, and the dukes-tutoring-war web application. The dukes-tutoring-common library project contains the entity classes and helper classes used by the dukes-tutoring-war web application, and dukes-tutoring-common is packaged and deployed with dukes-tutoring-war. The library JAR file is useful for allowing the entity classes and helper classes to be reused by other applications, such as a JavaFX client application.

Duke’s Tutoring uses the following Java EE 6 platform features:

- Java Persistence API entities
  - Java API for JavaBeans Validation (Bean Validation) annotations on the entities for verifying data
  - A custom Bean Validation annotation, @Email, for validating email addresses
- Enterprise beans
  - Local, no-interface-view session and singleton beans
  - JAX-RS resources in a session bean
  - Java EE security constraints on the administrative interface business methods
  - All enterprise beans packaged within the WAR
- JavaServer Faces technology, using Facelets for the web front end
The Duke's Tutoring application has two main user interfaces, both packaged within a single WAR file:

- The main interface, for students, guardians, and staff
- The administrative interface used by the staff to manage the students and guardians, and to generate attendance reports

Apart from the main and administrative interfaces, there is a JUnit test that demonstrates how to use the embedded EJB container to test the business logic of the session beans.

**Main Interface**

The main interface allows students and staff to check students in and out, and record when students are outside at the playground.

**Java Persistence API Entities Used in the Main Interface**

The entities used in the main interface encapsulate data stored and manipulated by Duke's Tutoring, and are located in the dukestutoring.entity package in the dukes-tutoring-common project.

The Person entity defines attributes common to students, guardians, and administrators tracked by the application. These attributes are the person's name and contact information, including phone numbers and email address. The phone number and email address attributes have Bean Validation annotations to ensure that the submitted data is well-formed. The email attribute uses a custom validation class, dukestutoring.util.Email. The Person entity has three subclasses, Student, Guardian, and Administrator. For additional data common to all people, the PersonDetails entity is used to store attributes like pictures and the person's birthday, which aren't included in the Person entity for performance reasons.

The Student entity stores attributes specific to the students who come to tutoring. This includes information like the student's grade level and school. The Guardian entity's attributes are specific to the parents or guardians of a Student. Students and guardians have a many-to-many relationship. That is, a student may have a one or more guardians, and a guardian may have one or more students. The Administrator entity is for staff who manage the tutoring center.
The Address entity represents a mailing address, and is associated with Person entities. Addresses and people have a many-to-one relationship. That is, one person may have many addresses.

The TutoringSession entity represents a particular day at the tutoring center. A particular tutoring session tracks which students attended that day, and which students went to the park. Associated with TutoringSession is the StatusEntry entity, which logs when a student's status changes. Students' status changes when they check in to a tutoring session, when they go to the park, and when they check out. The status entry allows the tutoring center staff to track exactly which students attended a tutoring session, when they checked in and out, which students went to the park while they were at the tutoring center, and when they went to and came back from the park.

For information on creating Java Persistence API entities, see Chapter 32, "Introduction to the Java Persistence API." For information on validating entity data, see "Validating Persistent Fields and Properties" on page 583 and Chapter 49, "Bean Validation: Advanced Topics."

**Enterprise Beans Used in the Main Interface**

The enterprise beans used in the main interface provide the business logic for Duke's Tutoring, and are located in the dukestutoring.ejb package in the dukes-tutoring-war project.

ConfigBean is singleton session bean used to create the default students, guardians, and administrator when the application is initially deployed, and to create an automatic EJB timer that creates tutoring session entities every weekday.

RequestBean is a stateless session bean containing the business methods for the main interface. Students or staff can check students in and out and track when they go to and come back from the park. The bean also has business methods for retrieving lists of students. The business methods in RequestBean use strongly-typed Criteria API queries to retrieve data from the database.

For information on creating and using enterprise beans, see Part IV, "Enterprise Beans." For information on creating strongly-typed Criteria API queries, see Chapter 35, "Using the Criteria API to Create Queries."

**Facelets Files Used in the Main Interface**

The Duke's Tutoring application uses Facelets to display the user interface, and makes extensive use of the templating features of Facelets. Facelets is the default display technology for JavaServer Faces, and consists of XHTML files located in the tut-install/examples/case-studies/dukes-tutoring/dukes-tutoring-war/web/ directory.

The following Facelets files are used in the main interface:
template.xhtml
    Template file for the main interface
error.xhtml
    Error file employed if something goes wrong
index.xhtml
    Landing page for the main interface
park.xhtml
    Page showing who is currently at the park
current.xhtml
    Page showing who is currently in today's tutoring session
statusEntries.xhtml
    Page showing the detailed status entry log for today's session
resources/components/allStudentsTable.xhtml
    A composite component for a table displaying all active students
resources/components/currentSessionTable.xhtml
    A composite component for a table displaying all students in today's session
resources/components/parkTable.xhtml
    A composite component for a table displaying all students currently at the park
WEB-INF/includes/navigation.xhtml
    XHTML fragment for the main interface’s navigation bar
WEB-INF/includes/footer.xhtml
    XHTML fragment for the main interface's footer

For information on using Facelets, see Chapter 5, “Introduction to Facelets.”

**Helper Classes Used in the Main Interface**

The following helper classes, found in the dukes-tutoring-common project’s dukestutoring.util package, are used in the main interface:

- `CalendarUtil`: A class that provides a method to strip the unnecessary time data from java.util.Calendar instances
- `Email`: A custom Bean Validation annotation class for validating email addresses in the Person entity
- `StatusType`: An enumerated type defining the different statuses that a student can have. Possible values are IN, OUT, and PARK. StatusType is used throughout the application, including in the StatusEntry entity, and throughout the main interface. StatusType also defines a toString method that returns a
The following helper classes, found in the dukes-tutoring-war project’s dukestutoring.web.util package, are used in the JavaServer Faces application:

**EntityConverter**
A parent class to StudentConverter and GuardianConverter that defines a cache to store the entity classes when converting the entities for use in JavaServer Faces user-interface components. The cache helps increase performance. The cache is stored in the JavaServer Faces context.

**StudentConverter**
A JavaServer Faces converter for the Student entity class. This class contains methods to convert Student instances to strings and back again, so they can be used in the user-interface components of the application.

**GuardianConverter**
Similar to StudentConverter, this class is a converter for the Guardian entity class.

### Properties Files

The strings used in the main interface are encapsulated into resource bundles to allow the display of localized strings in multiple locales. Each of the properties files has locale-specific files appended with locale codes, containing the translated strings for each locale. For example, Messages_es.properties contains the localized strings for Spanish locales.

The dukestutoring.util package in the dukes-tutoring-common project has the following resource bundle:

**StatusMessages**
Strings for each of the status types defined in the StatusType enumerated type for the default locale. Each supported locale has a property file of the form StatusMessages_locale_prefix.properties containing the localized strings. For example, the strings for Spanish-speaking locales are located in StatusMessages_es.properties.

The dukes-tutoring-war project has the following resource bundles:

**ValidationMessages.properties**
Strings for the default locale used by the Bean Validation runtime to display validation messages. This file must be named ValidationMessages.properties and located in the default package as required by the Bean Validation specification. Each supported locale has a property file of the form ValidationMessages_locale_prefix.properties containing the localized strings. For example, the strings for German-speaking locales are located in ValidationMessages_de.properties.
dukestutoring/web/messages/Messages.properties
Strings for the default locale for the main and administration Facelets interfaces. Each supported locale has a property file of the form Messages_locale_prefix.properties containing the localized strings. For example, the strings for simplified Chinese-speaking locales are located in Messages_zh.properties.

For information on localizing web applications, see “Registering Application Messages” on page 308.

**Deployment Descriptors Used in Duke’s Tutoring**

The following deployment descriptors in the dukes-tutoring-war project are used in Duke’s Tutoring:

- src/conf/beans.xml An empty deployment descriptor file used to enable the CDI runtime
- web/WEB-INF/faces-config.xml The JavaServer Faces configuration file
- web/WEB-INF/glassfish-web.xml The GlassFish-specific configuration file
- web/WEB-INF/web.xml The web application configuration file

The following deployment descriptor in the dukes-tutoring-common project is used in Duke’s Tutoring:

- src/META-INF/persistence.xml The Java Persistence API configuration file

No enterprise bean deployment descriptor is used in Duke’s Tutoring. Annotations in the enterprise bean class files are used for the configuration of enterprise beans in this application.

**Administration Interface**

The administration interface of Duke’s Tutoring is used by the tutoring center staff to manage the data employed by the main interface: the students, the students’ guardians, and the addresses. The administration interface uses many of the same components as the main interface. Additional components that are only used in the administration interface are described here.

**Enterprise Beans Used in the Administration Interface**

The following enterprise beans, in the dukestutoring.ejb package, are used in the administration interface:
AdminBean  A stateless session bean for all the business logic used in the administration interface. Contains security constraint annotations to allow invocation of the business methods only by authorized users.

Facelets Files Used in the Administration Interface

The following Facelets files are used in the administration interface:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin/adminTemplate.xhtml</td>
<td>Template for the administration interface</td>
</tr>
<tr>
<td>admin/index.xhtml</td>
<td>Landing page for the administration interface</td>
</tr>
<tr>
<td>admin/login.xhtml</td>
<td>Login page for the security-constrained administration interface</td>
</tr>
<tr>
<td>admin/loginError.xhtml</td>
<td>Page displayed if there are errors authenticating the administration user</td>
</tr>
<tr>
<td>admin/address directory</td>
<td>Pages that allow you to create, edit, and delete Address entities</td>
</tr>
<tr>
<td>admin/guardian directory</td>
<td>Pages that allow you to create, edit, and delete Guardian entities</td>
</tr>
<tr>
<td>admin/student directory</td>
<td>Pages that allow you to create, edit, and delete Student entities</td>
</tr>
<tr>
<td>resources/components/formLogin.xhtml</td>
<td>Composite component for a login form using Java EE security</td>
</tr>
<tr>
<td>WEB-INF/includes/adminNav.xhtml</td>
<td>XHTML fragment for the administration interface's navigation bar</td>
</tr>
</tbody>
</table>

Running the Duke’s Tutoring Case Study Application

This section describes how to build, package, deploy, and run the Duke's Tutoring application.

Setting Up GlassFish Server

Before running the Duke’s Tutoring application, set up the security realm used by Duke’s Tutoring with users and groups. The user names and passwords set in this security realm are used to log in to the administration interface of Duke’s Tutoring.

Duke’s Tutoring’s security realm maps members of the Administrator entity to the Administrator role used in the security constraint annotations in AdminBean.
To Create the JDBC Realm in GlassFish Server

Create the tutoringRealm JDBC security realm in GlassFish Server.

Before You Begin

Make sure GlassFish Server is started as described in "Starting and Stopping the GlassFish Server" on page 73, and JavaDB is started as described in "Starting and Stopping the JavaDB Server" on page 75.

1 In a terminal window, go to:
   tut-install/examples/case-studies/dukes-tutoring/dukes-tutoring-war/

2 Enter the following command:
   ant create-tutoring-realm
   This target creates a JDBC realm using the jdbc/tutoring JDBC resource, which will be created when dukes-tutoring-war has been deployed.

Running Duke’s Tutoring

You can use either NetBeans IDE or Ant to build, package, deploy, and run Duke’s Tutoring.

To Build and Deploy Duke’s Tutoring in NetBeans IDE

You must have already configured GlassFish Server as a Java EE server in NetBeans IDE, as described in "To Add GlassFish Server as a Server in NetBeans IDE" on page 72.

1 From the File menu, choose Open Project.

2 In the Open Project dialog, navigate to:
   tut-install/examples/case-studies/dukes-tutoring/

3 Select the dukes-tutoring-war folder.

4 Select the Open as Main Project check box and the Open Required Projects check box.
   The dukes-tutoring-common library project is required by dukes-tutoring-war, and will be opened along with dukes-tutoring-war.

5 Click Open Project.

Note – The first time you open Duke’s Tutoring in NetBeans, you will see error glyphs in the project pane. This is expected, as the metamodel files used by the enterprise beans for Criteria API queries have not yet been generated.
6 Right-click dukes-tutoring-war in the project pane and select Run.
   This will build and package the dukes-tutoring-common and dukes-tutoring-war projects
   and deploy dukes-tutoring-war to GlassFish Server, starting the Java DB database and
   GlassFish Server if they have not already been started. The jdbc/tutoring JDBC resource will
   be created at deploy time. After the application has been successfully deployed, the Duke's
   Tutoring main interface will open in a web browser if NetBeans IDE has been configured to
   open web applications in a web browser.

To Build and Deploy Duke's Tutoring Using Ant

Before You Begin
Make sure GlassFish Server is started as described in "Starting and Stopping the GlassFish
Server" on page 73, and Java DB server is started as described in "Starting and Stopping the Java
DB Server" on page 75.

1 In a terminal window, go to:
   tut-install/examples/case-studies/dukes-tutoring/dukes-tutoring-war/

2 Enter the following command:
   ant all
   This command builds and packages the dukes-tutoring-common and dukes-tutoring-war
   projects, and deploys dukes-tutoring-war to GlassFish Server.

Using Duke's Tutoring

Once Duke's Tutoring is running on GlassFish Server, use the main interface to experiment
with checking students in and out or sending them to the park.

To Use the Main Interface of Duke's Tutoring

1 In a web browser, open the main interface at the following URL:
   http://localhost:8080/dukes-tutoring/

2 Use the main interface to check students in and out, and to log when the students go to the
   park.

To Use the Administration Interface of Duke's Tutoring

Follow these instructions to log in to the administration interface of Duke's Tutoring and add
new students, guardians, and addresses.

1 In a web browser, open the administration interface at the following URL:
   http://localhost:8080/dukes-tutoring/admin/index.xhtml
This will redirect you to the login page.

2 At the login page, enter the user name admin@example.com and password javaee.

3 Use the administration interface to add or modify students, guardians, or addresses.
Duke’s Forest is a simple e-commerce application that contains two web applications and illustrates the use of multiple Java EE 6 APIs:

- JavaServer Faces technology, including Ajax
- Contexts and Dependency Injection for the Java EE Platform (CDI)
- Java API for XML Web Services (JAX-WS)
- Java API for RESTful Web Services (JAX-RS)
- Java Persistence API (JPA)
- Java API for JavaBeans Validation (Bean Validation)
- Enterprise JavaBeans (EJB) technology

The application consists of the following projects:

- Duke’s Store: A web application that has a product catalog, customer self-registration, and a shopping cart. It also has an administration interface for product, category, and user management. The project name is duke-store.
- Duke’s Shipment: A web application that provides an interface for order shipment management. The project name is duke-shipment.
- Duke’s Payment: A web service application that has a JAX-WS service for order payment. The project name is duke-payment.
- Duke’s Resources: A simple Java archive project that contains all resources used by the web projects. It includes messages, CSS style sheets, images, JavaScript files, and JavaServer Faces composite components. The project name is duke-resources.
- Entities: A simple Java archive project that contains all JPA entities. This project is shared among other projects that use the entities. The project name is entities.
- Events: A simple Java archive project that contains a POJO class that is used as a CDI event. The project name is events.
Design and Architecture of Duke’s Forest

Duke’s Forest is a complex application consisting of three main projects and three subprojects. Figure 53–1 shows the architecture of the three main projects that you will deploy: Duke’s Store, Duke’s Shipment, and Duke’s Payment. It also shows how Duke’s Store makes use of the Events and Entities projects.

The following topics are addressed here:
- “Design and Architecture of Duke’s Forest” on page 966
- “Building and Deploying the Duke’s Forest Case Study Application” on page 979
- “Running the Duke’s Forest Application” on page 982
Duke’s Forest uses the following Java EE 6 platform features:

- **Java Persistence API entities:**
  - Bean Validation annotations on the entities for verifying data
  - XML annotations for Java API for XML Binding (JAXB) serialization

- **Web services:**
  - A JAX-WS web service for payment, with security constraints
  - A JAX-RS web service that is EJB based

- **Enterprise beans:**
  - Local session beans
  - All enterprise beans packaged within the WAR

- **Contexts and Dependency Injection (CDI):**
  - CDI annotations for JavaServer Faces components
  - A CDI managed bean used as a shopping cart, with conversation scoping
  - Qualifiers
  - Events and event handlers

- **Servlets:**
  - A Servlet 3.0 file upload example
  - A servlet for dynamic image presentation

- **JavaServer Faces technology, using Facelets for the web front end**
  - Templating
  - Composite components
  - Resources packaged in a JAR file so they can be found in the classpath

- **Security:**
  - Java EE security constraints on the administrative interface business methods (enterprise beans)
  - Security constraints for customers and administrators (web components)

The Duke’s Forest application has two main user interfaces, both packaged within the Duke’s Store WAR file:

- The main interface, for customers and guests
- The administrative interface used to perform back office operations, such as adding new items to the catalog

The Duke’s Shipment application also has a user interface, accessible to administrators.

*Figure 53–2* shows how the web applications and the web service interact.
As illustrated in Figure 53–2, the customer interacts with the main interface of Duke’s Store, while the administrator interacts with the administration interface. Both interfaces access a façade consisting of managed beans and stateless session beans, which in turn interact with the entities that represent database tables. The façade also interacts with web services APIs that access the Duke’s Payment web service. The administrator also interacts with the interface of Duke’s Shipment, which can be accessed either directly through Duke’s Shipment or from the administration interface of Duke’s Store by means of a web service.

The most fundamental building blocks of the application are the Events and Entities projects, which are bundled into Duke’s Store and Duke’s Shipment along with the Duke’s Resources project.

**The events Project**

Events are one of the core components of Duke’s Forest. The events project, included in all three of the main projects, is the most simple project of the application. It has only one class, `OrderEvent`, but this class is responsible for most of the messages between objects in the application.
The application can send messages based on events to different components and react to them based on the qualification of the event. The application supports the following qualifiers:

- @LoggedIn: For authenticated users
- @New: When a new order is created by the shopping cart
- @Paid: When an order is paid for and ready for shipment

The following code snippet from the PaymentHandler class of Duke's Store shows how the @Paid event is handled:

```java
@Inject @Paid Event<OrderEvent> eventManager;
...
public void onNewOrder(@Observes @New OrderEvent event) {
    if (processPayment(convertForWS(event))) {
        orderBean.setOrderStatus(event.getOrderID(),
            OrderBean.Status.PENDING_PAYMENT.getStatus());
        logger.info("Payment Approved");
        eventManager.fire(event);
    } else {
        orderBean.setOrderStatus(event.getOrderID(),
            OrderBean.Status.CANCELLED_PAYMENT.getStatus())
        logger.info("Payment Denied");
    }
}...
```

To enable users to add more events to the project easily or update an event class with more fields for a new client, this component is a separate project within the application.

## The entities Project

The entities project is a Java Persistence API (JPA) project used by both Duke's Store and Duke's Shipment. It is generated from the database schema shown in Figure 53–3 and is also used as a base for the entities consumed and produced by the web services through JAXB. Each entity has validation rules based on business requirements, specified using Bean Validation.
Design and Architecture of Duke’s Forest

The database schema contains eight tables:

- **PERSON**, which has a one-to-many relationship with **PERSON_GROUPS** and **CUSTOMER_ORDER**
- **GROUPS**, which has a one-to-many relationship with **PERSON_GROUPS**
- **PERSON_GROUPS**, which has a many-to-one relationship with **PERSON** and **GROUPS** (it is the join table between those two tables)
- **PRODUCT**, which has a many-to-one relationship with **CATEGORY** and a one-to-many relationship with **ORDER_DETAIL**
- **CATEGORY**, which has a one-to-many relationship with **PRODUCT**
ORDER_DETAIL, which has a many-to-one relationship with PRODUCT and CUSTOMER_ORDER (it is the join table between those two tables)

CUSTOMER_ORDER, which has a one-to-many relationship with ORDER_DETAIL and a many-to-one relationship with PERSON and ORDER_STATUS

ORDER_STATUS, which has a one-to-many relationship with CUSTOMER_ORDER

The entity classes that correspond to these tables are as follows:

- **Person**, which defines attributes common to customers and administrators. These attributes are the person’s name and contact information, including street and email addresses. The email address has a Bean Validation annotation to ensure that the submitted data is well-formed. The generated table for Person entities also has a DTYPE field that represents the discriminator column. Its value identifies the subclass (Customer or Administrator) to which the person belongs.

- **Customer**, a specialization of Person with a specific field for CustomerOrder objects.

- **Administrator**, a specialization of Person with fields for administration privileges.

- **Groups**, which represents the group (USERS or ADMINS) to which the user belongs.

- **Product**, which defines attributes for products. These attributes include name, price, description, associated image, and category.

- **Category**, which defines attributes for product categories. These attributes include a name and a set of tags.

- **CustomerOrder**, which defines attributes for orders placed by customers. These attributes include an amount and a date, along with id values for the customer and the order detail.

- **OrderDetail**, which defines attributes for the order detail. These attributes include a quantity, along with id values for the product and the customer.

- **OrderStatus**, which defines a status attribute for each order.

### The duk es-payment Project

The duk es-payment project is a web project that holds a simple Payment web service. Since this is an example application, it does not obtain any real credit information or even customer status to validate the payment. For now, the only rule imposed by the payment system is to deny all orders above $1,000. This application illustrates a common scenario where a third-party payment service is used to validate credit cards or bank payments.

The project uses HTTP Basic Authentication and JAAS (Java Authentication and Authorization Service) to authenticate a customer to a JAX-WS web service. The implementation itself exposes a simple method, processPayment, which receives an OrderEvent to evaluate and approve or deny the order payment. The method is called from the checkout process of Duke's Store.
The dukes-resources Project

The dukes-resources project contains a number of files used by both Duke's Store and Duke's Shipment, bundled into a JAR file placed in the classpath. The resources are in the src/META-INF/resources directory:

- src/META-INF/resources/css: Two style sheets, default.css and jsfcrud.css
- src/META-INF/resources/img: Images used by the projects
- src/META-INF/resources/js: A JavaScript file, util.js
- src/META-INF/resources/util: Composite components used by the projects

The Duke's Store Project

Duke's Store, a web application, is the core application of Duke's Forest. It is responsible for the main store interface for customers as well as the administration interface.

The main interface of Duke's Store allows the user to perform the following tasks:

- Browsing the product catalog
- Signing up as a new customer
- Adding products to the shopping cart
- Checking out
- Viewing order status

The administration interface of Duke's Store allows administrators to perform the following tasks:

- Product maintenance (create, edit, update, delete)
- Category maintenance (create, edit, update, delete)
- Customer maintenance (create, edit, update, delete)
- Group maintenance (create, edit, update, delete)

The project also uses stateless session beans as façades for interactions with the JPA entities described in "The entities Project" on page 969, and CDI managed beans as controllers for interactions with Facelets pages. The project thus follows the MVC (Model-View-Controller) pattern and applies the same pattern to all entities and pages, as in the following example:

- AbstractFacade is an abstract class that receives a Type<\text{T}> and implements the common operations (CRUD) for this type, where \text{T} is a JPA entity.
- ProductBean is a stateless session bean that extends AbstractFacade, applying Product as Type<\text{T}>, and injects the PersistenceContext for the Entity Manager. This bean implements any custom methods needed to interact with the Product entity or to call a custom query.
ProductController is a CDI managed bean that interacts with the necessary enterprise beans and Facelets pages to control the way the data will be displayed.

ProductBean begins as follows:

```java
@Stateless
public class ProductBean extends AbstractFacade<Product> {
    private static final Logger logger =
        Logger.getLogger(ProductBean.class.getCanonicalName());

    @PersistenceContext(unitName="forestPU")
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }

    ...
```

**Enterprise Beans Used in Duke's Store**

The enterprise beans used in Duke's Store provide the business logic for the application and are located in the `com.forest.ejb` package. All are stateless session beans.

`AbstractFacade` is not an enterprise bean but an abstract class that implements common operations for Type<T>, where <T> is a JPA entity.

Most of the other beans extend `AbstractFacade`, inject the `PersistenceContext`, and implement any needed custom methods:

- AdministratorBean
- CategoryBean
- GroupsBean
- OrderBean
- OrderDetailBean
- OrderStatusBean
- ProductBean
- ShoppingCart
- UserBean

The `ShoppingCart`, although it is in the `ejb` package, is a CDI managed bean with conversation scope, which means that the request information will persist across multiple requests. Also, `ShoppingCart` is responsible for starting the event chain for customer orders, as described in "The events Project" on page 968.
**Facelets Files Used in the Main Interface of Duke’s Store**

Like the other case study examples, Duke’s Store uses Facelets to display the user interface. The main interface uses a large number of Facelets pages to display different areas. The pages are grouped into directories based on which module they handle.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template.xhtml</td>
<td>Template file, used for both main and administration interfaces. It first performs a browser check to verify that the user's browser supports HTML 5, which is required for Duke's Forest. It divides the screen into several areas and specifies the client page for each area.</td>
</tr>
<tr>
<td>topbar.xhtml</td>
<td>Page for the login area at the top of the screen</td>
</tr>
<tr>
<td>top.xhtml</td>
<td>Page for the title area</td>
</tr>
<tr>
<td>left.xhtml</td>
<td>Page for the left sidebar</td>
</tr>
<tr>
<td>index.xhtml</td>
<td>Page for the main screen content</td>
</tr>
<tr>
<td>login.xhtml</td>
<td>Login page specified in web.xml. The main login interface is provided in topbar.xhtml, but this page appears if there is a login error.</td>
</tr>
<tr>
<td>admin directory</td>
<td>Pages related to the administration interface, described in &quot;Facelets Files Used in the Administration Interface of Duke’s Store&quot; on page 974</td>
</tr>
<tr>
<td>customer directory</td>
<td>Pages related to customers (Create.xhtml, Edit.xhtml, List.xhtml, Profile.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>order directory</td>
<td>Pages related to orders (Create.xhtml, List.xhtml, MyOrders.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>orderDetail directory</td>
<td>Popup page allowing users to view details of an order (View_popup.xhtml)</td>
</tr>
<tr>
<td>orderStatus directory</td>
<td>Pages related to order status (Create.xhtml, Edit.xhtml, List.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>product directory</td>
<td>Pages related to products (List.xhtml, ListCategory.xhtml, View.xhtml)</td>
</tr>
</tbody>
</table>

**Facelets Files Used in the Administration Interface of Duke’s Store**

The Facelets pages for the administration interface of Duke’s Store are found in the web/admin directory.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>administrator directory</td>
<td>Pages related to administrator management (Create.xhtml, Edit.xhtml, List.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>Directory</td>
<td>Pages related to management (Create.xhtml, Edit.xhtml, List.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Category directory</td>
<td>Pages related to product category management (Create.xhtml, Edit.xhtml, List.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>Customer directory</td>
<td>Pages related to customer management (Create.xhtml, Edit.xhtml, List.xhtml, Profile.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>Groups folder</td>
<td>Pages related to group management (Create.xhtml, Edit.xhtml, List.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>Order directory</td>
<td>Pages related to order management (Create.xhtml, Edit.xhtml, List.xhtml, View.xhtml)</td>
</tr>
<tr>
<td>OrderDetail directory</td>
<td>Popup page allowing the administrator to view details of an order (View_popup.xhtml)</td>
</tr>
<tr>
<td>Product directory</td>
<td>Pages related to product management (Confirm.xhtml, Create.xhtml, Edit.xhtml, List.xhtml, View.xhtml)</td>
</tr>
</tbody>
</table>

**Managed Beans Used in Duke’s Store**

Duke’s Store uses the following CDI managed beans, which correspond to the enterprise beans. The beans are in the `com.forest.web` package.

- AdministratorController
- CategoryController
- CustomerController
- CustomerOrderController
- GroupsController
- OrderDetailController
- OrderStatusController
- ProductController
- UserController

**Helper Classes Used in Duke’s Store**

The CDI managed beans in the main interface of Duke’s Store use the following helper classes, found in the `com.forest.web.util` package:

- AbstractPaginationHelper: An abstract class with methods used by the managed beans
- FileUploadServlet, ImageServlet: Classes used for image processing. FileUploadServlet uploads an image and stores its content in the database. ImageServlet retrieves the image content from the database and displays it. (JavaServer Faces technology does not provide this functionality, so a servlet is needed.)
JsFsUtil  
Class used for JavaServer Faces operations, such as queuing messages on a FacesContext instance

MD5Util  
Class used by the CustomerController managed bean to generate an encrypted password for a user

**Qualifiers Used in Duke’s Store**

Duke’s Store defines the following qualifiers in the `com.forest.qualifiers` package:

- @LoggedIn: Qualifies a user as having logged in.
- @New: Qualifies an order as new.
- @Paid: Qualifies an order as paid.

**Event Handlers Used in Duke’s Store**

Duke’s Store defines event handlers related to the `OrderEvent` class packaged in “The events Project” on page 968. The event handlers are in the `com.forest.handlers` package:

- IOrderHandler: The `IOrderHandler` interface defines a method, `onNewOrder`, implemented by the two handler classes.
- PaymentHandler: The `ShoppingCart` bean fires an `OrderEvent` qualified as `@New`. The `onNewOrder` method of `PaymentHandler` observes these events and, when it intercepts them, processes the payment using the Duke’s Payment web service. After a successful response from the web service, `PaymentHandler` fires the `OrderEvent` again, this time qualified as `@Paid`.
- DeliveryHandler: The `onNewOrder` method of `DeliveryHandler` observes `OrderEvent` objects qualified as `@Paid` (orders paid and ready for delivery) and modifies the order status to `PENDING_SHIPMENT`. When an administrator accesses Duke’s Shipment, it will call the Order Service, a RESTful web service, and ask for all orders in the database that are ready for delivery.

**Properties Files Used in Duke’s Store**

The strings used in the main and administration interfaces of Duke’s Store are encapsulated into resource bundles to allow the display of localized strings in multiple locales. The resource bundles are located in the default package.

- Bundle.properties: Application messages in English
- Bundle_es.properties: Application messages in Spanish
- ValidationMessages.properties: Bean Validation messages in English
- ValidationMessages_es.properties: Bean Validation messages in Spanish
Deployment Descriptors Used in Duke's Store

Duke's Store uses the following deployment descriptors, located in the web/WEB-INF directory:

- beans.xml: An empty deployment descriptor file used to enable the CDI runtime
- faces-config.xml: The JavaServer Faces configuration file
- glassfish-web.xml: The configuration file specific to GlassFish Server
- jaxws-catalog.xml: A deployment descriptor for a JAX-WS web service client
- web.xml: The web application configuration file

The Duke’s Shipment Project

Duke's Shipment is a web application with a login page, a main Facelets page, and some other objects. This application, which is accessible only to administrators, calls the Order Service (the RESTful web service exposed by Duke's Store) and lists all orders under two status headings: Pending and Shipped. The administrator can either approve or deny a pending order. If approved, the order is shipped, and it appears under the Shipped heading. If denied, the order disappears from the page, and on the customer’s Orders list it appears as cancelled.

There is also a gear icon on the Pending list that makes an Ajax call to the Order Service to refresh the list without refreshing the page. The code looks like this:

```xml
<h:commandLink>
    <h:graphicImage library="img" title="Check for new orders"
                    style="border:0px" name="refresh.png"/>
    <f:ajax execute="refresh" render="out" />
</h:commandLink>
```

Enterprise Bean Used in Duke’s Shipment

The enterprise bean used in Duke’s Shipment, UserBean, provides the business logic for the application and is located in the com.forest.shipment.session package. It is a stateless session bean.

Like Duke's Store, Duke's Shipment uses the AbstractFacade class. This class is not an enterprise bean but an abstract class that implements common operations for Type<T>, where <T> is a JPA entity.

Facelets Files Used in Duke’s Shipment

Duke’s Shipment has only one page, so it has many fewer Facelets files than Duke’s Store.
The template file, like the one in Duke's Store, first performs a browser check to verify that the user's browser supports HTML 5, which is required for Duke's Forest. It divides the screen into areas and specifies the client page for each area.

topbar.xhtml  Page for the login area at the top of the screen
top.xhtml  Page for the title area
left.xhtml  Page for the left sidebar (not used in Duke's Shipment)
index.xhtml  Page for the initial main screen content
login.xhtml  Login page specified in web.xml. The main login interface is provided in topbar.xhtml, but this page appears if there is a login error
admin/index.xhtml  Page for the main screen content after authentication

**Managed Beans Used in Duke's Shipment**

Duke's Shipment uses the following CDI managed beans, in the `com.forest.shipment` package:

- `control.ShippingBean`  Managed bean that acts as a client to the Order Service
- `web.UserController`  Managed bean that corresponds to the UserBean session bean

**Helper Class Used in Duke's Shipment**

The Duke's Shipment managed beans use only one helper class, found in the `com.forest.shipment.web.util` package:

- `JsfUtil`  Class used for JavaServer Faces operations, such as queuing messages on a `FacesContext` instance

**Qualifier Used in Duke's Shipment**

Duke's Shipment defines the following qualifier in the `com.forest.qualifiers` package:

- `@LoggedIn`  Qualifies a user as having logged in

**Properties Files Used in Duke's Shipment**

The Duke's Shipment properties files, located in the default package, are `Bundle.properties`, containing English strings, and `Bundle_es.properties`, containing Spanish strings. They are identical to the files in Duke's Store.
Deployment Descriptors Used in Duke’s Shipment

Duke’s Shipment uses the following deployment descriptors:

- **web/WEB-INF/beans.xml**: An empty deployment descriptor file used to enable the CDI runtime
- **web/WEB-INF/faces-config.xml**: The JavaServer Faces configuration file
- **web/WEB-INF/glassfish-web.xml**: The configuration file specific to GlassFish Server
- **web/WEB-INF/web.xml**: The web application configuration file
- **src/conf/persistence.xml**: The Java Persistence API configuration file

Building and Deploying the Duke’s Forest Case Study Application

You can use NetBeans IDE or Ant to build and deploy Duke’s Forest. The prerequisite task requires Ant.

Prerequisite Task

Before you begin this task, you must have already configured GlassFish Server as a Java EE server in NetBeans IDE, as described in “To Add GlassFish Server as a Server in NetBeans IDE” on page 72.

To Create the JDBC Realm and Populate the Database

1. Enable Default Principal to Role Mapping on the GlassFish Server, if you have not done so previously:
   a. From the Administration Console, expand the Configurations node, then expand the server-config node.
   b. Select the Security node.
   c. Select the Default Principal to Role Mapping Enabled check box.
   d. Click Save.

2. In a terminal window, go to:
   `tut-install/examples/case-studies/dukes-forest/entities/`
3 **Execute the `create-forest-realm` Ant task:**
   
   ```
   ant create-forest-realm
   ```
   
   This task creates a JDBC connection pool and a JDBC resource as well as the realm.

4 **Execute the Ant task:**
   
   ```
   ant
   ```
   
   This task creates the tables (dropping any existing tables) and builds the JAR file. Ignore any errors if you are running the task for the first time.

---

### To Build and Deploy the Duke's Forest Application Using NetBeans IDE

1 **From the File menu, choose Open Project.**

2 **In the Open Project dialog, navigate to:**
   
   `tut-install/examples/case-studies/dukes-forest/

3 **Select the `dukes-store` folder.**

4 **Select the Open Required Projects check box.**

5 **Click Open Project.**
   
   The IDE will open the `dukes-store`, `dukes-resources`, `entities`, and `events` projects.
   
   The project opens with a message stating that there is a data source problem.

6 **Right-click the project and select Resolve Data Source Problem.**

7 **In the dialog that opens, select `jdbc/forest` and click Add Connection.**

8 **Click Finish.**
   
   The connection to the `forest` database is now established.
   
   If the project still indicates there is a data source problem but the dialog does not indicate a missing connection, close and reopen the project.

9 **Repeat steps 1–5 to open the `dukes-shipment` project.**

10 **Repeat steps 1–5 to open the `dukes-payment` project.**

11 **Right-click the `events` project and select Build.**
Right-click the dukes-resources project and select Build.

Right-click the dukes-payment project and select Deploy.

Right-click the dukes-store project and select Deploy.

Right-click the dukes-shipment project and select Deploy.

The dukes-shipment project requires the file jersey-client.jar, which is located in as-install/lib/modules/. If you get a Resolve References error when you first try to build dukes-shipment, you can resolve the error by locating this file.

To Build and Deploy the Duke’s Forest Application Using Ant

1. In a terminal window, go to:
   tut-install/examples/case-studies/dukes-forest/events/

2. Enter the following command to build the events.jar file:
   ant

3. Go to the dukes-resources directory:
   cd ../dukes-resources

4. Enter the following command to build the dukes-resources.jar file:
   ant

5. Go to the dukes-payment directory:
   cd ../dukes-payment

6. Enter the following command:
   ant all

7. Go to the dukes-store directory:
   cd ../dukes-store

8. Enter the following command:
   ant all

9. Go to the dukes-shipment directory:
   cd ../dukes-shipment
Enter the following command:

```
ant all
```
When you have finished choosing products, click Checkout. A message appears, reporting that your order is being processed.

Click Orders in the left sidebar to verify your order. If the total of the order was over $1,000, the status of the order is “Order cancelled,” because the Payment web service denies orders over that limit. Otherwise, the status is “Ready to ship.”

When you have finished placing orders, click the Logout button at the top of the page.

To Approve Shipment of a Product

1. **Log in to Duke’s Store as an administrator.**
   
   Your user name is admin@example.com, and your password is 1234. The main administration page allows you to view categories, customers, administrators, groups, products, and orders, and to create new objects of all types except orders.

2. **At the bottom of the page, click Approve Shipment.**
   
   This action takes you to Duke's Shipment.

3. **Log in to Duke’s Shipment at the top of the page, as admin@example.com.**

4. **On the Pending list, click Approve to approve an order and move it to the Shipped area of the page.**
   
   If you click Deny, the order disappears from the page. If you log in to Duke's Store again as the customer, it will appear in the Orders list as “Order Cancelled.”

To Create a New Product

You can create other kinds of objects as well as products. Creating products is more complex than the other creation processes.

1. **Log in to Duke’s Store as an administrator.**

2. **On the main administration page, click Create New Product.**

3. **Enter values in the Name, Price, and Description fields.**
4 Select a category, then click Next.

5 On the Upload the Product Image page, click Browse to locate an image on your file system using a file chooser.

6 Click Next.

7 On the next page, view the product fields, then click Done.

8 Click Products in the left sidebar, then click the category to verify that the product has been added.

9 Click Administration at the top of the page to return to the main administration page, or click Logout to log out.
Index

Numbers and Symbols
@AccessTimeout annotation, 471
@Alternative annotation, 539–541
@ApplicationScoped annotation, 88, 296, 520–522
@AroundInvoke annotation, 934
@AroundTimeout annotation, 934
@Asynchronous annotation, 506
@ConcurrencyManagement annotation, 470
@Consumes annotation, 384, 391–392
@Context annotation, 403–406
@ConversationScoped annotation, 520–522
@CookieParam annotation, 403–406
@CustomScoped annotation, 297
@DeclareRoles annotation, 743–745, 750–752
@Decorator annotation, 549–550
@Dependent annotation, 549–550
@DELETE annotation, 383, 387–388
@DenyAll annotation, 744
@Dependent annotation, 520–522
@DependsOn annotation, 469
@DiscriminatorColumn annotation, 593–594
@DiscriminatorValue annotation, 593–594
@Disposes annotation, 543
@Embeddable annotation, 590
@EmbeddedId annotation, 585
@Entity annotation, 580
@FormParam annotation, 403–406
@GET annotation, 383, 387–388
@GroupSequence annotation, 931–932
@HEAD annotation, 383
@HeaderParam annotation, 403–406
@HttpConstraint annotation, 715, 731
@HttpMethodConstraint annotation, 715, 731
@Id annotation, 585
@IdClass annotation, 585
@Inject annotation, 520
@Local annotation, 443, 462
@Lock annotation, 470–472
@ManagedBean annotation, 106–107, 113–114, 296–297
@ManyToMany annotation, 587, 588
@ManyToOne annotation, 587
@MatrixParam annotation, 403–406
@MessageDriven annotation, 899–900
@MultipartConfig annotation, 347–348
@Named annotation, 522–523
@NamedQuery annotation, 590
@NoneScoped annotation, 297
@Observe annotation, 545–546
@OneToMany annotation, 587, 588, 589
@OneToOne annotation, 587, 588, 589
@Path annotation, 383, 385–387
@PathParam annotation, 383, 392–395, 403–406
@PermitAll annotation, 744
@PersistenceContext annotation, 595
@PersistenceUnit annotation, 596
@POST annotation, 383, 387–388
@PostActivate annotation, 463, 464
@PostConstruct annotation, 450–453, 463, 464, 934
session beans using JMS, 899
@PreDestroy annotation, 450–453, 463, 464, 934
session beans using JMS, 899
@PrePassivate annotation, 463, 464
applet container, 49
applets, 43, 44
application client container, 49
application clients, 42–43
  securing, 778–779
application clients, JMS
  building, 865, 868, 871
  examples, 492, 860–866
  packaging, 874
  running, 865–867, 868–870, 872–873, 874–875
  running on multiple systems, 880–886
applications
  JavaServer Faces, 104
  security, 698
  undeploying, 92–93
asadmin tool, 67
asynchronous message consumption, 832
  See also message-driven beans
  JMS client example, 870–875
asynchronous method invocation
  calling asynchronous business methods, 507–508
  cancelling, 507
  checking status, 508
  creating asynchronous business methods, 506
  example, 508–511
  java.util.concurrent.Future<V> interface, 505
  retrieving results, 507
  session beans, 505–511
attributes referencing managed bean methods, 186
  action attribute, 186, 187
  actionListener attribute, 186, 187
  validator attribute, 186, 187
  valueChangeListener attribute, 186, 188
audit modules, pluggable, 702
auditing, 697
auth-constraint element, 717
authenticate method, 724–726
authenticating users, 719–722, 722–723
authentication, 696–697, 710
  basic, 720
  basic with EJB, 746
  certificate-based mutual, 765
  client, 764, 766–767
  digest, 722
authentication (Continued)
  form-based, 721–722, 734–738
  mutual, 764–768
  user name/password-based mutual, 765
authorization, 696–697
authorization constraints, 716, 717
authorization providers, pluggable, 702
AUTO_ACKNOWLEDGE mode, 844
auto commit, 61

B
basic authentication, 720
  EJB, 746
  example, 730–733
bean-managed transactions, 856
  See transactions, bean-managed
Bean Validation, 62
  advanced, 929–932
  constraints, 623–624
  custom constraints, 929–930
  examples, 623–627
  groups, 931–932
  Java Persistence API, 583–585
  JavaServer Faces applications, 204–208, 625
  localization, 930
  messages, 930
  ordering, 931–932
  resource bundles, 930
beans, defined for CDI, 517
beans.xml file, 525
beforeCompletion method, 799
BLOBs, See persistence, BLOBs
bookmarkable URLs
  component tags, 165–166
  example, 167–168
  view parameters, 166–167
BufferedReader class, 333
build artifacts, removing, 92–93
bundles, See resource bundles
business logic, 436
business methods, 445
  client calls, 465
  exceptions, 465
business methods (Continued)
  locating, 456
  requirements, 465
  transactions, 797, 799, 801, 802
BytesMessage interface, 841

C
CallbackHandler interface, 778
capture-schema tool, 67
CDI
  See Contexts and Dependency Injection (CDI) for the Java EE platform
certificate authorities, 760
certificates, 698
  client, 767–768
  digital, 699, 759–764
  server, 763–764
  server, generating, 760–762
  using for authentication, 762
character encodings, 360
character sets, 359–360
class files, removing, 92–93
CLIENT_ACKNOWLEDGE mode, 844
client certificates, generating, 767–768
client ID, for durable subscriptions, 847
clients
  authenticating, 764, 766–767
  securing, 778–779
CLOBs, See persistence, CLOBs
collections
  persistence, 581–583, 671
commit method, 799
commit method (JMS), 849–851
commits, See transactions, commits
Common Client Interface, Connector
  architecture, 815–816
component binding, 191, 192, 288, 292–293
  binding attribute, 191, 288, 292
component-managed sign-on, 780
component properties, See managed bean properties
component rendering model, 217, 219
  decode method, 213, 267, 275, 280
  decoding, 254, 262
component rendering model (Continued)
  delegated implementation, 255
  direct implementation, 255
  encode method, 281
  encodeBegin method, 265
  encodeChildren method, 265
  encodeEnd method, 265, 271
  encoding, 254, 262
  HTML render kit, 276, 317
  render kit, 219
  Renderer class, 219
  Renderer implementation, 317
  RenderKit class, 220
  RenderKit implementation, 317
component tag attributes
  action attribute, 152, 201, 258
  actionListener attribute, 152, 186, 202, 258
  alt attribute, 258
  binding attribute, 143, 145, 191, 288, 292
  columns attribute, 156
  converter attribute, 148, 176–177, 282
  for attribute, 151, 165
  id attribute, 143
  immediate attribute, 143, 258
  redisplay attribute, 150
  rendered attribute, 143, 144–145, 292
  style attribute, 143, 145, 165
  styleClass attribute, 143, 145
  validator attribute, 148, 202
  value attribute, 143, 145, 192, 259, 288, 289–291
  valueChangeListener attribute, 149, 188, 203
  var attribute, 359
component tags, 192, 220, 222
attributes
  See component tag attributes
body tag, 145–146
book makable URLs, 165–166
button tag, 165–166
column tag, 141
commandButton tag, 141, 152–153
commandLink tag, 141, 153
dataTable tag, 141, 161–164, 194
form tag, 141, 146–147, 147
graphicImage tag, 141, 153–154
component tags (Continued)

h:graphicImage tag, 258
head tag, 145–146
inputHidden tag, 141, 148
inputSecret tag, 141, 148, 150
inputText tag, 141, 148, 149–150
inputTextarea tag, 141, 148
link tag, 165–166
message tag, 141, 164–165
messages tag, 141, 164–165
outputFormat tag, 141, 149, 151
outputLabel tag, 141, 149, 150–151
outputLink tag, 142, 149, 151
outputScript tag, 169–170
outputStylesheet tag, 169–170
outputText tag, 142, 149, 153, 195
panelGrid tag, 142, 154–156
panelGroup tag, 142, 154–156
resource relocation, 169–170
selectBooleanCheckbox tag, 142, 157, 195
selectItems tag, 198
selectManyCheckbox tag, 142, 158–159, 196
selectManyListbox tag, 142, 158
selectManyMenu tag, 142, 158
selectOneListbox tag, 142, 157
selectOneMenu tag, 142, 157–158, 197
selectOneRadio tag, 142, 157

components

buttons, 141, 152–153
check boxes, 142, 157, 158–159
combo boxes, 142
data grids, 141
hidden fields, 141
hyperlinks, 141, 153
images, 153–154
labels, 141, 142
list boxes, 142
menus, 157–158, 158
password fields, 141
radio buttons, 142
table columns, 141
tables, 142, 154–156, 161–164
text areas, 141
text fields, 141

composite components

advanced features, 243–249
attributes, 243–244
default attribute, 243
eexample, 245–249
f:validateBean tag, 245
f:validateRegex tag, 245
f:validateRequired tag, 245
Facelets, 121–123
invoking managed beans, 244
method-signature attribute, 244
name attribute, 243
required attribute, 243
type attribute, 244
validating values, 245

concurrent access, 793
concurrent access to entity data, 677–679
conditional HTTP requests, JAX-RS, 409–410
confidentiality, 710
configuring JavaServer Faces applications

Application class, 298
application configuration resource files, 297–300
configuring managed beans, 296–297, 300–308
configuring navigation rules

See configuring navigation rules
error message registration, 284
faces-config.xml files, 315
including the classes, pages, and other
resources, 324–325
javax.faces.application.CONFIG_FILES context
parameter, 298
registering custom converters, 313
registering custom renderers, 317–319
registering custom UI components, 319–320
registering custom validators, 312
registering messages, 308–311
specifying where UI component state is saved, 270
value binding, 289–291
configuring managed beans, 259, 300–308
configuring navigation rules, 223, 313–317
from-action element, 315
from-view-id element, 315
navigation-case element, 314, 315
navigation-rule element, 315
configuring navigation rules (Continued)
to-view-id element, 315
connection factories, 834
creating, 495, 863–864
injecting resources, 492, 834
specifying for remote servers, 881–882
Connection interface, 799, 803
Connection interface (JMS), 835
connection pooling, 806
ConnectionFactory interface (JMS), 834
connections, securing, 710–711
connections, JMS
introduction, 835
managing in enterprise bean applications, 853
connectors, See Java EE Connector architecture
container-managed sign-on, 780
container-managed transactions, See transactions,
container-managed
containers, 47–49
See also EJB container, web container
application client, 49
configurable services, 47
nonconfigurable services, 47
securing, 700–701
security, 692–697
services, 47
trust between, 749
web, 82
context parameters, 88
specifying, 97–98
context roots, 89

Contexts and Dependency Injection (CDI) for the Java
EE platform (Continued)
examples, 527–538, 553–576
Facelets pages, 524
injectable objects, 518
injecting beans, 520
integrating with JAX-RS, 408–409
interceptors, 547–549
managed beans, 517–518
observer methods, 545–546
overview, 516
producer fields, 541–543, 543
producer methods, 524–525, 541–543
qualifiers, 519
scopes, 520–522
setter and getter methods, 523
specialization, 540–541
stereotypes, 550–551
conversational state, 437
conversion model, 217, 220–221
See also converter tags
See also converters
converter attribute, 148, 176–177, 282
Converter implementations, 175–180, 220, 282
Converter interface, 279
converterId attribute, 176
converting data between model and
presentation, 220
javax.faces.convert package, 175
model view, 279, 280
presentation view, 279, 280
Converter implementation classes
BigDecimalConverter class, 175
BigIntegerConverter class, 175
BooleanConverter class, 175
ByteConverter class, 175
CharacterConverter class, 175
DateTimeConverter class, 175, 176, 177
DoubleConverter class, 175
EnumConverter class, 175
FloatConverter class, 176
IntegerConverter class, 176
LongConverter class, 176
NumberConverter class, 176, 177, 179–180
databases (Continued)
   clients, 436
   connections, 465, 801
   data recovery, 793
   EIS tier, 40
   message-driven beans and, 440
   multiple, 800, 802
DataSource interface, 806
debugging, Java EE applications, 77–78
declarative security, 692, 714, 740
   example, 750–754
decorators
   CDI, 549–550
   example, 574–576
delivery modes, 845
   JMSDeliveryMode message header field, 840
DeliveryMode interface, 845
Dependency Injection for Java (JSR 330), 62, 515
developer roles, 54
deployment, 457–458
deployment descriptors, 51, 692, 701
   enterprise bean, 448, 701, 740, 742
   Java EE, 51
   runtime, 51, 84
   security-role-mapping element, 709–710
   security-role-ref element, 728–729
   web application, 82, 84, 320, 701
Destination interface, 834–835
destinations, 834–835
   See also queues, temporary destinations, topics creating, 495, 863–864
   injecting resources, 492, 834
   JMSDestination message header field, 840
temporary, 846–847, 904–905, 918–919
destroy method, 342
development roles, 52–55
   application assemblers, 54
   application client developers, 54
   application component providers, 53–54
   application developers and administrators, 54
   enterprise bean developers, 53
   Java EE product providers, 53
tool providers, 53
   web component developers, 53
digest authentication, 722
digital signatures, 760
disposable methods, CDI, 543
document roots, 84
dofilter method, 335, 336, 337
doGet method, 332
doPost method, 332
downloading, GlassFish Server, 70
DUPS_OK_ACKNOWLEDGE mode, 844
durable subscriptions, 847–849
   examples, 889–891, 897–902

e
   eager attribute, managed beans, 297
   EAR files, 51
   EIS tier, 46
   security, 780–783
   EJB container, 48
   See also embedded enterprise bean container
   container-managed transactions, 794
   message-driven beans, 853–855
   onMessage method, invoking, 494–495
   services, 435, 436, 739–749
   EJB JAR files, 447
ejb-jar.xml file, 448, 701, 742
   EJBContext interface, 799, 801
   EL, 107, 125–137
   CDI managed beans, 522–523
   composite expressions, 131
   deferred evaluation expressions, 126
   expression examples, 137
   immediate evaluation expressions, 126
   literal expressions, 131, 135
   literals, 130
   lvalue expressions, 126, 128
   managed beans, 191–192
   method-binding expressions, 224
   method expressions, 126, 132
   operators, 136
   overview, 125–126
   parameterized method calls, 133
   reserved words, 136
EL (Continued)
rvalue expressions, 126, 128
tag attribute type, 134
type conversion during expression evaluation, 131
value expressions, 126, 128
embeddable classes, See persistence: embeddable classes
embedded enterprise bean container
See also EJB container, enterprise beans
creating, 500–501
developing applications, 499
eamples, 502–504
initializing enterprise bean modules, 501
overview, 499
running applications, 500
session bean references, 502
shutting down, 502
end-to-end security, 699–700
enterprise applications, 37
securing, 739–757
enterprise beans, 45, 58–59
See also business methods
See also embedded enterprise bean container
See also Java EE components
See also message-driven beans
See also session beans
accessing, 441
classes, 447
compiling, 457–458
contents, 447–449
converting to JAX-RS root resource
classes, 408–409
defined, 435
dependency injection, 441
deployment, 447
distribution, 443
exceptions, 489
finding, 502
getCallerPrincipal method, 746–747
implementor of business logic, 45
integrating with JAX-RS, 408–409
interceptors, 933–941
interfaces, 441–447, 447
isCallerInRole method, 746–747
JAX-RS resources, 398–400
enterprise beans (Continued)
JNDI lookup, 441
lifecycles, 450–453
local access, 443–445
local interfaces, 444
packaging, 447–448, 457–458
performance, 443
programmatic security, 746–747
remote access, 445–446
remote interfaces, 445
securing, 739–749
singletons, 398–399
testing, 502–504	
timer service, 479–489
types, 436
web services, 437, 446, 476–479
Enterprise Information Systems, See EIS tier
entities
abstract, 591
abstract schema names, 632
application-managed entity managers, 596–597
cascading operations, 588–589, 589
collections, 644–645
container-managed entity managers, 595
controlling caching, 684–685
creating, 612
discriminator columns, 593
dentity manager, 595–599
finding, 597, 613
inheritance, 591–595, 617–618
inheritance mapping, 592–595
lifecycle, 597
managing, 595–600, 612–614
mapping to multiple tables, 610
non-entity superclasses, 592
overview, 579–590
persistent fields, 580–585
persistent properties, 580–585
persisting, 597–598
primary keys, 585–587
querying, 600–601
relationships, 613
removing, 598, 614
requirements, 580
entities (Continued)
  superclasses, 591–592
  synchronizing, 598–599
  validating, 583–585
entity data
  lock modes, 679–681
  optimistic locking, 677, 678–679
  pessimistic locking, 677, 680–681
entity providers, JAX-RS, 389–390
entity relationships
  bidirectional, 587–588
  many-to-many, 587, 617
  many-to-one, 587
  multiplicity, 587
  one-to-many, 587
  one-to-one, 587
  query language, 588
  unidirectional, 588
equals method, 586
event and listener model, 217, 221–222
  See also action events
  See also value-change events
  binding listeners to managed bean properties, 293–294
  Event class, 221
  event handlers, 213, 261
  event listeners, 213, 214
  handling events of custom UI components, 275–276
  implementing event listeners, 273–274
  Listener class, 221
  listener class, 200
  queueEvent method, 267
  ValueChangeEvent class, 188
events
  CDI, 545–547
  example, 568–574
examples, 69–78
  Ajax, 238–242
  asynchronous method invocation, session beans, 508–511
  basic authentication, 730–733
  Bean Validation, 623–627
  bookmarkable URLs, 167–168
examples (Continued)
  building, 75
  CDI, 527–538, 553–576
  classpath, 458
  composite components, 245–249
  connectors, 819–823
  Criteria API, 619–620
  directory structure, 76
  Duke’s Bookstore case study, 945–952
  Duke’s Forest case study, 965–984
  Duke’s Tutoring case study, 953–963
  embedded enterprise bean container, 502–504
  file upload using servlets, 349–353
  interceptors, 939–941
  Java EE Connector architecture, 819–823
  JAX-RS, 396–401, 418–432
  JAX-WS, 368–376
  JMS asynchronous message consumption, 870–875
  JMS durable subscriptions, 889–891
  JMS local transactions, 891–897
  JMS message acknowledgment, 887–889
  JMS message-driven beans, 491–497
  JMS on multiple systems, 880–886, 910–916, 916–927
  JMS queue browsing, 875–880
  JMS synchronous message consumption, 860–870
  JMS with entities, 902–910
  JMS with session beans, 897–902
  message-driven beans, 491–497
  persistence, 603–627
  primary keys, 586
  query language, 613–614, 633–637
  required software, 69–73
  resource adapters, 819–823
  security, 734–738, 750–754, 754–757
  servlet, 344–345
  servlets, 93–101, 456
  session beans, 456, 461–468
  singleton session beans, 468–476
  timer service, 485–488
  web clients, 456
  web services, 476–479
exceptions
  business methods, 465
exceptions (Continued)
    enterprise beans, 489
    JMS, 842
    mapping to error screens, 98–99
    rolling back transactions, 489, 799
    transactions, 796, 797
expiration of JMS messages, 846
    JMSExpiration message header field, 840
Expression Language
    See EL
expressions
    lvalue expressions, 191
    tag attribute type, 134

F
Facelets, 111–124
    See also EL
    composite components, 121–123
    configuring applications, 116–117
    f:ajax tag, 229–231
    features, 111–113
    resources, 123–124
    templating, 119–121
    using Ajax with, 229–231
    XHTML pages, 114–116
Facelets applications
    developing, 113–119
    lifecycle, 216
    using JavaScript in, 237–238
faces-config.xml file, 297–300
FacesContext class, 212, 278
    Apply Request Values phase, 213
    custom converters, 280
    performing encoding, 266
    Process Validations phase, 214
    Update Model Values phase, 214
    validation methods, 202
    Validator interface, 285
FacesServlet, mapping, 108
    filter chains, 335, 337
    Filter interface, 335
    filters, 334
        defining, 335
filters (Continued)
    mapping to web components, 336
    mapping to web resources, 336
    overriding request methods, 336
    overriding response methods, 336
    response wrappers, 336
    foreign keys, 606
    form-based authentication, 721–722
    form parameters, 395
    JAX-RS, 405–406
    forward method, 339

G
garbage collection, 453
GenericServlet interface, 328
    getCallerPrincipal method, 746–747, 754–757
    getConnection method, 806
    getPart method, 348
    getRemoteUser method, 726
    getRequestDispatcher method, 338
    getRollbackOnly method, 801, 857
    getServletContext method, 339
    getSession method, 340
    getStatus method, 801
    getUserPrincipal method, 726
GlassFish Server
    adding users to, 706–707
    downloading, 70
    enabling debugging, 78
    installation tips, 70
    securing, 702
    server log, 77–78
    SSL connectors, 711
    starting, 73–74
    stopping, 73
    tools, 67–68
    groups, 705
    managing, 706–707
Index

H
handling events, See event and listener model 586
hashCode method, 586
header parameters, JAX-RS, 395
helper classes, 447
  session bean example, 466
HTTP, 367
  basic authentication, 720
  over SSL, 764
HTTP methods, 387–390
HTTP request and response entity bodies, 389–390
  supported types, 389
HTTP request URLs, 333
  query strings, 333
  request paths, 333
HTTP requests, 333, 387–390
  See also requests
HTTP responses, 334
  See also responses
status codes, 98–99
HTTPS, 699, 711, 717–718, 760
HttpServletRequest interface, 333
HttpServletResponse interface, 334
HttpSession interface, 340
initParams attribute, 332
injectable objects, CDI, 518
integrity, 710
  of data, 696
interceptors, 933–941
  CDI, 547–549
  classes, 934
  example, 939–941
  example (CDI), 568–574
  lifecycle, 934
  using, 935–939
internationalization, 355–360
internationalizing JavaServer Faces applications
  f:loadBundle tag, 359
  FacesContext.getLocale method, 178
  using the FacesMessage class to create a message, 310
invalidate method, 341
isCallerInRole method, 746–747, 754–757
isUserInRole method, 726

J
JAAS, 67, 697, 779
  login modules, 779
JACC, 63–64, 702
JAF, 65
JAR files, 51
  query language, 643
JAR signatures, 698
JASPIC, 64
Java API for JavaBeans Validation, See Bean Validation
Java API for RESTful Web Services, See JAX-RS
Java API for XML Binding (JAXB), 66
  using with JAX-RS, 412–418
Java API for XML Processing (JAXP), 65–66
Java API for XML Web Services, See JAX-WS
Java Authentication and Authorization Service, See JAAS
Java Authentication Service Provider Interface for Containers (JASPIC), 64
Java Authorization Contract for Containers, See JACC
Java BluePrints, 76
Java Cryptography Extension (JCE), 697
Java Database Connectivity API, See JDBC API
Java DB, 67
  starting, 75
  stopping, 75
Java EE applications, 39–46
  debugging, 77–78
  deploying, 457–458
  iterative development, 458
  running on more than one system, 910–916, 916–927
Java EE clients, 42–43
  See also application clients, web clients
Java EE components, 42
Java EE Connector architecture, 63, 790, 810–813
  example, 819–823
Java EE modules, 51, 52
  See also web modules
    application client modules, 52
    EJB modules, 52, 447
    resource adapter modules, 52
Java EE platform
  APIs, 55–64
  JMS and, 827–828
  overview, 38–39
Java EE security model, 47
Java EE servers, 48
Java EE transaction model, 47
Java Generic Security Services, 697
Java GSS-API, 697
Java Message Service (JMS) API, See JMS
Java Naming and Directory Interface API, 65
  See also JNDI
Java Persistence API, 61, 579–601
  See also persistence
Java Persistence API query language, See query language
Java Persistence Criteria API, See Criteria API
Java Secure Sockets Extension (JSSE), 697
Java Servlet technology, 59, 327–346
  See also servlets
Java Transaction API, 61, 800
JavaBeans Activation Framework (JAF), 65
JavaBeans components, 43

JavaMail API, 63
  example, 819–823
JavaServer Faces application development, 106–110
  bean property, 194
  Bean Validation, 204–208
  managed beans, 189–192
  web pages, 139–173
JavaServer Faces applications
  configuring
    See configuring JavaServer Faces applications
JavaServer Faces core tag library, 139, 171
  actionListener tag, 171, 182–183
  attribute tag, 172
  convertDateTime tag, 171, 177
  convertDateTime tag attributes, 178–179
  converter tag, 171, 177
  converterId attribute, 176
  convertNumber tag, 171, 177, 179–180
  convertNumber tag attributes, 179–180
  f:actionListener tag, 254
  f:ajax tag, 229–231
  f:converter tag, 282
  f:validator tag, 223, 283
    custom validator tags, 286
  facet tag, 162, 171
  loadBundle tag, 172
  metadata tag, 166–167, 171
  param tag, 151, 172
  selectItem tag, 158, 159, 160, 172
  selectItems tag, 158, 159, 160, 172
  type attribute, 181
  validateDoubleRange tag, 172, 184
  validateLength tag, 172, 184
  validateLongRange tag, 172, 184, 185–186
  validator tag, 172
  valueChangeListener tag, 171, 181–182
  viewparam tag, 166–167
JavaServer Faces standard HTML render kit
  library, 220, 317
  html_basic TLD, 276
JavaServer Faces standard HTML tag library
  See also component tags
JavaServer Faces standard UI components, 217, 251
UIComponent component, 281
JavaServer Faces tag libraries, 112
JavaServer Faces core tag library, 139, 171
JavaServer Faces HTML render kit tag library, 139
namespace directives, 140
JavaServer Faces technology, 44, 59–60, 103–110
See also component rendering model, component
tags, conversion model, event and listener model,
FacesContext class, JavaServer Faces standard UI
components, lifecycle of a JavaServer Faces
application, UI component behavioral interfaces,
UI component classes, validation model
See also component tags
See also Facelets
advantages, 105–106
bookmarkable URLs, 165–166, 166–167
composite components, 243–249
FacesServlet class, 321
features, 104–105
partial processing, 216
partial rendering, 216
partial state saving, 263, 269–270
using Ajax with, 227–242
Validator interface, 202
JavaServer Pages Standard Tag Library, See JSTL
javax.servlet.http package, 328
javax.servlet package, 328
JAX-RS, 61, 381–401
accessing XML documents, 412–418
advanced features, 403–432
annotations, 403–406
application overview, 384–385
conditional HTTP requests, 409–410
converting CDI managed beans to root resource
classes, 409
converting enterprise beans to root resource
classes, 408–409
entity providers, 389–390
examples, 396–401, 418–432
extracting Java type of request or response, 406
form parameters, 405–406
integrating with CDI, 408–409
integrating with EJB technology, 408–409
JAX-RS (Continued)
 introduction, 364–365
 path parameters, 404–405
 path templates, 385–387
 query parameters, 405
 reference implementation, 381
 request headers, 403–406
 request method designators, 382–395
 resource class methods, 407–408
 resource classes, 382–395
 resource methods, 382–395
 runtime content negotiation, 410–412
 runtime resource resolution, 407–408
 static content negotiation, 410–412
 subresource locators, 407
 subresource methods, 407
 subresources, 407–408
 URI, 403–406
 using JSON representations, 417–418
 using with JAXB, 412–418
JAX-WS, 66–67
 defined, 367
 examples, 368–376
 introduction, 364
 service endpoint interfaces, 368
 specification, 379
JAXB, 66
 examples, 418–432
 generating Java entity classes from XML
 schema, 415–417
 generating XML schema from Java classes, 414–415
 Java entity classes, 414–415
 returning Java entity classes in JAX-RS
 resources, 414–415
 schema generator, 414–415
 using with JAX-RS, 412–418
 with JSON, 417–418
 xjc schema compiler tool, 415–417
JAXBElement, in JAX-RS resource methods, 415–417
JAXP, 65–66
JCE, 697
JDBC API, 64, 790–791, 806
JMS, 63
 achieving reliability and performance, 842–851
JMS (Continued)
administered objects, 833–835
application client examples, 860–886
architecture, 829
basic concepts, 828–832
definition, 826
type, 826
examples, 491–497, 859–927
introduction, 825–828
Java EE platform, 827–828, 851–858
messaging domains, 829–831
programming model, 832–842

K
Kerberos, 697, 698
key pairs, 760
keystores, 698, 759–764
managing, 760
keytool utility, 760

L
lifecycle callback events, intercepting, 937–938
action and value-change event processing, 221
Apply Request Values phase, 213, 267
custom converters, 280, 281
getRenderedType method (Render Response phase), 272
immediate attribute, 258
Invoke Application phase, 215
performing encoding (Render Response phase), 265
Process Validations phase, 214
Render Response phase, 215
derenderResponse method, 212, 213, 214
respons?eComplete method, 212, 214, 215
Restore View phase, 213
saving state, 269
Update Model Values phase, 214–215
updateModels method, 214
Validator interface, 285
views, 213
listener classes, 328
defining, 328
listener interfaces, 328
listeners
HTTP, 702
IIOP, 702
local interfaces, 444
local transactions, 849–851
localization, 355–360
Bean Validation, 930
log, server, 77–78
login configuration, 719–722, 722–723
login method, 724–726
login modules, 779
logout method, 724–726

M
managed bean creation facility, 300–308
managed bean declarations, 259
  key-class element, 305
  list-entries element, 303
managed-bean element, 300–303, 307
managed-bean-name element, 302
managed-property element, 303–308
map-entries element, 303, 305
map-entry element, 305
null-value elements, 303
value element, 303
managed bean methods
  attributes
    See attributes referencing managed bean methods
    referencing
    See referencing managed bean methods
    writing
    See writing managed bean methods
managed bean properties, 176, 190, 191, 288
  bound to component instances, 198–199
UIData properties, 194–195
UIInput and UIOutput properties, 193
UISelectBoolean properties, 195–196
UISelectItem properties, 197
UISelectItems properties, 198
UISelectMany properties, 196
UISelectOne properties, 196–197
writing, 192–200
managed beans, 104
  See also value binding
  composite components, 244
  configuring in JavaServer Faces technology, 296–297
  conversion model, 220
  custom component alternative, 254
defined for CDI, 517–518
developing, 106–107, 113–114
event and listener model, 222
JavaServer Faces technology, 189–192
loading JavaScript, 238
managed beans (Continued)
  method binding, 149
  properties
    See managed bean properties
Managed Beans specification, 61–62, 515
MapMessage interface, 841
matrix parameters, JAX-WS, 395
message acknowledgment, 843–844
  bean-managed transactions, 857
  message-driven beans, 854
message bodies, 840–841
message consumers, 837–839
message consumption, 832
  asynchronous, 832, 870–875
  synchronous, 832, 860–870
message-driven beans, 58, 439–441
  accessing, 439
  coding, 493–495, 899–900, 905, 919
  examples, 491–497, 897–902, 902–910, 910–916, 916–927
  garbage collection, 453
  introduction, 853–855
  onMessage method, 440, 494–495
  requirements, 493–495
  transactions, 440, 794, 800
message headers, 839–840
message IDs, JMSMessageID message header field, 840
Message interface, 841
message listeners, 439, 838–839
  examples, 871, 904–905, 918–919
message producers, 836–837
message properties, 840
message security, 715
message selectors, 839
MessageBodyReader interface, 389–390
MessageBodyWriter interface, 389–390
MessageConsumer interface, 837–839
MessageListener interface, 838–839
MessageProducer interface, 836–837
messages
  integrity, 764
  MessageFormat pattern, 151, 172
  outputFormat tag, 151
  param tag, 151, 172
messages (Continued)
  parameter substitution tags, 172
  queueing messages, 203, 308
  securing, 699–700
  using the FacesMessage class to create a message, 310
messages, JMS
  body formats, 840–841
  browsing, 841–842
  definition, 829
  delivery modes, 845
  expiration, 846
  headers, 839–840
  introduction, 839–841
  persistence, 845
  priority levels, 845
  properties, 840
messaging, definition, 825–826
messaging domains, 829–831
  common interfaces, 831
  point-to-point, 830
  publish/subscribe, 830–831
metadata annotations
  resource adapters, 814–815
  security, 700
Metamodel API, 661–663
  using, 619, 663–664
method binding
  method-binding expressions, 224, 315
  method expressions, 268
method expressions, 186, 222
method invocations, intercepting, 936–937
method permissions, 742
  annotations, 743–745
mutual authentication, 764–768

N
  naming contexts, 65
  naming environments, 65
navigation model (Continued)
  configuring navigation rules, 313–317
  logical outcome, 201, 314
  NavigationHandler class, 225
  referencing methods that perform navigation, 187, 201
  writing a managed bean method to perform navigation processing, 201–202
NetBeans IDE, 71–72
NON_PERSISTENT delivery mode, 845
non-repudiation, 696

O
  ObjectMessage interface, 841
objects, administered, 833–835
  creating and removing, 863–864
observer methods, CDI, 545–546
onMessage method
  introduction, 838–839
  message-driven beans, 440, 494–495, 854

P
  package-appclient tool, 67
  parameters, extracting, 392–395
Part interface (Servlet), 348
path parameters
  JAX-RS, 394, 404–405
path templates, JAX-RS, 385–387
permissions, security policy, 702
  persistence
    BLOBs, 611
    cascade operations, 610–611
    CLOBs, 611
    collections, 581–583
    concurrent access to entity data, 677–681
    configuration, 599
    context, 595–600
    embeddable classes, 590
    entities, 579–590
    examples, 603–627
    JMS example, 902–910
persistence (Continued)
  JMS messages, 845
  locking strategies, 677–681
  many-to-many, 617
  maps, 582
  one-to-many, 606
  one-to-one, 605–606
  overview, 579–601
  persistence units, 599–600
  persistent fields, 581
  primary keys, 585–587, 607
  properties, 581
  queries, 579–601, 613–614, 630–631
    See also query language
  creating, 665
  Criteria, 661–671
  dynamic, 630
  executing, 671
  expressions, 667–668, 668–669
  joins, 666
  parameters, 631
  path navigation, 667
  results, 667–669, 669–671
  static, 630
  typesafe, 661–671
  query language, 588
  relationships, 604–606
  scope, 599–600
  second-level cache, 683–688
  self-referential relationships, 605
  string-based criteria queries, 673–675
  temporal types, 612
  persistence units
    query language, 629, 643
  PERSISTENT delivery mode, 845
  pluggable audit modules, 702
  pluggable authorization providers, 702
  point-to-point messaging domain, 830
    See also queues
  POJOs, 38
  policy files, 698
  primary keys, 606
    compound, 607–610
    defined, 585–587
  primary keys (Continued)
    examples, 586
    generated, 607
    principal, 706
  PrintWriter class, 333
  priority levels, for messages, 845
    JMSPriority message header field, 840
  producer fields
    CDI, 541–543, 543
    example, 561–567
  producer methods
    CDI, 524–525, 541–543
    example, 558–561
  programmatic security, 692, 701, 714, 740
    example, 754–757
  programming model, JMS, 832–842
  providers, JMS, 829
  proxies, 367
  public key certificates, 764
  public-key cryptography, 760
  publish/subscribe messaging domain
    See also topics
    durable subscriptions, 847–849
    introduction, 830–831

Q
  qualifiers, using in CDI, 519
  Quality of Service, 697
  query language
    ABS function, 654
    abstract schemas, 630, 632, 644
    ALL expression, 652
    ANY expression, 652
    arithmetic functions, 652–654
    ASC keyword, 659
    AVG function, 657
    BETWEEN expression, 636, 649
    Boolean literals, 647
    Boolean logic, 655
    case expressions, 654–655
    collection member expressions, 644, 651
    collections, 644–645, 651
    compared to SQL, 634, 643, 646
### query language (Continued)
- comparison operators, 637, 649
- CONCAT function, 653
- conditional expressions, 635, 647, 648, 656
- constructors, 658
- COUNT function, 657
- DELETE expression, 637
- DELETE statement, 632
- DESC keyword, 659
- DISTINCT keyword, 633
- domain of query, 629, 642, 643
- duplicate values, 633
- enum literals, 648
- equality, 656–657
- ESCAPE clause, 650
- examples, 613–614, 633–637
- EXISTS expression, 652
- FETCH JOIN operator, 645
- FROM clause, 632, 642–645
- grammar, 637–659
- GROUP BY clause, 632, 659
- HAVING clause, 632, 659
- identification variables, 632, 642, 643–644
- identifiers, 642–643
- IN operator, 645, 649–650
- INNER JOIN operator, 645
- input parameters, 635, 648
- IS EMPTY expression, 636
- IS FALSE operator, 656
- IS NULL expression, 636
- IS TRUE operator, 656
- JOIN statement, 634, 645
- LEFT JOIN operator, 645
- LEFT OUTER JOIN operator, 645
- LENGTH function, 653
- LIKE expression, 636, 650
- literals, 647–648
- LOCATE function, 653
- LOWER function, 653
- MAX function, 657
- MEMBER expression, 651
- MIN function, 657
- MOD function, 654
- multiple declarations, 643
- multiple relationships, 635
- named parameters, 634, 648
- navigation, 634–635, 635, 644, 647
- negation, 656
- NOT operator, 656
- null values, 650–651, 655–656
- numeric comparisons, 656
- numeric literals, 647
- operator precedence, 648–649
- operators, 648–649
- ORDER BY clause, 632, 659
- parameters, 633
- parentheses, 648
- path expressions, 630, 646–647
- positional parameters, 648
- range variables, 644
- relationship fields, 630
- relationships, 630, 634, 635
- return types, 657
- root, 644
- scope, 629
- SELECT clause, 632, 657–658
- setNamedParameter method, 634
- SIZE function, 654
- SQRT function, 654
- state fields, 630
- string comparison, 656
- string functions, 652–654
- string literals, 647
- subqueries, 651–652
- SUBSTRING function, 653
- SUM function, 657
- syntax, 632, 637–659
- TRIM function, 653
- types, 646, 656
- UPDATE expression, 632, 637
- UPPER function, 653
- WHERE clause, 632, 647–657
- wildcards, 650
- query parameters
  - JAX-RS, 393, 405
- query roots, 665–666
- Queue interface, 834–835
QueueBrowser interface, 841–842
JMS client example, 875–880
queues, 834–835
browsing, 841–842, 875–880
creating, 834–835, 863–864
injecting resources, 492
temporary, 846–847, 904–905

reliability, JMS (Continued)
message expiration, 846
message persistence, 845
message priority levels, 845
temporary destinations, 846–847
remote interfaces, 445
Remote Method Invocation (RMI), and
messaging, 825–826
request headers, JAX-RS, 403–406
request method designators
JAX-RS, 382–395
Request objects, JAX-RS, 411
request parameters, extracting, 392–395
request/reply mechanism
JMSCorrelationID message header field, 840
JMSReplyTo message header field, 840
temporary destinations and, 846
RequestDispatcher interface, 338
requests, 332
See also HTTP requests
customizing, 336
getting information from, 332
retrieving a locale, 356
Required transaction attribute, 857
resource adapters, 63, 790, 810–813
example, 819–823
metadata annotations, 814–815
security, 781–782
resource bundles, 355
Bean Validation, 930
resource classes, JAX-RS, 382–395
resource injection, 807–810
resource methods, JAX-RS, 382–395
resources, 790–791, 805–817
See also data sources
JMS, 853
ResponseBuilderFactory class, 389–390
responses, 333
See also HTTP responses
buffering output, 334
customizing, 336
setting headers, 332
RESTful web services, 61, 381–401
defined, 381–382
roles, 705
  application, 709–710
declaring, 723–724
mapping to groups, 709–710
mapping to users, 709–710
referencing, 743–745
security, 707–709, 723–724, 742, 743–745
rollback method, 799, 800, 801
rollback method (JMS), 849–851
rollbacks, See transactions, rollbacks
root resource classes, 382
run-as identity, 747–749

security (Continued)
groups, 705
introduction, 691–712
JAAS login modules, 779
Java SE, 697–698
login forms, 778
login modules, 779
mechanism features, 695
mechanisms, 697–700
message, 715
message-layer, 699–700
method permissions, 742, 743–745
overview, 692–697
policy domain, 706
programmatic, 692, 701, 714, 724–729, 740
programmatic login, 779
propagating identity, 747–749
realms, 704–705
resource adapters, 781–782
role names, 723–724, 743–745
roles, 705, 707–709, 723–724, 742
run-as identity, 747–749
simple walkthrough, 692–695
transport-layer, 699, 710–711
users, 705
web applications, 713–738
web components, 713–738
security constraints, 715–719
multiple, 718–719
security domain, 706
security identity
  propagating, 747–749
  specific identity, 748
security-role-mapping element, 709–710
security-role-ref element, 728–729
security role references, 728–729
security roles, 707–709, 742
send method, 836–837
server authentication, 764
server certificates, 759–764
server log, 77–78
servers, Java EE
  deploying on more than one, 910–916, 916–927
  running JMS clients on more than one, 880–886

S
SAAJ, 66
SASL, 697
schema, deployment descriptors, 701
schemagen, JAXB, 414–415
schemagen tool, 68
scopes
  servlets, 330–331
  using in CDI, 520–522
  using in JavaServer Faces technology, 296–297
secure connections, 710–711
Secure Sockets Layer (SSL), 710–711
security
  annotations, 700, 731–732, 740
  application, 696–697, 698
  application clients, 778–779
  callback handlers, 778, 779
  component-managed sign-on, 780–781
  constraints, 715–719
  container-managed sign-on, 780
  container trust, 749
  containers, 692–697, 700–701
  context for enterprise beans, 746–747
  declarative, 692, 701, 714, 740
deploying enterprise beans, 749
ELS applications, 780–783
end-to-end, 699–700
enterprise applications, 739–757
enterprise beans, 739–749
examples, 734–738, 750–754, 754–757
Index

Service methods, servlets, 332
Servlet interface, 328
ServletContext interface, 339
ServletInputStream class, 333
ServletOutputStream class, 333
ServletRequest interface, 332
ServletResponse interface, 333

Servlets, 44, 328
  binary data, 333
  character data, 333
  compiling, 457–458
  creating, 331–332
  examples, 93–101, 344–345, 456
  finalizing, 342
  initializing, 332
  lifecycle, 328–330
  lifecycle events, 328
  packaging, 457–458
  scope objects, 330–331
  service methods, 332, 343
  specifying initialization parameters, 332
  tracking service requests, 342
  uploading files with, 347–353

Session beans, 58, 437–439
  See also asynchronous method invocation
    activation, 450
    bean-managed concurrency, 469, 472
    business interfaces, 441
    clients, 437
    concurrent access, 469–472
    container-managed concurrency, 469, 470–472
    databases, 799
    eager initialization, 468
    examples, 456, 461–468, 468–476, 476–479, 897–902
    handling errors, 472
    no-interface views, 441
    passivation, 450
    requirements, 463
    singleton, 438, 468–476
    stateful, 437, 438
    stateless, 437–438, 439
    transactions, 794, 799, 800
    web services, 446, 477

Session interface, 835–836
sessions, 340–341
  associating attributes, 340
  associating with user, 341
  invalidating, 341
  notifying objects associated with, 340
  sessions, JMS, 835–836
    managing in enterprise bean applications, 853
    SessionSynchronization interface, 799
    setRollbackOnly method, 799, 801, 857
    sign-on
      component-managed, 780
      container-managed, 780
    Simple Authentication and Security Layer (SASL), 697
    SingleThreadModel interface, 331
    SOAP, 363–365, 367, 379
    SOAP messages, 50, 66
      securing, 699–700
    SOAP with Attachments API for Java (SAAJ), 66
    specialization, CDI, 540–541
    SQL, 64, 634, 643, 646
    SQL92, 655
    SSL, 699, 710–711, 717–718, 764
      connectors, GlassFish Server, 711
      handshake, 710
      verifying support, 711
    standard converters, 221
      converter tags, 177
      NumberConverter class, 176
      using, 175–180
    standard validators, 222
      using, 183–186
    state fields, query language, 630
    stereotypes, CDI, 550–551
    StreamMessage interface, 841
    string-based criteria queries, 673–675
    subresources, JAX-RS, 407–408
    subscription names, for durable subscribers, 847
    substitution parameters, defining, See messages, param tag
    synchronous message consumption, 832
      JMS client example, 860–870
T

 templating, Facelets, 119–121
temporary JMS destinations, 846–847
testing
  enterprise beans, 502–504
  unit, 502–504
TextMessage interface, 840
timeout events, intercepting, 938–939
timer service, 479–489
  automatic timers, 479, 483–484
  calendar-based timer expressions, 479–481
  cancelling timers, 484
  creating timers, 482–483
  examples, 485–488
  exceptions, 484
  getInfo method, 485
  getNextTimeout method, 485
  getTimeRemaining method, 485
  getting information, 485
  programmatic timers, 479, 482–483
  saving timers, 484
  transactions, 485
timestamps, for messages, JMSTimestamp message header field, 840
Topic interface, 834–835
topics, 834–835
  creating, 834–835, 863–864
  durable subscriptions, 847–849
  temporary, 846–847, 918–919
transactions, 789–790, 793–803
  application-managed, 596–597
  attributes, 795–798
  bean-managed, 800–801, 801, 856
  boundaries, 794, 799, 800
  business methods
    See business methods, transactions
  commits, 794, 799
  container-managed, 794–799, 856
  container-managed transaction demarcation, 794
  defined, 794
  distributed, 856–857
  examples, 891–897

transactions (Continued)
  exceptions
    See exceptions, transactions
  JDBC, 802
  JMS and enterprise bean applications, 853
  JTA, 800
  local, 849–851
  managers, 797, 800, 802
  message-driven beans
    See message-driven beans, transactions
    nested, 794, 800
  Required attribute, 857
  rollbacks, 794, 799, 800
  scope, 795
  session beans
    See session beans, transactions
  timeouts, 801–802
  timer service, 485
  web components, 803
  transport-guarantee element, 717–718
  transport guarantees, 717–718
  transport-layer security, 699, 710–711
  truststores, 759–764
  managing, 760

U

UI component behavioral interfaces, 218
  ActionSource interface, 218, 221, 262, 273
  ActionSource2 interface, 218, 262
  ClientBehaviorHolder interface, 219
  ConvertibleValueHolder interface, 218
  EditableValueHolder interface, 218, 262
  NamingContainer interface, 218, 262
  StateHelper interface, 263, 268, 269–270
  StateHolder interface, 219, 262, 269–270
  SystemEventListenerHolder interface, 219
  ValueHolder interface, 219, 262
UI component classes, 217, 219, 253
  See also custom UI components
  javax.faces.component package, 262
  UIColumn class, 217
  UICustomCommand class, 217, 220
  UICustomComponent class, 217, 220
UI component classes (Continued)
  UIComponentBase class, 217, 262, 265
  UIData class, 194–195, 217
  UIForm class, 218
  UIGraphic class, 218
  UIInput and UIOutput classes, 193
  UIInput class, 218, 221
  UIMessage class, 218
  UIMessages class, 218
  UIOutput class, 218, 220
  UIPanel class, 218
  UIPassword class, 218
  UIParameter class, 218
  UISelectBoolean class, 195–196, 218
  UISelectItem class, 197, 218
  UISelectItems class, 198, 218
  UISelectMany class, 196, 218
  UISelectOne class, 196–197, 218, 219
  UIViewRoot class, 218
UnavailableException class, 332
undeploying modules and applications, 92–93
Unicode character set, 360
unified expression language, See EL
Uniform Resource Identifiers (URIs), 381
URI path parameters, JAX-RS, 394
URI path templates
JAX-RS, 385
URL paths, 93
US-ASCII character set, 359
user-data-constraint element, 717–718
user data constraints, 716, 717–718
users, 705
  adding to GlassFish Server, 706–707
  managing, 706–707
UserTransaction interface, 799, 800, 801, 803
  message-driven beans, 856
  using pages, 123
UTF-8 character encoding, 360
utility classes, 447

V
validating input
  See Bean Validation
  See validation model
validation
  customizing, 929–930
  entities, 583–585
  groups, 931–932
  localization, 930
  messages, 930
  ordering, 931–932
validation model, 217, 222–223
See also validators
  referencing a method that performs validation, 187–188
  validator attribute, 148, 186, 187, 202
  Validator implementation, 222, 287
  Validator interface, 200, 202–203, 223
    custom validator tags, 286
    implementing, 284
    writing a managed bean method to perform validation, 202–203
Validator implementation classes, 184, 222
  DoubleRangeValidator class, 172, 184
  LengthValidator class, 172, 184
  LongRangeValidator class, 172, 184, 185–186
validator tags
  composite components, 245
  f:validator tag, 223, 286
  validateDoubleRange tag, 184
  validateLength tag, 184
  validateLongRange tag, 184, 185–186
validators, 213, 217
  custom validators, 172, 287–288
  default, 311–312
  registering, 185
value binding
  acceptable types of component values, 193
  component instances to bean properties
    See component binding
  component values and instances to managed bean properties, 288–293
  component values to implicit objects, 291–292
  component values to managed bean properties, 289–291
  properties, 193–198
  value attribute, 192, 259, 288, 289–291
  value-binding expressions, 289
value binding (Continued)
  value expressions, 194, 268, 292
  value-change events, 221, 273
    f:valueChangeListener tag, 254
    processValueChange(ValueChangeEvent) method, 273
    processValueChangeEvent method, 203
    referencing methods that handle value-change events, 188
    type attribute, 181
    ValueChangeEvent class, 181, 273
    valueChangeListener attribute, 149, 186, 203
    ValueChangeEvent class, 181–182, 203, 273
    ValueChangeListener implementation, 273
    valueChangeListener tag, 171, 181–182
    writing a managed bean method to handle value-change events, 203–204
  value expressions, 191
  ValueExpression class, 192
  Variant class, JAX-RS, 411

W
  W3C, 66, 367, 379
  WAR files, 51
  web applications, 81–101
    configuring, 82, 93
    deployment descriptors, 82
    document roots, 84
    establishing the locale, 356
    internationalizing and localizing, 355
    maintaining state across requests, 340–341
    parsing and formatting localized dates and numbers, 359
    presentation-oriented, 81
    providing localized messages, 356
    retrieving localized messages, 358
    securing, 713–738
    service-oriented, 81
    setting the resource bundle, 357
    specifying context parameters, 97–98
    specifying welcome files, 97
  web beans, See Contexts and Dependency Injection (CDI) for the Java EE platform
  web clients, 42, 81–101
    examples, 456
  web components, 44, 81–83
    applets bundled with, 44
    concurrent access to shared resources, 331
    forwarding to other web components, 339
    including other web resources, 339
    invoking other web resources, 338
    JMS and, 858
    mapping exceptions to error screens, 98–99
    mapping filters to, 336
    scope objects, 330–331
    securing, 713–738
    sharing information, 330
    transactions, 803
    types, 44
    utility classes bundled with, 44
    web context, 339
  web container, 48, 82
    loading and initializing servlets, 328
    mapping URLs to web components, 93
  web modules, 52, 84
    deploying, 90
    dynamic reloading, 91
    undeploying, 92–93
    updating, 91
    viewing deployed, 91
  web pages
    XHTML, 107, 112
  web-resource-collection element, 716–717
  web resource collections, 716
  web resources, 84
    Facelets, 123–124
    mapping filters to, 336
    unprotected, 716
  web services, 49–50
    See also enterprise beans, web services
    declaring references to, 101
    endpoint implementation classes, 476
    examples, 368–376, 476–479
    introduction, 363
    JAX-RS compared to JAX-WS, 363–365
  web.xml file, 84, 320, 701, 742
  welcome files, specifying, 97
Index

work flows, 438
writing managed bean methods, 200–204
  for handling action events, 202
  for handling value-change events, 203–204
  for performing navigation, 201–202
  for performing validation, 202–203
writing managed bean properties
  converters, 199–200
  listeners, 199–200
  validators, 199–200
WSDL, 50, 363–365, 367, 379
wsgen tool, 68
wsimport tool, 68

X

xc schema compiler tool, JAXB, 415–417
xc tool, 68
XML, 49–50, 367
XML schema
  mappings of Java classes to XML data
    types, 378–379
  mappings to Java data types, 377–378