Building J2EE™ Applications

Sun™ ONE Studio 5 Programming Series

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 817-2327-10
June 2003, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Contents

Figures

Before You Begin

The Java Community Process, supported by Sun Microsystems, Inc., has evolved standards for designing distributed, enterprise applications with the Java™ 2 Platform, Enterprise Edition (J2EE™ platform). The J2EE platform documentation listed in “Before You Read This Book” on page 10 covers these standards for application design and architecture.

This book is about how you implement these architectures with the Sun™ ONE Studio 5, Standard Edition developer tools. It is about using the IDE to combine components and create J2EE modules, making sure that all of the components interact in the way that the application design specifies. It is also about combining J2EE modules to create J2EE applications, making sure that the distributed interactions between the modules function in the way that the application design calls for.

Screen shots vary slightly from one platform to another. Although almost all procedures use the interface of the Sun ONE Studio 5 software, occasionally you might be instructed to enter a command at the command line. Here too, there are slight differences from one platform to another. For example, a Microsoft Windows command might look like this:

```
c:>cd MyWorkDir\MyPackage
```

A UNIX command might look like this:

```
% cd MyWorkDir/MyPackage
```

Before You Read This Book

This book is intended for anyone who uses the Sun ONE Studio 5 IDE to assemble, deploy, or execute J2EE applications. The first chapter summarizes the J2EE platform concepts of assembly and deployment, and it should benefit anyone seeking a general understanding of assembly and deployment.

This book assumes a familiarity with the following subjects:

- Java programming language
- J2EE concepts
- Web and application server software

This book requires a knowledge of J2EE concepts, as described in the following resources:

- Java 2 Platform, Enterprise Edition Blueprints
  `http://java.sun.com/j2ee/blueprints`
- *Java 2 Platform, Enterprise Edition Specification*
  `http://java.sun.com/j2ee/download.html#platformspec`
- *The J2EE Tutorial*
  `http://java.sun.com/j2ee/tutorial`
- *Java Servlet Specification Version 2.3*
  `http://java.sun.com/products/servlet/download.html#specs`
- *JavaServer Pages Specification Version 1.2*
  `http://java.sun.com/products/jsp/download.html#specs`

Familiarity with the Java API for XML-Based RPC (JAX-RPC) is helpful. For more information, see this web page:

`http://java.sun.com/xml/jaxrpc`

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any 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 available on or through any such sites or resources.

How This Book Is Organized

The J2EE platform makes possible a component-oriented approach to developing enterprise applications. Application developer encapsulate business logic in Enterprise JavaBeans™ (EJB™) components and web components. After creating components, developers assemble their components into modules, which are units of logic that perform recognizable business tasks. After assembling modules, developers assemble their modules into J2EE applications. J2EE applications perform entire business processes.

This book is about using the Sun ONE Studio 5 development environment to assemble components into modules and modules into applications. The book presents this information in a series of “scenarios.”

Chapter 1 summarizes the J2EE concepts of assembly and deployment. It defines the J2EE units of modules and applications and examines module and application deployment descriptors. This chapter also explains how to assemble modules and applications in the IDE. In particular, this chapter explains how to use module and application property sheets to set up module and application deployment descriptors.

Chapter 2 is a scenario that shows how to assemble a web module. This chapter includes a short description of a web module that is used as the front end of a J2EE application. This chapter then shows how to program the web module.

Chapter 3 is a scenario that shows how to assemble an EJB module. This chapter includes a short description of an EJB module that is used in a J2EE application. This chapter then shows how to program the EJB module.

Chapter 4 is a scenario that shows how to assemble a J2EE application by combining a web module and an EJB module. This chapter includes a short description of a J2EE application that combines a web module and an EJB module. This chapter then shows how to assemble the application. This scenario features synchronous interaction between the two modules, using Java RMI.

Chapter 5 is a scenario that shows how to set up asynchronous communications between modules using a message-driven enterprise bean, or MDB. This chapter includes a short description of asynchronous communication used in a business application. This chapter then shows how to program both the sending and receiving sides of the application. The scenario in this chapter features a web module communicating with an EJB module, but the example can be applied to other combinations of modules.

Chapter 6 explains how to program container-managed transactions with the IDE.

Chapter 7 explains how to secure the resources in a J2EE application using the IDE. This chapter shows how to set up security roles at the module level and how to use the security roles to restrict access to web module resources and enterprise bean methods. This chapter also shows how to map security roles when modules are assembled into an application.

Chapter 8 explains how to deploy and execute assembled applications. In particular, it explains how to tailor an application for a specific server product before deployment.

Appendix A looks at the mechanism the IDE uses to interact with web and application servers. It includes a detailed account of the deployment process.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.cvspass` file. Use `DIR` to list all files. `Search is complete.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`> login` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* save your changes.
AaBbCc123	Command-line variable; replace with a real name or value	To delete a file, type **DEL** *filename.*

Related Documentation

Sun ONE Studio 5 documentation includes books delivered in Acrobat Reader (PDF) format, release notes, online help, readme files for example applications, and Javadoc™ documentation.

Documentation Available Online

The documents described in this section are available from the docs.sun.com℠ web site and from the documentation page of the Sun ONE Studio Developer Resources portal at http://forte.sun.com/ffj/documentation.

The docs.sun.com web site (http://docs.sun.com) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index that is installed with the product on your local system or network.

- Release notes (HTML format)

  Available for each Sun ONE Studio 5 edition. Describe last-minute release changes and technical notes.

  - *Sun ONE Studio 5, Standard Edition Release Notes* - part no. 817-2337-10

- Getting Started guides (PDF format)

  Describe how to install the Sun ONE Studio 5 integrated development environment (IDE) on each supported platform and include other pertinent information, such as system requirements, upgrade instructions, application server information, command-line switches, installed subdirectories, database integration, and information on how to use the Update Center.

  - *Sun ONE Studio 5, Standard Edition Getting Started Guide* - part no. 817-2318-10
  - *Sun ONE Studio 4, Mobile Edition Getting Started Guide* - part no. 817-1145-10

- Sun ONE Studio 5 Programming series (PDF format)

  This series provides in-depth information on how to use various Sun ONE Studio 5 features to develop well-formed J2EE applications.

  - *Building Web Components* - part no. 817-2334-10

    Describes how to build a web application as a J2EE web module using JSP pages, servlets, tag libraries, and supporting classes and files.

  - *Building J2EE Applications* - part no. 817-2327-10

    Describes how to assemble EJB modules and web modules into a J2EE application and how to deploy and run a J2EE application.

  - *Building Enterprise JavaBeans Components* - part no. 817-2330-10

    Describes how to build EJB components (session beans, message-driven beans, and entity beans with container-managed persistence or bean-managed persistence) using the Sun ONE Studio 5 EJB Builder wizard and other components of the IDE.

  - *Building Web Services* - part no. 817-2324-10

    Describes how to use the Sun ONE Studio 5 IDE to build web services, to make web services available to others through a UDDI registry, and to generate web service clients from a local web service or a UDDI registry.

- *Using Java DataBase Connectivity* - part no. 817-2332-10

  Describes how to use the JDBC productivity enhancement tools of the Sun ONE Studio 5 IDE, including how to use them to create a JDBC application.

- Sun ONE Studio 5 tutorials (PDF format)

  These tutorials demonstrate how to use the major features of Sun ONE Studio 5, Standard Edition:

  - *Sun ONE Studio 5 Web Application Tutorial* - part no. 817-2320-10

    Provides step-by-step instructions for building a simple J2EE web application.

  - *Sun ONE Studio 5 J2EE Application Tutorial* - part no. 817-2322-10

    Provides step-by-step instructions for building an application using EJB components and web services technology.

  - *Sun ONE Studio 4, Mobile Edition Tutorial* - part no. 816-7873-10

    Provides step-by-step instructions for building a simple application for a wireless device, such as a cellular phone or personal digital assistant (PDA). The application you build is compliant with the Java 2 Platform, Micro Edition (J2ME™ platform) and conforms to the Mobile Information Device Profile (MIDP) and Connected, Limited Device Configuration (CLDC).

You can also find the completed tutorial applications at:
`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

Online Help

Online help is available in the Sun ONE Studio 5 IDE. You can open help by pressing the help key (F1 in Microsoft Windows and Linux environments, Help key in the Solaris environment), or by choosing Help → Contents. Either action displays a list of help topics and a search facility.

Examples

You can download examples that illustrate a particular Sun ONE Studio 5 feature, as well as completed tutorial applications, from the Sun ONE Studio Developer Resources portal at:

`http://forte.sun.com/ffj/documentation/tutorialsandexamples.html`

The site includes the applications that are used in this document.

Javadoc Documentation

Javadoc documentation is available within the IDE for many Sun ONE Studio 5 modules. Refer to the release notes for instructions on installing this documentation.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table.

Type of Documentation	Format and Location of Accessible Version
Books and tutorials	HTML at `http://docs.sun.com`
Mini-tutorials	HTML at `http://forte.sun.com/ffj/tutorialsandexamples.html`
Integrated example readmes	HTML in the example subdirectories of *s1studio-install-directory*`/examples`
Release notes	HTML at `http://docs.sun.com`

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

`http://www.sun.com/service/contacting`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

Please include the part number (817-2327-10) of your document in the subject line of your email.

CHAPTER 1

Assembly, Deployment, and Execution Basics

The modular nature of developing with the J2EE platform means that you combine smaller units to create larger ones. You combine components to create modules, and you combine modules to create applications. Combining smaller J2EE software units to create larger units is known as assembly.

The modules and applications that you assemble need runtime services, such as container-managed persistence, container-managed transactions, and container-managed security validation, which are provided by the J2EE platform. When you assemble a module or an application, you must determine which runtime services are required. You must specify those services in a J2EE deployment descriptor.

This chapter describes some basic characteristics of J2EE modules and applications that influence the assembly process. It also introduces the basics of assembling with the Sun ONE Studio 5 developer tools.

Assembly Basics

J2EE assembly is a process that includes a number of separate tasks. When you assemble an application or a module correctly, you can deploy the application or module to a J2EE application server and execute the module or application.

The greatest obstacle to successful assembly is the variability of the assembly process. Every module or application you assemble requires a different combination of runtime services and, therefore, a different set of assembly tasks. There are no standard procedures for assembling modules and applications. You must understand what a correctly assembled module or application is before you begin the assembly process. This section provides background information on the J2EE platform that helps you recognize a correctly assembled module or application.

J2EE Applications Are Modular

A J2EE application is a set of modules. The modules in the application are sets of components. The J2EE platform's mechanism for combining components into modules and modules into applications is the deployment descriptor. The deployment descriptor is a "list of ingredients" for a module or an application.

- The deployment descriptor for an application lists the modules in the application.
- The deployment descriptor for a module lists the components in the module.

To understand why the J2EE platform uses deployment descriptors, consider how the source code for an application is deployed and executed. At development time, components exist as a number of source files in your development environment. These source files cannot be executed until you deploy them to a J2EE application server. The components must execute in the runtime environment that is provided by the application server.

Deploying an application compiles the source files that are listed in the application's deployment descriptor and installs the compiled files in directories that are managed by the application server. The source files are actually compiled into a J2EE application when you deploy. After you deploy, you can execute the deployed application in the application server's environment.

The deployment descriptor is a development time mechanism for listing the files that will be deployed together as a module or an application. When you assemble a module or an application at development time, you do not actually modify the source files. You prepare a deployment descriptor that describes your module or application for the deployment process.

Deployment descriptors are XML files. They use specific XML tags to identify an application and the modules that make up the application (or to identify a module and the components that make up the module). CODE EXAMPLE 1-1 shows the deployment descriptor for a J2EE application named `CatalogApp`. This deployment descriptor lists the modules in the `CatalogApp` application.

CODE EXAMPLE 1-1 Deployment Descriptor for `CatlaogApp`

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
      1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">
<application>
 <?xml version="1.0" encoding="UTF-8"?>
 <display-name>CatalogApp</display-name>
 <description>J2EE Application CatalogApp</description>
 <module>
  <ejb>CatalogData.jar</ejb>
  <alt-dd>CatalogData.xml</alt-dd>
 </module>
 <module>
```

CODE EXAMPLE 1-1 Deployment Descriptor for `CatlaogApp` *(Continued)*

```
   <web>
    <web-uri>CatalogWebModule.war</web-uri>
    <context-root>catalog</context-root>
   </web>
   <alt-dd>CatalogWebModule.xml</alt-dd>
  </module>
</application>
```

The `CatalogApp` application contains two modules, which are named `CatalogData` and `CatalogWebModule`. The deployment descriptor identifies the modules with `<module>` tags.

When you deploy the `CatalogApp` application, the application server reads the application's deployment descriptor. Each of the modules that are listed in the `CatalogApp` deployment descriptor has its own module-level deployment descriptor. The application server proceeds to read the deployment descriptors for the two modules. These deployment descriptors identify the source files for the J2EE components in the two modules. CODE EXAMPLE 1-2 and CODE EXAMPLE 1-3 show the module-level deployment descriptors.

When you work with the Sun ONE Studio 5 IDE, the IDE prepares the deployment descriptors for you. You do not write the deployment descriptor tags, but you should understand that the IDE prepares the deployment descriptors for you while you work in the IDE.

J2EE Applications Are Supported by the J2EE Runtime Environment

At runtime, J2EE applications use services that are provided by a J2EE application server. These services include container-managed persistence, container-managed transactions and container-managed security validation.

Applications, and the modules in applications, must tell the application server which services they need. The mechanism for telling the application server which services are needed is the deployment descriptor.

For example, consider J2EE container-managed transactions. To use the container-managed transaction service you must tell the J2EE application server what transaction services are needed. When you assemble enterprise beans into an EJB module, you define transaction boundaries by setting each enterprise bean's transaction attribute property. The IDE includes the value of each enterprise bean's transaction attribute property in the deployment descriptor.

CODE EXAMPLE 1-2 shows the deployment descriptor for an EJB module named `CatalogData` (the `CatalogData` module is one of the two modules listed in CODE EXAMPLE 1-1). The tags with the transaction attribute values appear at the end of the deployment descriptor.

CODE EXAMPLE 1-2 EJB Module Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
       2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
 <display-name>CatalogData</display-name>
 <enterprise-beans>
  <session>
   <display-name>CatalogManagerBean</display-name>
   <ejb-name>CatalogManagerBean</ejb-name>
   <home>CatalogBeans.CatalogManagerBeanHome</home>
   <remote>CatalogBeans.CatalogManagerBean</remote>
   <ejb-class>CatalogBeans.CatalogManagerBeanEJB</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Container</transaction-type>
   <ejb-local-ref>
    <ejb-ref-name>ejb/ItemBean</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>CatalogBeans.ItemBeanLocalHome</local-home>
    <local>CatalogBeans.ItemBeanLocal</local>
    <ejb-link>ItemBean</ejb-link>
   </ejb-local-ref>
   <ejb-local-ref>
    <ejb-ref-name>ejb/ItemDetailBean</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>CatalogBeans.ItemDetailBeanLocalHome</local-home>
    <local>CatalogBeans.ItemDetailBeanLocal</local>
    <ejb-link>ItemDetailBean</ejb-link>
   </ejb-local-ref>
  </session>
  <entity>
   <display-name>ItemBean</display-name>
   <ejb-name>ItemBean</ejb-name>
   <local-home>CatalogBeans.ItemBeanLocalHome</local-home>
   <local>CatalogBeans.ItemBeanLocal</local>
   <ejb-class>CatalogBeans.ItemBeanEJB</ejb-class>
   <persistence-type>Container</persistence-type>
   <prim-key-class>java.lang.String</prim-key-class>
   <reentrant>False</reentrant>
   <abstract-schema-name>ItemBean</abstract-schema-name>
   <cmp-field>
    <field-name>itemsku</field-name>
```

CODE EXAMPLE 1-2 EJB Module Deployment Descriptor *(Continued)*

```
    </cmp-field>
    <cmp-field>
     <field-name>itemname</field-name>
    </cmp-field>
    <primkey-field>itemsku</primkey-field>
    <query>
     <query-method>
      <method-name>findAll</method-name>
      <method-params/>
     </query-method>
     <ejb-ql>SELECT Object (I) FROM ItemBean AS I</ejb-ql>
    </query>
   </entity>
   <entity>
    <display-name>ItemDetailBean</display-name>
    <ejb-name>ItemDetailBean</ejb-name>
    <local-home>CatalogBeans.ItemDetailBeanLocalHome</local-home>
    <local>CatalogBeans.ItemDetailBeanLocal</local>
    <ejb-class>CatalogBeans.ItemDetailBeanEJB</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <abstract-schema-name>ItemDetailBean</abstract-schema-name>
     <cmp-field>
      <field-name>itemsku</field-name>
     </cmp-field>
     <cmp-field>
      <field-name>description</field-name>
     </cmp-field>
     <primkey-field>itemsku</primkey-field>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
   <container-transaction>
    <description>This value was set as a default by Sun ONE Studio.</description>
    <method>
     <ejb-name>CatalogManagerBean</ejb-name>
     <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
    <description>This value was set as a default by Sun ONE Studio.</description>
     <method>
      <ejb-name>ItemBean</ejb-name>
      <method-name>*</method-name>
     </method>
```

CODE EXAMPLE 1-2 EJB Module Deployment Descriptor *(Continued)*

```
     <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
    <description>This value was set as a default by Sun ONE Studio.</description>
     <method>
     <ejb-name>ItemDetailBean</ejb-name>
     <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
   </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

When the application that contains this EJB module is deployed and executed, the application server recognizes the transaction boundaries that are specified in the deployment descriptor. The application server opens and commits transactions (or rolls them back) at the specified points.

The other runtime services are handled much like container-managed transactions. Each service has its own deployment descriptor tags that indicate exactly what service is required. The IDE writes these tags for you automatically. You do not need to learn the tags that are used in the deployment descriptors.

J2EE Applications Are Distributed

In addition to being modular and making use of the J2EE platform's runtime services, J2EE applications are distributed. Each module in an application can be deployed to a different machine to create a distributed application. FIGURE 1-1 shows an application that is composed of two modules. This application implements a typical multi-tiered application architecture.

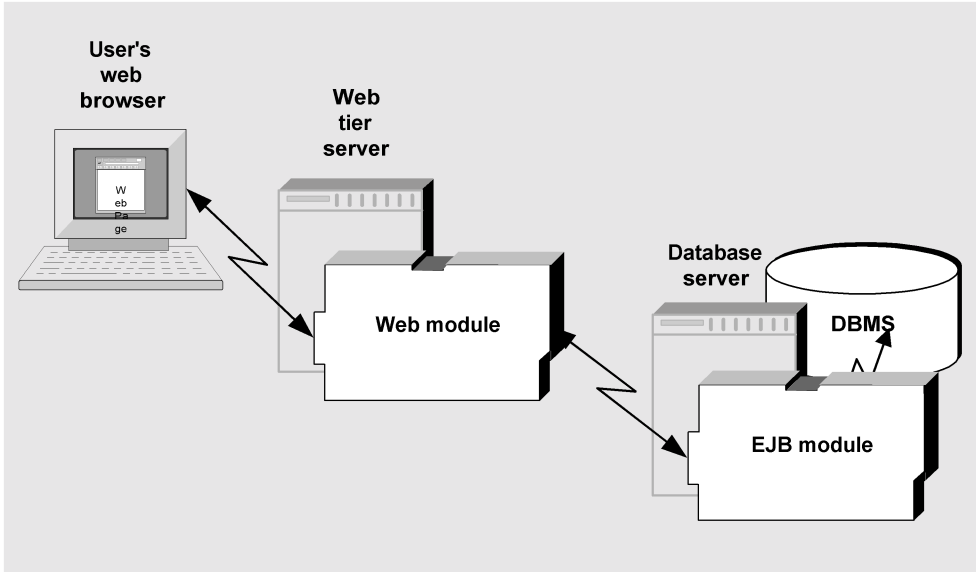

FIGURE 1-1 Multi-tiered Application Using J2EE Components and Modules

The web module is deployed on a machine that is dedicated to HTTP interactions with users. The application server provides the HTTP connections. Using the HTTP connections, the application server sends the web pages that are defined in the web module to browsers running on user desktop machines.

The EJB module is deployed on another machine, which is dedicated to database operations. The web module uses Java RMI to communicate with the EJB module. The application server provides the runtime support for the Java RMI interaction.

The J2EE platform provides several technologies that support distributed interactions between modules. The J2EE platform's distributed technologies include those listed below:

- **Web-based communications over HTTP connections.** This is often used between end users and applications.
- **Synchronous method invocation, using Java RMI-IIOP.** This is used to call enterprise bean methods.
- **Asynchronous messaging, using the Java Message Service (JMS).** Messages can addressed to queues or to topics.

The J2EE platform provides different kinds of components that support different kinds of interaction. For example, the J2EE platform uses message-driven enterprise beans to support asynchronous messaging between modules.

Deciding which technology to use for the distributed interactions in an application is an application design task. Deciding which type of component to use is also a design task. When you assemble an application, you need to know what kind of interaction was designed and how to implement it. You implement interactions by performing such assembly tasks as setting up EJB references to implement Java RMI interactions, and setting up queues to implement JMS messaging interactions.

The J2EE platform also supports interactions between J2EE modules and external resources such as data sources. The J2EE technologies that support these interactions include the following:

- JDBC™ technology
- Container-managed persistence

When you assemble an application, you need to make sure that any external resources that are used by the application are identified. The development time mechanism for identifying external resources is the deployment descriptor.

CODE EXAMPLE 1-3 shows the deployment descriptor for a web module named `CatalogWebModule`. This module and an EJB module named `CatalogData` are assembled into a J2EE application. The technology that is used for the interaction between the two modules is Java RMI. The Java RMI interaction requires a remote EJB reference.The remote EJB reference is declared near the end of CODE EXAMPLE 1-3, with the `<ejb-ref>` tag.

CODE EXAMPLE 1-3 Web Module Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
 <servlet>
  <servlet-name>AllItemsServlet</servlet-name>
  <servlet-class>AllItemsServlet</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>DetailServlet</servlet-name>
  <servlet-class>DetailServlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>AllItemsServlet</servlet-name>
  <url-pattern>/servlet/AllItemsServlet</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>DetailServlet</servlet-name>
  <url-pattern>/servlet/DetailServlet</url-pattern>
 </servlet-mapping>
 <session-config>
```

CODE EXAMPLE 1-3 Web Module Deployment Descriptor *(Continued)*

```
   <session-timeout>
    30
   </session-timeout>
  </session-config>
  <welcome-file-list>
   <welcome-file>
    index.jsp
    </welcome-file>
    <welcome-file>
     index.html
    </welcome-file>
    <welcome-file>
     index.htm
    </welcome-file>
  </welcome-file-list>
  <ejb-ref>
   <ejb-ref-name>ejb/CatalogManagerBean</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>CatalogBeans.CatalogManagerBeanHome</home>
   <remote>CatalogBeans.CatalogManagerBean</remote>
   <ejb-link>CatalogManagerBean</ejb-link>
  </ejb-ref>
</web-app>
```

When you work in the Sun ONE Studio 5 IDE, you do not code the deployment descriptors for your modules and applications. You work with the IDE's visual representations of components, modules, and applications, and the IDE prepares the deployment descriptors for you.

Visual Representations of Modules and Applications

Most explanations of J2EE assembly focus on the contents of deployment descriptor files, such as the files shown in CODE EXAMPLE 1-1, CODE EXAMPLE 1-2, and CODE EXAMPLE 1-3. These explanations tell you how to code the XML. The Sun ONE Studio 5 IDE provides visual representations of components, modules, and applications. Instead of coding deployment descriptor files, you work with Explorer window nodes that represent components, modules, and applications.

When you assemble in the IDE, the Explorer creates a visual representation of the module or application you are creating. The nodes that represent applications have subnodes for their modules, and the nodes that represent modules have subnodes for their components. While you work with the visual representation, the IDE creates a matching deployment descriptor.

Each node has a property sheet that allows you to configure the component, module, or application that the node represents. Most of the properties map to deployment descriptor tags. (There are more properties than deployment descriptor tags.) You configure a component, module, or application for deployment by setting its properties, and the IDE adds tags to its deployment descriptor. These tags identify the services that are needed from the application server.

The sections that follow introduce the IDE's visual representations of modules and applications.

Web Modules

Web modules have a standard directory structure (for more information see *Building Web Components*), and this structure is represented in the Explorer window. FIGURE 1-2 shows a web module in the Explorer. The nodes and subnodes of a web module represent the individual directories and files in the module.

The top-level node for a web module represents the web module's top-level directory. For the IDE to recognize a directory as a web module you must mount the directory as an Explorer window filesystem. If you mount a web module directory in the Explorer as a subdirectory of another filesystem, the IDE will not recognize the web module directory as a web module.

The top-level node has a subnode for a WEB-INF directory. The WEB-INF directory has subnodes for a lib subdirectory, which is used for web components in JAR file format, and a classes subdirectory, which is used for any web components in `.java` file format. The WEB-INF node also has a web subnode that represents the module's deployment descriptor file. This is the standard directory structure for a web module.

Web modules also have nodes for components and resources that are added by developers. FIGURE 1-2 shows a node for an HTML page named `index.html`. The `classes` directory contains nodes for two servlet classes, which are named `AllItemsServlet` and `ItemDetailServlet`.

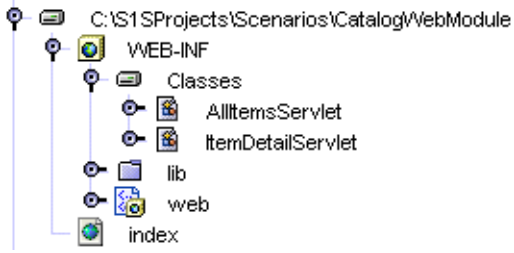

FIGURE 1-2 Web Module Node and Subnodes

Notice that this representation of a web module corresponds to a specific directory and its contents. The deployment descriptor is a file named `web.xml`, which is represented in the Explorer by the web node. The `web.xml` file is part of the web module's source code.

EJB Modules

EJB modules are represented differently from web modules. The top-level node for an EJB module does not represent a particular directory and its contents. Instead, the EJB module node represents the module's deployment descriptor. The EJB module node functions as a list of enterprise beans. These enterprise beans can be in one directory or in a number of directories in different filesystems. The top-level node, which represents the deployment descriptor, tracks where the source code for the components is located.

Representing EJB modules with logical nodes allows you to combine enterprise beans from different directories in one EJB module. It also keeps the configuration information in the deployment descriptor separate from the source code. When you deploy an EJB module, a deployment descriptor XML file is generated and the source files for the components that are identified in the deployment descriptor are compiled into an EJB JAR file.

FIGURE 1-3 shows an EJB module in the Explorer window. This module has subnodes for three enterprise beans that have been included in the module. Each of these enterprise beans can be in a different directory. It is even possible for a single enterprise bean to be in several different directories. For example, the source code for the enterprise bean's interfaces could be in one directory and the classes that implement those interfaces could be in a different directory.

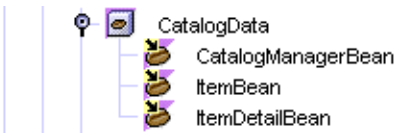

FIGURE 1-3 EJB Module Node and Subnodes

J2EE Applications

J2EE applications are also represented by logical nodes. Like an EJB module node, the top-level J2EE application node does not represent a single directory or filesystem. Instead, the application node represents a deployment descriptor for the application and functions as a list of the modules that make up the application. The source code for these modules can be in more than one directory or filesystem.

The IDE maintains the application-level deployment descriptor separately from the source code. You can include the same source code in more than one J2EE application. The deployment process compiles all the source files and associates the compiled versions with the deployment descriptor in an EAR file.

FIGURE 1-4 shows a J2EE application in the Explorer. The modules shown FIGURE 1-2 and FIGURE 1-3 have been added to the application. The modules are represented by subnodes of the application node.

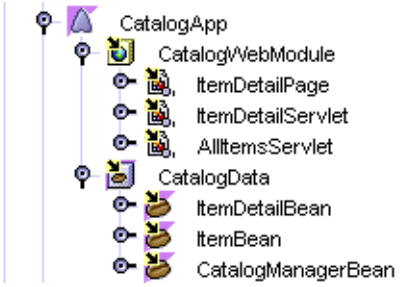

FIGURE 1-4 J2EE Application Node and Its Subnodes

Property Sheets

Every node that represents a J2EE module or application has a property sheet. The property sheet has properties that let you specify what services the module or application needs from the application server. These properties correspond to the tags that appear in the module or application deployment descriptor. When you set the values of properties, you supply the information that will be used in the deployment descriptor. You work with the property sheet instead of editing and formatting the XML deployment descriptor with a text editor.

- In the case of a web module, the deployment descriptor exists as a file, and the file appears in the Explorer window. You can see this in FIGURE 1-2. The values that you specify in the web module deployment descriptor are associated with the source files.

- In the case of an EJB module or a J2EE application the deployment descriptor file is not generated until you deploy the module or application. Deployment compiles the source files and generates an archive file, a EJB Jar or an EAR file. deployment descriptor file is included in the archive. The deployment descriptor is associated with a specific archived copy of the source files.

The properties have property editors that help you select the correct values. When you have the property sheet open, you open the editors for individual properties by clicking a property and clicking the ellipsis (…) button. Procedures for using the property editors vary widely—some let you browse for IDE nodes, some let you open another level of dialog, and so on. For the more complex property editors, online help is available.

The property sheets have a number of tabs. The tab labeled Properties lists the standard properties defined by the J2EE specifications. Other tabs have the names of application server products. These tabs collect additional information that is not defined in the J2EE specifications but is required by a specific application server product. These tabs are the property sheet's server-specific tabs. When you assemble a module or application you need to work with the standard properties and with the server-specific properties for the server product that you are using.

Deployment Basics

Deployment is the process of compiling the source files that make up a J2EE application and installing the compiled files into directories that are managed by a J2EE application server. Deployment is carried out by the application server or by software distributed with the server, such as a deployment tool or an administration tool.

The IDE communicates with application servers. It provides commands that deploy modules or applications to an application server and execute the modules or applications. You can deploy and execute without leaving the IDE. Deployment is performed by the plugin or the application server software, and execution happens in the application server's environment, but you manage deployment and execution in the IDE.

After you execute an application and test it, you can modify the source code, redeploy the application, and execute the new version. You can continue this process as long as necessary.

The actual procedure for deploying an application that you have assembled is very simple. You right-click the application node and choose the Deploy command. Before you deploy, however, the IDE must be set up to work with an application server. The steps for setting up an application server are summarized in the following list:

1. Install the application server.
2. Install the server plugin that enables communication between the IDE and the application server. The plugin for Sun ONE Application Server 7, Standard Edition is installed with the IDE. Plugins are available for other widely used application server products. Installing a plugin creates application server nodes in the Explorer's Runtime tab. (To learn more about the plugin, see Appendix A.)
3. Use the menu commands of the application server nodes to establish communications between the IDE and the application server.
4. Use the application's property sheet to specify the application server to which your application will be deployed.
5. Right-click the application node and choose the Deploy command. The deployment software reads the deployment descriptor represented by the application node and deploys the source files that are listed in the deployment descriptor.

Most installations of the IDE include Sun ONE Application Server 7, and the installer automatically configures the IDE to work with the application server. The *Sun ONE Studio 5, Standard Edition Getting Started Guide* explains this in detail.

Execution Basics

After you deploy an application you can execute the application. The IDE associates the Explorer node that represents the application with the deployed copy of the application. You can right-click the node and choose the Execute command, and the IDE instructs the application server to execute the deployed copy of the application.

A number of IDE nodes, including the J2EE application nodes, have Execute commands. When you right-click a node and choose Execute, the IDE, if necessary, deploys the application and then executes it. The results of the Execute command depend on the type of application. For example, if the application includes a web module, the Execute command will start a web browser and open the application's URL.

You can also execute deployed applications entirely in the application server's environment, without using the IDE. For example, to execute a deployed application that includes a web module, start a web browser and open one of the application's web pages.

Using This Book

The Java Community Process, supported by Sun Microsystems, Inc., has evolved standards for designing distributed, enterprise applications with J2EE components. The J2EE platform documentation listed in "Before You Read This Book" on page 10 covers these standards for application design and architecture.

This book is about how you implement these architectures with the Sun ONE Studio 5 IDE. It is about using the IDE to combine components and create J2EE modules, making sure that all of the components interact in the way that the application design specifies. It is also about combining J2EE modules to create J2EE applications, making sure that the distributed interactions between the modules function in the way that the application design calls for.

This book covers assembly by presenting a number of examples, or scenarios. Each scenario presents a realistic combination of components or modules and shows you how to assemble them into a module or an application. The business problems described in the scenarios are realistic, but this book is not really meant to be an exhaustive guide to designing J2EE applications.

The real purpose of these scenarios is to show you how to program specific kinds of interactions between components and modules. Once you have decided on your application design, you can use the scenarios in this book to help you program the interactions between the components and modules in your application.

You probably will not find everything you need in a single scenario, because each scenario focuses on one or two kinds of J2EE interaction, and your real-world J2EE application can include dozens or hundreds of components and interactions. For each kind of interaction, however, you should find an example in this book.

For example, to program one common type of J2EE application, with a web module and an EJB module, you can look at Chapter 2, which covers assembling a web module, Chapter 3, which covers assembling an EJB module, and Chapter 4, which covers assembling a web module and an EJB module into a J2EE application. Then, to see how you set up transactions, look at Chapter 6, and, for security, look at Chapter 7.

This book is your guide to developing distributed enterprise applications with the Sun ONE Studio 5 development environment. It shows you how to develop J2EE modules, how to program your modules for different kinds of interactions, and how to request enterprise services, such as security checking and transaction management, from the J2EE platform.

CHAPTER 2

Scenario: A Web Module

FIGURE 2-1 shows a web module and the interactions in which it participates. This module is part of a J2EE application, and the interactions shown in the figure are typical for web modules in J2EE applications. The web module interacts with users over HTTP connections (represented in the figure by the arrows labeled #1) and with middle-tier services provided by EJB modules (represented by the arrows labeled #3). Inside the web module, the web components interact with each other (represented by the arrows labeled #2). This chapter describes this web module and explains how to program the interactions shown in FIGURE 2-1.

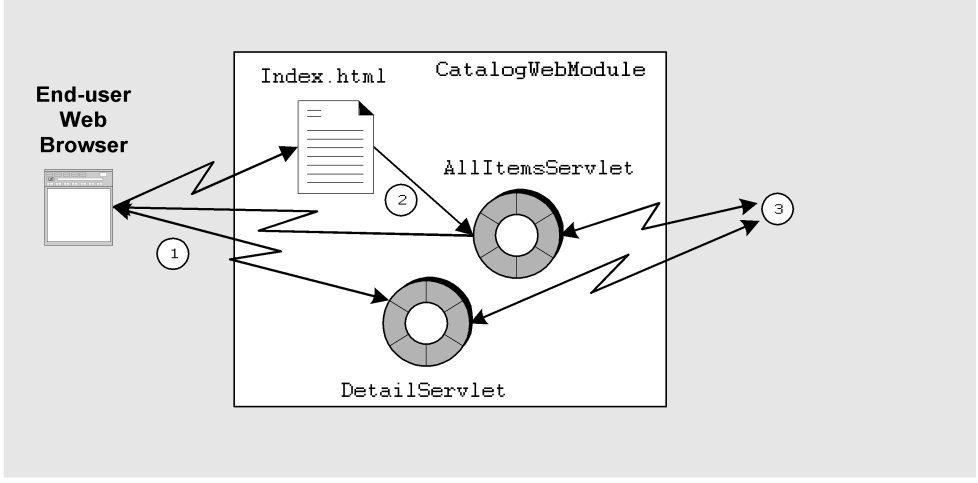

FIGURE 2-1 The `CatalogWebModule` Web Module

The Interactions in This Module

This scenario describes one possible use of the web module and interactions that are illustrated in FIGURE 2-1. In this scenario, the module is the front end for a J2EE application that supports a retail web site. The web module contains the web pages that are displayed to online shoppers who visit the site. The module also processes shopper input on those pages. Displaying pages and processing input are typical tasks for a web module in a J2EE application.

From the shopper's point of view this application is a series of web pages. Shoppers use web browsers to view the application's home page. Shoppers also provide input with the text fields, buttons, and other controls that appear on the web pages. From the developer's point of view, the application is a set of web components that receive HTTP requests and return HTTP and responses.

The web module in this scenario contains only a few components and processes only two different requests. Although simple, these components and requests show you how web components provide the necessary interaction between users, web module, and EJB module. The specific interactions that are covered in this scenario are outlined in the following list:

1. An online shopper opens a connection to the application by starting a web browser and opening the application's root URL. This action opens the application's home page. A real-world shopping site's home page would display a number of options, including requests for displaying items by category, requests for keyword search, requests for information about live customer service, and so on. In this simple example, the home page shows only one option, which is a link to another page that displays the entire catalog.

2. The shopper clicks the link. This action generates a request that is processed by the servlet that is named in the request, the `AllItemsServlet`. The `AllItemsServlet` processes the request by calling a business method of an EJB module, the `getAllItems` method, which returns the data.

3. `AllItemsServlet` prepares the data that is returned from the EJB module for display in the shopper's browser. The servlet combines the individual field values that are returned from the EJB module with HTML tags that format the field values. The `AllItemsServlet` writes this combination of field values and HTML tags to an output stream that is returned to the user's web browser. The shopper's web browser processes the HTML output and displays the catalog. In the catalog display, each item name is displayed as an HTML link to detailed information about the item.

4. The shopper browses the displayed catalog and clicks one of the links. This request is processed by another servlet, `DetailServlet`. `DetailServlet` calls another business method of the EJB module, `getOneItemDetail`, for the item detail.

5. `DetailServlet` processes the data returned from the EJB module for display in the shopper's web browser. Like `AllItemsServlet`, `DetailServlet` prepares an output stream that combines field values and HTML tags. The shopper's web browser processes this output stream and displays it as another web page.

The HTML outputs that are written by `AllItemsServlet` and `DetailServlet` contain only text. Although simple, these examples show you how web components can process HTTP requests by making remote methods calls to obtain data from an EJB module and then writing the data into an HTML output stream. An experienced web designer or web programmer can use this type of operation to write much more complex output.

The interaction between the web module and the EJB module is implemented by Java RMI method calls. Java RMI is required by the design of the EJB module. For more information about the design of the EJB module, see "The Interactions in This Module" on page 56.

For instructions on creating the web components, writing enterprise business logic in web components, and similar tasks, see *Building Web Components*.

Programming This Module

TABLE 2-1 summarizes the programming that is required to create the web module that is described in the preceding section and is illustrated in FIGURE 2-1.

TABLE 2-1 Programming Required for This Scenario

Application Element	Programming Required
Application server	None.
Web module	Create the web module. Create the welcome page, `index.html`. This page includes an HTML link that executes the servlet `AllItemsServlet`. Create two servlets, `AllItemsServlet` and `ItemDetailServlet`. Code the `processRequest` methods that process HTTP requests and generate HTTP responses. These `processRequest` methods: 1. Perform a Java Naming and Directory Interface (JNDI) lookup in order to call EJB module business methods. 2. Write the data obtained from the EJB module into the servlet's response. The HTML page output by `AllItemsServlet` contains an HTML link to `ItemDetailServlet`. Set up URL patterns for each servlet.
J2EE application	For instruction on adding a web module to a J2EE application, see Chapter 4.

The sections that follow show you how to perform many of these programming tasks. The method signatures specify the inputs and outputs of each interaction. The procedure sections show you how to connect these inputs and outputs to other components and other modules. Instructions for creating a web module or adding the web components to the module are not included. To learn about these tasks, see the online help or *Building Web Components*.

Creating the Welcome Page

The design for the retail web site calls for shoppers to begin their interaction with the site at a home page. The home page, a typical web site feature, provides an entry point for users that identifies the site and presents the options that are available to them. Shoppers view the home page and choose the features they want to use.

The shopper's first action is to start a browser and open the URL for the shopping site. Inside the web module, this URL has been mapped to the application's context root. If you want the application to display the home page whenever users open the site's URL, identify the home page as the welcome page.

Creating the HTML Page

In the Explorer window's hierarchy of nodes, the HTML file for the welcome page must be at the same level as the web module's WEB_INF node. For an example, see FIGURE 1-2.

To create an HTML file at the correct level:

1. **Right-click the node for the filesystem that contains the green WEB-INF node, and choose New → All Templates.**

   The New wizard's Choose Template page opens.

2. **Choose the HTML File template.**

   a. **In the Select a Template field, expand the JSPs & Servlets node and select the HTML File node.**

   b. **Click Next**

      The New wizard's New Object Name page opens.

3. **In the name field, type `index`. Click Finish.**

   The IDE creates the new HTML file and represents it in the Explorer with a new node named `index`. The Source Editor opens with the cursor in the new file.

4. **Type the HTML code for your welcome page.**

   CODE EXAMPLE 2-1 shows the HTML code for the simple welcome page that is used in this scenario.

CODE EXAMPLE 2-1 Welcome Page for the Catalog Display Module

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Online Catalog</title>
  </head>
  <body>
<h2>
Inventory List Application
</h2>

<p>
```

CODE EXAMPLE 2-1 Welcome Page for the Catalog Display Module *(Continued)*

```
<a href="allItems">Display the Catalog
</a>

</body>
</html>
```

This welcome page presents only one option to users, which is a text link named `Display the Catalog`. This link uses a URL pattern to specify one of the servlets in the module, the `AllItemsServlet`. When a user clicks this link, the browser sends another request to the web module. The request is processed by executing the `AllItemsServlet`. The next page the user sees is the page that is output by `AllItemsServlet`. To see the page that is output by the `AllItemsServlet`, refer to CODE EXAMPLE 2-2.

The welcome page for a real-world web site includes many links to different functions, but each link follows the principle that is demonstrated in this example. Pages that are displayed to the user contain links or actions that generate HTTP requests. Each request is processed by some component in the web module, and the web module responds by writing out another page for the user to see.

In this example, the link specifies a servlet, but not a method of the servlet. When a request specifies only a servlet name, the default action is to execute the servlet's `doGet` method. You can also write links that specify one of the servlet's other methods, such as `doPost`. For more information on servlet methods, see *Building Web Components*.

Identifying Your Page as the Module's Welcome Page

When you create a web module in the IDE, the module has a Welcome Files property that lists the default names for the welcome page files. FIGURE 2-2 shows the Welcome Files property editor with the default names. When a user accesses the root URL for the application, the application server searches the module directory for files with these names. The first file found is displayed as the welcome page.

The easiest way to create a welcome file for a module is to create a file with one of these default names. In this scenario, for example, you created a file named `index.html`.

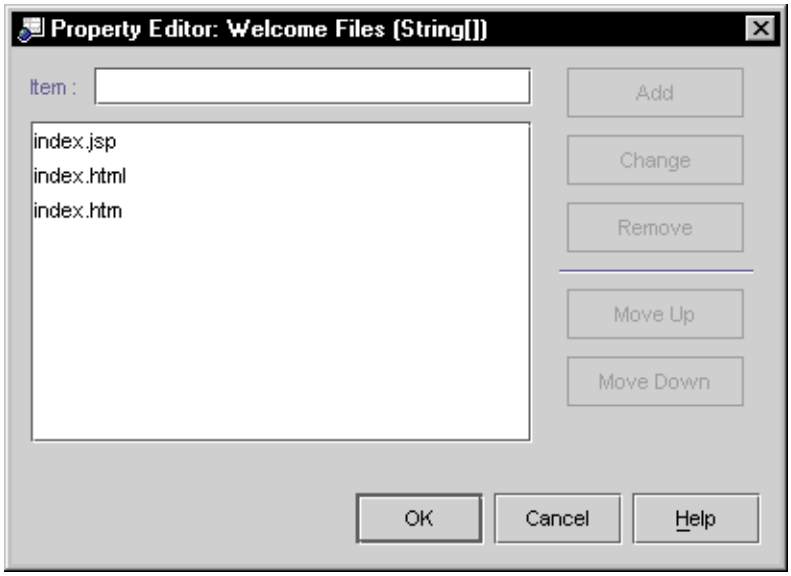

FIGURE 2-2 Welcome Files Property Editor

If you want to use a file with a different name for the module's welcome page, open the editor for the Welcome Files property and add the name of the file you want to use.

Programming the Servlet Methods

The servlets in this scenario obtain data from the EJB module by calling enterprise bean methods. There are two programming tasks for these method calls: coding the calling method and setting up a reference declaration, in the calling servlet, for the called enterprise bean. Coding the method body code is covered first.

Note – The servlets in this example are created with the IDE's servlet template. Servlets created with this template are HttpServlets, and they contain methods named `processRequest`. Both the `doGet` and `doPost` methods call `processRequest`, so you add the code that processes a request to the `processRequest` method.

The Method Body

CODE EXAMPLE 2-2 shows the implementation of the `processRequest` method in the `AllItemsServlet`. This method executes when a user clicks the `Display the Catalog` link that appears on the welcome page.

The URL pattern on the welcome page names the servlet `AllItemsServlet`, but the URL pattern does not specify a method. In this case the application server performs the default action and executes the servlet's `doGet` method. The `doGet` method calls the `processRequest` method.

CODE EXAMPLE 2-2 shows you how the `AllItemsServlet` obtains the catalog data from the EJB module and displays it to the user. There are three steps:

1. The `AllItemsServlet` uses a JNDI lookup to obtain a remote reference to a session enterprise bean in the `CatalogData` EJB module. For more information about the `CatalogData` module, see Chapter 3.
2. The `AllItemsServlet` calls the `getAllItems` method, a business method of the session bean.
3. The servlet writes the data that is returned by the remote method call into the HTML output stream. The outlet stream is returned to the user's browser window.

CODE EXAMPLE 2-2 The `processRequest` Method for the Servlet `AllItemsServlet`

```
protected void processRequest(HttpServletRequest request,
                              HttpServletResponse response)
                              throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  /* output your page here */
  out.println("<html>");
  out.println("<head>");
  out.println("<title>AllItemsServlet</title>");
  out.println("</head>");
  out.println("<body>");

  out.println("<h2>The Inventory List</h2>");

  out.println("<table>");
  out.println("<tr>");
  out.println("<td>Item");
  out.println("<td>Item SKU");
  out.println("<td>Detail");

  CatalogBeans.CatalogManagerBeanHome catHome;
  CatalogBeans.CatalogManagerBean catRemote;
```

CODE EXAMPLE 2-2 The `processRequest` Method for the Servlet `AllItemsServlet` *(Continued)*

```
try {
  InitialContext ic = new InitialContext();
  Object objref  = ic.lookup("java:comp/env/ejb/CatalogManagerBean");
  catHome = (CatalogBeans.CatalogManagerBeanHome) objref;
  catRemote = catHome.create();

  java.util.Vector allItems = catRemote.getAllItems();

  Iterator i = allItems.iterator();
  while (i.hasNext()) {

    CatalogBeans.iDetail itemDetail = (CatalogBeans.iDetail)i.next();
    out.println("<tr>" +
      "<td>" +
      itemDetail.getItemname() +

      "<td>" +
      itemDetail.getItemsku() +

      "<td>" +
      "<a href=\"" + response.encodeURL("itemDetail?sku=" +
      itemDetail.getItemsku()) +
      "\"> " +
      "Get Item Detail" +
      "</a>");
  }
}
catch (javax.naming.NamingException nE) {
  System.out.println("Naming Exception on Lookup" + nE.toString());
}
catch (javax.ejb.CreateException cE) {
  System.out.println("CreateException" + cE.toString());
}
catch (java.rmi.RemoteException rE) {
  System.out.println("RemoteException" + rE.toString());
}
catch (Exception e) {
  System.out.println(e.toString());
}

out.println("</table>");


out.println("</body>");
out.println("</html>");
```

CODE EXAMPLE 2-2 The `processRequest` Method for the Servlet `AllItemsServlet` *(Continued)*

```
  out.close();
}
```

The `lookup` statement specifies the `CatalogManagerBean`, but this string is actually the name of the reference, not the enterprise bean that is referenced. The enterprise bean's name is often used as the reference name to make it easier to remember which bean is meant. The referenced enterprise bean is actually specified in the reference declaration, which is covered in the next section.

The Reference Declaration for the EJB Reference

A web component, like the `AllItemsServlet`, that calls methods of an enterprise bean in an EJB module does so by means of EJB references. The two parts of an EJB reference are listed below:

- **JNDI lookup code**, which uses the JNDI naming facility to obtain a remote reference to a named enterprise bean.
- **A reference declaration**, which is part of the deployment descriptor. The reference declaration tells the application server which specific bean is referred to in the lookup code.

In this scenario, the lookup code is in the `processRequest` method of the `AllItemsServlet`. (See CODE EXAMPLE 2-2.) You can compile this code, but without the reference declaration it cannot return a reference at runtime. The reference declaration maps the reference name that is used in the lookup statement to the actual name of an enterprise bean.

To set up an EJB reference declaration for a web module:

1. **Right-click the web module's web node and choose Properties → References tab → EJB References → ellipsis (...) button.**

   The EJB references property editor opens.

2. **Click the Add button.**

   The Add EJB Reference dialog opens. Use this dialog to set up the reference declaration.

3. **To set up the reference declaration, type the reference name that is used in the `lookup` statement and the names of the home and remote interfaces that are used in the method calls.**

   FIGURE 2-3 shows the Add EJB Reference dialog box with these values in the fields. For the reference to work at runtime, for the JNDI lookup to return a remote reference to an enterprise bean, the reference must be linked to a specific enterprise bean in the same application. The reference must linked before you deploy and execute the application, but it need not be done now.

   At this point in development, you can leave the reference unlinked and link it later, after you assemble the web module into a J2EE application. In some circumstances, you might choose to resolve the reference at this stage of development. The conditions you should consider are listed below:

   - If the enterprise bean is not in your development environment, you cannot link the reference. Supply the names of the interfaces and click OK. You cannot compile the JNDI lookup code unless you have copies of the interfaces mounted in your development environment.

     FIGURE 2-3 shows the Add EJB Reference dialog box setting up an unlinked reference. The Home Interface and Remote Interface are specified but the Referenced EJB Name field is empty. The reference will be linked later on the application property sheet.

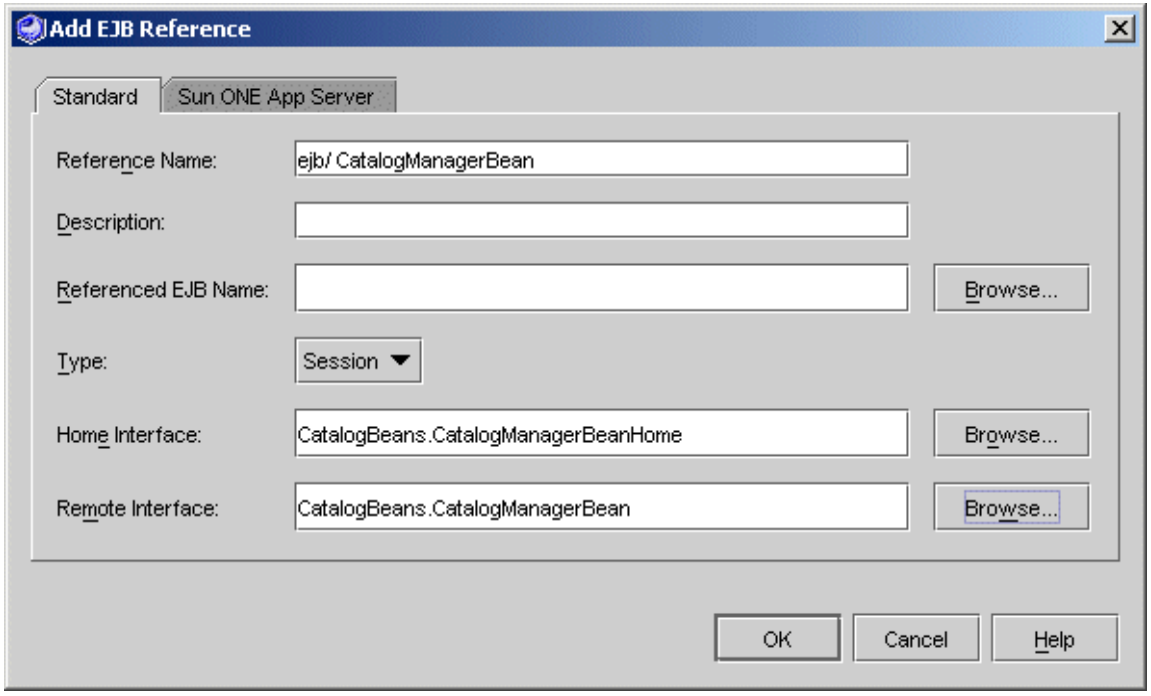

FIGURE 2-3 EJB Reference Property Editor With Unlinked Reference

- If the enterprise bean is available in your development environment, you can link the reference now. Click the Browse button next to the Referenced EJB Name field. Use the dialog box that appears to select the called enterprise bean. Click OK.

  FIGURE 2-4 shows the reference named `ejb/CatalogManagerBean` linked to `CatalogManagerBean`.

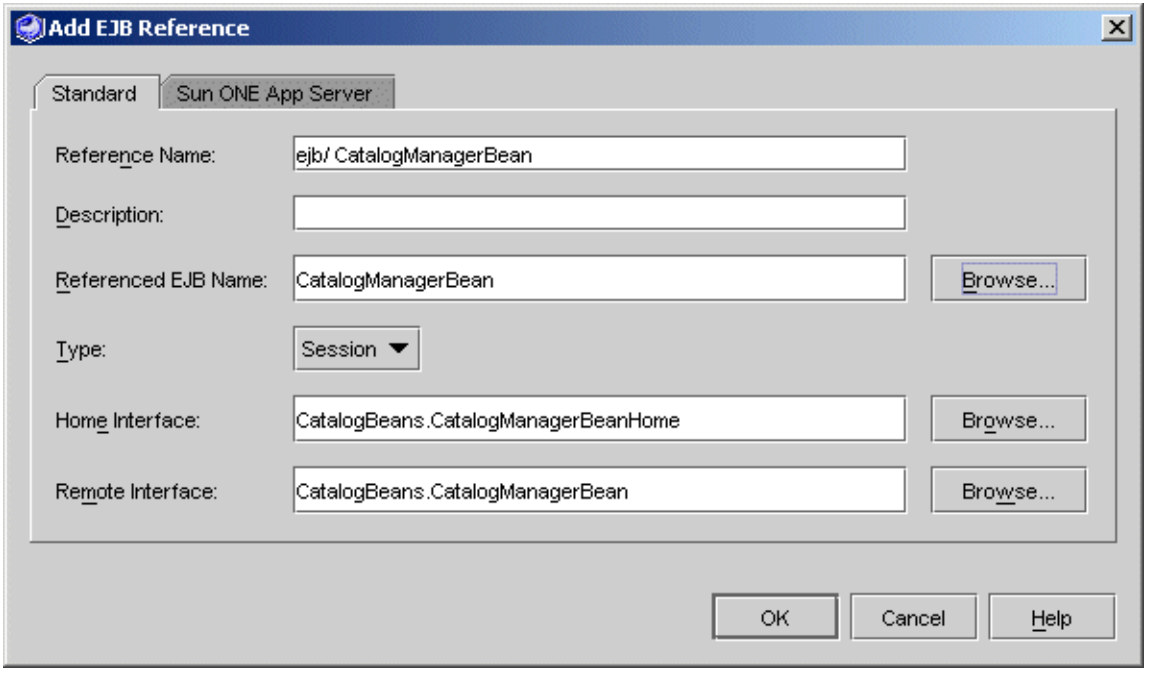

FIGURE 2-4 EJB Reference Property Editor With Linked Reference

Even if the called enterprise bean is available, you might choose not to link the reference now. There are a number of reasons to wait. For example, if there is any chance that your web component will be used in more than one application, you do not want to link the reference on the web module's property sheet. A developer who reuses the web module can relink the reference to some other enterprise bean that implements the same interfaces. Relinking the reference changes the value in the `web.xml` deployment descriptor file, and affects every use of the source code.

In this scenario the reference is left unlinked, as shown in FIGURE 2-3. The reference is linked after the J2EE application is created, on the application node's property sheet. See "Linking the EJB Reference" on page 78.

Mapping URLs to the Servlets

The link you set up on the welcome page is formatted with the following HTML tag:

```
<a href="allItems">Display the Catalog</a>
```

When a shopper clicks this link, the application server appends the URL pattern in the HTML tag to a URL path and executes the resulting URL. For the link to work properly, the URL that is generated by the application server must map to the servlet that you want to execute. The next section explains how URLs are mapped to servlets.

Understanding Servlet Mappings

In a deployed web module, the URLs for servlets and other web resources in the web module are the result of appending names to a URL path. For modules that are deployed to the Sun ONE Application Server 7, the URL path has this general form:

`http://`*hostname*`:`*port*`/`*web-context*`/`*URL-pattern*

The elements in this path are determined as follows:

- The hostname is the name of the machine that the application server is running on, and the port is the port that is specified for that server instance's HTTP requests. The port number is assigned when the application server is installed.
- The web context is a string that you specify as a property of the web module *after* you add the module to a J2EE application. The web context qualifies all of the web resources in the module.
- The URL pattern is a string that identifies a specific servlet or JSP page. You set up the URL pattern on the web module property sheet. You can do this before you assemble the web module into a J2EE application.

In other words, the URL patterns that you assign in your web module are relative to a web context that you will assign later when you add the module to a J2EE application. The URLs in this scenario use a format that makes them relative to the web context. Whatever web context you supply when you assemble the application, these links will work properly when the application executes. For information on setting the web context, see "Setting the Web Context for the Web Module" on page 77.

Examining Default Servlet Mappings

When you create a servlet, by default, the Servlet wizard uses the class name that you supply on the first page of the Servlet wizard for the servlet name and maps the servlet name to a URL pattern that includes the servlet name. FIGURE 2-5 shows the Servlet Mappings property editor for the web module with the default settings for the `AllItemsServlet`.

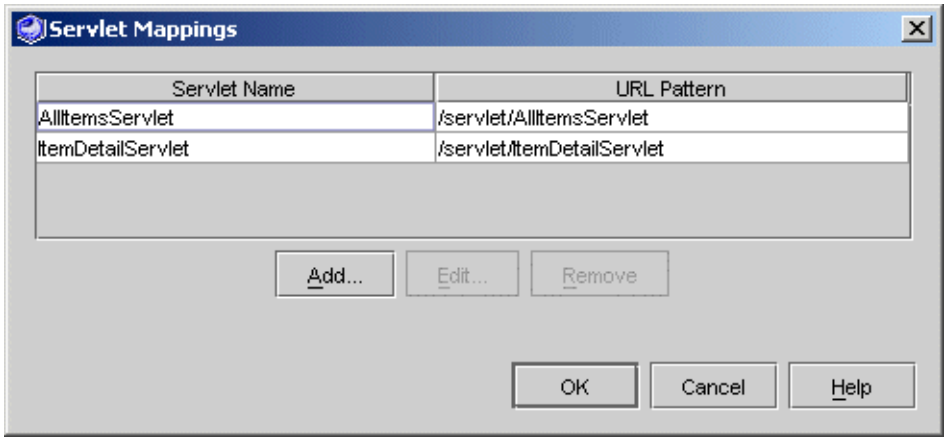

FIGURE 2-5 Servlet Mappings Property Editor

If you deploy the web module with this servlet mapping, `AllItemsServlet` is mapped to the following URL:

`http://`*hostname*`:`*port*`/`*web-context*`/servlet/AllItemsServlet`

Editing Servlet Mappings

If you want to map a different URL to the servlet, use the servlet mappings property editor to edit the mapping. In this scenario you change the URL pattern from the default value to a more meaningful value.

To edit the servlet mapping:

1. **Right-click the web module's web node and choose Properties → Servlet Mappings → ellipsis (...) button.**

   The Servlet Mappings property editor opens. The Servlet Mappings property editor lists all servlets in the module and any mappings that are set up for them.

2. **Select the current mapping for the `allItemsServlet` and click the Edit button.**

   The Edit Servlet Mapping dialog box opens.

3. **In the URL Pattern field, type** `/allItems` **and Click OK.**

FIGURE 2-6 shows the Servlet Mappings property editor with new mappings for the `AllItemsServlet` and the `DetailServlet`.

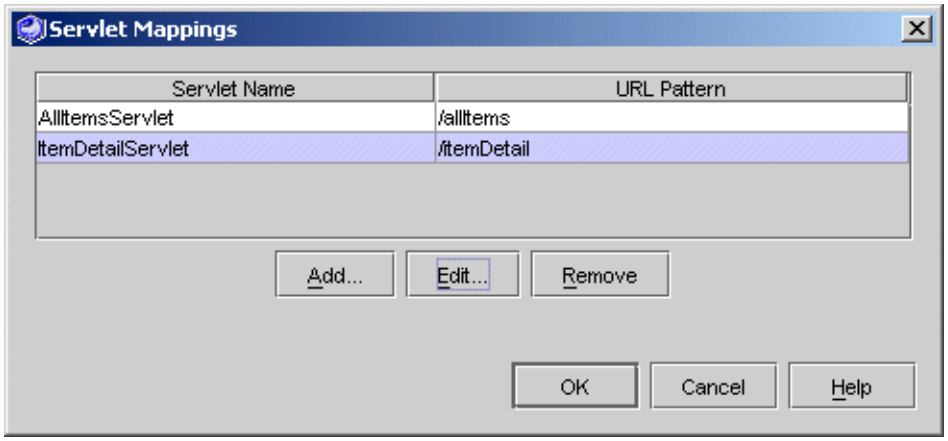

FIGURE 2-6 Servlet Mappings Property Editor

After you edit the servlet mappings, the `AllItemsServlet` can be executed with the following URL:

`http://`*hostname*`:`*port*`/`*web-context*`/allItems`

Notice that the new URL pattern is the string you used in the HTML tag that creates the link on the welcome page (see CODE EXAMPLE 2-1). Clicking the link now executes the `AllItemsServlet`.

Other Assembly Tasks

The preceding sections cover the assembly tasks that are required to assemble the `CatalogWebModule`. This section covers web module assembly tasks that are not required by the `CatalogWebModule` scenario.

Your web modules may require you to perform some of these other assembly tasks. This section covers several web module assembly tasks that you are likely to perform.

Setting Up Error Pages

If you want to specify error pages for a web module, you need to identify the error pages in the module's deployment descriptor. You do this in the Error Pages property editor.

To set up error pages for your web module:

1. **Right-click the web node and choose Properties → Deployment Tab → Error Pages → ellipsis (…) button.**

   The Error Pages property editor opens.

2. **Identify an error by HTTP error code or Java exception class and map it to a specific error page.**

   You can identify errors either by an HTTP error code or a Java exception class. Notice that the editor has two Add buttons, one for each category of error. For either type, you specify the error and map it to a page. FIGURE 2-7 shows the property editor after HTTP error code 404 has been mapped to a specific error page.

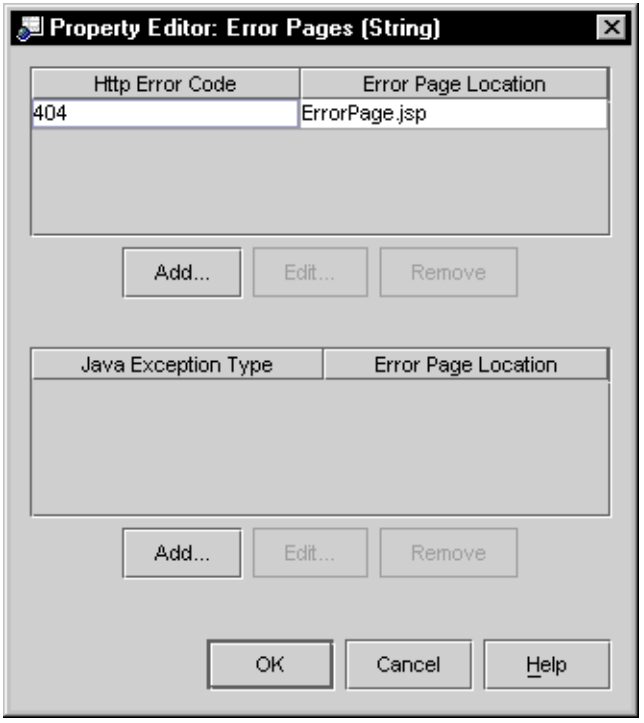

FIGURE 2-7 Error Pages Property Editor

Setting Up JSP Pages

If the web module you are assembling contains JSP page components, you have several ways to execute those components. If you create a new JSP page named `myJsp`, you can execute it in any of the following ways.

Executing JSP Pages With HTML Links

If you want to execute a JSP page from an HTML link, set up the link like the following example:

```
<a href=″myJsp.jsp″>Execute myJsp</a>
```

Executing JSP Pages Programmatically

If you create a JSP page with the IDE, no deployment descriptor entry is created for it. If your business logic accesses the JSP page programmatically, you do not need the deployment descriptor entry. For example, the following code is from a servlet that executes `myJsp`. Notice that the code identifies the JSP page to be executed by supplying its actual file name (`myJsp.jsp`).

CODE EXAMPLE 2-3 Code for Executing a JSP Page Programmatically

```
...
response.setContentType("text/html");
        RequestDispatcher dispatcher;
        dispatcher = getServletContext().getRequestDispatcher ("/myJsp.jsp");
        dispatcher.include(request, response);
...
```

Using URL to JSP Mappings

The preceding examples execute a JSP page by specifying its actual file name, `myJsp.jsp`. You can also map a URL pattern to a JSP page and then execute the page by referring to its URL pattern. This is a two step process. You first set up a servlet name for the JSP file.

To set up a URL mapping for a JSP page:

1. **Right-click the web node and choose Properties → Deployment tab → JSP Files → ellipsis (…) button.**

   The JSP Files property editor opens.

2. **Click the Add button.**

   The Add JSP File dialog box opens.

   a. **In the JSP File field, type the file name of the JSP file you are setting up.**

   b. **In the Servlet Name field, type the servlet name you are mapping to the JSP file.**

c. Click OK. The Add JSP File dialog box closes.

FIGURE 2-8 shows the JSP Files property editor after the servlet name `itemDetailPage` has been mapped to the file `myJsp.jsp`.

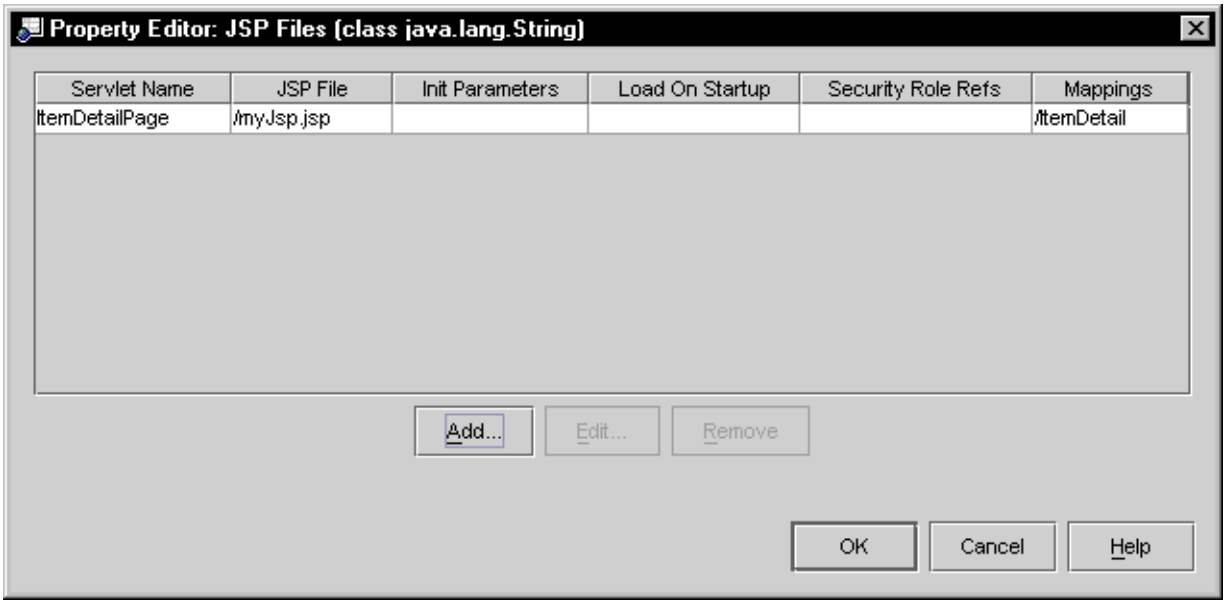

FIGURE 2-8 JSP Files Property Editor

3. **Click OK again to close the JSP Files property editor and return to the property sheet.**

4. **Still in the property sheet, right-click the Servlet Mappings property and click the ellipsis (…) button.**

   The Servlet Mappings property editor opens.

5. **Click the Add button.**

   The Add Servlet Mapping dialog box opens.

6. **In the Add Servlet Mapping dialog box, map a URL pattern to the new servlet name.**

   a. **In the Servlet Name field, type the servlet name you mapped to the JSP file.**

   b. **In the URL Pattern field, type the URL pattern you are mapping to the servlet name and, ultimately, to the JSP file.**

c. Click OK. The Add Servlet Mapping dialog box closes.

FIGURE 2-9 shows the Servlet Mappings Property Editor with the URL pattern `ItemDetail` mapped to the servlet name `ItemDetailPage`. The servlet name `ItemDetailPage` is already mapped to the JSP file `myJsp.jsp`.

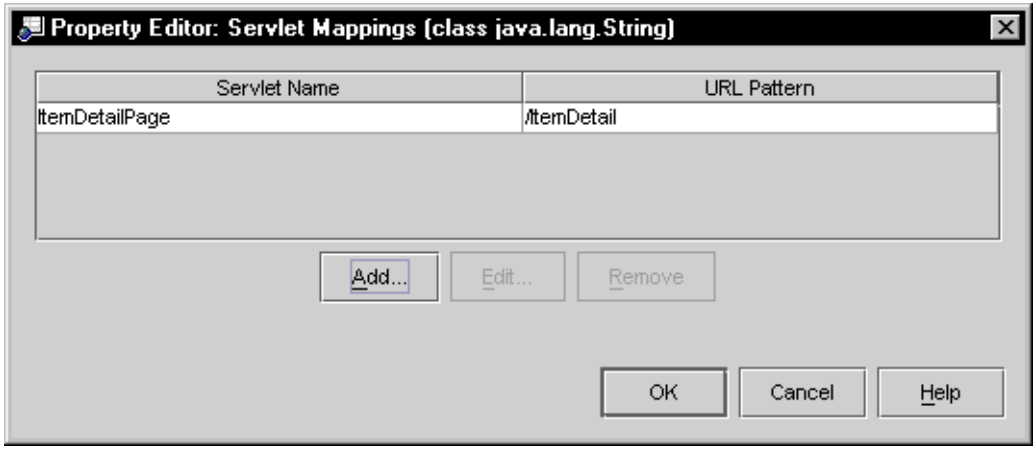

FIGURE 2-9 Servlet Mappings Property Editor

d. Click OK again to close the Servlet Mappings property editor.

After this mapping, the JSP page that is defined by the file `myJSP.jsp` can be executed with the following URL:

`http://`*hostname*`:`*port*`/`*web-context*`/ItemDetail`

Setting Up Environment Entry References

There are two parts to an environment entry:

- **A JNDI lookup.** The web component that uses the environment entry uses the JNDI naming facility to look up the entry's value.
- **An reference declaration in the web module's deployment descriptor.** The reference declaration declares the reference to the application server. The deployment descriptor also contains a value for the environment entry.

JNDI Lookup for Environment Entry References

A web component that uses the value of an environment entry needs lookup code like the code in CODE EXAMPLE 2-4.

CODE EXAMPLE 2-4 Lookup Statement for an Environment Entry

```
try {
       // Obtain Initial Context--Opens Communication With JNDI Naming:
       Context ic = new InitialContext();
       // Request Lookup of Environment Entry Named “Cache Size”:
       Integer cacheSize = (Integer)
        ic.lookup(“java:comp/env/NumberOfRecordsCached“);
}
catch(Exception e) {
       System.out.println(e.toString());
       e.printStackTrace();
       return;
}
```

The comments in the code explain what each line does.

Reference Declaration for Environment Entries

To set up a reference declaration for an environment entry:

1. **Right-click the web node and choose Properties → References tab → Environment Entries → ellipsis (...) button.**

   The Environment Entries property editor opens.

2. **Click the Add button.**

   The Add Environment Entry dialog box opens.

3. **Declare the environment entry reference.**

   a. **In the Name field, type the reference name that is used in the `lookup` statement.**

   b. **In the Type field, select the data type of the environment entry.**

   c. **In the Value field, type an initial value for the environment entry.**

      FIGURE 2-10 shows the Add Environment Entry dialog box with values that match the `lookup` statement in CODE EXAMPLE 2-4.

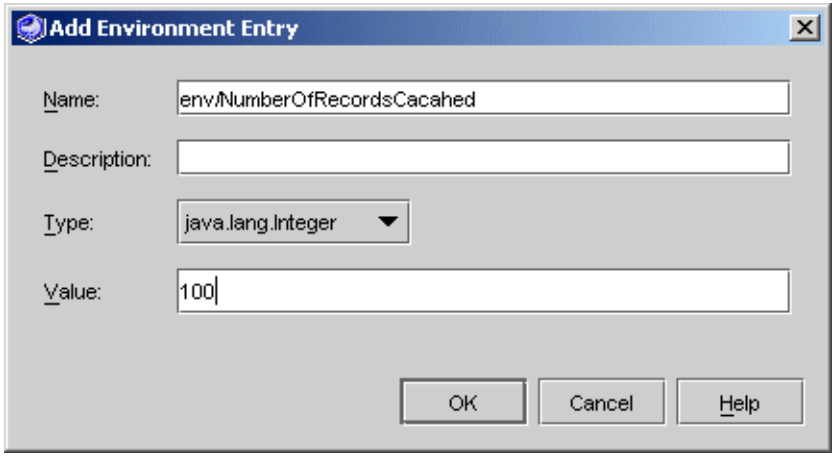

FIGURE 2-10 Add Environment Entry Dialog Box

4. **Click OK to close the dialog and process your environment entry definition.**

CHAPTER 3

Scenario: An EJB Module

FIGURE 3-1 shows an EJB module and the interactions in which it participates. This module is part of a J2EE application. and the interactions in the figure are typical for EJB modules in J2EE applications. The EJB module interacts with a client module (this interaction is represented in the figure by arrow labeled #1). The EJB module also interacts with an external resource, in this case a relational database management system (this interaction is represented by the arrows labeled #3). This EJB module, like most EJB modules, contains more than one enterprise bean, and the enterprise beans interact with each other (these interactions are represented by the arrows labeled #2).

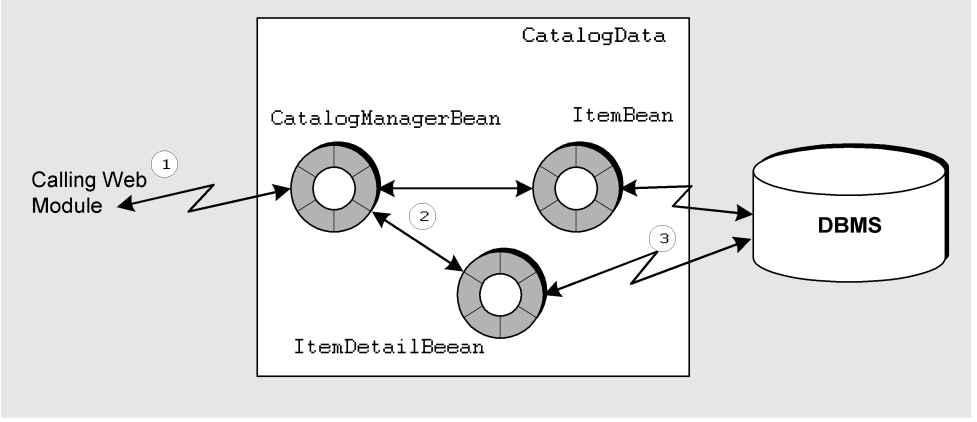

FIGURE 3-1 The `CatalogData` EJB Module

The Interactions in This Module

This scenario describes one possible use of the EJB module and the interactions that are illustrated in FIGURE 3-1. In this scenario, the EJB module is the back end for a J2EE application that supports a retail web site. The EJB module contains a set of enterprise beans that interact with the catalog data base. The web module that serves as the application's front end calls methods of this EJB module to get the data that it displays to users. Handling method calls and interacting with a data base are typical tasks for an EJB module in a J2EE application.

The web module in the application functions as a client of the `CatalogData` EJB module. Shoppers use web pages that are generated by the web module to request the data they want to view. The web module is responsible for calling the `CatalogData` module to obtain the data, and then formatting the data and displaying it to the shoppers.

The EJB module is responsible for getting the requested data from the database and passing it to the client module. The EJB module must be able to process requests for data and then return the correct data. The EJB module in this scenario contains only three enterprise beans and provides only two business methods. The business methods process only two types of requests. Although simple, these components and interactions show you how EJB modules provide the necessary interaction between client modules and data bases. The specific interactions that are covered in this scenario are outlined in the following list:

1. The client module requests some data by calling a method of the `CatalogData` EJB module. In this simple scenario, the client can request either a list of all the items in the catalog, or detailed information about one item.
2. The catalog data EJB module processes the request by generating a database query that will get the requested data.
3. The catalog data EJB module executes the query and returns the data to the client module. The client module then formats the data and displays it to the user.

The interactions between the client module and the `CatalogData` EJB module determine the J2EE technologies you choose to implement them and the internal architecture of the `CatalogData` EJB module. You should consider the points listed below:

- The interaction between client modules and the `CatalogData` module is synchronous. Online shoppers who ask to see the catalog wait for the application to display it. To provide a synchronous interaction, you implement this interaction with Java RMI. This means that client modules will request catalog data by calling methods of the `CatalogData` EJB module.

- The internal design of the EJB module is determined by the session-oriented nature of the interactions between the client modules and the `CatalogData` EJB module. An online shopper can look at a display of the whole catalog and then request detail for a single item. With multiple shoppers looking at the catalog simultaneously, the EJB module must match requests to user sessions and return the correct data to each session. To fulfill this need, the module has a single stateful session enterprise bean. The session bean manages all requests that are submitted by client modules. It handles the requests by calling methods of the entity beans, which generate database queries.

This is a common architecture for an EJB module. (For more information about modelling sessions with session beans, see *Building Enterprise JavaBeans Components*.)

For generating the database queries, the J2EE platform provides a type of enterprise bean known as the entity enterprise bean. Entity beans model database tables, and they have methods that execute queries. In this scenario the catalog data is stored in two tables, and the catalog data module has two entity beans. Each entity bean models one of the tables. The entity beans handle connections, query language, and other aspects of the `CatalogData` EJB module's interaction with the database.

Programming This Module

TABLE 3-1 summarizes the programming that is required to create the `CatalogData` EJB module that is described in the preceding section and illustrated in FIGURE 3-1.

TABLE 3-1 Programming Required for This Scenario

Application Element	Programming Required
Application server	None.
EJB module	Create a session enterprise bean (`CatalogManagerBean`) with remote interfaces. The remote interfaces are appropriate for methods that are called remotely by other modules. Add business methods to the session bean that return catalog data to the caller. One of these business methods returns all items in the catalog, the other returns detail for any item specified by the client. Create two entity enterprise beans (`ItemBean` and `ItemDetailBean`) to represent the two database tables that contain the catalog data. Create local interfaces for these entity enterprise beans. Local interfaces are appropriate for methods that are called by other enterprise beans in the same module. Add a method to the `ItemBean` that returns all items in the catalog. The `ItemDetailBean` has an automatically generated `findByPrimayKey` method that returns detail on a specific item. The session bean calls these methods to obtain the catalog data. Create two detail classes, one for Item data, one for ItemDetail data. The `CatalogData` EJB module returns instances of these classes to the caller. Create the `CatalogData` EJB module, which is represented in the Explorer by an EJB module node. Add the three enterprise beans to the module. Use the `CatalogData` EJB module's property sheet to specify the datasource for the catalog data.
J2EE application	To see how you add an EJB module to a J2EE application see Chapter 4.

The sections that follow show you how to perform many of these programming tasks. The method signatures in the interfaces specify the inputs and outputs of each interaction. The procedure sections show you how to connect these inputs and outputs to other components, other modules, and the catalog database. Instructions for creating the enterprise beans, adding business methods, and implementing the business methods with Java code are not included. To learn about these tasks, see the online help or *Building Enterprise JavaBeans Components.*

Creating Remote Interfaces for the Session Enterprise Bean

The design for the `CatalogData` EJB module calls for client modules to get catalog data by making remote calls to methods of the stateful session bean. To make the remote calls possible, the session bean must have remote interfaces. You generate the remote interfaces when you create the session bean with the Session Bean wizard. CODE EXAMPLE 3-1 shows the completed home and remote interfaces for the stateful session bean.

CODE EXAMPLE 3-1 Home and Remote Interfaces for the Session Bean

```
public interface CatalogManagerBeanHome extends javax.ejb.EJBHome {
  public CatalogBeans.CatalogManagerBean create()
    throws javax.ejb.CreateException, java.rmi.RemoteException;
}


public interface CatalogManagerBean extends javax.ejb.EJBObject {
  public java.util.Vector getAllItems() throws java.rmi.RemoteException;
  public CatalogBeans.idDetail getOneItemDetail(java.lang.String sku)
    throws java.rmi.RemoteException;
}
```

These interfaces define two interactions between the `CatalogData` EJB module and client modules. Clients can get a list of all the items that are in the catalog by calling the `getAllItems` method. Clients can also specify an item and get detailed information for it by calling the `getOneItemDetail` method.

Most real-world shopping applications provide more functionality than this. These real-world applications have more than two business methods in their interfaces.

When an EJB module returns references to entity bean instances, the application server tracks any changes that the client module makes to the entity bean instances and automatically generates database updates. Passing remote references to clients consumes network resources and should only be done when the client can update the data. In this scenario, the client only displays the data, and there is no need to use this feature. The `CatalogData` module returns instances of ordinary Java classes. These classes, known as detail classes, have the same fields as the entity beans. The `CatalogData` module copies data from entity bean instances to detail class instances and returns the detail class instances to the client for display. For more information on using detail classes with entity enterprise beans, see *Building Enterprise JavaBeans Components*.

For more information about client modules using remote interfaces to call the `CatalogData` module's methods, see "Programming the Servlet Methods" on page 39.

The `getAllItems` and `getOneItemDetail` methods are implemented with calls to methods of the entity beans. The entity beans generate the appropriate queries and return the requested data as entity bean instances. To see the implementation of the getAllItems method, see CODE EXAMPLE 3-3.

Creating Local Interfaces for the Entity Enterprise Beans

To obtain the data that clients request, the `CatalogManagerBean` calls methods of the two entity beans. Since these calls are within the module there is no need for the entity beans to have the resource-consuming remote interfaces. Instead, you can generate local interfaces for the entity beans. Local interfaces are faster and more efficient than remote interfaces. You should use local interfaces whenever a remote interaction is not required.

Generate the local interfaces when you create the entity beans with the Entity Bean wizard. CODE EXAMPLE 3-2 shows the completed local interfaces for the `Item` entity bean. The interfaces for the `ItemDetail` bean are similar.

CODE EXAMPLE 3-2 Local Home and Local Interfaces for the `ItemBean` Enterprise Bean

```
public interface LocalItemBeanHome extends javax.ejb.EJBLocalHome {
  public CatalogBeans.LocalItemBean findByPrimaryKey(java.lang.Integer aKey)
    throws javax.ejb.FinderException;
  public java.util.Collection findAll() throws javax.ejb.FinderException;
}


public interface LocalItemBean extends javax.ejb.EJBLocalObject {
  public CatalogBeans.iDetail getIDetail();
}
```

The `CatalogManagerBean` calls the `findAll` method to obtain a list of all the items in the catalog.

Note – If you plan to test individual enterprise beans with the IDE's test application feature, you need to generate both remote and local interfaces. The test application feature will generate a web module client that exercises the enterprise bean's methods, and the web module client requires the remote interfaces.

Using the Local Interfaces in the Session Enterprise Bean

Client modules request catalog data by calling one of the `CatalogManagerBean` business methods. These are the methods declared in the `CatalogManagerBean` interface, which is shown in CODE EXAMPLE 3-1. The implementations of these business methods call entity bean methods. To make these calls to the entity beans, the `CatalogManagerBean` needs local EJB references to the local interfaces you created for the entity bean. You must program two separate parts of these local EJB references.

- **JNDI lookup code**. Both of the session bean business methods include code that uses the JNDI naming facility to obtain a reference to an entity bean's LocalHome.
- **A declaration of the reference**. This is used by the runtime environment to identify the specific bean referred to by the lookup code.

JNDI Lookup Code for Local EJB References

CODE EXAMPLE 3-3 shows the implementation of the session bean's `getAllItems` method. Clients of the `CatalogData` EJB module call this method to get a list of all the items in the online catalog. The method implementation shows you how the `CatalogManagerBean` gets the catalog data by interacting with the entity beans. There are three steps in this code:

1. `CatalogManagerBean` uses JNDI lookup to obtain a local reference to the `ItemBean` local home interface.
2. `CatalogManagerBean` calls the `ItemBean.findAll` method.
3. `CatalogManagerBean` copies the catalog data from entity bean instances to detail class instances, adds the detail class instances to a vector, and returns the vector to the client.

Comments in CODE EXAMPLE 3-3 identify these steps

CODE EXAMPLE 3-3 The `getAllItems` Method of the `CatalogManagerBean`

```
public java.util.Vector getAllItems() {
  java.util.Vector itemsVector = new java.util.Vector();
  try{
    if (this.itemHome == null) {
      try {
         // Use JNDI lookup to obtain reference to Entity
         // Bean LocalHome.
        InitialContext iC = new InitialContext();
        Object objref = iC.lookup("java:comp/env/ejb/ItemBean");
```

CODE EXAMPLE 3-3 The `getAllItems` Method of the `CatalogManagerBean` *(Continued)*

```
          itemHome = (LocalItemBeanHome) objref;
        }
        catch (Exception e){
          System.out.println("lookup problem" + e);
        }
      }
      // Use the local reference to call findAll();
      java.util.Collection itemsColl = itemHome.findAll();
      if (itemsColl == null) {
        itemsVector = null;
      }
      else {
        // Copy data to detail class instances.
        java.util.Iterator iter = itemsColl.iterator();
        while (iter.hasNext()) {
          iDetail detail;
          detail=((CatalogBeans.LocalItemBean)iter.next()).getIDetail();
          itemsVector.addElement(detail);
        }
      }
    }
    catch (Exception e) {
      System.out.println(e);
    }
    return itemsVector;
}
```

The string specified in the `lookup` statement is the name of the reference and not the name of the referenced enterprise bean. The enterprise bean's name is often used as the reference name, to make it easier to remember which enterprise bean is meant. The actual mapping of reference name to enterprise bean is done with the reference declaration, which is covered in the next step.

Reference Declarations for Local EJB References

After you write the JNDI lookup code, you need to set up a local EJB reference declaration. The local EJB reference declaration maps the reference name used in the `lookup` statement to an actual enterprise bean name. The referenced enterprise bean must be in the same module as the calling bean.

To set up a local EJB reference declaration:

1. **Right click the calling bean's logical node and choose Properties → References tab → EJB Local References property → the ellipsis (...) button → the Add button.**

   The Add EJB Local Reference dialog box opens.

2. **Define the reference in the dialog box.**

   a. **In the Reference Name field, type the reference name used in the `lookup` statement.**

      FIGURE 3-2 shows the reference name that was used in CODE EXAMPLE 3-3.

   b. **Map the reference name to an enterprise bean.**

      Click the Browse button next to the Reference EJB Name field. This opens a browser that lets you select the matching enterprise bean. FIGURE 3-2 shows the field with `ItemBean` selected.

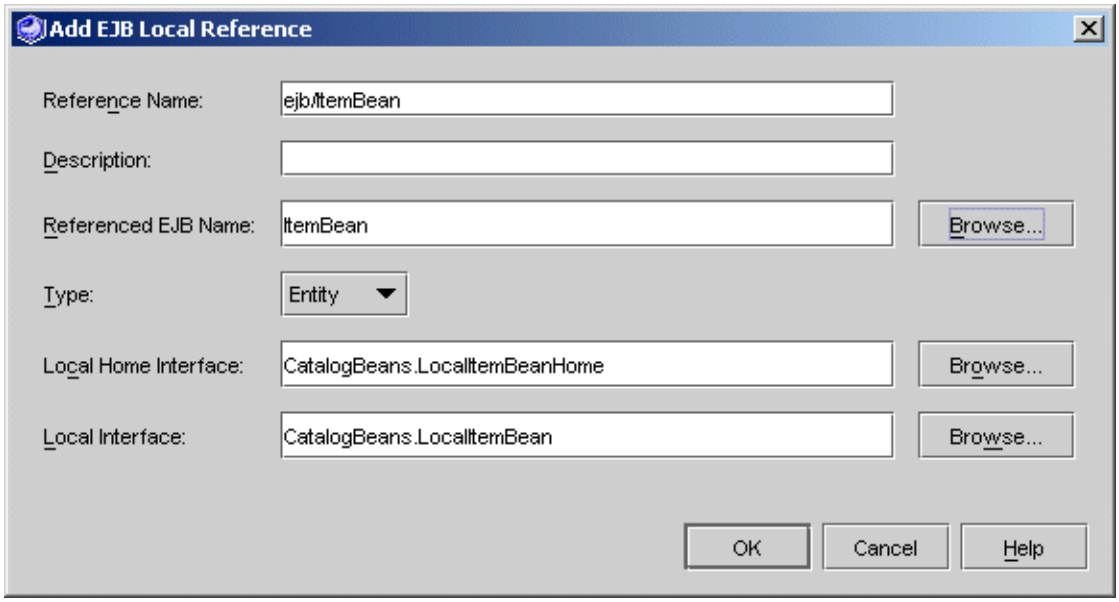

FIGURE 3-2 Add EJB Reference Dialog Box

Assembling the EJB Module

After you create the enterprise beans you create the `CatalogData` EJB module and add the enterprise beans. You also configure your EJB module by selecting property values that request specific runtime services from the application server.

Different application server products need different information for deployment, and the EJB module properties include some server-specific properties. This scenario uses some of the server-specific properties for the Sun ONE Application Server.

Creating the EJB Module

There are two ways to create an EJB module in the IDE. Both procedures create an EJB module node at the location you specify.

Your EJB module node represents the deployment descriptor for your module. Enterprise beans you add to the module appear as subnodes of the module node, but the IDE does not copy the source files for the enterprise beans to the directory that holds the module node. Keep this in mind when you decide where to put your EJB module.

- If you keep all of the source code for your module in a single filesystem, consider putting your EJB module node at the top level of that filesystem.
- If the source code for the module is in different filesystems, such as filesystems that are owned by different developers, consider creating a set of directories that contain only modules and J2EE applications. Keep these directories separate from the directories that contain the source code.

To create an EJB module beginning with an enterprise bean node:

1. **Right-click the enterprise bean node and choose Create New EJB Module.**

   The New EJB Module dialog box opens.

2. **Name the module and select a location in the filesystem for it.**

   a. **In the Name field, type the new module's name. In this scenario, the EJB module is named** `CatalogData`**.**

   b. **In the Select a package location for the new EJB module field, choose the filesystem, directory, or package where you want to create the new module.**

   c. **Click OK.**

      A node representing your new module is created under the filesystem, directory, or node you selected. The enterprise bean you right-clicked in Step 1 is automatically included in the new module.

You can add more enterprise beans to the module. See "Adding Enterprise Beans and Other Resources to the Module" on page 65.

To create an EJB module from a filesystem, package, or directory node:

1. **Right-click any Explorer window node and then choose New → All Templates.**

   The New wizard's Choose Template page opens.

2. **Choose the EJB Module template.**

   a. **In the Select a Template field, expand the J2EE node and select the EJB Module node.**

b. Click Next

The New Object Name page opens.

3. **Define the new EJB module.**

a. In the Name field, type the module name.

In this scenario the module is named `CatalogData`.

b. Click Finish.

The IDE creates the module and represents it in the Explorer with an EJB module under the filesystem, package, or directory you selected in step 1.

Both of these procedures create an EJB module node in the location you chose. Keep in mind that the IDE stores deployment descriptor information about the module in this directory, but the source code for the components in the module is not copied into this directory.

Adding Enterprise Beans and Other Resources to the Module

Once you have created a module, you can add enterprise beans to it.

To add an enterprise bean to an EJB module:

1. **Right-click the module node and choose Add EJB.**

This opens the Add EJB to EJB Module dialog box.

2. **Select one or more enterprise beans.**

a. In the Select the EJB(s) to add to this EJB Module field, select the enterprise beans you are adding to the module.

a. Click OK.

The IDE adds the selected enterprise beans to the module. In the Explorer, the IDE adds nodes representing the selected beans under the module node.

When you add an enterprise bean to a module, the IDE manages any dependencies of the enterprise bean on other kinds of resources (Java classes, image files, and so on). It automatically adds these resources to the module definition.

For the few exceptions to this, see “Identifying Extra Files” on page 71.

Specifying a Datasource for the Entity Enterprise Beans

In this scenario, the `CatalogData` EJB module accesses a database to get the catalog data. J2EE applications access databases through J2EE application servers, using connection pooling and other services provided by the application server. The

database must be set up as an application server resource with a JNDI name. Defining a database as an application server resource maps the databases JNDI name to the actual database URL.

A J2EE application that accesses a database must be configured with a resource reference that contains the database JNDI name. The way you set up the resource reference depends on the type of entity bean you are working with:

- If you are working with a container-managed entity bean, the application server determines how you identify the database. With most application servers, including the Sun ONE Application server, you use one of the module's server-specific properties to identify the datasource by JNDI name.
- If you are working with a bean-managed entity bean, you must write JNDI lookup code and set up a matching resource reference declaration that maps the lookup name to the JNDI name.

Note – At development time, you can open a live connection to a database in the IDE and use the Entity Bean wizard to create an entity bean that models a database table. This kind of connection is covered in *Building Enterprise JavaBeans Components*. This section covers specifying the database that will be used at runtime.

Using the Server-specific Tabs for Container-managed Entity Beans

For container-managed entity beans, you use the EJB module's CMP resource property to specify the datasource by JNDI name. The IDE creates the resource reference for you automatically. The CMP Resource property contains the JNDI name for the default Pointbase database.

To configure container-managed entity beans to use a database:

1. **Right-click the EJB Module node and choose Properties → Sun ONE AS tab → CMP Resource → the ellipsis (...) button.**

   The CMP Resource property editor opens.

2. **Type in the values that identify the database for the EJB module and its entity beans.**

   FIGURE 3-3 shows the CMP Resource property editor with values that give the `CatalogData` EJB module access to the Pointbase database named `sample`.

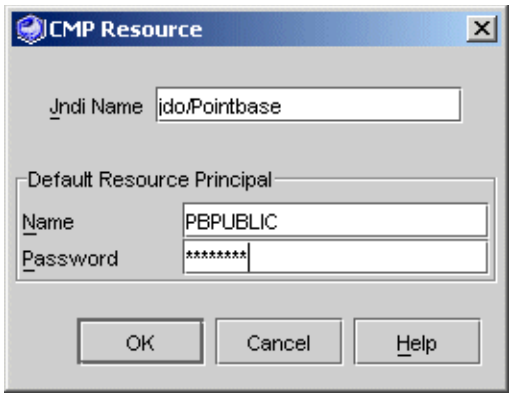

FIGURE 3-3 EJB Module CMP Resource Property Editor

Note – Most installations of the IDE include the Sun ONE Application Server and the PointBase database server. Installing the IDE sets up the PointBase database named `sample` as an application server resource with a JNDI name of `jdo/PointbasePM`. In this scenario, the `CatalogData` EJB module is configured to use the `sample` databse.

This procedure is for using the IDE with the Sun ONE Application Server and the installed PointBase database. If you are using another database product or another application server product, you must adapt the procedure.

- If you are using the IDE with the Sun ONE Application Server and a database product other than PointBase, you need to use the application server's administration tools to define your database to the application server as a JDBC datasource with a JNDI name. After you do this, you can use the procedure to configure your EJB module with the datasource JNDI name.
- If you are using an application server other than the Sun ONE Application Server, the database must be set up as a datasource resource of the application server and given a JNDI name. The procedures for this depend on the application server product. After the database is set up with a JNDI name, you can configure your EJB module with the JNDI name. In the EJB Module property sheet, use the server-specific tab for the application server product you are using. The property name may be different from CMP Resource.

If you are going to deploy your application in a managed test environment or a production environment, system administration will probably be responsible for setting up databases as application server resources and assigning JNDI names. In this case you simply need to obtain the datasource's JNDI name from system administration.

Creating Resource Factory References Explicitly for Bean-managed Entity Beans

If your EJB module contains entity beans that use bean-managed persistence instead of container-managed persistence, your entity beans already include JDBC and JTA code that prepares and executes database queries. This is covered in *Building Enterprise JavaBeans Components*. This code must include a JNDI lookup that obtains a resource factory reference to the datasource. The lookup uses the datasource's JNDI name to obtain the reference. Like other types of J2EE references, this type of reference has two parts:

- **A JNDI lookup**. Each entity bean includes code that uses the JNDI lookup statement to obtain a reference to a named resource.
- **A declaration of the reference**. The declaration maps the reference name used in the lookup statement to the datasource's JNDI name.

Any datasource you specify with an explicit lookup must be a named resource of the application server with an assigned JNDI name. The application server has the information to map the JNDI name to the actual database URL.

JNDI Lookup Code for Resource Factory References

CODE EXAMPLE 3-4 shows the code that you use in a BMP entity enterprise bean to look up a datasource:

CODE EXAMPLE 3-4 JNDI Lookup for a Database

```
try {
  // Obtain Initial Context--Opens Communication With JNDI Naming:
  Context ic = new InitialContext();
  // Request Lookup of Resource--In This Example a JDBC Datasource:
  javax.sql.DataSource hrDB = (javax.sql.DataSource)
                                  ic.lookup("java:comp/env/jdbc/Local_HR_DB");
}
catch(Exception e) {
  System.out.println(e.toString());
  e.printStackTrace();
  return;
}
```

Reference Declarations for Resource References

In addition to writing the lookup statement, you must set up a reference declaration for the datasource resource.

To set up a reference declaration for a datasource reference:

1. **Right-click the enterprise bean's logical node and choose Properties → References tab → Resource References → ellipsis (...) button → Add button.**

   The Add Resource Reference dialog box opens.

2. **Supply the information that identifies the datasource.**

   a. **In the Name field, type the reference name used in the `lookup` statement.**

   FIGURE 3-4 shows the reference name that was used in CODE EXAMPLE 3-4.

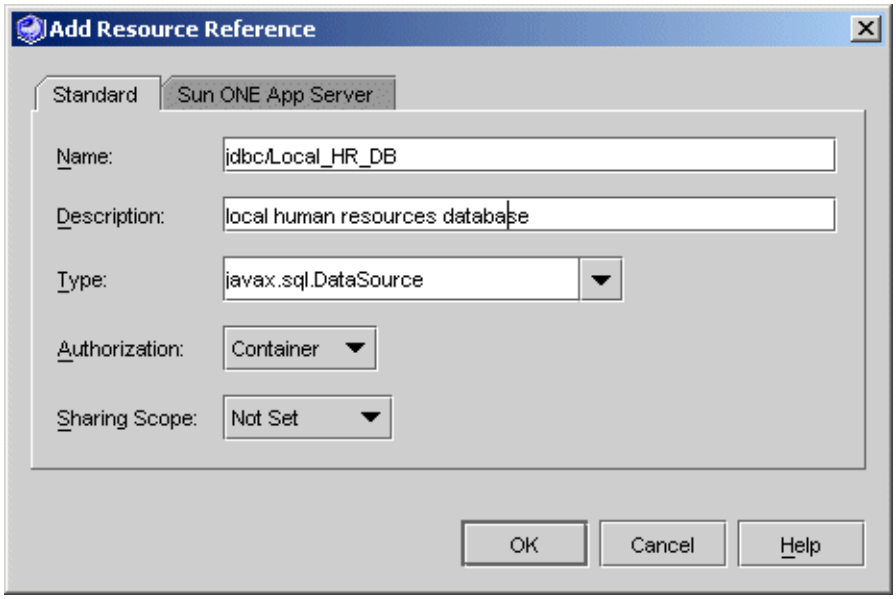

FIGURE 3-4 Add Resource Reference Dialog Box

   b. **Click the Sun ONE App Server tab.**

   c. **Map the reference name to a datasource.**

   FIGURE 3-5 shows the JNDI Name field with the value `jdbc/jdbc-pointbase`. This is the default name JNDI name for the PointBase database that is included in most installations of the IDE.

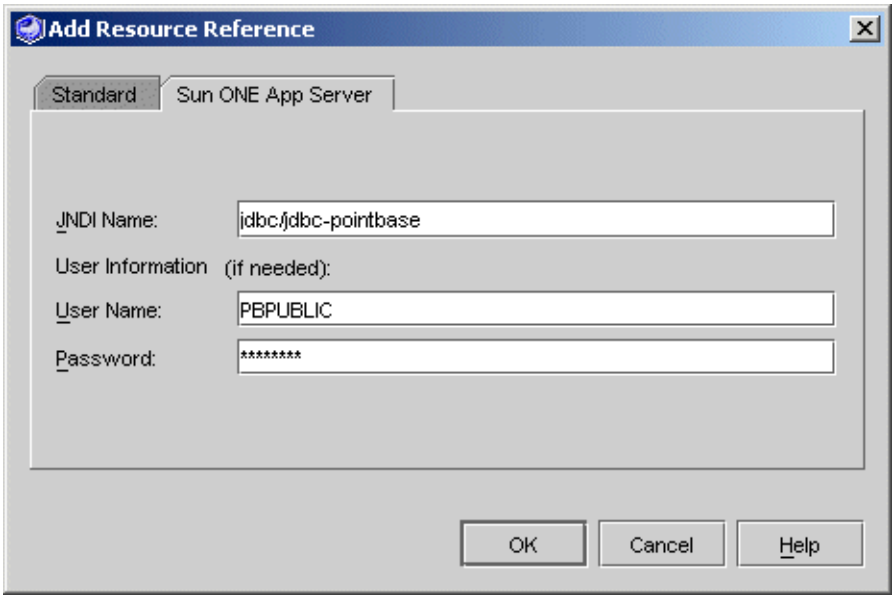

FIGURE 3-5 Add Resource Reference Dialog Box, Server-specific Tab

This procedure is for using the IDE with the Sun ONE Application Server and the installed PointBase database. If you are using another database product or another application server product, you must adapt the procedure. Adapt the procedure as follows:

- If you are using the IDE with the Sun ONE Application Server and a database product other than PointBase, you need to use the application server's administration tools to define your database to the application server as a JDBC datasource with a JNDI name. After you do this, you can use this procedure to configure your EJB module with the datasource JNDI name.
- If you are using an application server other than the Sun ONE Application Server, the database that you use must be set up with the application server as a datasource resource with a JNDI name. The procedures for this depend on the application server product. After the database is set up with a JNDI name, you can configure your EJB module with the JNDI name. In the Add Resource Reference dialog box, use the server-specific tab for the application server product you are using.

Other Module Assembly Tasks

The preceding sections cover the assembly tasks that are required to assemble the `CatalogData` EJB module. This section describes some EJB module assembly tasks that were not covered in the `CatalogData` scenario.

Your EJB modules may require you to perform some of these other assembly tasks. This section covers several web module assembly tasks that you are likely to perform.

For each module, you need to determine what assembly tasks are required. Questions that you might ask about a module include the following:

- Have all the references in the module been linked? Your module may contain some references to components in other modules. These references can only be linked after the module has been assembled into a J2EE application.
- Have generic security roles been set up for the module? Have any security role references in the module's enterprise beans been linked to these generic security roles? Have method permissions been mapped to these generic security roles? For more information on these issues see Chapter 7.
- Have container-managed transactions been defined? For more information on this issue see Chapter 6.

Some EJB module tasks that were not covered in the `CatalogData` EJB module scenario are described in the following sections.

Identifying Extra Files

In most cases, the IDE recognizes the dependencies of the enterprise beans that are in an EJB module and includes all of the needed files in the EJB JAR files that it generates at deployment time. There are some dependencies, however, that the IDE does not recognize. These unrecognized dependencies include:

- Enterprise beans in your module that use a help file without calling it directly
- Enterprise beans in your module that access a class dynamically and use the class name only as a string and not as a class declaration

In these and similar cases, the IDE does not recognize the enterprise bean's dependency and does not include the help file or class file in archive files that it creates. You need identify these files so that they are included in the archive and are available at run time.

To identify extra files:

1. **Right-click the EJB module node and choose Properties → Extra Files → ellipsis (…) button.**

   The Extra Files property editor opens.

2. **Select any extra file that should be deployed or archived as part of this module.**

   a. **In the Source field, select the extra files that belong in the module.**

   b. **Click the Add button to add the selected files to the list of extra files.**

3. **Click OK to close the editor and process you selections.**

Excluding Duplicate JAR Files

In some cases you want to prevent the IDE from acting on some of the file dependencies it recognizes. For example, an enterprise bean in your module has a dependency on a commonly used JAR file that you know will be present in the runtime environment. You can prevent the IDE from adding an unnecessary copy of the commonly used JAR file by identifying that JAR file as a duplicate JAR file.

To exclude a duplicate JAR file:

1. **Right-click the module node and choose Properties → Library Jars to Exclude → ellipsis (…) button.**

   This Library Jars to Exclude property editor opens. It displays a list of mounted JAR files.

2. **Select the JAR files that should be excluded and click the Add button to move them to the list of library JARs to exclude.**

3. **Click OK to close the property editor and process your selections.**

CHAPTER 4

Scenario: Web Module and EJB Module

FIGURE 4-1 shows a web module and an EJB module assembled into a J2EE application. Most of the interactions that appear in the figure are covered in Chapter 2 and Chapter 3. For example, the HTTP requests and responses between the user's web browser and the web module are covered in Chapter 2. The interaction between the modules, which is represented in the figure by the arrows labeled #1, is new.

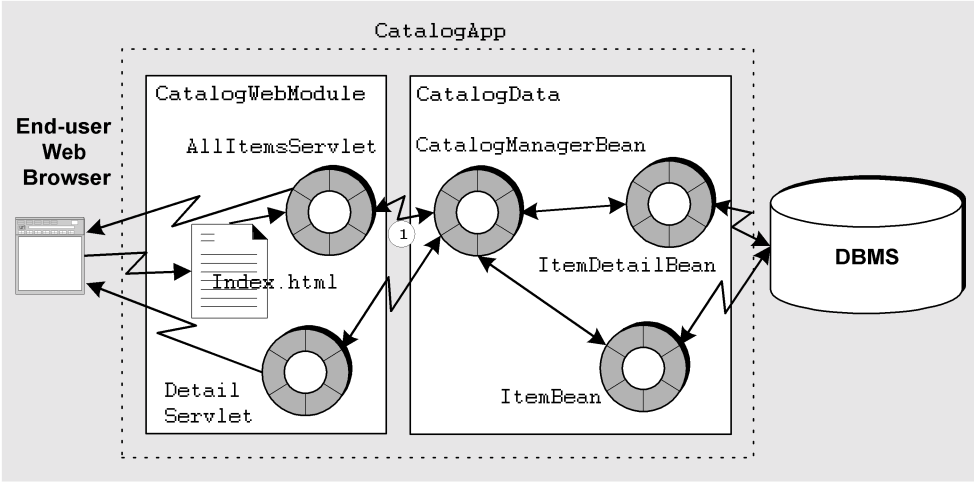

FIGURE 4-1 The `CatalogApp` J2EE Application

The Interactions in This Application

This scenario describes one possible use of the J2EE application and the interactions that are illustrated in FIGURE 4-1. This scenario continues the online shopping application that was described in Chapter 2 and Chapter 3. The programming required for the HTTP interactions with users is already complete in the web module. The programming required for the interactions with the database is already complete in the EJB module. This scenario explains how you combine the two modules into a J2EE application that performs end-to-end interactions.

There is one interaction between the modules that requires assembly work. This interaction is the remote method call from the web module to the EJB module. Most of the code needed for this interaction is already in the web module or the EJB module. The code already in the modules is summarized below:

- The web module contains JNDI lookup code and an EJB reference declaration. For more information about this code, see "The Reference Declaration for the EJB Reference" on page 42.
- The EJB module contains remote interfaces that support remote method calls. For more information about this code, see "Creating Remote Interfaces for the Session Enterprise Bean" on page 59.

The procedures in this chapter show you how to assemble the two modules into a J2EE application and configure the interaction between the modules. After the application is assembled you can deploy and execute the application.

Programming This Application

TABLE 4-1 summarizes the programming required to create and assemble the J2EE application that is described in this scenario.

TABLE 4-1 Programming Required for This Scenario

Application Element	Programming Required
Application server	None.

TABLE 4-1 Programming Required for This Scenario *(Continued)*

Application Element	Programming Required
Web module	See Chapter 2.
EJB module	See Chapter 3.
J2EE application	Create the `CatalogApp` J2EE application. This creates a J2EE application node in the Sun ONE Studio 5 Explorer. Add the web module and the EJB module to the application. Specify the web context for the web module. Make sure the web module's EJB references are correctly linked to enterprise beans in the EJB module.

The sections that follow demonstrate these programming tasks.

Creating the J2EE Application

There are two ways to create a J2EE application in the IDE. Both procedures create an application node in the location that you specify.

Your J2EE application node represents the deployment descriptor for your application. Modules you add to the application appear as subnodes of the application, but the IDE does not copy the source files for the modules to the directory that holds the application node. Keep this in mind when you decide where to put your J2EE application node.

- If you keep all of the source code for your application in a single filesystem, consider putting your application node at the top level of that filesystem.
- If the source code for the application is in different filesystems, such as filesystems that are owned by different developers, consider creating a set of directories that contain only J2EE applications and modules. Keep these directories separate from the filesystems that contain the source code.

To create a J2EE application from a module node:

1. **Right-click an EJB module node and choose New Application.**

   The New Application dialog opens.

2. **Name the J2EE application and select a location in the filesystem for it.**

   a. **In the name field, type the new application's name.**

      In this scenario, the new J2EE application is named `CatalogApp`.

   b. **In the Select a package location for the new Application field, choose the filesystem, directory, or package where you want to create the new application.**

c. Click OK.

A node representing the new application is created under the filesystem, directory, or node you selected. The module you right-clicked in Step 1 is automatically included in the new application.

You can add more modules to the application. For the procedures, see "Adding Modules to the J2EE Application" on page 76.

To create an EJB module from a filesystem, package, or directory node:

1. **Right-click any Explorer window node and choose New → All Templates.**

   The New wizard's Choose Template page opens.

2. **Select the J2EE Application template.**

   a. In the Select a Template field, expand the J2EE node and select the Application node.

   b. Click Next

   The New Object Name page opens.

3. **Define the new J2EE application.**

   a. In the Name field, type the new application's name.

   In this scenario the application is named `CatalogApp`.

   b. Click Finish.

   The IDE creates the application and represents it in the Explorer with an J2EE application node under the filesystem, package, or directory you selected in Step 1.

Both of these procedures create an application node in the location you chose. Keep in mind that the IDE stores deployment descriptor information about the application in this directory, but the source code for the components in the application's modules is not copied into this directory.

Adding Modules to the J2EE Application

Once you have created a J2EE application you can add modules to it. To add a module to an application:

1. **Right-click the application and choose Add Module.**

   The Add Module to Application dialog box opens.

2. **Select one or more modules.**

   a. In the Select the Module(s) to add to this Application field, select the modules you are adding to the application.

- To add a web module, select the module's WEB-INF node.
- To add an EJB module, select the module node.

b. Click OK.

The IDE adds the selected modules to the J2EE application. In the Explorer, the IDE adds nodes representing the selected modules under the J2EE application node.

When you add a module to a J2EE application, the IDE notes any dependencies of the module on other kinds of resources (Java classes, image files, and so on) and automatically includes those resources in the application.

Setting the Web Context for the Web Module

When you deploy a J2EE application to a J2EE application server, URLs are mapped to the web resources in the web module. URLs are mapped by appending URL patterns to a URL path. For the Sun ONE application server, the URL path has this general form:

`http://`*hostname*`:`*port*`/`*web-context*`/`*URL-pattern*

The elements in this path are determined as follows:

- The hostname is the name of the machine that the application server is running on, and the port is the port number that is specified for that server instance's HTTP requests. The port number is assigned when the application server is installed.
- The web context is a string that you specify in this procedure. The web context qualifies all of the web resources in the module.
- The URL pattern is a string that identifies a specific servlet or JSP page. You assign the URL pattern on the web module property sheet. You can do this before you assemble the web module into a J2EE application. For information on this procedure, see "Mapping URLs to the Servlets" on page 45

In other words, the URL patterns that you assigned in the web module property sheet are relative to the web context that you assign in this procedure.

To set the web context:

1. **Right-click the included web module node (this is the web module node under the J2EE application node) and choose Properties.**
2. **Click the Web Context property and type in the string you want to use.**

FIGURE 4-2 shows the property sheet for the `CatalogWebModule` described in this scenario. The web context property is set to `catalog`.

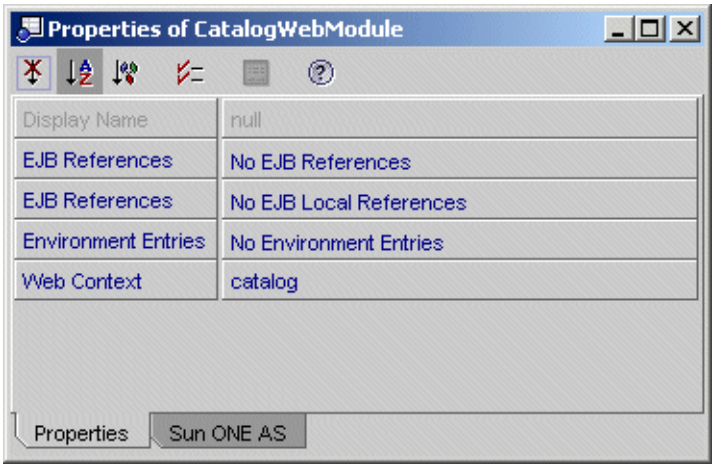

FIGURE 4-2 Property Sheet for `CatalogWebModule`

When the web context property is set to `catalog`, URLs for web resources in the `CatalogApp` J2EE application will have this general form:

`http://`*hostname*`:`*port*`/catalog/`*URL-pattern*

If you do not supply a web context, it defaults to blank. If you let the web context for `CatalogApp` J2EE application default to blank, the URLs for web resources in the application would have this general form:

`http://`*hostname*`:`*port*`/`*URL-pattern*

Linking the EJB Reference

The `CatalogWebModule` contains JNDI lookup code and an EJB reference declaration. The `CatalogWebModule` code and the deployment descriptor specify interfaces of type `CatalogManagerBeanHome` and `CatalogManagerBean`. For information on how the lookup code and the reference declaration are set up, see “The Reference Declaration for the EJB Reference” on page 42.

The `CatalogData` EJB module contains remote interfaces of type `CatalogManagerBeanHome` and `CatalogManagerBean`, and a bean class named `CatalogManagerBeanBean` that implements the interfaces. For information on creating the interfaces, see “Creating Remote Interfaces for the Session Enterprise Bean” on page 59.

Before executing the `CatalogApp` J2EE application you must link the EJB reference in the web module to the enterprise bean in the EJB module.

In some circumstances you link the reference on the web module's property sheet before creating the application. The reference in the `CatalogWebModule` was not linked in the web module property sheet. In this scenario, the reference is linked on the `CatalogApp` property sheet.

To link an EJB reference on the application node property sheet:

1. **Right-click the application node and choose Properties → EJB References → ellipsis (...) button.**

   The EJB References property editor opens.

2. **Check the status of the EJB references.**

   This editor shows you all of the references that have been declared in the application. The references are identified by module and reference name.

   FIGURE 4-3 shows the EJB References property editor for the `CatalogApp` J2EE application. There is one EJB reference, which is declared in the `CatalogWebModule`. The reference is named `ejb/CatalogManagerBean`, and it is not resolved.

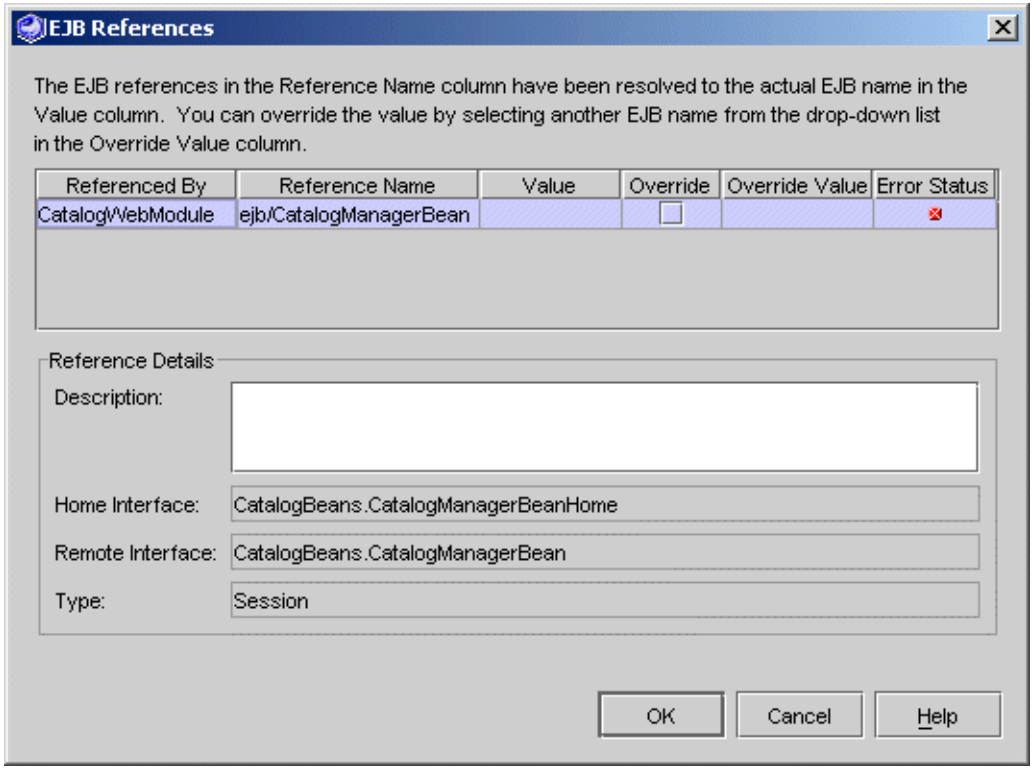

FIGURE 4-3 Unlinked EJB Reference

If a reference is not resolved, the Value field is empty and the Error Status field displays an icon that represents an error.

3. **Link any unresolved references.**

   To do this, click the Override Value field. The field displays a list of the enterprise beans in the application that implement the interfaces that are specified in the reference. Select one of these enterprise beans.

   FIGURE 4-4 shows the same EJB reference as FIGURE 4-3, but the reference has been linked with an override. When the application executes, the method calls coded in `CatalogWebModule` will call the enterprise bean specified in the Override Value field. In FIGURE 4-4, the Override Value field specifies an enterprise bean in the `CatalogData` EJB module, named `CatalogBeans.CatalogManagerBean`.

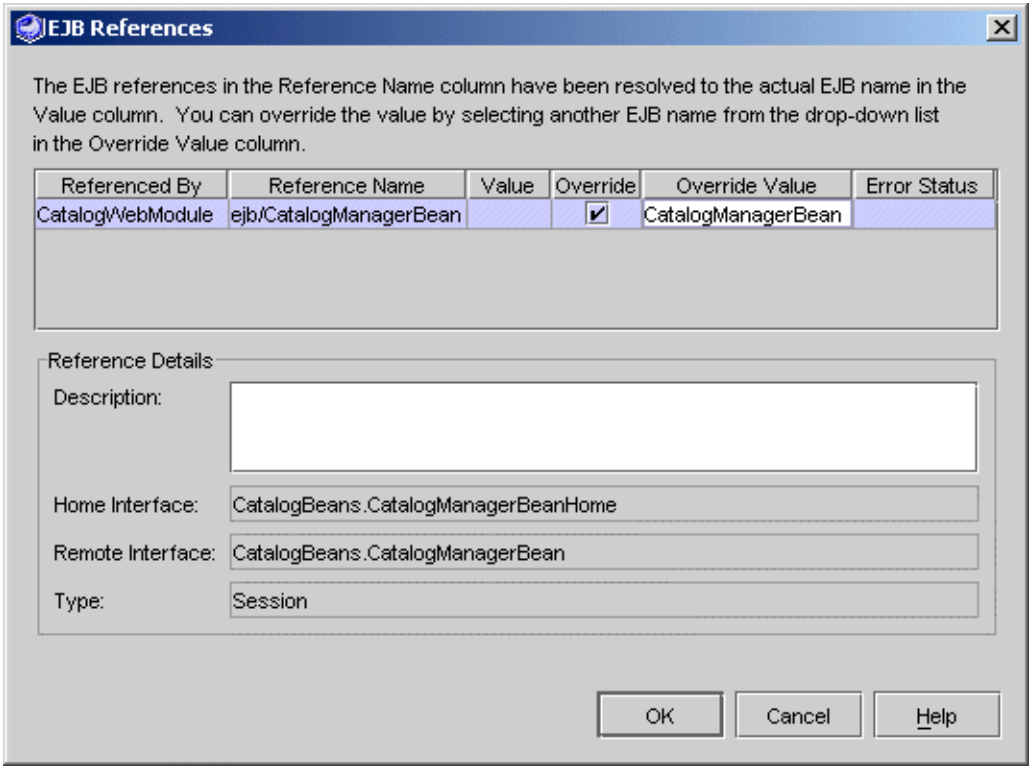

FIGURE 4-4 EJB Reference Linked by Override

When a reference is resolved, the Override Value field displays the name of the linked enterprise bean, and the Error Status field is empty.

Additional Assembly Tasks

The preceding sections cover the assembly tasks that are required to assemble the `CatalogApp` J2EE application. This section covers other J2EE application assembly tasks that are not required by the `CatalogApp` scenario.

Your J2EE applications may require you to some of these other assembly tasks. This section covers an application assembly task that you are likely to perform.

Overriding Environment Entries

If an application contains environment entries, you may need to override the values that were set for them on the module property sheets. You do this on the application's Environment Entries property editor.

To override an environment entry value:

1. **Right-click the application node and choose Properties → Environment Entries → ellipsis (...) button.**

   The Environment Entries property editor opens. This editor shows you all of the environment entries that have been declared in the application. The environment entries are identified by reference name and module.

2. **Study the environment entries in the application.**

   FIGURE 4-5 shows the Environment Entries property editor for the `CatalogApp` application. The property editor displays an environment entry that was declared in the web module. (To see how this environment entry was declared in the web module, see "Setting Up Environment Entry References" on page 51.)

   The Value field displays `100`, which is the initial value that was set on the web module's property sheet.

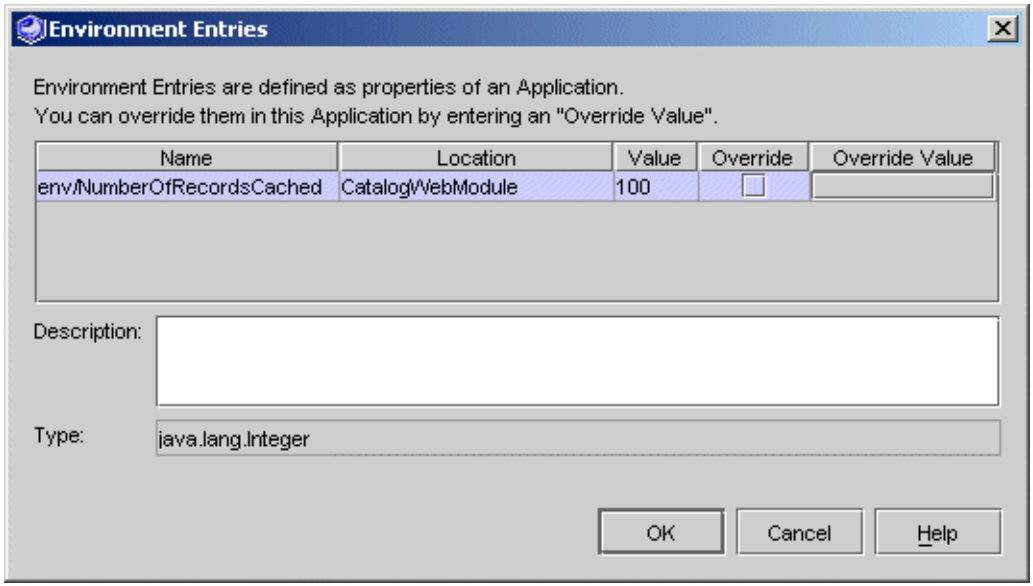

FIGURE 4-5 J2EE Application Environment Entries Property Editor

3. **When the initial values are not appropriate for the assembled application, override them.**

   a. **For the environment entry you are overriding, click the Override Value field and the ellipsis (...) button.**

      The Override Value dialog box opens.

   b. **In the Value field, type the override value. Click OK to close the dialog box and return to the property editor.**

      FIGURE 4-6 shows the Environment Entries property editor with a value in the override field.

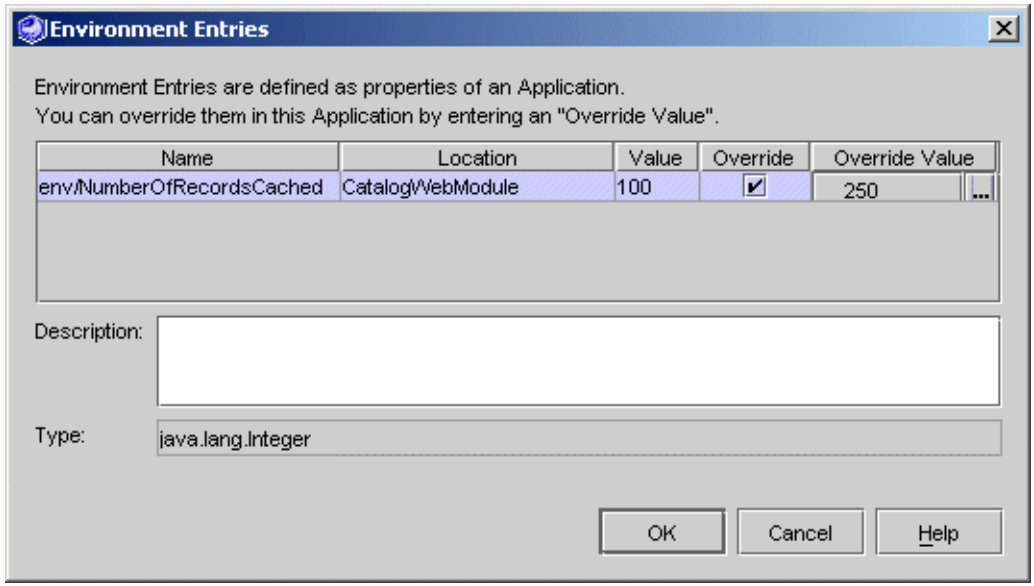

FIGURE 4-6 Overriding an Environment Entry Value

If you have the web module source files in your development environment you can change the environment entry's value on the web module property sheet. But, if there is any chance that the web module will be used in more than one application, it is better to override this value on the application property sheet. If another developer reuses the web module in another application and changes the value of the environment entry on the web module property sheet, it would change the value in your application when you redeployed.

Viewing and Editing Deployment Descriptors

In general, you should control the contents of deployment descriptors by working with module and application property sheets. By setting properties, you control the contents of the deployment descriptor. The IDE does allow you to view the actual XML deployment descriptors for modules and applicatons.

Viewing a Module Deployment Descriptor

You can view deployment descriptors for J2EE applications, included web modules and included EJB modules. To view a deployment descriptor:

- **Right-click a J2EE application node, included EJB module node (an EJB module node underneath a J2EE application node), or an included web module, and choose View Deployment Descriptor.**

  The IDE opens the deployment descriptor in the Source Editor, in read-only mode.

Editing Module Deployment Descriptors

To edit an EJB module deployment descriptor:

- **Right-click an EJB module node and choose Deployment Descriptor Final Edit.**

  The IDE opens the deployment descriptor in the Source Editor.

To edit a web module deployment descriptor:

- **Right-click the web module's web node and choose Edit.**

  The IDE opens the deployment descriptor in the Source Editor.

CHAPTER 5

Scenario: Web Module and Queue-mode Message-driven Bean

FIGURE 5-1 shows a web module and an EJB module assembled into a J2EE application. The interaction between the modules is asynchronous messaging. The web module sends a message to a queue and a message-driven enterprise bean in the EJB module reads the message from the queue. Sending the message to the queue is represented in the figure by the arrow labeled #1. Reading the message from the queue is represented by the arrow labeled #2. The message-driven bean reads the message and then initiates processing by calling methods of other enterprise beans in the module.

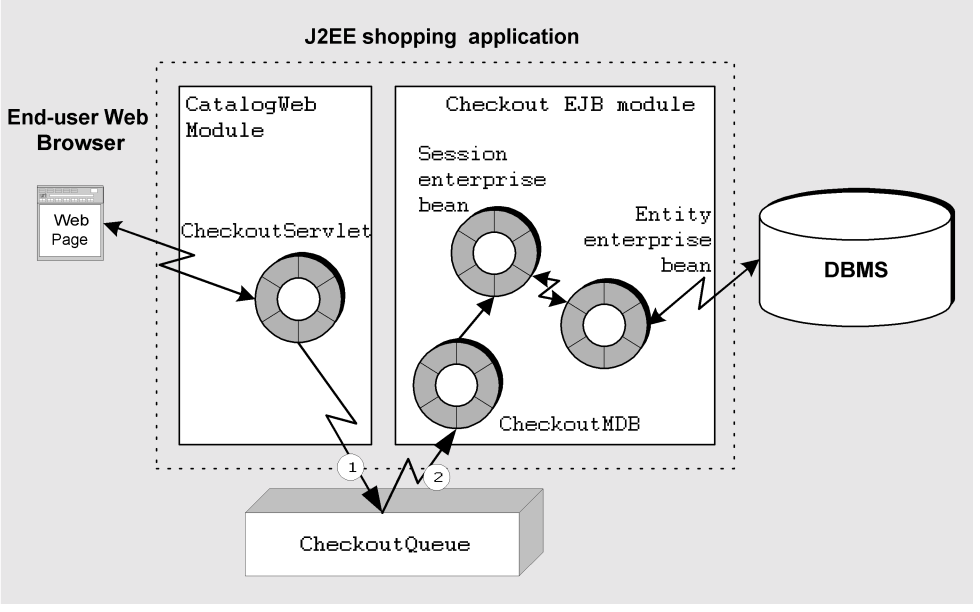

FIGURE 5-1 J2EE Application With Queue-mode Message-driven Bean

The Interactions in This Application

This scenario describes one possible use of the J2EE application and the interactions illustrated in FIGURE 5-1. This scenario continues the retail web site application, but it explores a different kind of interaction, which is implemented in a different EJB module.

In this scenario a shopper checks out. Before checking out, the shopper interacts with web pages that are defined in the web module. The shopper selects items and adds them to a shopping cart. Some of the shopper's actions invoke the business methods of an EJB module in order to select data. The application processes these actions synchronously—the shopper requests some information and then waits for the application to respond before continuing. Chapter 2, Chapter 3, and Chapter 4, show you how to implement this kind of logic in a J2EE application.

Eventually the shopper is ready to check out. The shopper reviews the contents of the shopping cart, selects a shipping method, approves the total amount, and provides a credit card number. The shopper reviews and approves the order, and leaves the site. Sometime later the application processes the order and notifies the shopper by email. The specific interactions in this checkout scenario are outlined in the following list:

1. The web module displays a page that shows the items ordered, the delivery address, the shipping method, and the payment method. The shopper approves the order and leaves the site. The details of the order are saved in a database.
2. When the shopper approves the order, the web module sends a message to a message queue. The message identifies the order to be processed.
3. The message queue is outside the application. It is maintained by the application server.
4. The queue delivers the message to a message-driven enterprise bean in an EJB module that performs order processing. The queue delivers the message by calling the message-driven bean's `onMessage` method, passing the message as a parameter.
5. The message-driven bean does not contain the business logic for processing the order. It only examines the message and initiates order completion processing. The message-driven bean initiates order completion processing by calling business methods of other enterprise beans in the module. This is a typical way of using a message-driven bean.
6. When the order is processed, the application sends an email message to notify the shopper.

The procedures in this chapter show you how to set up a message queue and a queue connection factory, and how to configure the sending module and the receiving module to use the queue.

Programming the Message-driven Communication

TABLE 5-1 summarizes the programming required to implement the message-driven interaction described in this scenario and illustrated in FIGURE 5-1.

TABLE 5-1 Programming Required by This Application

Application Element	Setup Needed
Application Server	Set up a queue name `CheckoutQueue` and a queue connection factory name `CheckoutQCF`. Do this in the IDE with the application server's administration tools.
Web Module	Create the servlet `CheckoutServlet` that sends the message. Add code to the `processRequest` method of `CheckoutServlet` that: 1. Uses JNDI lookups to obtain references to `CheckoutQueue` and `CheckoutQCF`. 2. Calls `CheckoutQueue` methods to format and send a message.
EJB Module	Create the message-driven enterprise bean `CheckoutMDB`. Use the `CheckoutMDB` property sheet to configure `CheckoutMDB` as a message destination for the `CheckoutQueue`. Code the `onMessage` method of `CheckoutMDB`.

The sections that follow show you how to perform these programming tasks.

Other programming tasks are required to program the complete application. These tasks include creating the web components and the web module, creating the session and entity enterprise beans, and creating the EJB module. These tasks are covered in other chapters that focus on those issues. This chapter focuses on the message-driven interaction.

Setting up the Application Server

The design for the checkout interaction calls for the web module to send messages to a queue and for the EJB module to read the messages from the queue and then process the orders identified in the messages. This interaction requires a queue and a queue connection factory. The queue and queue connection factory are resources of the application server that are created outside the IDE.

- If you are working in a stand-alone development environment, you need to manage your own queue and queue connection factory.
- If you are working in a managed test environment or a production environment, system administration will probably define, configure, and manage the queues and queue connection factories. In this case you simply need to obtain the JNDI names for the queue and queue connection factory. Nevertheless, reading the following instructions for setting up queues and queue connection factories will help you understand how the application uses these resources.

Before you can complete the procedures in this section, you need to have an application server and an IDE application server plugin installed. You also need an application server instance create. The application server plugin and the server instance are represented by nodes that appear in the Explorer's runtime tab. For more information on the application server plugin and application server instance nodes, see "The Server Product Nodes" on page 127.

Setting up a Queue

This section explains how to set up a message queue for the Sun ONE application server. Procedures for other application servers should be similar.

To add a queue to the Sun ONE application server:

1. **Click the Explorer's Runtime tab.**
2. **Expand the Sun ONE Application Server 7 node.**
3. **Right-click the Unregistered JMS Resources node and choose Add New JMS Resource.**

   The New wizard's JMS Resources page opens.
4. **Define the queue:**

   a. **In the JNDI Name field, type `jms/CheckoutQueue`.**

   b. **Make sure that the Resource Type field is set to `javax.jms.Queue`.**

   c. **Click Next.**

   The New wizard's Properties page opens.

5. **Define an** `imqDestinationName` **property.**

   a. **Click Add.**

      The first property line is activated.

   b. **In the Name field, select** `imqDestinationName`**.**

   c. **In the Value field, type** `Checkout`**.**

      `Checkout` is the name for the physical queue that you are creating. Your J2EE application will use the JNDI name that you assigned, `CheckoutQueue`, to access the queue named `Checkout`.

   d. **Click Finish.**

      The Do you want to continue with registration dialog box opens.

6. **Register the queue:**

   a. **Click Register.**

      The Java Mail Session Registration dialog box opens.

   b. **In the Server Instance field, select the application server instance you are registering the queue with.**

      Select the application server to which you will deploy you J2EE application.

   c. **Click Register.**

      A message that reads Resource Registered Successfully is displayed.

   d. **Click Close.**

Setting up a QueueConnectionFactory

This section explains how to set up a queue connection factory for the Sun ONE application server. Procedures for other application servers should be similar.

To add a queue connection factory to the Sun ONE application server:

1. **Right-click the Unregistered JMS Resources node and choose Add New JMS Resource.**

   The New wizard's JMS Resources page opens.

2. **Define the queue connection factory:**

   a. **In the JNDI Name field, type** `jms/CheckoutQCF`**.**

   b. **Make sure that the Resource Type field is set to** `javax.jms.QueueConnectionFactory`**.**

c. **Click Finish.**

   The Do you want to continue with registration dialog box opens.

3. **Register the queue connection factory:**

   a. **Click Register.**

      The Java Mail Session Registration dialog box opens.

   b. **In the Server Instance field, select the application server instance you want to register the queue connection factory with.**

      Select the application server instance you selected when you created the queue.

   c. **Click Register.**

      A message that reads Resource Registered Successfully is displayed.

   d. **Click Close.**

Programming the Web Module

In this scenario, the `CheckoutServlet` sends a message that requests final processing of an order. The message identifies the order to be processed. To send a message, the `CheckoutServlet` calls methods of the queue connection factory and the queue.

To call queue and queue connection factory methods, the CheckoutServlet needs references to the queue and queue connection factory. `CheckoutServlet` uses JNDI lookups to obtain queue and queue connection factory references from the application server environment.

Like most J2EE reference lookups, the queue and queue connection factory reference lookups have two parts:

- **JNDI lookup code.** The servlet includes code that uses the JNDI naming facility to obtain references to the queue or queue connection factory.
- **A declaration of the reference.** This maps the name that is used in a JNDI lookup statement to the actual JNDI name of a queue or queue connection factory.

The queue and queue connection factory are named resources of the application server. You application components use JNDI names to obtain the references. To see how JNDI names are assigned to the queue and queue connection factory, see “Setting up the Application Server” on page 88.

The JNDI Lookup Code

CODE EXAMPLE 5-1 shows the `processRequest` method of the `CheckoutServlet`. The `processRequest` method performs the JNDI lookups. After obtaining queue and queue connection factory references, `processRequest` calls methods of the queue and queue connection factory to create and send message. The code example contains comments that identify these operations.

CODE EXAMPLE 5-1 is from a servlet, but any type of J2EE component can use similar code to send a message. You can reuse this code in an application client or in an enterprise bean that acts as a message sender.

CODE EXAMPLE 5-1 The `processRequest` Method of the `CheckoutServlet`

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.jms.*;
import javax.naming.*;

// ...

protected void processRequest(HttpServletRequest request,
                              HttpServletResponse response)
                              throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  //output your page here

  // Delete the default method body and insert the following lines:
  Context jndiContext = null;
  javax.jms.TextMessage msg = null;
  QueueConnectionFactory queueConnectionFactory = null;
  QueueConnection queueConnection = null;
  QueueSession queueSession = null;
  Queue queue = null;
  QueueSender queueSender = null;
  TextMessage message = null;

  out.println("<html>");
  out.println("<head>");
  out.println("<title>Servlet</title>");
  out.println("</head>");
  out.println("<body>");

  try {
    // Connect to default naming service -- managed by app server
```

CODE EXAMPLE 5-1 The `processRequest` Method of the `CheckoutServlet` *(Continued)*

```
    jndiContext = new InitialContext();
  }
  catch (NamingException e) {
    out.println("Could not create JNDI " + "context: " +
    e.toString());
  }

  try {
    // Perform JNDI lookup for default QueueConnectionFactory and the
    // Queue created in this scenario.
    queueConnectionFactory = (QueueConnectionFactory)
                jndiContext.lookup("java:comp/env/jms/CheckoutQCF");
    queue = (Queue)
               jndiContext.lookup("java:comp/env/jms/CheckoutQueue");
  }
  catch (NamingException e) {
    out.println("JNDI lookup failed: " + e.toString());
  }

  try {
    // Use references to connect to the queue and send a message.
    queueConnection =
                      queueConnectionFactory.createQueueConnection();
    queueSession = queueConnection.createQueueSession(false,
                                           Session.AUTO_ACKNOWLEDGE);
    queueSender = queueSession.createSender(queue);
    message = queueSession.createTextMessage();
    message.setText("Order #33454344");
    queueSender.send(message);
  }
  catch (JMSException e) {
    out.println("Exception occurred: " + e.toString());
  }
  finally {
    if (queueConnection != null) {
      try {
        queueConnection.close();
      }
      catch (JMSException e) {}
    } // end of if
  } // end of finally
  // and the end of code to insert

  out.close();
}
```

For more information on creating and sending messages, see *Building Enterprise JavaBeans Components*.

The Reference Declaration for the Queue

Reference declarations appear in the module's deployment descriptor. A reference declaration maps the reference name used in the lookup statement to a JNDI name in the application server environment.

To set up a reference declaration for a queue:

1. **Right-click the web module's web node and choose Properties → References tab → Resource Environment References → ellipsis (...) button.**

   The Resource Environment Reference property editor opens.

2. **Click the Add button.**

   The Add Resource Environment Reference dialog box opens.

3. **Declare the Resource Environment Reference.**

   a. **In the Name field, type the reference name that appears in the `lookup` statement.**

      FIGURE 5-1 shows the value `jms/CheckoutQueue` in the Name field. This is the reference name used in CODE EXAMPLE 5-1.

   b. **In the Type field, select `javax.jms.Queue`.**

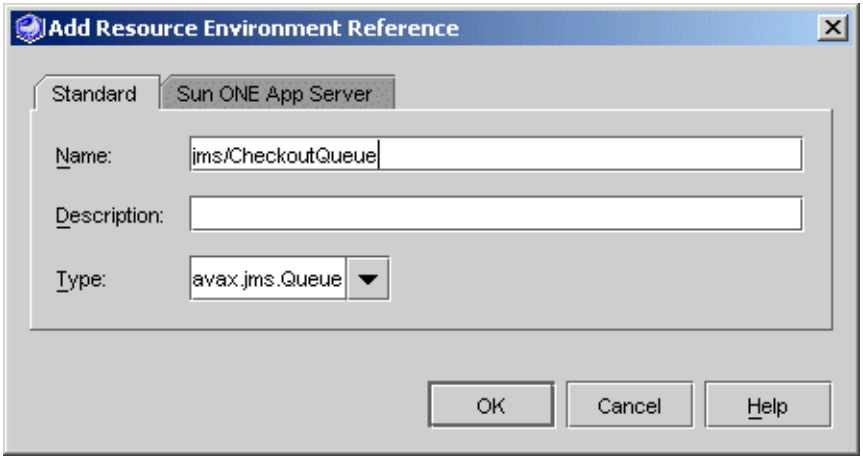

FIGURE 5-2 Adding a Resource Environment Reference for `CheckoutQueue`

4. **Map the reference name to a JNDI name.**

a. Click the Add dialog's Sun ONE App Server tab.

b. In the JNDI Name field, type the JNDI name of the queue.

FIGURE 5-3 shows the value `jms/CheckoutQueue` in the JNDI Name field. This value maps the reference name on the Standard tab to the queue named `CheckoutQueue`. To see how the queue was named, see "Setting up the Application Server" on page 88.

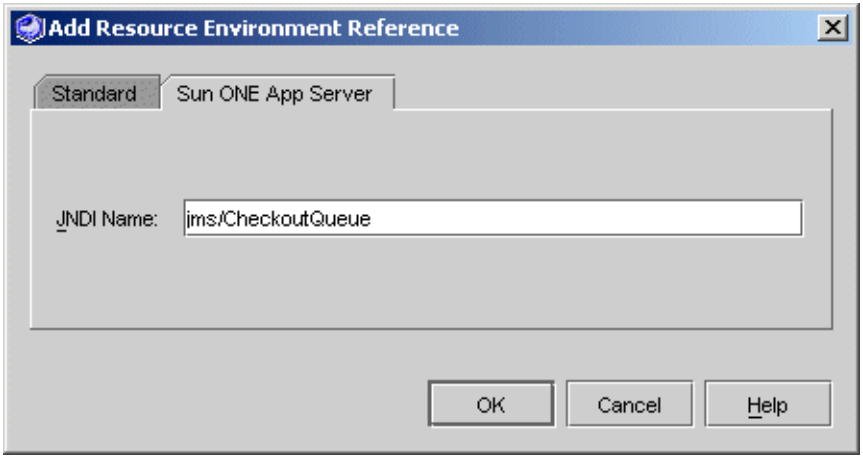

FIGURE 5-3 Supplying JNDI Name for the Queue Reference

The Reference Declaration for the QueueConnectionFactory

To set up a reference declaration for a queue connection factory:

1. **Right-click the web module's web node and choose Properties → References tab → Resource References → ellipsis (...) button.**

   The Resource Reference property editor opens.

2. **Click the Add button.**

   The Add Resource References dialog box opens.

3. **Declare the resource reference.**

   a. In the Name field, type the reference name that appears in the `lookup` statement.

   FIGURE 5-4 shows the value `jms/CheckoutQCF` in the Name field. This is the reference name used in CODE EXAMPLE 5-1.

   b. In the Type field, select `javax.jms.QueueConnectionFactory`.

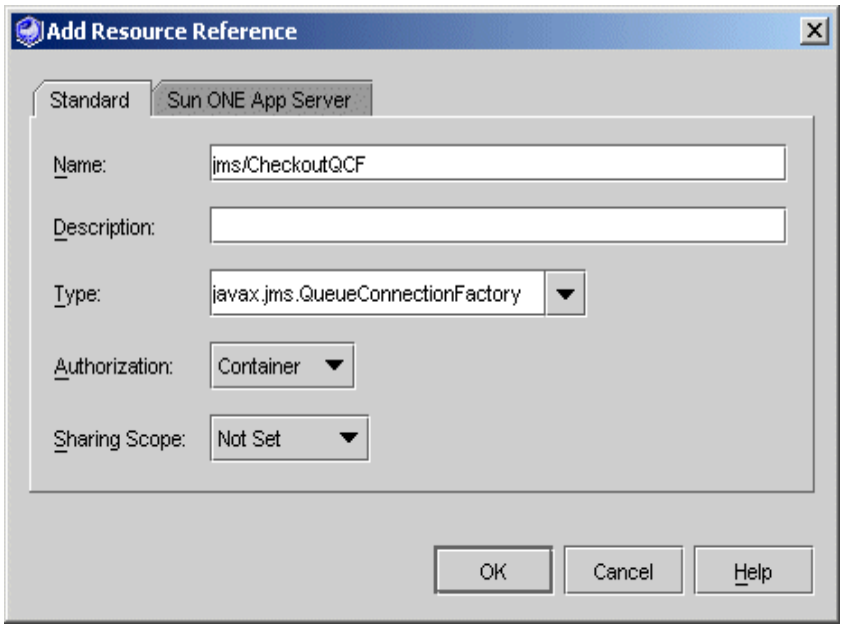

FIGURE 5-4 Resource Reference for QueueConnectionFactory

4. **Map the reference name to a JNDI name.**

   a. **Click the Add dialog's Sun ONE App Server tab.**

   b. **In the JNDI Name field, type the JNDI name of the queue connection factory.**

      FIGURE 5-5 shows the value `jms/CheckoutQCF` in the JNDI Name field. This value maps the reference name on the Standard tab to the queue connection factory named `CheckoutQCF`. To see how the queue connection factory was named, see "Setting up the Application Server" on page 88.

      For information on the other authorization types, see the coverage of message-driven beans in *Building Enterprise JavaBeans Components*.

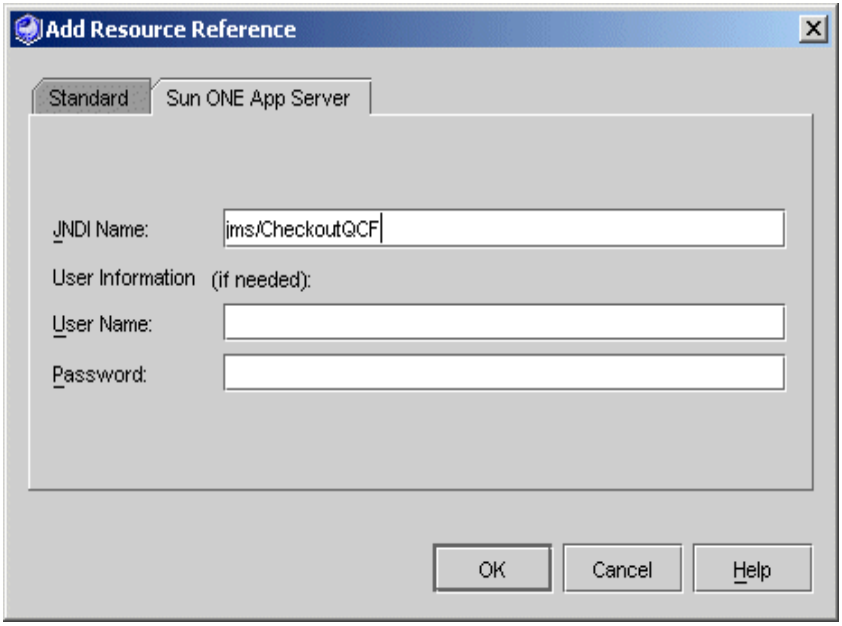

FIGURE 5-5 JNDI Name for the QueueConnectionFactory Reference

Programming the EJB Module

In this checkout scenario, the business logic for processing a shopper's checkout request is in the `Checkout` EJB module. "Programming the Web Module" on page 90 shows how the web module looks up a queue reference and a queue connection factory reference and sends a message. To complete this interaction, a message-driven bean in the EJB module needs to receive the message from the queue.

A message-driven bean does not use programmatic references. The message-driven bean does not need JNDI lookup code. You use the message-driven bean's property sheet to specify the queue and queue connection factory that should be used. Setting these properties sets up tags in the deployment descriptor. The properties that you set to configure a message-driven bean are listed below:

- **The Message-Driven Destination property.** This property specifies the type of destination used by the message-driven bean.
- **The Mdb Connection Factory property.** This property specifies the queue connection factory.
- **The JNDI Name property.** This property specifies the queue. This property is specific to the Sun ONE application server. Other application servers will use different properties to specify the queue.

When the application is deployed, the application server automatically uses the queue connection factory specified in the deployment descriptor to open a connection from the message-driven bean to the queue that is specified in the deployment descriptor.

Configuring the Message Driven Destination Property

Before you specify the queue and queue connection factory, you need to configure the message-driven bean as a queue consumer.

To configure a message-driven bean as a queue consumer:

1. **Right-click the message-driven bean's logical node and choose Properties → Message-Driven Destination → ellipsis (...) button.**

   The Message-Driven Destination property editor opens.

2. **Identify the message driven bean as a queue consumer.**

   a. In the Destination Type field, select `Queue`**.**

FIGURE 5-6 Message-driven Bean Property Sheet

   b. Click OK.

Specifying the Connection Factory

To configure a message-driven bean for a queue connection factory:

1. **Right-click the message-driven bean's logical node and choose Properties → Sun ONE AS tab → Mdb Connection Factory→ ellipsis (...) button.**

   The Mdb Connection Factory property editor opens.

2. **Specify the queue connection factory.**

a. In the Jndi Name field, type the queue connection factory's JNDI name.

FIGURE 5-7 shows the value `jms/CheckoutQCF` in the Jndi Name field. `jms/CheckoutQCF` is the queue connection factory that was specified in the sending web module.

b. If a user name and password are needed, type them in the Name and Password fields.

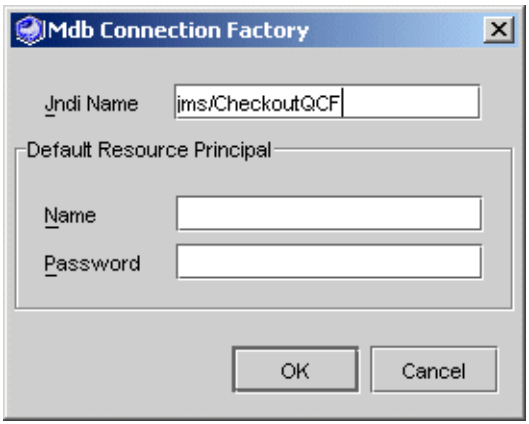

FIGURE 5-7 Message-driven Bean's Connection Factory Property Editor

c. Click OK.

Specifying the Queue

To configure a message-driven bean for a queue:

1. **Right-click the message-driven bean's local node and choose Properties → Sun ONE AS tab → JNDI Name→ ellipsis (...) button.**

   The JNDI Name property editor opens.

2. **Specify the queue.**

   In this scenario, use `jms/CheckoutQueue`. `jms/CheckoutQueue` is the queue that was specified in the sending web module.

Coding the onMessage Method

The application server delivers messages to the message-driven bean by calling the bean's onMessage method. The message is delivered as a parameter of the onMessage method. CODE EXAMPLE 5-2 shows the onMessage method. You can see the message passed as a parameter and where you add your message-handling code.

CODE EXAMPLE 5-2 The onMessage Method

```
public void onMessage(javax.jms.Message aMessage) {
        // Process the message here.
    }
```

In this scenario, as shown in FIGURE 5-1, the message-driven bean immediately calls a business method of a session bean in the same EJB module. The session bean controls the processing of the order. This is typical onMessage behavior. For more information about writing onMessage methods, see *Building Enterprise JavaBeans Components*.

The message-driven bean calls the session bean using an EJB local reference. For information about how you implement method calls using EJB local references, see "JNDI Lookup Code for Local EJB References" on page 61 and "Reference Declarations for Local EJB References" on page 62.

Assembling the J2EE Application

Figure FIGURE 5-1 shows a web module that sends messages and an EJB module that receives messages assembled into a J2EE application. The modules are programmed as described in this chapter. The application is created and the two modules are added to the application. Both modules have been configured to use the CheckoutQueue and CheckoutQCF. For the message-driven interaction, there is no need to open the J2EE application property sheet and perform any additional assembly work.

For information about creating an application and adding modules, see "Creating the J2EE Application" on page 75.

CHAPTER 6

Transactions

This chapter covers the use of EJB module property sheets to program container-managed transactions. For bean-managed transactions, see *Building Enterprise JavaBeans Components*.

Default Transaction Boundaries

Transaction boundaries are determined by the Transaction Attribute properties of the enterprise beans involved in the transaction.

When you create an enterprise bean with the IDE's EJB wizards and select container-managed transactions, the wizard creates the enterprise bean with the default value for the transaction attribute property. This section shows you how to view the default transaction attribute settings and interpret them.

To open the Transaction Settings property editor and review the default settings:

- **Right-click the EJB module node and choose Properties → Transaction Settings property → ellipsis (...) button.**

  The Transaction Setting property editor opens.

  FIGURE 6-1 shows the Transaction Settings Property Editor for the `CatalogData` EJB module that was covered in Chapter 3. The values displayed in FIGURE 6-1 are the default transaction attribute settings.

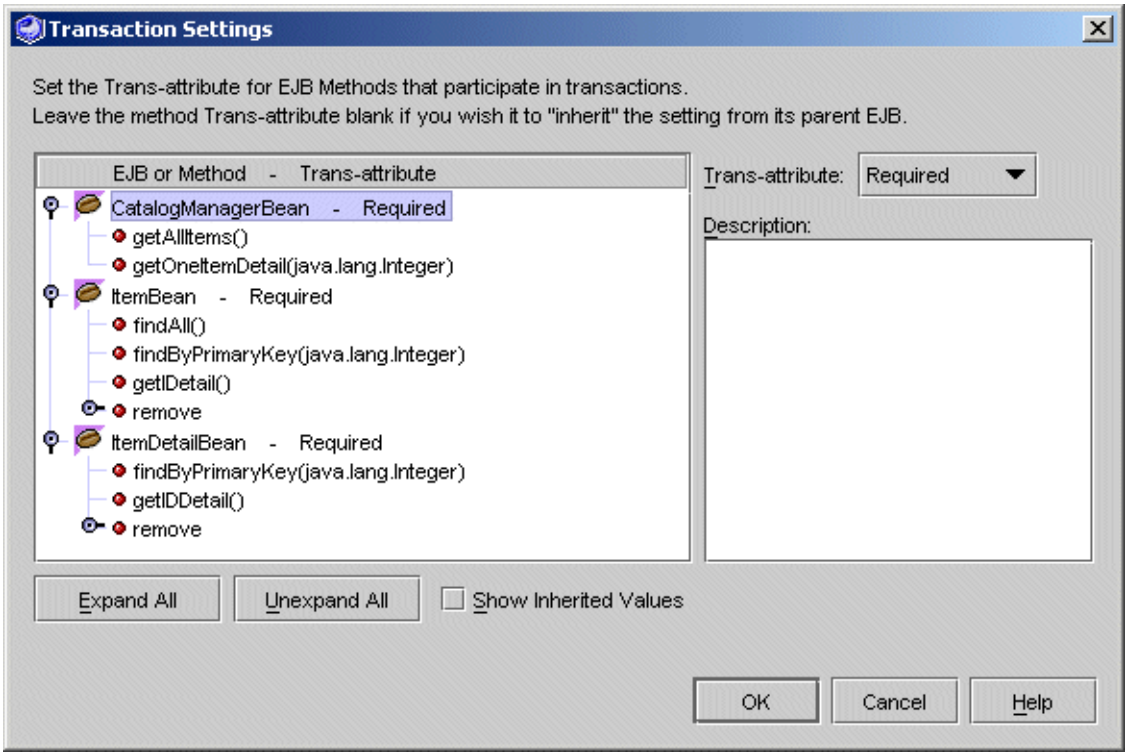

FIGURE 6-1 Default Transaction Attribute Settings

The default transaction attribute settings are displayed in the EJB or Method - Trans-attribute field. To understand this display, notice the following features of the display:

- Every enterprise beans in the module appears in this display. Each enterprise bean is represented by a node.
- Each enterprise bean name is followed by the bean's transaction attribute. For example, the first node in FIGURE 6-1, CatalogManagerBean, is followed by the word `Required`. `Required` is the default setting for the transaction attribute property. In FIGURE 6-1, all of the enterprise beans have the default setting.
- The nodes that represent enterprise beans can be expanded to display nodes that represent the methods of each enterprise bean. In FIGURE 6-1, all of the enterprise bean nodes are expanded.
- Methods have their own transaction attribute values. The transaction attribute values appear after the method names. If the transaction attribute value is null, the method inherits its transaction attribute value from the enterprise beans.
- In FIGURE 6-1, none of the method nodes displays a transaction attribute value. All of the methods inherit the `Required` value. This is the default setting.

`Required` is one of the transaction values defined in the *Java 2 Platform, Enterprise Edition Specification*. The rules for methods that have the `Required` transaction attribute are listed below:

- If a method with the `Required` attribute is called without an active transaction, the application server starts a new transaction.
- If a method with the `Required` attribute is called with an active transaction in progress, the application server executes the method within the active transaction.

This is the default behavior for enterprise bean methods.

Redefining the Transaction Boundaries

A business transaction in an EJB module often spans several enterprise beans. A common architecture for enterprise bean business logic is one session bean and several entity beans. Clients call the session bean, and then the session bean calls methods of the entity beans.

For example, the client has data for two new related database records. The session bean generates database inserts by calling create methods of the two entity beans. The two database inserts must be in the same transaction. FIGURE 6-2 shows a transaction of this type.

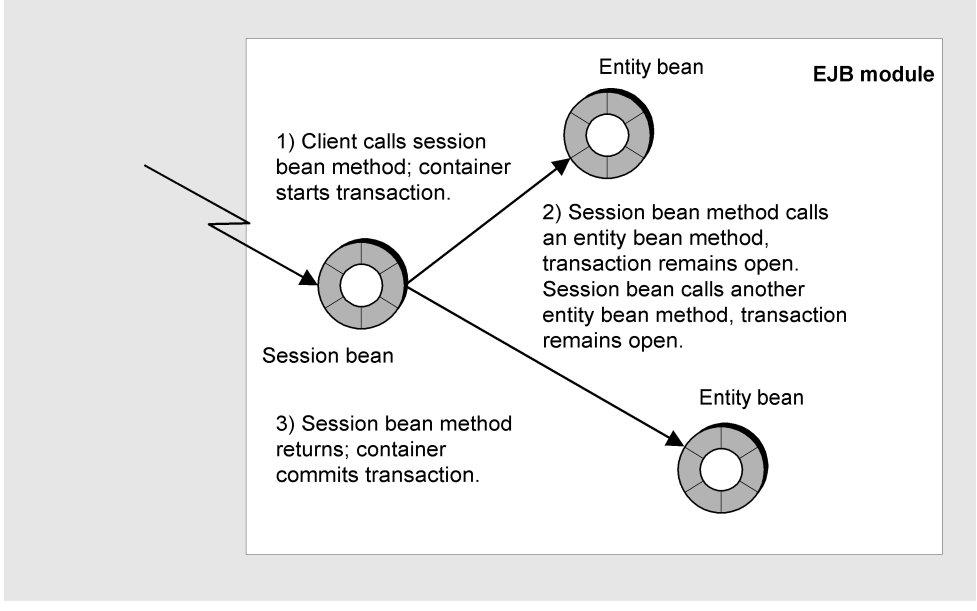

FIGURE 6-2 Complex Transaction

When you assemble these entity beans into an EJB module, you want the application server to recognize the boundaries of the business transaction. You want the application server to include all of the work that the EJB module performs after a client call in a single database transaction. You want the application server to do the following:

- You want the application server to open a transaction when a client calls the session bean.
- You want the application server to keep the transaction open while the session bean calls the entity beans.
- You want the application server to close the transaction when the last call from the session bean to an entity bean returns and the session bean method that was called by the client completes.

If the application server recognizes these transaction boundaries, all of the work performed by the EJB module will be committed or rolled back together.

To configure transaction boundaries, you open the Transaction Settings property editor and modify the transaction attributes of the enterprise beans involved in the transaction.

The procedure that follows demonstrates editing the transaction attribute settings. This procedure changes the default transaction settings that are shown in FIGURE 6-1 to transaction attribute settings that produce the transaction boundaries shown in FIGURE 6-2.

To modify transaction attributes:

1. **Right-click the EJB module node and choose Properties → Transaction Settings property → ellipsis (...) button.**

   The Transaction Settings property editor opens.

2. **Change the transaction attributes of the session bean methods to** `RequiresNew`**.**

   You want to change the behavior of the `CatalogManagerBean` methods so that a call to either of its business methods (`getAllItems` and `getOneItemDetail`) starts a new transaction. Changing the transaction attribute to `RequiresNew` accomplishes this.

   a. **Click a session bean method to select it.**

   b. **Change the value of the Trans-attribute field to** `RequiresNew`**.**

      Notice that the new transaction attribute value appears after the method node.

3. **Change the transaction attributes of the entity bean methods to `Mandatory`.**

   You want to change the behavior of the entity bean methods so that these methods execute within the boundaries of the transactions opened by the session bean methods. Changing the transaction attribute to `Mandatory` accomplishes this. It tells the application server that these methods can only be executed if a transaction is already in progress.

   a. **Click an entity bean method to select it.**

   b. **Change the value of the Trans-attribute field to `Mandatory`.**

      Notice that the new transaction attribute value appears after the method node.

4. **Click OK to close the editor and save your changes.**

   FIGURE 6-3 shows the Transaction Settings Property Editor with the new transaction attributes settings.

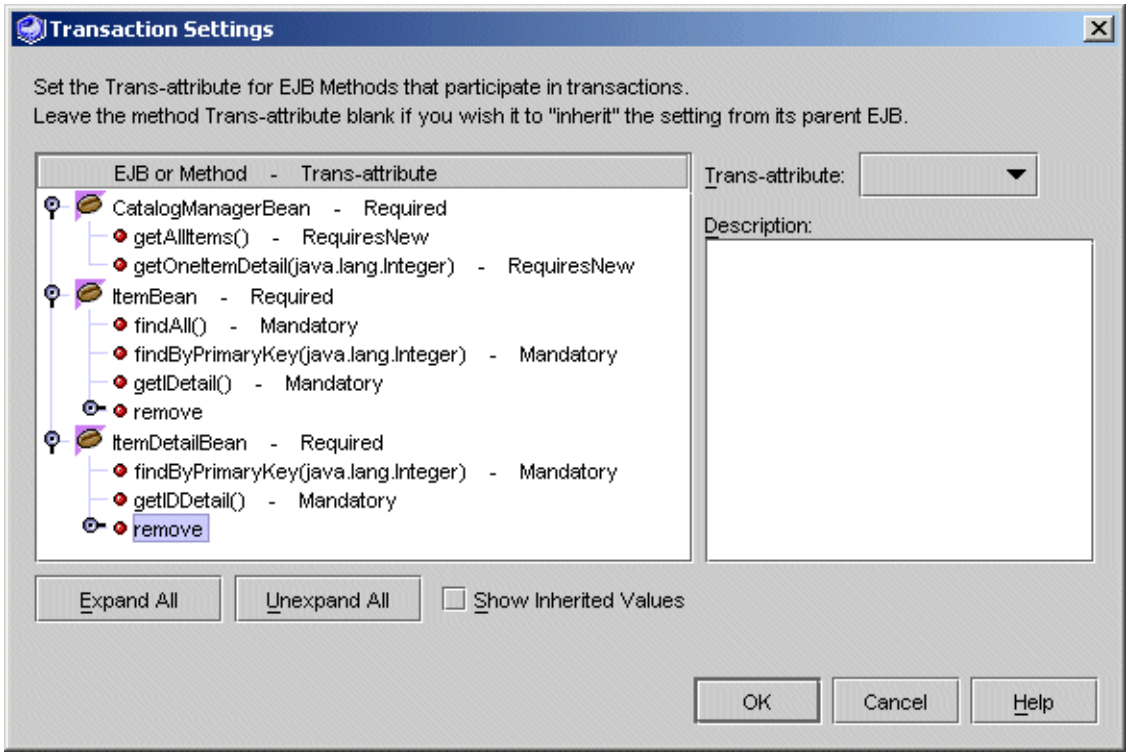

FIGURE 6-3 Modified Transaction Settings

The transaction boundaries now match the business transactions. The changes that are visible in the property editor are described in the following list:

- The transaction attributes of the session bean methods have been set to `RequiresNew`, because you want the container to open a new transaction whenever a client calls one of these methods.
- The transaction attributes of entity bean business methods have been set to `Mandatory`, which means that these methods must be called with a transaction in progress. You want these methods to be called after the session bean has opened a transaction, and you want these methods to execute entirely within the boundaries of the session bean's transaction.
- Notice that the transaction attributes were modified on the method level, and the method nodes now display the new transaction attribute values after the method names.

For each EJB module that you work with, you must analyze the business logic in the module and determine what transaction boundaries are implied by the logic. Then use the Transaction Attribute property editor to implement those transaction boundaries. Set the transaction attributes of the enterprise beans, or the individual methods, involved in those transactions to specify the transaction boundaries.

CHAPTER 7

Security

Securing an application limits access to its features and functions. Limiting access to its features and functions limits access to the data that the application manages.

J2EE applications are secured by working with application resources and user roles. The two concepts are defined in the following list:

- Resources are visible or callable features of the applications. For EJB modules, resources are public EJB methods declared on home or remote interfaces. For web modules, resources are URL patterns that are mapped to JSP pages, servlet methods, and other components.
- Roles are names that are associated with users.

J2EE applications are secured by mapping resource to roles. When a resource is called, the caller must supply a role name that is authorized to access the resource. If the caller cannot supply an authorized role, the call is rejected. In J2EE applications, the application server verifies the caller's role before allowing the caller to execute the resource.

The authorized combinations of roles and resources are declared in deployment descriptors. The application server reads them from the deployment descriptors and applies them. This is known as declarative security.

The tasks that you perform to secure a J2EE application are described in the following list:

- You declare the roles.
- You specify which roles are permitted to access the resources.

For example, suppose you are developing a web module that works with human resources data. The specification for the module tells you which web resources are available to all employees for maintenance of their personal information and which web resources are available only to human resources clerical roles, to human resources supervisory roles, to auditing roles, and so on. You declare security roles that represent these types of users and map the resources in your web module to these roles.

Securing an EJB module is similar. The specification for the module identifies different types of users and tells you which types of user are allowed to access the data that is returned by the enterprise bean methods. You declare security roles that represent these groups of users and map the EJB methods in the module to the appropriate roles.

In general you set up J2EE security on module property sheets. You declare a set of roles for the module. You map the module's resource to the set of roles declared in the module.

When modules are assembled into an application and deployed, you map the roles declared in the modules to the actual user names and group names in the application server environment. You map the roles to users and groups on the J2EE application's property sheet. You perform this mapping when the application is deployed into an environment, like a production environment, that has declared users and groups.

The sections that follow show you how to use the IDE to set up security for web modules, for EJB modules, and how to merge several sets of security declarations when you assemble modules into an J2EE application.

Web Module Security

Web modules have several property editors and dialogs for declaring security roles and mapping security roles to web resources. This section covers these property editors and dialog boxes in two separate tasks:

- Declaring security roles for web modules
- Defining the web resources that you want to secure and mapping the resources to security roles

Each task is covered in its own section.

Declaring Security Roles for Web Modules

To declare web module security roles:

1. **Right-click the web module's web node and choose Properties → Security Tab → Security Roles → ellipsis (...) button.**

   The Security Roles property editor opens.

2. **Click the Add button.**

   The Add Security Role dialog box opens.

3. **Define a new security role.**

   a. **In the Role Name field, type a name for the new security role.**

   b. **In the Description field, type a brief description of the role.**

      A description is very useful when you merge the security roles of several modules that are assembled into an application. It is also useful when you map security roles to the users and groups that are in an application server environment.

   c. **Click OK to close the dialog box.**

      FIGURE 7-1 shows the Security Roles Property editor after two roles have been declared, `Me` and `EveryoneElse`.

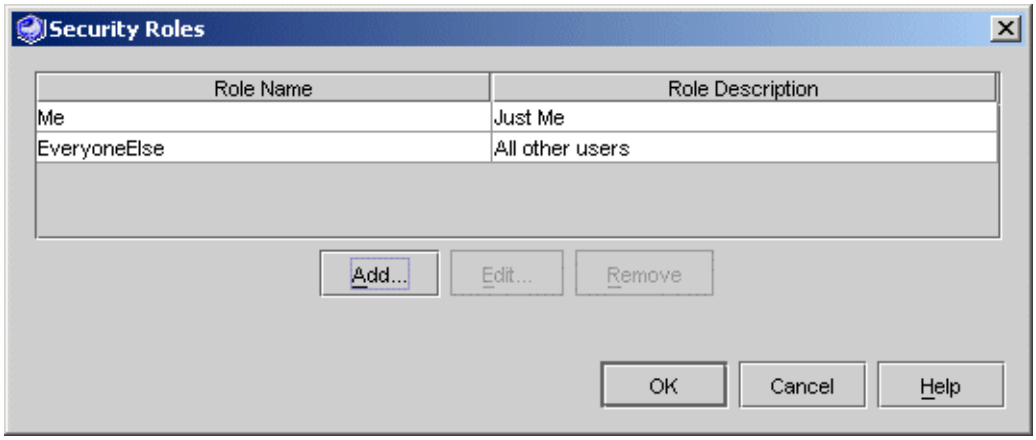

FIGURE 7-1 The Security Roles `Me` and `EveryoneElse` Declared for the Web Module

4. **Click OK again to close the property editor.**

Defining Web Resources and Mapping Resources to Security Roles

To define a web resource and secure it:

1. **Right-click the module's web node and choose Properties → Security Tab → Security Constraints → ellipsis (...) button.**

   The Security Constraints property editor opens.

2. **Click the Add button.**

   The Add Security Constraint dialog box opens.

3. **Click the Add button on the Add Security Constraint dialog.**

   The Add Web Resource Collection dialog box opens.

4. **Define a web resource.**

   a. **In the Resource Name field, type a name for the resource.**

   b. **In the URL patterns field, type a URL pattern.**

      The URL pattern must already be defined in the module. FIGURE 7-2 shows field values that define the URL pattern `/allItems` as a web resource. This URL pattern is already mapped to the servlet `AllItemsServlet`. The values in FIGURE 7-2 set up a web resource for executing the `AllItemsServlet`. To see how the URL pattern `/allItems` is mapped to the servlet `AllItemsServlet`, see "Mapping URLs to the Servlets" on page 45 and "Setting Up JSP Pages" on page 48.

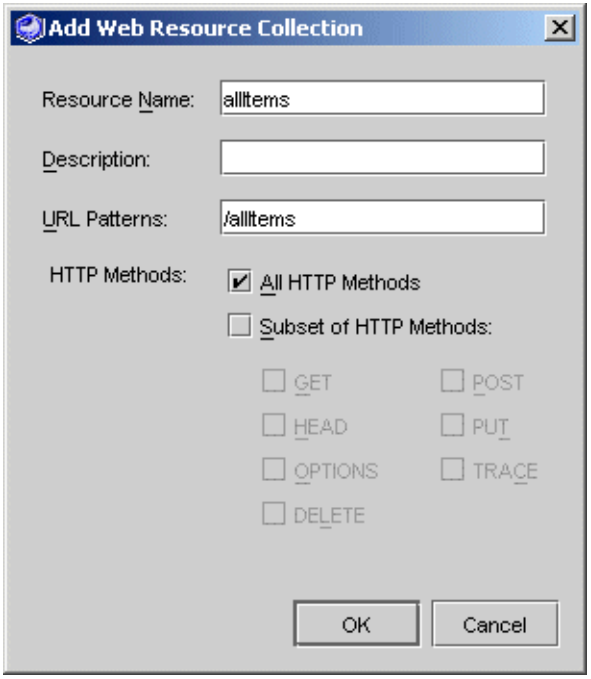

FIGURE 7-2 Defining a Web Resource Named `allItems`

Notice that you can define the web resource to apply to all of the HTTP methods associated with the URL pattern or just a subset of them.

   c. **Click OK to close this dialog and return to the Add Security Constraint dialog.**

      FIGURE 7-3 shows the Add Security Constraint dialog after two web resources are set up, under the names `AllItems` and `ItemDetail`.

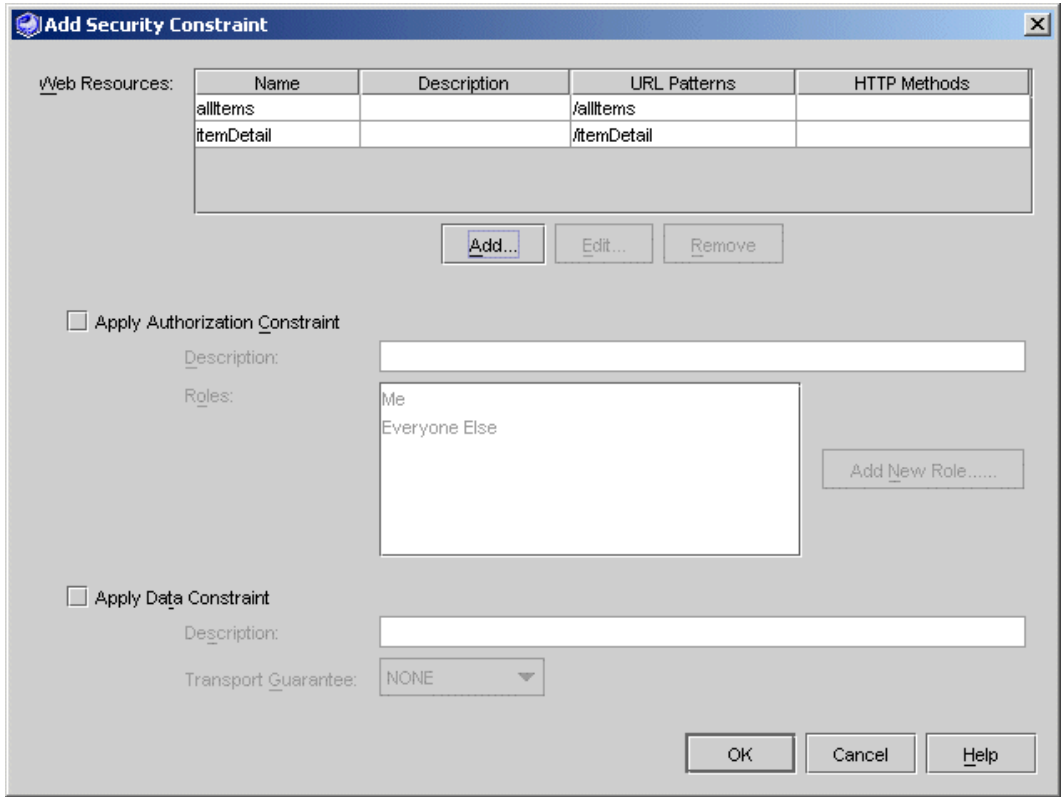

FIGURE 7-3 The `allItems` Resource in the Add Security Constraints Dialog Box

5. **Secure the resource by specifying any constraints on access to the resource.**

   a. **Click a resource to select it.**

   b. **Use the fields on the Add Security Constraints dialog box to describe the constraints on accessing the resource.**

   c. **To use security roles as constraints, click the Apply Authorization Constraint check box, and then select one or more of the roles in the Roles field.**

      In FIGURE 7-4 the `allItems` resource is selected. The `EveryoneElse` role is also selected. These selection specify that the `allItems` resource must be called with the `EveryoneElse` role. Callers with other roles will be rejected.

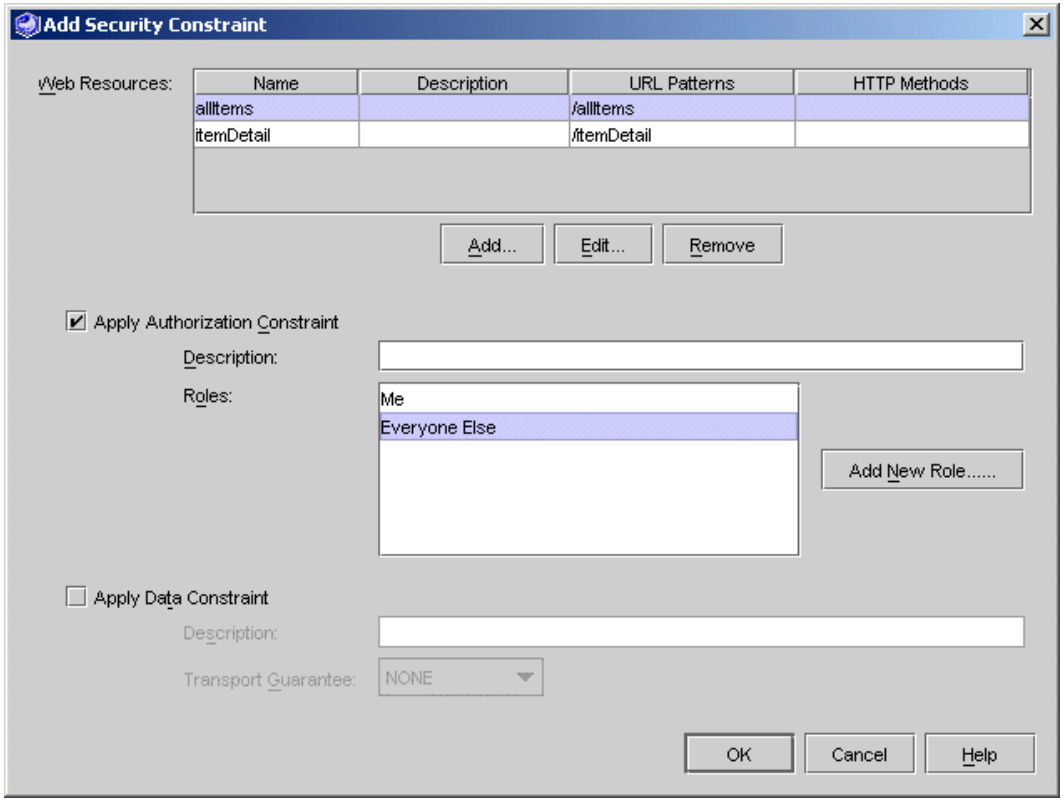

FIGURE 7-4 Specifying Constraints for the Web Resource Named `allItems`

d. Click OK to save the constraints you have specified and return to the Security Constraints property editor.

6. Click OK again to save your work and return to the property sheet.

Programmatic Security for Web Modules

If any of the web components in your module use programmatic security, you need to map the security role references used in the method code to security roles declared on the module property sheet.

Web components that use the programmatic security feature contain code that accesses the caller's credential directly and performs verification beyond the verification that is performed by the application server's declarative security mechanism. CODE EXAMPLE 7-1 shows a few lines of method code. This code uses a security role reference name `roleRefMe`.

CODE EXAMPLE 7-1 Method Code Using the `roleRefMe` Security Role Reference

```
...
context.isCallerInRole(roleRefMe);
...
```

Security role references like `roleRefMe` are placeholders for actual reference names. The method code is written before the roles are declared at the module level, and the actual role name is not known. When the module is assembled, the security role reference is declared and mapped to a declared security role.

To declare a security role reference and map it to a security role:

1. **Right-click the module's web node and choose Properties → Deployment Tab → Servlets → ellipsis (...) button.**

   The Servlets property editor opens.

2. **Select the servlet that contains the security role reference and click the Edit button.**

   The Edit Servlet dialog box opens.

3. **Click in the Security Role References field and click Add.**

   The Add Security Role Reference dialog box opens.

4. **Declare the security role reference and map the security role reference to a role.**

   a. **In the Role Ref Name field, type the role reference name that was used in the method code.**

   b. **In the Role Ref Link field, type the name of the existing security role that the reference name will be linked to.**

   c. **Click OK to close the dialog and return to the Edit Servlet dialog box.**

      FIGURE 7-5 shows the Edit Servlet dialog box. The security role reference named `roleRefMe` is mapped to the role `Me`.

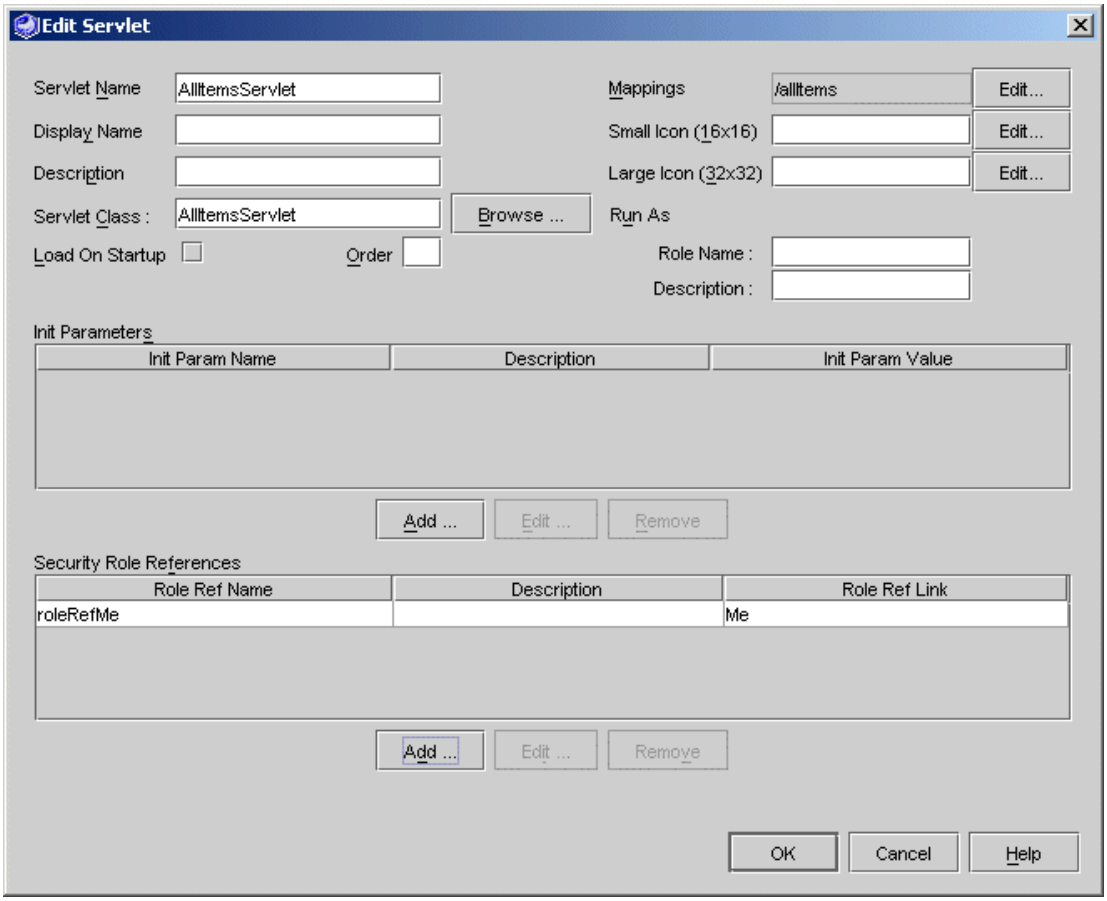

FIGURE 7-5 The Security Role Reference Named `roleRefMe` Mapped to the Role `Me`

When the method code executes, it will map the role reference to the role `Me`, and test the caller's credential for the role named `Me`. Security roles must be declared on the module property sheet before you can perform this type of mapping.

EJB Module Security

EJB modules have several property editors and dialogs for declaring security roles and mapping security roles to enterprise bean methods. This section covers these property editors and dialog boxes in two separate tasks:

- Declaring security roles for EJB modules
- Mapping enterprise bean methods to security roles

Each task is covered in its own section.

Declaring EJB Module Security Roles

To declare security roles for an EJB module:

1. **Right-click the EJB module node and choose Properties → Security Roles → ellipsis (...) button.**

   The Security Roles Property Editor opens.

2. **Click the Add button.**

   The Add Security Role dialog box opens.

3. **Type in the values that define a role.**

   a. **In the Name field, type a name for the role.**

   b. **In the Description Field, type a description of the role.**

      A description is very useful when you merge the security roles of several modules that are assembled into an application. It is also useful when you map security roles to the users and groups that are in a deployment environment.

   c. **Click OK.**

      The add dialog box closes and you return to the Security Roles property editor. FIGURE 7-6 shows the Security Roles property editor after two roles have been declared, `Me` and `EveryoneElse`.

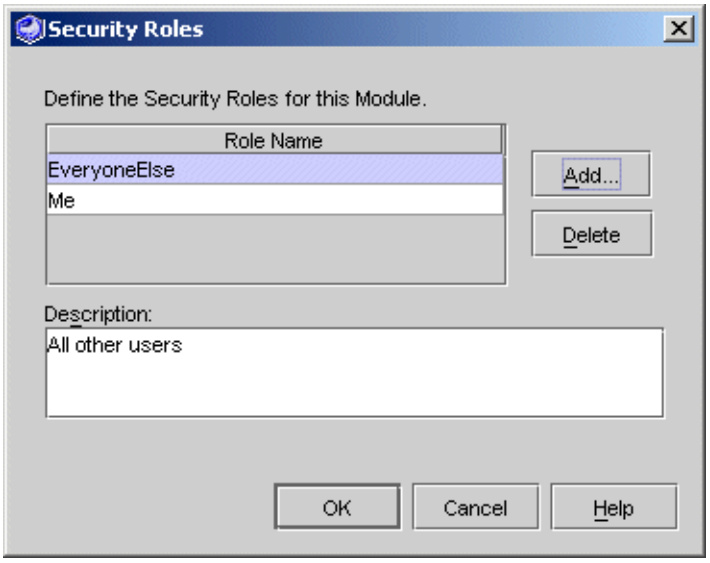

FIGURE 7-6 EJB Module's Security Roles Property Editor

Mapping Security Roles to Method Permissions

After declaring security roles for the module, you use the roles to limit access to the enterprise bean methods in the module.

Note – To map security roles you work with *included EJB nodes.* These are the subnodes of an EJB module that represent enterprise beans in the module.

To map security roles to method permissions:

- **Right-click an included EJB node and choose Properties → Method Permissions → ellipsis (...) button.**

  This Method Permissions property editor opens.

  The Method Permissions property editor is a table, with a row for each of the enterprise bean's methods and a column for each security role that has been declared for the module. FIGURE 7-7 shows how the property editor for the `CatalogManagerBean`. The two security roles declared in the `CatalogData` EJB module, `EveryoneElse` and Me, are represented by columns.

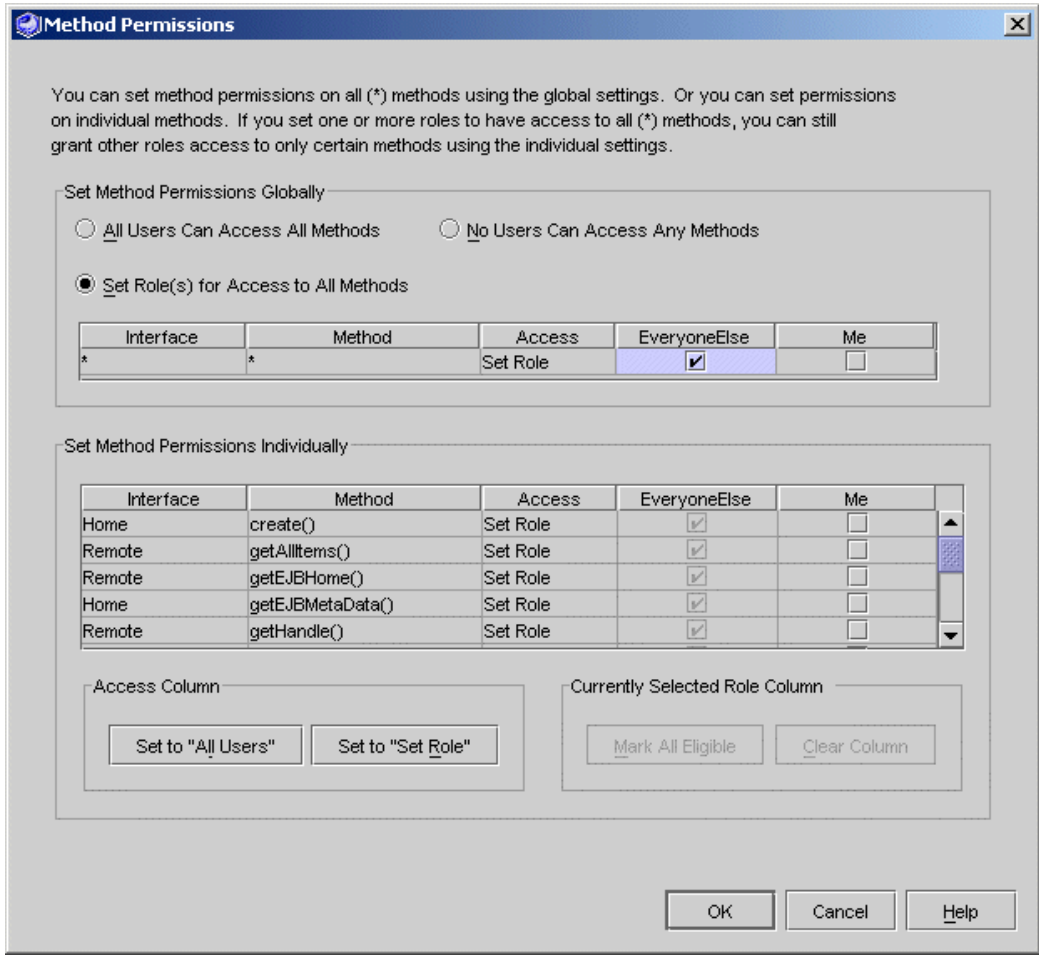

FIGURE 7-7 EJB Method Permissions Property Editor

There are a number of ways to use the Method Permissions property editor.

- The buttons in the upper panel let you apply permissions globally. You can allow any user to call any method, or deny all access.
- A finer focus of control is available if you select Set Roles for Access to All Methods. You can use the small table below the button to choose from the roles declared for the module. If you put a check in a column, the role will be given permission to execute all of the enterprise bean's methods. In FIGURE 7-7, the `EveryoneElse` column is checked. As a result, users with this role can execute any of the enterprise bean's methods. The `Me` column was not checked. Users with the `Me` role cannot execute any of the enterprise bean's methods.

- The finest focus of control is available when you work in the lower table. Click in a row, and you can define permissions for just one method. Setting permissions for one method is entirely independent of the settings for the other methods in the editor.

  For example, you could click the second row, for the `getAllItems` method, and set the Access field to `All Users`. This lets any role execute the `getAllItems` method. You could then click the row that represents another method and set the Access field to `Set Role`, and select roles individually for the selected method.

This buttons below this table provide several shortcuts for editing in the table.

Programmatic Security for EJB Modules

If any of the enterprise beans in your module use programmatic security, you need to map the security references used in the method code to the security roles declared in the EJB module's property sheet.

Enterprise beans that use the programmatic security feature contain method code that accesses the caller's credential directly and performs verification beyond the verification that is performed by the application server's declarative security mechanism. CODE EXAMPLE 7-2 shows a few lines of method code. This code uses a security role reference name `everyOne`.

CODE EXAMPLE 7-2 Method Code Using the `everyOne` Security Role Reference

```
...
context.isCallerInRole(everyOne);
...
```

Security role references are placeholders for actual reference names. The method code is written before the roles are declared at the module level, and the actual role name is not known. When the module is assembled, the security role reference is declared and mapped to a declared security role.

To declare a security role reference and map it to a security role:

1. **Right-click an EJB logical node and choose Properties → Reference tab → Security Role References → ellipsis (...) button.**

   The Security Role References property editor opens.

2. **Click the Add button.**

   The Add Security Role dialog box opens.

3. **Declare the security role reference and link it to a security role.**

a. In the Name field, type the security role reference name that is used in the method code.

b. In the Security Role Link field, type the name of security role. You can also choose to leave the field blank, which leaves the reference declared but unlinked.

c. click OK.

The Add Security Role dialog box closes and you return to the Security Role References property editor. FIGURE 7-8 shows the Security Role References property editor displaying the security role reference used in CODE EXAMPLE 7-2, named `everyOne`. The role is unlinked.

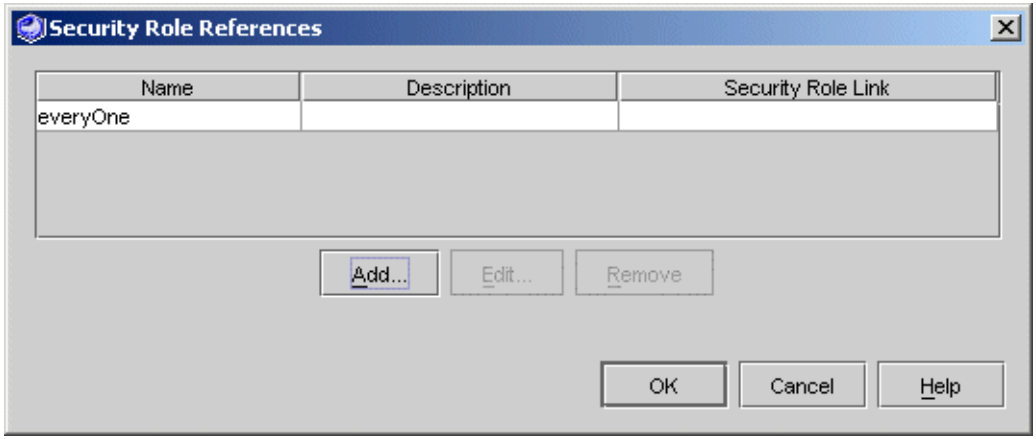

FIGURE 7-8 The Security Role Reference `everyOne` Declared

In this scenario, the `everyOne` security role reference is linked in the EJB module's Security Roles Property Editor. You perform this mapping when the enterprise bean is assembled into an EJB module.

To link a security role reference in the EJB module's Security Role References property editor:

1. **Right-click an EJB module node and choose Properties → Security Role References → ellipsis (...) button.**

   The Security Role References property editor opens.

2. **Evaluate the references in the module.**

   The EJB module Security Role References property editor shows all of the security role references in the module and indicates whether they are linked or unlinked. In FIGURE 7-9 the `everyOne` reference is unlinked.

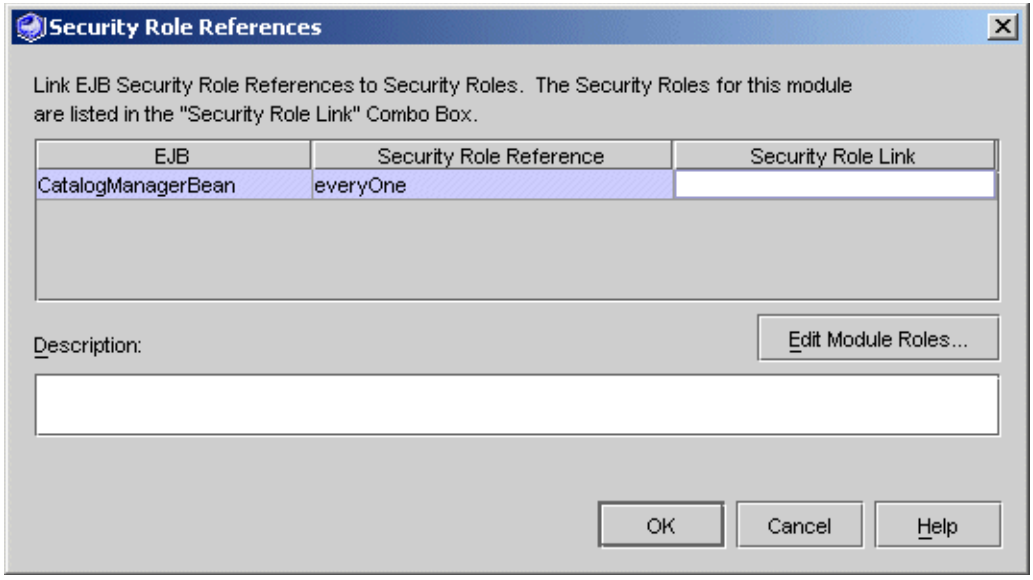

FIGURE 7-9 The `everyOne` Security Role Reference in the EJB Module Property Editor

3. **Link the unlinked roles.**

   a. **Click a reference to select it.**

   b. **In the Security Role Link field, select the appropriate security role.**

   c. **Click OK.**

   FIGURE 7-10 shows the EJB module Security Role References property editor. The security role reference named `everyOne` link has been mapped to the security role named EveryoneElse.

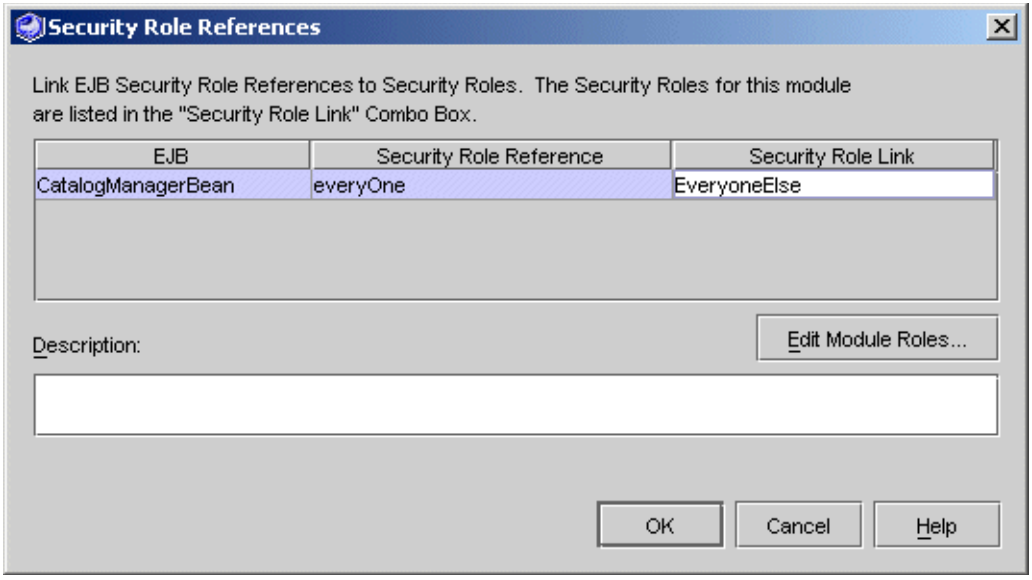

FIGURE 7-10 EJB Module's Security Role Reference Property Editor

J2EE Application Security

When you assemble a J2EE application you should determine whether it has been secured. Look for the following conditions:

- If security roles are not defined in one or more modules in your application, you need to define security roles at the module level. See "Web Module Security" on page 108 and "EJB Module Security" on page 114.
- If the modules are secured, they may container similar roles with different names in different modules. If this is true you need to map all of the equivalent roles to the same role in the J2EE application's property sheet.

To map security roles at the J2EE application level:

- **Right-click the application node and choose Properties → Security Roles → ellipsis (...) button.**

  The J2EE application's Security Roles property editor opens. FIGURE 7-11 shows this editor with the security roles that are defined in the modules.

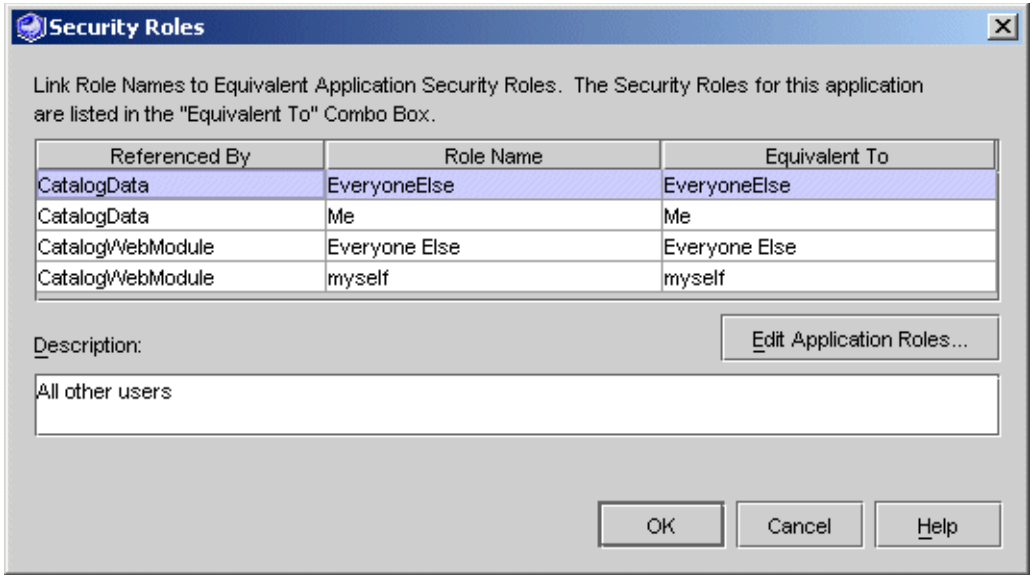

FIGURE 7-11 Security Roles in the J2EE Application's Security Roles Property Editor

The security roles that are declared in the modules are displayed in the first two columns. Each role is identified by its module and role name. Each module-level security role is mapped to a default application-level role. The application-level roles have the same names as the module-level role. The application-level roles are displayed in the Equivalent To column.

For example, the first security role displayed in the editor is `EveryoneElse` from the `CatalogData` module. This role is mapped to an application level role that is named `EveryoneElse`.

FIGURE 7-11 shows a discrepancy in the security role mapping. `CatalogWebModule` has a role named `myself` and the `CatalogData` module has a role named `Me`. These roles are equivalent, and you would prefer to have only one application-level role.

In the FIGURE 7-12 this discrepancy is resolved by remapping the role `myself` to the role `Me`.

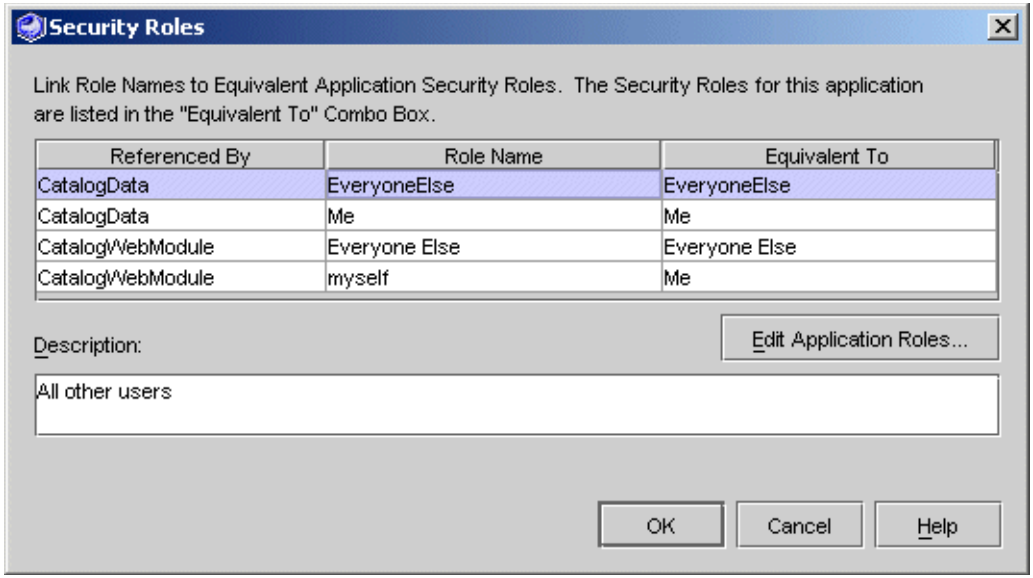

FIGURE 7-12 The Role Named `myself` is Mapped to the Role Named `Me`

Both module-level roles, `Me` and `myself`, are now mapped to the same application-level role, `Me`.

You can also create an entirely new role at the application level and map several module-level roles to it. Suppose one of the modules in your applications has a role named `sa`, and the other has a role named `sadmin`. You decide to resolve this discrepancy by creating a new application-level role named `sysadmin`. To do this click Edit Application Roles and declare a new application-level role named `sysadmin`.

After declaring the `sysadmin` role, you return to the Security Roles property editor. Remap both of the module-level roles by clicking in their “Equivalent To” column. This will display the application-level roles. Select the `sysadmin` role.

CHAPTER 8

Deploying and Executing J2EE Modules and Applications

The IDE's deployment and execution feature supports the iterative development of enterprise applications. You can develop and assemble an application, deploy the application, and execute the application, all without leaving the IDE.

After you execute your application, you can modify the source code or the properties, redeploy the application, and execute it again. You do not need to reassemble before you redeploy, unless your testing uncovers a problem with the assembly.

This chapter describes the basics of deploying and executing assembled applications from within the IDE.

Visual Representations of Servers

To deploy an application to an application server you need to interact with the application server. To simplify your interaction with the application server, the IDE represents application servers as nodes in the Explorer window.

Like other Explorer window nodes, the application server nodes have property sheets and menu commands. You use these property sheets and menu commands to deploy and execute your applications from inside the IDE. Depending on the application server product you are working with, you may also be able to administer the application server.

This section identifies and describes the server nodes. It also describes some basic tasks you perform with these nodes.

The Server Registry Node

FIGURE 8-1 shows the Explorer's Runtime tab with the nodes you use for server configuration, deployment, and execution. The top level node is the Server Registry node. This node groups the other server-related nodes. It has no commands or properties of its own.

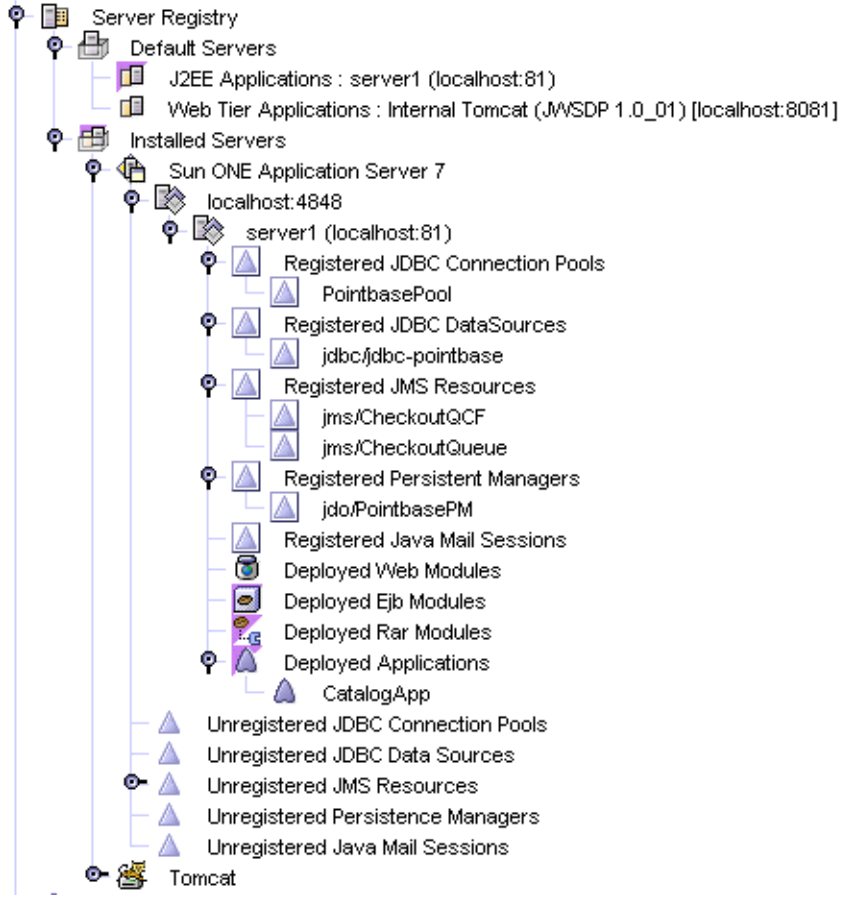

FIGURE 8-1 Server Registry Node

The Server Registry in FIGURE 8-1 is from an installation of the IDE on the Microsoft Windows platform. This particular installation is a standalone installation by a user with administrative, or superuser, privileges. When the IDE is installed by a system administrator for a user who does not have administrative privileges, the hostnames and port numbers displayed by the nodes are different. The host names and port numbers will also be different for multiuser installations.

For a complete description of the different installation options, see *Sun ONE Studio 5, Standard Edition Getting Started Guide*.

The Installed Servers Node

The Installed Servers node groups the server product nodes. It has no commands or properties of its own. In FIGURE 8-1 the Installed Servers node has subnodes for two server products that are included in most installations of the IDE, the Sun ONE Application Server and the Tomcat web server.

The Server Product Nodes

Below the Installed Servers node are nodes for specific web and application server products. Each of these server product nodes represents an installed IDE server plugin.

The server product nodes have contextual menus and property sheets. The capabilities of each sever product node are determined by the server product and the plugin module.

Procedures vary with server product, but, in general, to use an application server you configure the appropriate server product node to recognize a specific installation of the server product. You may also use the server product node to create a server instance.

In FIGURE 8-1 the Installed Servers node has subnodes for the two plugins that are included in most installations of the IDE, the Sun ONE Application Server plugin and the Tomcat web server plugin. The next section describes the Sun ONE Application Server 7 node and its subnodes.

For information on setting up other application server products with the IDE, see the *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide*.

The Sun ONE Application Server Nodes

This section identifies and describes the nodes that represent the Sun ONE Application Server in the Explorer.

The Sun ONE Application Server 7 Node

The Sun ONE Application 7 server node is for administering the application server.

Installing the IDE creates an application server domain. It also creates an admin server instance that administers the application server domain. In most cases you can do all of your work in the installed application server domain and admin server.

- You can deploy and execute your applications in the application server domain that is created by installing the IDE.
- You can administer this domain with the admin server instance that is created by installing the IDE.

The Server Registry shown in FIGURE 8-1 contains the application server domain and admin server that were created by installing the IDE. The admin server instance is represented by the node just below the Sun ONE Application Server 7 node that is labeled localhost:4848. You can view the application server domain name by opening the admin server node's property sheet.

If you need to, and if you have root or administrative privileges, you can use the Sun ONE Application Server 7 node to create additional application server domains and admin servers.

The Admin Server Node

Below the Sun ONE Application Server 7 node is an admin server node. In FIGURE 8-1, the admin server node is labeled localhost:4848. Admin server nodes represent instances of the Sun ONE Application Server's admin server. Each admin server instance administers an application server domain.

Application server domains and admin server instances are created by installing the IDE with the Sun ONE Application Server and administering the Sun ONE Application Server. The type of installation you work with determines how the application server domain and server instance are created. Some of the possibilities are described in the following list:

- If you install the IDE with root or administrator privileges, the installation will create an initial server domain and admin server instance. FIGURE 8-1 shows a server domain and admin server instance that were created by the installation process.
- If you have standard user privileges, a system administrator will create a server domain and an admin server instance for your use. You can administer your domain with the admin server node.

The hostname displayed with the admin server node is the name of the machine on which the application server is running. FIGURE 8-1 shows a stand-alone, single-user installation, and the application server domain is running on the local host. In a multi-user installation, the application server may be running on another machine.

The port number that is displayed with the admin server is the port number for communicating with the admin server. This port number is set when the application server domain and admin server instance are created. FIGURE 8-1 shows the default port number for a single-user installation on the Microsoft Windows platform.

Some of the tasks you perform with the admin server node are starting and stopping server instances in the server domain that is controlled by the admin server.

To start up the admin server and server instance:

1. **Right-click the Admin Server node and choose Start.**

   A progress monitor window opens. When the admin server starts, the progress monitor closes. The IDE displays a server instance node below the admin server node.

2. **Right-click server instance node and choose Status.**

   The Status dialog box opens. The Status field displays the status of the server instance, which is `Stopped`.

3. **Click Start Server.**

   The Status dialog box displays a message to tell you that it is starting the application server instance. When the server instance starts, the Status field displays `Running`.

4. **Click Close.**

   The server instance is ready for use.

   The contextual menu lists the other tasks you can perform with the admin server node.

The Server Instance Node

Below the admin server node is an application server instance node. In FIGURE 8-1 the application server instance node is labeled server1(localhost:81). The application server instance node represents a server instance.

When you deploy a module or an application, you deploy to a specific server instance. You must have a server instance node and the instance it represents must be running before you can deploy and execute.

FIGURE 8-1 shows a server instance that was created by installing the IDE. You can also create server instances by administering you application server domain.

The Registered Resource Nodes

Below the server instance node is a set of nodes that represent the named resources available to applications running in the server instance. FIGURE 8-1 shows nodes for the resources that are used in and created by the scenarios in this book:

- Under the nodes labeled Registered JDBC Connection Pools, Registered JDBC DataSources, and Registered Persistence Managers are nodes that represent the installed PointBase database named `sample`. These resources were preconfigured when the IDE was installed. In one scenario, the `CatalogData` EJB module is configured to use the PointBase `sample` database by typing in these resource names. For the procedure that configures the `CatalogData` EJB module, see "Specifying a Datasource for the Entity Enterprise Beans" on page 65.
- Under the node labeled Registered JMS Resources are nodes for queue and queue connection factory resources that were created in a scenario in this manual. For the procedures that create and register these resources, see "Setting up the Application Server" on page 88. For procedures that use these resource to configure an application, see "Programming the Web Module" on page 90 and "Programming the EJB Module" on page 96.

The Deployed Application Nodes

Underneath the registered resource nodes is a set of nodes that represent modules and applications deployed to the server instance. FIGURE 8-1 shows a node for an application named `CatalogApp`. The scenarios in this manual cover programming and deploying `CatalogApp`. For the procedure that deploys `CatalogApp`, see "Creating the J2EE Application" on page 75.

The Unregistered Resource Nodes

Below the deployed application nodes is a set of nodes that represent unregistered server resources. These nodes are subnodes of the admin server node, and they represent resources that are not yet registered with a server instance.

These nodes have menu commands for creating and registering new resources. A scenario in this book uses the Unregistered JMS Resources node to create a queue and queue connection factory. For the procedure that creates resources, see "Setting up the Application Server" on page 88.

The Default Server Nodes

These nodes identify the server instances that are currently designated as the default server instances. When you deploy an application, it is deployed to the default server instance unless you specify otherwise on the application's property sheet.

In FIGURE 8-1, the default nodes show that the default server for J2EE applications is Sun ONE application server.

To make a server instance the default server:

- **Right-click the server instance node and then choose Make Default.**

Server-specific Properties

The modules and applications that you work with have property sheets. You use the property sheets to identify the services your modules and applications need from the application server.

Many property sheets have server-specific tabs. The server-specific tabs list the properties defined for specific server products.

For example, FIGURE 8-2 shows the property sheet for the `CatalogData` EJB module. The Sun ONE AS tab is selected. This tab has a CMP resource property, and you use this property to identify the datasource for the CMP entity beans in the `CatalogData` module. Notice that the datasource named in FIGURE 8-2, `jdo/PointbasePM`, appears in FIGURE 8-1 as Registered Persistence Manager of the Sun ONE Application Server instance.

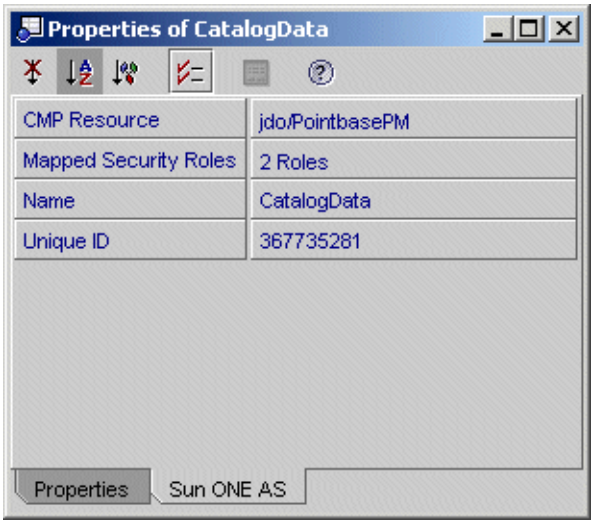

FIGURE 8-2 EJB Module's Sun ONE AS Tab

Using Server Instance Nodes to Deploy and Execute

This section outlines the procedures for deploying and executing a J2EE application from within the IDE.

To deploy and execute an application:

1. **Begin with an assembled J2EE application. Review the application for completeness of assembly.**

2. **Choose an application server instance.**

   The application node has an Application Server property. The initial setting of the Application Server property is `Default Application Server`. If you proceed with this setting, the IDE will deploy your application to the server instance that is currently specified as the default server for J2EE applications.

   You can also open the property editor for this property and choose a server instance by name. The property editor is a browser dialog that lets you review all server instances in the server registry and select one.

3. **Deploy and execute the application by right-clicking the application node and choosing the Execute command.**

   This will begin the deployment process. Monitor the process on the output window. When deployment is complete, the IDE will execute the application in the application server's environment. What you see depends on the application. For example, if the application contains a web module, the application server will start a web browser and open the application's welcome page.

4. **You can also deploy and execute in separate steps. Right-click the application and choose the Deploy command. When deployment is complete, execute the application yourself.**

   For example, if the application contains a web module, you can start a web browser outside the IDE and open the application's welcome page.

APPENDIX **A**

How the IDE Supports Deployment of J2EE Modules and Applications

This appendix briefly describes the server plugin. The server plugin is the IDE's mechanism for deploying and executing J2EE module and application. The server plugin provides the following IDE features:

- The server product nodes and their subnodes
- The server-specific property sheets
- The deploy and execute menu commands

This appendix focuses on the deploy and execute menu commands. It describes processing required to deploy an application and explains how the plugins perform this processing.

If you understand how the deployment facility works you can use it effectively. Procedures for using the deployment facility are included in a number of the scenarios in this book.

The Deployment Process

Deployment is the process of delivering the executable form of a module or an application to a J2EE application server. The executable form that is delivered to the server as an archive that contains compiled versions of the source files that make up the module or application. The compiled files are accompanied by a deployment descriptor that describes the contents and organization of the archive. The archived files are installed in directories that are managed by the application server.

When you execute a module or application, the application server executes the installed copy of the application in a process that is controlled by the application server. This process provides the necessary runtime environment.

To deploy successfully, the application source files must be compiled in a manner that is compatible with the specific application server product. The deployment descriptor must include all the information that is required by the specific application server product.

These needs are met by identifying a target application server for the product and compiling the source files and generating the deployment descriptor specifically for the target application server.)

The Server Plugin Concept

To enable the IDE to deploy to a variety of web and application servers, the concept of a server plug-in has been developed. A plugin is an IDE module that manages the interaction between the IDE and a specific server product. When you deploy an application, you choose the server to which it will be deployed. The IDE uses the appropriate plugin to process your Deploy command. This enables it to generate the appropriate commands for the server's deployment tool and include the appropriate server-specific deployment descriptor files in the files it passes to the server. This is illustrated in FIGURE A-1.

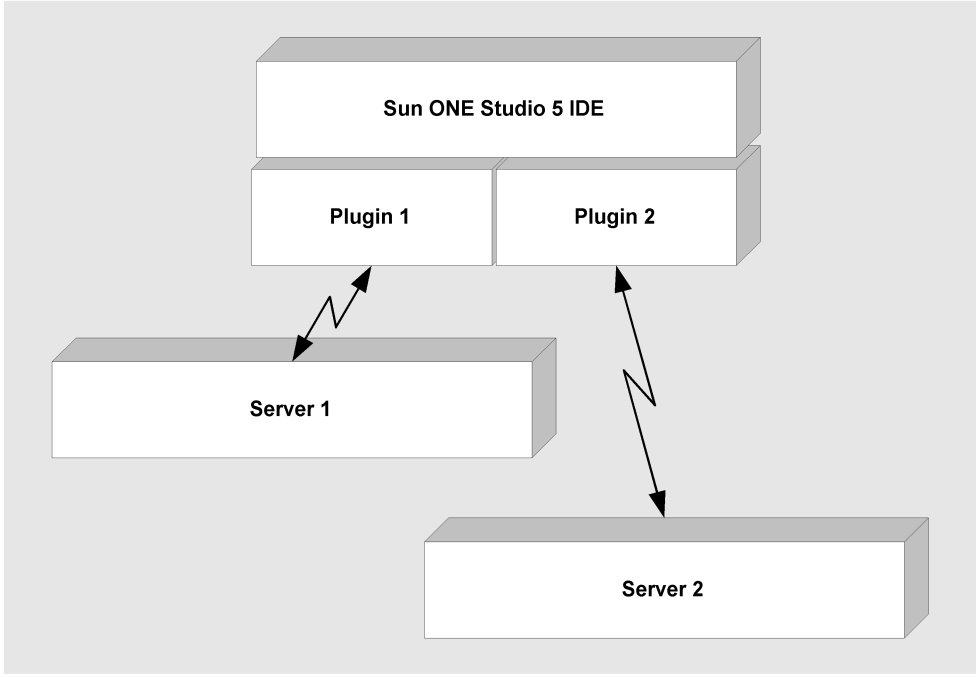

FIGURE A-1 Server Plugins Enable the IDE to Communicate With J2EE Runtime Environments

For the application developer who is deploying an application, the plugins provide:

- Visual representations of the plugins in the IDE's Explorer window. Each plugin is represented by a server product node. For more information on the appearance and use of server product nodes, see "The Server Product Nodes" on page 127.
- Visual representations of running server instances as subnodes of the server product nodes. You can choose any server instance represented in the Explorer window as the target for a deployment. For more information on the appearance and use of the server instance nodes, see "The Server Instance Node" on page 129.
- Server-specific tabs on component, module, and property sheets. These show the non-standard properties required by a server product, and prompt developers for the values required by a server product.
- A mechanism for processing Deploy commands specifically for the selected server. The details of this processing are covered in the next section.

The Deployment Process Using a Plugin

This section summarizes the processing performed by the plugin when you deploy and execute an application.

1. Assemble your application. Use property sheets to supply J2EE standard deployment descriptor elements and the non-standard elements required by the server.
2. After you assemble the application, specify a target server instance.
3. Choose the IDE's Deploy command to begin the deployment process.
4. The IDE identifies all of the files needed to create a WAR or EAR file for the application. This includes the J2EE components identified in the deployment descriptor, and any Java classes or static resources used by those files. The IDE identifies all file dependencies in the components.
5. The IDE identifies the server product to which the application is being deployed.
6. The plugin validates the files for the WAR or EAR file.
7. The IDE generates the WAR or EAR file for the application. This includes a J2EE deployment descriptor, separate files with server-specific deployment tags, and any stub or skeleton classes required for remote method invocations.
8. The plugin passes the WAR or EAR file to the server.

   Depending on the server product, the plugin may automatically clean up earlier deployments of the same application or attempt to resolve conflicts with applications already deployed to the server instance.
9. The server takes over, reads the deployment descriptors and the server-specific deployment files, and deploys the WAR or EAR file according to its own standards.

When this process is complete, the IDE will automatically start a web browser and open the application's welcome page. If you chose to deploy and execute separately, you can start a web browser and one of the application's web pages.

Deploying Components Other Than Web Modules and J2EE Applications

Web modules and J2EE applications are the only items that can actually be deployed to servers and executed. However, you may want to test smaller units of business logic that you are developing. The Sun ONE Studio 5 IDE makes it possible to deploy and execute smaller units of business logic by automatically generating modules and applications for the components you want to test. It can also generate test clients for some types of components. For more information on these features, see *Building Web Components* and *Building Enterprise JavaBeans Components*.

Index

E

H

I

J

L

M

N

P

Q

R

S

T

U

W

X