



Building Web Components

Sun™ ONE Studio 5 Programming Series

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

Part No. 817-2334-10
June 2003, Revision A

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

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Forte, Java, NetBeans, iPlanet, docs.sun.com, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à <http://www.sun.com/patents> et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats - Unis et dans les autres pays.

Ce produit est un document protégé par un copyright et distribué avec des licences qui est en restreignent l'utilisation, la copie, la distribution et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Forte, Java, NetBeans, iPlanet, docs.sun.com, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



Contents

Before You Read This Book	10
How This Book Is Organized	11
What Is Not in This Book	12
Useful References	12
Valuable Web Sites	12
Typographic Conventions	13
Related Documentation	14
Documentation Available Online	14
Online Help	15
Examples	16
Javadoc Documentation	16
Documentation in Accessible Formats	16
Contacting Sun Technical Support	16
Sun Welcomes Your Comments	17
1. Addressing the Challenges of Web Application Development	19
What Is a Web Application?	19
Challenges in Developing Web Applications	20
How Web Application Development is Different	21
Implications of Server-Centered Execution	21

How the IDE Helps	22
Full Web Component Support	22
Execution Support	23
Debugging and Monitoring Tools	23
Deployment Support	24
Open Runtime Environment Integration	25
Common Errors in Web Applications	25
Facing the Challenges of Web Application Debugging	26
Viewing Requests With the HTTP Monitor	26
Summary of IDE Features	27
2. The Structure of Web Modules	31
Web Servers	32
Servlet Containers and Web Components	32
Servlet Context	33
Web Modules	34
JSP Pages	34
JSP Page Life Cycle	34
Code Constructs in JSP Pages	36
HTTP Sessions	39
Scopes and Implicit Objects	40
Servlets	42
Servlet Life Cycle	42
Tag Libraries	44
JSP Standard Tag Library	45
Servlet Filters	46
Filter Life Cycle	47
Listeners	48

3. Design Patterns and Frameworks	51
Design Patterns	51
Front Controllers	52
Composite Views	56
View Creation Helpers	57
Model Objects	58
Frameworks	59
Struts	59
JavaServer Faces	60
4. Developing Your Own Web Application	61
Development Work Flow	61
Web Modules in the IDE	62
Creating a Web Module	63
Importing an Existing Web Module	64
Creating JSP Pages	65
Using page Directives	66
Creating and Invalidating Sessions	66
Modifying the JSP File	66
Working With JSP Includes	67
Creating a Composite View Template	69
Using Additional Classes or Beans	70
Creating Servlets	71
Declaring the Servlet in the Deployment Descriptor	72
Modifying the Servlet	73
Servlet-Generated HTTP Responses	74
Using the Servlet as a Front Controller	74
Using Additional Classes or Beans	75
Creating Filters	76

Declaring the Filter in the Deployment Descriptor	76
Processing HTTP Requests and Responses	77
Using Tag Libraries	77
Using Existing Tag Libraries	78
Tag Library Descriptors	79
Inserting Custom Actions From a Tag Library Into a JSP Page	79
Developing Your Own Tag Libraries	81
Testing Tag Libraries	87
Working With Databases	87
5. Running, Debugging, and Deploying Your Web Application	91
Running and Debugging Tasks	91
Configuring the Web Module Deployment Descriptor	92
Using Property Sheets to Edit the <code>web.xml</code> File	93
Registering Servlets and Filters	94
Registering Tag Libraries	94
Specifying the Default URI Within the <code>Taglib</code> Element	95
Using the IDE to Edit the <code>web.xml</code> File in the Source Editor	96
Using the Integrated Servers	96
Test Running Web Modules	96
Configuring the Server	97
Setting Up the Web Server Environment for Database Access	97
Executing a Single Web Module	98
Creating and Executing a Web Module Group	99
Executing on External Servers	100
Debugging Web Applications	101
Using the HTTP Monitor to Debug a Web Application	101
Source-Level Debugging	106
Packaging Your Web Applications	109

Building a WAR File From a Web Module	109
Specifying Options	109
Viewing Contents	110
Packaging and Deploying a Custom Tag Library	110
Packaging a Tag Library as a JAR File	111
Deploying a Tag Library Using the Tag Library Repository	111
Deploying a Tag Library by Adding a JAR File From the Filesystem	111
Deploying a Tag Library by Copying and Pasting a JAR File From Another Module or Filesystem	112
Including a Web Module Within a J2EE Application	112
Glossary	113
Index	121

Before You Begin

Building Web Components provides essential information for anyone involved in the creation of web applications with Java™ 2 Platform, Enterprise Edition (J2EE™) web components. It is part of the Sun™ ONE Studio 5 Programming series. This book focuses on web application development in the context of the J2EE and its supporting technologies. These technologies include Java servlets and JavaServer Pages™ (JSP™).

This book introduces web applications and provides suggestions for their structure. It describes the work flow of developing a web application. The book proposes design practices and provides pointers to suggested structures and frameworks for scalable, maintainable web applications. It places these concepts within the context of the Sun ONE Studio 5 integrated development environment (IDE) with discussions of the creation of JSP pages and servlets, coding, testing, debugging, and deployment.

In particular, this book describes how web applications typically use JSP pages, Java servlets, JSP tag libraries, and supporting classes and files. Web applications might use persistent data, for example, a database. They could be independent applications with features managed by a web container. Or, they might provide a user interface while depending on components in a J2EE Enterprise JavaBeans (EJB™) container for other services. Such services might include execution of business logic and access to persistent data.

See the release notes for a list of environments in which you can create the examples in this book. The release notes are available on this web page:

<http://forte.sun.com/ffj/documentation/index.html>

Screen shots vary slightly from one platform to another. Although almost all procedures use the interface of the IDE, 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
```

To translate for UNIX® platforms or Linux platforms, change the prompt and use forward slashes:

```
% cd MyWorkDir/MyPackage
```

Before You Read This Book

This book assumes you are a web application developer or a web application designer. A web application developer writes the application code. A web application designer specifies how users interact with an application, chooses interface components, and arranges them in a set of views. Unless otherwise stated, this book uses the term *web application* to refer to a *J2EE web application*. The web application developer might or might not be the same person as the web application designer. In either case, it is assumed you have a general knowledge of Java programming, JSP page programming, and HTML coding. Information in this book might also prove useful for any professionals who participate in the creation of applications based on web components. Such professionals might include technical writers, graphic artists, production and marketing specialists, and testers. This book presents the ways in which web application development work flow is facilitated by the use of the IDE. It provides a context in which to use this productivity tool.

The development of web applications differs markedly from that of traditional Java applications. It requires an understanding of several different technologies. Before starting, you should be familiar with the following subjects:

- HTML syntax
- XML syntax
- Java™ programming language
- Java Servlet syntax
- JavaServer Pages™ syntax
- HTTP protocol
- Web server concepts
- Security issues

This book enables you to use your current skills. It also provides you with references to help you become productive in the building of web applications with the IDE.

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>
- **JavaServer Pages™ Standard Tag Library 1.0 Specification**
<http://www.jcp.org/aboutJava/communityprocess/review/jsr052/index.html>

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>

The *Sun ONE Studio 5 Web Application Tutorial* can be downloaded from the Developer Resources web site. To gain access to the tutorial, choose Help, then Learning, then Examples, then Tutorials/Examples from the IDE's main menu.

The *J2EE Tutorial* describes the process of developing web applications. Visit the web site at:

<http://java.sun.com/j2ee/tutorial>

The *Java Web Services Tutorial* provides helpful background. It could prove useful as a reference while reading this book. Available at:

<http://java.sun.com/webservices/docs/1.0/tutorial/>

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

Building Web Components contains the following information:

Chapter 1 provides an introduction to web applications and the challenges they present to developers. It describes the ways that the IDE helps the developer face those challenges.

Chapter 2 discusses the structure of web applications. It includes an overview of the core J2EE technologies used in building the components of web applications.

Chapter 3 gives an overview of useful design patterns and frameworks for web applications.

Chapter 4 describes the process of developing your own web application using the IDE.

Chapter 5 provides details on the process of executing, debugging, and deploying your web application using the IDE.

The **Glossary** defines important words and phrases found in the book. Glossary terms appear in *italics* throughout the book.

What Is Not in This Book

This book is intended to provide sufficient information to get you started using the IDE as a productivity tool. However, it is not designed as a tutorial. The book is neither a comprehensive reference, nor does it supply all possible designs for a web application. It is not a visual design guide. Nor is the focus on developing the J2EE web tier. The book does not delve into how to develop EJB components. See “Useful References” on page 12 for suggested readings. See “Before You Read This Book” on page 10 for references to tutorials regarding the development of web applications.

Useful References

This section provides the names of references you might find helpful when reading this book.

This volume covers topics related to web application design and implementation:

Alur, Deepak, Crupi, John, and Malks, Dan, *Core J2EE Patterns*, Sun Microsystems Press, Prentice Hall, 2001. This excellent book on web application architecture and models provides solutions, including J2EE-based answers, to problems in context. It reflects the collective experience of Java architects and the Sun Java Center.

Valuable Web Sites

Here is a selection of web sites dealing with web application technologies:

- **The Source for Java Technology** provides a wealth of information on web component technologies. Topics include products and APIs, access to the Developer Connection, documentation and training, and online support. It also supplies community discussion, industry news, marketplace solutions, and case studies. Available at <http://java.sun.com>
- **JSP Insider** is a JavaServer Page web site offering design information, articles, code, links to other web site, news stories, and book reviews. Available at <http://www.jspinsider.com>
- **The JSP Resource Index** is a place to find tutorials, scripts, and even job listings. Available at <http://www.JSPin.com>
- **The Jakarta Project** supplies commercial-quality server solutions based on the Java platform that are developed in an open and cooperative fashion. Jakarta is the overall project name for many subprojects, including the Jakarta tag libraries and the Tomcat server. Available at <http://jakarta.apache.org>
- **TheServerSide.com** is a J2EE news source and developer community. Available at <http://www.theserverside.com>.

Typographic Conventions

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

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`SM 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 the 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. In particular, you should explore the JSP/Servlet help set and the help sets for the supported server software.

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>

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 <i>sIstudio-install-directory/examples</i>
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 (817-2334-10) of your document in the subject line of your email.

Addressing the Challenges of Web Application Development

This chapter introduces web applications and discusses how they differ from standalone desktop applications you might have developed previously. It then explores the ways the Sun ONE Studio 5 IDE helps you to build these applications.

What Is a Web Application?

A *web application* is a collection of web components. It provides features to end users through an interface typically presented in a *web browser*. Examples of web applications might include an electronic shopping mall or auction site. A web application is based on a client-server model. In this model, the *client* is the *web browser*, and the *web server* is the feature set that runs remotely.

In the simplest form of a web application, the browser is the client. The browser sends requests to a web server. When the web server receives the request, it passes the request to a web application that is running inside a *servlet container*. The JSPs and servlets that are in the web application process the request and generate a response. The server sends the response back to the browser.

FIGURE 1-1 shows the relationship of the requests and responses to filters, servlets, and JSP pages and of session data to servlets and JSP pages in a sample web application. This sample application uses a front controller. Front controllers are explained in Chapter 3.

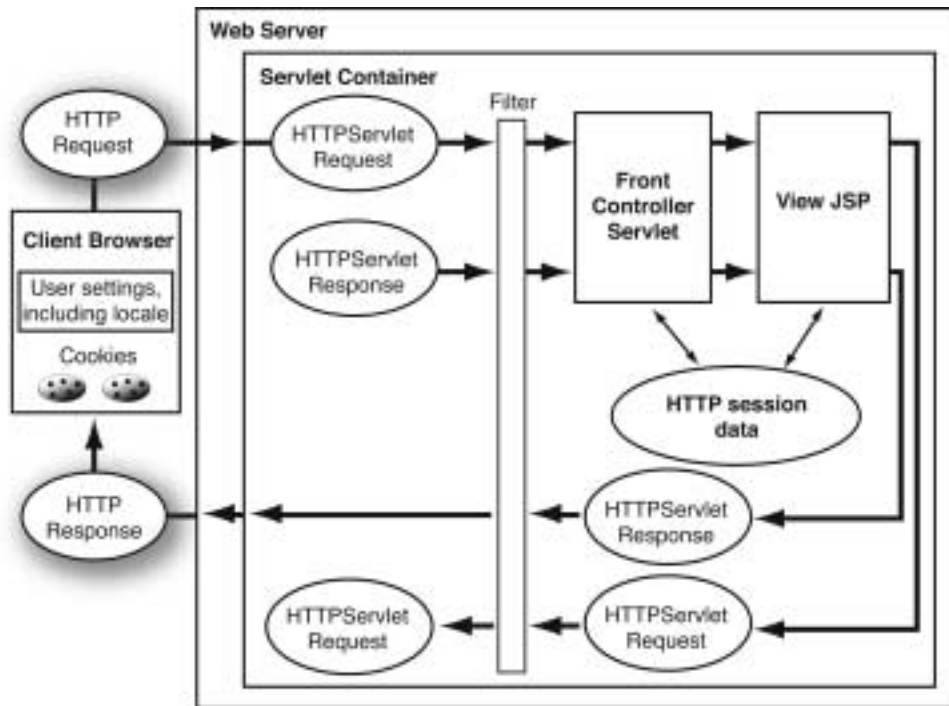


FIGURE 1-1 Requests and Responses in a Web Application

Web applications consist of a varied set of interrelated web *components*. This set includes JSP pages, servlets, and tag libraries that work together. Resources utilized within the web application must be coordinated using a deployment descriptor file. This file contains the meta-information that describes the web application. The *servlet container* translates JSP files into servlets in order to run them in the web server. Chapter 2 provides more detailed explanations of the components in this illustration.

Challenges in Developing Web Applications

Two key characteristics distinguish web applications from standalone applications:

- Web components do not interact directly with one another. The servlet container and the client browser mediate the links between components.

- Data representation, data flow, and data processing are distributed among the client browser, servlet container, and individual web components. The mechanisms enabling web components to maintain links to one another and to share data are described throughout this book

As shown in FIGURE 1-1, the web server is the center of the action. Information storage and database access all take place in the server.

When you implement a project as a web application, it is easy to deploy and maintain. Since the application resides on a server, users do not need to install any software on their own computers. Moreover, upgrades are performed in one place by the server administrator instead of having to perform upgrades on all the clients.

In addition, the architecture of a web application can make the creation of reusable components easier. They are more loosely coupled than components in a standalone desktop application.

How Web Application Development is Different

In a stand-alone application, the components can interact directly with one another. For instance, data objects can each have a specific type and are passed from method to method. In addition, data representation, data flow, and data processing are all managed by the application itself. For example, in a wizard, each pane is part of the application with direct access to application data. Developers focus on the data and features within their applications. Hence, they do not need to understand the runtime environment, that is, the virtual machine.

On the other hand, in a web application, web components are more encapsulated than their standalone counterparts. Web components communicate with one another through the servlet container. Data is passed as strings. Hence, information can travel safely among the cooperative processes of the client browser, web server, and servlet container. When creating a web application, developers need to understand variations among execution environments. Testing must take place on multiple browsers and, sometimes, on multiple web servers. Actual file locations and structures vary, as do steps to deploy the web application. Deployment onto servers and execution are typically time-consuming tasks.

Implications of Server-Centered Execution

Everything in a web application executes in the server rather than in the client browser. Therefore, web applications face associated challenges and rely on the HTTP protocol for support.

As shown in FIGURE 1-1, data representation, data flow, and data processing are distributed among the browser, servlet container, and individual web components. See “Common Errors in Web Applications” on page 25 for how this situation affects compile time, runtime, and debugging tasks.

In distributed web server situations, the servlet container must be dispersed among the web servers running on multiple machines. Furthermore, session information must be shared to process requests from a single client browser encountering different server instances.

How the IDE Helps

To improve productivity and deliver the benefits of web application implementation, the IDE addresses the web application developer’s unique challenges. The full web component support, execution support, debugging and monitoring tools, deployment support, and open runtime environment integration provided by the IDE are discussed in this section.

Full Web Component Support

The IDE provides the following features to support the full set of web components:

- **Editors for JSP pages, servlets, tag libraries, and tag library handlers.** Support includes source editors with syntax coloring and checking, code completion, and compilation. See “Modifying the JSP File” on page 66, “Modifying the Servlet” on page 73, and “Inserting Custom Actions From a Tag Library Into a JSP Page” on page 79 for details.
- **Automated and manual editing of the deployment descriptor file.** The deployment descriptor is also referred to as the `web.xml` file. See “Using Property Sheets to Edit the `web.xml` File” on page 93 and “Using the IDE to Edit the `web.xml` File in the Source Editor” on page 96 for more information.
- **Development within a web module structure.** Every task you perform is done within a context delineated by the *Servlet 2.3 Specification*. See “Web Modules in the IDE” on page 62 for details.
- **Views of generated servlet code for a JSP page.** While debugging the servlet, you can view the current location in the JSP page code and servlet code simultaneously. See “Viewing Both JSP and Servlet Files During Debugging” on page 108 for more information.
- **Easy creation of tag libraries.** See “Creating a Tag Library and Tags” on page 81 for details.

- **Support for incorporating existing tag libraries in your web application.** See “Using Existing Tag Libraries” on page 78 for more information.

Execution Support

The IDE supports web application execution in the following ways:

- One-click actions for building, deploying, and running a web application
- Test interaction using different client browsers
- Test execution on different servers

For details, see “Test Running Web Modules” on page 96 and “Executing on External Servers” on page 100.

Debugging and Monitoring Tools

A web application’s data management is distributed among its components. Many of the bugs captured at compile time in standalone applications must be tracked at runtime in web applications. This situation leads to an increased focus on runtime testing during the debugging of web applications. The IDE provides source-level debugging for JSP pages, servlets, and helper classes. HTTP requests received and generated by the web server are captured. These features aid in the debugging and monitoring of data flow in your web application. For more information, see “Debugging Web Applications” on page 101.

The way that a web application shares data among its components differs from a traditional application. A web application’s components (JSPs, servlets, and so forth) cannot communicate directly with each other. Instead, a component shares data with other components through attributes (name/value pairs) that are stored with the `ServletContext`, `ServletRequest`, `HTTPSession`, and `PageContext` objects.

As shown in FIGURE 1-1, data moves as an HTTP request from the client browser to the web application. Information then moves back through an HTTP response. When the request enters the servlet container from the client, a response is created. Then both the request and the response are available until the request processing is complete.

An *HTTP request* is a message created in the client browser. Headers with user settings, including locale information, and cookies from the client browser are contained in the request. The web server routes the request to the servlet container. In the case of the sample web application shown in FIGURE 1-1, the request passes through the filter, a front controller servlet, the JSP file, and the filter again. Additional information can be obtained and possibly modified in all these locations. The HTTP request finishes when the JSP file and filter processing is complete. For more information on front controllers, see “Front Controllers” on page 52.

An *HTTP response* is a message created in the servlet container. The message contains attributes, including cookies, headers, and output, which eventually go to the client browser. Response processing also occurs in a front controller servlet, in a JSP file, and in a filter. Information from the request can be accessed and possibly modified, and information can be added to the response. When the request is finished, the web server routes the response from the servlet container through the web server. Its final destination is the client browser. For details, see “Servlet-Generated HTTP Responses” on page 74.

Requests from a particular client can be associated with one another. This connection is achieved using an *HTTP session*. An HTTP session is a conversation spanning multiple requests between a client browser and a web server. Each session has its own set of session data. The JSP file and the servlet have access to the session data as well as information in the request and response. The session data resides in the session scope. That is, a component can access a session’s data only if the component is processing a request that is associated with the session. For details, see “HTTP Sessions” on page 39.

Servlet filters are web components that can be used to inspect or modify the `HttpServletRequest` and `HttpServletResponse` objects before and after the requested servlet or JSP file has been processed. For more on this use of filters, see “Processing HTTP Requests and Responses” on page 77.

In traditional applications, source-level debugging can be used to follow interfaces among objects. This technique is also helpful in web application development. However, a need for a debugging tool that can operate at the component interface level is indicated as well. Here HTTP messages are passed between the client and the web server to isolate the component with an error. At this point, you can typically find the error through code inspection. For details, see “Common Errors in Web Applications” on page 25. For details on the IDE’s tool for tracking HTTP requests, see “Starting the HTTP Monitor” on page 103.

Deployment Support

Because web applications consist of multiple resources, the various web components must be associated and registered in a deployment descriptor. As described previously, much of the process of coordinating web application resources is automated in the IDE.

The IDE provides deployment support with the following features:

- **Deployment descriptor editing.** Web application resources need to be associated and registered using a property-based editor or through direct editing of the XML file. For details, see “Configuring the Web Module Deployment Descriptor” on page 92.

- **WAR file packaging creation.** Resources can be packaged for deployment on any J2EE-compliant server. For more information, see “Packaging Your Web Applications” on page 109.

Open Runtime Environment Integration

The IDE provides support for executing your web application on multiple servers. It offers deployment direct from the source within the IDE using an API-based mechanism to integrate with third-party servers. For details, see “Test Running Web Modules” on page 96 and “Executing on External Servers” on page 100.

Common Errors in Web Applications

With web applications, problems arise from web components not receiving the proper data. Difficulties also occur when a request does not get the server into a particular state. Common sources of bugs in web applications include coding errors and deployment errors.

The following are common coding errors:

- Misspelled names of links
- Misspelled or missing parameter names or values
- Incorrect cookie values or domains
- Misspelling of session attributes that have not been initialized
- Expected errors not sent by the client
- Forgotten or misspelled initialization parameters

The following are common deployment errors:

- Components that have not been deployed
- Libraries that have not been deployed
- Servlet URL mappings that have been misspelled or omitted from the deployment descriptor

Web application data must be set through attributes (name/value pairs) that are stored with the ServletContext, ServletRequest, HttpSession, and PageContext objects. Hence, errors are frequently introduced during the flow of data from one component to another. Source-level debugging is useful for following programming logic within a component of a web application. However, it does not provide a way to view data passing among components. For example, one component might set the value for a “title” attribute and another component might try to get the value for a “Title” attribute. This type of error is not exposed at compile time.

If a bug causes a runtime exception, then you can easily determine where you need to set a breakpoint to see what is going wrong. However, if your problem is a blank page or missing data, identifying which component is causing the problem is not always easy. The problem might be that the front controller did not receive the data or the problem might be that the front controller is not passing the data correctly. Another reason might be that the session is not supplying some necessary data. Unless you can determine which component is the source of the problem, source-level debugging becomes a hit and miss approach. Tracking requests and looking at session attributes can help pinpoint the origin of the bug.

To determine error sources in web applications, you can use the IDE's debugging tools to do the following actions:

- Track data flow among components
- Get the server or session into a particular state
- Step through generated output of a web resource, line by line

Facing the Challenges of Web Application Debugging

Debugging web applications involves identifying the components sharing the data, debugging the code, and working with the deployment process. The IDE provides the following features to facilitate your efforts:

- **The HTTP Monitor.** Records information regarding requests to your web application and the data flow among its components. For details, see “Starting the HTTP Monitor” on page 103.
- **The Source-Code Debugger.** Enables you to debug the components. If the specified component is a JSP page, you can view the JSP file and its generated servlet code side by side. This perspective provides improved visibility into the processing that is taking place in the server. For more, see “Source-Level Debugging” on page 106.
- **A Common Interface for Controlling and Deploying the Application to Different Web Servers.** Facilitates the identification of deployment issues from within the IDE. See “Executing on External Servers” on page 100.

Viewing Requests With the HTTP Monitor

The built-in *HTTP Monitor* provides a way to view current requests and to record these requests for later playback. Current requests persist until the current IDE session ends. Saved requests persist across sessions of the IDE until you explicitly delete them.

You can use the HTTP Monitor tool to identify where in the sequence of data flow an error has recurred. Often the problem is not in processing of a JSP file but in specifying an input parameter.

The Monitor displays all messages sent between the client browser and the web server, along with relevant context information. This feature helps identify inconsistencies. It also enables you to find information that has been incorrectly passed from one component to another. For details, see “Viewing Monitor Data Records” on page 103.

The Monitor records both incoming and outgoing cookies on a request making it easier for you to view that information.

Summary of IDE Features

Table 1-1 summarizes the features that distinguish the IDE’s support for key web development needs, as described earlier in this chapter.

TABLE 1-1 How the IDE Addresses Developer Needs

Developer Need	Feature and Description	For more information, see
Quick start	Creation Wizards. Providing templates for creation of JSP pages, servlets, filters, listeners, web modules, web module groups, tag libraries, and HTML files.	“Creating a Web Module” on page 63 “Creating JSP Pages” on page 65, “Creating Servlets” on page 71, “Creating Filters” on page 76 “Developing Your Own Tag Libraries” on page 81
Full web component support	JSP Editor. Editing JSP source code is similar to editing HTML. The IDE provides support for both HTML and JSP tags.	“Modifying the JSP File” on page 66
Efficient web application development	Code Completion. Accessing available completion for JSP files, servlets, and tag libraries.	“Modifying the JSP File” on page 66 and “Modifying the Servlet” on page 73.
Code reuse and separation of developer and page designer	Tag Library Editing. Providing tag library descriptors and customizers as efficient ways to develop tags.	“Using Existing Tag Libraries” on page 78 “Developing Your Own Tag Libraries” on page 81.

TABLE 1-1 How the IDE Addresses Developer Needs (*Continued*)

Developer Need	Feature and Description	For more information, see
Mechanism for viewing data flow	HTTP Monitor. Collecting information about the execution of JSP files and servlets in the servlet engine. Streamlining the work involved in debugging JSP files and servlets.	“Starting the HTTP Monitor” on page 103
Comparing a JSP file to its generated servlet	JSP and Servlet Debugging. Displaying JSP files and their generated servlets side by side	“Viewing Both JSP and Servlet Files During Debugging” on page 108
Support for newest servers and most current specifications	J2EE 1.3 Compliance. Accessing the J2EE service and communication APIs, which provide for security, concurrence, transaction, and deployment	“Servlet Containers and Web Components” on page 32

TABLE 1-1 How the IDE Addresses Developer Needs (*Continued*)

Developer Need	Feature and Description	For more information, see
Common interface for deploying to and testing on different servers	Deployment Descriptor Editing. Browsing and configuring the elements of the deployment descriptor in property sheets for the <code>web.xml</code> file. Or, opening the <code>web.xml</code> file in the Source Editor and editing it manually.	“Configuring the Web Module Deployment Descriptor” on page 92
Extensibility	Integration with Enterprise Edition. Creating a J2EE application from an existing EJB module or web module. Or, creating the application from an Explorer filesystem or package node	“Including a Web Module Within a J2EE Application” on page 112
Easy access to web servers	Web server plug-ins. Configuring the Sun ONE Application Server 7 plugin and the Tomcat plugin to integrate with web servers that are installed on your computer	“Using the Integrated Servers” on page 96

The Structure of Web Modules

The chapter describes the structure of web modules, including:

- Servlet containers
- JSP pages
- Servlets
- Servlet filters
- Tag libraries
- Listeners

This chapter discusses concepts that are fundamental only to the servlet container and its web components. It also describes supporting classes and files not directly managed by the servlet container. These classes and files are logically part of the web module and are deployed together with the web components.

A web module runs inside a servlet container. A servlet container, in turn, is contained within a web server. A web module contains presentation, controller, and model elements. The presentation element, sometimes called the view element, is the physical page that users see and interact with. It can consist of either a JSP page or an HTML file. The controller element, for example, a servlet, controls what users see and how they interact. The model element contains data that is used by the presentation and controller elements. The data might be included in Java libraries or other resources. These resources might include GIF images, HTML files, and so forth.

Web Servers

A *web server* provides the mechanisms for clients on the Internet, intranet, or extranet to access a repository of web resources. These resources might consist of HTML pages, CGI (Common Gateway Interface) scripts, images, and so forth. The web server mechanisms include:

- Support for HTTP and other protocols
- Execution of server-side programs, such as servlets or CGI scripts, which perform certain functions
- Support for servlet containers, which characteristically depend on web servers for HTTP message handling
- Hosting of one or more servlet containers from the same vendor

An *HTTP request* is a message created in the client browser that includes attributes and cookies from the client browser. The web server routes the request to the servlet container. Request processing can occur in a servlet, in a JSP file, and in a servlet filter. The HTTP request ends when this processing is complete.

An *HTTP response* is a message that is generated in the servlet container and includes cookies, headers, and output that eventually go to the client browser. Response processing can also occur in a servlet, in a JSP file, and in a servlet filter. The web server routes the response from the servlet container to the client browser.

Servlet Containers and Web Components

Web servers can provide servlet containers. *Servlet containers* offer runtime services to support the execution of the web components of a web module. These services include:

- Life-cycle management
- Network service, by which requests and responses are sent
- Decoding of requests and formatting of responses
- Interpretation and processing of JSP pages into servlets
- Deployment
- Access to the J2EE service and communication APIs, which provide for security, concurrence, transaction, and deployment

Servlet containers forward client requests from a web server to web components in the module. They also forward the client-bound responses from the web components back to the web server. Servlet containers typically run in a web server process as a web server plug-in or in a J2EE application server process.

Servlet containers also provide the network services over which requests and responses are sent, requests decoded, and responses formatted. All servlet containers support *HTTP* (Hypertext Transfer Protocol) as a protocol for requests and responses. They also might provide for additional request-response protocols such as *HTTPS* (Hypertext Transfer Protocol Secure Sockets).

A *distributed* servlet container can run a web module that is tagged as distributable. The server might execute the web module across multiple Java virtual machines that are running on the same host or on different hosts, and the module must gracefully deal with being run in such a distributed manner. In this situation, the scope of the objects in the web module is extended. A certain amount of overhead is involved in maintaining the synchronization of the common session information among multiple servers. This overhead includes performance penalties and storage considerations. For more information, see “HTTP Sessions” on page 39.

Web components are server-side J2EE components. They are managed by and communicate directly with a servlet container. They are capable of receiving HTTP requests through the servlet container, processing them, and returning HTTP responses. The J2EE platform defines two web component types: servlets and JSP pages.

Servlet Context

The *servlet context* for a web module is an object containing the servlet’s view of the web module in which it is running. It defines a set of methods that a servlet uses to communicate with its servlet container, for instance to:

- Dispatch results
- Log events
- Obtain URL references to resources
- Set and store attributes other servlets in the context can use

A web module is represented at runtime by an object implementing the `ServletContext` interface. The servlet context provides web components with access to resources available within the web module. There is one servlet context per web module per Java virtual machine.

A `ServletContext` instance is unique within a nondistributed web module. The instance is also shared by all web components within the web module. This object is implicitly available in JSP pages as the `application` instance variable. This variable is always available. It does not need to be declared.

A `ServletContext` instance (as well as the web module it represents) is rooted at a specific path within a web server. It could, for example, be rooted at `http://www.myStore.com/productList`. In this case, all requests starting with the context path of `/productList` would be routed to the `ServletContext` instance.

Web Modules

A *web module* is unit that consists of one or more web resources and that is deployable as a J2EE web application, where a web resource can be a web component or a static web content file, such as an image. A web module might be used to implement a web-based product catalog application. Such a program might contain:

- JSP pages for displaying products to the user
- Servlets for controlling navigation through the catalog
- Beans for getting information from a database

Web modules are self contained. Only the contents of a single root directory are typically required to deploy a web module to a web server.

For more information, see “Web Modules in the IDE” on page 62.

JSP Pages

A *JSP page* describes the client presentation, for example, a page displayed in a web browser, that enables your web module to interact with end users. A JSP page is translated to a servlet class within the servlet container. The JSP page describes how to process an HTTP request, and it generates an HTTP response. The JSP page’s HTML-like syntax lets it focus on presentation and document issues rather than features provided by Java code. For more information on servlets, see “Servlets” on page 42. For details on support for JSP pages in the IDE, see “Creating JSP Pages” on page 65.

JSP Page Life Cycle

A JSP page is processed by its runtime environment, that is, the servlet container. The servlet container manages the creation, routing, and removal of requests and responses. Furthermore, it orchestrates the processing of requests and responses by

activating the appropriate web components. The JSP page performs processing on an HTTP request and generates an HTTP response. The processes involved in this phase are JSP page translation, instantiation, request processing, and destruction.

Translation and Instantiation

JSP page translation refers to the process by which the servlet container converts a JSP file into a servlet class. The details of this process are specific to the servlet container. In the web server or servlet container, the JSP file is converted to a Java servlet source file. It is then compiled to a class file.

The servlet container translates a JSP file the first time it receives a request for it. On subsequent requests for the same JSP page, the servlet container typically bypasses this phase. Translation could occur if the date on the JSP implementation class is older than the date on the JSP file. Hence, JSP pages can be redeployed without restarting the web server. The translation process is managed by the servlet container.

After a JSP page has been translated to a servlet, it is instantiated by a call to its `jspInit` method. The `jspInit` method is typically used to prepare resources required by the JSP page.

Request Processing

In the simplest case, the JSP page receives a client request from the servlet container. It then processes the request according to its programmed logic and sends a response to the container. By default, each request executes in its own thread. Request processing can involve other components such as servlets, filters, or other JSP files that forward requests from the client.

Destruction

Servlet containers typically provide a way to limit how long a JSP instance can persist without receiving a request. After the user-specified limit, the servlet container can reclaim resources by destroying a JSP instance. Before doing so, it calls the instance's `jspDestroy` method, which corresponds to the `jspDestroy` method of the JSP file. The `jspDestroy` method is used to close resources that are no longer needed.

Code Constructs in JSP Pages

A JSP page can contain template data and elements. *Template data* consists of non-JSP constructs, such as HTML and XML code, passed through to the HTTP response word for word. Template data is generally used to provide static content and to format dynamic data. Because HTML is passed through literally, coding presentation content is very natural for a web page designer. *Elements* are constructs, recognized by the servlet container, that provide dynamic capabilities.

JSP elements are grouped into three categories: directive elements, action elements, and scripting elements.

Directive Elements

Directive elements provide global declarative information about a JSP page that is unrelated to any particular request. Directives are processed at translation time.

Directives are placed between `<%@` and `%>` symbols. For example, the following `page` directive imports the `java.util` package and associates the JSP page with the current error page.

```
<%@ page import="java.util.*" errorPage="showError.jsp" %>
```

TABLE 2-1 describes the directives defined in the JSP specification:

TABLE 2-1 JSP Directives

JSP Directive	Description
<code>page</code>	Imports classes, sets session participation, and selects an error page
<code>taglib</code>	Identifies a tag library so that its tags can be used within a JSP file
<code>include</code>	Includes another file within a JSP file

Action Elements

Action elements are XML-style tags that provide a means of working with Java objects without writing Java code. For example, you can use actions to locate and instantiate objects, and to get and set an object's properties. Actions are processed at *request time*, that is, when the request is received by the servlet container. Some actions write output to the HTTP response.

Standard actions are implemented by the servlet container.

TABLE 2-2 describes JSP standard actions defined by the JSP specification.

TABLE 2-2 JSP Standard Actions

Standard Action	Description
forward	Immediately sends the processing of the request to another resource, including JSP pages, servlets, HTML pages, and so forth
include	Subsumes another file within this JSP file. The name of the included file can be computed at request time.
useBean	Identifies a bean that can be accessed from the JSP file
getProperty	Gets the value of a property from a bean associated with the JSP file through the useBean action
setProperty	Sets the value of a property in a bean associated with the JSP file through the useBean action
plugin	Enables the Java plug-in to be loaded in the client browser. If necessary, it executes an embedded applet or JavaBeans component.

The JSP specification also supports the development of *custom actions* to provide features not available through standard actions. Custom actions are implemented by creating or importing tag libraries. See “Tag Libraries” on page 44 for more information.

Because actions use XML syntax, they provide web page designers with a familiar paradigm for working with dynamic data. Web page designers might not code actions themselves. However, they need to understand enough to work in a file containing actions. Web page designers might have to provide HTML template data for actions that produce output to a web page.

Scripting Elements

Scripting elements enable you to embed Java code within a JSP file. These elements can then be used for programming logic and for writing output to the HTTP response. Scripting elements are executed on the web server. The response page sent to the client displays only the result of the scripting element code. By contrast, JavaScript is routed through the server and returned to the client for processing.

Three syntactically distinct types of scripting elements are described in this section: declarations, expressions, and scriptlets.

Declarations help you declare and initialize variables, instantiate objects, and declare methods. Declarations are processed at translation time and do not write output to the HTTP response. Declarations are placed between `<%!` and `%>` symbols. The following example declares and initializes two `String` variables:

```
<%!  
    String name = null;  
    String title = null;  
%>
```

Expression elements enable you to enter any valid and complete Java expression. The servlet container converts an expression element to a `String` at request time. The resulting `String` is then written to the HTTP response. Expressions are placed between `<%=` and `%>` symbols.

The following example inserts a piece of dynamic data into an HTML string.

```
<p>Hail the <%= title %>!
```

Scriptlets are useful to manipulate data for viewing purposes. They enable you to enter any piece of valid Java code. However, they are not recommended for performing business logic because they can be difficult to maintain. It is better to encapsulate business logic within reusable Java classes like beans or tag handlers.

You might start out with scriptlets for creating prototypes or for the testing of new code. After confirming that the feature set meets your requirements, move the code into a bean or tag library. For complicated view logic, see “View Creation Helpers” on page 57. For details about creating a tag library, see “Developing Your Own Tag Libraries” on page 81.

You should not use scriptlets in production web modules. This recommendation is especially important if a developer is maintaining the code and a web page designer is in charge of the visual aspects of a web module.

Variables and methods declared in a declaration element are available to scriptlets in the same JSP page. A Java statement can begin in one scriptlet and end in another, interspersed, for example, with HTML code. Scriptlets are processed at request time and write output to the HTTP response, if you code them to do so.

The following scriptlet example shows a Java `if` statement that spans two scriptlets. The HTML code is included in the HTTP response only if the `if` statement evaluates to `true`. Note that scriptlets are placed between `<%` and `%>` symbols.

```
<% if (name.equals("Elvis Presley")){  
%>  
<p>Let's hear it for Elvis!  
<%     title = "King";  
}  
%>
```

HTTP Sessions

An *HTTP session* is a Servlet API mechanism that associates the many requests representing a conversation between a client browser and a server.

When a user requests a JSP page, a new session is automatically created if it does not already exist. Subsequent requests to pages within the web module are usually associated with the session. Requests are not associated with the session when the page directive `session` attribute is set to `false`.

```
<%@page contentType="text/html" session="false" %>
```

The session ends when a time-out occurs or when the web module explicitly invalidates it. The `session timeout` value is set in the web module's deployment descriptor file. A JSP page or servlet class can invalidate a session using the `invalidate` method. For details on setting the session time-out value, see "Configuring the Web Module Deployment Descriptor" on page 92. For more information on the using the `invalidate` method, see "Creating and Invalidating Sessions" on page 66.

The typical JSP page is associated with a session. However, you might also include dynamic pages that are not specific to a particular client. Such pages can be generated once and shared among many sessions. The use of JSP pages not associated with a session can reduce your web module's requirements for system resources.

In a distributed environment, information for a particular session might only exist in the web server that initiated the session. If requests are routed to different web servers depending on overall load, the session information must be replicated. In this situation, the web servers need to access the shared session information and must keep in sync. The web server can manage the shared session information. However, this practice adds to the total load on the system, consuming time and resources.

The HTTP Monitor indicates whether a page is associated with a session or not. It also provides other useful information about HTTP requests. For details on the IDE's HTTP Monitor, see "Viewing Monitor Data Records" on page 103.

Scopes and Implicit Objects

When instantiating an object in a JSP file, you most likely want to make it available to other objects in your application. You might want to ensure that all objects in your application can access the object. On the other hand, you might want to restrict its availability to some subset of these objects. For example, you might want to make the object available only to other objects associated with the current user's HTTP session. The JSP specification defines a number of scopes in which you can place a reference to an object.

Scopes enable you to control the availability of an object. You can place a reference to the object in any of these scopes. You typically only put an object in a single scope. The object is then available to all subsets of the chosen scope. In other words, page scope objects are available to objects in the request, session, and application scopes.

The concept of a scope in a web module is different from its use in traditional, standalone applications. In a web module, *scope* refers to an object's availability to a module's various components. These are the page, request, session, and application scopes. In standalone applications, scope refers to a variable or object's availability within blocks of code.

At runtime, these scopes are implemented as Java objects, as described in TABLE 2-3.

TABLE 2-3 Scopes in JSP Pages

Scope	Description	Object Type
page	Represents the current JSP page. This object is available only to JSP elements in the current page or in pages included by an <code>include</code> directive. This object is not available to pages included by an <code>include</code> action. The directive is executed at page translation time, and the included pages are concatenated into the same JSP implementation class.	<code>javax.servlet.jsp.PageContext</code>
request	Represents the current HTTP request. This object is available only to JSP pages and servlets executing in the current HTTP request. For example, if one JSP forwards to another using a <code>forward</code> action, both pages access the same <code>ServletRequest</code> object.	<code>javax.servlet.ServletRequest</code>
session	Represents the current user's HTTP session. This object is available only to JSP pages and servlets executing in requests associated with the current user's HTTP session.	<code>javax.servlet.http.HttpSession</code>
application	Represents the runtime web module. This object is available to all JSP pages and servlets in the web module.	<code>javax.servlet.ServletContext</code>

You can locate or make a *bean* available within one of these scopes with a `useBean` action. In this action, you supply a `scope` attribute in order to specify the availability of the bean instance, for example:

```
<jsp:useBean id="myCart" scope="session" class="Cart">
```

Scopes and the objects they represent are implicitly available to the scripting elements of a page. They use the scripting variables that the page automatically instantiates. By default, JSP pages have access to the session scope. However, if a page is not participating in a session, it cannot use the session scope. Furthermore, it cannot reference the `session` implicit variable. When a page is not participating in a session, the `page` directive `session` attribute is set to `false`. Some parts of a web module do not require session data. An example of such data is background information about a site that does not require a user to log in. If the user remains only in those sections, then the overhead of creating a user session can be avoided.

You can use the IDE's code completion feature to show the methods available to objects of different scopes. For details, see "Modifying the JSP File" on page 66.

You can set and use scope objects from within servlets, tag handlers, and scriptlets. To set objects in scopes, use the `setAttribute` method on the relevant scope object. To retrieve objects in scopes, use the `getAttribute` method on the relevant scope object. See TABLE 2-3 regarding the available scopes.

Servlets

Servlets are Java classes that execute within a servlet container. They are used to:

- Extend the capabilities of web servers and web-enabled application servers
- Generate dynamic content
- Interact with web clients using a request-response paradigm

For details on support for developing servlets in the IDE, see "Creating Servlets" on page 71.

Servlets are typically used as front controllers and dispatchers to control navigation through a web module. They are also used to control application flow. Servlets enable and disable access to certain web resources, depending on the particular state being tracked.

For more information on the use of the Front Controller design pattern, see "Front Controllers" on page 52.

Servlet Life Cycle

The servlet life cycle defines how the servlet is loaded, instantiated, and initialized. It also describes how it handles requests from clients, and how it is taken out of service.

Loading and Instantiation

Servlets execute within containers that provide the network services. These services include sending requests and responses, decoding requests, and formatting responses. All servlet containers support HTTP as a protocol for requests and responses. They might also provide for additional request-response protocols such as HTTPS.

The servlet container loads and instantiates servlets. An option enables loading and instantiation to occur once the servlet container is started. Servlets designated “load on startup” are loaded when the servlet container starts up. For more information, see “Load on Startup” on page 72. When the loading is complete, the container instantiates the `Servlet` class for use.

Initialization

Before the servlet can handle requests from clients, the servlet container initializes it. Initialization enables the servlet to:

- Read persistent configuration data
- Initialize two kinds of resources:
 - Those you only want one instance of
 - Those that are time-consuming to initialize, for instance, database connections
- Perform any application-specific start-up activities

Request Handling

Once the servlet has been initialized, the servlet container can route requests to it. The servlet processes the requests and builds appropriate responses. The servlet passes these objects as parameters of the `service` method of the `HTTPServlet` interface. The `service` method is called by the servlet container. It enables the servlet to respond to a request only after the servlet’s `init` method has been initialized.

Servlets typically run inside multi-threaded servlet containers that can handle multiple requests concurrently. As a developer, make sure to synchronize access to any shared resource such as files and network connections. Also make sure to synchronize access to the servlet’s class and instance variables. If necessary, the servlet container can also serialize requests to a servlet.

A new servlet instance is created for each concurrent request. Therefore, if many requests arrive simultaneously, many new threads are created. To avoid overloading the server, the servlet container can limit the maximum number of servlets. For information about setting this limit see one of the following:

- For Sun ONE Application Server 7, see the *Sun ONE Application Server Administrator's Configuration File Reference* at <http://docs.sun.com>.
- For Tomcat, see the Configuration reference at <http://jakarta.apache.org/tomcat/tomcat-4.0-doc/config>

Destruction

When a servlet container decides to remove a servlet from service, it calls the `destroy` method of the `Servlet` interface. This method enables the servlet to release its resources and save any persistent states.

Once the `destroy` method is called, the container cannot route other requests to that servlet instance. Furthermore, any currently running threads of the `service` method are permitted to complete their execution. Upon the completion of the `destroy` method, the container releases the servlet instance for garbage collection.

Unlike other Java classes, the servlet's life span might be hours, days or even weeks. Hence, you need to manage the use and creation of resources, such as database connections, so that they are available only when needed.

Tag Libraries

A *tag library* is a collection of custom actions. As shown in FIGURE 2-1, a tag library consists of a set of tag handlers and a tag library descriptor file. Each custom action, or tag, is implemented as a tag handler bean that contains its features. The tag library descriptor (TLD) is an XML document that maps each tag in the library to its associated tag handler. The TLD describes parameters and scripting variables associated with the tags in the tag library.

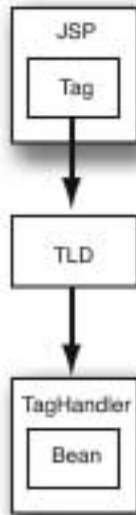


FIGURE 2-1 The Structure of a Tag Library

A tag library is typically packaged as a JAR file. It is made available to a JSP file through a `taglib` directive. The IDE supports the use of existing tag libraries such as *JSTL* (Java Server Pages™ Standard Tag Library) and other third-party libraries. It also provides for the creation of your own tag libraries. For details on tag library support in the IDE, see “Using Tag Libraries” on page 77.

JSP Standard Tag Library

You can use tags from the JSP Standard Tag Library (JSTL) to extend the set of actions available for use in your web module. The use of tags based on a standard helps to increase the portability of components within a web module. It also reduces the need for Java code in scriptlets, which can create maintenance difficulties in JSP files.

For example, instead of iterating over lists using different iteration tags from numerous vendors, you can use JSTL tags. They work the same way in all web environments. This standardization means you learn a single tag and use it on multiple JSP containers. Furthermore, the specification enables containers to recognize standard tags and optimize their implementations.

The JSTL introduces the concept of an expression language to simplify page development. It includes an experimental version of a language for testing purposes. JSTL also provides a framework for integrating existing custom tags with JSTL tags.

For a tutorial and description of JSTL tags, see *Java Server Pages Standard Tag Library* at the following web site:

<http://java.sun.com/webservices/docs/1.1/tutorial/doc/JSTL.html>

To download the JSTL and the associated specification, visit the following web site:

<http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html>

You can use the tag library editing and management features of the IDE to facilitate the usage of JSTL tags. For details, see “Using Existing Tag Libraries” on page 78.

Servlet Filters

Servlet filters, also called *filters* in this book, are Java classes that modify requests to and responses from servlets. Filters can be used to perform many functions, such as:

- **Authentication.** Assurance that a user can only access certain web resources
- **Logging and auditing web application users.** Keeping track of each time a user accesses a web resource and recording the access in a log file
- **Localization.** Choice of the appropriate resource for the user’s locale
- **Data compression.** Compression or decompression of data on its way to or from a servlet
- **Style translations of XML content.** Translation of XML content before viewing a web resource

Filters are defined by version 2.3 of the *Java Servlet Specification*. When a servlet context receives a request, it calls all the filters that are associated with the request before passing the request to the processing resource. Filters are not called if the request is redispached internally, for example, because the request is forwarded. This behavior makes filters useful in some gate-keeping roles. For instance, you could use a filter for:

- **Logging.** If a property of incoming requests interests you, the filter can extract the requests and write them to a file.
- **Changing or adding to data associated with the request.** You can replace the original request and response with wrappers. They enable you to change the incoming request before it is processed by a JSP page or a servlet.
- **Handling preprocessing required by a group of resources.** For example, you could ensure that a user profile is loaded into a session.

By contrast, you might require logging or preprocessing in case a request is forwarded from a servlet or JSP page using the `RequestDispatcher` API. In this situation, you need to implement the action as a servlet instead.

When the servlet context receives a request, it determines whether any filters match the path to the resource. If one or more filters are found, the servlet context constructs a `FilterChain` object. The chain consists of all matching filters in the order they are declared in the deployment descriptor. The servlet context calls the `doFilter()` method on the first filter. Each filter must call the next filter in the chain. Each filter gives control to the next filter and so forth. When the last filter in the chain calls the `doFilter` method, the request is processed by a JSP page or a servlet. The request could also be processed in a static content file such as an image being served. Once this process is complete, control is returned to the last filter. When the last filter's `doFilter` method is completed, control is returned to the next-to-the-last filter and so forth.

An example of a filter in action is the HTTP Monitor in the IDE. It uses a filter to gather data about the request and the servlet context before and after the other web module resources process the request. The filter is also responsible for handling replay requests. The filter accomplishes this task by replacing the incoming request with a wrapper. Then the filter populates the wrapper with data from the original request. Hence, the HTTP Monitor filter needs to be the first one in the chain, so that it is called before any application-defined filters. Filters are usually declared in the deployment descriptor of individual web modules.

For information on the support for servlet filters in the IDE, see “Creating Filters” on page 76.

Filter Life Cycle

The filter life cycle delineates how the filter is loaded, instantiated, and initialized. It also describes how the filter handles requests from clients and how it is taken out of service.

Loading and Instantiation

Before a servlet container accesses a web resource, it locates the list of filters to be applied to the web resource. The servlet container ensures that it has instantiated a filter of the appropriate class for each filter on the list. It also calls the `FilterConfig` method for each filter.

Initialization

When the container receives a request, it calls the `doFilter` method for the first filter in the list of filters in the chain. The container then passes in the servlet's requests and responses and a reference to the filter chain it is to use. The filter chain is defined in the deployment descriptor.

The `doFilter` method for a filter might examine the headers of the request. In addition, the method could modify request or response headers or alter data by wrapping the request or response object.

The filter then calls the next entity in the filter chain. The entity could be a filter. A request works its way through the filter chain to the last filter. The last filter then calls the associated web resource, for instance, a servlet or a JSP page.

Destruction

Before a filter can be removed from service by the container, the container calls the `destroy` method on the filter. It also releases any other resources and performs cleanup operations.

Listeners

Application event listeners are new with the *Servlet 2.3 Specification*. They are classes that implement one or more of the servlet event listener interfaces. Application event listeners are instantiated and registered in the web module at the time the web module is deployed.

Listener classes provide a way to track sessions within a web module. It is often useful to determine why a session has become invalid. Either the container timed out the session or the module called the `invalidate` method. You can make this distinction using listeners and `HttpSession` API methods.

Servlet event listeners support event notifications for state changes in:

- **Servlet context objects.** Useful for managing resources at the virtual machine level for the module
- **HTTP session objects.** Useful for managing state. Also helpful for handling resources associated with a series of requests from the same client related to the module.

Use multiple listeners to monitor changes occurring within the life cycle. They are also useful to track attributes of servlet contexts and HTTP session objects.

- **Servlet context events** include:
 - **Life cycle.** Servlet context has just been created and can service its first request, or it is about to be shut down.
 - **Changes to attributes.** Servlet context attributes have been added, removed, or replaced.

- **HTTP Session events** include:
 - **Life cycle.** An HTTP session has been created or invalidated, or has timed out.
 - **Changes to attributes.** Attributes have been added, removed, or replaced on an HTTP session.

Design Patterns and Frameworks

This chapter presents some terminology and general concepts surrounding design patterns that are useful in the architecture of web applications. It outlines several commonly used patterns and introduces some frameworks that simplify the process of developing web applications.

Design Patterns

Design patterns provide architectural solutions to common software design problems. These patterns have emerged over time through the experience and insights of developers. This section introduces some basic patterns that might prove helpful in the context of developing web applications.

You can also find information on these patterns by visiting the following web site, which is part of the Java Developer Connection:

<http://developer.java.sun.com/developer/restricted/patterns/J2EERPatternsAtAGlance.html>

The patterns listed below are of particular relevance to the building of web applications. More details are provided in the sections that follow:

- **Front Controller.** Coordinates handling of incoming requests. See “Front Controllers” on page 52 for more information.
 - **Dispatcher.** A subpattern of the Front Controller pattern, describing how to control which view the user sees. See “Dispatchers” on page 53 for details.
 - **View Helper.** A subpattern, in this case, to the Front Controller pattern, encapsulating the processing functions, or business rules, such as data access or business logic. See “Helpers” on page 55 for more.
- **Composite View.** (also called a *template*) Creates an aggregate view from subcomponents. See “Composite Views” on page 56 for additional information.

A full treatment of the use of design patterns in web applications can be found in *Core J2EE Patterns* by Deepak, Crupi, and Malks. See “Useful References” on page 12 for more on this book.

Front Controllers

Front controllers are responsible for routing incoming user requests. In addition, they can enforce navigation in web applications. When users are in sections of a web application where they can browse freely, the front controller simply relays the request to the appropriate page. For example, in an e-commerce application, the customer browses through a product catalog. In controlled sections in which users must follow a specific path through the application, the front controller can validate the incoming requests and route them accordingly. For example, a customer wants to buy the items in a shopping cart. That customer is required to follow a particular route to complete the purchase successfully.

A front controller provides a single entry point through which requests for several resources in a web application pass. One front controller might handle all requests to the application. Several front controllers might handle requests for portions of the application. Typically implemented as servlets, front controllers are frequently used for the following tasks:

- Controlling page flow and navigation
- Accessing and managing model data
- Handling business processing
- Accessing data that is relevant to GUI presentation, for example a user profile

Front controllers can reduce duplication of code in JSP pages, especially in cases where several resources require the same processing. Examples might include ensuring that the user’s profile has been found, obtaining the data corresponding to a product ID, and so forth.

You can maintain and control a web application more effectively if you funnel all client requests through a front controller. Functions such as view selection, security, and template creation can be centralized in this design pattern. The front controller applies these functions consistently across all pages or views. Consequently, when the behavior of these functions needs to change, only the front controller and its helper classes need to be changed. They constitute a relatively small portion of the application.

In the two-tier form of a web application, shown in FIGURE 3-1, the recommended approach is for the front controller to deal with user requests. The front controller also determines which presentation element is to be shown to users and which data is to be used for the chosen presentation. This strategy contrasts to the traditional approach in which each user request must be mapped to a separate view.

Note that front controllers do not have to route requests directly to views. You can chain them so that, for instance, one front controller accesses user profile information. Then it could forward that profile to another front controller.

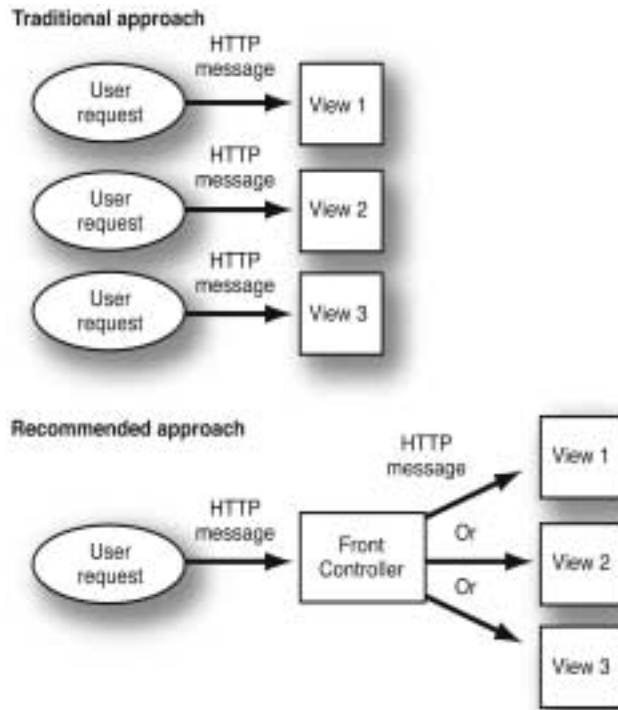


FIGURE 3-1 Determining the User View With a Front Controller

For information on creating a servlet as a front controller using the IDE, see “Using the Servlet as a Front Controller” on page 74.

Dispatchers

Typically, the front controller coordinates user navigation, using the *Dispatcher* subpattern for this purpose. As shown in FIGURE 3-2, the front controller processes a request. Perhaps the user might want to check out items in a shopping cart of an e-commerce application.

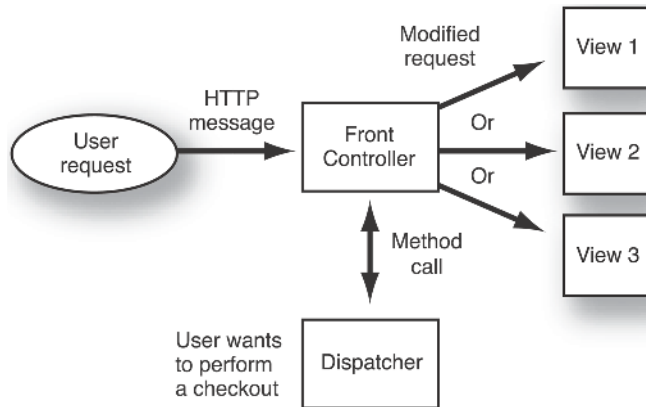


FIGURE 3-2 Dispatching as a Function of a Front Controller

Dispatcher code could be contained within the front controller servlet, or in a separate class. In practice, the dispatcher instructs the front controller where to forward the request. In the Front Controller design pattern, the dispatcher encapsulates the behavior that controls which views the user sees.

View Mappers

When web resources differ based on the type of client, you can use a *View Mapper* to assist the dispatcher mechanism. Such clients could include a web browser, personal desktop assistant, or cell phone. For instance, you might be developing a web application that retrieves information about waves and tides. In this situation, your users might want to view this data from desktop personal computers or cell phones. Instead of dispatching to a single JSP page, your web application might use the View Mapper to send a different JSP page, depending on the type of client.

1. For example, when your web application receives incoming requests, it routes them to a front controller servlet.
2. The front controller retrieves the appropriate data using a Helper bean.
3. It then determines the appropriate view type based on the client within the View Mapper.
4. Based on input from the View Mapper, the dispatcher returns the view information to the front controller.
5. The application subsequently forwards the request to the specific view intended for the user's client, as shown in FIGURE 3-3.

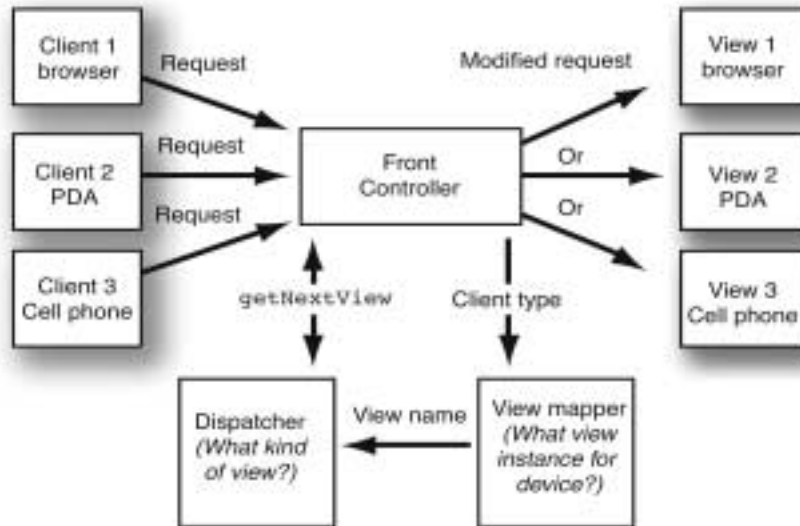


FIGURE 3-3 Using View Mappers

In your wave and tide application, you might not initially know whether you wanted to display the information on a PDA or on a phone. In this case, the View Mapper would enable you to create alternative objects or families of objects.

Use View Mappers not only to redirect information to different devices, but to different locales or different views.

Helpers

The front controller servlet can easily become quite large and unwieldy. Therefore, use Helper classes to break out specific features and make the application easier to build and maintain. Here are some tasks that can be encapsulated as Helper classes:

- Retrieval of content from a file, another web site, or even a web service
- Validation of user-entered information
- If the front controller needs to delegate processing of business logic, it can use Helpers for this purpose, as shown in FIGURE 3-4.
- Data processing

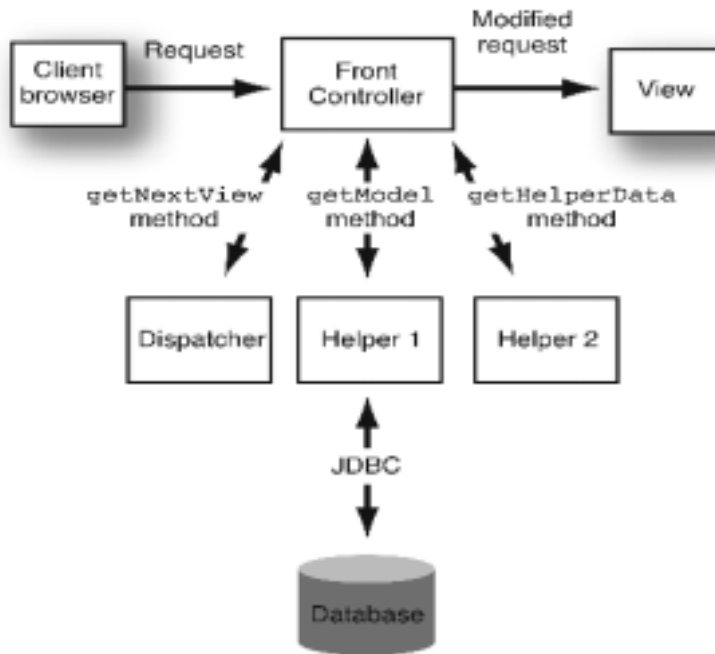


FIGURE 3-4 Delegating Processing of Business Logic With Helpers

Helpers that do the processing and retrieval of data for the presentation layer are called View Helpers. These types of helpers make the maintenance of applications easier because the helpers separate business logic from presentation logic.

You can implement Helpers as regular Java classes. See “Using Additional Classes or Beans” on page 70 for details.

Composite Views

A *Composite View* is a design pattern that creates an aggregate view from component views. Component views might include dynamic, modular portions of the page. This design pattern pertains to web application design when you are creating a view from numerous subviews. Complex web pages frequently consist of content derived from various resources. The layout of the page is managed independently of the content of its subviews. For instance, a view might have subviews like Navigation, Search, Feature Story, and Headline. An included view is a subview that is one portion of a greater whole. The included view might, in turn, consist of subviews, as shown in FIGURE 3-5.

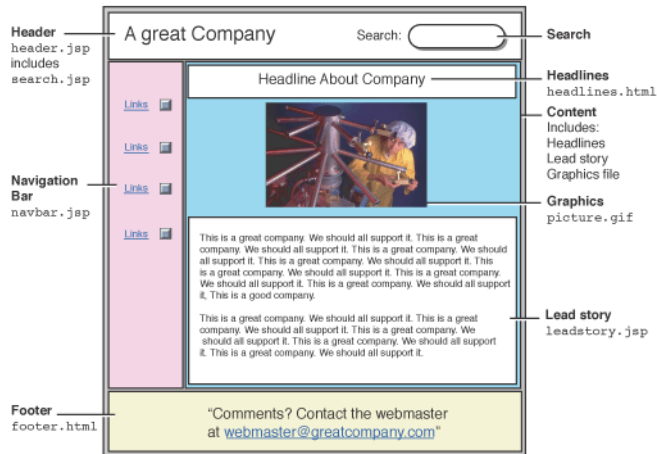


FIGURE 3-5 Managing Content Independently From Layout With a Composite View

When creating a composite view, you can include static content and dynamic content. Static content might consist of an HTML file. Dynamic content might be something like a JSP page. You can also include content at JSP translation time and runtime.

For information on using the IDE to implement a Composite View pattern, see “Creating a Composite View Template” on page 69.

View Creation Helpers

Typically, web pages need to be reused and maintained over time. Use *View Creation Helper* beans when you need to display data as it is received. Examples of such information might be tables or sets of links. A View Creation Helper can be any bean or Java class. However, since they are specifically geared toward presentation in JSP pages, View Creation Helpers are typically tag handler classes.

The View Creation Helper class provides a way to avoid placing Java code related to specific presentation features directly in the JSP file or front controller servlet. For instance, your web application might contain a catalog search that results in certain display results. In this situation, encapsulate the behavior into JSP tags, as shown in FIGURE 3-6:

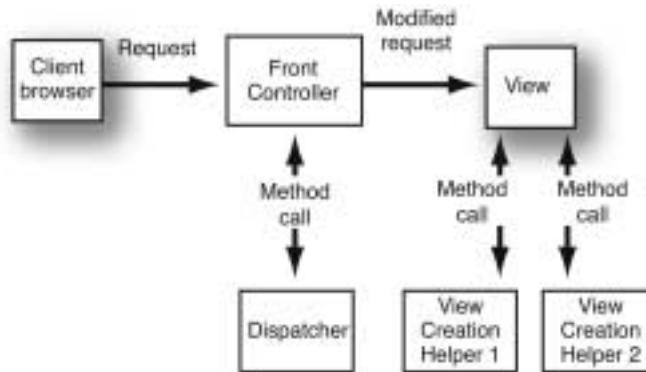


FIGURE 3-6 Using View Creation Helpers

Similarly, your web application might require that logic to format data within its views. View Creation Helper beans can be obtained and used in the same manner as any other beans from within a JSP file. See “Using Additional Classes or Beans” on page 70 for more information on using the IDE with beans.

Model Objects

Model objects are Java objects that encapsulate application data inside a web application. For instance, in a shop cart e-commerce application, a model object might be an abstraction for a user profile. Examples of the data would include the user’s name, e-mail address, phone number, and so forth.

Typically, you store this type of information in a database, and the application retrieves the data by issuing a network request. Because network requests are expensive, the application should minimize the number of requests for the same data. Hence, you must design session data carefully. For additional information on accessing databases with JDBC, see *Using Java DataBase Connectivity*, a volume in the Sun ONE Studio 5 Programming series.

You can pass the model object to the JSP file from a front controller servlet in two ways:

- Through a request attribute
- By placement in the web application’s session data

If it makes sense for the application to store product data for a term longer than a single request, then this information can be placed in the application’s session.

Typically, all requests from a single user go to a single server where the session information is stored. In a load-balancing system, a single user's requests might be routed to different servers. Hence, session information for a particular user must be shared among servers. This situation can result in slower performance if the web application must synchronize access to the session data.

If the application is to be deployed on a load-balancing system, distributing session data is inefficient. In this case, you might want to store the data as hidden parameters in the page that is sent back to the user. The data is then sent back to the server with any subsequent requests that are sent from that page.

Frameworks

Frameworks are sets of design patterns, APIs, and runtime implementations intended to simplify the design and coding process for building new applications. Examples of frameworks are Struts, the Sun ONE Application Framework, and JavaServer Faces, described in the subsequent sections.

Framework designers factor out common functions from existing applications and implement them using appropriate design patterns. A framework enables application developers to concentrate on the unique feature set required by their applications.

Web application frameworks typically provide support for functions such as:

- Navigating among the application's pages
- Building forms that accept input from the end user
- Creating views to display information to the end user
- Providing access to processing and data with JavaBeans and Enterprise JavaBeans
- Supplying database access to incorporate information directly from database systems
- Using directory services such as JNDI (Java Naming and Directory Interface) and LDAP (Lightweight Directory Access Protocol)
- Providing secure and non-secure access through authentication mechanisms

Struts

Struts is an open-source framework from the Jakarta Project. It is designed for building web applications with the Java Servlet API and JSP technology. The Struts package provides an integrated set of reusable components. The set includes a controller servlet, JSP custom tag libraries, and utility classes. The components are

for creating user interfaces that can be applied to any web-based connection. Struts facilitates the standardization of workflow and the achievement of simplicity and reusability. For more on the Struts framework, see <http://jakarta.apache.org/struts/>

JavaServer Faces

JavaServer Faces proposes to define a standard set of JSP tags and Java classes. This set aims to simplify the building of Java web application graphical user interfaces (GUIs). JavaServer Faces defines complex HTML forms and other common GUI elements. Furthermore, it enables tools and third-party component vendors to focus on a single component framework for JSP pages and servlets. It intends to bridge the gap between conventional GUI toolkit developers and web-based GUI developers. It hopes to provide familiar APIs for GUI components, component states, and for rendering and input processing. Comprehensive support for internationalization and basic input validation is proposed. This support should ensure that developers include internationalization and input validation in their first releases. For more information on JavaServer Faces, see <http://www.jcp.org/jsr/detail/127.jsp>

Developing Your Own Web Application

This chapter provides an overview of how to program web modules using the IDE. This high-level view ties together the tasks involved in creating your application. It then provides some details on individual programming tasks.

See Chapter 5 for a description of the process of configuring, executing, debugging, and deploying a web application.

It is recommended that you work through the *Sun ONE Studio 5 Web Application Tutorial* before building your own web application with the IDE.

Development Work Flow

This section provides an overview of the work flow typically involved in developing a web module using the IDE. When developing a new web application, it often makes sense to proceed in an iterative fashion. The list below describes the major development tasks. Each task refers to a section later in the chapter that provides more detailed information. The IDE's online help for the JSP/Servlet module also provides information on these tasks.

To develop a web module using the IDE:

1. Create the web module's document root node in the Explorer window. For details, see "Creating a Web Module" on page 63.
2. Create the web components you plan to use. This process might include:
 - Creating JSP pages for forms and other dynamic web pages presented to end users. For more information, see "Creating JSP Pages" on page 65.
 - Creating the servlets, beans, filters, and utility classes required for your web module. You can use servlets to control the flow of your web module or to gain access to external resources. You can use filters to authenticate users

attempting to access servlets or JSP pages within your web application. See “Creating Servlets” on page 71 and “Creating Filters” on page 76 for more information.

The servlets, filters, beans, and utility classes go in the `WEB-INF/classes` directory of the web module. However, if the classes are packaged as JAR files, they go in the `WEB-INF/lib` directory.

The IDE supplies templates for all these objects in the New wizard. Templates are also provided for simple and advanced filters and four kinds of listeners. Each other object has a single template. See “Creating a New File” in the Core IDE online help for details.

- Importing the servlets, beans, and other classes required for your web module. See “Importing an Existing Web Module” on page 64 for details.
- Importing existing tag libraries, such as JSTL. See “Using Existing Tag Libraries” on page 78 for more information.
- Developing your own custom tag libraries for any other encapsulated features your JSP pages require

The recommended process is to develop your tag library in place within the web module in which you want to use it. Then when you are finished developing the tags, move the tag library source files to a separate filesystem for maintenance. At this point, package the library as a JAR file and place it in the `WEB-INF/lib` directory of the web module that contains the dependent JSP pages. See “Developing Your Own Tag Libraries” on page 81 for details.

3. Configure the web module and its components by editing the deployment descriptor file. See “Configuring the Web Module Deployment Descriptor” on page 92 for details.

Web Modules in the IDE

A web module is a J2EE *deployment* construct. The *Java Servlet 2.3 Specification* and *JSP 1.2 Specification* require that JSP pages be executed inside a web module. When you develop web applications using the IDE, it creates the necessary web module structure for you. By enforcing the web module structure, the IDE ensures that web modules can be packaged for delivery. The IDE also ensures that the web module contains the deployment descriptor information (the `web.xml` file) that is required for deployment on servlet containers.

Note – To execute or debug JSP pages and servlets in the IDE, you need to put them into a web module. Both JSP pages and servlets must be executed from within a web module. This behavior differs from some earlier versions of the IDE.

A J2EE web module corresponds to a “web application” as defined in the *Java Servlet Specification version 2.3*. In the IDE, a construct called a *web module group* can be used to deploy several web modules together. See “Creating a Web Module Group” and “Executing Web Modules” in the JSP/Servlet online help for more information.

A *web module* is the smallest deployable and usable unit of web resources in a J2EE application. It corresponds to the *servlet context*, as defined in the J2EE specification. Web modules are typically packaged and deployed as web archive (WAR) files. However, depending on the web server you use, web modules might not have to be packaged in order to be deployed. The format of a WAR file is identical to that of a JAR file. However, the contents and use of WAR files differ from JAR files, so WAR file names use a `.war` extension.

Web modules use a hierarchical structure for storing their resources. This structure is represented at development time as a filesystem mounted at the web module’s root directory with the following files and folders:


- JSP pages, HTML files, image files, such as GIF, and so forth that are directly accessible by end users
- The `WEB-INF` folder and its contents as follows:
 - The `classes` folder containing Java class files, including servlets, filters, and listeners
 - The `lib` folder containing JAR files, including tag libraries, JDBC (Java Database Connectivity) drivers, and other Java class libraries. These files provide many of the features for the web application.
 - JSP pages not directly accessible by end users, but that can be displayed using a servlet or filter class
 - The `web.xml` file, which is the web module’s deployment descriptor
 - Tag library descriptor files

Creating a Web Module

The first step in creating a web module from the IDE is to create a document root node for the module. Use the New wizard to create a web module in the designated filesystem. Upon completion of the wizard steps, the web module’s document root directory appears as a node in the Explorer. When you expand the node, the web module structure is reflected in the node contents:

- The `WEB-INF` node, which contains:

- The `classes` directory
- The `lib` directory
- The `web.xml` file, called the deployment descriptor file in this book

The web module directory structure is treated as an object within the IDE's Explorer window. It is displayed within a filesystem mounted at the web module's document root directory. For instance, a web module object type has attributes you can set in its Properties window and a set of pertinent commands available in its contextual menu. The web module object is represented by the icon for the WEB-INF node:  Like any other object type in the Explorer, a web module document root node can be created from a template.

Note – You mount a web module in the Explorer exactly as you would mount any other filesystem. See “Mounting a Filesystem” in the Core IDE Help Set for information on mounting filesystems. However, you must mount the web module at its root directory, which is the directory containing the `WEB-INF` directory. If you mount a directory *containing* a web module, rather than the web module itself, the web module is not properly recognized. In this situation, when the web module's document root is a subdirectory of a mounted filesystem, you cannot perform some operations usually associated with a web module. These operations might include the execution or deployment of your web module or its components.

Importing an Existing Web Module

The IDE enables you to continue development on web modules that have been created externally. To import an existing web module, you can use one of two methods, depending on how the web module is delivered:

- For a web module in a root directory, mount the root directory of the web module in the Filesystem Explorer.
- For a web module in a WAR file, mount the directory containing the WAR file. Then use the Unpack as Web Module action on the desired WAR file node to create a web module directory.

Indicate the location of the desired web module in the Mount Filesystem dialog box. It is also possible for you to mount a CVS filesystem containing a web module.

For details, see “Mounting an Existing Web Module” in the JSP/Servlet online help.

To be mounted in the IDE, your web module must conform to the structured directory format or the web archive format (WAR). The formats are described in the *Java Servlet 2.3 specification*.

Before executing an imported WAR file, you must first unpack and mount it as a web module. Unpack and mount a WAR file from the Filesystems tab of the Explorer. Mount the directory containing the relevant WAR file, if you haven't already done so. Right-click the icon for the desired WAR file. Then choose Unpack as Web Module from the contextual menu. In the Unpack WAR Folder dialog box, specify the directory where you want the unpacked files to be stored. Click Unpack Here. The WAR file is unpacked in the specified directory. The directory is mounted and appears as a web module in the Filesystems tab of the Explorer. If the WAR file contains no Java source files, then any servlets and JavaBeans components are not editable once the WAR file is unpacked.

For details, see “Unpacking and Mounting a WAR File” in the JSP/Servlet online help.

Creating JSP Pages

JSP pages are used in web applications to present information to end users and to enable data from end users to flow back to the server. An example would be data presentation and modification through the use of forms.

JSP files are created and managed in the IDE in a similar way to other file types. To execute a JSP page, you must place it within a properly mounted web module.

You can create a JSP page in one of two ways:

- Use the New wizard. For details, see “Creating a JSP or Servlet Source File” in the JSP/Servlet online help.
- Generate it based upon a Dreamweaver template. For more information on this technique, see “Working With Dreamweaver Files” in the JSP/Servlet online help.

You can create JSP pages in the root directory of your web module or within a subdirectory of this root directory.

JSP pages that are placed in the `WEB-INF` directory or any of its subdirectories are not directly accessible from a client browser. However, they are accessible as resources from a servlet. This feature is often used in conjunction with a Front Controller design pattern. It is used to control access to JSP pages that need to be displayed in a specific order. It is also used to control access based on security constraints. An example of JSP pages requiring controlled access might be pages in the middle of a checkout procedure. For more on the Front Controller design pattern, see “Front Controllers” on page 52 and “Using the Servlet as a Front Controller” on page 74.

You can work with `.dwt` files, Dreamweaver version 3.0 and earlier. In addition to generating a JSP page from a `.dwt` template, the IDE enables you to open and edit `.dwt` files in the Source Editor. You can also configure the IDE to open `.dwt` templates in an editor of your choice

Using page Directives

You can use `page` directives to specify global declarative information about a JSP page that is unrelated to any particular request. For example, use directives to:

- Set participation of a page in a session
- Import classes, for instance, packages, into a page
- Select an error page

For details on `page` directive syntax, see “Directive Elements” on page 36.

Creating and Invalidating Sessions

If a session does not already exist, executing the JSP page creates one. Any additional session-related pages continue to use that session until it is invalidated. Invalidation can occur either through a time-out or an explicit call. You can use the call below within a JSP scriptlet:

```
session.invalidate();
```

You can also invalidate the session from within a servlet using the following code:

```
request.getSession().invalidate();
```

For more information on HTTP sessions, see “HTTP Sessions” on page 39.

Modifying the JSP File

The IDE offers support for JSP syntax, including code completion for JSP tags. For example, you can press Ctrl-Spacebar to show the code completion box. For details on code completion, see “Completing an HTML Tag” in the Core IDE online help.

Editing JSP source code is similar to editing HTML. Hence, the IDE provides support for both HTML and JSP tags. For details, see “Editing a JSP or Servlet Source File” in the JSP/Servlet online help.

When executing a web module with modified JSP pages, the IDE saves and recompiles the JSP files as they are accessed. See “Executing JSP and Servlet Source Files” in the online help for additional information.

What Is a Scriptlet?

A *scriptlet* is a scripting element that enables you to enter valid Java code into a JSP page. Variables and methods declared in a declaration element are available to other scriptlets in the same JSP page. For more information on scripting elements, see “Scripting Elements” on page 37.

When to Use Scriptlets

In the past, many JSP development guides have focused on using scriptlets. It is now recommended that you avoid using scriptlets. They make JSP pages more difficult to understand and maintain. Typically, the content developers who might be creating and modifying the JSP pages are unaccustomed to Java code. Hence, they might use tools that would ruin the scriptlet code. Sometimes Java code is necessary to perform processing within the JSP file. An example would be the formatting of a table or the presentation of a large amount of information in chunks. In these circumstances, consider using custom tag library features. See “Using Existing Tag Libraries” on page 78. You can also create View Creation Helper beans with properties accessible with JSP tags. See “View Creation Helpers” on page 57 and “Using Additional Classes or Beans” on page 70 for more information.

The use of scriptlets can be beneficial in limited circumstances and is fully supported by the IDE. An example is creating early prototypes of new features you later plan to encapsulate in a bean or tag handler.

Working With JSP Includes

To make it easier to create complex pages from modular components, JSP pages can include other pages. These pages could include JSP pages and HTML or XML files. You can use either the `<jsp:include>` action or the `<%@include%>` directive to subsume these pages. For details on the JSP life cycle, including translation and runtime, see “JSP Page Life Cycle” on page 34. For an example of how to include a composite view in your JSP page, see “Creating a Composite View Template” on page 69.

Using the `<jsp:include>` Action

When you use JSP include actions in your code, the complete page using the included files is built at request time. A JSP page using included actions has its own implementation class. It cannot affect the processing of the remainder of the page or alter the HTTP headers of the response. Use the `<jsp:include>` action for pages whose dynamic content is likely to change after the translation of the containing JSP file. You should also use the `<jsp:include>` action when you do not know which page to include until request time. This usage is recommended because the `page` attribute can be set by an expression, for example:

```
<jsp:include page="<%= dynamicPageName %>" flush="true">
```

Here is an example of including a file with the action:

```
<jsp:include page="/foo.jsp"/>
```

Using the `<%@include%>` Directive

When you provide JSP includes using a directive, they are included at translation time in the implementation class of the containing JSP file. They can affect the remainder of the page and alter the HTTP headers of the response. Use the `<%@include%>` directive to include pages that are static and unlikely to change.

Here is an example of including a file with the directive:

```
<%@include file="bar.html"%>
```

Creating a Composite View Template

The following code sample creates a template JSP page using the include action within embedded HTML tables. The header, footer, and navigation bar cells always include the same JSP files (that is, `header.jsp`, `footer.jsp`, and `navbar.jsp`). The main cell includes the content JSP file, which is passed to the template using the `usePage` attribute.

```
<table width="100%" border="0" cellspacing="0" cellpadding="0">
<colgroup span="2">
<col width="140">
<col>
</colgroup>
<tr>
<td colspan="2">
<jsp:include page="header.jsp" flush="true"/>
</td>
</tr>
<tr>
<td align="left" valign="top" bgcolor="#dddddd">
<jsp:include page="navbar.jsp" flush="true"/>
</td>
<td align="left" valign="top">
<table width="100%" cellpadding="10px" border="0">
<tr><td>
<jsp:include page="<%= usePage%>" flush="true"/>
</font></td></tr>
</table>
</td>
</tr>
<tr>
<td colspan="2">
<jsp:include page="footer.jsp" flush="true"/>
</td>
</tr>
</table>
```

Typically, a template declares the start and the end of two items:

- The HTML document
- The table that defines the overall grid layout

It's a good practice to design included files to describe a complete HTML element, with both a start tag and an end tag (if an end tag is required).

Another approach is to use specially designed template tags. You can find an example of template tags in the Struts framework. For details, see the Template Tags section of the *Struts User Guide* at:

<http://jakarta.apache.org/struts/userGuide/struts-template.html>

Using Additional Classes or Beans

Use beans in your web module to gain access to external resources such as a database or flat files, that is, plain text files that might contain data. Additional utility classes can perform Java functions through methods accessible from servlets, beans, or JSP scriptlets. Note that a Java class must be in a package in order to use it from a JSP page.

The IDE provides a template for the creation of beans within your web application. In the New wizard, click the Beans category and select the Java Bean item. See “Creating a File” in the Core IDE online help for more information.

To specify beans used by your JSP page, you can use the `<jsp:useBean>` action. To get and set properties, respectively, on your beans, you can use the `<jsp:getProperty>` and `<jsp:setProperty>` actions.

To populate all the values of the bean from values in the JSP file, use the `setProperty` action with an `*` in the `property` attribute and remove the `value` attribute. The names of the properties in the bean must match the names of the request parameters. Each of the request parameters typically corresponds to an element of an HTML form.

Beware that if a property value is empty, the attribute is left unchanged in the bean. An example is when the user left a field blank, or cleared out the value. If this situation is a possibility, explicitly set each value by name, as shown in the example below. Do not use the `*`.

Here is an example of using a bean in your JSP page:

```
<jsp:useBean id="myFoo" scope="session" class="com.sun.FooBean"/>
```

Here is an example of using a bean and setting a property.

```
<jsp:useBean id="myBar" scope="page" class="com.sun.BarBean">
<jsp:setProperty id="myBar" property="blah" value="0"/>
</jsp:useBean>
```

Here is an example of getting a property:

```
<jsp:getProperty id="myFoo" property="fooProp"/>
```

Creating Servlets

Servlets must be created within the `WEB-INF/classes` directory of your web module or in a package that you create in that directory. Putting the servlet in a package is recommended because, as of J2SE 1.4.0, a Java class must be in a package if you reference the class from another class or from a JSP page.

Note that the `WEB-INF/classes` directory is included in the IDE's internal class path when you mount the web module in the Explorer.

When you create a servlet with the New wizard, the IDE enables you to configure deployment entries for the new servlet. Use the Deployment Configuration panel of the wizard to perform this task. A servlet created outside the IDE can be added to your web application. However, the servlet does not just execute on its own. You must add the servlet to the deployment descriptor file to coordinate it with other web resources within the web module. For additional information, see "Creating a JSP or Servlet Source File" in the JSP/Servlet online help.

As provided in the IDE for other Java classes and for JSP pages, code completion is available for servlets. Use code completion to remind yourself of available methods and values. See "Editing a JSP or Servlet Source File" in the online help for details.

You can compile the servlet from contextual menus in the Source Editor or from its node in the Explorer. With the servlet open in the Source Editor, press the F9 button to compile. If you create a servlet outside the IDE, it is treated as an ordinary Java class. For servlet features to be available, you must instruct the IDE to treat this class as a servlet. For details, see "Adding an Existing Servlet to a Web Module" in the online help.

Declaring the Servlet in the Deployment Descriptor

When you are creating reusable servlets, the deployment descriptor entry is helpful. It contains data that can be changed at deployment time without recompiling the servlet code. You can define initialization parameters to adapt the functioning of your servlet to a particular deployment situation. For example, you could specify currency, date, and time formats in `init` parameters for your servlet.

Servlets are not available to the application unless you declare them. As described above, this declaration is part of the process when you create servlets with the New wizard. However, you must add declarations to the deployment descriptor for servlets brought into your web module any other way.

Servlet Entries

A servlet entry contains either the name of the class that implements the servlet, or a path to the JSP page. It also contains a unique name that identifies it within the web application.

Servlet Mappings

For the servlet to be accessible and to receive requests, one or more servlet mappings must accompany the servlet entry. You can also map JSP pages. However, unlike servlets, JSP pages can process requests even if you do not map them. Servlet mappings match a named servlet or a JSP page to a URL pattern. The servlet or JSP page is activated if a specific condition is met. The portion of the request URI following the server identifier and the context path must match the URL pattern string. The context path is the path associated with the web module when it was deployed to the server. The URL pattern can be a definite string, or it can contain wildcard characters. A URL mapping of `/catalog/*` matches any request path beginning with `/catalog/`.

Load on Startup

The servlet container can initialize resources at any time unless a servlet or JSP page is specified as “loaded on startup.”

When a servlet is loaded at startup, the servlet is instantiated and its `init()` method is called when the container starts the web application. Similarly, when a JSP page is loaded on startup, it is compiled and initialized during startup. The JSP page is also compiled and initialized any time later, if it is changed.

When a servlet is loaded on startup, it can set up resources that are used by multiple resources. For example, a servlet could add a parameter to the servlet context. Configuring the servlet to be loaded on startup guarantees the parameter is available when other resources try to access it. If the value of load on startup is negative or not set, then the container is free to load the servlet at will. If the value is a positive integer, then the container initializes the servlet on startup. The container also loads servlets with lower values before servlets with higher values.

Adding a Servlet Entry

To add a servlet to your deployment descriptor, right-click your `web.xml` file, and choose Properties. In the Deployment pane of the Properties window, select Servlets. In the Servlet Property Editor, you can add, edit, or delete servlet entries. See “Servlets Property Editor” in the JSP/Servlet online help for details.

Displaying and Changing Servlet Entries

To display or alter deployment descriptor entries for a selected servlet, choose Properties from the servlet’s contextual menu. Then click the Deployment Entries field to edit servlet properties.

Modifying the Servlet

Modify your servlet in the IDE the same way you would proceed with any other Java class. When you make changes to a servlet or a deployment descriptor, you must redeploy the whole web module. If you are using Sun ONE Application Server 7 and if the server instance is already running, the instance will dynamically reload all the module’s components. If you are using the internal Tomcat server or a Tomcat 4.0 server you can choose Execute (Force Reload) from the contextual menu of your web module to save, recompile, and deploy the web module and to restart the server before executing the module. Note that only the servlets are recompiled. JSP files are not recompiled until you execute or debug the web module.

Servlet-Generated HTTP Responses

A servlet can write to an HTTP response. This simple example of outputting HTML from a servlet is taken from the `processRequest` method of the IDE's servlet template in the New wizard.

CODE EXAMPLE 4-1 Outputting HTML From a Servlet

```
response.setContentType("text/html");
java.io.PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Hello World Servlet</title>");
out.println("</head>");
out.println("<body>");
out.println("Hello, World!");
out.println("</body>");
out.println("</html>");
out.close();
```

It is not required that a servlet output HTML. Instead, a servlet can be used to modify the `HttpServletRequest` and `HttpServletResponse` objects. For example, the Front Controller servlet forwards the request to another servlet or JSP file, which then writes to the response object.

Using the Servlet as a Front Controller

To construct a Front Controller:

1. Create a new servlet using the IDE's servlet template. From the main menu bar's File menu, choose New, then JSP & Servlets, then Servlet.
2. On the Deployment Configuration panel of the New Servlet wizard, specify a URL pattern that matches the request URLs you want to capture. For example, specify `/ShowProducts/*`.

The servlet container attempts to match request paths to servlets in two cases. The match is attempted when the container first receives the request. It is attempted again if the request is redispached internally through a `forward` or an `include`. You cannot map a servlet that forwards to another resource using the path `/*`. This mapping would cause a recursive call to your servlet.

3. Once the servlet is created, insert processing code into the `processRequest` method to forward the request to the appropriate page. For instance, the following code sample shows how the request below might be handled:

`http://my.company.com/ShowProducts?product=stuffedbear`

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    java.io.IOException {
    String sendTo;
    if (request.getQueryString().equals("product=stuffedbear"))
        sendTo = "/WEB-INF/showStuffedBearInfo.jsp";
    else sendTo = "/WEB-INF/noProductSpecified.jsp";

    RequestDispatcher sendPage =
        getServletContext().getRequestDispatcher(sendTo);
    sendPage.forward(request, response);
}
```

In this case, the Front Controller servlet `ShowProducts` would use the query string `product=stuffedbear` to select the appropriate view. The view might consist of a product description page for the stuffed bear. The Front Controller servlet would then forward the request to that JSP page.

Use the Front Controller servlet to direct page flow when the application, rather than the user, controls the order in which pages are accessed. An example of this situation would be a set of pages that implements the checkout process. While purchasing products in a shopping cart, the user shouldn't be able to bookmark or return to any of the checkout pages without going through each step in the sequence. To implement this scenario, you would place the checkout JSP pages inside the `WEB-INF` directory of the web module. This location is not directly accessible by requests from a browser. It is only accessible through another resource that uses the `RequestDispatcher` API to serve those pages. This situation is shown in the previous example code. In this example, you can control how the pages are obtained and used by creating a Front Controller servlet. The Front Controller processes the input during the checkout process, and then determines what page to show the user next.

See "Front Controllers" on page 52 for more information on the Front Controller design pattern.

Using Additional Classes or Beans

Servlets can obtain and use additional classes and beans in the same manner as any other Java class. Note that the class or bean must be in a package in order to be used by the servlet. The IDE facilitates the creation of beans using the New wizard. For more information, see "Creating a New File" in the Core IDE online help.

Creating Filters

You can create and add a filter to your web application using the New wizard. This method is the same one used to create a JSP file or a servlet. As with servlets, you use a checkbox in the New wizard to generate deployment descriptor elements with default settings. See “Creating a Filter” in the JSP/Servlet online help for more information.

Declaring the Filter in the Deployment Descriptor

To declare your filter, use the `Filters` element in the deployment descriptor to define:

- A filter name, used to map the filter to a servlet or URL
- A filter class, used by the container to identify the filter type
- Initialization parameters for the filter, which are called `init-params`

You can also define:

- Large and small icons
- A textual description
- A display name for tool manipulation

For details see, “Adding a Filter to the Deployment Descriptor” and “Filters Property Editor” in the JSP/Servlet online help.

In order for use a filter from a web module, you must declare it in the deployment descriptor using a `<filter>` tag. A filter entry contains the name of the class (`filter-class`) that implements the filter and a unique name for the filter. You can also specify initialization parameters for the filter.

Similar to mapping a servlet, you must add one or more mappings to the deployment descriptor in the form of `<filter-mapping>` tags in order for a filter to receive requests. You can map a filter to a specific servlet by the servlet’s logical name. You can also map a filter to a group of servlets and static content by using a URL pattern

The Filter Mappings element shows the number of filter mappings defined for the web module. Filter mappings specify the URL patterns that should be processed by a particular filter. To specify additional filter mappings, use the Filter Mappings Property Editor. See “Editing Web Module Deployment Properties” and “Filter Mappings Property Editor” in the JSP/Servlet online help for details.

When the servlet container receives a request for the web application, it constructs a filter chain. The chain consists of all filters whose URL mappings match the request URI and the context path in the order the filters appear in the deployment

descriptor. The context path is the path associated with the web module when it was deployed to the server. The filters in the chain are activated when the request enters the container. Unlike servlets, no filter chain is constructed for internal dispatches. Hence, you can map a filter to `/*` to have it called for every request coming into the container.

Processing HTTP Requests and Responses

You can use filters to modify an HTTP request or an HTTP response. Filters can be used for authentication. For instance, when a JSP file or servlet is requested, the filter can determine whether an end user is permitted to view the desired web component. The end user's name can be located in the session information. If the name is blank, the filter routes the request to a login page. If the name is not blank, but has no authorization to view a page, a different outcome transpires. The filter routes the request to a page explaining the required authorization. You can also use a filter to create log entries each time a response is sent to a particular client. The entries are based on information from the request and the session.

For an example, see the `SignOnFilter` class, which is part of the Java Pet Store, at:

<http://java.sun.com/blueprints/code/jps13/src/>

Using Tag Libraries

As previously described, a tag library consists of a set of tag handler classes that implements the tag library's feature set and a Tag Library Descriptor (TLD) file that describes the tags in the library and maps each tag to a tag handler. You can extend the standard set of actions by creating your own custom actions and tags. By doing so, you can make the code more reusable by separating the code into modules and encapsulating the functional units of the code within your application. With proper design, you can cleanly separate logic from formatting. This separation should enable you to eliminate the use of Java code embedded in your JSP pages.

The IDE supports:

- **The use of existing tag libraries in your web applications.** The JSP Tag Library Repository provides the JSTL (JavaServer Pages Standard Tag Library). See “Using Existing Tag Libraries” on page 78 and “Inserting Custom Actions From a Tag Library Into a JSP Page” on page 79 for details.
- **The development of new tag libraries.** The Tag Library Editor facilitates the creation and modification of your own tag libraries. See “Developing Your Own Tag Libraries” on page 81 for more information.

- **The packaging and deployment of tag libraries.** You can package your custom tag library as a JAR and then add it your web module for deployment. For details, see “Packaging and Deploying a Custom Tag Library” on page 110.

Using Existing Tag Libraries

The IDE provides the JSP Tag Library Repository to facilitate the management of existing custom tag libraries. Initially, the JSP Tag Library Repository includes the JSTL (JavaServer Pages Standard Tag Library) 1.0 from the Apache Jakarta group.

You can add tag libraries that you have created or downloaded from external sources to the JSP Tag Library Repository. Choose the JSP Tag Library Repository from the Tools menu. Use the JSP Tag Library dialog box that is displayed to add tag libraries to the repository.

Adding a Tag Library to a Web Module

In order to use a tag library in your application, you must first add it to your web module.

From the contextual menu of the WEB-INF node of your web module, choose Add JSP Tag Library. You can find the tag library either in the Tag Library Repository or in the filesystem. Some tag libraries are delivered as a single JAR file. Other tag libraries might contain additional dependent JAR files.

If you choose `standard` from the JSP Tag Library Repository, then all tag libraries that are part of the JSTL are added to the `WEB-INF/lib` directory. For more information about the JSTL, see “Tag Libraries” on page 44.

Using Tag Libraries From External Sources

To add and use a tag library from external sources in a JSP file, add its associated JAR file to the JSP Tag Library Repository. The repository enables you to store tag libraries so you can add them to web modules. The placement of other custom tag libraries in the repository makes them readily available for addition to web modules.

From the main menu bar’s Tools menu, choose Add Tag Library to Repository. In the JSP Tag Library Repository dialog box, click Add to locate the desired library.

For more information, see “Adding a Custom Tag Library to the Repository” in the online help.

Tag Library Descriptors

A *TLD* (tag library descriptor) is an XML document that defines a tag library. The servlet container uses the TLD for a tag library to interpret custom actions on certain JSP pages. The JSP pages reference that tag library through a `taglib` directive. At the highest level, the TLD defines attributes of the tag library as a whole. These properties might include its version number and the version number of its intended servlet container. At a lower level, the TLD defines each tag in the library.

The IDE enables you to create and edit TLDs without writing XML code. You create a TLD from the tag library template provided by the IDE. After you have created a TLD, you can edit it from the Explorer through menu commands. You can also edit the TLD through its customizer windows and its elements.

Inserting Custom Actions From a Tag Library Into a JSP Page

Custom actions are also commonly referred to as custom tags. However, the term *custom action* generally refers to the code construct used in a JSP page. The term *custom tag* generally refers to the code that implements the functions of a custom action.

Use the features of a tag library by coding custom actions in your JSP pages. For custom actions to use the tag library, the JSP page must declare the tag library with a `taglib` directive.

For example:

```
<%@taglib prefix="mt" uri="/WEB-INF/lib/myTagLib.jar" %>
```

The `uri` attribute of a `taglib` directive references either the TLD or a packaged tag library JAR file. The JAR file contains both the TLD and the tag handler beans. You must place the `taglib` directive before any custom actions that use the tag library. As an alternative, the `uri` attribute can be specified in the web module's deployment descriptor.

The previous example's `uri` attribute specifies a path relative to the root of the web module. The leading slash denotes the web module root.

Code completion works for the standard tag library in the `<%@taglib>` directive. For instance, in the example above, you need only type in the " (quotes) after the `uri` attribute name.

When you get to `uri`, then type `=` and press Ctrl-Spacebar. The list of URIs available in this web context is displayed.

New with the JSP 1.2 specification, some tag libraries can contain more than one TLD in the JAR file. To obtain and use these tag libraries, place the JAR file in the `WEB-INF/lib` directory, and use the URI for the desired TLD in the `uri` attribute of the `taglib` directive.

To find the URIs for the TLD files in a JAR file, place the JAR in the `WEB-INF/lib` directory. The JAR file is then mounted in the Filesystems tab below the web module. Open the mounted JAR file in the Explorer, and select the `META-INF` directory, which contains the TLD files. If you double-click any of the TLD files, a customizer appears with the URI.

For example, the `taglib` directive for the core TLD from JSTL's `standard.jar` file would be:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
```

You must place the `taglib` directive somewhere in the JSP page before the first custom action that uses the tag library.

During tag library development, your `taglib` directive should reference a TLD file rather than a tag library JAR file. The IDE inserts the class names of your tag handlers into the TLD. The tag handlers must also be in your web module's class path. Place the tag handlers in the `WEB-INF/classes` directory.

Use the `prefix` attribute of the `taglib` directive to specify an identifier. Then use this identifier to refer to the tag library from custom actions coded in the JSP page. In the example, the `taglib` directive and the custom action (defined in the specified tag library) must be in the same JSP page. The action uses the prefix `mt` to refer to the tag library. The string `table` is the name of the tag.

```
<mt:table results="productDS"/>
```

The mapping between the tag name (in this case, `table`) and the tag handler bean is specified in the TLD. Edit this mapping in the Tag Customizer window, accessible in the Explorer from the tag's contextual menu.

Developing Your Own Tag Libraries

Many useful tag libraries, in addition to the JSTL and the tag libraries included in the Struts framework, are available. For more information on the Struts framework, see “Struts” on page 59. In addition, you might visit the following web site, which is devoted to custom tag libraries, as a source: <http://jsptags.com/>

In many cases, you might want to encapsulate your business logic in your own custom tags. This way JSP pages in your web module can obtain and use the business logic easily. The tag format is familiar to web designers. Hence, the creation of custom tag libraries facilitates the insertion of these features into their JSP pages.

This section describes the support provided by the IDE for the process of developing custom tag libraries, including:

- Creation and specification of tag libraries
- Creation and specification of tags
- Creation and specification of tag attributes
- Creation and specification of scripting variables
- Generation of tag handlers

Creating a Tag Library and Tags

As previously described, a tag library consists of a set of tag handler classes that implement the tag library’s feature set and a TLD that describes the tags in the library and maps each tag to a tag handler. When you create a new custom tag library, you are actually generating a new TLD file.

You can create a tag library in either a new or an existing web module or filesystem. Right-click the root directory of the web module or filesystem in which you want to create a tag library. From the contextual menu, choose New, then choose JSP & Servlet, then choose Tag Library. This operation creates a TLD file. Use the Tag Library Customizer to define the properties of your custom tag library.

In the Tag Library Customizer, specify a short name, a display name, a Tag Library version, and a URI for your tag library. Set the Tag Handler Generation Root to the `WEB-INF/classes` directory in your web module. The tag handler classes that you generate are placed in this directory. You can also set code generation options and provide descriptive information about the tag library’s features. For more information about the properties in the Tag Library Customizer, see the “Customizing Tag Libraries” in the JSP/Servlet online help.

The contents of the Tag Library Customizer reflect the properties of the selected tag library. Once the properties for your tag library are specified, you can add and customize tags.

For more information about creating and using custom tag libraries, see *JavaServer Pages Specification, Version 1.2* available at <http://java.sun.com/products/jsp>

It is recommended that you add a tag library directly to the web module when you create it using the IDE. Make sure to create the Tag Library TLD within the web module's `WEB-INF` directory. Moreover, the tag handlers should be generated into a package within the `classes` directory. When you are ready to deploy the web module, package the tag library. Then replace the tag handler package in the `classes` directory with the JAR file. The JAR file is then placed in the `lib` directory. See “Packaging and Deploying a Custom Tag Library” on page 110 for details on packaging.

Adding and Customizing Tags

A custom tag consists of a tag signature plus a tag handler, which is a body of Java code. The IDE generates skeleton code for tag handlers based on specifications you supply in the Tag Customizer dialog box. You then edit the tag handler code directly to insert the logic that implements the features of the tag.

You can create a tag in the Explorer. Right-click the node representing the TLD to which you want to add a tag. From the contextual menu, choose Add New Tag. In the Add New Tag dialog box, edit the tag. See the “Tag Customizer Fields” section in “Customizing Tags” in the JSP/Servlet online help for more information.

You can also edit an existing tag in the Explorer. Right-click the tag you want to customize. From the contextual menu, choose Customize. In the Tag Customizer dialog box, edit the tag. See the “Tag Customizer Fields” section in “Customizing Tags” in the JSP/Servlet online help for details.

The Tag Customizer dialog box has several tabs. The General tab contains values that are to be inserted in the TLD. The Code Generation tab provides options pertaining to the tag handler class generated for the tag.

You must choose the type of content that occurs in the body of the tag. See “Specifying How the Body of a Custom Action Is Handled” on page 83 for descriptions of possible options.

Additionally, you must specify the name of the Java package for the generated tag handler classes. The default value is the tag library's Short Name.

If the generated tag handler is to contain code to find the parent, that is, the enclosing tag, click the Find Parent checkbox. The type is determined by the Of Type property. The parent's tag handler instance, if found, is placed in the variable specified by the As Variable property. If checked, Find Parent enables the Of Type and As Variable properties. The default value is unchecked. If Find Parent is checked, Of Type and As Variable must have values entered. See the “Tag Customizer Fields” section in “Customizing Tags” in the online help for more information.

Custom Actions With Bodies

Custom actions, in principle, can contain bodies. They have begin and end tags that enclose other actions, scripting elements, or plain text. For example, this sample custom action contains a body composed of plain text:

```
<mt:convertToTable>
type distance / a 30,000 / g 5,500 / z 200
</mt:convertToTable>
```

Specifying How the Body of a Custom Action Is Handled

Use the Body Content field in the Tag Customizer dialog box to specify how the body is handled. Display this window from the contextual menu of the custom action's tag handler. Then choose one of these values: `JSP`, `empty`, or `tagdependent`.

TABLE 4-1 explains the meaning of each choice.

TABLE 4-1 Meaning of Body Content Field in Tag Customizer Dialog Box

Body Content Field	Meaning
JSP	Body content is optional. The servlet container <i>evaluates JSP elements</i> and then passes the body to the tag handler. The tag handler processes the body and writes output to the <code>out</code> object according to your programming logic.
empty	Body content is not permitted.
tagdependent	Body content is optional. The servlet container <i>does not evaluate JSP elements</i> , but does pass the body to the tag handler. The tag handler processes the body and writes output to the <code>out</code> object according to your programming logic.

All tag handlers implement `javax.servlet.jsp.tagext.Tag`. Tag handlers that do not accept or process a body need only implement this interface. If there is a body, it is simply passed through to the output with the usual JSP processing. Tag handlers that process a body must also implement `javax.servlet.jsp.tagext.BodyTag`. This interface provides additional methods for handling this processing.

Adding and Customizing Tag Attributes

Tag attributes are parameters associated with tags. These parameters denote or provide values used during tag processing.

Use the Add New Tag Attribute dialog box to create tag attributes. The Tag Attribute Customizer dialog box enables you to edit existing tag attributes.

In the Add New Tag Attribute dialog, specify various properties for your new tag attribute. For details about the New Tag Attribute and the Tag Attribute customizers, see “Customizing Tag Attributes” in the online help.

Once the new tag attribute is added to the tag, the Tag Attribute Customizer is displayed. Then you can edit the attribute in the customizer.

Three fields are important to consider when creating an attribute:

- **Required Attribute checkbox.** Select to indicate that the attribute must be given an argument whenever the tag is called. By default, the box is set to `False`.
- **Value evaluated at request time radio button.** Select to specify that the value of the attribute can be dynamically calculated at request time. The button is set to `True` by default. This value is mutually exclusive with the Value evaluated at JSP translation time attribute, described below.
- **Value evaluated at JSP translation time radio button.** Select to specify that the value of the attribute is static and determined at translation time. The button is set to `False` by default. This value is mutually exclusive with the Value evaluated request time attribute, described above.

For details on the kinds of attribute properties you can provide, see the “Tag Attribute Information” section in “Customizing Tag Attributes” in the online help.

Adding and Customizing Scripting Variables

A *scripting variable* is a value that a tag exports to a JSP page. This value can then be used in an expression or scriptlet.

Use the Add New Tag Scripting Variable dialog box to create new scripting variables. In the Add New Tag Scripting Variable dialog box, specify various properties for your new scripting variable. When creating a scripting variable, the Variable Type field is particularly important. You can either choose a standard type from the list in the combo box, or you can enter a Java class name.

The Tag Scripting Variable Customizer enables you to edit the properties of existing scripting variables. To edit a scripting variable, right-click the variable’s node in the Explorer window and choose *Customize* from the contextual menu.

For more on the Add New Scripting Variable dialog and the Tag Scripting Variable Customizer, see “Customizing Scripting Variables” in the online help. For details on scripting variable properties, see the “Tag Scripting Variable Information” section of “Customizing Scripting Variables” in the online help.

Generating Tag Handlers

As you develop your tag library, you add code to the tag handler classes to implement the features your custom actions require. The IDE generates the tag handler classes for you. Generation includes any properties set in the Tag Customizer and the tag attributes and scripting variables you have added to the tag. You customize your tag library by adding tags as well as adding or modifying attributes and scripting variables. During this process, you can regenerate the tag handlers. This task accounts for the changes without losing your edits to the tag handler code. For additional information, see “Generating Tag Handlers” in the online help.

Generated Tag Handlers

Generated code appears in a package directory determined by the tag library’s Tag Handler Generation Root. This attribute is a code generation property you set in the Tag Customizer. If the value is blank, the Generation Root defaults to the directory that contains the tag library.

As described previously, you generate tag handlers from a TLD. These generated tag handlers implement the interfaces appropriate for their corresponding custom actions. Additionally, all the tag handlers’ required class members, including fields, methods, and properties, are generated. The exact list of class members depends on your TLD. Nevertheless, it always includes the methods required by the interfaces that your tag handler implements.

The specific class members generated depend on the attributes and scripting variables you have declared in your TLD. For example, if you declare an attribute called `myAttribute`, a property called `myAttribute` is generated in the tag handler.

Methods Generated

TABLE 4-2 lists the methods the IDE creates when you generate tag handlers. Methods used to get and set properties are not listed. The methods marked with an asterisk (*) denote that they are part of the `Tag` and `BodyTag` interfaces. These methods call the others, which are helper methods.

Not all methods in the `Tag` and `BodyTag` interfaces are generated. The tag handler class can be generated as extending the `TagSupport` or `BodyTagSupport` helper classes. These helper classes implement all the methods in their respective interfaces. Only the methods that need to be overridden are generated. If you need to override any other methods for the `Tag` or `BodyTag` interfaces, include them in the `TagHandler` file.

TABLE 4-2 Generated Methods in Tag Handlers

Interface	Method
Tag	<code>*doEndTag</code> <code>*doStartTag</code> <code>otherDoEndTagOperations</code> <code>otherDoStartTagOperations</code> <code>shouldEvaluateRestOfPageAfterEndTag</code> <code>theBodyShouldBeEvaluated</code> <code>theBodyShouldBeEvaluatedAgain</code>
BodyTag	All the methods generated for the <code>Tag</code> interface plus the following: <code>*doAfterBody</code> <code>writeTagBodyContent</code>

Regenerating Tag Handlers

To develop your tag library, add programming logic to the tag handler class files to provide features required by your custom actions. During development, you might need to add additional attributes or scripting variables to your TLD. If so, regenerate your tag handlers so the corresponding class members are created. In this case, some of the tag handler's methods are regenerated, and some are left untouched.

The IDE regenerates the methods `doStartTag`, `doEndTag`, and `doAfterBody`. The Source Editor does not permit you to edit these methods. This restriction is because your changes would be overwritten when you regenerated tag handlers.

Instead of editing the methods that get regenerated, place your custom code in methods that these regenerated methods call. For example, the `doStartTag` method calls the `otherDoStartTagOperations` and `theBodyShouldBeEvaluated` methods.

The `doStartTag` method also returns an `int` code value to indicate whether the body *should* be evaluated. In the IDE, use the `otherDoStartTagOperations` method for the processing that needs to be performed at the beginning of the tag. Use the `theBodyShouldBeEvaluated` method to return a boolean value that is translated into the correct code value. Code placed in these two methods is not affected by regeneration.

TABLE 4-3 indicates which methods are regenerated and which methods you can edit.

TABLE 4-3 Editable Methods in Tag Handlers

Do Not Edit These Methods	Put Your Custom Code in These Methods Instead
doEndTag	otherDoEndTagOperations shouldEvaluateRestOfPageAfterEndTag
doStartTag	otherDoStartTagOperations theBodyShouldBeEvaluated
doAfterBody	writeTagBodyContent theBodyShouldBeEvaluatedAgain

Testing Tag Libraries

You can create tag libraries in their own IDE filesystem, or within an existing web module's filesystem. If you want to test a tag library that isn't already in a web module, you can test the tag library by performing the following steps:

1. Right-click the root of the tag library filesystem, and select Convert Filesystem into Web Module from the Tools menu. Leave the `.tld` file and the generated and compiled Java tag handler classes in place.
2. Add the Tag Libraries element in the deployment descriptor for *your-tag-lib.tld*.
3. Create a JSP page, and add references to your new tags.
4. Deploy and execute the web module.

After testing your tag library, you can include it in other web modules by packaging it as a JAR file. For more information, see “Packaging and Deploying a Custom Tag Library” in the JSP/Servlet online help.

To facilitate the use of this tag library in other web modules, you should add your new tag library JAR file to the Tag Library Repository in the IDE. For details, see “Adding a Custom Tag Library to the Repository” in the JSP/Servlet online help.

Working With Databases

Database interaction is a significant aspect of web application development. The information contained within a database is used to drive the dynamic nature of JSP files and servlets within a web module. Hence, access to this data can be crucial.

When you want to work with a database in your application, create beans to access the database. Then use tag handlers to access the beans.

The IDE provides several tools that enable you to view and modify databases that supply JDBC drivers. See *Using Java Database Connectivity* for more detailed information.

When using JDBC to gain access to a database from within your web application, perform the following steps:

1. Create data access beans with methods to initialize connections to the database, including the management of connection pools and caches.
2. Gain access to specific information using queries.
3. View and update the information in row sets.

For an example of creating a data access bean, see: “Duke’s Bakery, Part II, A JDBC Order Entry Prototype - Continued” by visiting the following web site:

<http://developer.java.sun.com/developer/technicalArticles/Database/dukesbakery2/>

These data access beans can then be used directly by servlet or filter classes. You can also refer to them from within a JSP file using the `jsp:useBean` tag. For more information, see “Using Additional Classes or Beans” on page 70.

The PointBase database and the internal Sun ONE Application Server 7 are configured by default to work together in the IDE. For information on starting the PointBase database, see “Using Databases With the IDE” in the *Sun ONE Studio 5, Standard Edition Getting Started Guide*. For information on setting up database connectivity with a Tomcat server, see “Setting Up the Web Server Environment for Database Access” on page 97.

When using other databases or deploying to other web servers, you have two options. Add the database driver files to one of two locations:

- The web module’s `WEB-INF/lib` directory
- The web server’s common `lib` directory.

Even though the IDE’s DB Explorer can be configured to use a particular database, you are still required to place the appropriate driver in the web module or the web server. Use the IDE’s Database Explorer to confirm your connection to a database and your access to an appropriate driver for it.

To utilize tags that simplify query and presentation tasks, you can add custom tag libraries to the web module. For details, see “Developing Your Own Tag Libraries” on page 81.

It is recommended that you create custom tags to:

- Display data to the user
- Query data
- Update data in the database

Adding these tags to your web module simplifies the addition of query and presentation tasks within the JSP page. It also provides the clean separation of business logic code from its presentation to the end user. See the *J2EE Tutorial* for an example of an IDE-developed web application that accesses a database.

The tag handler classes for these tags can directly gain access to the methods in the data access bean. In order to use a tag library, you import it into the JSP file. See “Using Tag Libraries” on page 77 for details.

For information on the Database access tags within the JSTL, visit this web site:

<http://jakarta.apache.org/taglibs/doc/standard-doc/Overview.html>

Running, Debugging, and Deploying Your Web Application

This chapter assumes you have considered how to develop your application. It also assumes you are ready to run, debug, and deploy your application using the IDE. The chapter provides a high-level view that ties together the tasks you perform in test running and modifying your application. Then it offers some details on individual programming tasks.

See Chapter 4 for a description of developing your web application from standard components. Such components might include JSP pages, servlets, filters, and beans.

It is recommended that you work through the *Sun ONE Studio 5 Web Application Tutorial* before building your own web application with IDE. The tutorial contains more information on many aspects of the web module development process.

Running and Debugging Tasks

This section provides an overview of the tasks involved in running a web module using the IDE. When working on a web application, it often makes sense to proceed in an iterative fashion. You redeploy and check that the application works each time a component or set of components is modified. You repeat this process each time you add a new component or set of components to your application. Each task refers to a section later in the chapter that provides more details. The IDE's online help for the JSP/Servlet module also provides information on these tasks.

To run a web module using the IDE:

1. Make modifications to the component (JSP page, servlet, and so forth). This process is covered for each component type in Chapter 4.

2. Configure the web module. The IDE takes care of much of the deployment configuration with default settings. Then it provides editors, called property sheets, for you to edit the deployment descriptor file. See “Configuring the Web Module Deployment Descriptor” on page 92.
3. Deploy the web module by right-clicking the WEB-INF node and choosing Deploy from the contextual menu.
4. Test run your application in the IDE. See “Test Running Web Modules” on page 96.
5. Track data flow within HTTP transactions using the HTTP Monitor window. Use the monitor to examine data sent by JSP pages and servlets, and the data within the web application. The monitor can also be used during the debugging process to replay previous requests. For additional information on using the HTTP monitor, see “Using the HTTP Monitor to Debug a Web Application” on page 101.
6. If necessary, debug your JSP pages, servlets, and web modules using source-level debugging. For more, see “Source-Level Debugging” on page 106.
7. Package the web module as a WAR file and deploy it. A web application archive is a single file that contains all of the components of a web application. For details, see “Packaging Your Web Applications” on page 109.

Configuring the Web Module Deployment Descriptor

The *deployment descriptor file* is located in the web module’s WEB-INF directory. It provides the necessary configuration information to the web module’s deployment environment, that is, the servlet container. The deployment descriptor can contain descriptions of servlets, filters, or tag libraries that are used by the web application. It can also include requirements for external resources and security, environment parameters, and other component-specific and application-specific parameters.

In the IDE, you can modify the deployment descriptor file in two ways:

- Browse the elements of the deployment descriptor in the property sheets for the `web.xml` file.
- Open the `web.xml` file in the Source Editor, and edit it manually.

Some web modules are simple, containing only JSP files and servlets. To deploy simple web modules on an internal server, you might not need to change the deployment descriptor. Many changes are taken care of by the IDE, sparing you the effort. In some cases, however, you need to modify the following deployment descriptor elements:

- **Welcome file list.** Resources the servlet container looks for if the web application receives a request with a path name ending in a slash. The initial settings for the Welcome files are: `index.jsp`, `index.html` and `index.htm`
- **Context parameters.** Shared, usually immutable, resources used by many resources in the application. These parameters enable you to change certain values in your application without modifying any code. Context parameters often provide database access information, which can vary among deployment platforms.
- **MIME types.** The application can associate file extensions with MIME types.
- **Session time-out.** Defines the default session time-out interval for all sessions created by the web context in a whole number of minutes. If session time-out is set to less than zero, the container never times out a session. This setting is usually not a good idea in a deployed application.
- **Error page.** Creates an Error Page entry in response to something going amiss within the application. The entry instructs the application to dispatch to a specific resource (that is, a JSP page or servlet). The resource is mapped either to an Exception type or to an HTTP error code. An example might be `Error type 404, object not found`.
- **Security and authentication properties.** Enables you to set a login configuration and view the number of security constraints or security roles defined for the web module. For details, see “Editing Web Module Security Properties” in the JSP/Servlet online help.

Many other web application properties can be specified in the deployment descriptor. Refer to the *Java Servlet Specification, v2.3* for details.

Note – In order for changes to the deployment descriptor to take effect, you must redeploy the web module.

Using Property Sheets to Edit the `web.xml` File

Use the `web.xml` property sheet to edit the various elements that make up the `web.xml` file. This property sheet is also referred to as the deployment descriptor editor. These properties appear on tabbed panes:

- **Deployment.** Specify context parameters, description information, a display name, error pages, filters, JSP files, and whether the web module is distributable. You also designate large and small icons, listeners, servlets, session time-out, version information, and welcome files for the web module.
- **Security.** Specify login configuration, security constraints, and security role properties for the web module.
- **References.** Specify EJB local references, EJB references, environment entries, resource environment references, and resource references for the web module. These values are used with web modules that are deployed as part of a larger J2EE application on a J2EE application server.
- **Server-Specific Tabs.** Specify server-specific parameters, mappings, and settings.

You can modify the values in the deployment descriptor for each deployment situation. You do not have to recompile or change the code in your application. This feature is useful when you need to provide external server data. Examples might include the name of an SMTP (Simple Mail Transfer Protocol) mail server or a database server, user name and password information, or paths to locations for external resources such as HTML templates.

Registering Servlets and Filters

When you create a servlet or filter, the New wizard immediately asks for the important information and inserts the information in the deployment descriptor. You might want to update the deployment descriptor entry to specify special features, such as:

- Additional mappings for the servlet or filter
- Initialization parameters for the servlet or filter
- Large and small icons for use by external deployment tools

For more information about declaring servlets and filters, see “Declaring the Servlet in the Deployment Descriptor” on page 72 and “Declaring the Filter in the Deployment Descriptor” on page 76. For details on updating deployment descriptor entries for servlets and filters, see “Servlets Property Editor” and “Filters Property Editor” in the JSP/Servlet online help.

Registering Tag Libraries

Whether you need to register your tag library in the deployment descriptor depends on how it is packaged. You can use the Tag Libraries Property Editor for the deployment descriptor to view any entries for the tag libraries in the web module

Automatic Registration

If your tag library JAR file has a default TLD (that is, `META-INF/taglib.tld`), an entry for the tag library is made in the deployment descriptor file when the JAR file is added to the `WEB-INF/lib` directory.

If a tag library JAR file contains multiple TLD files, then it does not require a default TLD. The standard tag library (JSTL) is an example of this situation. In this case, an explicit tag library entry is not required in the deployment descriptor. The JSP translation extracts the URI fields from the TLD files in any JAR file in the `WEB-INF/lib` directory to create an implicit tag library mapping.

Explicit Registration

Not all tag libraries have a default TLD specified. You can use the Tag Libraries Property Editor to explicitly create a tag library entry in the deployment descriptor.

- The tag library has subelements for URI and for location. The URI should be relative to the `web.xml` file that points to the library.
- The location of the TLD file should be relative to the library.

For example, the following `taglib` directive makes the TLD located at `/WEB-INF/tlds/myTagLib.tld` accessible through the URI `myTags`:

```
<taglib>
  <taglib-uri>/myTags</taglib-uri>
  <taglib-location>/WEB-INF/tlds/myTagLib.tld</taglib-location>
</taglib>
```

For details, see “Packaging and Deploying a Custom Tag Library” on page 110

If you remove a tag library declaration, it does not automatically remove the corresponding JAR file from the `WEB-INF/lib` directory.

Specifying the Default URI Within the Taglib Element

The Tag Libraries Property Editor is primarily used to indicate which tag libraries have been added. It also enables you to change the URI mappings of the tag libraries. Each entry corresponds to a `taglib` element in the deployment descriptor. Refer to the URI mapping at the top of your JSP file to use the specified tag library. The URI reference also ensures that custom tag library code completion works. For more information, see “Using Tag Libraries” on page 77.

Use the Tag Libraries Property Editor to add, edit, or remove references to the tag library JAR file from the selected web module. The addition or removal of items using this property editor does not affect the tag library JAR file in the `WEB-INF/lib` directory.

If you choose Add JSP Tag Library from the selected web module's contextual menu, an entry for each tag library is created in the deployment descriptor.

Using the IDE to Edit the `web.xml` File in the Source Editor

If you know the deployment descriptor XML syntax, you can edit the `web.xml` file directly by double-clicking the `web.xml` icon. When you save your changes, the IDE automatically parses the file. It then displays any errors in the Output window on the XML Parser tab. You can also edit the file using an external text editor.

Using the Integrated Servers

The IDE is installed with two integrated servers, Sun ONE Application Server 7 and Tomcat. Both servers appear under the Installed Servers node in the Server Registry on the Explorer's Runtime tab. By default, the IDE uses Sun ONE Application Server 7 to compile and execute web components such as JSP pages and servlets.

You can use the IDE to perform the following actions:

- Deploy and delete web modules and web module groups
- Set general properties for installations of the server
- Start, restart, and stop the server
- Edit the configuration file for the server

For more information about the integrated servers, see the Sun ONE Application Server Integration Module help set and the Tomcat Plugin help set in the online help.

Test Running Web Modules

Before you can run a web module, you must first deploy it by right-clicking the `WEB-INF` node and choosing Deploy from the contextual menu. The IDE deploys the web module to the context that is designated by the Context Root value on the `WEB-INF` node's property sheet. The context root must have a leading forward slash,

for example, `/test`. This context root is used to form the web module's *document root*. For example, if a web module's context root is `/test`, the document root might be `http://host:8080/test`.

When you initiate the Deploy action the IDE saves the files, recompiles any servlets, and then deploys the web module. Note that the JSP files are not compiled until they are actually loaded at run time.

Once you have deployed a web module, you can test run the module within the IDE by using the Execute action. This action executes the application from a client web browser using the specified default browser.

You can set a web module's execution properties on the Execution tab of the WEB-INF node's property sheet.

If you want to execute several web modules as a group, you must first create a *web module group*. For details, see "Creating and Executing a Web Module Group" on page 99. A web module group is an IDE-specific construct. It enables you to specify a set of web modules to deploy together on a web server with a single action. Each module in the group must have a unique context root. The context root ensures that requests are properly sent to the correct module by the web server.

As you execute web modules and web module groups, you can use the HTTP Monitor. It enables you to examine the flow of record data. See "Using the HTTP Monitor to Debug a Web Application" on page 101 for more information.

Configuring the Server

When you deploy or execute a web module, the IDE automatically updates the server's configuration file. For more information about server configuration, see "Administration Server" in the Sun ONE Application Server Integration Module online help and "Editing the Tomcat Server Configuration File" in the Tomcat Plugin online help.

Setting Up the Web Server Environment for Database Access

PointBase is installed with the Sun ONE Studio 5 software in the subdirectory that contains the Sun ONE Application Server 7 software. If the IDE and Sun ONE Application Server 7 were installed separately, the server may or may not include PointBase software. If it does not, you must download and install PointBase software manually. See "Using Databases With the IDE" in the *Sun ONE Studio 5, Standard*

Edition Getting Started Guide for information about starting the PointBase Server and setting up database connectivity. The same chapter also contains information about using other JDBC enabled databases with Sun ONE Application Server 7.

If you are using Tomcat, you must do one of the following to make a database driver available to the Tomcat servlet container:

- Copy the driver to the web module's `WEB-INF/lib` directory.
- To add a driver as a shared resource to the internal Tomcat server, copy it into the `s1studio-install-directory/jwsdp/common/lib` directory. Note that files in this directory are shared among all users of this IDE installation.
- To add the driver as a shared resource to an external Tomcat installation, copy the driver into the `tomcat-install-directory/common/lib` directory.

See the External Execution Process property description in the “Setting Tomcat Installation Properties” section of the Tomcat 4.0 Plugin online help for details.

Note – If you installed Sun ONE Application Server 7 with the IDE, the PointBase driver is in the `pbclient42RE.jar` file in the `s1studio-install-directory/appserver7/pointbase/client_tools/lib` directory.

Note that adding a database driver to your system class path does not necessarily make it accessible to your web server.

You can test communications using your database driver with the Database Explorer in the IDE. This practice ensures that the driver you are using works properly with both the Java language and the IDE.

Executing a Single Web Module

To execute a single web module, right-click the module and choose Execute from the contextual menu. The web module executes and is displayed in your selected default browser. The execute action instructs the browser to request the root of the web module. This action causes the server to attempt to serve a welcome file within the root directory. You can set the welcome files by editing the Welcome Files property in the property sheet for the `web.xml` file. If the server cannot find a welcome file, it simply lists the contents of the directory.

Note that if you have not yet deployed the module or if you have made changes to the module, you must first deploy the module by right-clicking the WEB-INF node and choosing Deploy from the contextual menu. This deploy action causes the IDE to take the following actions:

- Save all the modified files in your web module.
- Recompile the class files (JSP pages are recompiled when they are obtained and used by the web server).

- If the web server is not running, start the server for the deployment. Note that the Admin Server must be running if you are using Sun ONE Application Server 7. Otherwise, the IDE cannot start the web server and you will receive an error message, such as “connection refused.” If the IDE needs to start the web server for deployment, it will stop the server upon completion of the deployment process.
- Deploy the module to the server directory that is specified by the Context Root value on the WEB-INF node’s property sheet. For example, if the Context Root value is /catalog, then the URL for the web module is `http://host:port/catalog`.

Note – If you are deploying to the internal Tomcat server, then you do not need to do the deploy action. The execute action both deploys and executes.

You can use the HTTP Monitor to track the requests to the web module and the data flow inside the web application. See “Using the HTTP Monitor to Debug a Web Application” on page 101 for more information. If problems occur, check the Output window that appears upon execution.

Class Path Construction

When a web module is executed in the server, a class path is constructed. The class path ensures that all necessary classes and libraries in the web module can be located. By default, the order of web module elements in the class path for servlet execution is:

1. WEB-INF/classes
2. Any JAR files in WEB-INF/lib
3. Other classes or libraries, depending on your server’s implementation of class loading

Creating and Executing a Web Module Group

The IDE provides a mechanism to deploy several web modules together so they can be run at the same time. You create a web module group node in the IDE and add web modules to it. Then you use the deploy and execute actions.

To create and execute a web module group, you perform four main tasks:

1. Create a web module group object by right-clicking a mounted filesystem, choosing New from the contextual menu, and choosing All Templates. In the New wizard that appears, expand JSPs & Servlets and select Web Module Group. Click Next and complete the steps in the wizard. A web module group node appears in the root directory of the mounted filesystem.

The web module group node should be placed outside a web module. This placement ensures that the web group is not inadvertently included when being packaged for deployment.

2. Set a URL mapping for each of the web modules to be loaded as part of the web module group by right-clicking on the web module group node and choosing Add Web Module from the contextual menu.
3. (Optional) Open the property sheet for the web module group node and specify a Target Server for the web module group. If no target server is specified, the default web server is used.
4. Right-click the web module group node and choose Deploy. After the IDE completes the deploy action, right-click the web module group node and choose Execute.

See “Creating a Web Module Group” in the JSP/Servlet online help for details on these tasks.

You can set a target server for an individual web module. You perform this task by editing the Target Server property for that web module. However, a component of that web module might be executed while the web module is running as part of a web module group. In this case, the web module runs within the server specified by the Target Server property for the web module group. For more information, see “Setting WEB-INF Properties” in the JSP/Servlet online help.

Executing on External Servers

You might have your own installation of Sun ONE Application Server 7 or the Tomcat 4.0 web server on your system. You can set your own installation as the default web server. Alternatively, you can specify that your installed server be used for a designated web module. For more information, see the section on setting the Target Server property in “Creating a Web Module Group” in the JSP/Servlet online help.

See the *Sun ONE Studio 5, Standard Edition Getting Started Guide* for information about using an external Sun ONE Application Server 7. For an external Tomcat 4.0 web server, configure the IDE’s Tomcat plugin module to use that instance rather than the internal server. See “Adding a Tomcat 4.0 Installation” in the Tomcat Plugin online help for details.

The integration with the Tomcat web server has two modes of operation:

- **Full mode.** HTTP monitoring and JSP and servlet debugging features are enabled. The IDE adds some elements to the server's `server.xml` configuration file.
- **Minimum mode.** HTTP monitoring and JSP and servlet debugging are not available. The server's configuration file is modified only to add the deployed module's context path.

Debugging Web Applications

The IDE provides two tools for debugging web applications:

- The HTTP Monitor
- Source-level debugging for both JSP files and servlets

Using the HTTP Monitor to Debug a Web Application

The HTTP Monitor is a lightweight tool that can be used as an alternative or a complement to source-level debugging. It is provided by the IDE for collecting data related to the execution of JSP files and servlets in the servlet container. The data records generated by the HTTP Monitor streamline the work involved in debugging JSP files and servlets. The HTTP Monitor records information about each request that comes into the server. This data includes incoming and outgoing cookies, session information maintained on the server, the servlet environment information, and HTTP headers.

The HTTP Monitor provides a view into the data flow among web application components of the web application. You can use the monitor to examine the properties of the HTTP requests. You can also use it to inspect any data maintained on the server whenever a request is processed by a JSP page or a servlet. For example, you can find out if:

- The JSP page or servlet received any form data with the request. If so, you can determine what the values were.
- An HTTP session was associated with the request. If so, you can determine what attributes were set before or after the JSP page or servlet was activated.
- The JSP page or servlet received or created any cookies.

If you discover a bug while test running, use the HTTP Monitor to view data records associated with requests leading up to it. You can quickly find out if the problem was caused by a JSP page or servlet not receiving the data it expected. In this case, you can determine which resource started the problem. By using the Edit and Replay

feature, you can tweak requests before resending them. This technique enables you to determine if sending different data fixes the bug. Often this knowledge is enough to identify the lines of code causing the problem. If not, you might narrow down an individual JSP page or servlet to step through with the source-level debugger. For details, see “Source-Level Debugging” on page 106.

Once you have fixed the bug, use the HTTP Monitor again to replay the request or requests that exposed the problem. This technique can save time if reproducing the steps involves entering data through a browser, for example.

The HTTP Monitor enables you to save recorded request data. If you find a problem you do not have time to fix immediately, save the relevant records and go through the process described above later. You can also, for example, save a sequence of requests representing an application sanity test. You can then run it every time you change something.

In summary, use the monitor to:

- View data, store information for future sessions, and replay and edit previous requests
- Determine which component is sending or receiving incorrect data before you begin source-level debugging

For details on error sources in web applications, see “Common Errors in Web Applications” on page 25. For details on source-level debugging, see “Source-Level Debugging” on page 106.

Deploying the HTTP Monitor

Before you can use the HTTP Monitor on a web application, you must perform the following steps to make the monitor available to the server:

1. Copy *IDE-install-directory/modules/schema2beans.jar* and *IDE-install-directory/modules/ext/httpmonitor.jar* to the web module's `WEB-INF/lib` directory.
1. In the Deployment tab of the property sheet, display the property sheet for the Deployment Descriptor node (`WEB-INF/web.xml`), click the Filters value field, click the ellipses (...) button to display the Filters dialog box, and click Add.
2. In the Add Filter dialog box, set the Filter Logical Name to `HTTPMonitorFilter` or a similar descriptive name.
3. In the Filter Class text field, type the following:

```
org.netbeans.modules.web.monitor.server.MonitorFilter
```
4. Click Add under the Filter Mapping table to display the Add Filter Mapping dialog box and click OK to accept the default of `/*`.

5. Click Add under the Init Parameters table to display the Add Init Parameter dialog box.
6. In the Init Param Name text field, type `netbeans.monitor.ide`.
7. In the Init Param Value text field, type the following (replace the variables with the actual values):
name-of-host-that-runs-IDE:port-of-internal-HTTP-server
If you are unsure of the port number, open the property sheet for the HTTP Server node in the Runtime tab of the Explorer window. The Port property shows the port number.
8. Click OK to dismiss the Add Init Parameter dialog box and click OK to dismiss the Add Filter dialog box.

Remember to remove the JAR files, the filter, and the filter mappings before you deploy to a production server.

Note – If you are using a Tomcat server that is managed by the IDE’s Tomcat server plugin, and if the integration mode is set to Full, you do not need to perform these steps. By default, the internal Tomcat server is set to full integration mode.

Starting the HTTP Monitor

The HTTP Monitor consists of a top level window within the IDE, and a server-side component that runs on the execution server. To open the HTTP Monitor, choose HTTP Monitor from either the IDE’s View menu or Debug menu. For more information, see “Monitoring Data Flow on the Web Server” in the HTTP Monitor module online help.

Viewing Monitor Data Records

The HTTP Monitor consists of two panels. On the left, the HTTP Monitor records panel contains a tree view of known records. On the right, the HTTP Monitor records display panel presents the data associated with the selected monitor record.

In the tree view, the All Records category contains two subcategories: Current Records and Saved Records. Individual monitor data records reside in either of these subcategories. Entries in Current Records are available only during the current IDE session. Current monitor data records persist across restarts of the web server. They are only cleared on a restart of the IDE or when deleted. Entries in Saved Records persist until deleted.

Sometimes, a data record is created as the result of one resource dispatching the request to another resource. An example of this situation is a JSP forwarding to a servlet. In this case, the record is shown as an expandable node in the tree. To see how the request was processed by any other resources activated by the `forward` or `include` actions, expand the node.

Monitor data records in all categories can be sorted according to various criteria using the buttons above the tree view. For details on sorting criteria, see “Using the HTTP Monitor Toolbar” in the HTTP Monitor online help.

For each selected monitor data record, corresponding information appears in the HTTP Monitor records display panel. The records display panel consists of these panes:

- **Request.** Shows the request URI, method, query string, parameters or posted data, protocol, client IP address, scheme, exit status, and list of request attributes before and after the request.
- **Cookies.** Shows a list of incoming and outgoing cookies. For incoming cookies, information includes the cookie name and the cookie value. For outgoing cookies, information includes the name, the value, the expiration time, which domain the cookie should be sent to. Outgoing *cookie* information also includes which path the cookie should be sent to and whether it requires a secure protocol.
- **Session.** Shows status of any HTTP session associated with the request before and after the request’s processing. The pane describes whether the session was created as a result of the request’s processing. Session properties such as ID, creation time, last accessed time, and the maximum inactive interval are displayed. It also shows session attributes set before and after the request’s processing.
- **Servlet.** Shows the name of the servlet as configured, class name, package name, and optional servlet information. It also displays relative and translated paths.
- **Context.** Shows the name of the servlet context created from the web module and the absolute path to the context. This pane displays all the context attributes set when the request started processing. It also shows any initialization parameters given to the request.
- **Client and Server.** Shows data about the application that generated the HTTP request. The data includes which protocol it used, its IP address, and if the information is provided, which application was used. The pane displays what locales, encodings, character sets, and file formats the application supports. It also shows data about the servlet engine that processed the request. This data includes the Java version, platform, host name, and port number of the HTTP service.
- **Headers.** Shows the HTTP headers that came in with the request. The headers are constructed by the HTTP client, for instance, a browser. The exact nature of headers varies from client to client. They usually include information such as the nature of the client, for instance, software and operating system. They also contain language preferences and the file formats the browser accepts.

Replaying Requests

You can replay HTTP requests associated with the Current Records and Saved Records subcategories of the All records tree view. Replaying a request makes the server process a request identical to the request that was recorded. The response is shown in the IDE's default web browser, just as it is when you execute a web module.

This feature can simplify the process of testing the correction to a bug. For example, you might have found a bug and modified some code in a resource that processes form data. To verify the correction, you would usually load the form into the browser, enter data, and submit it. Using the Replay action on the original request record, you could bypass the need to reenter the data. Another example would be if you needed to step through several requests before you could reproduce a bug. The bug might be in a resource that is part of the implementation of a checkout. Moreover, users cannot enter the checkout unless they have placed something in the shopping cart. In that case, you could replay several recorded requests. This action would place the web application in a state where the bug correction could be confirmed.

See “Editing and Replaying Requests in the HTTP Monitor” in the HTTP Monitor online help for details.

Editing and Replaying a Specified Request

The Edit and Replay dialog box enables you to edit a number of properties prior to resending the request information. These properties include the request parameters and/or the query string, the cookies, and the headers. An option is also provided to send the request to a different server, if it fully supports the HTTP Monitor.

You can use the Edit and Replay dialog box to send requests to a different server.

Use the Edit and Replay feature to:

- Check if a web component that is not functioning works with different input
- Change items such as the locale setting of the client without having to modify your browser's settings

Specifying Session Cookies for Replay Requests

If you replay a request, the HTTP Monitor, by default, uses the session cookie that the browser is sending, if it sends a cookie. The monitor does not use the session cookie recorded with the request. You can control this behavior so the recorded cookie, or a cookie you specify, is used instead. See “Editing and Replaying Requests in the HTTP Monitor” in HTTP Monitor online help for details.

Source-Level Debugging

When a JSP page is executed as part of a web application, a servlet is actually what is executed. The JSP page is translated into a servlet, called the *generated servlet*. The generated servlet is then compiled and run. When using Sun ONE Application Server 7 or the Tomcat server, the IDE maintains a mapping between lines in the original JSP source and lines in the generated servlet. This feature enables you to debug JSP pages from either:

- The perspective of the JSP file itself
- The source of the servlet into which the JSP file was translated

Source-level debugging consists of several enhancements to the standard IDE debugging environment. These enhancements enable JSP and generated servlet files to be viewed simultaneously. Breakpoints set in one are automatically reflected in the other. However, breakpoints removed from the generated servlet are not removed from the corresponding JSP source file.

A JSP file, with all its statically included files, maps to a single servlet file. (See “Using the `<%@include%>` Directive” on page 68 for details.) Specifically, one line in the JSP file maps to one or more lines in the servlet file. The servlet also contains code generated for all JSP pages, with no explicit representation in the JSP source.

Using the Debugger

To debug your application using the IDE’s source-level debugging features, you set breakpoints, view the generated servlet, and start the debugger, as described in the following sections.

Setting Breakpoints

Setting breakpoints in a regular servlet or class file is no different from debugging other Java classes. For JSP pages, you can set breakpoints in either the JSP source file or in the generated servlet.

If you set a breakpoint in a JSP source file, it is automatically reflected in the generated servlet. (Remember that no one-to-one-line correspondence exists between the two files). If you set a breakpoint in a generated servlet, it is reflected in the JSP page with one exception. If the breakpoint is in part of the servlet derived from a statically included JSP page, it is not reflected. Removing a breakpoint from the JSP file removes the corresponding breakpoint in the generated servlet. However, the reverse is not the case.

Viewing the Generated Servlet

From the Explorer, click the Filesystems tab. Select the JSP source file. Right-click to display the contextual menu. If the JSP file has not yet been compiled, that is, the View Servlet action is not enabled, choose Compile from the contextual menu. Once the JSP file has been compiled, choose View Servlet from the contextual menu.

The Source Editor opens with a view of the generated servlet code. If the JSP file is already opened in the editor, you can also view its generated servlet from the editor's contextual menu. Right-click inside the editor window, and select the Show code in servlet action.

By default, the generated servlet file appears in a new tabbed pane of the Source Editor.

Note that the generated servlet is not automatically updated if you make changes to the JSP source. After you make changes, run the View Servlet or the Show code in servlet action again to see the new version.

Note – Sometimes you encounter a compilation error while compiling a JSP page, and the problem is not obvious when you are looking at the JSP source code. In this situation, it often helps to look at the generated servlet. This practice can help in finding mistakes such as missing quotes.

Starting the Debugger

Start the debugger by selecting the resource you would like to debug in the Explorer. Then choose the Start item from the IDE's Debug menu. It is not necessary to stop and restart the debugger to debug a different resource. With the web module running, either follow a link in the browser window, or type the URL directly into the browser's location field.

When debugging a helper class that cannot be run on its own, you must start the debugger on a different resource, for example, on the web module's WEB-INF node.

To debug a web module group, select the WEB-INF node from any web module within the web module group. Then choose Start from the Debug menu.

For more information about standard debugging in the IDE, see “Debugging a Program” in the Core IDE online help.

When you start the debugger, the HTTP Monitor window appears. See “Using the HTTP Monitor to Debug a Web Application” on page 101 for more information on using the HTTP Monitor to aid in debugging.

Setting JSP Debugger Options

You can customize your JSP debugging sessions through the IDE's global options as follows:

- Specify the class of files in which errors are to be shown with the JSP source file and generated servlet file open. From the Options window, expand the Building node. Select the JSP & Servlets node. Then in the Properties pane, choose Show in Servlet Source or Show in JSP Source from the combo box for the JSP Compiler Errors option.
- Designate whether to skip static HTML lines between JSP tags in the JSP source, as well as the corresponding lines in the generated servlet source when debugging. Static HTML lines are lines that do not contain any JSP element or scripting language. From the Tools menu in the IDE's menu bar, choose Options. Then expand the Debugging and Executing node. Then choose JSP & Servlets Settings. In the Properties pane, set Skip Static Lines to True.

For details, see “JSP Debugger Options” in the JSP/Servlet online help.

Viewing Both JSP and Servlet Files During Debugging

You can view the JSP source file and its generated servlet file side by side during debugging. This feature helps you detect JSP errors during the debugging cycle. You can set breakpoints and step through code with both files in sync.

Open the generated servlet in the Source Editor. (See “Viewing the Generated Servlet” on page 107 for details.) With the JSP file still selected in the Explorer, choose Open from the contextual menu. By default, the JSP source file code appears in a new tabbed pane of the Source Editor. At this point, you can see the servlet code or the JSP code, but not both at the same time. Right-click the current view in the editor, and choose Clone View from the contextual menu. This action opens a new Source Editor window with a view on the same JSP code.

Now two editor windows are displayed side by side. One window provides a view of the JSP code. Another window offers a view of the servlet code. You must compile the JSP file in order to show the changes in the generated servlet. For details, see “Viewing JSP and Servlet Files During Debugging” in the JSP/Servlet online help.

Note – Debug commands apply to the view, that is, file, with focus at the time the command is issued. To issue a command on the alternate view, click that window to set focus. Then proceed with your command.

Packaging Your Web Applications

During development, a web module is typically executed in its unpacked form. This practice makes it easy to update frequently. However, when you are ready to deploy to a production environment, you can package the web module into a Web ARchive (WAR) file. This practice facilitates the transfer process. When you create a WAR file from the IDE, the IDE, by default, includes all files in the web module's filesystem except those files with the `.java` extension. For directions on using the IDE to package a web module directory into a WAR file, see “Packaging a Web Module” in the JSP/Servlet online help.

In the IDE, you can:

- Create a WAR file from a web module
- Specify WAR packaging options
- View the contents of WAR files generated from web modules
- Mount WAR files as web modules

Building a WAR File From a Web Module

You can build a WAR file from a web module in one of two ways:

- Select the root of the web module in the Filesystems Explorer. Then choose Export WAR File from the Tools menu
- Right-click the web module's WEB-INF node. Then choose Export WAR File from the contextual menu.

Then you must specify a name for the WAR file. All files under the web module filesystem are included in your WAR file by default.

For additional information, see “Packaging a Web Module” in the JSP/Servlet online help.

Specifying Options

Packaging options for web modules appear on Archive tab of the WEB-INF node property sheet. You can add files that are external to the web module to the WAR package. Files can be added from any filesystem mounted in the Explorer. Possible choices include all files, all files except `.java` (the default option), and all files except `.java`, `.jar`, and `.form`. Alternatively, you can specify a regular expression such as `*.txt` or `Pro.*`. You can choose either the predefined filter or a regular

expression, but not both the filter and the expression. Note that the regular expressions used for filters are not the same as UNIX regular expressions. For details, see “Editing Web Module Archive Properties” in the JSP/Servlet online help.

Viewing Contents

You can view the contents of a WAR package using the WAR Contents window. Files included in a WAR package are determined by properties set using the Archive tab of the WEB-INF node’s property sheet.

The WAR Contents window lists the files to be included when the web module is exported as a WAR file. It also provides path and extension information for each file.

By default, the list is sorted alphabetically by path name. Sort the list alphabetically for another column by clicking the header for that column. For more information, see “Viewing the Contents of a WAR File” in the JSP/Servlet online help.

Packaging and Deploying a Custom Tag Library

For a JSP page to reference a custom tag library, the tag library must exist within the web module containing the JSP page. When you are using a tag library within a production web module, it is a good idea to package it as a JAR file. This practice makes it easier to use your tag library in other web modules. The tag library can be added to a web module in any of the following ways:

- Using the Add JSP Tag Library menu action to add a JAR file from the Tag Library Repository
- Using the Add JSP Tag Library menu action to add a JAR file from the filesystem
- Copying and pasting a JAR file from another web module or filesystem.

When you use one of these methods to add a tag library JAR file to a web module, the IDE also mounts the tag library in the Filesystems pane of the Explorer.

Packaging a Tag Library as a JAR File

You package a tag library into a JAR file to make it easier to deploy. To create the tag library JAR, choose Create Tag Library JAR from the tag library's contextual menu. When you create a tag library JAR, the IDE creates a file with a `.jar` extension. It also creates an associated `jarContent` file, which adds additional classes or packages to the library. The `jarContent` file is also referred to as the *recipe* file.

For information on the properties of the newly created JAR file, see “JAR Recipe Nodes in the Explorer” in the JSP/Servlet online help.

Note – Be sure that your tag handlers are generated and compiled before creating the tag library JAR file.

Deploying a Tag Library Using the Tag Library Repository

You can deploy a tag library using the Tag Library Repository.

To add a tag library from the Tag Library Repository to a web module, choose Add JSP Tag Library from the WEB-INF node's contextual menu. Then choose the Find in Tag Library Repository item, which displays the Tag Library Repository Browser dialog box. Select one or more items from the list, and click the OK button. The appropriate JAR files for the selected tag libraries are added to the `WEB-INF/lib` directory along with any necessary entries in the deployment descriptor.

When the tag library JAR files are added to the web module, they are also mounted in the Filesystems tab of the Explorer

Deploying a Tag Library by Adding a JAR File From the Filesystem

If a tag library JAR is in another web module or mounted filesystem in the IDE, you can use the Add JSP Tag Library > Find in Filesystem action. This action locates the JAR and copies it into the `WEB-INF/lib` directory of your web module. You can add a tag library JAR file from any mounted filesystem to a web module. Right-click the WEB-INF node. Choose Add JSP Tag Library, then Find in Filesystem. If necessary, the deployment descriptor is automatically updated to include an entry for the tag library. This entry is used to map the tag library to a URI for your JSP pages to use in accessing the tags

Deploying a Tag Library by Copying and Pasting a JAR File From Another Module or Filesystem

If there is a tag library JAR in another web module or mounted filesystem in the IDE, you can choose the Copy action from the contextual menu on the Explorer's WEB-INF node. Then paste the JAR file into the WEB-INF/lib directory of your web module. When copying and pasting a tag library JAR file into a web module's WEB-INF/lib directory, a taglib element in the deployment descriptor might be created. In most cases, the IDE adds the Taglib element automatically. You can verify that the correct information is present. To perform this task, follow the instructions in "Packaging and Deploying a Custom Tag Library" in the JSP/Servlet online help.

Including a Web Module Within a J2EE Application

When using the IDE, you can create a J2EE application from an existing EJB module or web module. Alternatively, you can create the application from an Explorer filesystem or package node. In both cases, after instructing the IDE to create the new application, the new J2EE application's node appears in the Explorer. Then you can begin adding modules to the application.

Note – You must assemble a web module with an EJB module in order to build an enterprise (J2EE) application.

A J2EE application is composed of EJB modules, or web modules, or both. You create J2EE applications in the IDE, either from an existing EJB module or from a filesystem in the Explorer. After creating an J2EE application, you can add web modules to it.

For details on adding a web module to a J2EE application, see "Adding an EJB or Web Module to a J2EE Application" in the J2EE Application Assembler online help.

For more information on assembling web modules into a web application, see *Building J2EE Applications*.

Glossary

Application event

listener A class that implements one or more of the servlet event listener interfaces. It is instantiated and registered in the web application at deployment time. See also *servlet event listener*.

bean A reusable software component written to the JavaBeans specification. See also *JavaBeans*.

browser See *web browser*.

client In the client-server model of communications, a process that requests the resources of a remote server. For instance, computation and storage space.

Composite View

A design pattern that creates an aggregate view from component views. Component views might include dynamic, modular portions of the page. Sometimes called a *template*. See also *design pattern*.

cookie A mechanism used by servers to keep track of individual clients. It sends small amounts of data in the headers of HTTP requests and responses. The web resource sets an outgoing cookie in the header of the response. The client receiving the response stores the cookie until it expires. The client then sends it as part of any HTTP request to the server that the cookie comes from. The client can also send the cookie to the server specified in the cookie domain. The client also sends the cookie to requests to that server that matches the cookie's path, if a path was specified. Web sites remember a user ID between browser sessions using this technique, for example. See also *HTTP session*.

context root A name that gets mapped to the *document root* of a web client. For example, if the web module's context root is */catalog*, then the request URL might be `http://host:8081/catalog/index.html`.

custom tag A tag created by a web application developer to provide additional Java features within a JSP page. See also *JSP tag*.

deployment The process of installing software into an operational environment.

- deployment descriptor** A file that describes deployment configuration information. In this book, *deployment descriptor* refers to a file, named `web.xml`, located in the web module's `WEB-INF` directory. The deployment descriptor provides the necessary configuration information to the web module's deployment environment, that is, the servlet container. This information includes requirements for external resources and security and environment parameters. It also includes other component-specific and application-specific parameters.
- design pattern** An architectural solution to a recurring software design problem. Design patterns also consist of considered best practices for attending to the context and pressures surrounding the issue, and the outcomes and effects of the solution.
- Dispatcher** A design subpattern describing the control over which view the user sees. See also *design pattern*, *Front Controller*, *View Mapper*.
- document root** The root URL of a web module. For example, if a web module's *context root* is `/catalog`, the document root might be `http://host:8080/catalog`.
- EJB** (Enterprise JavaBeans) A component architecture for development and deployment of object-oriented, distributed, enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, multi-user, and secure. See also *JavaBeans* and *bean*.
- filter** See *servlet filter*.
- framework** APIs that are intended to simplify the design and coding process. See also *Struts* and *JSF*.
- Front Controller** A design pattern centralizing business logic for part of a web application in a single object handling incoming client requests. The client requests are for several different resources. The Front Controller might be responsible for activating Helpers and Delegates that perform business logic. Front Controllers can be used for managing model data, controlling page flow, and dispatching the request to the appropriate view. See also *Helper*, *Dispatcher*.
- Helper** A design subpattern that encapsulates processing functions (or business rules) or data retrieval behavior. See also *design pattern*, *Front Controller*.
- HTTP** (Hypertext Transfer Protocol) An application protocol governing the exchange of files, on the World Wide Web. Examples of files would include text, images, sound, and video.
- HTTP Monitor** An IDE module for the collection of information concerning the execution of JSP files and servlets in the servlet engine. For each request associated with a JSP file or servlet, the monitor records data. The data includes the incoming request, incoming and outgoing cookies, session information maintained by the server, and more.

- HTTP response** A message generated in the servlet container that includes cookies, headers, and output that eventually go to the client browser. It is a response as specified by an HTTP method. Often referred to as *response* in this book. The response is encapsulated in an `HTTPResponse` object in the servlet container.
- HTTP request** A message created in the client browser that includes attributes and cookies from the client browser. It is a request as specified by a GET or a POST method. Often referred to as a *request* in this book. The request is encapsulated in an `HTTPRequest` object in the servlet container.
- HTTP session** A Servlet API construct representing a conversation spanning multiple requests between a client browser and a web server.
- HTTPS** (Hypertext Transfer Protocol Secure Sockets) A secure HTTP protocol used widely in Internet and intranet environments. HTTPS is for exchanging secure information between clients and servers. It provides a secure connection through which applets or beans can be downloaded into the web browser. In addition, HTTPS enables these applets or beans to make secure connections to the server.
- J2EE** (Java 2 Platform, Enterprise Edition) The edition of the Java 2 platform that combines a number of technologies in one architecture. Examples of technologies are enterprise beans, JSP pages, and XML. J2EE provides a comprehensive application programming model and compatibility test suite for building enterprise-class server-side applications. See also *EJB*, *JSP technology*, *servlet*.
- J2EE application** An application that consists of J2EE components that run on the J2EE platform. Examples of J2EE components are application clients, applets, HTML pages, servlets, and enterprise beans. J2EE applications are typically designed for distribution across multiple computing tiers. For deployment, a J2EE application is packaged in an `.ear` (Enterprise Archive) file. See also *J2EE*, *J2EE web tier*.
- J2EE web tier** One of three tiers in the J2EE architecture. The web tier creates presentation logic. It accepts responses from presentation clients such as HTML and web clients. Then it provides the appropriate response. This tier is to be differentiated from the client and business tiers.
- JAR** (Java Archive file) A platform-independent file format that bundles classes, images, and other files into one compressed file, speeding up download time.
- JavaBeans** An architecture that defines a portable, platform-independent reusable component model. Beans are the basic unit in this model. See also *EJB*.
- JSF** (JavaServer Faces) A proposal to define an architecture and APIs that simplify the process of building J2EE web tier applications. See also *J2EE web tier*.

- JSTL** (JavaServer Pages™ Standard Tag Library). A standard tag library that encapsulates core features common to many JSP pages as simple tags. JSTL contains support for common, structural tasks. Support includes iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags.
- JDBC** (Java Database Connectivity) An industry standard for database-independent connectivity between the Java platform and a wide range of databases. The JDBC interface provides a call-level API for SQL-based database access.
- JSP action** A JSP object that can act on implicit objects and other server-side objects. It also can define new scripting variables. Actions follow the XML syntax for elements with a start tag, a body, and an end tag. If the body is empty, it can also use the empty tag syntax. The tag must use a prefix. An action is the abstract term that is implemented by a tag.
- A *standard* action is defined in the JSP specification. It is always available to a JSP file without being imported.
- A *custom* action is described in a portable manner by a TLD. It is a collection of Java classes imported into a JSP page by a `taglib` directive.
- JSP element** A part of the JSP page recognized by the JSP translator. An element can be a JSP action, a directive, or a JSP scripting element.
- JSP expression** A scripting element containing a valid scripting language expression. It is evaluated, converted to a `String`, and placed into the implicit `out` object.
- JSP page** A text-based web component that is dynamically translated into a servlet by the servlet container before execution. See also *JSP file*, *servlet*.
- JSP file** The physical representation of JSP page code. See *JSP page*.
- JSP scripting element** A JSP declaration, scriptlet, or expression whose tag syntax is defined by the JSP specification. Its content is written according to the scripting language used in the JSP page.
- JSP tag** A tag defined by the JSP Specification. It is a text element within a document. The document represents format information or processing logic contained in an external library. It is distinguishable as markup, instead of as data, because it is delineated in XML format. By using tags, you can avoid including Java code in the JSP page. See also *JSP tag library*.
- JSP tag library** A collection of tag handlers (Java classes) that encapsulates dynamic content or processes. They can then called through a tag in a JSP page. JSP tag libraries are part of the JSP specification and can be translated by any JSP engine. See also *JSP tag*, *custom tag*.

JSP technology	(JavaServer Pages™) Extensible web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically, the content consists of HTML or XML elements. In many cases, the client is a web browser. JSP technology is an extension of servlet technology. See also <i>JSP pages</i> , <i>servlets</i> .
listener	See <i>application event listener</i> and <i>servlet event listener</i> .
model object	Java objects that encapsulate application data inside a web application.
MIME	(Multipurpose Internet Mail Extensions) An Internet standard for sending and receiving non-ASCII email attachments, (including video, audio, and graphics). Web browsers also use MIME types to determine how to display or interpret files that are not formatted in HTML.
scope	Definition of an object's availability in relationship to other objects in the web application. The Servlet and JSP specifications define four scopes: <code>ServletContext</code> (application), <code>Session</code> , <code>Page</code> (JSP page only), and <code>Request</code> .
scripting element	See <i>JSP scripting element</i> .
scripting variable	A value that a tag exports to a JSP page. This value can then be used in an expression or scriptlet.
scriptlet	A scripting element that enables you to enter any piece of valid Java code into a JSP page. Variables and methods declared in a declaration element are available to other scriptlets in the same JSP page. The use of scriptlets in JSP pages is not recommended. Instead, encapsulate the code in a tag or a bean.
server	A network device that manages resources and supplies services to a client. A J2EE server provides a web or EJB container. See also <i>client</i> , <i>web server</i> .
server plugin	A module of the IDE sometimes provided by a web server vendor. The plugin facilitates the use of the IDE in the configuration of applications and their deployment to the server.
servlet	Any class that implements <code>javax.servlet</code> , typically subclasses of <code>javax.servlet.http.HttpServlet</code> . Servlets extend the features of web servers and web-enabled application servers. They execute within a servlet container. Servlets are typically used as Front Controllers and to generate simple HTTP responses that are not complex. See also <i>Front Controller</i> and <i>HTTP response</i> .

servlet container A container providing network services. Here requests and responses are sent, requests decoded, and responses formatted. All servlet containers support HTTP as a protocol for requests and responses. They might also support additional request-response protocols such as HTTPS. In this case, requests are handled by servlets.

A *distributed* servlet container can run a web application that is tagged as distributable. It executes across multiple Java virtual machines running on the same or on different hosts. In this situation, the scope of objects in the web application is extended. Synchronization overhead occurs because the session data must be shared among the different servers.

servlet context An object containing a servlet's view of the web application within which the servlet is running. The servlet context can be used to manage the resources of a web module. Using the context, a servlet can perform a number of tasks. It can log events and obtain URL references to resources, and set and store attributes other servlets in the context can use.

servlet event listener A class that supports event notifications for state changes in the servlet context and HTTP session objects. See also *application event listener*.

servlet filter Reusable code that inspects and modifies `HttpServletRequest` and `HttpServletResponse` objects as the servlet receives the HTTP request. (Often called a *filter* in this book)

servlet mapping The definition of an association between a URL pattern and a servlet. It is used to map requests to servlets through examining the URI of the incoming request. See *servlet*, *URI*, *HTTP request*.

session See *HTTP session*.

Struts An open-source framework from the Jakarta Project. Struts is designed for building web applications with the Java Servlet API and JSP technology. The Struts package supplies an integrated set of reusable components. They include a controller servlet, JSP custom tag libraries, and utility classes. These components for building user interfaces can be applied to any web-based connection. See *frameworks*.

Sun ONE Application Framework

A J2EE web application framework from Sun Microsystems that is geared towards enterprise web application development. The framework combines concepts such as display fields, application events, component hierarchies, and a page-centric development approach. Also known as JATO.

tag See *JSP tag*.

tag attribute A parameter associated with a tag that denotes or provides a value used during tag processing.

TLD (tag library descriptor). An XML file that describes a tag library. A JSP container uses the TLD file to interpret pages that include `taglib` directives referring to that tag library. The TLD file contains documentation on the library

as a whole. It also contains documentation on its individual tags, version information on the JSP container and on the tag library. The TLD has information about each of the actions defined in the tag library. In the IDE, the TLD file is generated when a custom tag library is created.

- URI** (Uniform Resource Identifier) The property used when a servlet is executed (or debugged) to build the URL to be displayed in browser. URIs to web applications typically have the following syntax:
`http://server:port/context path/
local resource identifier ? query string`
- value object** An intermediate representation of the model used, for example by a Helper. See also *model object*, *Helper*.
- View Creation Helper** A tag handler class used to display data in JSP pages in different design patterns.
- View Mapper** An object that can determine the processing of the request when resources differ based on the type of client.
- WAR** (Web Application Archive) A JAR file format similar to the package used for Java class libraries. A WAR file format is installed or deployed into a servlet container. In addition to web components, a WAR usually contains other files, called web resources. They include server-side utility classes (database beans, shopping carts, and so forth). Web resources include static web content (HTML, image, and sound files), and client-side classes (applets and utility classes). A web application can run from a WAR file or from an unpacked directory organized in the same format as a WAR. See also *JAR*.
- web application** A term sometimes used interchangeably in this book with *web module*. At other times, it is used to denote everything on a set of servers. Web application generally signifies a program combining all the features users need to perform a specific group of tasks on a dynamic web page with a web browser. Examples of web applications might include an electronic shopping mall or an auction site. A web application is based on a client-server model. In this model, the client is the web browser and the server is the feature set that runs remotely. A web application's set of components can include servlets, JSP pages, and utility classes. In addition, it includes static documents, client-side applets, Java classes, and some meta information tying all the elements together. See also *web module*, *web server*.
- web browser** An application that enables users to view, navigate through, and interact with HTML documents and applets. A web browser is also called a *browser*, which is sometimes referred to as the *client*. See also *client*, *web server*.
- web container** See *servlet container*.
- web component** Components defined by the Servlet and JSP specifications. Web components can only be executed if part of a web module directory or archive deployed onto a web server. See also *JSP page*, *servlet*.

- web context** The identified path to a web application's components. The path permits access to multiple web modules on a single server. See also *servlet context*.
- web client** Typically the web browser (for instance, Netscape). However, it could be a specialized application that sends HTTP requests and interprets HTTP responses. See also *web browser*.
- web module** The smallest deployable and usable unit of web resources in a J2EE application. Web modules can be packaged and deployed as web archive (WAR) files. See also *web module group*, *WAR files*, and *web application*.
- web module group** In the Sun ONE Studio 4 IDE, several web modules deployed together. See also *web module*, *web application*, *WAR files* and *web application*.
- web server** Software that supplies services to access the Internet, an intranet, or an extranet. A web server hosts web sites and provides support for HTTP and other protocols. It executes server-side programs such as CGI scripts or servlets that perform specified functions. In the J2EE architecture, a web server provides services to a web container. For instance, a web container usually depends on a web server for HTTP message handling. The J2EE architecture assumes that a web container is hosted by a web server from the same vendor. Hence, it does not specify the contract between these two entities. A web server can host one or more web containers.
- XML** (Extensible Markup Language) A markup language that enables the definition of the necessary tags to identify the data and text in XML documents. J2EE deployment descriptors are expressed in XML.

Index

A

- action elements, JSP, 36 to 37
- Add New Tag Attribute dialog box, 84
- Add New Tag dialog box, 82
- Add New Tag Scripting Variable dialog box, 84
- `addAttribute` method, 42
- application event listeners, 48
- application instance variable, 33
- application scope, 41
- audience for this book, 10
- authentication properties, defining, 93
- authentication, servlet filters and, 46

B

- beans
 - accessing from servlets, 75
 - accessing with tag handlers, 88
 - data access, creation of, 88
 - implementing Helpers as, 56
 - setting properties for, 70
 - specifying in servlets, 75
 - specifying use in JSP pages, 70
 - using in web modules, 70 to 71
 - View Creation Helpers, 57
- Body Content field, meaning in Tag Customizer dialog box, 83
- `BodyTag` interface, 83
- breakpoints, setting, 106
- browsers. *See* web browsers

C

- classpaths
 - adding tag handlers to, 80
 - IDE's internal, 71
 - order of web module elements in, 99
- Client and Server pane of records display panel, 104
- clients. *See* web browsers
- code completion
 - for available servlet methods and values, 71
 - for JSP tags, 66
 - for standard tag library, 79
- coding errors, 25
 - HTTP Monitor and, 101
 - JSP debugger options and, 108
- compiling
 - JSP pages, 72
 - servlets, 71
- Composite Views, 56 to 57
 - creating a template JSP page for, 69
- Context pane of records display panel, 104
- context parameters, in deployment descriptor file, 93
- context roots
 - property sheet, specified in WEB-INF, 96
 - server directory, mapping to, 99
- controller element, in web modules, 31

- cookies
 - HTTP request headers, in, 23
 - HTTP responses, in, 32
 - incoming and outgoing, list of, 104
 - incorrect values in, 25
 - replay requests, specifying for, 105
- Cookies pane in records display panel, 104
- custom actions
 - and custom tags, 79
 - inserting using tags, 77 to 83
 - specifying how body is handled, 83
 - tag handlers and, 83
- custom tags, adding and customizing, 82

D

- data compression, servlet filters and, 46
- Database Explorer, 98
- databases
 - access to, 21
 - accessing with beans, 34, 70
 - accessing with Helpers, 56
 - drivers, 98
 - frameworks and, 59
 - initializing connections, 43
 - managing connections to, 44
 - model objects and, 58
 - setting up web server environment for access, 97
 - updating data with custom tags, 89
 - working with, 87 to 89
- debugger
 - options, setting for JSP, 108
 - setting breakpoints, 106
 - starting, 106
 - viewing generated servlets, 107
- debugging web applications
 - HTTP Monitor, using to debug, 101 to 106
 - monitoring data flow on the web server, 101 to 105
 - setting JSP debugger options, 108
 - source-level debugging, 106 to 108
 - tools for, 23 to 24
 - viewing both JSP and servlet files during, 108
 - viewing monitor data records, 103
- declarations, JSP, 38
- deploying
 - custom tag libraries, 110 to 112

- HTTP Monitor, 102
 - production environment, to, 109
 - web modules, 73, 92, 96, 99
- deployment descriptor files
 - adding servlet entries in, 73
 - configuring, 92 to 95
 - displaying and changing servlet entries in, 73
 - editing, 24, 29
 - editing in Source Editor, 96
 - editing with property editors, 93
 - errors in, 25
 - introduced, 20
 - load on startup specification, 72
 - registering servlets and filters in, 94
 - registering tag libraries in, 94
 - servlet entries in, 72
 - servlet mappings in, 72
- deployment errors, 25
- design patterns
 - Composite Views, 56 to 58
 - Dispatchers, 53
 - frameworks for, 59
 - Front Controllers, 52 to 56
 - Helpers, 55
 - introduced, 51
 - View Creation Helpers, 57
 - View Helpers, 56
 - View Mappers, 54
- destruction of
 - JSP pages, 35
 - servlet filters, 48
 - servlets, 44
- directive elements
 - in JSP pages, 36
 - in tag libraries, 79
- Dispatchers, 53 to 54
- distributed servlet containers, 33
- document root, 97
- doFilter method, 47
- Dreamweaver templates, creating JSP pages from, 65

E

- Edit and Replay dialog box, 105
- error page entries, creating, 93

- error sources in web applications, determining, 26
- example applications
 - where to download, 16
- executing
 - on external servers, 100
 - single web modules, 98 to 99
 - web module groups, 99 to 101
- execution properties, setting web module, 97
- Explorer window
 - mounting root directory of web module in, 64
 - opening mounted JAR files in, 80
 - Server Registry on Runtime tab, 96
 - unpacking and mounting a WAR file from, 65
 - web module structure in, 63
- expression elements, JSP, 38
- external servers
 - executing web modules on, 100

F

- filter chains, 48, 76
- FilterChain object, 47
- FilterConfig method, 47
- filters. *See* servlet filters
- forward action, 36, 37
- frameworks
 - defined, 59
 - JavaServer Faces, 60
 - Struts, 59
- Front Controllers
 - constructing, 74 to 75
 - defined, 52 to 56
 - directing page flow with, 75
 - Dispatchers in, 53
 - Helpers and, 55 to 56
 - response processing in, 23
 - servlets as, 42
 - View Creation Helpers in, 57 to 58
 - View Mappers and, 54

G

- generated servlets, viewing, 107
- getAttribute method, 42
- getProperty action, 37, 71

H

- headers
 - altering HTTP response, 68
 - displayed in HTTP Monitor, 104
 - in HTTP requests, 23
 - in HTTP responses, 24
- Headers pane of records display panel, 104
- Helpers, 55 to 56
- HTTP Monitor
 - enabling, 101
 - introduced, 26
 - records display panel, 104
 - replaying requests, 105
 - starting, 103
 - viewing data records, 103 to 105
- HTTP requests
 - defined, 23, 32
 - monitoring, 23
 - Request pane in HTTP monitor records display panel, 104
 - scope in JSP pages, 41
 - web components and, 33
- HTTP responses
 - action elements and, 36
 - defined, 24, 32
 - expression elements and, 38
 - generating with servlets, 74
 - scripting elements and, 37
 - scriptlets and, 38
 - servlet filters and, 77
 - template data and, 36
 - web components and, 33
- HTTP sessions, 39 to 40
 - creating, 66
 - data, 19, 22, 24
 - defined, 24
 - displaying properties of, 104
 - invalidating, 66
 - misspelled attributes as coding errors, 25
 - servlet filters and, 77
 - showing status of, 104
 - storing data for future, 102
- HttpServletRequest objects
 - modifying, 74
 - servlet filters and, 24

- HTTPResponse objects
 - modifying, 74
 - servlet filters and, 24
- HTTPServlet interface, 43
- HTTPSession API methods, 48

I

- implicit objects in JSP pages, 40
- include action, 37
- include directive, 36, 68
- included views, in Composite View design pattern, 56
- initialization of
 - filters, 47 to 48
 - servlets, 43
- instantiation of
 - filters, 47
 - JSP pages, 35
 - servlets, 43
- interfaces
 - BodyTag, 83
 - HTTPServlet, 43
 - Servlet, 44
 - ServletContext, 33
 - Tag, 83
- invalidate method, 48

J

- J2EE (Java 2 Platform, Enterprise Edition)
 - 1.3 specification compliance, 28
 - architecture, 32 to 34
 - compliant servers, 25
 - web applications, 10, 112
 - web modules and, 62
- JAR (Java Archive) files, 63
 - in classpath construction, 99
 - servlets, classes, and beans packaged as, 62
 - Tag Libraries Property Editor and, 96
- Javadoc technology
 - using in the IDE, 16
- JavaServer Faces framework, 60
- JDBC (Java Database Connectivity)
 - driver, location in web module, 63
 - PointBase driver for, 98
 - Tomcat, setting up access for, 98
 - using Helpers to access information in a database with, 56
 - using to access database from a web application, 88
- ServletContext instance, 33
- JSP include actions, 68
- JSP pages
 - action elements, 36
 - beans, using, 70
 - code constructs, 36
 - Composite View template, 69
 - creating, 65
 - destruction of, 35
 - directive elements, 36
 - element types, 36
 - expression elements, 38
 - importing packages, 36
 - includes, working with, 67
 - instantiation of, 35
 - joining a session, 36
 - life cycle of, 34 to 35
 - modifying, 66
 - page directive, 66
 - root directory for, 65
 - scripting elements in, 37 to 38
 - scriptlets and, 67
 - session scope and, 42
 - sessions, creating and invalidating, 66
 - setting debugger options, 108
 - specifying beans for, 70
 - template data, 36
 - translation of, 35
- JSP source code
 - editing, 66
 - viewing next to servlet source code, 108
- JSP Tag Library dialog box, 78
- JSP Tag Library Repository
 - adding existing tag libraries with, 78
 - adding tag libraries from external sources with, 78
- JSP Tag Library Repository dialog box, 78
- jspDestroy method, 35
- jspInit method, 35

JSTL (JSP Standard Tag Library)
adding using JSP Tag Library Repository, 78
defined, 45 to 46
multiple TLD files in JAR file, 95

L

life cycle
JSP pages, 34 to 35
servlet filters, 47 to 48
servlets, 42 to 44
listeners, 48 to 49
load-balancing systems, 59
loading of
filters, 47
servlets, 43
localization, servlet filters and, 46
logging and auditing web application users, servlet filters and, 46

M

MIME type of file, deployment descriptor field for, 93
model element, in web modules, 31
model objects, 58 to 59
monitoring data flow on the web server, 101 to 105
Mount Filesystem dialog box, 64
mounted filesystems, web modules and, 64

N

New wizard, 62, 63, 65, 71

P

packaging custom tag libraries, 110 to 112
packaging web applications, 109 to 110
page directive, 36, 66
page scope, 41
plugin action, 37
PointBase database, 97
prefix attribute of `taglib` directive, 80

presentation element, in web modules, 31
`processRequest` method, 74

R

recommended reading, 12 to 13
records display panel in HTTP Monitor, 104
request processing
filters and, 77
in JSP pages, 35
in servlets, 43
request scope, 41
request time, 36
`RequestDispatcher` API, 46

S

scopes in JSP pages, 40
scripting elements
declarations, 38
defined, 37
expressions, 38
scriptlets, 38
scripting variables
adding new, 84
customizing existing, 84
scriptlets
defined, 38
disadvantages of, 67
security properties, defining, 93
server configuration file, 97
servers. *See* web servers
`service` method, 43
`Servlet` class, 43
servlet containers
database systems and, 98
defined, 32 to 34
deploying web modules on, 62
distributed web servers and, 22
introduced, 19, 20
matching request paths to servlets, 74
mediating links between components, 20
Welcome file list and, 93

- servlet context
 - defined, 33 to 34
 - events, 48
 - methods defined by, 33
 - name on HTTP Monitor pane, 104
 - objects, 48
 - servlet filters and, 46
 - showing name created from web module, 104
 - web module and, 63
 - servlet event listeners, 48
 - servlet filters, 46 to 48, 76 to 77
 - data compression and, 46
 - declaring, 76 to 77
 - destroy method, 48
 - destruction of, 48
 - introduced, 24
 - life cycle, 47 to 48
 - processing requests and responses with, 77
 - registering in deployment descriptor files, 94
 - Servlet interface, 44
 - Servlet pane of records display panel, 104
 - ServletContext interface, 33
 - servlets
 - creating, 71 to 75
 - declaring, 72 to 73
 - defined, 42
 - destroy method, 44
 - destruction of, 44
 - dispatchers and, 42
 - execution, 99
 - front controllers and, 42
 - generated, viewing, 107
 - init method, 43
 - initialization of, 43
 - instantiation of, 43
 - life cycle, 42 to 44
 - loading of, 43
 - modifying, 73
 - outputting HTML from, 74
 - registering in deployment descriptor files, 94
 - service method, 44
 - showing name information for, 104
 - session scope, 41
 - sessions
 - creating and invalidating, 66
 - distributing data, 59
 - time-out interval, 93
 - tracking with listeners, 48

- setProperty action, 37, 70
 - source-level debugging, 106 to 108
 - Struts framework, 59
 - style translations of XML content by servlet filters, 46
 - Sun ONE Application Server
 - Admin Server, required by web server, 99
 - integration with the IDE, 96
 - PointBase software in, 97
 - Sun ONE Studio IDE
 - debugging tools, 23 to 24
 - deployment support, 24
 - execution support in, 23
 - full web component support, 22 to 23
 - monitoring tools, 23 to 24
 - open runtime environment integration, 25
 - summary of features, 27 to 29

T

- Tag Attribute Customizer dialog box, 84
- tag attributes
 - adding, 84
 - customizing, 84
- Tag Customizer dialog box
 - body content field in, 83
 - edit a tag, using to, 82
 - generating skeleton code, using to, 82
 - generating tag handlers and, 85
- tag handlers, 85 to 87
- Tag interface, 83
- tag libraries
 - adding existing, 78
 - creating, 81
 - database access and, 89
 - defined, 44 to 46
 - deploying, 111
 - developing custom, 81 to 87
 - generating tag handler classes, 85 to 87
 - inserting custom actions into a JSP page, 79
 - JSP Tag Library Repository and, 78
 - packaging and deploying custom tag libraries, 110 to 112
 - packaging as JAR files, 111
 - recommended development procedure, 62
 - registering in deployment descriptor, 94

- scripting variables, 84 to 85
- tag attributes, 84
- testing in place, 87
- TLD and, 79
- URI mappings for, 95
- using, 77 to 87
- using from external sources, 78

Tag Libraries Property Editor, 95

Tag Library Customizer dialog box

- defining properties of custom tag libraries, 81
- Tag Handler Generation Root field in, 81

Tag Library Repository

- adding new tag library JAR file to, 87
- deploying a tag library with, 111
- using, 78
- using tag libraries from external sources with, 78

Tag Library Repository Browser dialog box, 111

Tag Scripting Variable Customizer, 84

taglib directive, 36, 45, 79

TLD (tag library descriptor)

- adding tag attributes and scripting variables to, 86
- code completion, using for, 79, 95
- creating, 81
- declaring tag attributes and scripting variables in, 85
- defined, 44, 79
- generating tag handlers from, 85
- IDE support for, 80, 81, 82
- location in web module, 82
- mapping tag name and tag handler in, 80
- placement of, 82
- referencing from JSP page, 79
- regenerating tag handlers for, 86
- URI, finding, 80, 95

Tomcat web server

- integration with the IDE, 96
- JDBC drivers and, 97
- PointBase database and, 88

translation of JSP pages, 35

U

Unpack WAR Folder dialog box, 65

URL

- references to resources, obtaining, 33
- specifying web application's, 99

useBean action, 37, 70

V

View Creation Helpers, 57 to 58

View Helpers, 56

View Mappers, 54 to 55

W

WAR (Web Archive) files

- building from a web module, 109
- format of, 63
- packaging options for, 109
- viewing contents of package, 110

web applications

- advantages of developing, 21
- challenges in developing, 20 to 22
- compared to standalone applications, 20 to 21
- complexity of tasks, 21
- data representation, flow, and processing in, 21
- debugging, 101 to 108
- defined, 10
- design patterns for, 51 to 59
- developing, 61 to 89
- error sources in, 25 to 26
- frameworks for, 59 to 60
- introduced, 19 to 20
- running and debugging tasks, 91 to 92
- structure of, 31 to 49
- test running, 96

web browsers

- accessibility of JSP pages and, 65
- creation of HTTP requests in, 23
- destination for HTTP responses, 24
- executing web applications from, 96
- HTTP Monitor and, 27
- introduced, 19
- languages and file format accepted by, 104
- replaying requests and, 105
- requests and responses in, 20
- test interaction using different, 23
- web components and, 20

- web components
 - defined, 34
 - indirect action among, 20
 - introduced, 20, 33
 - JSP pages, 34 to 42
 - servlets, 42 to 44
 - types, 33
- web containers. *See* servlet containers
- web module groups, 99
- web modules
 - classpath construction for servlet execution, 99
 - configuring, 92 to 98
 - creating, 62
 - debugging flow in, 91 to 92
 - defined, 34
 - deployment descriptors, 92
 - executing single, 98 to 99
 - execution properties, setting, 97
 - hierarchy, 63
 - importing existing, 64 to 65
 - introduced, 63
 - mounting, 64
 - programming flow in, 61 to 62
 - root of, 34
 - setting a target server, 100
 - structure of, 31
 - test running, 96
 - URL and context root for, 99
 - WEB-INF/classes directory of, 71
- web servers
 - defined, 32
 - deploying simple web modules on, 93
 - environment for database access, 97
 - executing a single web module on, 98
 - executing group of web modules on, 99
 - executing on external, 100
 - implications of server-centered execution, 21
 - integrated servers, using, 96
 - introduced, 19
 - monitoring data flow on, 101 to 105
 - open runtime environment integration, 25
 - server configuration file for, 97
 - test running web modules on, 96
- web.xml files *See* deployment descriptor files
- welcome files
 - designating, 94
 - in deployment descriptor file, 93