Sun Java System Web Server 6.1 SP11 Programmer's Guide to Web Applications

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–7658
May 2009

090430@21990

Contents

Preface

This guide describes how to create and run Java™ 2 Platform, Standard Edition (J2SE platform) applications that follow the new open Java standards model for servlets and JavaServer Pages™ (JSP) technology on the Sun Java System Web Server 6.1. In addition to describing programming concepts and tasks, this guide offers implementation tips and reference material.

This preface contains the following topics:

Who Should Use This Guide

The intended audience for this guide is the person who develops, assembles, and deploys web applications (servlets and JSPs) in a corporate enterprise.

This guide assumes you are familiar with the following topics:

- The J2SE specification
- HTML
- The Java™ programming language
- Java APIs as defined in servlet, JSP, and Java™ DataBase Connectivity (JDBC) specifications
- Structured database query languages such as SQL
- Relational database concepts
- Software development processes, including debugging and source code control

Using the Sun Java System Web Server Documentation

The Sun Java System Web Server 6.1 SP11 manuals are available as online files in PDF and HTML formats at: `http://docs.sun.com/app/docs/coll/1308.8`

The following table lists the tasks and concepts described in the Sun Java System Web Server manuals.

TABLE P–1 Sun Java System Web Server Documentation Roadmap

For Information About	See the Following
Late-breaking information about the software and documentation	*Sun Java System Web Server 6.1 SP11 Release Notes*
Information about Sun Java System Web Server 6.1 FastCGI plug-in, including information about server application functions (SAFs), installation, configuration, technical notes, and pointers to additional resources.	*Sun Java System Web Server 6.1 SP11 FastCGI Plug-in Release Notes*
Information about Sun Java System Web Server 6.1 Reverse Proxy plug-in, including information about server application functions (SAFs), installation, configuration, technical notes, and pointers to additional resources.	*Sun Java System Web Server 6.1 SP11 Reverse Proxy Plug-in Release Notes*
Getting started with Sun Java System Web Server, including hands-on exercises that introduce server basics and features (recommended for first-time users)	*Sun Java System Web Server 6.1 SP11 Getting Started Guide*
Performing installation and migration tasks: ■ Installing Sun Java System Web Server and its various components, supported platforms, and environments ■ Migrating from Sun ONE Web Server 4.1 or 6.0 to Sun Java System Web Server 6.1	*Sun Java System Web Server 6.1 SP11 Installation and Migration Guide* **Note:** If you have the Sun Java™ Enterprise System 1 installed on your system and you want to upgrade the Sun Java System Web Server 6.1 that is part of Sun Java Enterprise System 1 to Sun Java System Web Server 6.1 SP11, you must use the Java Enterprise System (JES) installer to perform the upgrade. Do not use the separate component installer included with Sun Java System Web Server 6.1 SP11.

TABLE P–1 Sun Java System Web Server Documentation Roadmap *(Continued)*

For Information About	See the Following
Performing the following administration tasks: ■ Using the Administration and command-line interfaces ■ Configuring server preferences ■ Using server instances ■ Monitoring and logging server activity ■ Using certificates and public key cryptography to secure the server ■ Configuring access control to secure the server ■ Using Java™ 2 Platform, Standard Edition (J2SE platform) security features ■ Deploying applications ■ Managing virtual servers ■ Defining server workload and sizing the system to meet performance needs ■ Searching the contents and attributes of server documents, and creating a text search interface ■ Configuring the server for content compression ■ Configuring the server for web publishing and content authoring using WebDAV	*Sun Java System Web Server 6.1 SP11 Administrator's Guide*
Using programming technologies and APIs to do the following: ■ Extend and modify Sun Java System Web Server ■ Dynamically generate content in response to client requests ■ Modify the content of the server	*Sun Java System Web Server 6.1 SP11 Programmer's Guide*
Creating custom Netscape Server Application Programmer's Interface (NSAPI) plugins	*Sun Java System Web Server 6.1 SP11 NSAPI Programmer's Guide*
Implementing servlets and JavaServer Pages™ (JSP™) technology in Sun Java System Web Server	*Sun Java System Web Server 6.1 SP11 Programmer's Guide to Web Applications*
Editing configuration files	*Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*
Tuning Sun Java System Web Server to optimize performance	*Sun Java System Web Server 6.1 SP11 Performance Tuning, Sizing, and Scaling Guide*

How This Guide Is Organized

This guide provides a Sun Java System Web Server environment overview for designing web applications, and has the following chapters:

- Chapter 2, Web Applications

  This chapter introduces web applications and describes how they are supported in Sun Java System Web Server.
- Chapter 3, Using Servlets

  This chapter describes how to create and use servlets.
- Chapter 4, Using JavaServer Pages

  This chapter describes how to create and use JSPs.
- Chapter 5, Session Managers

  This chapter describes how to create and manage a session that allows users and transaction information to persist between interactions.
- Chapter 6, Securing Web Applications

  This chapter describes the basic security features of the Sun Java System Web Server and how to write secure web applications.
- Chapter 7, Deploying Web Applications

  This chapter describes how web applications are assembled and deployed in the Sun Java System Web Server.
- Chapter 8, Debugging Web Applications

  This chapter provides guidelines for debugging web applications in Sun Java System Web Server.
- Appendix A, Internationalization Issues

  This appendix discusses internationalization issues pertaining to servlets and JSPs.
- Appendix B, Migrating Legacy Servlets

  This appendix discusses migrating legacy servlets.

Documentation Conventions

This section describes the types of conventions used throughout this guide.

- **File and directory paths**

  These are given in UNIX® format (with forward slashes separating directory names). For Windows versions, the directory paths are the same, except that backslashes are used to separate directories.
- **URLs** are given in the format:

`http://server.domain/path/file.html`

In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual file name. Italic items in URLs are placeholders.

- **Font conventions** include:
  - The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, path names, directory names, and HTML tags.
  - `Italic` monospace type is used for code variables.
  - *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
  - **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.

Installation root directories are indicated by *install_dir* in this guide.

By default, the location of *install_dir* is as follows:

- On UNIX-based platforms: `/opt/SUNWwbsvr/`
- On Windows: `C:\Sun\WebServer6.1`

Product Support

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:

  `http://www.sun.com/service/sunone/software`

- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems.

- Description of the problem, including the situation where the problem occurs and its impact on your operation.
- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem.
- Detailed steps on the methods you have used to reproduce the problem.
- Any error logs or core dumps.

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Feedback.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–2 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–3 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

◆ ◆ ◆ CHAPTER 1

Web Applications

This chapter provides a basic overview of how web applications are supported in the Sun Java™ System Web Server 6.1.

This chapter includes the following sections:

Introducing Web Applications

Sun Java System Web Server 6.1 supports the Java™ Servlet 2.3 API specification and the JavaServer Pages (JSP) 1.2 specification, which allows servlets and JSPs to be included in web applications.

A web application is a collection of servlets, JavaServer Pages, HTML documents, and other web resources that might include image files, compressed archives, and other data. A web application can be packaged into a Web ARchive file (a WAR file) or exist in an open directory structure.

In addition, Sun Java System Web Server 6.1 supports SHTML and CGI, which are non-Java™ 2 Platform, Standard Edition (J2SE platform) application components.

This section includes summaries of the following topics:

Servlets

Java servlets are server-side Java programs that application servers can run to generate content in response to a client request. Servlets can be thought of as applets that run on the server side without a user interface. Servlets are invoked through URL invocation or by other servlets.

Sun Java System Web Server 6.1 supports the Java Servlet 2.3 specification.

Note – Java Servlet API version 2.3 is fully compatible with versions 2.1 and 2.2, so all existing servlets will continue to work without modification or recompilation.

To develop servlets, use Sun's Java Servlet API. For information about using the Java Servlet API, see the documentation provided by Sun at:

`http://java.sun.com/products/servlet/index.jsp`

For the Java Servlet 2.3 specification, please visit:

`http://java.sun.com/products/servlet/download.html`

For information about developing servlets in Sun Java System Web Server, see Chapter 3, Using Servlets.

JSP

Sun Java System Web Server 6.1 supports the JSP 1.2 specification.

A JSP is a page, much like an HTML page, that can be viewed in a web browser. However, in addition to HTML tags, it can include a set of JSP tags and directives intermixed with Java code that extend the ability of the web page designer to incorporate dynamic content in a page. These additional features provide functionality such as displaying property values and using simple conditionals.

One of the main benefits of JSPs is that they are like HTML pages. The web page designer simply writes a page that uses HTML and JSP tags and puts it on his or her web server. The page is compiled automatically when it is deployed. The web page designer needs to know little about Java classes and Java compilers. Sun Java System Web Server supports precompilation of JSPs, however, and this is recommended for production servers.

JSP pages can access full Java functionality by:

- Embedding Java code directly in scriptlets in the page
- Using server-side tags that include Java servlets

Servlets are Java classes that must be compiled, but they can be defined and compiled by a Java programmer, who then publishes the interface to the servlet. The web page designer can access a precompiled servlet from a JSP page.

Sun Java System Web Server 6.1 supports JSP tag libraries and standard portable tags.

For information about creating JSPs, see Sun's JavaServer Pages web site at:

`http://java.sun.com/products/jsp/index.jsp`

For information about developing JSPs in Sun Java System Web Server, see Chapter 4, Using JavaServer Pages.

SHTML

HTML files can contain tags that are executed on the server. In addition to supporting the standard server-side tags, or SSIs, Sun Java System Web Server 6.1 allows you to embed servlets and define your own server-side tags. For more information, see the *Sun Java System Web Server 6.1 SP11 Programmer's Guide*.

CGI

Common Gateway Interface (CGI) programs run on the server and generate a response to return to the requesting client. CGI programs can be written in various languages, including C, C++, Java, Perl, and as shell scripts. CGI programs are invoked through URL invocation. Sun Java System Web Server complies with the version 1.1 CGI specification. For more information, see the *Sun Java System Web Server 6.1 SP11 Programmer's Guide*.

Creating a Web Application

▼ To create a web application

1. **Create a directory for all of the web application's files. This is the web application's document root.**

2. **Create any needed HTML files, image files, and other static content. Place these files in the document root directory or a subdirectory where they can be accessed by other parts of the application.**

3. **Create any needed JSP files. For more information, see Chapter 4, Using JavaServer Pages**

4. **Create any needed servlets. For more information, see Chapter 3, Using Servlets**

5. **Compile the servlets. For details about precompiling JSPs, see "Compiling JSPs: The Command-Line Compiler" on page 47**

6. **Organize the web application as described in "Web Application Structure" on page 103**

7. **Create the deployment descriptor files. For more information, see "Creating Web Deployment Descriptors" on page 104**

8. **Package the web application in a `.war` file . This is optional. For example:**

   ```
   jar -cvf module_name.war *
   ```

9. **Deploy the web application. For more information, see "Deploying Web Applications" on page 104**

   You can create a web application manually, or you can use Sun™ Java System Studio.

Securing a Web Application

You can write secure web applications for the Sun Java System Web Server with components that perform user authentication and access authorization. You can build security into web applications using the following mechanisms:

- User authentication by servlets
- User authentication for single sign-on
- User authorization by servlets
- Fetching the client certificate

For detailed information about these mechanisms, see Chapter 6, Securing Web Applications.

Deploying a Web Application

Web application deployment descriptor files are packaged within `.war` files. They contain metadata and information that identifies the servlet or JSP, and establishes its application role. For more information about these descriptor files, see Chapter 7, Deploying Web Applications.

Virtual Servers

A virtual server is a virtual web server that uses a unique combination of IP address, port number, and host name to identify it. You might have several virtual servers, all of which use the same IP address and port number but are distinguished by their unique host names.

When you first install Sun Java System Web Server, a default virtual server is created. You can also assign a default virtual server to each new HTTP listener you create. For details, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

Web applications can be hosted under virtual servers.

Default Web Applications

A web application that is deployed in a virtual server at a URI "/' becomes the default web application for that virtual server. For details, see "Virtual Servers" on page 21 virtual server, point your browser to the URL for the virtual server, but do not supply a context root. For example:

```
http://myvirtualserver:3184/
```

If none of the web applications under a virtual server are deployed at the URI "/', the virtual server serves HTML or JSP content from its document root, which is usually *install_dir*`/docs`. To access this HTML or JSP content, point your browser to the URL for the virtual server, and do not supply a context root but rather specify the target file. For example:

```
http://myvirtualserver:3184/hellothere.jsp
```

Servlet and JSP Caching

The Sun Java System Web Server has the ability to cache servlet or JSP results in order to make subsequent calls to the same servlet or JSP faster.

The Sun Java System Web Server caches the request results for a specific amount of time. In this way, if another data call occurs, the Sun Java System Web Server can return the cached data instead of performing the operation again. For example, if your servlet returns a stock quote that updates every 5 minutes, you set the cache to expire after 300 seconds.

For more information about response caching as it pertains to servlets, see "Caching Servlet Results" on page 37.

For more information about JSP caching, see "JSP Cache Tags" on page 50.

Database Connection Pooling

Database connection pooling enhances the performance of servlet or JSP database interactions. For more information about Java™ DataBase Connectivity (JDBC™), see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

Configuring the Web Container

You can configure logging in the web container for the entire server by:

- Using the Administration interface. For more information, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.
- Editing the `server.xml` file. For more information, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

Web Application Samples

Sun Java System Web Server 6.1 includes a set of sample web applications, which can be found in the following directory:

server_root`/plugins/java/samples/webapps/`

The directory contains the directories and samples listed in the following table. It also contains an `index.html` file that provides more information about configuring and deploying the samples.

TABLE 1–1 Sample Directories

Directory	Contains
`caching`	JSP and servlet examples that demonstrate how to cache results of JSP and servlet execution.
`i18n`	A basic J2SE web application that demonstrates how to dynamically change the display language based on user preference.
`javamail`	A servlet that uses the Javamail API to send an email message.
`jdbc`	Java DataBase Connectivity examples in the following directories: ■ `blob`: A servlet that accesses Binary Large Objects (BLOBs) via JDBC. ■ `simple`: A basic servlet that accesses an RDBMS via JDBC. ■ `transactions`: A servlet that uses the transaction API with JDBC to control a local transaction.

TABLE 1–1 Sample Directories *(Continued)*

Directory	Contains
`jndi`	Java Naming and Directory Interface™ examples in the following directories: ■ `custom`: Demonstrates using the custom resource. ■ `external`: Demonstrates using the external resource. ■ `readenv`: Demonstrates using the environment entries specified in the `web.xml` file. ■ `url`: A servlet that uses the URL resource facility to access a resource.
`jstl`	Basic examples that demonstrate usage of the JSP Standard Tag Library.
`rmi-iiop`	Basic example that demonstrates using a servlet to access a stateless EJB™ using RMI/IIOP running in Sun™ Java System Application Server 7.
`security`	Security examples in the following directories: ■ `basic-auth`: Demonstrates how to develop, configure, and exercise basic authentication. ■ `client-cert`: Demonstrates how to develop, configure, and exercise client certificate authentication. ■ `form-auth`: Demonstrates how to develop, configure, and exercise form-based authentication. ■ `jdbcrealm`: Demonstrates how to develop, configure, and exercise JDBC realm authentication.
`simple`	Basic JSP and servlet examples combined into a single web application (Tomcat 3.2.2 samples).

You can also deploy these examples using the `wdeploy` utility. For more information about this utility, see "Deploying Web Applications" on page 104.

◆ ◆ ◆ CHAPTER 2

Using Servlets

This chapter describes how to create effective servlets to control web application interactions running on a Sun Java System Web Server, including standard servlets. In addition, this chapter describes the Sun Java System Web Server features used to augment the standards.

This chapter has the following sections:

For information about internationalization issues for servlets, see Appendix A, Internationalization Issues and Appendix B, Migrating Legacy Servlets.

About Servlets

Servlets, like applets, are reusable Java applications. Servlets, however, run on a Web Server rather than in a Web Browser.

Servlets provide a component-based, platform-independent method for building web-based applications, without the performance overheads, process limitations, and platform-specific liabilities of CGI programs.

Servlets supported by the Sun Java System Web Server are based on the Java Servlet 2.3 specification.

The following list describes the fundamental characteristics of servlets.

Servlets:

- Are created and managed at runtime by the Sun Java System Web Server servlet engine.

- Operate on input data that is encapsulated in a request object.
- Respond to a query with data encapsulated in a response object.
- Are extensible.
- Provide user session information persistence between interactions.
- Can be dynamically reloaded while the server is running.
- Are addressable with URLs. Buttons on an application's pages often point to servlets.
- Can call other servlets and/or JSPs.

This section includes the following topics:

- "Servlet Data Flow" on page 26
- "Servlet Types" on page 27

Servlet Data Flow

When a user clicks a Submit button, information entered in a display page is sent to a servlet. The servlet processes the incoming data and orchestrates a response by generating content. Once the content is generated, the servlet creates a response page, usually by forwarding the content to a JSP. The response is sent back to the client, which sets up the next user interaction.

The following illustration shows the information flow to and from the servlet.

▼ To show the servlet data flow

1. **The servlet processes the client request.**
2. **The servlet generates content.**
3. **The servlet creates a response and either:**

   a. **Sends it back directly to the client**

      - or -

   b. **Dispatches the task to a JSP**

      The servlet remains in memory, available to process another request.

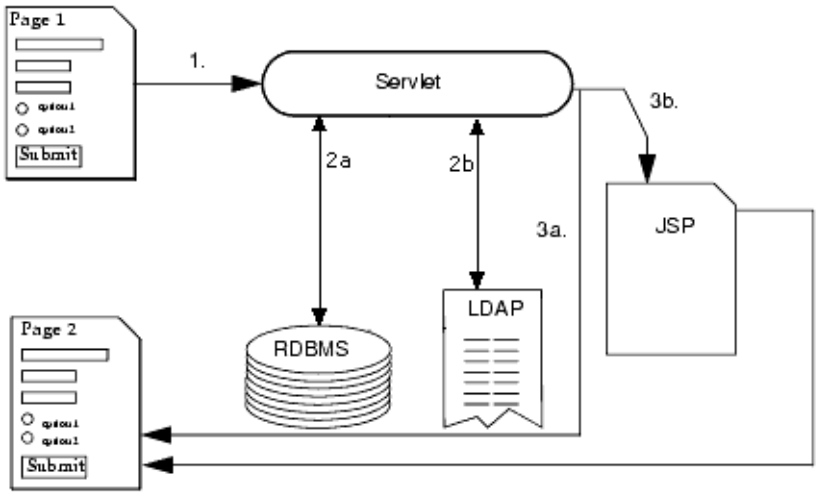

FIGURE 2–1 Servlet Data Flow Steps

Servlet Types

There are two main servlet types, generic and HTTP:

- Generic servlets
  - Extend `javax.servlet.GenericServlet`.
  - Are protocol independent. They contain no inherent HTTP support or any other transport protocol.
- HTTP servlets
  - Extend `javax.servlet.HttpServlet`.
  - Have built-in HTTP protocol support and are more useful in a Sun Java System Web Server environment.

    For both servlet types, you implement the constructor method `init()` and the destructor method `destroy()` to initialize or deallocate resources.

    All servlets must implement a `service()` method, which is responsible for handling servlet requests. For generic servlets, simply override the service method to provide routines for handling requests. HTTP servlets provide a service method that automatically routes the request to another method in the servlet based on which HTTP transfer method is used. So, for HTTP servlets, override `doPost()` to process POST requests, `doGet()` to process GET requests, and so on.

Creating Servlets

To create a servlet, perform the following tasks:

- Design the servlet into your web application, or, if accessed in a generic way, design it to access no application data.
- Create a class that extends either `GenericServlet` or `HttpServlet`, overriding the appropriate methods so it handles requests.
- Use the Sun Java System Web Server Administration interface to create a web application deployment descriptor. For details, see Chapter 7, Deploying Web Applications.

The rest of this section discusses the following topics:

Creating the Class Declaration

To create a servlet, write a public Java class that includes basic I/O support as well as the package `javax.servlet`. The class must extend either `GenericServlet` or `HttpServlet`. Since Sun Java System Web Server servlets exist in an HTTP environment, the latter class is recommended. If the servlet is part of a package, you must also declare the package name so the class loader can properly locate it.

The following example header shows the HTTP servlet declaration called `myServlet`:

```
import java.io.*;
     import javax.servlet.*;
     import javax.servlet.http.*;

     public class myServlet extends HttpServlet {
        ...servlet methods...
     }
```

Overriding Methods

Override one or more methods to provide servlet instructions to perform its intended task. A servlet does all the processing on a request-by-request basis and happens in the service methods, either `service()` for generic servlets or one of the `do`*Operation*`()` methods for HTTP servlets. This method accepts incoming requests, processes them according to the instructions you provide, and directs the output appropriately. You can create other methods in a servlet as well.

Overriding Initialize

Override the class initializer `init()` to initialize or allocate resources for the servlet instance's life, such as a counter. The `init()` method runs after the servlet is instantiated but before it accepts any requests. For more information, see the servlet API specification.

Note – All `init()` methods must call `super.init(ServletConfig)` to set their scope. This makes the servlet's configuration object available to other servlet methods. If this call is omitted, a `500 SC_INTERNAL_SERVER_ERROR` displays in the browser when the servlet starts up.

The following example of the `init()` method initializes a counter by creating a public integer variable called `thisMany`:

```
public class myServlet extends HttpServlet {
        int thisMany;
public void init (ServletConfig config) throws ServletException
        {
           super.init(config);
           thisMany = 0;
       }
       }
```

Now other servlet methods can access the variable.

Overriding Destroy

Override the class destructor `destroy()` to write log messages or to release resources that have been opened in the servlet's life cycle. Resources should be appropriately closed and dereferenced so that they are recycled or garbage collected. The `destroy()` method runs just before the servlet itself is deallocated from memory. For more information, see the servlet API specification.

Based on the example for "Overriding Initialize" on page 29, the `destroy()` method could write a log message like the following:

```
out.println("myServlet was accessed " + thisMany " times.\n");
```

Overriding Service, Get, and Post

When a request is made, the Sun Java System Web Server hands the incoming data to the servlet engine to process the request. The request includes form data, cookies, session information, and URL name-value pairs, all in a type `HttpServletRequest` object called the request object. Client metadata is encapsulated as a type `HttpServletResponse` object called the response object. The servlet engine passes both objects as the servlet's `service()` method parameters.

The default `service()` method in an HTTP servlet routes the request to another method based on the HTTP transfer method (POST, GET, and so on). For example, HTTP POST requests are routed to the `doPost()` method, HTTP GET requests are routed to the `doGet()` method, and so on. This enables the servlet to perform different request data processing depending on the transfer method. Since the routing takes place in `service()`, there is no need to generally override `service()` in an HTTP servlet. Instead, override `doGet()`, `doPost()`, and so on, depending on the expected request type.

The automatic routing in an HTTP servlet is based simply on a call to `request.get`*Method*`()`, which provides the HTTP transfer method. In a Sun Java System Web Server, request data is already preprocessed into a name-value list by the time the servlet sees the data, so simply overriding the `service()` method in an HTTP servlet does not lose any functionality. However, this does make the servlet less portable, since it is now dependent on preprocessed request data.

Override the `service()` method (for generic servlets) or the `doGet()` or `doPost()` methods (for HTTP servlets) to perform tasks needed to answer the request. Very often, this means collating the needed information (in the request object or in a JDBC result set object), and then passing the newly generated content to a JSP for formatting and delivery back to the client.

Most operations that involve forms use either a GET or a POST operation, so for most servlets you override either `doGet()` or `doPost()`. Implement both methods to provide for both input types or simply pass the request object to a central processing method, as shown in the following example:

```
public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
         throws ServletException, IOException {
   doPost(request, response);
}
```

All request-by-request traffic in an HTTP servlet is handled in the appropriate `do`*Operation*`()` method, including session management, user authentication, JSPs, and accessing Sun Java System Web Server features.

If a servlet intends to call the `RequestDispatcher` method `include()` or `forward()`, be aware the request information is no longer sent as HTTP POST, GET, and so on. In other words, if a servlet overrides `doPost()`, it may not process anything if another servlet calls it, if the calling servlet happens to receive its data through HTTP GET. For this reason, be sure to implement routines for all possible input types, as explained above. `RequestDispatcher` methods always call `service()`.

For more information, see "Calling a Servlet Programmatically" on page 36.

Note – Arbitrary binary data, such as uploaded files or images, can be problematic, because the web connector translates incoming data into name-value pairs by default. You can program the web connector to properly handle these kinds of data and package them correctly in the request object.

Accessing Parameters and Storing Data

Incoming data is encapsulated in a request object. For HTTP servlets, the request object type is `HttpServletRequest`. For generic servlets, the request object type is `ServletRequest`. The request object contains all request parameters, including your own request values called attributes.

To access all incoming request parameters, use the `getParameter()` method. For example:

```
String username = request.getParameter("username");
```

Set and retrieve values in a request object using `setAttribute()` and `getAttribute()`, respectively. For example:

```
request.setAttribute("favoriteDwarf", "Dwalin");
```

Handling Sessions and Security

From a web server's perspective, a web application is a series of unrelated server hits. There is no automatic recognition if a user has visited the site before, even if their last interaction was seconds before. A session provides a context between multiple user interactions by remembering the application state. Clients identify themselves during each interaction by a cookie, or, in the case of a cookie-less browser, by placing the session identifier in the URL.

A session object can store objects, such as tabular data, information about the application's current state, and information about the current user. Objects bound to a session are available to other components that use the same session.

For more information, see Chapter 5, Session Managers.

After a successful login, you should direct a servlet to establish the user's identity in a standard object called a session object. This object holds information about the current session, including the user's login name and any additional information to retain. Application components can then query the session object to obtain user authentication.

For more information about providing a secure user session for your application, see Chapter 6, Securing Web Applications.

Handling Threading Issues

By default, servlets are not thread-safe. The methods in a single servlet instance are usually executed numerous times simultaneously (up to the available memory limit). Each execution occurs in a different thread, though only one servlet copy exists in the servlet engine.

This is efficient system resource usage, but is dangerous because of the way Java manages memory. Because variables belonging to the servlet class are passed by reference, different threads can overwrite the same memory space as a side effect. To make a servlet (or a block within a servlet) thread-safe, do one of the following:

- Synchronize write access to all instance variables, as in public synchronized `void method()` (whole method) or `synchronized(this) {...}` (block only). Because synchronizing slows response time considerably, synchronize only blocks, or make sure that the blocks in the servlet do not need synchronization.

  For example, this servlet has a thread-safe block in `doGet()` and a thread-safe method called `mySafeMethod()`:

```
import java.io.*;
  import javax.servlet.*;
  import javax.servlet.http.*;

  public class myServlet extends HttpServlet {

  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
           throws ServletException, IOException {
     //pre-processing
      synchronized (this) {
        //code in this block is thread-safe
     }
     //other processing;
```

```
    }

  public synchronized int mySafeMethod (HttpServletRequest request)
    {
    //everything that happens in this method is thread-safe
    }
}
```

- Use the `SingleThreadModel` class to create a single-threaded servlet. When a single-threaded servlet is deployed to the Sun Java System Web Server, the servlet engine creates a servlet instance pool used for incoming requests (multiple copies of the same servlet in memory). You can change the number of servlet instances in the pool by setting the `singleThreadedServletPoolSize` property in the Sun Java System Web Server-specific web application deployment descriptor. For more information, see Chapter 7, Deploying Web Applications. Servlet is slower under load because new requests must wait for a free instance in order to proceed. In load-balanced applications, the load automatically shifts to a less busy process.

  For example, this servlet is completely single-threaded:

```
import java.io.*;
   import javax.servlet.*;
   import javax.servlet.http.*;

   public class myServlet extends HttpServlet
       implements SingleThreadModel {
      servlet methods...
   }
```

Note – If a servlet is specifically written as a single thread, the servlet engine creates a pool of servlet instances to be used for incoming requests. If a request arrives when all instances are busy, it is queued until an instance becomes available. The number of pool instances is configurable in the `sun-web.xml` file, in the `singleThreadedServletPoolSize` property of the `sun-web-app` element.

Delivering Client Results

The final user interaction activity is to provide a response page to the client. The response page can be delivered in two ways, as described in the following topics:

- "Creating a Servlet Response Page" on page 34
- "Creating a JSP Response Page" on page 34

Creating a Servlet Response Page

Generate the output page within a servlet by writing to the output stream. The recommended way to do this depends on the output type.

Always specify the output MIME type using `setContentType()` before any output commences, as in this example:

```
response.setContentType("text/html");
```

For textual output, such as plain HTML, create a `PrintWriter` object and then write to it using `println`. For example:

```
PrintWriter output = response.getWriter();
output.println("Hello, World\n");
```

For binary output, write to the output stream directly by creating a `ServletOutputStream` object and then write to it using `print()`. For example:

```
ServletOutputStream output = response.getOutputStream();
output.print(binary_data);
```

Creating a JSP Response Page

Servlets can invoke JSPs in two ways, the `include()` method and the `forward()` method:

- The `include()` method in the `RequestDispatcher` interface calls a JSP and waits for it to return before continuing to process the interaction. The `include()` method can be called multiple times within a given servlet.

  This example shows a JSP using `include()`:

```
RequestDispatcher dispatcher =
       getServletContext().getRequestDispatcher("JSP_URI");
 dispatcher.include(request, response);
   ... //processing continues
```

- The `forward()` method in the `RequestDispatcher` interface hands the JSP interaction control. The servlet is no longer involved with the current interaction's output after invoking `forward()`, thus only one call to the `forward()` method can be made in a particular servlet.

Note – You cannot use the `forward()` method if you have already defined a `PrintWriter` or `ServletOutputStream` object.

This example shows a JSP using `forward()`:

```
RequestDispatcher dispatcher =
       getServletContext().getRequestDispatcher("JSP_URI");
 dispatcher.forward(request, response);
```

Note – Identify which JSP to call by specifying a Universal Resource Identifier (URI). The path is a `String` describing a path within the `ServletContext` scope. There is also a `getRequestDispatcher()` method in the request object that takes a `String` argument indicating a complete path. For more information about this method, see the Java Servlet 2.3 specification, section 8.

For more information about JSPs, see Chapter 4, Using JavaServer Pages.

Invoking Servlets

You can invoke a servlet by directly addressing it from a Web page with a URL or by calling it programmatically from an already running servlet, as described in the following sections:

- "Calling a Servlet with a URL" on page 35
- "Calling a Servlet Programmatically" on page 36

Calling a Servlet with a URL

You can call servlets by using URLs embedded as links in HTML or JSP pages. The format of these URLs is as follows:

`http://`*server*`:`*port*`/`*context_root*`/servlet/`*servlet_name*`?`*name*`=`*value*

The following table describes each URL section. The left column lists the URL elements, and the right column lists descriptions of each URL element.

TABLE 2–1 URL Fields for Servlets within a Web Application

URL Element	Description
server:port	The IP address (or host name) and optional port number. To access the default web application for a virtual server, specify only this URL section. You do not need to specify the *context_root* or *servlet_name* unless you also wish to specify name-value parameters.
context_root	The context path without the leading "/" at which the web application is installed.
`servlet`	Only needed if no `servlet-mapping` is defined in the `web.xml` file.
servlet_name	The `servlet-name` (or `servlet-mapping` if defined) as configured in the `web.xml` file.
?name=value...	Optional servlet name-value parameters.

In this example, `leMort` is the host name, `MortPages` is the context root, and `calcMortgage` is the servlet name:

```
http://www.leMort.com/MortPages/servlet/calcMortgage?rate=8.0&per=360&bal=180000
```

Calling a Servlet Programmatically

First, identify which servlet to call by specifying a URI. This is normally a path relative to the current application. For example, if your servlet is part of an application with a context root called `OfficeFrontEnd`, the URL to a servlet called `ShowSupplies` from a browser is as follows:

`http://`*`server:port`*`/OfficeApp/OfficeFrontEnd/servlet/ShowSupplies?name=`*`value`*

You can call this servlet programmatically from another servlet in one of two ways, as described below.

- To include another servlet's output, use the `include()` method from the `RequestDispatcher` interface. This method calls a servlet by its URI and waits for it to return before continuing to process the interaction. The `include()` method can be called multiple times within a given servlet.

  For example:

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/ShowSupplies");
 dispatcher.include(request, response);
```

- To hand interaction control to another servlet, use the `RequestDispatcher` interface's `forward()` method with the servlet's URI as a parameter.

  This example shows a servlet using `forward()`:

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/ShowSupplies");
 dispatcher.forward(request, response);
```

Servlet Output

By default, the `System.out` and `System.err` output of servlets is sent to the server log, and during startup server log messages are echoed to the `System.err` output. Also by default, on Windows, there is no console created for the `System.err` output.

You can change these defaults using the Administration interface. For more information, see "Setting Error Logging Options" in Chapter 10 of the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

Caching Servlet Results

The Sun Java System Web Server can cache the results of invoking a servlet, a JSP, or any URL pattern to make subsequent invocations of the same servlet, JSP, or URL pattern faster. The Sun Java System Web Server caches the request results for a specific amount of time. In this way, if another data call occurs, the Sun Java System Web Server can return the cached data instead of performing the operation again. For example, if your servlet returns a stock quote that updates every 5 minutes, you set the cache to expire after 300 seconds.

Whether to cache results and how to cache them depends on the data involved. For example, it makes no sense to cache the results of a quiz submission, because the input to the servlet is different each time. However, you could cache a high-level report showing demographic data taken from quiz results that is updated once an hour.

You can define how a Sun Java System Web Server web application handles response caching by editing specific fields in the `sun-web.xml` file. In this way, you can create programmatically standard servlets that still take advantage of this valuable Sun Java System Web Server feature.

For more information about JSP caching, see "JSP Cache Tags" on page 50.

The rest of this section covers the following topics:

Caching Features

Sun Java System Web Server 6.1 has the following web application response caching capabilities:

- Caching is configurable based on the servlet name or the URI.
- When caching is based on the URI, this includes user-specified parameters in the query string. For example, a response from `/garden/catalog?category=roses` is different from a response from `/garden/catalog?category=lilies`. These responses are stored under different keys in the cache.
- Cache size, entry timeout, and other caching behaviors are configurable.
- Entry timeout is measured from the time an entry is created or refreshed. You can override this timeout for an individual cache mapping by specifying the `cache-mapping` subelement timeout.
- You can determine caching criteria programmatically by writing a cache helper class. For example, if a servlet only knows when a back-end data source was last modified, you can write a helper class to retrieve the last-modified timestamp from the data source and decide whether to cache the response based on that timestamp. See "CacheHelper Interface" on page 40.
- You can determine cache key generation programmatically by writing a cache key generator class. See "CacheKeyGenerator Interface" on page 41.
- All non-ASCII request parameter values specified in cache key elements must be URL encoded. The caching subsystem attempts to match the raw parameter values in the request query string.
- Since newly updated classes impact what gets cached, the web container clears the cache during dynamic deployment or reloading of classes.
- The following `HttpServletRequest` request attributes are exposed:
  - `com.sun.appserv.web.cachedServletName`, the cached servlet target
  - `com.sun.appserv.web.cachedURLPattern`, the URL pattern being cached

Default Cache Configuration

If you enable caching but do not provide any special configuration for a servlet or JSP, the default cache configuration is as follows:

- The default cache timeout is 30 seconds.

- Only the HTTP GET method is eligible for caching.
- HTTP requests with cookies or sessions automatically disable caching.
- No special consideration is given to `Pragma:`, `Cache-control:`, or `Vary:` headers.
- The default key consists of the Servlet Path (minus `pathInfo` and the query string).
- A "least recently used" list is maintained to evict cache entries if the maximum cache size is exceeded.
- Key generation concatenates the servlet path with key field values, if any are specified.

Caching Example

Here is an example cache element in the `sun-web.xml` file:

```
<cache max-capacity="8192" timeout="60">
   <cache-helper name="myHelper" class-name="MyCacheHelper"/>
   <cache-mapping>
      <servlet-name>myservlet</servlet name>
      <timeout name="timefield">120</timeout>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
   </cache-mapping>
   <cache-mapping>
      <url-pattern> /catalog/* </url-pattern>
      <!-- cache the best selling category; cache the responses to
        -- this resource only when the given parameters exist. cache
        -- only when the catalog parameter has 'lilies' or 'roses'
        -- but no other catalog varieties:
        -- /orchard/catalog?best&category='lilies'
        -- /orchard/catalog?best&category='roses'
        -- but not the result of
        -- /orchard/catalog?best&category='wild'
      -->
      <constraint-field name='best' scope='request.parameter'/>
      <constraint-field name='category' scope='request.parameter'>
        <value> roses </value>
        <value> lilies </value>
      </constraint-field>
      <!-- Specify that a particular field is of given range but the
        -- field doesn't need to be present in all the requests -->
      <constraint-field name='SKUnum' scope='request.parameter'>
        <value match-expr='in-range'> 1000 - 2000 </value>
      </constraint-field>
      <!-- cache when the category matches with any value other than
        -- a specific value -->
      <constraint-field name="category" scope="request.parameter>
```

```
        <value match-expr="equals" cache-on-match-failure="true">bogus</value>
      </constraint-field>
   </cache-mapping>
   <cache-mapping>
      <servlet-name> InfoServlet </servlet name>
      <cache-helper-ref>myHelper</cache-helper-ref>
   </cache-mapping>
</cache>
```

For more information about the `sun-web.xml` caching settings, see "Caching Elements" on page 130.

CacheHelper Interface

Here is the `CacheHelper` interface:

```
package com.sun.appserv.web.cache;
import java.util.Map
import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
/** CacheHelper interface is an user-extensible interface to customize:
 * a) the key generation b) whether to cache the response.
 */
public interface CacheHelper {
// name of request attributes
   public static final String ATTR_CACHE_MAPPED_SERVLET_NAME =
                           "com.sun.appserv.web.cachedServletName";
   public static final String ATTR_CACHE_MAPPED_URL_PATTERN =
                           "com.sun.appserv.web.cachedURLPattern";
public static final int TIMEOUT_VALUE_NOT_SET = -2;
/** initialize the helper
    * @param context the web application context this helper belongs to
    * @exception Exception if a startup error occurs
    */
   public void init(ServletContext context, Map props) throws Exception;
/** getCacheKey: generate the key to be used to cache this request
    * @param request incoming <code>HttpServletRequest</code> object
    * @returns the generated key for this requested cacheable resource.
    */
   public String getCacheKey(HttpServletRequest request);
/** isCacheable: is the response to given request cacheable?
    * @param request incoming <code>HttpServletRequest</code> object
    * @returns <code>true</code> if the response could be cached. or
    * <code>false</code> if the results of this request must not be cached.
    */
   public boolean isCacheable(HttpServletRequest request);
/** isRefreshNeeded: is the response to given request to be refreshed?
```

```
    * @param request incoming <code>HttpServletRequest</code> object
    * @returns <code>true</code> if the response needs to be refreshed.
    * or return <code>false</code> if the results of this request
    * don't need to be refreshed.
    */
   public boolean isRefreshNeeded(HttpServletRequest request);
/** get timeout for the cached response.
    * @param request incoming <code>HttpServletRequest</code> object
    * @returns the timeout in seconds for the cached response; a return
    * value of -1 means the response never expires and a value of -2 indicates
    * helper cannot determine the timeout (container assigns default timeout)
   */
public int getTimeout(HttpServletRequest request);
/**
    * Stop the helper from active use
    * @exception Exception if an error occurs
    */
   public void destroy() throws Exception;
}
```

CacheKeyGenerator Interface

The built-in default `CacheHelper` implementation allows web applications to customize the key generation. An application component (in a servlet or JSP) can set up a custom `CacheKeyGenerator` implementation as an attribute in the `ServletContext`.

The name of the context attribute is configurable as the value of the `cacheKeyGeneratorAttrName` property in the `default-helper` element of the `sun-web.xml` deployment descriptor. For more information, see "default-helper" on page 133.

Here is the `CacheKeyGenerator` interface:

```
package com.sun.appserv.web.cache;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
/** CacheKeyGenerator: a helper interface to generate the key that
 * is used to cache this request.
 *
 * Name of the ServletContext attribute implementing the
 * CacheKeyGenerator is configurable via a property of the
 * default-helper in sun-web.xml:
 * <default-helper>
 * <property
 * name="cacheKeyGeneratorAttrName"
 * value="com.acme.web.MyCacheKeyGenerator" />
 * </default-helper>
 *
```

```
 * Caching engine looks up the specified attribute in the servlet
 * context; the result of the lookup must be an implementation of the
 * CacheKeyGenerator interface.
 */
public interface CacheKeyGenerator {
/** getCacheKey: generate the key to be used to cache the
    * response.
    * @param context the web application context
    * @param request incoming <code>HttpServletRequest</code>
    * @returns key string used to access the cache entry.
    * if the return value is null, a default key is used.
    */
   public String getCacheKey(ServletContext context,
                             HttpServletRequest request);
}
```

Maximizing Servlet Performance

Consider the following guidelines for improving servlet performance:

- Increase the Java heap size to help garbage collection. The Java heap size can be defined by adjusting the values specified to the –Xms and –Xmx `jvm-` options in `server.xml`. For example, `<jvm-options>-Xms128m-Xmx256m</jvm-options>` sets the minimum Java heap size to 128 MB and 256 MB. For more information see, *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.
- Sun Java System Web Server 6.1 may occasionally run out of stack space if applications use deep recursion when a JIT (just-in-time) compiler is enabled, especially on UNIX platforms where the default stack size is small, or in any cases where very complex JSP pages are used.

  You can set the stack space using the `StackSize` directive in the `magnus.conf` file. For more information, see the *Sun Java System Web Server 6.1 SP11 NSAPI Programmer's Guide*.
- Create servlet sessions only if required as the Session ID generator employs cryptographically unique random number generation algorithms. While sessions are being created, limit the size of sessions and configure them depending on whether the application uses short-lived or long-lived sessions. For more information on optimal session configuration, see "Configuring the Web Application's Session Settings" in *Sun Java System Web Server 6.1 SP11 Performance Tuning, Sizing, and Scaling Guide*.
- Use servlet cache when possible. For more information, see "Caching Servlet Results" on page 37.
- Use precompiled JSPs if the JSPs do not change after deployment to the Web Server. The precompiled JSPs should include the `reload-interval` property setting in the `jsp-config` element in `sun-web.xml`. This eliminates time-consuming compilation and periodic checking by the Web Server.
- Reduce the number of directories in the classpath.

- Disable dynamic reloading.
- Disable the Java Security Manager.

◆ ◆ ◆ CHAPTER 3

Using JavaServer Pages

This chapter describes how to use JavaServer Pages (JSPs) as page templates in a Sun Java System Web Server web application.

This chapter has the following sections:

For information about internationalization issues for JSPs, see Appendix A, Internationalization Issues.

Introducing JSPs

JSPs are browser pages in HTML or XML. They also contain Java code, which enables them to perform complex processing, conditionalize output, and communicate with other application objects. JSPs in Sun Java System Web Server are based on the JSP 1.2 specification.

In a Sun Java System Web Server application, JSPs are the individual pages that make up an application. You can call a JSP from a servlet to handle the user interaction output. You can also use a JSP as an interaction destination as JSPs have the same application environment access as any other application component.

JSPs are made up of JSP elements and template data. Template data is anything not in the JSP specification, including text and HTML tags. For example, the minimal JSP requires no processing by the JSP engine and is a static HTML page.

The Sun Java System Web Server compiles JSPs into HTTP servlets the first time they are called (or they can be precompiled for better performance). This makes them available to the application environment as standard objects and enables them to be called from a client using a URL.

JSPs run inside the server's JSP engine, which is responsible for interpreting JSP-specific tags and performing the actions they specify in order to generate dynamic content. This content, along with any template data surrounding it, is assembled into an output page and is returned to the caller.

Creating JSPs

You create JSPs in basically the same way you create HTML files. You can use an HTML editor to create pages and edit the page layout. You make a page a JSP by inserting JSP-specific tags into the raw source code where needed, and by giving the file a `.jsp` extension.

JSPs that adhere to the JSP 1.2 specification follow XML syntax for the most part, which is consistent with HTML. For a summary of the JSP tags you can use, see "JSP Tag Libraries and Standard Portable Tags" on page 50

JSPs are compiled into servlets, so servlet design considerations also apply to JSPs. JSPs and servlets can perform the same tasks, but each excels at one task at the expense of the other. Servlets are strong in processing and adaptability. However, performing HTML output from them involves many cumbersome `println` statements that must be coded by hand. Conversely, JSPs excel at layout tasks because they are simply HTML files and can be created with HTML editors, though performing complex computational or processing tasks with them is awkward. For information about servlets, see Chapter 3, Using Servlets

Additional JSP design tips are described in the following sections:

Designing for Ease of Maintenance

Each JSP can call or include any other JSP. For example, you can create a generic corporate banner, a standard navigation bar, and a left-side column table of contents, where each element is in a separate JSP and is included for each page built. The page can be constructed with a JSP functioning as a frameset, dynamically determining the pages to load into each subframe. A JSP can also be included when the JSP is compiled into a servlet or when a request arrives.

Designing for Portability

JSPs can be completely portable between different applications and different servers. A disadvantage is that they have no particular application data knowledge, but this is only a problem if they require that kind of data.

One possible use for generic JSPs is for portable page elements such as navigation bars or corporate headers and footers, which are meant to be included in other JSPs. You can create a library of reusable generic page elements to use throughout an application, or even among several applications.

For example, the minimal generic JSP is a static HTML page with no JSP-specific tags. A slightly less minimal JSP might contain some Java code that operates on generic data such as printing the date and time, or that makes a change to the page's structure based on a standard value set in the request object.

Handling Exceptions

If an uncaught exception occurs in a JSP file, Sun Java System Web Server generates an exception, usually a 404 or 500 error. To avoid this problem, set the `errorPage` attribute of the <%@ page%> tag.

Compiling JSPs: The Command-Line Compiler

Sun Java System Web Server provides the following ways of compiling JSP 1.2-compliant source files into servlets:

- JSPs are automatically compiled at runtime.
- The `jspc` command-line tool, described in this section, allows you to precompile JSPs at the command line.

To allow the JSP container to pick up the precompiled JSPs from a JAR file, you must disable dynamic reloading of JSPs. To do this, set the `reload-interval` property to `-1` in the `jsp-config` element of the `sun-web.xml` file. For more information, see "JSP Elements" on page 141

The `jspc` command-line tool is located under install_dir/`bin/https/bin` (make sure this directory is in your path). The format of the `jspc` command is as follows:

```
jspc [options] file_specifier
```

The following table shows what file_specifier can be in the `jspc` command. The left column lists file specifiers, and the right column lists descriptions of those file specifiers.

TABLE 3–1 File Specifiers for the jspc Command

File Specifier	Description
files	One or more JSP files to be compiled.
`-webapp` *dir*	A directory containing a web application. All JSPs in the directory and its subdirectories are compiled. You cannot specify a WAR, JAR, or ZIP file; you must first extract it to an open directory structure.

The following table shows the basic *options* for the `jspc` command. The left column lists the option, and the right column describes what the option does.

TABLE 3–2 Basic jspc Options

Option	Description
`-q`	Enables quiet mode (same as `-v0`). Only fatal error messages are displayed.
`-d` *dir*	Specifies the output directory for the compiled JSPs. Package directories are automatically generated based on the directories containing the uncompiled JSPs. The default top-level directory is the directory from which `jspc` is invoked.
`-p` *name*	Specifies the name of the target package for all specified JSPs, overriding the default package generation performed by the `-d` option.
`-c` *name*	Specifies the target class name of the first JSP compiled. Subsequent JSPs are unaffected.
`-uribase` *dir*	Specifies the URI directory to which compilations are relative. Applies only to explicitly declared JSP files. This is the location of each JSP file relative to the `uriroot`. If this cannot be determined, the default is `/`.
`-uriroot` *dir*	Specifies the root directory against which URI files are resolved. Applies only to explicitly declared JSP files. If this option is not specified, all parent directories of the first JSP page are searched for a `WEB-INF` subdirectory. The closest directory to the JSP page that has one is used. If none of the JSP's parent directories have a `WEB-INF` subdirectory, the directory from which `jspc` is invoked is used.
`-genclass`	Generates class files in addition to Java files.

The following table shows the advanced *options* for the `jspc` command. The left column lists the option, and the right column describes what the option does.

TABLE 3–3 Advanced jspc Options

Option	Description
`-v[`*level*`]`	Enables verbose mode. The *level* is optional; the default is `2`. Possible *level* values are: ■ `0` - fatal error messages only ■ `1` - error messages only ■ `2` - error and warning messages only ■ `3` - error, warning, and informational messages ■ `4` - error, warning, informational, and debugging messages
`-mapped`	Generates separate `write` calls for each HTML line and comments that describe the location of each line in the JSP file. By default, all adjacent `write` calls are combined and no location comments are generated.
`-die[`*code*`]`	Causes the JVM to exit and generates an error return *code* if a fatal error occurs. If the *code* is absent or unparsable it defaults to `1`.
`-webinc` *file*	Creates partial servlet mappings for the `-webapp` option, which can be pasted into a `web.xml` file.
`-webxml` *file*	Creates an entire `web.xml` file for the `-webapp` option.
`-ieplugin` *class_id*	Specifies the Java plugin COM class ID for Internet Explorer. Used by the `<jsp:plugin>` tags.
`-deprecatedjavac`	Forces compilation of generated servlets using the deprecated `sun.tools.javac.Main`.

For example, this command compiles the `hello` JSP file and writes the compiled JSP under `hellodir`:

```
jspc -d hellodir -genclass hello.jsp
```

This command compiles all of the JSP files in the web application under `webappdir` into class files under `jspclassdir`:

```
jspc -d jspclassdir -genclass -webapp webappdir
```

To use either of these precompiled JSPs in a web application, put the classes under `hellodir` or `jspclassdir` into a JAR file, place the JAR file under `WEB-INF/lib`, and set the `reload-interval` property to `-1` in the `sun-web.xml` file.

Package Names Generated by the JSP Compiler

When a JSP is compiled, a package is created for it. The package name starts with `_jsps`, and each path name component of the JSP is prefixed with an underscore. For example, the generated package name for `/myjsps/hello.jsp` would be `_jsps._myjsps`.

Other JSP Configuration Parameters

For information about the various JSP configuration parameters, see "jsp-config" on page 141 The JSP compiler uses the default values for parameters that are not included in the file.

Debugging JSPs

When you use Sun Java Studio to debug JSPs, you can set breakpoints in either the JSP code or the generated servlet code, and you can switch between them and see the same breakpoints in both.

For information about setting up debugging in Sun Java System Studio, see "Using Sun Java Studio for Debugging" on page 153.

JSP Tag Libraries and Standard Portable Tags

Sun Java System Web Server supports tag libraries and standard portable tags. For more information about tag libraries, see the JSP 1.2 specification at:

`http://java.sun.com/products/jsp/download.html`

For a handy summary of JSP 1.2 tag syntax, see the following PDF file:

`http://java.sun.com/products/jsp/syntax/1.2/card12.pdf`

JSP Cache Tags

JSP cache tags allow you to cache JSP page fragments within the Java engine. Each can be cached using different cache criteria. For example, suppose you have page fragments to view stock quotes, weather information, and so on. The stock quote fragment can be cached for 10 minutes, the weather report fragment for 30 minutes, and so on.

For more information about response caching as it pertains to servlets, see "Caching Servlet Results" on page 37

JSP caching uses the custom tag library support provided by JSP 1.2. JSP caching is implemented by a tag library packaged into the *install_dir*`/bin/https/jar/webserv-rt.jar`

file, which is always on the server classpath. The `sun-web-cache.tld` tag description file can be found in *install_dir*`/bin/https/tlds` directory and can be copied into the `WEB-INF` directory of your web application.

You can add a taglib mapping in the `web.xml` of your application as follows:

```
<taglib>
  <taglib-uri>/com/sun/web/taglibs/cache</taglib-uri>
  <taglib-location>/WEB-INF/sun-web-cache.tld</taglib-location>
</taglib>
```

You can then refer to these tags in your JSP files as follows:

```
<%@ taglib prefix="mypfx" uri="/com/sun/web/taglibs/cache" %>
```

Subsequently, the cache tags are available as `<mypfx:cache>` and `<mypfx:flush>`.

The tags are as follows:

- "cache" on page 51
- "flush" on page 53

cache

The `cache` tag allows you to cache fragments of your JSP pages. It caches the body between the beginning and ending tags according to the attributes specified. The first time the tag is encountered, the body content is executed and cached. Each subsequent time it is run, the cached content is checked to see if it needs to be refreshed and if so, it is executed again, and the cached data is refreshed. Otherwise, the cached data is served.

Attributes

The following table describes attributes for the `cache` tag. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 3–4 cache Attributes

Attribute	Default	Description
`key`	*ServletPath_Suffix*	(optional) The name used by the container to access the cached entry. The cache key is suffixed to the servlet path to generate a key to access the cached entry. If no key is specified, a number is generated according to the position of the tag in the page.

TABLE 3–4 cache Attributes *(Continued)*

Attribute	Default	Description
`timeout`	`60s`	(optional) The time in seconds after which the body of the tag is executed and the cache is refreshed. By default, this value is interpreted in seconds. To specify a different unit of time, add a suffix to the timeout value as follows: `s` for seconds, `m` for minutes, `h` for hours, `d` for days. For example, `2h` specifies two hours.
`nocache`	`false`	(optional) If set to `true`, the body content is executed and served as if there were no cache tag. This offers a way to programmatically decide whether the cached response should be sent or whether the body must be executed, though the response is not cached.
`refresh`	`false`	(optional) If set to `true`, the body content is executed and the response is cached again. This lets you programmatically refresh the cache immediately, regardless of the timeout setting.

Example

The following example represents a cached JSP page:

```
<%@ taglib prefix="mypfx" uri="/com/sun/web/taglibs/cache" %>

<%
   String cacheKey = null;
    if (session != null)
      cacheKey = (String)session.getAttribute("loginId");

   // check for nocache
   boolean noCache = false;
   String nc = request.getParameter("nocache");
   if (nc != null)
      noCache = "true";

   // force reload
   boolean reload=false;
   String refresh = request.getParameter("refresh");
   if (refresh != null)
      reload = true;
 %>

 <mypfx:cache key="<%= cacheKey %>" nocache="<%= noCache %>" refresh="
<%= reload %>" timeout="10m">
<%
    String page = request.getParameter("page");
   if (page.equals("frontPage") {
       // get headlines from database
```

```
    } else {
       .....
 %>
</mypfx:cache>

<mypfx:cache timeout="1h">
<h2> Local News </h2>
<%
   // get the headline news and cache them
%>
</mypfx:cache>
```

flush

Forces the cache to be flushed. If a key is specified, only the entry with that key is flushed. If no key is specified, the entire cache is flushed.

Attributes

The following table describes attributes for the `flush` tag. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 3–5 flush Attributes

Attribute	Default	Description
`key`	*ServletPath_Suffix*	(optional) The name used by the container to access the cached entry. The cache key is suffixed to the servlet path to generate a key to access the cached entry. If no key is specified, a number is generated according to the position of the tag in the page.

Examples

To flush the entry with `key="foobar"`:

```
<mypfx:flush key="foobar">
```

To flush the entire cache:

```
<% if (session != null && session.getAttribute("clearCache") != null) { %>
     <mypfx:flush >
 <% } %>
```

JSP Search Tags

Sun Java System Web Server includes a set of JSP tags that can be used to customize the search query and search results pages in the search interface. This section describes the tags and how they're used. For more information about using the search feature, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

The search tag library is packaged into the *install_dir*`/bin/https/jar/webserv-rt.jar` file, which is always on the server classpath. The `sun-web-search.tld` tag description file can be found in the *install_dir*`/bin/https/tlds` directory, and can be copied into the `WEB-INF` directory of your web application.

The search tags are as follows:

Note – The Sun Java System Web Server search feature is i18n compliant, and supports multiple character encoding schemes in the same collection. Custom JSPs that expose search can be in any encoding.

<searchForm>

Function

Constructs an HTML form that contains default and hidden form elements such as the current search result index and number of records per page by default. The default names for the form, index, and number of records are `searchform`, `si`, and `ns`.

Attributes

Name. Specifies the name of the form. The default is `searchform`. The name of a form is the identifier for all other tags.

Action. (optional) Specifies the form action.

Method. (optional) Specifies the method of submission, GET or POST. The default is GET.

elemStart. (optional) Specifies the name of the hidden `Start` element. If not specified, the default "si" will be used.

Start. (optional) An integer indicating the starting index for displaying the search result. The default is 1.

elemNumShown. (optional) The name of the `nShown` element. If not specified, the default "ns" is used.

numShown. (optional) An integer indicating the number of results to be shown per page. The value of the attribute will be retrieved by requesting parameter `elemNumShown`. The default is 10.

Usage

```
<s1ws:form action="results.jsp" >
...
</s1ws:searchForm>
```

This creates an HTML form tag `<form name="searchform" action="results.jsp" method="GET">` along with two hidden inputboxes:

- A hidden inputbox for starting index named "si" with a value from the "si" parameter or default 1, and
- A hidden inputbox for number of records per page named "ns" with a value from the `ns` parameter or default 20.

<CollElem>

Function

Creates a hidden inputbox or select box, or multiple checkboxes depending on the attribute input. If there is only one collection, the tag creates a hidden inputbox by default.

Attributes

Name. Specifies the name of the form element created. The default is "c."

items. (optional) A comma-delimited string representing search collections available. The tag retrieves all collections available on the server if the attribute is empty.

Type. (optional) The type of form element used for displaying collections. Valid options are `hidden`, `select`, and `checkbox`. If there is only one collection, the default value is `hidden` else the default value is `checkbox`.

Rows. (optional) Represents size if type is `select`, or number of rows if checkboxes. The default behavior is to satisfy the `Cols` attribute first. That is, the collections will be listed in columns as specified by the `Cols` attribute.

Cols. Represents number of columns and is only required if type is `checkbox`. If `Cols` and `Rows` are not specified, the collections will be listed horizontally, that is, in one row.

Defaults. Specifies a comma-delimited string containing 1s or 0s indicating the selection status of the search collections. An item is selected if 1, and not selected if 0. If there is a form action, these values will be retrieved from the form elements.

cssClass. (optional) The class name that will be used in every HTML tag created in this tag. This is particularly useful when the type is `checkbox`, since an HTML table will be used for the layout. See the sample code below for details.

Usage

```
<s1ws:collElem type="checkbox" cols="2" values="1,0,1,0" cssClass="body" >
```

This creates checkboxes in 2 columns with a default name "c," with the first and third items selected. Fonts and any other HTML styles are defined in css classes "body," which includes `tr.body`, `td.body`, and `input.body`.

<collection>

Function

Retrieves the name of search collections on the server, and iterates through and passes each of them to the `collectionitem` tag. Users may choose to use this tag along with collection item tags so they can write their own HTMLs.

Attributes

items. (optional) A comma-delimited string representing the search collections available. The tag retrieves all collections available on the server if the attribute is empty.

Usage

```
<table border=0>
<s1ws:collection>
<tr><td><input type=checkbox name="c"
```

```
value="<s1ws:collItem property="name" >">
<s1ws:collItem property="display name" >
</td></tr>
</s1ws:collection>
</table>
```

The above code will display all collections with checkboxes.

```
<select name=elementname>
<s1ws:collection>
<option value="<s1ws:collItem property=\"name\" >">
<s1ws:colleItem property="display name" >
</s1ws:collection>
</select>
```

This iterates through the available collections and passes each item to the collection item tag, which in turn displays the name and display name of the item.

<colllitem>

Function

Displays the name and label of one collection item. Must be used inside the `collection` tag.

Attributes

property. Specifies a keyword indicating which property the tag should output. Valid inputs include `name`, `display name`, and `description`. Default is `name`.

Usage

```
<select name=elementname>
<s1ws:collection>
<option value="<s1ws:collItem property=\"name\" >">
<s1ws:collItem property="display name" >
</s1ws:collection>
</select>
```

This iterates through the available collections and passes each item to the collection item tag, which in turn displays the name and display name of the item.

<queryBox>

Function

Creates an inputbox.

Attributes

name. (optional) The name of the inputbox. The default is "qt."

default. (optional) The default value for the query box. If the form is submitted, its value will be set using what has been submitted.

size. (optional) The size of the inputbox. The default is 50.

maxlength. (optional) The `maxlength` of the inputbox. The default is 50.

cssClass. (optional) The CSS class.

Usage

```
<s1ws:queryBox size="30" >
```

This creates an inputbox with default name "qt" and `size=30`.

<submitButton>

Function

Creates a submit button.

Attributes

name. (optional) The name of the button. The default is "sb."

default. (optional) The default value of the button, which will be "search."

cssClass. (optional) The CSS class name.

style. The CSS style.

image. The optional image for the button.

Usage

```
<s1ws:submitButton cssClass="navBar" style="padding: 0px; margin: 0px; width: 50px">
```

This creates a submit button with default name "sb."

<formAction>

Function

Handles form action. It retrieves all elements on the search form. It validates that there is at least one collection selected and the query is not empty. After validation, it passes the values on to search and results tags as parent or through the page context.

Attributes

formId. Specifies the name of the form. If not specified, the default form "searchForm" will be used.

elemColl. (optional) The name of the `Collection` element. The default name "c" is used.

elemQuery. (optional) The name of the `Start` element. The default name "qt" is used.

elemStart. (optional) The name of the `Start` element. The default name "si" is used.

elemNumShown. (optional) The name of the `numShown` element. The default name "ns" is used.

Usage

```
<s1ws:formAction >
```

<formSubmission>

Function

Tests if the form submission is successful.

Attributes

formId. Specifies the name of the form in question. It must be the same as that for `<formAction>`.

success. Indicates if the form submission is successful. The values `true` or `yes` represents successful action. All other inputs will be rendered as failure.

Usage

```
<s1ws:formSubmission success="true" >
         <s1ws:search>
         ...
   </s1ws:formSubmission>
```

<formActionMsg>

Function

Prints out an error message from `formAction`, if any.

Attributes

formId. (optional) Specifies the name of the form in question. If not specified, the default id is "searchForm."

elem. (optional) Specifies the name of the element. Valid inputs are "query" and "collection." When specified, the tag returns an error message, if any, regarding the element. Otherwise, it prints out all of the error messages generated.

Usage

```
<FormActionMsg elem="query">
```

This tag will display a message "query text not specified" if a query is not submitted.

The message is printed from the form action where the tag is placed.

<Search>

Function

Executes a search query and returns search results. The search tag retrieves a query string and collection(s) from either a `form` parent tag or the `query` and `collection` attributes. The search results are saved in the page context with a page or session scope.

Attributes

formId. Specifies the name of the form used for the search. The default form will be used if the attribute is left empty. If this tag is used without a form, this attribute must be set to provide an identifier for the `resultIterate` tag.

Collection. (optional) A comma-delimited string representing collection(s) used for a search. If there is a form action, this attribute is ignored and the values are retrieved by requesting the `collection` element.

Query. (optional) Specifies the query text. If not provided, it is retrieved from the `query` element.

Scope. Specifies an integer indicating the scope of the search results. The value 1, which is the default, indicates page scope, and 2 indicates session scope.

Usage

```
<s1ws:search >
```

This search tag uses the default parameters and executes a search. The search results will be saved in `pageContext` with a page scope.

```
<s1ws:search Collection="col1, col2" Query="Java Web Service" scope="2" >
```

This search tag will execute a search in `col1` and `col2` using "`Java Web Service`" as the query string. The search results will be saved in `pageContext` with a session scope.

<resultIteration>

Function

Retrieves the search results from either the parent search tag or the page context. The tag is responsible for iterating through the results and passing the `searchitems` to the item tags.

Attributes

formId. Specifies the name of the form associated. The default form will be used if the attribute is left empty. If this tag is used without a form, this attribute must be set as a reference to the search tag.

Start. Specifies an integer representing the starting position in the search results. The default is 0. The value is retrieved from the parent `<formAction>` tag or `pageContext`, or the parameter value.

numShown. Specifies an integer indicating the number of results to be shown in one page. The default is 20. The value is retrieved from the parent `<formAction>` tag or `pageContext`.

<Item>

Function

Retrieves a `searchitem` from the `Results` parent tag and outputs its properties as specified by the `property` attribute.

Attributes

Property. Specifies a case-sensitive string representing field names in a search item, such as title, number, score, filename, URL, size, and date.

<resultStat>

Function

Returns numbers indicating number of records returned and the range currently displayed.

Attributes

formId. Specifies the name of the form associated. The default form will be used if the attribute is left empty. If this tag is used without a form, this attribute must be set as a reference to the search tag.

type. Specifies the type of statistics data. Valid inputs are "start," "end," "range" (for example, 1-20), and "total."

<resultNav>

Function

Creates a navigation bar.

Attributes

formId. Specifies the name of the form associated. The default form will be used if the attribute is left empty. If this tag is used without a form, this attribute must be set as a reference to the search tag.

type. Specifies the type of navigation bar. Valid inputs are "full," "previous," and "next." A full navigation bar looks like this: `previous 1 2 3 4 5 6 7 8 9 10 next`, where `5` is currently selected. The number of "page number" links is determined by the "offset" attribute and the number of pages available.

caption. Only necessary if type is `previous` or `next` and the default text is not wanted. Caption can be any HTML.

offset. Specifies the number of page links around the selected page. For example, if `offset=2`, the sample bar above would look like this: `previous 3 4 5 6 7 next` . Only required for type "full."

◆ ◆ ◆ CHAPTER 4

Session Managers

Session objects maintain state and user identity across multiple page requests over the normally stateless HTTP protocol. A session persists for a specified period of time, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The server can maintain a session either by using cookies or by rewriting URLs. Servlets can access the session objects to retrieve state information about the session.

This chapter describes sessions and session managers, and has the following sections:

Introducing Sessions

The term *user session* refers to a series of user application interactions that are tracked by the server. Sessions are used for maintaining user specific state, including persistent objects (such as handles to database result sets) and authenticated user identities, among many interactions. For example, a session could be used to track a validated user login followed by a series of directed activities for a particular user.

The session itself resides in the server. For each request, the client transmits the session ID in a cookie or, if the browser does not allow cookies, the server automatically writes the session ID into the URL.

The Sun Java System Web Server supports the servlet standard session interface, called `HttpSession`, for all session activities. This interface enables you to write portable, secure servlets.

This section includes the following topics:

- "Sessions and Security" on page 64

Note – As of Sun Java System Web Server 6.1, form-login sessions are no longer supported. You can use single sign-on sessions instead.

Sessions and Cookies

A cookie is a small collection of information that can be transmitted to a calling browser, which retrieves it on each subsequent call from the browser so that the server can recognize calls from the same client. A cookie is returned with each call to the site that created it, unless it expires.

Sessions are maintained automatically by a session cookie that is sent to the client when the session is first created. The session cookie contains the session ID, which identifies the client to the browser on each successive interaction. If a client does not support or allow cookies, the server rewrites the URLs where the session ID appears in the URLs from that client.

You can configure whether and how sessions use cookies. See "session-properties" on page 124 and "cookie-properties" on page 125 elements in the `sun-web.xml` file, described in Chapter 7, Deploying Web Applications.

Sessions and URL Rewriting

There are two situations in which the Sun Java System Web Server plugin performs implicit URL rewriting:

- When a response comes back from the Sun Java System Web Server. If implicit URL rewriting has been chosen, the plugin rewrites the URLs in the response before passing the response to the client.
- When the request given by a client need not be sent to the Sun Java System Web Server and can be served on the web server side. Such requests may occur in the middle of a session and the response may need to be rewritten.

You can configure whether sessions use URL rewriting. See the "session-properties" on page 124 element in the `sun-web.xml` file, described in Chapter 7, Deploying Web Applications.

Sessions and Security

The Sun Java System Web Server security model is based on an authenticated user session. Once a session has been created, the application user is authenticated (if authentication is used) and logged in to the session. Each interaction step from the servlet that receives a request does two things: generates content for a JSP to format the output, and checks if the user is properly authenticated.

Additionally, you can specify that a session cookie is only passed on a secured connection (that is, HTTPS), so the session can only remain active on a secure channel.

For more information about security, see Chapter 6, Securing Web Applications.

How to Use Sessions

To use a session, first create a session using the `HttpServletRequest` method `getSession()`. Once the session is established, examine and set its properties using the provided methods. If desired, set the session to time out after being inactive for a defined time period, or invalidate it manually. You can also bind objects to the session, which store them for use by other components.

This section includes the following topics:

Creating or Accessing a Session

To create a new session or gain access to an existing session, use the `HttpServletRequest` method `getSession()`, as shown in the following example:

```
HttpSession mySession = request.getSession();
```

`getSession()` returns the valid session object associated with the request, identified in the session cookie that is encapsulated in the request object. Calling the method with no arguments creates a session if one does not exist that is associated with the request. Additionally, calling the method with a Boolean argument creates a session only if the argument is `true`.

The following example shows the `doPost()` method from a servlet that only performs the servlet's main functions, if the session is present. Note that the false parameter to `getSession()` prevents the servlet from creating a new session if one does not already exist:

```
public void doPost (HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException
 {
        if ( HttpSession session = req.getSession(false) ) {
           // session retrieved, continue with servlet operations
        }
        else{
           // no session, return an error page
```

```
        }
}
```

Note – The `getSession()` method should be called before anything is written to the response stream.

For more information about `getSession()`, see the Java Servlet 2.3 specification.

Examining Session Properties

Once a session ID has been established, use the methods in the `HttpSession` interface to examine session properties, and the methods in the `HttpServletRequest` interface to examine request properties that relate to the session.

The following table shows the methods used to examine session properties. The left column lists `HttpSession` methods, and the right column lists descriptions of these methods.

TABLE 4–1 HttpSession Methods

HttpSession Method	Description
`getCreationTime()`	Returns the session time in milliseconds since January 1, 1970, 00:00:00 GMT.
`getId()`	Returns the assigned session identifier. An HTTP session's identifier is a unique string that is created and maintained by the server.
`getLastAccessedTime()`	Returns the last time the client sent a request carrying the assigned session identifier (or -1 if it's a new session) in milliseconds since January 1, 1970, 00:00:00 GMT.
`isNew()`	Returns a Boolean value indicating if the session is new. It's a new session if the server has created it and the client has not sent a request to it. This means the client has not acknowledged or joined the session and may not return the correct session identification information when making its next request.

For example:

```
String mySessionID = mySession.getId();
     if ( mySession.isNew() ) {
         log.println(currentDate);
        log.println("client has not yet joined session " + mySessionID);
     }
```

The following table shows the methods used to examine servlet request properties. The left column lists `HttpServletRequest` methods, and the right column lists descriptions of these methods.

TABLE 4–2 HttpServletRequest Methods

HttpServletRequest Method	Description
`getRequestedSessionId()`	Returns the session ID specified with the request. This may differ from the session ID in the current session if the session ID given by the client is invalid and a new session was created. Returns null if the request does not have a session associated with it.
`isRequestedSessionIdValid()`	Checks if the request is associated to a currently valid session. If the session requested is not valid, it is not returned through the `getSession()` method.
`isRequestedSessionIdFromCookie()`	Returns `true` if the request's session ID provided by the client is a cookie, or `false` otherwise.
`isRequestedSessionIdFromURL()`	Returns `true` if the request's session ID provided by the client is a part of a URL, or `false` otherwise.

For example:

```
if ( request.isRequestedSessionIdValid() ) {
        if ( request.isRequestedSessionIdFromCookie() ) {
           // this session is maintained in a session cookie
         }
        // any other tasks that require a valid session
      } else {
        // log an application error
      }
```

Binding Data to a Session

You can bind objects to sessions to make them available across multiple user interactions.

The following table shows the `HttpSession` methods that provide support for binding objects to the session object. The left column lists `HttpSession` methods, and the right column lists descriptions of these methods.

TABLE 4–3 HttpSession Methods

HttpSession Method	Description
`getAttribute()`	Returns the object bound to a given name in the session, or null if there is no such binding.
`getAttributeNames()`	Returns an array of names of all attributes bound to the session.
`setAttribute()`	Binds the specified object into the session with the given name. Any existing binding with the same name is overwritten. For an object bound into the session to be distributed it must implement the serializable interface.
`removeAttribute()`	Unbinds an object in the session with the given name. If there is no object bound to the given name, this method does nothing.

Binding Notification with HttpSessionBindingListener

Some objects require you to know when they are placed in or removed from a session. To obtain this information, implement the `HttpSessionBindingListener` interface in those objects. When your application stores or removes data with the session, the servlet engine checks whether the object being bound or unbound implements `HttpSessionBindingListener`. If it does, the Sun Java System Web Server notifies the object under consideration, through the `HttpSessionBindingListener` interface, that it is being bound into or unbound from the session.

Invalidating a Session

Direct the session to invalidate itself automatically after being inactive for a defined time period. Alternatively, invalidate the session manually with the `HttpSession` method `invalidate()`.

Invalidating a Session Manually

To invalidate a session manually, simply call the following method:

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

All objects bound to the session are removed.

Setting a Session Timeout

Session timeout is set using the `session-timeout` element in the `web.xml` deployment descriptor file. For more information, see the Java Servlet 2.3 specification.

Session Managers

A session manager automatically creates new session objects whenever a new session starts. In some circumstances, clients do not join the session, for example, if the session manager uses cookies and the client does not accept cookies.

In compliance with the Java Servlet 2.3 API specification, session managers allow for session scoping only by web application.

To successfully restore the state of session attributes, all such attributes must implement the `java.io.Serializable` interface. You can configure the session manager to enforce this restriction by including the `distributable` element in the web application deployment descriptor file, `web.xml`.

Sun Java System Web Server 6.1 provides the following session management options, which are described in this section:

- "StandardManager" on page 69, the default session manager
- "PersistentManager" on page 70, a provided session manager that uses a persistent data store
- "IWSSessionManager" on page 72, a provided session manager that allows backward compatibility with any custom session managers you may have created using Sun Java System Web Server 6.0
- "MMapSessionManager (UNIX Only)" on page 78, a provided persistent memory map (`mmap`) file-based session manager that works in both single-process and multi-process mode

---

Note – The session manager interface is unstable. An unstable interface may be experimental or transitional, and thus may change incompatibly, be removed, or be replaced by a more stable interface in the next release.

StandardManager

The StandardManager is the default session manager.

Enabling StandardManager

You may want to specify StandardManager explicitly to change its default parameters. To do so, edit the `sun-web.xml` file for the web application as in the following example. Note that `persistence-type` must be set to `memory`.

```
<sun-web-app>
          ...
        <session-config>
```

```
            <session-manager persistence-type=”memory”>
               <manager-properties>
                  <property name="reapIntervalSeconds" value="20" >
               </manager-properties>
            </session-manager>
             ...
         </session-config>
         ...
</sun-web-app>
```

For more information about the `sun-web.xml` file, Chapter 7, Deploying Web Applications.

Manager Properties for StandardManager

The following table describes `manager-properties` properties for the StandardManager session manager. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 4–4 manager-properties Properties for StandardManager

Property Name	Default Value	Description
`reapIntervalSeconds`	`60`	Specifies the number of seconds between checks for expired sessions. Setting this value lower than the frequency at which session data changes is recommended. For example, this value should be as low as possible (1 second) for a hit counter servlet on a frequently accessed web site, or you could lose the last few hits each time you restart the server.
`maxSessions`	-1	Specifies the maximum number of active sessions, or `-1` (the default) for no limit.
`sessionFilename`	`SESSIONS`	Specifies the absolute or relative path name of the file in which the session state is preserved between application restarts, if preserving the state is possible. A relative path name is relative to the temporary directory for this web application.

PersistentManager

The PersistentManager is another session manager provided with Sun Java System Web Server. For session persistence, PersistentManager can use a file, to which each session is serialized. You can also create your own persistence mechanism.

Enabling PersistentManager

You may want to specify PersistentManager explicitly to change its default parameters. To do so, edit the `sun-web.xml` file for the web application as in the following example. Note that `persistence-type` must be set to `file`.

```
<sun-web-app>
        ...
        <session-config>
           <session-manager persistence-type=”file”>
              <manager-properties>
                 <property name=reapIntervalSeconds value=20 >
              </manager-properties>
              <store-properties>
                 <property name=directory value=sessions >
              </store-properties>
           </session-manage>
           ...
        </session-config>
        ...
</sun-web-app>
```

For more information about the `sun-web.xml` file, Chapter 7, Deploying Web Applications.

Manager Properties for PersistentManager

The following table describes `manager-properties` properties for the PersistentManager session manager. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 4–5 manager-properties Properties for PersistentManager

Property Name	Default Value	Description
`reapIntervalSeconds`	`60`	Specifies the number of seconds between checks for expired sessions. Setting this value lower than the frequency at which session data changes is recommended. For example, this value should be as low as possible (1 second) for a hit counter servlet on a frequently accessed web site, or you could lose the last few hits each time you restart the server.
`maxSessions`	`-1`	Specifies the maximum number of active sessions, or `-1` (the default) for no limit.

IWSSessionManager

The IWSSessionManager ensures backward compatibility with any custom session managers that you may have created on Sun Java System Web Server 6.0.

IWSSessionManager works in both single-process and multi-process mode. It can be used for sharing session information across multiple processes possibly running on different machines. The `MaxProcs` directive in the `magnus.conf` file determines whether the server is running in single-process or multi-process mode.

Note – If the value of `MaxProcs` is higher than `1` and no session manager is configured, then by default the session manager used is the IWSSessionManager with file-based persistence.

For more information on the `MaxProcs` directive, see the *Sun Java System Web Server 6.1 SP11 NSAPI Programmer's Guide*.

For session persistence, IWSSessionManager can use a database or a distributed file system (DFS) path that is accessible from all servers in a server farm. Each session is serialized to the database or distributed file system. You can also create your own persistence mechanism.

If Sun Java System Web Server is running in single-process mode, then by default, no session persistence mode is defined and therefore sessions are not persistent.

If Sun Java System Web Server is running in multi-process mode, sessions are persistent by default. If a persistence mode is not defined, IWSSessionManager uses a DFS.

Multi-process mode is supported only on UNIX platforms. All multi-process mode features of IWSSessionManager are ignored on Windows.

Enabling IWSSessionManager

You may want to enable IWSSessionManager to change its default parameters. You can also enable IWSSessionManager for a particular context if the server is running in single-process mode. To do so, edit the `sun-web.xml` file for the web application as in the following example. Note that `persistence-type` must be set to `s1ws60`.

```
<sun-web-app>
   ...
   <session-config>
    <session-manager persistence-type="s1ws60">
     <manager-properties>
      <property name="classname" value="com.iplanet.server.http.
                                        session.IWSSessionManager">
      // other manager-related properties
     </manager-properties>
```

```
    </session-manager>
     ...
   </session-config>
   ...
</sun-web-app>
```

In the case of persistent sessions:

```
<sun-web-app>
   ...
   <session-config>
    <session-manager persistence-type="s1ws60">
     <manager-properties>
      <property name="classname" value="com.iplanet.server.http.
                                        session.IWSSessionManager">
      // other manager-related properties
     </manager-properties>
     <store-properties>
       <property name="classname" value="com.iplanet.server.http.
                                                session.FileStore">
       <property name="directory" value="<directory name to
                        store the persistant sessions>">
       // other store-related properties
     </store-properties>
    </session-manager>
     ...
   </session-config>
   ...
</sun-web-app>
```

For more information about the `sun-web.xml` file, Chapter 7, Deploying Web Applications.

Manager Properties for IWSSessionManager

The following table describes `manager-properties` properties for the IWSSessionManager session manager. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 4–6 manager-properties Properties for IWSSessionManager

Property Name	Default Value	Description
`maxSessions`	`1000`	The maximum number of sessions maintained by the session manager at any given time. The session manager refuses to create any more new sessions if there are already `maxSessions` number of sessions present at that time.

TABLE 4–6 manager-properties Properties for IWSSessionManager *(Continued)*

Property Name	Default Value	Description
`timeOut`	`1800`	The amount of time in seconds after a session is accessed by the client before the session manager destroys it. Those sessions that haven't been accessed for at least `timeOut` seconds are destroyed by the reaper method. If `session-timeout` is specified in `web.xml`, it overrides this `timeOut` parameter value.
`reapInterval`	`600`	The amount of time in seconds that the `SessionReaper` thread sleeps before calling the reaper method again.
`maxLocks`	`10`	The number of cross-process locks to use for synchronizing access to individual sessions across processes. The default value is used if the value `0` is specified. This parameter is ignored in single-process mode.
`session-data-store`		The name of the class that determines the means of session persistence. The classes supplied with Sun Java System Web Server are: ■ `com.iplanet.server.http.session.JdbcStore` ■ `com.iplanet.server.http.session.FileStore` If you do not specify the `session-data-store` parameter, sessions are not persistent in single-process mode, and `FileStore` is the default in multi-process mode. The `JdbcStore` and `FileStore` classes are subclasses of the `session-data-store` class. You can create your own class that implements session persistence by extending `SessionDataStore`.
`session-failover-enabled`		Specifies whether sessions are reloaded from the persistent store for every request, and always forced to `true` in multi-process mode. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` or `FileStore` class.

TABLE 4–6 manager-properties Properties for IWSSessionManager *(Continued)*

Property Name	Default Value	Description
`session-data-dir`	(this should all be on one line) `server_root/server_id /SessionData/virtual_ server_id/web_app_URI`	The directory in which session data for all servers and web applications is kept. Applicable only if the `session-data-store` parameter is set to the `FileStore` class.
`provider`		The JDBC driver (the default is `sun.jdbc.odbc.JdbcOdbcDriver`). For more information about the JDBC API, see the following web site: `http://java.sun.com/ products/jdbc/index.jsp` Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`url`	`jdbc:odbc:LocalServer`	Specifies the data source. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`table`	`sessions`	Name of the SQL table that store sessions. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`username`	none	The login user name for the database. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`password`	none	The login password for the database. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`reaperActive`	`true`	Tells the session manager whether to run session reaper to remove expired sessions from the database when `true`, which is the default value. It is recommended that only one server in the cluster be running the reaper. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.

TABLE 4–6 manager-properties Properties for IWSSessionManager *(Continued)*

Property Name	Default Value	Description
`accessTimeColumn`	`AccessTime`	The name of the column that holds the last access time in minutes. The SQL type is NUMERIC(9). Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`timeOutColumn`	`TimeOut`	The name of the column that holds the session timeout in minutes. The SQL type is NUMERIC(9). Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`sessionIdColumn`	`SessionID`	The name of the column that holds the session ID. The SQL type is VARCHAR(100). Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`valueColumn`	`Value`	The name of the column that holds the session object. The SQL type is VARBINARY(4096). This column must be large enough to accommodate all of your session data. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`lookupPool`	4	The number of dedicated connections that perform search operations on the database. Each of these connections would have a precompiled SQL statement for higher performance. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`insertPool`	4	The number of dedicated connections that perform insert operations on the database. Each of these connections would have a precompiled SQL statement for higher performance. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.

TABLE 4–6 manager-properties Properties for IWSSessionManager *(Continued)*

Property Name	Default Value	Description
`updatePool`	`4`	The number of dedicated connections that perform update operations on the database. Each of these connections would have a precompiled SQL statement for higher performance. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.
`deletePool`	`2`	The number of dedicated connections that perform delete operations on the database. Each of these connections would have a precompiled SQL statement for higher performance. Applicable only if the `session-data-store` parameter is set to the `JdbcStore` class.

Note –

- Prior to using `JdbcStore`, you must create the table in which the session information is stored. The name of the table is specified by the `table` parameter, and the table's four columns are specified by the `accessTimeColumn`, `timeOutColumn`, `sessionIdColumn`, and `valueColumn` parameters.
- `FileStore`, `JdbcStore`, `IWSSessionManager`, `IWSHttpSession`, `IWSHttpSessionManager`, and `SessionDataStore` have been deprecated in Sun Java System Web Server 6.1.

Source Code for IWSSessionManager

The IWSSessionManager creates an `IWSHttpSession` object for each session. The source files for `IWSSessionManager.java` and `IWSHttpSession.java` are in the *server_root*`/plugins/java/apis` directory. The source code files for `IWSSessionManager.java` and `IWSHttpSession.java` are provided so you can use them as the starting point for defining your own session managers and session objects.

`IWSSessionManager` extends `IWSHttpSessionManager`. The class file for `IWSHttpSessionManager` is in the JAR file `webserv-rt.jar` in the directory *server_root*`/bin/https/jar`. The IWSSessionManager implements all of the methods in `IWSHttpSessionManager` that need to be implemented, so you can use `IWSSessionManager` as an example of how to extend `IWSHttpSessionManager`. When compiling your subclass of `IWSSessionManager` or `IWSHttpSessionManager`, be sure that the JAR file `webserv-rt.jar` is in your compiler's classpath.

The `JdbcStore.java` and `FileStore.java` source files and the source file for the parent class, `SessionDataStore.java`, are provided so you can modify the session persistence mechanism of IWSSessionManager. These files are also located in the directory *server_root*`/plugins/java/apis` directory.

MMapSessionManager (UNIX Only)

This is a persistent memory map (`mmap`) file-based session manager that works in both single-process and multi-process mode.

The `MaxProcs` directive in the `magnus.conf` file determines whether the server is running in single-process or multi-process mode. For more information, see the *Sun Java System Web Server 6.1 SP11 NSAPI Programmer's Guide*.

Enabling MMapSessionManager

You may want to enable MMapSessionManager to change its default parameters. You can also enable MMapSessionManager for a particular context if the server is running in single-process mode. To do so, edit the `sun-web.xml` file for the web application as in the following example. Note that `persistence-type` must be set to `mmap`.

```
<sun-web-app>
        ...
        <session-config>
            <session-manager persistence-type=”mmap”>
              ...
          </session-manager>
           ...
        </session-config>
        ...
</sun-web-app>
```

For more information about the `sun-web.xml` file, see Chapter 7, Deploying Web Applications

Manager Properties for MMapSessionManager

The following table describes `manager-properties` properties for the MMapSessionManager session manager. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 4–7 manager-properties Properties for MMapSessionManager

Property Name	Default Value	Description
`maxSessions`	`1000`	The maximum number of sessions maintained by the session manager at any given time. The session manager refuses to create any more new sessions if there are already `maxSessions` number of sessions present at that time.
`maxValuesPerSession`	`10`	The maximum number of values or objects a session can hold.
`maxValueSize`	`4096`	The maximum size of each value or object that can be stored in the session.
`timeOut`	`1800`	The amount of time in seconds after a session is last accessed by the client before the session manager destroys it. Those sessions that haven't been accessed for at least `timeOut` seconds are destroyed by the reaper method. If `session-timeout` is specified in `web.xml`, it overrides this `timeOut` parameter value.
`reapInterval`	`600`	The amount of time in seconds that the `SessionReaper` thread sleeps before calling the reaper method again.
`maxLocks`	`1`	The number of cross-process locks to use for synchronizing access to individual sessions across processes. The default value is used if the value `0` is specified. This parameter is ignored in single-process mode.

Note – The MMapSessionManager can only store objects that implement `java.io.Serializable`.

◆ ◆ ◆ CHAPTER 5

Securing Web Applications

This chapter describes the basic goals and features of Sun Java System Web Server 6.1 security, and describes how to write secure web applications containing components that perform user authentication and access authorization tasks. Since it is helpful to have a basic understanding of security responsibilities and terminology, the beginning of the chapter discusses those topics.

This chapter has the following sections:

Sun Java System Web Server Security Goals

In an enterprise computing environment there are many security risks. The goal of Sun Java System Web Server is to provide highly secure, interoperable, and distributed component computing based on the J2SE security model. The security goals for Sun Java System Web Server include the following:

- Full compliance with the Java Servlet 2.3 security model. This includes servlet role-based authorization. For more information, see the Security chapter in the Java Servlet 2.3 specification, which can be downloaded from:

`http://java.sun.com/products/servlet/download.html`

- Support for single sign-on across all Sun Java System Web Server applications within a single security domain.
- Support for several underlying authentication realms, such as simple file and LDAP. Certificate authentication is also supported for SSL client authentication. For Solaris, OS platform authentication is supported in addition to these.
- Support for declarative security via Sun Java System Web Server-specific XML-based role mapping.
- Support for Java policy (Security Manager) enforcement.

The Sun Java System Web Server 6.1 *Administrator's Guide* also contains detailed information about J2SE-based security.

Security Responsibilities Overview

Before delving into the specific security features of Sun Java System Web Server, it is helpful to first understand responsibilities pertaining to security. This section provides that overview.

A J2SE platform's primary goal is to isolate the developer from the security mechanism details and facilitate a secure application deployment in diverse environments. This goal is addressed by providing mechanisms for the application security specification requirements declaratively and outside the application.

When developing applications for Sun Java System Web Server, it is necessary to decide whether your application should use the traditional Sun Java System Web Server native ACL subsystem or the J2SE/Servlet access control model. For guidelines on how to make this decision, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

The roles described in this section apply to the J2SE/Servlet model and are described in more detail in the J2SE specification:

- "Application Developer" on page 82
- "Application Assembler" on page 83
- "Application Deployer" on page 83

Application Developer

The application developer is responsible for the following:

- Specifying application roles.
- Defining role-based access restrictions for the application components (servlets and JSPs components).

- If programmatic security is used, verifying the user roles and authorizing access to features based on these roles. (Programmatic security management is discouraged because it hard-codes the security login in the application instead of allowing the containers to manage it.)

Application Assembler

The application assembler or application component provider must identify all security dependencies embedded in a component, including:

- All role names used by the components that call `isUserInRole`.
- References to all external resources accessed by the components.
- References to all intercomponent calls made by the component.

Application Deployer

The application deployer takes all component security views provided by the assembler and uses them to secure a particular enterprise environment in the application, including:

- Assigning users or groups (or both) to security roles.
- Refining the privileges required to access component methods to suit the requirements of the specific deployment scenario.

Common Security Terminology

Before getting into the specific security features and capabilities of Sun Java System Web Server, it is helpful to have a basic understanding of the common security terminology used throughout this chapter. This section provides that overview.

The most common security processes are authentication, authorization, realm assignment, and role mapping. The following sections define this and other common security terminology:

- "Authentication" on page 84
- "Authorization" on page 84
- "Realms" on page 84
- "J2SE Application Role Mapping" on page 84
- "Container Security" on page 85

Authentication

Authentication verifies the user. For example, the user may enter a user name and password in a web browser, and if those credentials match the permanent profile stored in the active realm, the user is authenticated. The user is associated with a security identity for the remainder of the session.

Authorization

Authorization permits a user to perform the desired operations, after being authenticated. For example, a human resources application may authorize managers to view personal employee information for all employees, but allow employees to view only their own personal information.

Realms

A realm, also called a security policy domain or a security domain in the J2SE specification, is a scope over which a common security policy is defined and enforced by the security administrator of the security service. Supported realms in Sun Java System Web Server are `file`, `ldap`, `certificate`, `solaris`, `custom`, and `nativerealm`. For more information about supported realms, see "Realm Configuration" on page 93.

J2SE Application Role Mapping

In the J2SE/Servlet security model, a client may be defined in terms of a security role. For example, a company might use its employee database to generate both a company-wide phone book application and payroll information. Obviously, while all employees might have access to phone numbers and email addresses, only some employees would have access to the salary information. Employees with the right to view or change salaries might be defined as having a special security role.

A role is different from a user group in that a role defines a function in an application, while a group is a set of users who are related in some way. For example, members of the groups astronauts, scientists, and (occasionally) politicians all fit into the role of SpaceShuttlePassenger.

In Sun Java System Web Server, roles correspond to users or groups (or both) configured in the active realm.

Container Security

The component containers are responsible for providing J2SE application security. Two security forms are provided by the container, as discussed below: programmatic security and declarative security.

Programmatic Security

Programmatic security is when a servlet uses method calls to the security API, as specified by the J2SE security model, to make business logic decisions based on the caller or remote user's security role. Programmatic security should only be used when declarative security alone is insufficient to meet the application's security model.

The J2SE 1.3 specification defines programmatic security with respect to servlets as consisting of two methods of the servlet `HttpServletRequest` interface. Sun Java System Web Server supports these interfaces as defined in the specification.

In addition to the programmatic security defined in the J2SE specifications, Sun Java System Web Server also supports programmatic login. For more information, see "Programmatic Login" on page 98.

Declarative Security

Declarative security means that the security mechanism for an application is declared and handled external to the application. Deployment descriptors describe the J2SE application's security structure, including security roles, access control, and authentication requirements.

Sun Java System Web Server supports the DTDs specified by the J2SE 1.3 specification, and has additional security elements included in its own deployment descriptors. Declarative security is the application deployer's responsibility.

Sun Java System Web Server-specific Security Features

In addition to supporting the J2SE 1.3 security model, Sun Java System Web Server also supports the following features that are specific to the Web Server:

- Single sign-on across all Sun Java System Web Server applications within a single security domain.
- Programmatic login.
- The parallel Access Control List (ACL)-based security model, in addition to the J2SE/Servlet security model.
- Support for secure ACL-based Java web applications, in addition to native content.

This section discusses the following:

Sun Java System Web Server Security Model

Secure applications require a client to be authenticated as a valid application user and have authorization to access servlets and JSPs.

Applications with a secure web container may enforce the following security processes for clients:

- Authenticate the caller
- Authorize the caller for access to each servlet/JSP based on the applicable access control configuration

Authentication is the process of confirming an identity. Authorization means granting access to a restricted resource to an identity, and access control mechanisms enforce these restrictions. Authentication and authorization can be enforced by a number of security models and services.

Sun Java System Web Server 6.1 provides authentication and authorization support through the following mechanisms, which are discussed in this section:

- ACL-based authentication and authorization
- J2SE/Servlet-based authentication and authorization

Whether performed by the ACL subsystem or the J2SE/Servlet authentication subsystem, authentication and authorization are still the two fundamental operations that define secure web content.

ACL-based Authentication and Authorization

ACL-based access control is described at length in the Sun Java System Web Server 6.1 *Administrator's Guide*. This section provides a brief overview of the key concepts.

Sun Java System Web Server 6.1 supports authentication and authorization through the use of locally stored access control lists (ACLs), which describe what access rights a user has for a resource. For example, an entry in an ACL can grant a user named John `read` permission to a particular folder named `misc`:

```
acl "path=/export/user/990628.1/docs/misc/";
  authenticate (user,group) {
      database = "default";
      method = "basic";
   };
   deny (all)
  (user = "John");
```

```
allow (read);
```

The core ACLs in Sun Java System Web Server 6.1 support three types of authentication: basic, certificate, and digest.

Basic authentication relies on lists of user names and passwords passed as cleartext. Certificates bind a name to a public key. Digest authentication uses encryption techniques to encrypt the user's credentials.

The main features of the ACL-based access control model are described below:

- ACL-based authentication uses the following configuration files:
  - *server-install*`/httpacl/*.acl` files
  - *server-install*`/userdb/dbswitch.conf`
  - *server-install*`/config/server.xml`

  Authentication is performed by `auth-db` modules that are configured in the `dbswitch.conf` file.
- Authorization is performed by access control rules set in the *server-install*`/httpacl/*.acl` files, if ACLs are configured.

In addition, the Sun Java System Web Server 6.1 SSL engine supports external crypto hardware to offload SSL processing and to provide optional tamper-resistant key storage.

For more information about access control and the use of external crypto hardware, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

J2SE/Servlet-based Authentication and Authorization

Sun Java System Web Server 6.1, apart from providing ACL-based authentication, also leverages the security model defined in the J2SE 1.3 specification to provide several features that help you develop and deploy secure Java web applications.

A typical J2SE-based web application consists of the following parts, access to any or all of which can be restricted:

- Servlets
- JavaServer Pages (JSP) components
- HTML documents
- Miscellaneous resources, such as image files and compressed archives

The J2SE/Servlet-based access control infrastructure relies on the use of security realms. When a user tries to access the main page of an application through a web browser, the web container prompts for the user's credential information, and then passes it for verification to the realm that is currently active in the security service.

A realm, also called a security policy domain or security domain in the J2SE specification, is a scope over which a common security policy is defined and enforced by the security administrator of the security service.

The main features of the J2SE/Servlet-based access control model are described below:

- J2SE/Servlet-based authentication uses the following configuration files:
  - The web application deployment descriptor files `web.xml` and `sun-web.xml`
  - *server-install*`/config/server.xml`

  Authentication is performed by Java security realms that are configured through `AUTHREALM` entries in the `server.xml` file.
- Authorization is performed by access control rules in the deployment descriptor file, `web.xml`, in case any such rules have been set.

Web Application and URL Authorizations

Secure web applications may have authentication and authorization properties. The web container supports three types of authentication: basic, certificate, and form-based. The core ACLs support basic, certificate, and digest. For more information about ACL configuration, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

When a browser requests the main application URL, the web container collects the user authentication information (for example, user name and password) and passes it to the security service for authentication.

For J2SE web applications, Sun Java System Web Server consults the security policies (derived from the deployment descriptors) associated with the web resource to determine the security roles used to permit resource access. The web container tests the user credentials against each role to determine if it can map the user to the role.

User Authentication by Servlets

The web-based login mechanisms required by the J2SE 1.3 specification are supported by the Sun Java System Web Server. These mechanisms are discussed in this section:

- "HTTP Basic Authentication" on page 89
- "SSL Mutual Authentication" on page 89
- "Form-Based Login" on page 90

The `login-config` element in the `web.xml` deployment descriptor file describes the authentication method used, the application's realm name displayed by the HTTP basic authentication, and the form login mechanism's attributes.

The `login-config` element syntax is as follows:

```
<!ELEMENT login-config (auth-method?,realm-name?,form-login-config?)>
```

Note – The `auth-method` subelement of `login-config` is officially optional, but if it is not included, the server defaults to HTTP Basic Authentication, which is not very secure.

For more information about `web.xml` elements, see the Java Servlet 2.3 specification (chapter SRV.13, "Deployment Descriptor"). You can find the specification here:

`http://java.sun.com/products/servlet/download.html`

For more information regarding `sun-web.xml` elements, see Chapter 7, Deploying Web Applications.

HTTP Basic Authentication

HTTP basic authentication (RFC 2617) is supported by the Sun Java System Web Server. Because passwords are sent with base64 encoding, this authentication type is not very secure. Use of SSL or another equivalent transport encryption is recommended to protect the password during transmission.

SSL Mutual Authentication

SSL 3.0 and the means to perform mutual (client/server) certificate-based authentication is a J2SE 1.3 specification requirement. This security mechanism provides user authentication using HTTPS (HTTP over SSL).

The Sun Java System Web Server SSL mutual authentication mechanism (also known as HTTPS authentication) supports the following cipher suites:

SSL_RSA_EXPORT_WITH_RC4_40_MD5

SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5

SSL_RSA_EXPORT_WITH_DES40_CBC_SHA

SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA

SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA

SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA

Form-Based Login

The login screen's look and feel cannot be controlled with the HTTP browser's built-in mechanisms. J2SE introduces the ability to package a standard HTML or servlet/JSP based form for logging in. The login form is associated with a web protection domain (an HTTP realm) and is used to authenticate previously unauthenticated users.

Because passwords are sent in the clear (unless protected by the underlying transport), this authentication type is not very secure. Use of SSL or another equivalent transport encryption is recommended to protect the password during transmission.

For the authentication to proceed appropriately, the login form action must always be `j_security_check`.

The following is an HTML sample showing how to program the form in an HTML page:

```
<form method="POST" action="j_security_check">
        <input type="text" name="j_username">
         <input type="password" name="j_password">
      </form>
```

You can specify the parameter encoding for the form. For details, see "parameter-encoding" on page 143.

User Authentication for Single Sign-on

The single sign-on across applications on the Sun Java System Web Server is supported by the Sun Java System Web Server servlets and JSPs. This feature allows multiple applications that require the same user sign-on information to share this information between them, rather than having the user sign on separately for each application. These applications are created to authenticate the user one time, and when needed this authentication information is propagated to all other involved applications.

An example application using the single sign-on scenario could be a consolidated airline booking service that searches all airlines and provides links to different airline web sites. Once the user signs on to the consolidated booking service, the user information can be used by each individual airline site without requiring another sign on.

Single sign-on operates according to the following rules:

- Single sign-on applies to web applications configured for the same realm and virtual server. The realm is defined by the `realm-name` element in the `web.xml` file. For information about virtual servers, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide* or the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

- As long as users access only unprotected resources in any of the web applications on a virtual server, they are not challenged to authenticate themselves.
- As soon as user accesses a protected resource in any web application associated with a virtual server, the user is challenged to authenticate, using the login method defined for the web application currently being accessed.
- Once authenticated, the roles associated with this user are used for access control decisions across all associated web applications, without challenging the user to authenticate to each application individually.
- When the user logs out of one web application (for example, by invalidating or timing out the corresponding session if form-based login is used), the user's sessions in all web applications are invalidated. Any subsequent attempt to access a protected resource in any application requires the user to authenticate him or herself again.
- The single sign-on feature utilizes HTTP cookies to transmit a token that associates each request with the saved user identity, so it can only be used in client environments that support cookies.

To configure single sign-on, set the following properties in the `VS` element of the `server.xml` file:

- `sso-enabled`: If `false`, single sign-on is disabled for this virtual server, and users must authenticate separately to every application on the virtual server. The default value is set to `false`.
- `sso-max-inactive-seconds`: Specifies the time after which a user's single sign-on record becomes eligible for purging if no client activity is received. Since single sign-on applies across several applications on the same virtual server, access to any of the applications keeps the single sign-on record active. The default value is 5 minutes (300 seconds). Higher values provide longer single sign-on persistence for the users at the expense of more memory use on the server.
- `sso-reap-interval-seconds`: Specifies the interval between purges of expired single sign-on records. The default value is `60`.

Here is an example configuration with all default values:

```
<VS id="server1" ... >
         ...
        <property name="sso-enabled" value="true">
         <property name="sso-max-inactive-seconds" value="300">
         <property name="sso-reap-interval-seconds" value="60">
      </VS>
```

User Authorization by Servlets

Servlets can be configured to only permit access to users with the appropriate authorization level. This section covers the following topics:

Defining Roles

Security roles define an application function, made up of a number of users, groups, or both. The relationship between users and groups is determined by the specific realm implementation being used.

You define roles in the J2SE deployment descriptor file,`web.xml`, and the corresponding role mappings in the Sun Java System Web Server deployment descriptor file, `sun-web.xml`. For more information about `sun-web.xml`, see Chapter 7, Deploying Web Applications

Each `security-role-mapping` element in the `sun-web.xml` file maps a role name permitted by the web application to principals and groups. For example, a `sun-web.xml` file for a deployed web application might contain the following:

```
<sun-web-app>
   <security-role-mapping>
      <role-name>manager</role-name>
      <principal-name>jgarcia</principal-name>
      <principal-name>mwebster</principal-name>
      <group-name>team-leads</group-name>
   </security-role-mapping>
   <security-role-mapping>
      <role-name>administrator</role-name>
      <principal-name>dsmith</principal-name>
   </security-role-mapping>
</sun-web-app>
```

Note that the `role-name` in this example must match the `role-name` in the `security-role` element of the corresponding `web.xml` file.

For web applications, the roles are always specified in the `sun-web.xml` file. A role can be mapped to either specific principals or to groups (or both). The principal or group names used must be valid principals or groups in the current realm.

Defining Servlet Authorization Constraints

On the servlet level, you define access permissions using the `auth-constraint` element of the `web.xml` file.

The `auth-constraint` element on the resource collection must be used to indicate the user roles permitted to the resource collection. Refer to the Java Servlet specification for details on configuring servlet authorization constraints.

Fetching the Client Certificate

When you enable SSL and require client certificate authorization, your servlets have access to the client certificate as shown in the following example:

```
if (request.isSecure()) {
         java.security.cert.X509Certificate[] certs;
         certs = request.getAttribute("javax.servlet.request.X509Certificate");
        if (certs != null) {
            clientCert = certs[0];
            if (clientCert != null) {
               // Get the Distinguised Name for the user.
               java.security.Principal userDN = clientCert.getSubjectDN();
               ...
            }
         }
      }
```

The `userDn` is the fully qualified Distinguished Name for the user.

Realm Configuration

This section provides an overview of the configuration characteristics of the supported realms. For detailed information about configuring realms, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

The section describes the following realms:

- "Native Realm" on page 97

File

The file realm is the default realm when you first install Sun Java System Web Server, and has the following configuration characteristics:

- Name: `file`
- Classname: `com.iplanet.ias.security.auth.realm.file.FileRealm`

Required properties are as follows:

- `file`: The name of the file that stores user information. By default this file is *instance_dir*`/config/keyfile`.
- `jaas-context`: The value must be `fileRealm`.

The user information file is initially empty, so you must add users before you can use the `file` realm.

LDAP

The LDAP realm allows you to use an LDAP database for user security information, and has the following configuration characteristics:

- Name: `ldap`
- Classname: `com.iplanet.ias.security.auth.realm.ldap.LDAPRealm`

Required properties are as follows:

- `directory`: The LDAP URL to your server.
- `base-dn`: The base DN for the location of user data. This base DN can be at any level above the user data, since a tree scope search is performed. The smaller the search tree, the better the performance.
- `jaas-context`: The value must be `ldapRealm`.

You can add the following optional properties to tailor the LDAP realm behavior:

- `search-filter`: The search filter to use to find the user. The default is `uid=%s` (`%s` expands to the subject name).
- `group-base-dn`: The base DN for the location of group data. By default it is same as the `base-dn`, but it can be tuned if necessary.
- `group-search-filter`: The search filter to find group memberships for the user. The default is `uniquemember=%d` (`%d` expands to the user element DN).
- `group-target`: The LDAP attribute name that contains group name entries. The default is `CN`.

- `search-bind-dn`: An optional DN used to authenticate to the directory for performing the `search-filter` lookup. Only required for directories that do not allow anonymous search.
- `search-bind-password`: The LDAP password for the DN given in `search-bind-dn`.

You must create the desired user(s) in your LDAP directory. You can do this from the Sun™ Java System Directory Server console, or through any other administration tool that supports LDAP and your directory's schema. User and group information is stored in the external LDAP directory.

The `principal-name` used in the deployment descriptors must correspond to your LDAP user information.

Solaris

The Solaris realm allows authentication using Solaris user name and password data. This realm is supported only on Solaris 9, and has the following configuration characteristics:

- Name: `solaris`
- Classname: `com.iplanet.ias.security.auth.realm.file.SolarisRealm`

Required properties are as follows:

- `jaas-context`: The value must be `solarisRealm`.

Users and groups are stored in the underlying Solaris user database, as determined by the system's PAM (Pluggable Authentication Module) configuration.

Note – The Solaris realm invokes the underlying PAM infrastructure for authentication. If the configured PAM modules require root privileges, the instance must run as root to use this realm. For details, see the "Using Authentication Services (Tasks)" chapter in the Solaris 9 *System Administration Guide: Security Services*.

Certificate

The certificate realm supports SSL authentication. The certificate realm sets up the user identity in Sun Java System Web Server's security context and populates it with user data from the client certificate. The J2SE containers then handle authorization processing based on each user's DN from his or her certificate. The certificate realm has the following configuration characteristics:

- Name: `certificate`
- Classname: `com.iplanet.ias.security.auth.realm.certificate.CertificateRealm`

You can add the following optional property to tailor the certificate realm behavior:

- `assign-groups`: If this property is set, its value is taken to be a comma-separated list of group names. All clients presenting valid certificates are assigned membership to these groups for the purposes of authorization decisions in the web container.

When you deploy an application, you must specify `CLIENT-CERT` as the authentication mechanism in the `web.xml` file as follows:

```
<login-config>
        <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

You must obtain a client certificate and install it in your browser to complete the setup for client certificate authentication. For details on how to set up the server and client certificates, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

You can configure the server instance for SSL authentication in these ways:

- Configure an `SSLPARAMS` element in `server.xml`, then restart the server. For more information about the `server.xml file`, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.
- Use the Administration interface as described in the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

Note – In most cases, it is not necessary to configure a certificate realm in `server.xml` when using CLIENT-CERT authentication in web applications. Since the CLIENT-CERT authentication method inherently implies certificate-based authentication, Sun Java System Web Server will internally use a certificate realm even if one is not configured in `server.xml`. You can still configure a certificate realm if you want to specify properties for it (for example, `assign-groups`).

Custom Realm

You can create a custom realm by providing a Java™ Authentication and Authorization Service (JAAS) login module and a realm implementation. Note that client-side JAAS login modules are not suitable for use with Sun Java System Web Server. For more information about JAAS, refer to the JAAS specification for Java 2 SDK, v1.4, available here:

`http://java.sun.com/products/jaas/`

A sample application that uses a custom realm is available with Sun Java System Web Server at the following location:

server_root`/plugins/java/samples/webapps/security`

Native Realm

The native realm is a special realm that provides a bridge between the core Sun Java System Web Server ACL-based authentication and the J2SE/Servlet authentication model. By using the native realm for Java web applications, it becomes possible to have the ACL subsystem perform the authentication (instead of having the Java web container do so) and yet have this identity available for Java web applications.

This functionality is provided by pluggable realm called `NativeRealm`, which acts as a bridge between the J2SE security subsystem and the access control security subsystem.

Depending on whether a security constraint is configured for a web application, the two modes of operation described below are supported by the native realm:

- If a security constraint is defined in the application's deployment descriptor file, `web.xml`, the web container carries out normal authentication and authorization processing. When the `NativeRealm` realm is invoked for validating user information, the task of verification is delegated to the core `auth-db` specified in the realm configuration. See the *Sun Java System Web Server 6.1 SP11 Administrator's Guide* for more information on how to configure `auth-db` in `dbswitch.conf` and `server.xml`.

  For example (`classname=` is all on one line, with no spaces):

  ```
  <AUTHREALM name="native"
  classname="com.iplanet.ias.security.auth.realm.webcore.
  NativeRealm">
          <PROPERTY name="auth-db" value="name">
           <PROPERTY name="jaas-context" value="nativeRealm">
  </AUTHREALM>
  ```

- If a security constraint is not defined in the application's deployment descriptor file `web.xml` when using `NativeRealm`, the Java web container does not carry out authentication and authorization tasks. These tasks are left to the core access control lists (ACLs). ACLs are collections of rules that follow a hierarchy and determine whether access should be granted or denied for the requested resource. The ACLs yield the user's identity, which is then made available to the Java web application. In other words, if the servlet later invokes a principal's identity with the `request.getUserPrincipal()` method, the correct user identity will be returned.

  In this scenario it is not necessary to provide an `auth-db` to the `NativeRealm` configuration, since the access control list that was applied to the given request is already bound to an `auth-db`.

  For example:

  ```
  <AUTHREALM name="native"
  classname="com.iplanet.ias.security.auth.realm.webcore.
  NativeRealm">
  </AUTHREALM>
  ```

For more details about access control lists, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

Note – While it is possible to apply both ACL access control rules and `web.xml` security constraints on a single application, this usage is discouraged. It may lead to duplicate authentication prompts or otherwise confusing behavior. You should always pick either core ACL or J2SE `web.xml`-based access control mechanisms for a given web application.

Programmatic Login

Programmatic login allows a deployed J2SE application to invoke a login method. If the login is successful, a `SecurityContext` is established as if the client had authenticated using any of the conventional J2SE mechanisms.

Programmatic login is useful for an application with unique needs that cannot be accommodated by any of the J2SE standard authentication mechanisms.

This section discusses the following topics:

Precautions

The Sun Java System Web Server is not involved in how the login information (user name and password) is obtained by the deployed application. Programmatic login places the burden on the application developer with respect to assuring that the resulting system meets security requirements. If the application code reads the authentication information across the network, it is up to the application to determine whether to trust the user.

Programmatic login allows the application developer to bypass the application server-supported authentication mechanisms and feed authentication data directly to the security service. While flexible, this capability should not be used without some understanding of security issues.

Since this mechanism bypasses the container-managed authentication process and sequence, the application developer must be very careful in making sure that authentication is established before accessing any restricted resources or methods. It is also the application developer's responsibility to verify the status of the login attempt and to alter the behavior of the application accordingly.

The programmatic login state does not necessarily persist in sessions or participate in single sign-on.

Lazy authentication is not supported for programmatic login. If an access check is reached and the deployed application has not properly authenticated using the programmatic login method, access is denied immediately and the application may fail if not properly coded to account for this occurrence.

Granting Programmatic Login Permission

The `ProgrammaticLoginPermission` permission is required to invoke the programmatic login mechanism for an application. This permission is not granted by default to deployed applications because this is not a standard J2SE mechanism.

To grant the required permission to the application, add the following to the *instance_dir*`/config/server.policy` file:

```
grant codeBase "file:jar_file_path" {
    permission com.sun.appserv.security.ProgrammaticLoginPermission
    "login";
 };
```

The `jar_file_path` is the path to the application's JAR file.

For more information about the `server.policy` file, see "The server.policy File" on page 100.

The ProgrammaticLogin Class

The `com.sun.appserv.security.ProgrammaticLogin` class enables a user to perform login programmatically.

The login method for servlets or JSPs has the following signature:

```
public Boolean login(String user, String password,
javax.servlet.http.HttpServletRequest request,
 javax.servlet.http.HttpServletResponse response)
```

This method:

- Performs the authentication
- Returns `true` if login succeeded, `false` if login failed

Enabling the Java Security Manager

Sun Java System Web Server 6.1 supports the Java Security Manager. The Java Security Manager is disabled by default when you install the product, which may improve performance significantly for some types of applications. Enabling the Java Security Manager may improve security by restricting the rights granted to your J2SE web applications. To enable the Java Security Manager, "uncomment" entries in the `server.xml` file:

```
<JVMOPTIONS>-Djava.security.manager</JVMOPTIONS>

<JVMOPTIONS>-Djava.security.policy=instance_dir

/config/server.policy</JVMOPTIONS>
```

where *instance_dir* is the path to the installation directory of this server instance.

Based on your application and deployment needs, you should evaluate whether to run with or without the Security Manager.

Running with the Security Manager on will help catch some spec-compliance issues with J2SE applications. All J2SE applications should be able to run with the Security Manager active and with only the default permissions. For this reason it is recommended that the Security Manager be turned on during development. This will help produce applications that can easily be deployed in environments where the Security Manager is always active (such as Sun™ Java System Application Server). Running with the Security Manager also helps isolate applications and may catch inappropriate operations.

The main drawback of running with the Security Manager is that it negatively impacts performance. Depending on the application details and the deployment environment, this impact may be negligible or quite significant.

The server.policy File

Each Sun Java System Web Server instance has its own standard Java™ 2 Platform, Standard Edition (J2SE) policy file, located in the *instance_dir*/`config` directory. The file is named `server.policy`.

Sun Java System Web Server 6.1 is a J2SE 1.3-compliant web server. As such, it follows the recommendations and requirements of the J2SE specification, including the optional presence of the Security Manager (the Java component that enforces the policy), and a limited permission set for J2SE application code.

This section includes the following topics:

Default Permissions

Internal server code is granted all permissions. These are covered by the `AllPermission` grant blocks to various parts of the server infrastructure code. Do not modify these entries.

Application permissions are granted in the default grant block. These permissions apply to all code not part of the internal server code listed previously.

A few permissions above the minimal set are also granted in the default `server.policy` file. These are necessary due to various internal dependencies of the server implementation. J2SE application developers must not rely on these additional permissions.

Changing Permissions for an Application

The default policy for each instance limits the permissions of J2SE-deployed applications to the minimal set of permissions required for these applications to operate correctly. If you develop applications that require more than this default set of permissions, you can edit the `server.policy` file to add the custom permissions that your applications need.

You should add the extra permissions only to the applications that require them, not to all applications deployed to a server instance. Do not add extra permissions to the default set (the grant block with no `codebase`, which applies to all code). Instead, add a new grant block with a codebase specific to the application requiring the extra permissions, and only add the minimally necessary permissions in that block.

Note – Do not add `java.security.AllPermission` to the `server.policy` file for application code. Doing so completely defeats the purpose of the Security Manager, yet you still get the performance overhead associated with it.

As noted in the J2SE specification, an application should provide documentation of the additional permissions it needs. If an application requires extra permissions but does not document the set it needs, contact the application author for details.

As a last resort, you can iteratively determine the permission set an application needs by observing `AccessControlException` occurrences in the server log. If this is not sufficient, you can add the `-Djava.security.debug=all` JVM option to the server instance. For details, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide* or the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

You can use the J2SE standard policy tool or any text editor to edit the `server.policy` file. For more information, see:

`http://java.sun.com/docs/books/tutorial/security1.2/tour2/index.html`

For detailed information about the permissions you can set in the `server.policy` file, see:

`http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html`

The Javadoc for the `Permission` class is here:

`http://java.sun.com/j2se/1.4.2/docs/api/java/security/Permission.html`

For More Information

The following table describes where you can find more information about security and security configuration topics in the Sun Java System Web Server 6.1 documentation:

TABLE 5–1 More Information on Security-related Issues

For Information On	See
Configuring Java security and realm-based authentication	The chapter "Securing Your Web Server" in the Sun Java System Web Server 6.1 *Administrator's Guide*.
Certificates and public key cryptography	The chapter "Using Certificates and Keys" in the Sun Java System Web Server 6.1 *Administrator's Guide*.
ACL-based security	The chapter "Controlling Access to Your Server" in the Sun Java System Web Server 6.1 *Administrator's Guide*.
Configuring `auth-db` in the `dbswitch.conf` and `server.xml` files	The chapter "Controlling Access to Your Server" in the Sun Java System Web Server 6.1 *Administrator's Guide*.

◆ ◆ ◆ CHAPTER 6

Deploying Web Applications

This chapter describes how web applications are assembled and deployed in Sun Java System Web Server.

This chapter includes the following sections:

Web Application Structure

Web applications have a directory structure, which is accessible from a mapping to the application's document root (for example, `/hello`). The document root contains JSP files, HTML files, and static files such as image files.

A WAR file (Web ARchive file) contains a web application in compressed form.

A special directory under the document root, `WEB-INF`, contains information related to the application that is not in the public document tree. No file contained in `WEB-INF` can be served directly to the client. The contents of `WEB-INF` include:

- `/WEB-INF/classes/*`: The directory for servlet and other classes.
- `/WEB-INF/web.xml` and `/WEB-INF/sun-web.xml`: XML-based deployment descriptors that specify the web application configuration, including mappings, initialization parameters, and security constraints.

The web application directory structure follows the structure outlined in the J2SE specification. The following is an example directory structure of a simple web application:

```
+ hello/
       |--- index.jsp
       |--+ META-INF/
       |  |--- MANIFEST.MF
       '--+ WEB-INF/
          |--- web.xml
          '--- sun-web.xml
```

Creating Web Deployment Descriptors

Sun Java System Web Server web applications include two deployment descriptor files:

- A J2SE standard file (`web.xml`), described in the Java Servlet 2.3 specification (chapter SRV .13, "Deployment Descriptor"). You can find the specification here:

  `http://java.sun.com/products/servlet/download.html`

- An optional Sun Java System Web Server-specific file (`sun-web.xml`), described later in this chapter.

The easiest way to create the `web.xml` and `sun-web.xml` files is to deploy a web application using the Sun Java Studio. For example, `web.xml` and `sun-web.xml` files, see "Sample Web Application XML Files" on page 147.

Deploying Web Applications

When you deploy, undeploy, or redeploy a web application, you do not need to restart the server. Deployment is dynamic.

You can deploy a web application in the following ways, which are described briefly in these sections:

- "Using the Administration Interface" on page 105
- "Deploying a Web Application using wdeploy" on page 105
- "Using Sun Java Studio" on page 107

Using the Administration Interface

▼ To deploy web applications using the administration interface

1. **In the Server Manager interface, click the Virtual Server Class tab.**
   The Manage a Class of Virtual Servers page displays.
2. **Select a class of virtual servers, and then click Manage.**
   The Class Manager interface displays.
3. **Select a virtual server, and then click Manage.**
   The Virtual Server Manager interface displays.
4. **Click the Web Applications tab, and then click the Deploy Web Application link.**
   The Deploy Web Application page displays.
5. **Enter the following information:**
   - **WAR File On.** Select a Local Machine when uploading a WAR file to your server, or Server Machine when the WAR file already resides there.
     - **WAR File Path.** Enter the path on the local or server machine to the WAR file containing the web application. On server machines, enter the absolute path to the WAR file. On local machines, you can browse the available paths.
     - **Application URI**. Enter the URI on the virtual server for the web application.
     - **Installation Directory.** Enter the absolute path to the directory on the server machine from which the contents of the WAR file will be extracted. If the directory does not exist, a directory will be created.
6. **Click OK.**
   For more information about using the Administration interface to manage Sun Java System Web Server, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*.

Deploying a Web Application using wdeploy

Before you can deploy a web application manually, you must make sure that the *server_root*`/bin/https/bin` directory is in your path.

You can use the `wdeploy` utility at the command line to deploy a WAR file into a virtual server web application environment as follows:

```
wdeploy deploy -u uri_path -i instance -v vs_id [-d directory] war_file
```

You can also delete a virtual server web application:

```
wdeploy delete -u uri_path -i instance -v vs_id hard|soft
```

You can also list the web application URIs and directories for a virtual server:

```
wdeploy list -i instance -v vs_id
```

The following table describes the command parameters. The left column lists the parameter, and the right column describes the parameter.

TABLE 6–1 command Parameters

Parameter	Description
uri_path *instance*	The URI prefix for the web application (requires a leading "/"). The server instance name.
vs_id	The virtual server ID.
directory	(optional) The directory to which the application is deployed, or from which the application is deleted. If not specified for deployment, the application is deployed to *instance_directory*/`webapps`/*vs_id*/*webappname*. For example: `/opt/SUNWwbsvr/https-test/webapps/https-test/testapp`
`hard|soft`	Specifies whether the directory and the `server.xml` entry are deleted (`hard`), or just the `server.xml` entry (`soft`).
war_file	The WAR file name.

When you execute the `wdeploy deploy` command, two things happen:

- A web application with the given *uri_path* and *directory* gets added to the `server.xml` file.
- The WAR file gets extracted at the target *directory*.

Example usage of the command is as follows:

```
wdeploy deploy -u /hello -i server.sun.com -v acme.com
<server_root>/plugins/java/sample/webapps/simple/webapps-simple.war
```

After you have deployed an application, you can access it from a browser as follows:

```
http://vs_urlhost[:vs_port]/uri_path/[index_page]
```

The following table describes the parts of the URL. The left column lists the part, and the right column describes what the part means.

TABLE 6–2 Parts of the URL

Part	Description
vs_urlhost	One of the `urlhosts` values for the virtual server.
vs_port	(optional) Only needed if the virtual server uses a nondefault port.
uri_path	The same one you used to deploy the application. This is also the context path.
index_page	(optional) The page in the application that end users are meant to access first.

For example:

```
http://acme.com:80/hello/index.jsp
```

- or -

```
http://acme.com/hello/
```

Using Sun Java Studio

Sun Java System Web Server 6.1 supports Sun Java Studio 5, Standard Edition. You can use Sun Java Studio to assemble and deploy web applications.

Sun Java Studio technology is Sun's powerful, extensible, integrated development environment (IDE) for Java technology developers. Sun Java Studio 8 is based on NetBeans™ software, and integrated with the Sun Java System platform. (Sun Java System Web Server 6.1 also supports NetBeans 3.5 and 3.5.1.)

Sun Java Studio support is available on all platforms supported by Sun Java System Web Server 6.1. You can obtain the plugin for the Web Server in the following ways:

- From the Companion CD in the Sun Java System Web Server 6.1 Media Kit
- By using the AutoUpdate feature of Sun Java Studio
- From the download center for Sun Java System Web Server 6.1 at

  `http://www.sun.com/software/download/inter_ecom.html`

Note – The Sun Java Studio 5 plugin for Sun Java System Web Server 6.1 works only with a local Web Server (that is, with the IDE and the Web Server on the same machine).

For information about using the web application features in Sun Java Studio 8, explore the resources at

```
http://developers.sun.com/
prodtech/javatools/jsstandard/reference/docs/index.html
```

The behavior of the Sun Java Studio 8 plugin for Sun Java System Web Server 6.1 is the same as that for Sun™ Java System Application Server 7. If you are using the "Web Application Tutorial" at the web site listed above, for instance, you would set the Sun Java System Web Server 6.1 instance as the default, and then take the same actions described in the tutorial.

For more information about Sun Java Studio 8, visit

```
http://www.sun.com/software/sundev/jde/
```

Note – For information about using Sun Java Studio to debug web applications, see "Using Sun Java Studio for Debugging" on page 153 in this guide.

Enabling and Disabling Web Applications

Sun Java System Web Server 6.1 allows you to enable or disable a web application. You can do so in either of the following ways, as discussed in this section:

Using the Administration Interface

▼ To enable or disable a deployed web application using the administrator interface

1. **Access the Administration Server, select the server instance, and click Manage.**
2. **Click the Virtual Server Class tab.**
3. **Select the virtual server class that contains the virtual server instance in which the web application is deployed, and click Manage.**
4. **Select the virtual server in which the web application is deployed, and click Manage.**
5. **Click the Web Applications tab, and then click the Edit Web Applications link.**
6. **From the Action drop-down list, select Enable or Disable to enable or disable a specific web application.**

7 **Click OK.**

Editing the server.xml File

By default, an application is automatically enabled with the value set to `true` in the *server-id*`/config/server.xml` file. You can disable the application by setting the value to `false`.

Example:

```
<WEBAPP uri="/catalog" path="/export/apps/catalog" enabled="false">
```

For more information, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

Dynamic Reloading of Web Applications

If you make code changes to a web application and dynamic reloading is enabled, you do not need to redeploy the web application or restart the server.

To enable dynamic reloading, you must edit the following attribute of the `server.xml` file's `JAVA` element, then restart the server:

`dynamicreloadinterval=`*integer*

where *integer* specifies the interval (in seconds) after which a deployed application will be checked for modifications and reloaded if necessary. To enable dynamic reloading, you must specify a value greater than 0.

The `server.xml` setting is the default value for all applications. An individual application can override the value for `dynamicreloadinterval` by specifying a value to the `class-loader` element in the `sun-web.xml` file.

For information about `sun-web.xml`, see the section, "The sun-web-app_2_3-1.dtd File" on page 112 Server 6.1 *Administrator's Configuration File Reference*.

▼ To load new servlet files or reload deployment descriptor changes

1 **Create an empty file named** `.reload` **at the root of the deployed module. For example:**

`instance_dir``/webapps/`**`vs_id`**`/`**`uri`**`/.reload`

where *vs_id* is the virtual server ID in which the web application is deployed, and *uri* is the value of the `uri` attribute of the `<WEBAPP>` element.

2 **Explicitly update the `.reload` file's timestamp (`touch .reload` in UNIX) each time you make the above changes.**

For JSPs, changes are reloaded automatically at a frequency set in the `reload-interval` property of the `jsp-config` element in the `sun-web.xml` file. To disable dynamic reloading of JSPs, set the `reload-interval` property to `-1`.

Classloaders

In a Java Virtual Machine (JVM), the classloaders dynamically load a specific Java class file needed for resolving a dependency. For example, when an instance of `java.util.Enumeration` needs to be created, one of the classloaders loads the relevant class into the environment.

Classloaders in the Sun Java System Web Server 6.1 runtime follow the hierarchy shown in the following figure.

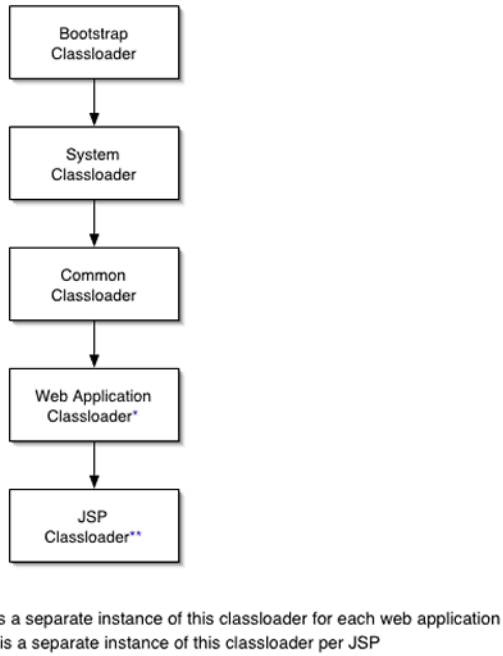

FIGURE 6–1 Classloader Runtime Hierarchy

Note that this is not a Java inheritance hierarchy, but a delegation hierarchy. In the delegation design, a classloader delegates classloading to its parent before attempting to load a class itself. If the parent classloader can't load a class, the `findClass()` method is called on the classloader subclass. In effect, a classloader is responsible for loading only the classes not available to the parent.

The exception is the Web Application Classloader, which follows the delegation model in the Servlet specification. The Web Application Classloader searches in the local classloader before delegating to its parent. You can make the Web Application Classloader delegate to its parent first by setting `delegate="true"` in the `class-loader` element of the `sun-web.xml` file. For more information, see "Classloader Elements" on page 140

The following table describes Sun Java System Web Server 6.1 classloaders. The left column lists the classloaders, and the right column lists descriptions of those classloaders and the files they examine.

TABLE 6–3 Sun Java System Web Server 6.1 Classloaders

Classloader	Description
Bootstrap	The Bootstrap Classloader loads the JDK classes. Only one instance of this classloader exists in the entire server.
System	The System Classloader loads the core Sun Java System Web Server 6.1 classes. It is created based on the `classpathprefix`, `serverclasspath`, and `classpathsuffix` attributes of the `<JAVA>` element in the `server.xml` file. The environment classpath is included if `envclasspathignored="false"` is set in the `<JAVA>` element. Only one instance of this classloader exists in the entire server. If any changes are made to these attributes/classes, the server must be restarted for the changes to take effect. For more information about the `<JAVA>` element in `server.xml`, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.
Common	The Common Classloader loads classes in the *instance_dir*`/lib/classes` directory, followed by JAR and ZIP files in the *instance_dir*`/lib` directory. The existence of these directories is optional; if they don't exist, the Common Classloader is not created. If any changes are made to these classes, the server must be restarted for the changes to take effect.
Web Application	The Web Application Classloader loads the servlets and other classes in a specific web application. That is, from `WEB-INF/lib` and `WEB-INF/classes` and from any additional classpaths specified in the `extra-class-path` attribute of the `class-loader` element in `sun-web.xml`. For more information, see "Classloader Elements" on page 140 An instance of this classloader is created for each web application. If dynamic reloading is enabled, any changes made to these attributes/classes are reloaded by the server without the need for a restart. For more information, see "Dynamic Reloading of Web Applications" on page 109

TABLE 6–3 Sun Java System Web Server 6.1 Classloaders *(Continued)*

Classloader	Description
JSP	The JSP Classloader loads the compiled JSP classes of JSPs. An instance of this classloader is created for each JSP file. Any changes made to a JSP are automatically detected and reloaded by the server, unless dynamic reloading of JSPs has been disabled by setting the `reload-interval` property to `-1` in the `jsp-config` element of the `sun-web.xml` file. For more information, see"jsp-config" on page 141

The sun-web-app_2_3-1.dtd File

The `sun-web-app_2_3-1.dtd` file defines the structure of the `sun-web.xml` file, including the elements it can contain and the subelements and attributes these elements can include. The `sun-web-app_2_3-1.dtd` file is located in the *install_dir*`/bin/https/dtds` directory.

Note – Do not edit the `sun-web-app_2_3-1.dtd` file. The file contents change only with new versions of Sun Java System Web Server.

For more information about DTD files and XML, see the XML specification at:

`http://www.sun.com/software/dtd/appserver/sun-web-app_2_3-1.dtd`

Each element defined in a DTD file (which may be present in the corresponding XML file) can contain the following:

- "Subelements" on page 112
- "Data" on page 113
- "Attributes" on page 113

Subelements

Elements can contain subelements. For example, the following file fragment defines the cache element.

```
<!ELEMENT cache (cache-helper*, default-helper?, property*, cache-mapping*)>
```

The `ELEMENT` tag specifies that a `cache` element can contain `cache-helper`, `default-helper`, `property`, and `cache-mapping` subelements.

The following table shows how optional suffix characters of subelements determine the requirement rules, or number of allowed occurrences, for the subelements. The left column lists the subelement ending character, and the right column lists the corresponding requirement rule.

TABLE 6–4 Requirement Rules and Subelement Suffixes

Subelement Suffix	Requirement Rule
*element**	Can contain *zero or more* of this subelement.
element?	Can contain *zero or one* of this subelement.
element+	Must contain *one or more* of this subelement.
element (no suffix)	Must contain *only one* of this subelement.

If an element cannot contain other elements, you see `EMPTY` or `(#PCDATA)` instead of a list of element names in parentheses.

Data

Some elements contain character data instead of subelements. These elements have definitions of the following format:

```
<!ELEMENT element-name (#PCDATA)>
```

For example:

```
<!ELEMENT description (#PCDATA)>
```

In the `sun-web.xml` file, white space is treated as part of the data in a data element. Therefore, there should be no extra white space before or after the data delimited by a data element. For example:

```
<description>class name of session manager</description>
```

Attributes

Elements that have `ATTLIST` tags contain attributes (name-value pairs). For example:

```
<!ATTLIST     cachemax-capacity CDATA     "4096"
              timeout           CDATA     "30"
              enabled           %boolean; "false">
A cache element can contain max-capacity, timeout, and enabled attributes.
```

The `#REQUIRED` label means that a value must be supplied. The `#IMPLIED` label means that the attribute is optional, and that Sun Java System Web Server generates a default value. Wherever possible, explicit defaults for optional attributes (such as "`true`") are listed.

Attribute declarations specify the type of the attribute. For example, `CDATA` means character data, and `%boolean` is a predefined enumeration.

Elements in the sun-web.xml File

This section describes the XML elements in the `sun-web.xml` file. Elements are grouped as follows:

This section also includes an alphabetical list of the elements for quick reference. See "Alphabetical List of sun-web.xml Elements" on page 146.

Note – Subelements must be defined in the order in which they are listed under each Subelements heading, unless otherwise noted.

Note – Each `sun-web.xml` file must begin with the following `DOCTYPE` header:

```
<!DOCTYPE sun-web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Sun Java System Web
Server 6.1 Servlet
2.3//EN''http://www.sun.com/software/sunone/webserver/dtds/sun-web-app_2_3-1.dtd'>
```

For an alphabetical list of elements in `sun-web.xml`, see "Alphabetical List of sun-web.xml Elements" on page 146

General Elements

General elements are as follows:

sun-web-app

Defines Sun Java System Web Server-specific configuration for a web application. This is the root element; there can only be one `sun-web-app` element in a `sun-web.xml` file.

Subelements

The following table describes subelements for the `sun-web-app` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–5 sun-web-app Subelements

Element	Required	Description
"sun-web-app" on page 114sun-web-app	zero or more	Maps roles to users or groups in the currently active realm.
"servlet" on page 118servlet	zero or more	Specifies a principal name for a servlet, which is used for the run-as role defined in `web.xml`.
"session-config" on page 120	zero or one	Specifies session manager, session cookie, and other session-related information.
"resource-env-ref" on page 126	zero or more	Maps the absolute JNDI name to the `resource-env-ref` in the corresponding J2SE XML file.
"resource-ref" on page 127	zero or more	Maps the absolute JNDI name to the `resource-ref` in the corresponding J2SE XML file.
"cache" on page 130	zero or one	Configures caching for web application components.
"class-loader" on page 140	zero or one	Specifies classloader configuration information.
"jsp-config" on page 141	zero or one	Specifies JSP configuration information.
"locale-charset-info" on page 144	zero or one	Specifies internationalization settings.
property"property" on page 116	zero or more	Specifies a property, which contains a name and a value.

Attributes

none

Properties

The following table describes properties for the `sun-web-app` element. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 6–6 sun-web-app Properties

Property Name	Default Value	Description
`crossContextAllowed`	`true`	If `true`, allows this web application to access the contexts of other web applications using the `ServletContext.getContext()` method.
`encodeCookies`	`true`	If `true`, Sun Java System Web Server URL encodes cookies before sending them to the client. If you do not want cookies to be encoded, add the following to `sun-web.xml`: `<property name="encodeCookies" value="false">` For the above example, enter the line directly under the `<sun-web-app>` tag; do not embed this in any other tag.
`tempdir`	*instance_dir*`/ClassCache/`*vs_id*`/uri`	Specifies a temporary directory for use by this web application. This value is used to construct the value of the `javax.servlet.context.tempdir` context attribute. Compiled JSPs are also placed in this directory.
`singleThreadedServletPoolSize`	`5`	Specifies the maximum number of servlet instances allocated for each `SingleThreadModel` servlet in the web application.
`reuseSessionID`	`false`	If `true`, this property causes the web application to reuse the `JSESSIONID` value (if present) in the request header as the session ID when creating sessions. The default behavior of web applications is to not reuse session IDs and instead generate cryptographically random session IDs for new sessions.
`relativeRedirectAllowed`	`false`	If `true`, allows the web application to send a relative URL to the client using the `HttpServletResponse.sendRedirect()` API (that is, it suppresses the container from translating a relative URL to a fully qualified URL).

property

Specifies a property, which contains a name and a value. A property adds configuration information to its parent element that is one or both of the following:

- Optional with respect to Sun Java System Web Server.

- Needed by a system or object that Sun Java System Web Server does not have knowledge of, such as an LDAP server or a Java class.

For example, a `manager-properties` element can include property subelements:

```
<manager-properties>
        <property name="reapIntervalSeconds" value="20" >
    </manager-properties>
```

Which properties a `manager-properties` element uses depends on the value of the parent `session-manager` element's `persistence-type` attribute. For details, see the description of the `session-manager` element.

Subelements

The following table describes subelements for the `property` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–7 property Subelements

Element	Required	Description
"description" on page 117	zero or one	Contains a text description of this element.

Attributes

The following table describes attributes for the `property` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–8 property Attributes

Attribute	Default	Description
`name`	none	Specifies the name of the property or variable.
`value`	none	Specifies the value of the property or variable.

description

Contains a text description of the parent element.

Subelements

none

Attributes

none

Security Elements

Security elements are as follows:

security-role-mapping

Maps roles to users or groups in the currently active realm.

Subelements

The following table describes subelements for the `security-role-mapping` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–9 security-role-mapping Subelements

Element	Required	Description
"role-name" on page 119	only one	Contains the role name.
"principal-name" on page 119	requires at least one `principal-name` or `group-name`	Contains a principal (user) name in the current realm.
"group-name" on page 120	requires at least one `principal-name` or `group-name`	Contains a group name in the current realm.

Attributes

none

servlet

Specifies a principal name for a servlet, which is used for the `run-as` role defined in `web-xml`.

Subelements

The following table describes subelements for the `servlet` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–10 servlet Subelements

Element	Required	Description
"servlet-name" on page 119	only one	Contains the name of a servlet, which is matched to a `servlet-name` in `web.xml`.
"principal-name" on page 119	only one	Contains a principal (user) name in the current realm.

Attributes

none

servlet-name

Contains data that specifies the name of a servlet, which is matched to a servlet-name in the `web.xml` file. This name must be present in the `web.xml` file.

Subelements

none

Attributes

none

role-name

Contains data that specifies the `role-name` in the `security-role` element of the `web.xml` file.

Subelements

none

Attributes

none

principal-name

Contains data that specifies a principal (user) name in the current realm.

Subelements

none

Attributes

none

group-name

Contains data that specifies a group name in the current realm.

Subelements

none

Attributes

none

Session Elements

Session elements are as follows:

Note – The session manager interface is unstable. An unstable interface may be experimental or transitional, and thus may change incompatibly, be removed, or be replaced by a more stable interface in the next release.

session-config

Specifies session configuration information.

Subelements

The following table describes subelements for the `session-config` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–11 session-config Subelements

Element	Required	Description
"session-manager" on page 121	zero or one	Specifies session manager configuration information.
"session-properties" on page 124	zero or one	Specifies session properties.
"cookie-properties" on page 125	zero or one	Specifies session cookie properties.

Attributes

none

session-manager

Specifies session manager information.

Note – In Sun Java System Web Server 6.1, you cannot define a session manager either for a single sign-on session or for a virtual server. You must define session managers at the level of web applications.

Subelements

The following table describes subelements for the `session-manager` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–12 session-manager Subelements

Element	Required	Description
"manager-properties" on page 122	zero or one	Specifies session manager properties.
"store-properties" on page 123	zero or one	Specifies session persistence (storage) properties.

Attributes

The following table describes attributes for the `session-manager` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–13 session-manager Attributes

Attribute	Default Value	Description
`persistence-type`	`memory`	(optional) Specifies the session persistence mechanism. Allowed values are `memory`, `file`, `s1ws60`, and `mmap`. Setting the value of persistence type to `memory` is equivalent to using Sun Java System Web Server's IWSSessionManager without any store. Setting the value of persistence type to `file` is equivalent to using Sun Java System Web Server's IWSSessionManager with `FileStore`.

manager-properties

Specifies session manager properties.

Subelements

The following table describes subelements for the `manager-properties` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–14 manager-properties Subelements

Element	Required	Description
"property" on page 116	zero or more	Specifies a property, which contains a name and a value.

Attributes

none

Properties

The following table describes properties for the `manager-properties` element. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 6–15 manager-properties Properties

Property Name	Default Value	Description
`reapIntervalSeconds`	`60`	Specifies the number of seconds between checks for expired sessions. Setting this value lower than the frequency at which session data changes is recommended. For example, this value should be as low as possible (1 second) for a hit counter servlet on a frequently accessed web site, or you could lose the last few hits each time you restart the server.
`maxSessions`	`-1`	Specifies the maximum number of active sessions, or `-1` (the default) for no limit.
`sessionFilename`	none; state is not preserved across restarts	Specifies the absolute or relative path name of the file in which the session state is preserved between application restarts, if preserving the state is possible. A relative path name is relative to the temporary directory for this web application. Applicable only if the `persistence-type` attribute of the "session-manager" on page 121 element is `memory`.

store-properties

Specifies session persistence (storage) properties.

Subelements

The following table describes subelements for the `store-properties` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–16 store-properties Subelements

Element	Required	Description
"property" on page 116	zero or more	Specifies a property, which contains a name and a value.

Attributes

none

Properties

The following table describes properties for the `store-properties` element. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 6–17 store-properties Properties

Property Name	Default Value	Description
`reapIntervalSeconds`	`60`	Specifies the number of seconds between checks for expired sessions for those sessions that are currently swapped out. Setting this value lower than the frequency at which session data changes is recommended. For example, this value should be as low as possible (1 second) for a hit counter servlet on a frequently accessed web site, or you could lose the last few hits each time you restart the server.
`directory`	directory specified by `javax.servlet.context.tempdir` context attribute	Specifies the absolute or relative path name of the directory into which individual session files are written. A relative path is relative to the temporary work directory for this web application.

session-properties

Specifies session properties.

Subelements

The following table describes subelements for the `session-properties` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–18 session-properties Subelements

Element	Required	Description
"property" on page 116	zero or more	Specifies a property, which contains a name and a value.

Attributes

none

Properties

The following table describes properties for the `session-properties` element. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 6–19 session-properties Properties

Property Name	Default Value	Description
`timeoutSeconds`	`600`	Specifies the default maximum inactive interval (in seconds) for all sessions created in this web application. If set to `0` or less, sessions in this web application do not expire. If a session-timeout element is specified in the `web.xml` file, the `session-timeout` value overrides any `timeoutSeconds` value. If neither `session-timeout` nor `timeoutSeconds` is specified, the `timeoutSeconds` default is used. Note that the `session-timeout` element in `web.xml` is specified in minutes, not seconds.
`enableCookies`	`true`	Uses cookies for session tracking if set to `true`.
`enableURLRewriting`	`true`	Enables URL rewriting. This provides session tracking via URL rewriting when the browser does not accept cookies. You must also use an `encodeURL` or `encodeRedirectURL` call in the servlet or JSP.

cookie-properties

Specifies session cookie properties.

Subelements

The following table describes subelements for the `cookie-properties` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–20 cookie-properties Subelements

Element	Required	Description
"property" on page 116	zero or more	Specifies a property, which contains a name and a value.

Attributes

none

Properties

The following table describes properties for the `cookie-properties` element. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 6–21 cookie-properties Properties

Property Name	Default Value	Description
`cookiePath`	context path at which the web application is installed.	Specifies the path name that is set when the session tracking cookie is created. The browser sends the cookie if the path name for the request contains this path name. If set to `/` (slash), the browser sends cookies to all URLs served by the Sun Java System Web Server. You can set the path to a narrower mapping to limit the request URLs to which the browser sends cookies.
`cookieMaxAgeSeconds`	`-1`	Specifies the expiration time (in seconds) after which the browser expires the cookie. The default value of `-1` indicates that the cookie never expires.
`cookieDomain`	(unset)	Specifies the domain for which the cookie is valid.
`cookieComment`	Sun Java System Web Server session tracking cookie	Specifies the comment that identifies the session tracking cookie in the cookie file. Applications can provide a specific comment for the cookie.

Reference Elements

Reference elements are as follows:

resource-env-ref

Maps the res-ref-name in the corresponding J2SE `web.xml` file "resource-env-ref" on page 126entry to the absolute `jndi-name` of a resource.

Subelements

The following table describes subelements for the `resource-env-ref` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–22 resource-env-ref Subelements

Element	Required	Description
"resource-env-ref-name" on page 127	only one	Specifies the `res-ref-name` in the corresponding J2SE `web.xml` file `resource-env-ref` entry.
"jndi-name" on page 129	only one	Specifies the absolute `jndi-name` of a resource.

Attributes

none

resource-env-ref-name

Contains data that specifies the res-ref-name in the corresponding J2SE `web.xml` file `resource-env-ref` entry.

Subelements

none

Attributes

none

resource-ref

Maps the "res-ref-name" on page 128e in the corresponding J2SE `web.xml` file `resource-ref` entry to the absolute "jndi-name" on page 129 of a resource.

Subelements

The following table describes subelements for the `resource-ref` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–23 resource-ref Subelements

Element	Required	Description
"res-ref-name" on page 128	only one	Specifies the `res-ref-name` in the corresponding J2SE `web.xml` file `resource-ref` entry.
"jndi-name" on page 129	only one	Specifies the absolute `jndi-name` of a resource.
"default-resource-principal" on page 128	zero or one	Specifies the default principal (user) for the resource.

Attributes

none

res-ref-name

Contains data that specifies the `res-ref-name` in the corresponding J2SE `web.xml` file `resource-ref` entry.

Subelements

none

Attributes

none

default-resource-principal

Specifies the default principal (user) for the resource.

If this element is used in conjunction with a JMS Connection Factory resource, the name and password subelements must be valid entries in Sun™ Java System Message Queue's broker user repository.

Subelements

The following table describes subelements for the `default-resource-principal` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–24 default-resource-principal Subelements

Element	Required	Description
"name" on page 129	only one	Contains the name of the principal.
"password" on page 129	only one	Contains the password for the principal.

Attributes

none

name

Contains data that specifies the name of the principal.

Subelements

none

Attributes

none

password

Contains data that specifies the password for the principal.

Subelements

none

Attributes

none

jndi-name

Contains data that specifies the absolute `jndi-name` of a URL resource or a resource in the `server.xml` file.

Note – To avoid collisions with names of other enterprise resources in JNDI, and to avoid portability problems, all names in a Sun Java System Web Server application should begin with the string `java:comp/env`.

Subelements

none

Attributes

none

Caching Elements

For details about response caching as it pertains to servlets, see “Caching Servlet Results” on page 37 “JSP Cache Tags” on page 50

Caching elements are as follows:

cache

Configures caching for web application components.

Subelements

The following table describes subelements for the `cache` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–25 cache Subelements

Element	Required	Description
“cache-helper” on page 132	zero or more	Specifies a custom class that implements the `CacheHelper` interface.
“default-helper” on page 133	zero or one	Allows you to change the properties of the default, built-in `cache-helper` class.

TABLE 6–25 cache Subelements *(Continued)*

Element	Required	Description
"property" on page 116	zero or more	Specifies a cache property, which contains a name and a value.
"cache-mapping" on page 134	zero or more	Maps a URL pattern or a servlet name to its cacheability constraints.

Attributes

The following table describes attributes for the `cache` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–26 cache Attributes

Attribute	Default Value	Description
`max-entries`	`4096`	(optional) Specifies the maximum number of entries the cache can contain. Must be a positive integer.
`timeout-in-seconds`	30	(optional) Specifies the maximum amount of time in seconds that an entry can remain in the cache after it is created or refreshed. Can be overridden by a timeout element.
`enabled`	false	(optional) Determines whether servlet and JSP caching is enabled. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`, `true`, `false`.

Properties

The following table describes properties for the `cache` element. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 6–27 cache Properties

Property Name	Default Value	Description
`cacheClassName`	`com.sun.appserv.web.cache.LruCache`	Specifies the fully qualified name of the class that implements the cache functionality. See The Table 6–28

TABLE 6–27 cache Properties *(Continued)*

Property Name	Default Value	Description
`MultiLRUSegmentSize`	`4096`	Specifies the number of entries in a segment of the cache table that should have its own LRU (least recently used) list. Applicable only if `cacheClassName` is set to `com.sun.appserv.web.cache.MultiLruCache`.
`MaxSize`	`unlimited; Long.MAX_VALUE`	Specifies an upper bound on the cache memory size in bytes (KB or MB units). Example values are 32 KB or 2 MB. Applicable only if `cacheClassName` is set to `com.sun.appserv.web.cache.BoundedMultiLruCache`.

Cache Class Names

The following table lists possible values of the `cacheClassName` property. The left column lists the value, and the right column describes the kind of cache the value specifies.

TABLE 6–28 cacheClassName Values

Value	Description
`com.sun.appserv.web.cache.LruCache`	A bounded cache with an LRU cache replacement policy.
`com.sun.appserv.web.cache.BaseCache`	An unbounded cache suitable if the maximum number of entries is known.
`com.sun.appserv.web.cache.MultiLruCache`	A cache suitable for a large number of entries (>4096). Uses the `MultiLRUSegmentSize` property.
`com.sun.appserv.web.cache.BoundedMultiLruCache`	A cache suitable for limiting the cache size by memory rather than number of entries. Uses the `MaxSize` property.

cache-helper

Specifies a class that implements the CacheHelper interface. For details, see "CacheHelper Interface" on page 40

Subelements

The following table describes subelements for the `cache-helper` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–29 cache-helper Subelements

Element	Required	Description
"property" on page 116	zero or more	Specifies a property, which contains a name and a value.

Attributes

The following table describes attributes for the `cache-helper` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–30 cache-helper Attributes

Attribute	Default Value	Description
`name`	default	Specifies a unique name for the helper class, which is referenced in the cache-mapping element.
`class-name`	none	Specifies the fully qualified class name of the cache helper, which must implement the `com.sun.appserv.web.CacheHelper` interface.

default-helper

Allows you to change the properties of the built-in default `cache-helper` class.

Subelements

The following table describes subelements for the `default-helper` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–31 default-helper Subelements

Element	Required	Description
"property" on page 116	zero or more	Specifies a property, which contains a name and a value.

Attributes

none

Properties

The following table describes properties for the `default-helper` element. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 6–32 default-helper Properties

Property Name	Default Value	Description
`cacheKeyGeneratorAttrName`	uses the built-in default `cache-helper` key generation, which concatenates the servlet path with `key-field` values, if any.	The caching engine searches in the `ServletContext` for an attribute with a name equal to the value specified for this property to determine whether a customized `CacheKeyGenerator` implementation is used. An application may provide a customized key generator rather than using the default helper. For more information, see CacheKeyGenerator Interface

cache-mapping

Maps a URL pattern or a servlet name to its cacheability constraints.

Subelements

The following table describes subelements for the `cache-mapping` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–33 cache-mapping Subelements

Element	Required	Description
"servlet-name" on page 119	requires one `servlet-name` or `url-pattern`	Contains the name of a servlet.
"url-pattern" on page 135	requires one `servlet-name` or `url-pattern`	Contains a servlet URL pattern for which caching is enabled.
"cache-helper-ref" on page 135	required if `timeout`, `refresh-field`, `http-method`, `key-field`, and `constraint-field` are not used	Contains the name of the `cache-helper` used by the parent `cache-mapping` element.

TABLE 6–33 cache-mapping Subelements *(Continued)*

Element	Required	Description
"timeout" on page 136	zero or one if `cache-helper-ref` is not used	Contains the `cache-mapping` specific maximum amount of time in seconds that an entry can remain in the cache after it is created or refreshed
"refresh-field" on page 136	zero or one if `cache-helper-ref` is not used	Specifies a field that gives the application component, programmatic way to refresh a cached entry.
"http-method" on page 137	zero or more if `cache-helper-ref` is not used	Contains an HTTP method that is eligible for caching.
"key-field" on page 137	zero or more if `cache-helper-ref` is not used	Specifies a component of the key used to look up and extract cache entries.
"constraint-field" on page 138	zero or more if `cache-helper-ref` is not used	Specifies a cacheability constraint for the given `url-pattern` or `servlet-name`.

Attributes

none

url-pattern

Contains data that specifies a servlet URL pattern for which caching is enabled. See the Java Servlet 2.3 specification, section SRV 11.2 for applicable patterns.

Subelements

none

Attributes

none

cache-helper-ref

Contains data that specifies the name of the `cache-helper` used by the parent `cache-mapping` element.

Subelements

none

Attributes

none

timeout

Contains data that specifies the `cache-mapping` specific maximum amount of time in seconds that an entry can remain in the cache after it is created or refreshed. If not specified, the default is the value of the `timeout` attribute of the `cache` element.

Subelements

none

Attributes

The following table describes attributes for the `timeout` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–34 timeout Attributes

Attribute	Default Value	Description
`name`	none	Specifies the timeout input parameter, whose value is interpreted in seconds. The field's type must be `java.lang.Long` or `java.lang.Integer`.
`scope`	`context.attribute`	(optional) Specifies the scope in which the input parameter can be present. Allowed values are `context.attribute`, `request.header`, `request.parameter`, `request.cookie`, `session.id`, and `session.attribute`.

refresh-field

Specifies a field that gives the application component a programmatic way to refresh a cached entry.

Subelements

none

Attributes

The following table describes attributes for the `refresh-field` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–35 refresh-field Attributes

Attribute	Default Value	Description
`name`	none	Specifies the input parameter name. If the parameter is present in the specified scope and it's value is `true`, the cache will be refreshed.
`scope`	`request.parameter`	(optional) Specifies the scope in which the input parameter can be present. Allowed values are `context.attribute`, `request.header`, `request.parameter`, `request.cookie`, `session.id`, and `session.attribute`.

http-method

Contains data that specifies an HTTP method that is eligible for caching. The default is `GET`.

Subelements

none

Attributes

none

key-field

Specifies a component of the key used to look up and extract cache entries. The web container searches for the named parameter, or field, in the specified scope.

If this element is not present, the web container uses the Servlet Path (the path section that corresponds to the servlet mapping that activated the current request). See the Servlet 2.3 specification, section SRV 4.4, for details on the Servlet Path.

Subelements

none

Attributes

The following table describes attributes for the `key-field` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–36 key-field Attributes

Attribute	Default Value	Description
`name`	none	Specifies the input parameter name.
`scope`	`request.parameter`	(optional) Specifies the scope in which the input parameter can be present. Allowed values are `context.attribute`, `request.header`, `request.parameter`, `request.cookie`, `session.id`, and `session.attribute`.

constraint-field

Specifies a cacheability constraint for the given `url-pattern` or `servlet-name`.

All constraint-field constraints must pass for a response to be cached. If there are value constraints, at least one of them must pass.

Subelements

The following table describes subelements for the `constraint-field` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–37 constraint-field Subelements

Element	Required	Description
"value" on page 139	zero or more	Contains a value to be matched to the input parameter value.

Attributes

The following table describes attributes for the `constraint-field` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–38 constraint-field Attributes

Attribute	Default Value	Description
`name`	none	Specifies the input parameter name.

TABLE 6–38 constraint-field Attributes *(Continued)*

Attribute	Default Value	Description
`scope`	`request.parameter`	(optional) Specifies the scope in which the input parameter can be present. Allowed values are `context.attribute`, `request.header`, `request.parameter`, `request.cookie`, `session.id`, and `session.attribute`.
`cache-on-match`	true	(optional) If `true`, caches the response if matching succeeds. Overrides the same attribute in a `value` subelement.
`cache-on-match-failure`	false	(optional) If `true`, caches the response if matching fails. Overrides the same attribute in a `value` subelement.

value

Contains data that specifies a value to be matched to the input parameter value. The matching is case sensitive. For example:

```
<value match-expr="in-range">1-60</value>
```

Subelements

none

Attributes

The following table describes attributes for the `value` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–39 value Attributes

Attribute	Default Value	Description
`match-expr`	`equals`	(optional) Specifies the type of comparison performed with the value. Allowed values are `equals`, `not-equals`, `greater`, `lesser`, and `in-range`. If `match-expr` is greater or lesser, the value must be a number. If `match-expr` is in-range, the value must be of the form *n1*-*n2*, where *n1* and *n2* are numbers.
`cache-on-match`	`true`	(optional) If `true`, caches the response if matching succeeds.

TABLE 6–39 value Attributes *(Continued)*

Attribute	Default Value	Description
`cache-on-match-failure`	false	(optional) If `true`, caches the response if matching fails.

Classloader Elements

Classloader elements are as follows:

- "class-loader" on page 140

class-loader

Configures the classloader for the web application.

Subelements

none

Attributes

The following table describes attributes for the `class-loader` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–40 class-loader Attributes

Attribute	Default Value	Description
`extra-class-path`	null	(optional) Specifies additional classpath settings for this web application.
`delegate`	`false`	(optional) If `true`, the web application follows the standard classloader delegation model and delegates to its parent classloader first before looking in the local classloader. If `false`, the web application follows the delegation model specified in the Servlet specification and searches in its classloader before looking in the parent classloader. For a web component of a web service, you must set this value to `true`. Legal values are `on`, `off`, `yes`, `no`, `1`, `0`, `true`, `false`.

TABLE 6–40 class-loader Attributes *(Continued)*

Attribute	Default Value	Description
`dynamic-reload-interval`	value of the `dynamicreloadinterval` attribute of the `<JAVA>` element in `server.xml`	(optional) Allows an application to override the `dynamicreloadinterval` setting in `server.xml`. Specifies the frequency (in seconds) at which a web application is checked for modifications, and then reloaded if modifications have been made. Setting this value to less than or equal to 0 disables dynamic reloading of the application. If not specified, the value from `server.xml` is used. For more information about `server.xml`, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

JSP Elements

JSP elements are as follows:

- "jsp-config" on page 141

jsp-config

Specifies JSP configuration information.

Subelements

The following table describes subelements for the `jsp-config` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–41 jsp-config Subelements

Element	Required	Description
"name" on page 129	zero or more	Specifies a property.

Attributes

none

Properties

The following table describes properties for the `jsp-config` element. The left column lists the property name, the middle column indicates the default value, and the right column describes what the property does.

TABLE 6–42 jsp-config Properties

Property Name	Default Value	Description
`ieClassId`	`clsid:8AD9C840-044E-11D1-B3E9-00805F499D93`	The Java plugin COM class ID for Internet Explorer. Used by the <jsp:plugin> tags.
`javaCompilerPlugin`	internal JDK compiler (`javac`)	The fully qualified class name of the Java compiler plugin to be used. Not needed for the default compiler. For example, to use the `jikes` compiler for JSP pages, set the `javaCompilerPlugin` property to `org.apache.jasper.compiler.JikesJavaCompiler`, then set the `javaCompilerPath` property to point to the `jikes` executable. To use `sun.tools.javac.Main` to compile JSP-generated servlets, set the `javaCompilerPlugin` property to `org.apache.jasper.compiler.SunJavaCompiler` (see also the `-deprecatedjavac` switch of jspc, described in "Compiling JSPs: The Command-Line Compiler" on page 47
`javaCompilerPath`	none	Specifies the path to the executable of an out-of-process Java compiler such as `jikes`. Ignored for the default compiler. It is required only if the `javaCompilerPlugin` property is specified.
`javaEncoding`	`UTF8`	Specifies the encoding for the generated Java servlet. This encoding is passed to the Java compiler used to compile the servlet as well. By default, the web container tries to use UTF8. If that fails, it uses the `javaEncoding` value. For encodings, see: `http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html`
`classdebuginfo`	`false`	Specifies whether the generated Java servlets should be compiled with the debug option set (`-g` for `javac`).
`keepgenerated`	`true`	If set to `true`, keeps the generated Java files. If `false`, deletes the Java files.
`largefile`	`false`	If set to `true`, static HTML is stored in a separate data file when a JSP is compiled. This is useful when a JSP is very large, because it minimizes the size of the generated servlet.

TABLE 6–42 jsp-config Properties *(Continued)*

Property Name	Default Value	Description
`mappedfile`	`false`	If set to `true`, generates separate write calls for each HTML line and comments that describe the location of each line in the JSP file. By default, all adjacent write calls are combined and no location comments are generated.
`scratchdir`	default work directory for the web application	The working directory created for storing all of the generated code.
`reload-interval`	`0`	Specifies the frequency (in seconds) at which JSP files are checked for modifications. Setting this value to 0 checks JSPs for modifications on every request. Setting this value to -1 disables checks for JSP modifications and JSP recompilation.
`initial-capacity`	`32`	Specifies the initial size of the hash table of compiled JSP classes.

The following example illustrates the use of the `initial-capacity` property described in the table above. The example shows how you can configure a value of 1024:

```
<jsp-config>
                <property name="initial-capacity" value="1024" >
</jsp-config>
```

Internationalization Elements

Internationalization elements are as follows:

- "parameter-encoding" on page 143
- "locale-charset-info" on page 144
- "locale-charset-map" on page 145

parameter-encoding

Specifies a hidden field or default charset that determines the character encoding the web container. This web container is used to decode parameters for `request.getParameter` calls when the charset is not set in the request's `Content-Type`.

For encodings you can use, see:

`http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html`

Subelements

none

Attributes

The following table describes attributes for the `parameter-encoding` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–43 parameter-encoding Attributes

Attribute	Default Value	Description
`form-hint-field`	`j_encoding`	The value of the hidden field in the form that specifies the parameter encoding.
`default-charset`	none	This value is used for parameter encoding if neither `request.setCharacterEncoding()` is called nor `form-hint-field` is found in the request.

locale-charset-info

Specifies the mapping between the locale and the character encoding that should be set in the `Content-type` header of the response if a servlet or JSP sets the response locale using the `ServletResponse.setLocale` method. This overrides the web container's default locale-to-charset mapping.

Subelements

The following table describes subelements for the `locale-charset-info` element. The left column lists the subelement name, the middle column indicates the requirement rule, and the right column describes what the element does.

TABLE 6–44 locale-charset-info Subelements

Element	Required	Description
"locale-charset-map" on page 145	one or more	Maps a locale to a character set.
"parameter-encoding" on page 143	zero or one	Deprecated. Use the `parameter-encoding` element under `sun-web-app` instead. This is supported only for backward compatibility with applications developed under Sun Java System Application Server 7.

Attributes

The following table describes attributes for the `locale-charset-info` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–45 locale-charset-info Attributes

Attribute	Default Value	Description
`default-locale`	none	Ignored in Sun Java System Web Server 6.1.

locale-charset-map

Maps a locale to a specific character encoding.

For encodings you can use, see:

`http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html`

Attributes

The following table describes attributes for the `locale-charset-map` element. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

TABLE 6–46 locale-charset-map Attributes

Attribute	Default Value	Description
`locale`	none	Specifies the locale name.
`agent`	none	Ignored in Sun Java System Web Server 6.1.
`charset`	none	Specifies the character set for that locale.

The following table provides a `locale-charset-map` example, listing the locale and the corresponding charset:

TABLE 6–47 locale-charset-map Example

Locale	Charset
ja	EUC-JP
zh	UTF-8

Alphabetical List of sun-web.xml Elements

This section provides an alphabetical list for the easy lookup of `sun-web.xml` elements.

"res-ref-name" on page 128

"resource-env-ref" on page 126

"resource-env-ref-name" on page 127

"resource-ref" on page 127

"role-name" on page 119

"security-role-mapping" on page 118

"servlet" on page 118

"servlet-name" on page 119

"session-config" on page 120

"session-manager" on page 121

"session-properties" on page 124

"store-properties" on page 123

"sun-web-app" on page 114

"timeout" on page 136

"url-pattern" on page 135

"value" on page 139

Note – For a list of `sun-web.xml` elements by category, see "Elements in the sun-web.xml File" on page 114.

Sample Web Application XML Files

This section includes the following:

Sample web.xml File

The following is a sample `web.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
     Copyright 2002 Sun Microsystems, Inc. All rights reserved.
-->
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web
     Application 2.3//EN' 'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
      <display-name>i18n-simple</display-name>
      <distributable></distributable>
      <filter>
            <filter-name>Simple Filter</filter-name>
            <filter-class>samples.i18n.simple.servlet.SimpleFilter
            </filter-class>
            <init-param>
                <param-name>encoding</param-name>
                <param-value>UTF-8</param-value>
             </init-param>
             <init-param>
                 <param-name>usefilter</param-name>
                 <param-value>true</param-value>
               </init-param>
      </filter>
      <filter-mapping>
          <filter-name>Simple Filter</filter-name>
          <url-pattern>/SimpleFilterServlet</url-pattern>
    </filter-mapping>
    <servlet>
         <servlet-name>SimpleI18nServlet</servlet-name>
         <servlet-class>samples.i18n.simple.servlet.SimpleI18nServlet
         </servlet-class>
         <load-on-startup>0</load-on-startup>
     </servlet>
     <servlet>
         <servlet-name>IncludedServlet</servlet-name>
         <servlet-class>samples.i18n.simple.servlet.IncludedServlet
         </servlet-class>
     </servlet>
     <servlet>
         <servlet-name>ForwardedServlet</servlet-name>
         <servlet-class>samples.i18n.simple.servlet.ForwardedServlet
         </servlet-class>
     </servlet>
     <servlet>
       <servlet-name>SimpleFilterServlet</servlet-name>
       <servlet-class>samples.i18n.simple.servlet.SimpleFilterServlet
       </servlet-class>
  </servlet>
```

```
  <servlet>
    <servlet-name>LocaleCharsetServlet</servlet-name>
    <servlet-class>samples.i18n.simple.servlet.LocaleCharsetServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
       <servlet-name>SimpleI18nServlet</servlet-name>
       <url-pattern>/SimpleI18nServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
           <servlet-name>IncludedServlet</servlet-name>
           <url-pattern>/IncludedServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
           <servlet-name>ForwardedServlet</servlet-name>
           <url-pattern>/ForwardedServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
           <servlet-name>SimpleFilterServlet</servlet-name>
           <url-pattern>/SimpleFilterServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
                <servlet-name>LocaleCharsetServlet</servlet-name>
                <url-pattern>/LocaleCharsetServlet</url-pattern>
  </servlet-mapping>
  <taglib>
                   <taglib-uri>/i18ntaglib</taglib-uri>
                   <taglib-location>/WEB-INF/tlds/i18ntaglib.tld
                   </taglib-location>
  </taglib>
</web-app>
```

Sample sun-web.xml File

The following is a sample `sun-web.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Copyright 2002 Sun Microsystems, Inc. All rights reserved.
-->

<!DOCTYPE sun-web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Sun ONE
Web Server 6.1 Servlet 2.3//EN' 'http://www.sun.com/software/sunone
/webserver/dtds/sun-web-app_2_3-1.dtd'>

<sun-web-app>
   <session-config>
```

```
            <session-manager/>
                      </session-config>
        <cache enabled="true" timeout-in-seconds="300" >
            <cache-mapping>
                      <servlet-name>ServCache</servlet-name>
                      <key-field name="inputtext"
                          scope="request.parameter"/>
                      <constraint-field name="inputtext"
                           scope="request.parameter">
                                    <value>one</value>
                                    <value>two</value>
                      </constraint-field>
                  </cache-mapping>
    </cache>
</sun-web-app>
```

◆ ◆ ◆ CHAPTER 7

Debugging Web Applications

This chapter provides guidelines for debugging web applications in Sun Java System Web Server.

This includes the following sections:

In order to debug applications, you need to edit the `server.xml` file as described in this chapter. For more general information about this file, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

Enabling Debugging

When you enable debugging, you need to enable both local and remote debugging.

You can enable debugging in one of these ways, as described in the following sections:

Sun Java System Application Server debugging is based on the JPDA (Java™ Platform Debugger Architecture software). For more information, see "JPDA Options" on page 152

Using the Administration Interface

▼ To enable debugging

1. **Access the Server Manager and click the Java tab.**
2. **Click JVM General.**
3. **Specify the Java home in the Java Home field.**
   The Java home is the path to the directory where the JDK is installed.
4. **To enable debugging, select On from the Debug Enabled drop-down list**
5. **Specify debug options in the Debug Options field.**
   For more information about debug options, see "JPDA Options" on page 152
6. **Click OK.**

Editing the server.xml File

To enable debugging, set the following attributes of the `JAVA` element in the `server.xml` file:

- Set `debug="true"` to turn on debugging.
- Add any desired JPDA debugging options in the `debugoptions` attribute. See "JPDA Options" on page 152
- To specify the port to use when attaching the JVM to a debugger, specify `address=`*port_number* in the `debugoptions` attribute.

For details about the `server.xml` file, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

JPDA Options

The default JPDA options are as follows:

```
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n
```

If you substitute `suspend=y`, the JVM starts in suspended mode and stays suspended until a debugger attaches to it. This is helpful if you want to start debugging as soon as the JVM starts.

To specify the port to use when attaching the JVM to a debugger, specify `address=`*port_number*.

You can include additional options. A list of JPDA debugging options is available here:

`http://java.sun.com/products/jpda/doc/conninv.html#Invocation`

Using Sun Java Studio for Debugging

You can use Sun™ Java System Studio 5 technology for "remote debugging", in order to manually attach the IDE to a remote Web Server started in debug mode.

▼ To manually attach the IDE to a remote Web Server

1. **Using the Sun Java System Web Server Administration interface, restart the server instance in debug mode (Server Manager > JVM General > Debug Enabled).**
2. **Note the JPDA port number.**
3. **Start the IDE.**
4. **Choose Debug > Start.**
5. **Select the `dt_socket` method, and then enter the remote machine name and the JPDA port number.**
6. **At that moment, any breakpoint created in the IDE on servlet source code of a deployed application will be active.**

   For more information about the Sun Java Studio 8 plugin for Sun Java System Web Server, and about using Sun Java Studio, see "Using Sun Java Studio" on page 107.

Debugging JSPs

When you use Sun Java Studio to debug JSPs, you can set breakpoints in either the JSP code or the generated servlet code, and you can switch between them and see the same breakpoints in both.

To set up debugging in Sun Java Studio, see the previous section.

Generating a Stack Trace for Debugging

You can generate a Java stack trace for debugging as described here:

If the `-Xrs` flag is set (for reduced signal usage) in the `server.xml` file (under `<JVMOPTIONS>`), comment it out before generating the stack trace. If the `-Xrs` flag is used, the server may simply dump core and restart when you send the signal to generate the trace.

The stack trace goes to the system log file or to stderr based on the `LOG` attributes in `server.xml`.

For more information about the `server.xml` file, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

Logging

You can use the Sun Java System Web Server's log files to help debug your applications. For general information about logging, see the *Sun Java System Web Server 6.1 SP11 Administrator's Guide*. For information about configuring logging in the `server.xml` file, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

You can change logging settings in one of these ways:

- "Using the Administration Interface" on page 154
- "Editing the server.xml File" on page 154

Using the Administration Interface

▼ To change logging settings

1. **Access the Server Manager and click the Logs tab.**
2. **Set log preferences as desired.**
3. **Apply your changes.**

Editing the server.xml File

To change logging settings, set the attributes of the `LOG` element in the `server.xml` file. For details about `server.xml` file, see the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

Profiling

You can use a profile to perform remote profiling on the Sun Java System Web Server to discover bottlenecks in server-side performance. This section describes how to configure these profilers for use with Sun Java System Web Server:

The HPROF Profiler

HPROF is a simple profiler agent shipped with the Java™ 2 SDK. It is a dynamically linked library that interacts with the JVMPI (Java™ Virtual Machine Profiler Interface) and writes out profiling information either to a file or to a socket in ASCII or binary format. This information can be further processed by a profiler front-end tool such as HAT.

HPROF can present CPU usage, heap allocation statistics, and monitor contention profiles. In addition, it can also report complete heap dumps and states of all of the monitors and threads in the Java virtual machine. For more details on the HPROF profiler, see the JDK documentation at:

`http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html`

Once HPROF is installed using the following instructions, its libraries are loaded into the server process.

▼ To use HPROF profiling on UNIX

1 Configure Sun Java System Web Server in one of these ways:

- Go to the server instance page in the Administration interface, select the Java tab, click the JVM Profiler link, and edit the following fields before clicking OK:
  - Profiler: Enable
    - Classpath: (leave blank)
    - Native Lib Path: (leave blank)
    - JVM Option: For each of these options, type the option in the JVM Option field, select Add, then check its box in the JVM Options list:

      `-Xrunhprof:file=log.txt,options`

  Edit the `server.xml` file:

```
<!-- hprof options -->
<PROFILER name="hprof" enabled="true">
    <JVMOPTIONS>
```

```
    -Xrunhprof:file=log.txt,options
     </JVMOPTIONS>
</PROFILER>
```

Note – Do not use the -Xrs flag.

Here is an example of options that you can use:

```
-Xrunhprof:file=log.txt,thread=y,depth=3
```

The `file` option is important because it determines where the stack dump is written in step 6.

The syntax of HPROF options is as follows:

```
-Xrunhprof[:help]|[:option=value,option2=value2, ...]
```

Using `help` lists options that can be passed to HPROF. The output is as follows:

```
Hprof usage: -Xrunhprof[:help]|[:<option>=<value>, ...]
Option Name and Value Description Default
--------------------- ----------- -------
heap=dump|sites|all heap profiling all
cpu=samples|old CPU usage off
format=a|b ascii or binary output a
file=<file> write data to file java.hprof
(.txt for ascii)
net=<host>:<port> send data over a socket write to file
depth=<size> stack trace depth 4
cutoff=<value> output cutoff point 0.0001
lineno=y|n line number in traces? y
thread=y|n thread in traces? n
doe=y|n dump on exit? y
```

2 You must also change a line in the Sun Java System Web Server start script. The start script file is *instance_dir*`/start`. Change the following line:

```
PRODUCT_BIN=webservd-wdog
```

to this:

```
PRODUCT_BIN=webservd
```

3 Start the server by running the start script. Since the server runs in the foreground (the change in step 2), the command prompt returns only after the server is stopped.

4 In another window or terminal, find the process ID of the server process.

```
% ps -ef | grep webservd
```

This command lists two `webservd` processes. Look at the PPID (parent process ID) column and identify which of the two processes is the parent process and which is the child process. Note the PID (process ID) of the child process ID.

5 **Send a SIGQUIT signal (signal 3) to the child process:**

```
% kill -QUIT child_PID
```

6 **To stop the Web Server, run the stop script from another window.**

```
% ./stop
```

This writes an HPROF stack dump to the file you specified using the file HPROF option in step 1. For general information about using a stack dump, see "Generating a Stack Trace for Debugging" on page 154

7 **To return your Web Server to its original configuration, undo the changes in steps 1 and 2.**

The Optimizeit Profiler

Information about Optimizeit™ is available at:

`http://www.borland.com/us/products/optimizeit/index.html`

Once Optimizeit is installed using the following instructions, its libraries are loaded into the server process.

To enable remote profiling with Optimizeit, do one of the following:

- Go to the server instance page in the Administration interface, select the Java tab, click the JVM Profiler link, and edit the following fields before selecting OK:
  - Profiler: Enable
  - Classpath: *Optimizeit_dir*`/lib/optit.jar`
  - Native Lib Path: *Optimizeit_dir*`/lib`
  - JVM Option: For each of these options, type the option in the JVM Option field, select Add, then check its box in the JVM Options list:
    - -DOPTITHOME=*Optimizeit_dir*
    - `-Xrunoii`
    - `-Xbootclasspath/a:`*Optimizeit_dir*`/lib/oibcp.jar`

    Edit the `server.xml` file:

```
<!-- Optimizeit options -->
<PROFILER classpath="Optimizeit_dir/lib/optit.jar"
    nativelibrarypath="Optimizeit_dir/lib" enabled="true">
```

```
    <JVMOPTIONS>
        -DOPTIT_HOME=Optimizeit_dir -Xboundthreads -Xrunoii
        -Xbootclasspath/a:Optimizeit_dir/lib/oibcp.jar
     </JVMOPTIONS>
</PROFILER>
```

In addition, you may need to set the following in your `server.policy` file:

```
grant codeBase "file:Optimizeit_dir/lib/optit.jar" {
permission java.security.AllPermission;
};
```

For more information about the `server.policy` file, see "The server.policy File" on page 100

When the server starts up with this configuration, you can attach the profiler. For further details, see the Optimizeit documentation.

Note – If any of the configuration options are missing or incorrect, the profiler may experience problems that affect the performance of the Sun Java System Web Server.

◆ ◆ ◆ APPENDIX A

Internationalization Issues

This appendix explains internationalization issues as it applies to the following components:

Servlets

This section describes how the Sun Java System Web Server determines the character encoding for the servlet request and the servlet response.

For encoding, see
`http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html`

Servlet Request

When processing the servlet request, the server uses the following order of precedence, first to last, to determine the character encoding for the request parameters:

- The `ServletRequest.setCharacterEncoding()` method.
- A hidden field in the form, if specified using the `form-hint-field` attribute of the `parameter-encoding` element in the `sun-web.xml` file. For more information about this element, see "parameter-encoding" on page 143
- The character encoding specified in the `default-charset` attribute of the `parameter-encoding` element in the `sun-web.xml` file. For more information about this element, see "parameter-encoding" on page 143
- The default encoding, which is ISO-8859-1.

Servlet Response

When processing a servlet response, the server uses the following order of precedence, first to last, to determine the response character encoding:

- The `ServletResponse.setContentType()` method or the `ServletResponse.setLocale()` method.
- The default encoding, which is ISO-8859-1.

To specify the character encoding that should be set in the `Content-type` header of the response if the response locale is set using the `ServletResponse.setLocale` method. You can use the `locale-charset-map` under the `locale-charset-info` element in the `sun-web.xml` file. For more information about this element, see"locale-charset-info" on page 144

JSPs

A JSP page uses a character encoding. For encodings you can use, see:

`http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html`

The encoding can be described explicitly using the `pageEncoding` attribute of the `page` directive. The character encoding defaults to the encoding indicated in the `contentType` attribute of the `page` directive if it is given, or to ISO-8859-1.

For more information, see the "Localization Issues" chapter of the JSP 1.2 specification, which you can be find at the following location:

`http://java.sun.com/products/jsp/index.jsp`

Also see the article *Developing Multilingual Applications Using JavaServer Pages Technology* at:

`http://java.sun.com/developer/technicalArticles/Intl/MultilingualJSP/index.html`

◆ ◆ ◆ APPENDIX B

Migrating Legacy Servlets

Netscape Enterprise Server/ iPlanet Web Server 4.0 and 4.1 supported the Java Servlet 2.1 specification. This specification did not include web applications. A deployment scheme was developed to make servlet deployment simpler. With the advent of Java web applications (`.war` files) and their deployment descriptors, it is no longer necessary to maintain a proprietary deployment system.

iPlanet Web Server 6.0 supported both types of deployment schemes, but the 4.x implementation (referred to as legacy servlets) was marked as deprecated (See Chapter 8: "Legacy Servlet and JSP Configuration" of the iPlanet Web Server, Enterprise Edition *Programmer's Guide to Servlets*).

Sun Java System Web Server 6.1 no longer supports legacy servlets. The legacy-style properties files for the server you want to migrate (`servlet.properties`, `context.properties`, and `rules.properties`) are removed during migration. For more information about these files, see "Appendix A" in the *Sun Java System Web Server 6.1 SP11 Administrator's Configuration File Reference*.

Because there is no one-to-one mapping for all of the features, legacy servlets cannot be migrated automatically. This section describes the main features involved in migrating legacy servlets to web applications.

This appendix includes the following topics:

JSP by Extension

In Sun Java System Web Server 6.1, JSP by extension works as it did in previous releases. Any file in the document tree that is named as an extension of `.jsp` will be treated as a JSP as long as the Java is turned on for the virtual server.

Servlet by Extension of Servlet by Directory

This is not supported in Sun Java System Web Server 6.1. You can deploy a web application to respond to a directory, but all of the servlets must be in the `WEB-INF/classes` directory of the web application. You can no longer copy a servlet in the `.class` file into the document tree and have it run as a servlet or have all of the contents of a directory run as a servlet. The web application will treat only `.class` files as servlets.

Registered Servlets

In the legacy servlet system there was a two-step process of registering servlets (`servlet.properties`) and mapping them to a URL (`rules.properties`). In Sun Java System Web Server 6.1, the servlets must be moved into a web application, and these settings will be maintained in the `web.xml` file of that web application.

Example

A registered servlet contains entries in both the `servlet.properties` and `rules.properties` files.

The following example uses a servlet file called `BuyNow1A.class`, which respons to `/buynow`. It is assumed that the web application is deployed at '`/`'.

The `servlet.properties` file has:

```
servlet.BuyNowServlet.classpath=
D:/Netscape/server4/docs/servlet/buy;D:/Netscape/server4/docs/myclasses
servlet.BuyNowServlet.code=BuyNow1A
servlet.BuyNowServlet.initArgs=arg1=45,arg2=online,arg3="quick shopping"
```

The `rules.properties` file has:

```
/buynow=BuyNowServlet
```

Those must be translated to a `web.xml` setting.

The `servlet.properties` setting will translate into the `<servlet>` element.

The classpath is automated so there is no classpath setting. All classes to be used must be in the WEB-INF/classes directory or in a .jar file in the WEB-INF/lib directory of the web application.

The servlet-name element is the part between the dots in the servlets.properties file. The code translates to the servlet-class. IntArgs translate to init-params. This entry would translate to:

```
<servlet>
    <servlet-name> BuyNowServlet </servlet-name>
    <servlet-class> BuyNow1A </servlet-class>
       <init-param>
          <param-name> arg1 </param-name>
          <param-value> 45 </param-value>
       </init-param>
       <init-param>
          <param-name> arg2 </param-name>
          <param-value> online </param-value>
       </init-param>
       <init-param>
          <param-name> arg3 </param-name>
          <param-value> “quick shopping” </param-value>
       </init-param>
</servlet
```

The rules.properties entries translate to servlet-mapping elements. This entry would translate to.

```
<servlet-mapping>
    <servlet-name> BuyNowServlet </servlet-name>
    <url-pattern> /buynow </url-pattern>
</servlet-mapping>
```

Some other entries in the servlets.properties file map to the web.xml file. These include.

- Servlets.startup: The servlet listed here should have a load-on-startup element in it.
- Servlets.config.reloadInterval: This translates to the dynamicreloadinterval attribute of the JAVA element in server.xml. This is an instance-wide setting so it affects all virtual servers and all web applications.
- Servlets.sessionmgr: This translates to the session-manager element in the sun-web.xml file.

Index

D

E

F

G

H

I

J

R

S

T

U

V

W