Copyright

Copyright © 2001 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks


All other trademarks are the property of their respective companies.

Programming WebLogic HTTP Servlets

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Document Date</th>
<th>Software Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>December 19, 2001</td>
<td>BEA WebLogic Server Version 6.1</td>
</tr>
</tbody>
</table>
Contents

About This Document

Audience

e-docs Web Site

How to Print the Document

Related Information

Contact Us!

Documentation Conventions

1. Overview of HTTP Servlets

What Is a Servlet?

What You Can Do with Servlets

Overview of Servlet Development

Servlets and J2EE

WebLogic Server 6.1 with J2EE 1.2 and J2EE 1.3 Functionality

WebLogic Server 6.1 with J2EE 1.2 Plus Additional J2EE 1.3 Features.

HTTP Servlet API Reference

2. Introduction to Programming

Writing a Simple HTTP Servlet

Advanced Features

Complete HelloWorldServlet Example

3. Programming Tasks

Initializing a Servlet

Initializing a Servlet when WebLogic Server Starts

Overriding the init() Method

Programming WebLogic HTTP Servlets
Providing an HTTP Response ................................................................. 3-4
Retrieving Client Input .............................................................................. 3-6
   Securing Client Input in Servlets.............................................................. 3-8
      Using a WebLogic Server Utility Method ....................................... 3-8
   Methods for Using the HTTP Request .................................................... 3-9
Example: Retrieving Input by Using Query Parameters ....................... 3-10
Session Tracking from a Servlet............................................................... 3-12
   A History of Session Tracking .............................................................. 3-12
   Tracking a Session with an HttpSession Object............................... 3-13
   Lifetime of a Session .......................................................................... 3-14
   How Session Tracking Works .............................................................. 3-15
Detecting the Start of a Session ............................................................. 3-15
Setting and Getting Session Name/Value Attributes ............................. 3-16
Logging Out and Ending a Session .......................................................... 3-17
   Using session.invalidate() for a Single Web Application ................. 3-17
   Implementing Single Sign-On for Multiple Applications ................. 3-17
   Exempting a Web Application for Single Sign-on ......................... 3-18
Configuring Session Tracking ................................................................. 3-18
Using URL Rewriting Instead of Cookies ................................................ 3-18
URL Rewriting and Wireless Access Protocol (WAP) ............................ 3-20
Making Sessions Persistent ...................................................................... 3-20
   Scenarios to Avoid When Using Sessions ........................................... 3-21
   Use Serializable Attribute Values ...................................................... 3-21
   Configuring Session Persistence ......................................................... 3-22
Using Cookies in a Servlet ...................................................................... 3-22
   Setting Cookies in an HTTP Servlet ...................................................... 3-22
   Retrieving Cookies in an HTTP Servlet .............................................. 3-23
Using Cookies That Are Transmitted by Both HTTP and HTTPS ........... 3-24
   Secure Cookies .................................................................................. 3-25
   Application Security and Cookies ....................................................... 3-25
Using WebLogic Services from an HTTP Servlet ...................................... 3-26
Accessing Databases .............................................................................. 3-26
   Connecting to a Database Using a JDBC Connection Pool .............. 3-27
   Using a Connection Pool in a Servlet .................................................. 3-27
   Connecting to a Database Using a DataSource Object ....................... 3-28
Using a DataSource in a Servlet ....................................................... 3-29
Connecting Directly to a Database Using a JDBC Driver ....................... 3-29
Threading Issues in HTTP Servlets ..................................................... 3-30
SingleThreadModel............................................................................. 3-30
Shared Resources ................................................................................ 3-31
Dispatching Requests to Another Resource ............................................. 3-31
Forwarding a Request ......................................................................... 3-33
Including a Request ............................................................................... 3-33

4. Administration and Configuration
    Overview of WebLogic HTTP Servlet Administration.......................... 4-2
    Using Deployment Descriptors to Configure and Deploy Servlets ......... 4-2
        web.xml (Web Application Deployment Descriptor) ...................... 4-2
        weblogic.xml (Weblogic-Specific Deployment Descriptor) ............. 4-3
    WebLogic Server Administration Console ...................................... 4-4
    Directory Structure for Web Applications ......................................... 4-5
    Referencing a Servlet in a Web Application ....................................... 4-5
    Servlet Security ................................................................................ 4-6
        Authentication ............................................................................. 4-6
        Authorization (Security Constraints) ............................................. 4-7
    Servlet Development Tips ................................................................. 4-7
    Clustering Servlets............................................................................ 4-8

Index
About This Document

This document provides information on programming and deploying WebLogic HTTP Servlets.

The document is organized as follows:

- Chapter 1, “Overview of HTTP Servlets,” provides an overview of Hypertext Transfer Protocol (HTTP) servlet programming and explains how to use HTTP servlets with WebLogic Server.
- Chapter 2, “Introduction to Programming,” introduces basic HTTP servlet programming.
- Chapter 3, “Programming Tasks,” provides information about writing HTTP servlets in a WebLogic Server environment.
- Chapter 4, “Administration and Configuration,” provides information about writing HTTP servlets in a WebLogic Server environment.

Audience

This document is written for application developers who want to build e-commerce applications using HTTP servlets and the Java 2 Platform, Enterprise Edition (J2EE) from Sun Microsystems. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.
e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.


Related Information

- Package javax.servlet

- Package javax.servlet.http

- Servlet 2.2 specification
  (http://java.sun.com/products/servlet/download.html#specs)
Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.
### Convention | Usage
--- | ---
Ctrl+Tab | Keys you press simultaneously.
*italics* | Emphasis and book titles.
*monospace* *text* | Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard. *Examples:*
```java
import java.util.Enumeration;
chmod u+w *
config/examples/applications
.java
config.xml
float
```
*monospace* *italic* *text* | Variables in code. *Example:*
```java
String CustomerName;
```
**UPPERCASE** **TEXT** | Device names, environment variables, and logical operators. *Examples:*
LPT1
BEA_HOME
OR

( ) | A set of choices in a syntax line. *Example:*
```java
java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]
```
[ ] | Optional items in a syntax line. *Example:*
```java
java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}
```
### Convention | Usage
--- | ---
... | Indicates one of the following in a command line:
- An argument can be repeated several times in the command line.
- The statement omits additional optional arguments.
- You can enter additional parameters, values, or other information

. | Indicates the omission of items from a code example or from a syntax line.
.
.
Programming WebLogic HTTP Servlets
CHAPTER

1 Overview of HTTP Servlets

The following sections provide an overview of Hypertext Transfer Protocol (HTTP) servlet programming and explain how to use HTTP servlets with WebLogic Server:

- What Is a Servlet?
- What You Can Do with Servlets
- Overview of Servlet Development
- Servlets and J2EE
- HTTP Servlet API Reference

What Is a Servlet?

A servlet is a Java class that runs in a Java-enabled server. An HTTP servlet is a special type of servlet that handles an HTTP request and provides an HTTP response, usually in the form of an HTML page. The most common use of WebLogic HTTP Servlets is to create interactive applications using standard Web browsers for the client-side presentation while WebLogic Server handles the business logic as a server-side process. WebLogic HTTP servlets can access databases, Enterprise JavaBeans, messaging APIs, HTTP sessions, and other facilities of WebLogic Server.

WebLogic Server fully supports HTTP servlets as defined in the Servlet 2.3 Proposed Final Draft, version 1 specification from Sun Microsystems. HTTP servlets form an integral part of the Java 2 Enterprise Edition (J2EE) standard.
1 Overview of HTTP Servlets

What You Can Do with Servlets

- Create dynamic Web pages that use HTML forms to get end-user input and provide HTML pages that respond to that input. Examples of this utilization include online shopping carts, financial services, and personalized content.

- Create collaborative systems such as online conferencing.

- Servlets running in WebLogic Server have access to a variety of APIs and services. For example:
  - Session tracking—Allows a Web site to track a user’s progress across multiple Web pages. This functionality supports Web sites such as e-commerce sites that use shopping carts. WebLogic Server supports session persistence to a database, providing fail-over between server down time and session sharing between clustered servers. For more information see “Session Tracking from a Servlet” on page 3-12.
  - JDBC drivers (including BEA)—JDBC drivers provide basic database access. With Weblogic Server’s multitier JDBC implementations, you can take advantage of connection pools, server-side data caching, and transactions. For more information see “Accessing Databases” on page 3-26.
  - Security—You can apply various types of security to servlets, including using ACLs for authentication and Secure Sockets Layer (SSL) to provide secure communications.
  - Enterprise JavaBeans—Servlets can use Enterprise JavaBeans (EJB) to encapsulate sessions, data from databases, and other functionality.
  - Java Messaging Service (JMS)—JMS allows your servlets to exchange messages with other servlets and Java programs.
  - Java JDK APIs—Servlets can use the standard Java JDK APIs.
  - Forwarding requests—Servlets can forward a request to another servlet or other resource.

- Servlets written for any J2EE-compliant servlet engine can be easily deployed on WebLogic Server.

- Servlets and Java Server Pages (JSP) can work together to create an application.
Overview of Servlet Development

- Programmers of HTTP servlets utilize a standard API from JavaSoft, javax.servlet.http, to create interactive applications.

- HTTP servlets can read HTTP headers and write HTML coding to deliver a response to a browser client.

- Servlets are deployed on WebLogic Server as part of a Web Application. A Web Application is a grouping of application components such as servlet classes, JavaServer Pages (JSP), static HTML pages, images, and security. For more information see “Administration and Configuration” on page 4-1.

Servlets and J2EE

The Servlet 2.2 specification (available at http://java.sun.com/products/servlet/download.html#specs), part of the Java 2 Platform, Enterprise Edition, defines the implementation of the servlet API and the method by which servlets are deployed in enterprise applications. Deploying servlets on a J2EE-compliant server, such as WebLogic Server, is accomplished by packaging the servlets and other resources that make up an enterprise application into a single unit called a Web Application. A Web Application utilizes a specific directory structure to contain its resources and a deployment descriptor that defines how these resources interact and how the application is accessed by a client. A Web Application may also be deployed as an archive file called a .war file.

For more information on creating Web Applications, see Assembling and Configuring Web Applications at http://e-docs.bea.com/wls/docs61/webapp/index.html. For an overview of servlet administration and deployment issues, see “Administration and Configuration” on page 4-1.

Note:
The Servlet 2.3 specification is part of the J2EE 1.3 specification. To use J2EE 1.3 features, please see “WebLogic Server 6.1 with J2EE 1.2 and J2EE 1.3 Functionality” on page 1-4.

WebLogic Server 6.1 with J2EE 1.2 and J2EE 1.3 Functionality

BEA WebLogic Server 6.1 is the first e-commerce transaction platform to implement advanced J2EE 1.3 features. To comply with the rules governing J2EE, BEA Systems provides two separate downloads: one with J2EE 1.3 features enabled, and one that is limited to J2EE 1.2 features only. Both downloads offer the same container and differ only in the APIs that are available.

WebLogic Server 6.1 with J2EE 1.2 Plus Additional J2EE 1.3 Features

With this download, WebLogic Server defaults to running with J2EE 1.3 features enabled. These features include EJB 2.0, JSP 1.2, Servlet 2.3, and J2EE Connector Architecture 1.0. When you run WebLogic Server 6.1 with J2EE 1.3 features enabled, J2EE 1.2 applications are still fully supported. WebLogic Server 6.1 supports the Proposed Final Draft version 1 of the Servlet 2.3 Specification with the following exceptions:

The <run-as> element is not supported.

The Proposed Final Draft version 1 differs from the Final version in the following ways:

1. The Filter interface that we support has following methods:
   doFilter(ServletRequest,ServletResponse,FilterChain)
   getFilterConfig() Return the FilterConfig for this Filter.
   setFilterConfig(FilterConfig) The container calls this method when the Filter is instantiated and passes in a Filter-Config

2. The following listener classes have been renamed. The listeners which were affected were:
   old: javax.servlet.ServletContextAttributesListener;
   new: javax.servlet.ServletContextAttributeListener;
old: javax.servlet.http.HttpSessionAttributesListener;
new: javax.servlet.http.HttpSessionAttributeListener;

3. The getServletContext() method has been added to HttpSession
4. The ServletContext.getResourcePaths() method has been added to the directory argument
5. The status code 307 (temporary redirect) has been added to HttpServletResponse

WebLogic Server 6.1 with J2EE 1.2 Certification

With this download, WebLogic Server defaults to running with J2EE 1.3 features disabled and is fully compliant with the J2EE 1.2 specification and regulations.

HTTP Servlet API Reference

WebLogic Server supports the javax.servlet.http package in the Java Servlet 2.2 API. You can find additional documentation for the package from Sun Microsystems:

- API documentation
  - Package javax.servlet
  - Package javax.servlet.http

- Servlet 2.2 specification
  (http://java.sun.com/products/servlet/download.html#specs)
1 Overview of HTTP Servlets
Chapter 2

Introduction to Programming

The following sections introduce basic HTTP servlet programming:

- Writing a Simple HTTP Servlet
- Advanced Features
- Complete HelloWorldServlet Example

Writing a Simple HTTP Servlet

The section provides a procedure for writing a simple HTTP servlet, which prints out the message Hello World. A complete code example (the HelloWorldServlet) illustrating these steps is included at the end of this section. Additional information about using various J2EE and Weblogic Server services such as JDBC, RMI, and JMS, in your servlet are discussed later in this document.

1. Import the appropriate package and classes, including the following:
   ```java
   import javax.servlet.*;
   import javax.servlet.http.*;
   import java.io.*;
   ```

2. Extend javax.servlet.http.HttpServlet. For example:
   ```java
   public class HelloWorldServlet extends HttpServlet{
   ```

3. Implement a service() method.
The main function of a servlet is to accept an HTTP request from a Web browser, and return an HTTP response. This work is done by the \texttt{service()} method of your servlet. Service methods include \texttt{response} objects used to create output and \texttt{request} objects used to receive data from the client.

You may have seen other servlet examples implement the \texttt{doPost()} and/or \texttt{doGet()} methods. These methods reply only to POST or GET requests; if you want to handle all request types from a single method, your servlet can simply implement the \texttt{service()} method. (However, if you choose to implement the \texttt{service()} method, you cannot implement the \texttt{doPost()} or \texttt{doGet()} methods, unless you call \texttt{super.service()} at the beginning of the \texttt{service()} method.) The HTTP servlet specification describes other methods used to handle other request types, but all of these methods are collectively referred to as \textit{service} methods.

All the service methods take the same parameter arguments. An \texttt{HttpServletRequest} provides information about the request, and your servlet uses an \texttt{HttpServletResponse} to reply to the HTTP client. The service method looks like the following:

\begin{verbatim}
public void service(HttpServletRequest req,
        HttpServletResponse res) throws IOException
{

4. Set the content type, as follows:
        res.setContentType("text/html");

5. Get a reference to a \texttt{java.io.PrintWriter} object to use for output, as follows:
        PrintWriter out = res.getWriter();

6. Create some HTML using the \texttt{println()} method on the \texttt{PrintWriter} object, as shown in the following example:
        out.println("<html><head><title>Hello World!</title></head>");
        out.println("<body><h1>Hello World!</h1></body></html>");
    }
}

7. Compile the servlet, as follows:

a. Set up a development environment shell (see \url{http://e-docs.bea.com/wls/docs61/programming/environment.html}) with the correct classpath and path settings.
\end{verbatim}
b. From the directory containing the Java source code for your servlet, compile your servlet into the \texttt{WEB-INF/classes} directory of the Web Application that contains your servlet. For example:

```
javac -d /myWebApplication/WEB-INF/classes myServlet.java
```

8. Deploy the servlet as part of a Web Application hosted on WebLogic Server. For an overview of servlet deployment, see “Administration and Configuration” on page 4-1.

9. Call the servlet from a browser.

The URL you use to call a servlet is determined by: (a) the name of the Web Application containing the servlet and (b) the name of the servlet as mapped in the deployment descriptor of the Web Application. Request parameters can also be included in the URL used to call a servlet.

Generally the URL for a servlet conforms to the following:

```
http://host:port/webApplicationName/mappedServletName?parameter
```

The components of the URL are defined as follows:

- \textit{host} is the name of the machine running WebLogic Server.
- \textit{port} is the port at which the above machine is listening for HTTP requests.
- \textit{webApplicationName} is the name of the Web Application containing the servlet.
- \textit{parameters} are one or more name-value pairs containing information sent from the browser that can be used in your servlet.

For example, to use a Web browser to call the HelloWorldServlet (the example featured in this document), which is deployed in the examplesWebApp and served from a WebLogic Server running on your machine, enter the following URL:

```
http://localhost:7001/examplesWebApp/HelloWorldServlet
```

The \textit{host:port} portion of the URL can be replaced by a DNS name that is mapped to WebLogic Server.
Advanced Features

The preceding steps create a basic servlet. You will probably also use more advanced features of servlets:

- Handling HTML form data—HTTP servlets can receive and process data received from a browser client in HTML forms.
  - “Retrieving Client Input” on page 3-6
- Application design—HTTP servlets offer many ways to design your application. The following sections provide detailed information about writing servlets:
  - “Providing an HTTP Response” on page 3-4
  - “Threading Issues in HTTP Servlets” on page 3-30
  - “Dispatching Requests to Another Resource” on page 3-31
- Initializing a servlet—if your servlet needs to initialize data, accept initialization arguments, or perform other actions when the servlet is initialized, you can override the `init()` method.
  - “Initializing a Servlet” on page 3-1
- Use of sessions and persistence in your servlet—sessions and persistence allow you to track your users within and between HTTP sessions. Session management includes the use of cookies. For more information, see the following sections:
  - “Session Tracking from a Servlet” on page 3-12
  - “Using Cookies in a Servlet” on page 3-22
  - “Configuring Session Persistence” on page 3-22
- Use of WebLogic services in your servlet—WebLogic Server provides a variety of services and APIs that you can use in your Web applications. These services include Java Database Connectivity (JDBC) drivers, JDBC database connection pools, Java Messaging Service (JMS), Enterprise JavaBeans (EJB), and Remote Method Invocation (RMI). For more information, see the following sections:
  - “Using WebLogic Services from an HTTP Servlet” on page 3-26
  - “Servlet Security” on page 4-6
Complete HelloWorldServlet Example

This section provides the complete Java source code for the example used in the preceding procedure. The example is a simple servlet that provides a response to an HTTP request. Later in this document, this example is expanded to illustrate how to use HTTP parameters, cookies, and session tracking.

Listing 2-1  HelloWorldServlet.java

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

public class HelloWorldServlet extends HttpServlet {
    public void service(HttpServletRequest req,
                       HttpServletResponse res)
                       throws IOException {
        // Must set the content type first
        res.setContentType("text/html");
        // Now obtain a PrintWriter to insert HTML into
        PrintWriter out = res.getWriter();

        out.println("<html><head><title>" + "Hello World!" + "</title></head>");
        out.println("<body><h1>Hello World!</h1></body></html>");
    }
}
```

You can find the source code and instructions for compiling and running all the examples used in this document in the samples/examples/servlets directory of your WebLogic Server distribution.
Introduction to Programming
3 Programming Tasks

The following sections describe how to write HTTP servlets in a WebLogic Server environment:

- Initializing a Servlet
- Providing an HTTP Response
- Retrieving Client Input
- Session Tracking from a Servlet
- Using Cookies in a Servlet
- Using WebLogic Services from an HTTP Servlet
- Accessing Databases
- Threading Issues in HTTP Servlets
- Dispatching Requests to Another Resource

Initializing a Servlet

Normally, WebLogic Server initializes a servlet when the first request is made for the servlet. Subsequently, if the servlet is modified, the `destroy()` method is called on the existing version of the servlet. Then, after a request is made for the modified servlet, the `init()` method of the modified servlet is executed. For more information, see “Servlet Development Tips” on page 4-7.
When a servlet is initialized, WebLogic Server executes the `init()` method of the servlet. Once the servlet is initialized, it is not initialized again until you restart WebLogic Server or the servlet code when the servlet is modified. If you choose to override the `init()` method, your servlet can perform certain tasks, such as establishing database connections, when the servlet is initialized. (See “Overriding the init() Method” on page 3-3)

**Initializing a Servlet when WebLogic Server Starts**

Rather than having WebLogic Server initialize a servlet when the first request is made for it, you can first configure WebLogic Server to initialize a servlet when the server starts. You do this by specifying the servlet class in the `<load-on-startup>` element in the Web Application deployment descriptor. For more information see “Servlet Element” at http://e-docs.bea.com/wls/docs61/webapp/web_xml.html#web_xml_servlet.

You can pass parameters to an HTTP servlet during initialization by defining these parameters in the Web Application containing the servlet. You can use these parameters to pass values to your servlet every time the servlet is initialized without having to rewrite the servlet. For more information, see Writing Web Application Deployment Descriptors at http://e-docs.bea.com/wls/docs61/webapp/webappdeployment.html.

For example, the following entries in the Web Application deployment descriptor define two initialization parameters: `greeting`, which has a value of `Welcome` and `person`, which has a value of `WebLogic Developer`.

```xml
<servlet>
  ...
  <init-param>
    <param-name>greeting</param-name>
    <param-value>Welcome</param-value>
    <description>The salutation</description>
  </init-param>
  <init-param>
    <param-name>person</param-name>
    <param-value>WebLogic Developer</param-value>
    <description>name</description>
  </init-param>
</servlet>
```
To retrieve initialization parameters, call the `getInitParameter(String name)` method from the parent `javax.servlet.GenericServlet` class. When passed the name of the parameter, this method returns the parameter’s value as a `String`.

### Overriding the `init()` Method

You can have your servlet execute tasks at initialization time by overriding the `init()` method. The following code fragment reads the `<init-param>` tags that define a greeting and a name in the Web Application deployment descriptor:

```java
String defaultGreeting;
String defaultName;

public void init(ServletConfig config) throws ServletException {
    if ((defaultGreeting = getInitParameter("greeting")) == null)
        defaultGreeting = "Hello";
    if ((defaultName = getInitParameter("person")) == null)
        defaultName = "World";
}
```

The values of each parameter are stored in the class instance variables `defaultGreeting` and `defaultName`. The first code tests whether the parameters have null values, and if null values are returned, provides appropriate default values.

You can then use the `service()` method to include these variables in the response. For example:

```java
out.print("<body><h1>\n");
out.println(defaultGreeting + "$\" + defaultName + "$!\n");
out.println("</h1></body></html>\n");
```

The full source code and instructions for compiling, installing, and trying out an example called `HelloWorld2.java`, which illustrates the use of the `init()` method, can be found in the `samples/examples/servlets` directory of your WebLogic Server distribution.

The `init()` method of a servlet does whatever initialization work is required when WebLogic Server loads the servlet. The default `init()` method does all of the initial work that WebLogic Server requires, so you do not need to override it unless you have special initialization requirements. If you do override `init()`, first call `super.init()` so that the default initialization actions are done first.
Providing an HTTP Response

This section describes how to provide a response to the client in your HTTP servlet. Deliver all responses by using the HttpServletResponse object that is passed as a parameter to the service() method of your servlet.

1. Configure the HttpServletResponse.

Using the HttpServletResponse object, you can set several servlet properties that are translated into HTTP header information:

- **At a minimum**, set the content type using the setContentType() method before you obtain the output stream to which you write the page contents. For HTML pages, set the content type to text/html. For example:

  ```java
  res.setContentType("text/html");
  ```

- (optional) You can also use the setContentType() method to set the character encoding. For example:

  ```java
  res.setContentType("text/html;ISO-88859-4");
  ```

- Set header attributes using the setHeader() method. For dynamic responses, it is useful to set the “Pragma” attribute to no-cache, which causes the browser to always reload the page and ensures the data is current. For example:

  ```java
  res.setHeader("Pragma", "no-cache");
  ```

2. Compose the HTML page.

The response that your servlet sends back to the client must look like regular HTTP content, essentially formatted as an HTML page. Your servlet returns an HTTP response through an output stream that you obtain using the response parameter of the service() method. To send an HTTP response:

a. Obtain an output stream by using the HttpServletResponse object and one of the methods shown in the following two examples:

  - PrintWriter out = res.getWriter();
  - ServletOutputStream out = res.getOutputStream();

  You can use both PrintWriter and ServletOutputStream in the same servlet (or in another servlet that is included in a servlet). The output of both is written to the same buffer.
b. Write the contents of the response to the output stream using the `print()` method. You can use HTML tags in these statements. For example:

```java
out.print("<html><head><title>My Servlet</title>");
out.print("</head><body><h1>");
out.print("Welcome"");
out.print("</h1></body></html>");
```

Any time you print data that a user has previously supplied, BEA recommends that you remove any HTML special characters that a user might have entered. If you do not remove these characters, your Web site could be exploited by cross-site scripting. See “Securing Client Input in Servlets” on page 3-8.

Do not close the output stream by using the `close()` method, and avoid flushing the contents of the stream. If you do not close or flush the output stream, WebLogic Server can take advantage of persistent HTTP connections, as described in the next step.

3. Optimize the response.

By default, WebLogic Server attempts to use HTTP persistent connections whenever possible. A persistent connection attempts to reuse the same HTTP TCP/IP connection for a series of communications between client and server. Application performance improves because a new connection need not be opened for each request. Persistent connections are useful for HTML pages containing many in-line images, where each requested image would otherwise require a new TCP/IP connection.

Using the WebLogic Server Administration Console, you can configure the amount of time that WebLogic Server keeps an HTTP connection open. For more information, see “Duration” (see [http://e-docs.bea.com/wls/docs61/ConsoleHelp/virtualhost.html#KeepAliveSecs](http://e-docs.bea.com/wls/docs61/ConsoleHelp/virtualhost.html#KeepAliveSecs)).

WebLogic Server must know the length of the HTTP response in order to establish a persistent connection and automatically adds a `Content-Length` property to the HTTP response header. In order to determine the content length, WebLogic Server must buffer the response. However, if your servlet explicitly flushes the `ServletOutputStream`, WebLogic Server cannot determine the length of the response and therefore cannot use persistent connections. For this reason, you should avoid explicitly flushing the HTTP response in your servlets.

You may decide that, in some cases, it is better to flush the response early to display information in the client before the page has completed; for example, to
display a banner advertisement while some time-consuming page content is calculated. Conversely, you may want to increase the size of the buffer used by the servlet engine to accommodate a larger response before flushing the response. You can manipulate the size of the response buffer by using the related methods of the javax.servlet.ServletException interface (at http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/ServletResponse.html).

The WebLogic Server servlet engine now merges chunked data automatically. The user must continue to set the context length, and the Transfer-encoding parameter remains chunked, but the servlet or jsp code no longer needs to decode the chunked data. This operation has been taken over by the servlet engine itself.

## Retrieving Client Input

The HTTP servlet API provides a interface for retrieving user input from Web pages.

An HTTP request from a Web browser can contain more than the URL, such as information about the client, the browser, cookies, and user query parameters. Use query parameters to carry user input from the browser. Use the GET method appends parameters to the URL address, and the POST method includes them in the HTTP request body.

HTTP servlets need not deal with these details; information in a request is available through the HttpServletRequest object and can be accessed using the request.getParameter() method, regardless of the send method.

Read the following for more detailed information about the ways to send query parameters from the client:

- Encode the parameters directly into the URL of a link on a page. This approach uses the GET method for sending parameters. The parameters are appended to the URL after a ? character. Multiple parameters are separated by a & character. Parameters are always specified in name=value pairs so the order in which they are listed is not important. For example, you might include the following link in a Web page, which sends the parameter color with the value purple to an HTTP servlet called ColorServlet:
Retrieving Client Input

Manually enter the URL, with query parameters, into the browser location field. This is equivalent to clicking the link shown in the previous example.

Query the user for input with an HTML form. The contents of each user input field on the form are sent as query parameters when the user clicks the form’s Submit button. Specify the method used by the form to send the query parameters (POST or GET) in the `<FORM>` tag using the `METHOD="GET|POST"` attribute.

Query parameters are always sent in `name=value` pairs, and are accessed through the `HttpServletRequest` object. You can obtain an `Enumeration` of all parameter names in a query, and fetch each parameter value by using its parameter name. A parameter usually has only one value, but it can also hold an array of values. Parameter values are always interpreted as `Strings`, so you may need to cast them to a more appropriate type.

The following sample from a `service()` method examines query parameter names and their values from a form. Note that `request` is the `HttpServletRequest` object.

```java
Enumeration params = request.getParameterNames();
String paramName = null;
String[] paramValues = null;

while (params.hasMoreElements()) {
    paramName = (String) params.nextElement();
    paramValues = request.getParameterValues(paramName);
    System.out.println("Parameter name is " + paramName);
    for (int i = 0; i < paramValues.length; i++) {
        System.out.println("\tvalue " + i + " is " + paramValues[i].toString());
    }
}
```
Securing Client Input in Servlets

This ability to retrieve and return user-supplied data can present a security vulnerability called cross-site scripting, which can be exploited to steal a user’s security authorization. For a detailed description of cross-site scripting, refer to “Understanding Malicious Content Mitigation for Web Developers” (a CERT security advisory) at http://www.cert.org/tech_tips/malicious_code_mitigation.html.

To remove the security vulnerability, before you return data that a user has supplied, scan the data for any of the HTML special characters in Table 3-1. If you find any special characters, replace them with their HTML entity or character reference. Replacing the characters prevents the browser from executing the user-supplied data as HTML.

<table>
<thead>
<tr>
<th>Replace this special character:</th>
<th>With this entity/character reference:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>{</td>
<td>&amp;40;</td>
</tr>
<tr>
<td>}</td>
<td>&amp;41;</td>
</tr>
<tr>
<td>#</td>
<td>&amp;35;</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;38;</td>
</tr>
</tbody>
</table>

Using a WebLogic Server Utility Method

WebLogic Server provides the weblogic.servlet.security.Utils.encodeXSS() method to replace the special characters in user-supplied data. To use this method, provide the user-supplied data as input. For example, to secure the user-supplied data in Listing 3-1, replace the following line:

```java
out.print(defaultGreeting + " " + name + "!");
```

with:

```java
weblogic.servlet.security.Utils.encodeXSS(defaultGreeting + " " + name + "!");
```
with the following:

```java
out.print(defaultGreeting + " " +
weblogic.security.servlet.encodeXSS(name) + "!");
```

To secure an entire application, you must use the `encodeXSS()` method **each time** you return user-supplied data. While the previous example in Listing 3-1 is an obvious location in which to use the `encodeXSS()` method, Table 3-2 describes other locations to consider.

### Table 3-2 Code that Returns User-Supplied Data

<table>
<thead>
<tr>
<th>Page Type</th>
<th>User-Supplied Data</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error page</td>
<td>Erroneous input string, invalid URL,</td>
<td>An error page that says &quot;username is not</td>
</tr>
<tr>
<td></td>
<td>username</td>
<td>permitted access.&quot;</td>
</tr>
<tr>
<td>Status page</td>
<td>Username, summary of input from previous</td>
<td>A summary page that asks a user to confirm</td>
</tr>
<tr>
<td></td>
<td>pages</td>
<td>input from previous pages.</td>
</tr>
<tr>
<td>Database display</td>
<td>Data presented from a database</td>
<td>A page that displays a list of database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>entries that have been previously entered</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by a user.</td>
</tr>
</tbody>
</table>

### Methods for Using the HTTP Request

This section defines the methods of the `javax.servlet.HttpServletRequest` interface that you can use to get data from the request object. You should keep the following limitations in mind:

- You cannot read request parameters using any of the `getParameter()` methods described in this section and then attempt to read the request with the `getInputStream()` method.
- You cannot read the request with `getInputStream()` and then attempt to read request parameters with one of the `getParameter()` methods.

If you attempt either of the preceding procedures, an `IllegalStateException` is thrown.

You can use the following methods of `javax.servlet.HttpServletRequest` to retrieve data from the request object:

```java
HttpServletRequest.getMethod()
```

Allows you to determine the request method, such as `GET` or `POST`. 
HttpServletRequest.getQueryString()
   Allows you to access the query string. (The remainder of the requested URL,
   following the ? character.)

HttpServletRequest.getParameter()
   Returns the value of a parameter.

HttpServletRequest.getParameterNames()
   Returns an array of parameter names.

HttpServletRequest.getParameterValues()
   Returns an array of values for a parameter.

HttpServletRequest.getInputStream()
   Reads the body of the request as binary data. If you call this method after
   reading the request parameters with getParameter(),
   getParameterNames(), or getParameterValues(), an
   illegalStateException is thrown.

Example: Retrieving Input by Using Query Parameters

In this example, the HelloWorld2.java servlet example is modified to accept a
username as a query parameter, in order to display a more personal greeting. (For the
complete code, see the HelloWorld3.java servlet example, located in the
samples/examples/servlets directory of your WebLogic Server distribution.) The
service() method is shown here.

Listing 3-1  Retrieving Input with the service() Method

```java
public void service(HttpServletRequest req,
                   HttpServletResponse res)
   throws IOException
{
   String name, paramName[];
   if ((paramName = req.getParameterValues("name"))
      != null) {
      name = paramName[0];
   } else {
      name = defaultName;
   }
```
Retrieving Client Input

```java
// Set the content type first
res.setContentType("text/html");
// Obtain a PrintWriter as an output stream
PrintWriter out = res.getWriter();

out.print("<html><head><title>" +
        "Hello World!" + </title></head>");
out.print("<body><h1>");
out.print(defaultGreeting + " " + name + "!");
out.print("</h1></body></html>");
```

The `getParameterValues()` method retrieves the value of the name parameter from the HTTP query parameters. You retrieve these values in an array of type `String`. A single value for this parameter is returned and is assigned to the first element in the name array. If the parameter is not present in the query data, `null` is returned; in this case, `name` is assigned to the default name that was read from the `<init-param>` by the `init()` method.

Do not base your servlet code on the assumption that parameters are included in an HTTP request. The `getParameter()` method has been deprecated; as a result, you might be tempted to shorthand the `getParameterValues()` method by tagging an array subscript to the end. However, this method can return `null` if the specified parameter is not available, resulting in a `NullPointerException`.

For example, the following code triggers a `NullPointerException`:

```java
String myStr = req.getParameterValues("paramName")[0];
```

Instead, use the following code:

```java
if ((String myStr[] =
    req.getParameterValues("paramName")==null) {
   // Now you can use the myStr[0];
} else {
   // paramName was not in the query parameters!
}
```
Session Tracking from a Servlet

Session tracking enables you to track a user’s progress over multiple servlets or HTML pages, which, by nature, are stateless. A session is defined as a series of related browser requests that come from the same client during a certain time period. Session tracking ties together a series of browser requests—think of these requests as pages—that may have some meaning as a whole, such as a shopping cart application.

The following sections discuss various aspects of tracking sessions from an HTTP servlet:

- A History of Session Tracking
- Tracking a Session with an HttpSession Object
- Lifetime of a Session
- How Session Tracking Works
- Detecting the Start of a Session
- Setting and Getting Session Name/Value Attributes
- Logging Out and Ending a Session
- Configuring Session Tracking
- Using URL Rewriting Instead of Cookies
- URL Rewriting and Wireless Access Protocol (WAP)
- Making Sessions Persistent

A History of Session Tracking

Before session tracking matured conceptually, developers tried to build state into their pages by stuffing information into hidden fields on a page or embedding user choices into URLs used in links with a long string of appended characters. You can see good examples of this at most search engine sites, many of which still depend on CGI. These sites track user choices with URL parameter name=value pairs that are appended to
the URL, after the reserved HTTP character ?. This practice can result in a very long URL that the CGI script must carefully parse and manage. The problem with this approach is that you cannot pass this information from session to session. Once you lose control over the URL—that is, once the user leaves one of your pages—the user information is lost forever.

Later, Netscape introduced browser cookies, which enable you to store user-related information about the client for each server. However, some browsers still do not fully support cookies, and some users prefer to turn off the cookie option in their browsers. Another factor that should be considered is that most browsers limit the amount of data that can be stored with a cookie.

Unlike the CGI approach, the HTTP servlet specification defines a solution that allows the server to store user details on the server beyond a single session, and protects your code from the complexities of tracking sessions. Your servlets can use an HttpSession object to track a user’s input over the span of a single session and to share session details among multiple servlets. Session data can be persisted using a variety of methods available with WebLogic Service.

Tracking a Session with an HttpSession Object

According to the Java Servlet API, which WebLogic Server implements and supports, each servlet can access a server-side session by using its HttpSession object. You can access an HttpSession object in the service() method of the servlet by using the HttpServletRequest object with the variable request variable, as shown:

```java
HttpSession session = request.getSession(true);
```

An HttpSession object is created if one does not already exist for that client when the request.getSession(true) method is called with the argument true. The session object lives on WebLogic Server for the lifetime of the session, during which the session object accumulates information related to that client. Your servlet adds or removes information from the session object as necessary. A session is associated with a particular client. Each time the client visits your servlet, the same associated HttpSession object is retrieved when the getSession() method is called.

For more details on the methods supported by the HttpSession, refer to the HttpServlet API at http://java.sun.com/j2ee/j2adkee/techdocs/api/javax/servlet/http/HttpSession.html.
In the following example, the `service()` method counts the number of times a user requests the servlet during a session.

```java
public void service(HttpServletRequest request,
            HttpServletResponse, response)
    throws IOException
{
    // Get the session and the counter param attribute
    HttpSession session = request.getSession (true);
    Integer ival = (Integer)
        session.getAttribute("simplesession.counter");
    if (ival == null) // Initialize the counter
        ival = new Integer (1);
    else // Increment the counter
        ival = new Integer (ival.intValue () + 1);
    // Set the new attribute value in the session
    session.setAttribute("simplesession.counter", ival);
    // Output the HTML page
    out.print("<HTML><body>");
    out.print("<center> You have hit this page ");
    out.print(ival + " times!");
    out.print("</body></html>"));
}
```

**Lifetime of a Session**

A session tracks the selections of a user over a series of pages in a single transaction. A single transaction may consist of several tasks, such as searching for an item, adding it to a shopping cart, and then processing a payment. A session is transient, and its lifetime ends when one of the following occurs:

- A user leaves your site and the user’s browser does not accept cookies.
- A user quits the browser.
- The session is timed out due to inactivity.
- The session is completed and invalidated by the servlet.
- The user logs out and is invalidated by the servlet.
For more persistent, long-term storage of data, your servlet should write details to a
database using JDBC or EJB and associate the client with this data using a long-lived
cookie and/or username and password. Although this document states that sessions use
cookies and persistence internally, you should not use sessions as a general mechanism
for storing data about a user.

How Session Tracking Works

How does WebLogic Server know which session is associated with each client? When
an HttpSession is created in a servlet, it is associated with a unique ID. The browser
must provide this session ID with its request in order for the server to find the session
data again. The server attempts to store this ID by setting a cookie on the client. Once
the cookie is set, each time the browser sends a request to the server it includes the
cookie containing the ID. The server automatically parses the cookie and supplies the
session data when your servlet calls the getSession() method.

If the client does not accept cookies, the only alternative is to encode the ID into the
URL links in the pages sent back to the client. For this reason, you should always use
the encodeURL() method when you include URLs in your servlet response.
WebLogic Server detects whether the browser accepts cookies and does not
unnecessarily encode URLs. WebLogic automatically parses the session ID from an
encoded URL and retrieves the correct session data when you call the getSession() method. Using the encodeURL() method ensures no disruption to your servlet code,
regardless of the procedure used to track sessions. For more information, see “Using
URL Rewriting Instead of Cookies” on page 3-18.

Detecting the Start of a Session

After you obtain a session using the getSession(true) method, you can tell whether
the session has just been created by calling the HttpSession.isNew() method. If this
method returns true, then the client does not already have a valid session, and at this
point it is unaware of the new session. The client does not become aware of the new
session until a reply is posted back from the server.
Design your application to accommodate new or existing sessions in a way that suits your business logic. For example, your application might redirect the client’s URL to a login/password page if you determine that the session has not yet started, as shown in the following code example:

```java
HttpSession session = request.getSession(true);
if (session.isNew()) {
    response.sendRedirect(welcomeURL);
}
```

On the login page, provide an option to log in to the system or create a new account. You can also specify a login page in your Web Application. For more information, see [login-config at http://e-docs.bea.com/wls/docs61/webapp/web_xml.html#login-config](http://e-docs.bea.com/wls/docs61/webapp/web_xml.html#login-config).

### Setting and Getting Session Name/Value Attributes

You can store data in an HttpSession object using name=value pairs. Data stored in a session is available through the session. To store data in a session, use these methods from the HttpSession interface:

- `getAttribute`
- `getAttributeNames`
- `setAttribute`
- `removeAttribute`

The following code fragment shows how to get all the existing name=value pairs:

```java
Enumeration sessionNames = session.getAttributeNames();
String sessionName = null;
Object sessionValue = null;
while (sessionNames.hasMoreElements()) {
    sessionName = (String)sessionNames.nextElement();
    sessionValue = session.getAttribute(sessionName);
    System.out.println("Session name is ", value is ");
}
```

To add or overwrite a named attribute, use the `setAttribute()` method. To remove a named attribute altogether, use the `removeAttribute()` method.
**Session Tracking from a Servlet**

**Note:** You can add any Java descendant of `Object` as a session attribute and associate it with a name. However, if you are using session persistence, your attribute value objects must implement `java.io.Serializable`.

---

**Logging Out and Ending a Session**

If your application deals with sensitive information, consider offering the ability to log out of the session. This is a common feature when using shopping carts and Internet email accounts. When the same browser returns to the service, the user must log back in to the system.

**Using `session.invalidate()` for a Single Web Application**

User authentication information is stored both in the user’s session data and in the context of a server or virtual host that is targeted by a Web Application. Using the `session.invalidate()` method, which is often used to log out a user, only invalidates the current session for a user—the user’s authentication information still remains valid and is stored in the context of the server or virtual host. If the server or virtual host is hosting only one Web Application, the `session.invalidate()` method, in effect, logs out the user.

Do not reference an invalidated session after calling `session.invalidate()`. If you do, an `IllegalStateException` is thrown. The next time a user visits your servlet from the same browser, the session data will be missing, and a new session will be created when you call the `getSession(true)` method. At that time you can send the user to the login page again.

**Implementing Single Sign-On for Multiple Applications**

If the server or virtual host is targeted by many Web Applications, another means is required to log out a user from all Web Applications. Because the Servlet specification does not provide an API for logging out a user from all Web Applications, the following methods are provided.

```java
weblogic.servlet.security.ServletAuthentication.logout()
```

Removes the authentication data from the user’s session data, which logs out a user but allows the session to remain alive.
3 Programming Tasks

```java
weblogic.servlet.security.ServletAuthentication.invalidateAll()
Removes the authentication data from the session data, and from the server or
virtual host context. The cookie is also invalidated.

weblogic.servlet.security.ServletAuthentication.killCookie()
Invalidates the current cookie by setting the cookie so that it expires
immediately when the response is sent to the browser. This method depends
on a successful response reaching the user’s browser. The session remains
alive until it times out.
```

Exempting a Web Application for Single Sign-on

If you want to exempt a Web Application from participating in single sign-on, define
a different cookie name for the exempted Web Application. For more information, see
Configuring Session Cookies at
http://e-docs.bea.com/wls/docs61/webapp/sessions.html#session-coo
kie.

Configuring Session Tracking

WebLogic Server provides many configurable attributes that determine how
WebLogic Server handles session tracking. For details about configuring these session
tracking attributes, see “Session descriptor” at
http://e-docs.bea.com/wls/docs61/webapp/weblogic_xml.html#session
-descriptor.

Using URL Rewriting Instead of Cookies

In some situations, a browser may not accept cookies, which means that session
tracking with cookies is not possible. URL rewriting is a workaround to this scenario
that can be substituted automatically when WebLogic Server detects that the browser
does not accept cookies. URL rewriting involves encoding the session ID into the
hyperlinks on the Web pages that your servlet sends back to the browser. When the
user subsequently clicks these links, WebLogic Server extracts the ID from the URL
and finds the appropriate HttpSession. Then you use the getSession() method to
access session data.
To enable URL rewriting in WebLogic Server, set the `UrlRewritingEnabled` attribute to true in the "Session descriptor" element of the WebLogic-specific deployment descriptor (at http://e-docs.bea.com/wls/docs61/webapp/weblogic_xml.html#session-descriptor).

To make sure your code correctly handles URLs in order to support URL rewriting, consider the following guidelines:

- You should avoid writing a URL straight to the output stream, as shown here:
  ```java
go out.println("<a href="/myshop/catalog.jsp">catalog</a>");
```

  Instead, use the `HttpServletResponse.encodeURL()` method. For example:
  ```java
go out.println("<a href="" + response.encodeURL("myshop/catalog.jsp") + ">");
```

- Calling the `encodeURL()` method determines if the URL needs to be rewritten and, if necessary, rewrites the URL by including the session ID in the URL.

- Encode URLs that send redirects, as well as URLs that are returned as a response to WebLogic Server. For example:
  ```java
going if (session.isNew())
    response.sendRedirect(response.encodeRedirectUrl(welcomeURL));
```

  WebLogic Server uses URL rewriting when a session is new, even if the browser accepts cookies, because the server cannot determine, during the first visit of a session, whether the browser accepts cookies.

  Your servlet may determine whether a given session was returned from a cookie by checking the Boolean returned from the `HttpServletRequest.isRequestedSessionIdFromCookie()` method. Your application may respond appropriately, or it may simply rely on URL rewriting by WebLogic Server.

**Note:** The CISCO Local Director load balancer expects a question mark "?" delimiter for URL rewriting. Because the WLS URL-rewriting mechanism uses a semicolon ";" as the delimiter, our URL re-writing is incompatible with this load balancer.
URL Rewriting and Wireless Access Protocol (WAP)

If you are writing a WAP application, you must use URL rewriting because the WAP protocol does not support cookies. In addition, some WAP devices impose a 128-character limit (including parameters) on the length of a URL, which limits the amount of data that can be transmitted using URL rewriting. To allow more space for parameters, you can limit the size of the session ID that is randomly generated by WebLogic Server by specifying the number of bytes with the IDLength attribute in the <session-descriptor> element of the WebLogic-specific deployment descriptor, weblogic.xml (see http://e-docs.bea.com/wls/docs61/webapp/weblogic_xml.html#session-descriptor).

The minimum value is 8 bytes; the default value is 52 bytes; the maximum value is Integer.MAX_VALUE.

Making Sessions Persistent

You can set up WebLogic Server to record session data in a persistent store. If you are using session persistence, you can expect the following characteristics:

- Good failover, because sessions are saved when servers fail.
- Better load balancing, because any server can handle requests for any number of sessions, and use caching to optimize performance. For more information, see the cacheEntries property, under “Configuring session persistence” at http://e-docs.bea.com/wls/docs61/webapp/sessions.html#session-persistence.
- Sessions can be shared across clustered WebLogic Servers. Note that session persistence is no longer a requirement in a WebLogic Cluster. Instead, you can use in-memory replication of state. For more information, see Using WebLogic Server Clusters at http://e-docs.bea.com/wls/docs61/cluster/index.html.
- For customers who want the highest in servlet session persistence, JDBC-based persistence is the best choice. For customers who want to sacrifice some amount of session persistence in favor of drastically better performance, in-memory replication is the appropriate choice. JDBC-based persistence is noticeably
slower than in-memory replication. In some cases, in-memory replication has
outperformed JDBC-based persistence for servlet sessions by a factor of eight.

- You can put any kind of Java object into a session, but only objects that are
  java.io.Serializable can be stored in a session. For more information, see
  “Configuring session persistence” at
  http://e-docs.bea.com/wls/docs61/webapp/sessions.html#session-persistence.

Scenarios to Avoid When Using Sessions

Do not use session persistence for storing long-term data between sessions. In other
words, do not rely on a session still being active when a client returns to a site at some
later date. Instead, your application should record long-term or important information
in a database.

Sessions are not a convenience wrapper around cookies. Do not attempt to store
long-term or limited-term client data in a session. Instead, your application should
create and set its own cookies on the browser. Examples include an auto-login feature
that allows a cookie to live for a long period, or an auto-logout feature that allows a
cookie to expire after a short period of time. Here, you should not attempt to use HTTP
sessions. Instead, you should write your own application-specific logic.

The format of the session id is specified internally, and may change from one version
of WebLogic Server to another. For this reason, BEA Systems recommends that you
do not create applications which require a specific session id format.

Use Serializable Attribute Values

When you use persistent sessions, all attribute value objects that you add to the
session must implement java.io.Serializable. For more details on writing
serializable classes, refer to the online java tutorial about serializable objects at
http://java.sun.com/docs/books/tutorial/essential/io/providing.html. If you add your own serializable classes to a persistent session,
make sure that each instance variable of your class is also serializable. Otherwise, you
can declare it as transient, and WebLogic Server does not attempt to save that
variable to persistent storage. One common example of an instance variable that must
be made transient is the HttpSession object. (See the notes on using serialized
objects in sessions in the section “Making Sessions Persistent” on page 3-20.)
3 Programming Tasks

The HttpServletRequest, ServletContext, and HttpSession attributes will be serialized when a WebLogic Server instance detects a change in the web application classloader. The classloader changes when a webapp is redeployed, when there is a dynamic change in a servlet, or when there is a cross webapp forward or include.

To avoid having the attribute serialized during a dynamic change in a servlet, turn off servlet-relad-check-secs in weblogic.xml. There is no way to avoid serialization of attributes for cross webapp dispatch or webapp redeployment.

Configuring Session Persistence

For details about setting up persistent sessions, see “Configuring session persistence” at http://e-docs.bea.com/wls/docs61/webapp/sessions.html#session-persistence.

Using Cookies in a Servlet

A cookie is a piece of information that the server asks the client browser to save locally on the user’s disk. Each time the browser visits the same server, it sends all cookies relevant to that server with the HTTP request. Cookies are useful for identifying clients as they return to the server.

Each cookie has a name and a value. A browser that supports cookies generally allows each server domain to store up to 20 cookies of up to 4k per cookie.

Setting Cookies in an HTTP Servlet

To set a cookie on a browser, create the cookie, give it a value, and add it to the HttpServletRequest object that is the second parameter in your servlet’s service method. For example:
Cookie myCookie = new Cookie("ChocolateChip", "100");
myCookie.setMaxAge(Integer.MAX_VALUE);
response.addCookie(myCookie);

This example shows how to add a cookie called ChocolateChip with a value of 100 to the browser client when the response is sent. The expiration of the cookie is set to the largest possible value, which effectively makes the cookie last forever. Because cookies accept only string-type values, you should cast to and from the desired type that you want to store in the cookie. When using EJBs, a common practice is to use the home handle of an EJB instance for the cookie value and to store the user’s details in the EJB for later reference.

Retrieving Cookies in an HTTP Servlet

You can retrieve a cookie object from the HttpServletRequest that is passed to your servlet as an argument to the service() method. The cookie itself is presented as a javax.servlet.http.Cookie object.

In your servlet code, you can retrieve all the cookies sent from the browser by calling the getCookies() method. For example:

Cookie[] cookies = request.getCookies();

This method returns an array of all cookies sent from the browser, or null if no cookies were sent by the browser. Your servlet must process the array in order to find the correct named cookie. You can get the name of a cookie using the Cookie.getName() method. It is possible to have more than one cookie with the same name, but different path attributes. If your servlets set multiple cookies with the same names, but different path attributes, you also need to compare the cookies by using the Cookie.getPath() method. The following code illustrates how to access the details of a cookie sent from the browser. It assumes that all cookies sent to this server have unique names, and that you are looking for a cookie called ChocolateChip that may have been set previously in a browser client.

Cookie[] cookies = request.getCookies();
boolean cookieFound = false;
for(int i=0; i < cookies.length; i++) {
    thisCookie = cookies[i];
    if (thisCookie.getName().equals("ChocolateChip")) {
        cookieFound = true;
        break;
    }
if (cookieFound) {
    // We found the cookie! Now get its value
    int cookieOrder = String.parseInt(thisCookie.getValue());
}

For more details on cookies, see:


Using Cookies That Are Transmitted by Both HTTP and HTTPS

Because HTTP and HTTPS requests are sent to different ports, some browsers may not include the cookie sent in an HTTP request with a subsequent HTTPS request (or vice-versa). This may cause new sessions to be created when servlet requests alternate between HTTP and HTTPS. To ensure that all cookies set by a specific domain are sent to the server every time a request in a session is made, set the CookieDomain attribute to the name of the domain. Set the CookieDomain attribute with the <session-descriptor> element of the WebLogic-specific deployment descriptor (weblogic.xml) for the Web Application that contains your servlet. For example:

```xml
<session-descriptor>
  <session-param>
    <param-name>CookieDomain</param-name>
    <param-value>mydomain.com</param-value>
  </session-param>
</session-descriptor>
```

The CookieDomain attribute instructs the browser to include the proper cookie(s) for all requests to hosts in the domain specified by mydomain.com. For more information about this property or configuring session cookies, see “Setting Up Session Management” at http://e-docs.bea.com/wls/docs61/webapp/sessions.html#session-management.
Secure Cookies

As of Service Pack 2, we have introduced a new parameter that can be used to mark the sessionCookie as secure. If set, the client's browser will only send the cookie back over an HTTPS connection. This ensures that the cookie ID is secure and should only be used on websites that exclusively use HTTPS. Once this feature is enabled, session cookies over HTTP will no longer work; if your client is directed to a non-HTTPS location the session will not be sent. It is highly advisable to turn off URLRewriting if you use this feature; if your application attempts to encode URLs, the session ID will be shared over HTTP. Add the following to your weblogic.xml to use this feature:

```xml
<session-param>
  <param-name>CookieSecure</param-name>
  <param-value>true</param-value>
</session-param>
```

Application Security and Cookies

Using cookies that enable automatic account access on a machine is convenient, but can be undesirable from a security perspective. When designing an application that uses cookies, follow these guidelines:

- Do not assume that a cookie is always correct for a user. Sometimes machines are shared or the same user may want to access a different account.

- Allow your users to make a choice about leaving cookies on the server. On shared machines, users may not want to leave automatic logins for their account. Do not assume that users know what a cookie is; instead, ask a question like: Automatically login from this computer?

- Always ask for passwords from users logging on to obtain sensitive data. Unless a user requests otherwise, you can store this preference and the password in the user’s session data. Configure the session cookie to expire when the user quits the browser.
Using WebLogic Services from an HTTP Servlet

When you write an HTTP servlet, you have access to many rich features of WebLogic Server, such as JNDI, EJB, JDBC, and JMS.

The following documents provide additional information about these features:


Accessing Databases

WebLogic Server supports the use of Java Database Connectivity (JDBC) from server-side Java classes, including servlets. JDBC allows you to execute SQL queries from a Java class and to process the results of those queries. For more information on JDBC and WebLogic Server, see Using WebLogic JDBC at http://e-docs.bea.com/wls/docs61/jdbc/index.html.

You can use JDBC in servlets as described in the following sections:

- “Connecting to a Database Using a JDBC Connection Pool” on page 3-27.
- “Connecting to a Database Using a DataSource Object” on page 3-28.
- “Connecting Directly to a Database Using a JDBC Driver” on page 3-29.
Connecting to a Database Using a JDBC Connection Pool

A connection pool is a named group of identical JDBC connections to a database that are created when the connection pool is registered, usually when starting WebLogic Server. Your servlets “borrow” a connection from the pool, use it, and then return it to the pool by closing it. This process is far more efficient than creating a new connection for every client each time the client needs to access the database. Another advantage is that you do not need to include details about the database in your servlet code.

When connecting to a JDBC connection pool, use one of the following multitier JDBC drivers:

- **Pool driver**, used for most server-side operations:
  - Driver URL: `jdbc:weblogic:pool`
  - Driver package name: `weblogic.jdbc.pool.Driver`

- **JTS pool driver**, used when database operations require transactional support.
  - Driver URL: `jdbc:weblogic:jts`
  - Driver package name: `weblogic.jdbc.jts.Driver`

### Using a Connection Pool in a Servlet

The following example demonstrates how to use a database connection pool from a servlet.

1. Load the pool driver and cast it to `java.sql.Driver`. The full pathname of the driver is `weblogic.jdbc.pool.Driver`. For example:

   ```java
   Driver myDriver = (Driver)
   Class.forName("weblogic.jdbc.pool.Driver").newInstance();
   ```

2. Create a connection using the URL for the driver, plus (optionally) the name of the registered connection pool. The URL of the pool driver is `jdbc:weblogic:pool`.

   You can identify the pool in either of two ways:

   - Specify the name of the connection pool in a `java.util.Properties` object using the key `connectionPoolID`. For example:
Properties props = new Properties();
props.put("connectionPoolID", "myConnectionPool");
Connection conn =
    myDriver.connect("jdbc:weblogic:pool", props);

- Add the name of the pool to the end of the URL. In this case you do not need a Properties object unless you are setting a username and password for using a connection from the pool. For example:

  Connection conn =
      myDriver.connect("jdbc:weblogic:pool:myConnectionPool", null);

Note that the Driver.connect() method is used in these examples instead of the DriverManger.getConnection() method. Although you may use DriverManger.getConnection() to obtain a database connection, we recommend that you use Driver.connect() because this method is not synchronized and provides better performance.

Note that the Connection returned by connect() is an instance of weblogic.jdbc.pool.Connection.

3. Call the close() method on the Connection object when you finish with your JDBC calls, so that the connection is properly returned to the pool. A good coding practice is to create the connection in a try block and then close the connection in a finally block, to make sure the connection is closed in all cases.

   conn.close();

### Connecting to a Database Using a DataSource Object

A DataSource is a server-side object that references a connection pool. The connection pool registration defines the JDBC driver, database, login, and other parameters associated with a database connection. You create DataSource objects and connection pools through the Administration Console. Using a DataSource object is recommended when creating J2EE-compliant applications.
Using a DataSource in a Servlet

1. Register a connection pool using the Administration Console. For more information, see “Create a Connection Pool” at http://e-docs.bea.com/wls/docs61/ConsoleHelp/jdbcconnectionpool.html.

2. Register a DataSource object that points to the connection pool. For more information, see “JDBC DataSources” at http://e-docs.bea.com/wls/docs61/ConsoleHelp/jdbcdatasource.htm.

3. Look up the DataSource object in the JNDI tree. For example:

   ```java
   Context ctx = null;
   // Get a context for the JNDI look up
   ctx = new InitialContext(ht);
   // Look up the DataSource object
   javax.sql.DataSource ds
   = (javax.sql.DataSource) ctx.lookup("myDataSource");
   ```

4. Use the DataSource to create a JDBC connection. For example:

   ```java
   java.sql.Connection conn = ds.getConnection();
   ```

5. Use the connection to execute SQL statements. For example:

   ```java
   Statement stmt = conn.createStatement();
   stmt.executeUpdate("select * from emp");
   ```

Connecting Directly to a Database Using a JDBC Driver

Connecting directly to a database is the least efficient way of making a database connection because a new database connection must be established for each request. You can use any JDBC driver to connect to your database. BEA provides JDBC drivers for Oracle, Microsoft SQL Server, and Informix. For more information, see Programming WebLogic JDBC at http://e-docs.bea.com/wls/docs61/jdbc/index.html.
When you design a servlet, you should consider how the servlet is invoked by WebLogic Server under high load. It is inevitable that more than one client will hit your servlet simultaneously. Therefore, write your servlet code to guard against sharing violations on shared resources or instance variables. The following tips can help you to design around this issue.

**SingleThreadModel**

An instance of a class that implements the SingleThreadModel is guaranteed not to be invoked by multiple threads simultaneously. Multiple instances of a SingleThreadModel servlet are used to service simultaneous requests, each running in a single thread.

To use the SingleThreadModel efficiently, WebLogic Server creates a pool of servlet instances for each servlet that implements SingleThreadModel. WebLogic Server creates the pool of servlet instances when the first request is made to the servlet and increments the number of servlet instances in the pool as needed.

The attribute “SingleThreaded Servlet Pool Size” (see [http://e-docs.bea.com/wls/docs61/ConsoleHelp/webappcomponent.html#SingleThreadedServletPoolSize](http://e-docs.bea.com/wls/docs61/ConsoleHelp/webappcomponent.html#SingleThreadedServletPoolSize)), set using the WebLogic Server Administration Console, specifies the initial number of servlet instances that are created when the servlet is first requested. Set this attribute to the average number of concurrent requests that you expect your SingleThreadModel servlets to handle.

When designing your servlet, consider how you use shared resources outside of the servlet class such as file and database access. Because multiple instances of identical servlets exist, and may use exactly the same resources, there are still synchronization and sharing issues that must be resolved, even if you do implement the SingleThreadModel.
Shared Resources

It is recommended that shared-resource issues be handled on an individual servlet basis. Consider the following guidelines:

- Wherever possible, avoid synchronization, because it causes subsequent servlet requests to bottleneck until the current thread completes.
- Define variables that are specific to each servlet request within the scope of the service methods. Local scope variables are stored on the stack and, therefore, are not shared by multiple threads running within the same method, which avoids the need to be synchronized.
- Access to external resources should be synchronized on a Class level, or encapsulated in a transaction.

Dispatching Requests to Another Resource

This section provides an overview of commonly used methods for dispatching requests from a servlet to another resource.

A servlet can pass on a request to another resource, such as a servlet, JSP, or HTML page. This process is referred to as request dispatching. When you dispatch requests, you use either the `include()` or `forward()` method of the `RequestDispatcher` interface. There are limitations regarding when output can be written to the response object using the `forward()` or `include()` methods. These limitations are also discussed in this section.

For a complete discussion of request dispatching, see section 8.1 of the Servlet 2.2 specification (see http://java.sun.com/products/servlet/download.html#specs) from Sun Microsystems.

By using the `RequestDispatcher`, you can avoid sending an HTTP-redirect response back to the client. The `RequestDispatcher` passes the HTTP request to the requested resource.
To dispatch a request to a particular resource:

1. Get a reference to a ServletContext:
   ```java
   ServletContext sc = getServletConfig().getServletContext();
   ```

2. Look up the RequestDispatcher object using one of the following methods:
   - RequestDispatcher rd = sc.getRequestDispatcher(String path);
     - `path` should be relative to the root of the Web Application.
   - RequestDispatcher rd = sc.getNamedDispatcher(String name);
     - Replace `name` with the name assigned to the servlet in a Web Application deployment descriptor with the `<servlet-name>` element. For details, see “Servlet element” at http://e-docs.bea.com/wls/docs61/webapp/web_xml.html#web_xml_servlet.
   - RequestDispatcher rd = ServletRequest.getRequestDispatcher(String path);
     - This method returns a RequestDispatcher object and is similar to the ServletContext.getRequestDispatcher(String path) method except that it allows the `path` specified to be relative to the current servlet. If the path begins with a / character it is interpreted to be relative to the Web Application.

   You can obtain a RequestDispatcher for any HTTP resource within a Web Application, including HTTP Servlets, JSP pages, or plain HTML pages by requesting the appropriate URL for the resource in the `getRequestDispatcher()` method. Use the returned RequestDispatcher object to forward the request to another servlet.

3. Forward or include the request using the appropriate method:
   - rd.forward(request, response);
   - rd.include(request, response);

   These methods are discussed in the next two sections.
Dispatching Requests to Another Resource

Forwarding a Request

Once you have the correct RequestDispatcher, your servlet forwards a request using the RequestDispatcher.forward() method, passing HttpServletRequest and HttpServletResponse as arguments. If you call this method when output has already been sent to the client an IllegalStateException is thrown. If the response buffer contains pending output that has not been committed, the buffer is reset.

The servlet must not attempt to write any previous output to the response. If the servlet retrieves the ServletOutputStream or the PrintWriter for the response before forwarding the request, an IllegalStateException is thrown.

All other output from the original servlet is ignored after the request has been forwarded.

If you are using any type of authentication, a forwarded request, by default, does not require the user to be re-authenticated. You can change this behavior to require authentication of a forwarded request by adding the <check-auth-on-forward/> element to the <container-descriptor> element of the WebLogic-specific deployment descriptor, weblogic.xml. For example:

```
<container-descriptor>
  <check-auth-on-forward/>
</container-descriptor>
```

Note that the default behavior has changed with the release of the Servlet 2.3 specification, which states that authentication is not required for forwarded requests.

For information on editing the WebLogic-specific deployment descriptor, see Writing the WebLogic-Specific Deployment Descriptor at http://e-docs.bea.com/wls/docs61/webapp/webappdeployment.html#weblogic-xml.

Including a Request

Your servlet can include the output from another resource by using the RequestDispatcher.include() method, and passing HttpServletRequest and HttpServletResponse as arguments. When you include output from another resource, the included resource has access to the request object.
The included resource can write data back to the `ServletOutputStream` or `Writer` objects of the response object and then can either add data to the response buffer or call the `flush()` method on the response object. Any attempt to set the response status code or to set any HTTP header information from the included servlet response is ignored.

In effect, you can use the `include()` method to mimic a “server-side-include” of another HTTP resource from your servlet code.
CHAPTER

4 Administration and Configuration

The following sections provide an overview of administration and configuration tasks for WebLogic HTTP servlets. For a complete discussion of servlet administration and configuration, see Configuring Servlets at http://e-docs.bea.com/wls/docs61/webapp/components.html#configuring-servlets.

This section discusses the following topics:

- Overview of WebLogic HTTP Servlet Administration
- Referencing a Servlet in a Web Application
- Directory Structure for Web Applications
- Servlet Security
- Servlet Development Tips
- Clustering Servlets
Overview of WebLogic HTTP Servlet Administration

Consistent with the Java 2 Enterprise Edition standard, HTTP servlets are deployed as part of a Web Application. A Web Application is a grouping of application components, such as servlet classes, JavaServer Pages (JSP), static HTML pages, images, and utility classes.

In a Web Application the components are deployed using a standard directory structure. This directory structure can be archived into a file called a .war file and then deployed on WebLogic Server. Information about the resources and operating parameters of a Web Application are defined using two deployment descriptors, which are packaged with the Web Application.

Using Deployment Descriptors to Configure and Deploy Servlets

The first deployment descriptor, web.xml, is defined in the Servlet 2.2 specification from Sun Microsystems and provides a standardized format that describes the Web Application. The second deployment descriptor, weblogic.xml, is a WebLogic-specific deployment descriptor that maps resources defined in the web.xml file to resources available in WebLogic Server, defines JSP behavior, and defines HTTP session parameters.

web.xml (Web Application Deployment Descriptor)

In the Web Application deployment descriptor you define the following attributes for HTTP servlets:

- Servlet name
- Java class of the servlet
- Servlet initialization parameters
Overview of WebLogic HTTP Servlet Administration

- Whether or not the `init()` method of the servlet is executed when WebLogic Server starts
- URL pattern which, if matched, will call this servlet
- Security
- MIME type
- Error pages
- References to EJBs
- References to other resources

For a complete discussion of creating the `web.xml` file, see Writing Web Application Deployment Descriptors at http://e-docs.bea.com/wls/docs61/webapp/webappdeployment.html#web.xml.

**weblogic.xml (Weblogic-Specific Deployment Descriptor)**

In the WebLogic-specific deployment descriptor you define the following attributes for HTTP servlets:

- HTTP session configuration
- Cookie configuration
- EJB resource mapping
- JSP Configuration

For a complete discussion of creating the `weblogic.xml` file, see “Writing Web Application Deployment Descriptors at http://e-docs.bea.com/wls/docs61/webapp/webappdeployment.html#weblogic.xml.”
WebLogic Server Administration Console

Use the WebLogic Server Administration Console to set the following parameters:

- HTTP parameters
- Log files
- URL rewriting
- Keep alive
- Default MIME types
- Clustering parameters
- URL mapping for virtual hosting

For more information see the following resources:

Directory Structure for Web Applications

Use the following directory structure for all Web Applications:

```
WebApplicationRoot/(Publicly available files, such as
 | .jsp, .html, .jpg, .gif)
+WEB-INF/--+
 | + classes/(directory containing
 | Java classes including
 | servlets used by the
 | Web Application)
 | + lib/(directory containing
 | jar files used by the
 | Web Application)
 | + web.xml
 | + weblogic.xml
```

Referencing a Servlet in a Web Application

The URL used to reference a servlet in a Web Application is constructed as follows:

```
http://myHostName:port/myContextPath/myRequest/?myRequestParameters
```

The components of this URL are defined as follows:

- **myHostName**
  - The DNS name mapped to the Web Server defined in the WebLogic Server Administration Console.
  - This portion of the URL can be replaced with `host:port`, where `host` is the name of the machine running WebLogic Server and `port` is the port at which WebLogic Server is listening for requests.
Port

The port at which WebLogic Server is listening for requests.

/myContextPath

The name of the Web Application defined in the WebLogic Server Administration Console.

/myRequest

The name of the servlet as defined in the web.xml file.

?/myRequestParameters

Optional HTTP request parameters encoded in the URL, which can be read by an HTTP servlet.

Servlet Security

Security for servlets is defined in the context of the Web Application containing the servlet. Security can be handled by WebLogic Server, or it can be incorporated programmatically into your servlet classes.


Authentication

You can incorporate user authentication into your servlets using any of the following three techniques:

- BASIC—Uses the browser to collect a username and password.
- FORM—Uses HTML forms to collect a username and password.
- Client Certificate—Uses digital certificates to authenticate the user. For more information, see “Digital Certificates” at http://e-docs.bea.com/wls/docs61/security/concepts.html#concept_s008.
The BASIC and FORM techniques call into a security realm that contains user and password information. You can use a default realm provided with WebLogic Server, or a variety of existing realms, including realms for Windows NT, UNIX, RDBMS, and user-defined realms. For more information about security realms, see “Security Fundamentals” at http://e-docs.bea.com/wls/docs61/security/concepts.html.

Authorization (Security Constraints)

You can restrict access to servlets and other resources in a Web Application by using security constraints. Security constraints are defined in the Web Application deployment descriptor (web.xml). There are three basic types of security constraints:

- Constraining resources by roles and/or resource
- Secure Sockets Layer (SSL) encryption
- Programmatic authorization

Roles can be mapped to a principal in a realm. Specific resources can be constrained by matching a URL pattern to a resource in a Web Application. You can also use Secure Sockets Layer (SSL) as a security constraint.

You can perform authorization programmatically, using one of the following methods of the HttpServletRequest interface:

- getRemoteUser()
- isUserInRole()
- getUserPrincipal()

For more information see the javax.servlet API at http://java.sun.com/products/servlet/2.2/javadoc/index.html.

Servlet Development Tips

Consider the following tips when writing HTTP servlets:
Compile your servlet classes into the `WEB-INF/classes` directory of your Web Application.

Make sure your servlet is registered in the Web Applications deployment descriptor (`web.xml`).

When responding to a request for a servlet, WebLogic Server checks the time stamp of the servlet class file and compares it to the servlet instance in memory. If a newer version of the servlet class is found, WebLogic Server re-loads the servlet class. When a servlet is re-loaded, the `init()` method of the servlet is called.

You can set the interval (in seconds) at which WebLogic Server checks the time stamp with the `Servlet Reload` attribute. This attribute is set on the `Files` tab of your Web Application, in the Administration Console. If you set this attribute to zero, WebLogic Server checks the time stamp on every request, which can be useful while developing and testing servlets but is needlessly time consuming in a production environment. If this attribute is set to -1, WebLogic Server does not check for modified servlets.

## Clustering Servlets

Clustering servlets provides failover and load balancing benefits. To deploy a servlet in a WebLogic Server cluster, deploy the Web Application containing the servlet on all servers in the cluster. For instructions, see “Deploy Web Applications and EJBs” in Using WebLogic Server Clusters.

For information on failover for clustered servlets, see “Failover Support for Clustered Services” and “Understanding HTTP Session State Replication” in Using WebLogic Server Clusters.

**Note:** Automatic failover for servlets requires that the servlet session state be replicated in memory. For instructions, see “Configuring In-Memory HTTP Replication in a Cluster” in Using WebLogic Server Clusters.

For information on the load balancing support that a WebLogic Server cluster provides for servlets, and for related planning and configuration considerations for architects and administrators, see “Load Balancing for HTTP Session States” in Using WebLogic Server Clusters.
Index

A
addCookie() 3-23
administration
console 4-4
administration console 4-4
API 1-5
authentication 4-6

C
classpath 2-2
clustering 3-20, 4-8
compiling 2-2
connection pools 3-26
DataSource 3-28
driver 3-27
JDBC 3-27
using 3-27
contentType 2-2
cookies 3-22
and EJB 3-23
and logging in 3-25
and passwords 3-25
domain 3-24
HTTP and HTTPS 3-24
retrieving 3-23
using in servlets 3-22
customer support contact information vii

D
databases 3-26
DataSource 3-26, 3-28
deployment 2-3
deployment descriptor 4-2
Developing 1-3
development
classpath 2-2
compiling 4-7
tips 4-7
development environment 2-2
dispatching 3-31
documentation, where to find it vi

E
EJB 3-26
encodeURL() 3-19
environment, development
environment 2-2

F
forward() 3-31
forwarding 3-31, 3-33

G
getAttribute() 3-16
getAttributeNames() 3-16
getCookies() 3-23
getParameterValues() 3-11
getSession() 3-13, 3-15

**H**
- HelloWorldServlet 2-5
- HTTP
  - response 3-4
  - HttpServletRequest 2-1
    - methods 3-9
  - HttpServletResponse 2-1, 3-4
  - HttpSession object 3-13

**I**
- IDLength 3-20
- IllegalStateException 3-17
- import 2-1
- include() 3-31
- including 3-31
- including a request 3-33
- init parameters 3-2
- init() method 3-2, 3-3
- initialization
  - init() method 3-2
  - parameters 3-2
  - init-param 3-3
  - in-memory replication 3-20
- input
  - query parameters 3-10

**J**
- J2EE 1-3
- javax.servlet 1-5
- JDBC 3-26, 3-29
- JDBC session persistence 3-20
- JMS 3-26
- JNDI 3-26
- JTS pool driver 3-27

**K**
- keep alive 3-5

**L**
- logging out 3-17

**N**
- name/value pairs 3-16

**P**
- packages 2-1
- Pool driver 3-27
- printing product documentation vi
- PrintWriter object 2-2

**Q**
- query parameters 3-6, 3-7, 3-10

**R**
- removeAttribute() 3-16
- RequestDispatcher() 3-32
- requests
  - dispatching 3-31
  - forwarding 3-31, 3-33
  - including 3-31, 3-33
- response 3-4
  - buffer 3-5
  - optimizing 3-5
  - retrieving input 3-6

**S**
- security 4-6
  - applying programatically 4-7
  - authentication 4-6
  - authorization 4-7
  - constraints 4-7
realms 4-7
security constraints 4-7
service method 2-1
Servlet 2.2 Specification 1-5
servlets
  and clustering 4-8
session persistence
    JDBC 3-20
sessions
  and clusters 3-20
  and persistence 3-20
  cookies 3-15, 3-18
  detecting start of 3-15
  encodeURL() method 3-19
  ending 3-17
  history of tracking 3-12
  lifetime 3-14
  logging out 3-17
  name/value attributes 3-16
  tracking 3-12, 3-15
  tracking with HttpSession object 3-13
  tracking, configuration 3-18
ULATOR rewriting 3-18
setAttribute() 3-16
SingleThreadModel 3-30
SingleThreadModelPoolSize 3-30
support
  technical vii

T
threading 3-30
  SingleThreadModel 3-30

U
URL rewriting 3-18
  and WAP 3-20
  and Wireless Access Protocol 3-20
URLs 4-5

W
WAP 3-20
Web Applications
  and security 4-6
  deployment descriptor 4-2
  directory structure 4-5
  URLs 4-5
web.xml 4-2
weblogic.xml 4-2
Wireless Access Protocol 3-20