

Chapter 1 - Servlets	2
Chapter 2 - Servlet Lifecycle	8
Chapter 3 - Servlet-client Interaction	17
Chapter 4 - Writing Your First Servlet	21
Chapter 5 - Filters and Servlet Chaining	25
Chapter 6 - Processing Forms with Servlets	28
Chapter 7 - Writing Servlets as Beans	42
Chapter 8 - Session Tracking	50
Chapter 9 - Loading and Invoking Servlets	63
Chapter 10 - Java Server Pages	66
Chapter 11 - Writing Your First Java Server Page	69
Chapter 12 - Frequently Asked Questions.....	76

SERVLETS

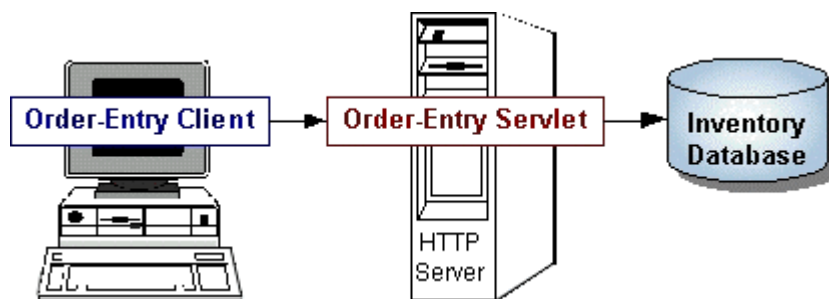
Servlets provide a Java based solution used to address the problems currently associated with doing server-side programming, including inextensible scripting solutions, platform-specific APIs, and incomplete interfaces.

Servlets are objects that conform to a specific interface that can be plugged into a Java-based server. Servlets are to the server-side what applets are to the client-side -- object bytecodes that can be dynamically loaded off the net. They differ from applets in that they are faceless objects (without graphics or a GUI component). They serve as platform-independent, dynamically loadable, pluggable helper bytecode objects on the server side that can be used to dynamically extend server-side functionality.

1.1 OVERVIEW OF SERVLETS

1.1.1 What is a Servlet?

Servlets are modules that extend request/response-oriented servers, such as Java-enabled web servers. For example, a servlet might be responsible for taking data in an HTML order-entry form and applying the business logic used to update a company's order database.



Servlets are to servers what applets are to browsers. Unlike applets, however, Servlets have no graphical user interface.

Servlets can be embedded in many different servers because the servlet API, which you use to write Servlets, assumes nothing about the server's environment or protocol. Servlets have become most widely used within HTTP servers; many web servers support the Servlet API.

1.2 USE SERVLETS INSTEAD OF CGI SCRIPTS!

Servlets are an effective replacement for CGI scripts. They provide a way to generate dynamic documents that is both easier to write and faster to run. Servlets also address the problem of doing server-side programming with platform-specific APIs: they are developed with the Java Servlet API, a standard Java extension.

So use Servlets to handle HTTP client requests. For example, have Servlets process data posted over HTTPS using an HTML form, including purchase order or credit card data. A servlet like this could be part of an order-entry and processing system, working with product and inventory databases, and perhaps an on-line payment system.

1.3 OTHER USES FOR SERVLETS

Here are a few more of the many applications for Servlets:

- Allowing collaboration between people. A servlet can handle multiple requests concurrently, and can synchronize requests. This allows Servlets to support systems such as on-line conferencing.**
- Forwarding requests. Servlets can forward requests to other servers and Servlets. Thus Servlets can be used to balance load among several servers that mirror the same content, and to partition a single logical service over several servers, according to task type or organizational boundaries.**

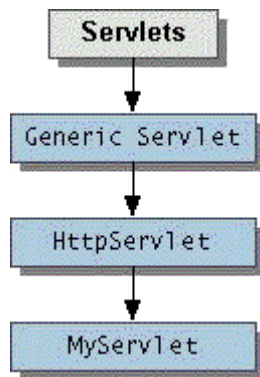
1.4 ARCHITECTURE OF THE SERVLET PACKAGE

The javax.servlet package provides interfaces and classes for writing Servlets. The architecture of the package is described below.

1.4.1 The Servlet Interface

The central abstraction in the Servlet API is the Servlet interface. All servlets implement this interface, either directly or, more commonly, by extending a class that implements it such as HttpServlet.

The Servlet interface declares, but does not implement, methods that manage the servlet and its communications with clients. Servlet writers provide some or all of



these methods when developing a servlet.

1.4.2 Client Interaction

When a servlet accepts a call from a client, it receives two objects:

- A **ServletRequest**, which encapsulates the communication from the client to the server.
- A **ServletResponse**, which encapsulates the communication from the servlet back to the client.

ServletRequest and **ServletResponse** are interfaces defined by the `javax.servlet` package.

1.4.3 The ServletRequest Interface

The **ServletRequest** interface allows the servlet access to:

- Information such as the names of the parameters passed in by the client, the protocol (scheme) being used by the client, and the names of the remote host that made the request and the server that received it.
- The input stream, `ServletInputStream`. Servlets use the input stream to get data from clients that use application protocols such as the HTTP POST and PUT methods.

Interfaces that extend `ServletRequest` interface allow the servlet to retrieve more protocol-specific data. For example, the `HttpServletRequest` interface contains methods for accessing HTTP-specific header information.

1.4.4 The `ServletResponse` Interface

The `ServletResponse` interface gives the servlet methods for replying to the client. It:

- Allows the servlet to set the content length and MIME type of the reply.
- Provides an output stream, `ServletOutputStream`, and a `Writer` through which the servlet can send the reply data.

Interfaces that extend the `ServletResponse` interface give the servlet more protocol-specific capabilities. For example, the `HttpServletResponse` interface contains methods that allow the servlet to manipulate HTTP-specific header information.

1.5 ADDITIONAL CAPABILITIES OF HTTP SERVLETS

The classes and interfaces described above make up a basic Servlet. HTTP servlets have some additional objects that provide session-tracking capabilities. The servlet writer can use these APIs to maintain state between the servlet and the client that persists across multiple connections during some time period. HTTP servlets also have objects that provide cookies. The servlet writer uses the cookie API to save data with the client and to retrieve this data.

1.6 A SIMPLE SERVLET

The following class completely defines servlet:

```
public class SimpleServlet extends HttpServlet
{
    /**
     * Handle the HTTP GET method by building a simple web page.
     */
    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        PrintWriter out;
        String title = "Simple Servlet Output";

        // set content type and other response header fields first
        response.setContentType("text/html");
        // then write the data of the response
        out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println(title);
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>This is output from SimpleServlet.");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

That's it!

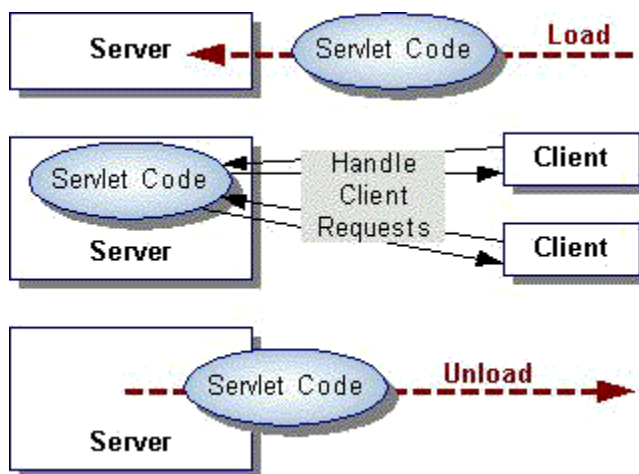
- SimpleServlet extends the HttpServlet class, which implements the Servlet interface.

- **SimpleServlet overrides the doGet method in the HttpServlet class. The doGet method is called when a client makes a GET request (the default HTTP request method), and results in the simple HTML page being returned to the client.**
- **Within the doGet method,**
 - **The user's request is represented by an HttpServletRequest object.**
 - **The response to the user is represented by an HttpServletResponse object.**
 - **Because text data is returned to the client, the reply is sent using the Writer object obtained from the HttpServletResponse object.**

CHAPTER 2 - SERVLET LIFECYCLE

Each servlet has the same life cycle:

- A server loads and initializes the servlet
- The servlet handles zero or more client requests
- The server removes the servlet



2.1 INITIALIZING A SERVLET

When a server loads a servlet, the server runs the Servlets init method. Initialization completes before client requests are handled and before the servlet is destroyed.

Even though most Servlets are run in multi-threaded servers, Servlets have no concurrency issues during servlet initialization. The server calls the init method once, when the server loads the servlet, and will not call the init method again unless the server is reloading the servlet. The server can not reload a servlet until after the server has destroyed the servlet by calling the destroy method.

2.1.1 The init Method

The init method provided by the HttpServlet class initializes the servlet and logs the initialization. To do initialization specific to your servlet, override the init() method following these rules:

- If an initialization error occurs that renders the servlet incapable of handling client requests, throw an `UnavailableException`.

An example of this type of error is the inability to establish a required network connection.

- Do not call the `System.exit` method

Here is an example init method:

```
public class BookDBServlet ... {  
    private BookstoreDB books;  
    public void init() throws ServletException {  
        // Load the database to prepare for requests  
        books = new BookstoreDB();  
    }  
    ...  
}
```

The init method is quite simple: it sets a private field.

If the BookDBServlet used an actual database, instead of simulating one with an object, the init method would be more complex. Here is pseudo-code for what the init method might look like:

```
public class BookDBServlet ... {  
  
    public void init() throws ServletException {
```

```
// Open a database connection to prepare for requests
try {
    databaseUrl = getInitParameter("databaseUrl");
    ... // get user and password parameters the same way
    connection = DriverManager.getConnection(databaseUrl,
                                             user, password);
} catch(Exception e) {
    throw new UnavailableException (this,
    "Could not open a connection to the database");
}
}
...
}
```

2.1.2 Initialization Parameters

The second version of the `init` method calls the `getInitParameter` method. This method takes the parameter name as an argument and returns a String representation of the parameter's value.

The specification of initialization parameters is server-specific. In the Java Web Server, the parameters are specified with a servlet is added then configured in the Administration Tool. For an explanation of the Administration screen where this setup is performed, see the Administration Tool: Adding Servlets online help document.

If, for some reason, you need to get the parameter names, use the `getParameterNames` method.

2.2 DESTROYING A SERVLET

Servlets run until the server destroys them, for example at the request of a system administrator. When a server destroys a servlet, the server runs the servlet's destroy method. The method is run once; the server will not run that servlet again until after the server reloads and reinitializes the servlet.

When the destroy method runs, another thread might be running a service request. The Handling Service Threads at Servlet Termination section shows you how to provide a clean shutdown when there could be long-running threads still running service requests.

2.2.1 Using the Destroy Method

The destroy method provided by the HttpServlet class destroys the servlet and logs the destruction. To destroy any resources specific to your servlet, override the destroy method. The destroy method should undo any initialization work and synchronize persistent state with the current in-memory state.

The following example shows the destroy method that accompanies the init method shown previously:

```
public class BookDBServlet extends GenericServlet {  
    private BookstoreDB books;  
    ... // the init method  
    public void destroy() {  
        // Allow the database to be garbage collected  
        books = null;  
    }  
}
```

A server calls the destroy method after all service calls have been completed, or a server-specific number of seconds have passed, whichever comes first. If your servlet handles any long-running operations, service methods might still be running when the server calls the destroy method. You are responsible for making sure those threads complete. The next section shows you how.

The destroy method shown above expects all client interactions to be completed when the destroy method is called, because the servlet has no long-running operations.

2.3 HANDLING SERVICE THREADS AT SERVLET TERMINATION

All of a servlet's service methods should be complete when a servlet is removed. The server tries to ensure this by calling the destroy method only after all service requests have returned, or after a server-specific grace period, whichever comes first. If your servlet has operations that take a long time to run (that is, operations that may run longer than the server's grace period), the operations could still be running when destroy is called. You must make sure that any threads still handling client requests complete; the remainder of this section describes a technique for doing this.

If your servlet has potentially long-running service requests, use the following techniques to:

- Keep track of how many threads are currently running the service method.
- Provide a clean shutdown by having the destroy method notify long-running threads of the shutdown and wait for them to complete
- Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up and return.

2.3.1 Tracking Service Requests

To track service requests, include a field in your servlet class that counts the number of service methods that are running. The field should have access methods to increment, decrement, and return its value. For example:

```
public ShutdownExample extends HttpServlet {  
    private int serviceCounter = 0;  
    ...  
    //Access methods for serviceCounter  
    protected synchronized void enteringServiceMethod() {  
        serviceCounter++;  
    }  
    protected synchronized void leavingServiceMethod() {  
        serviceCounter--;  
    }  
    protected synchronized int numServices() {  
        return serviceCounter;  
    }  
}
```

The service method should increment the service counter each time the method is entered and decrement the counter each time the method returns. This is one of the few times that your HttpServlet subclass should override the service method. The new method should call `super.service` to preserve all the original HttpServlet.service method's functionality.

```
protected void service(HttpServletRequest req, HttpServletResponse resp)  
    throws ServletException, IOException  
{  
    enteringServiceMethod();
```

```
try {  
    super.service(req, resp);  
} finally {  
    leavingServiceMethod();  
}  
}
```

2.3.2 Providing a Clean Shutdown

To provide a clean shutdown, your destroy method should not destroy any shared resources until all the service requests have completed. One part of doing this is to check the service counter. Another part is to notify the long-running methods that it is time to shut down. For this, another field is required along with the usual access methods. For example:

```
public ShutdownExample extends HttpServlet {  
    private boolean shuttingDown;  
    ...  
    //Access methods for shuttingDown  
    protected setShuttingDown(boolean flag) {  
        shuttingDown = flag;  
    }  
    protected boolean isShuttingDown() {  
        return shuttingDown;  
    }  
}
```

An example of the destroy method using these fields to provide a clean shutdown is shown below:

```
public void destroy() {
```

```
/* Check to see whether there are still service methods running,  
 * and if there are, tell them to stop. */  
if (numServices() > 0) {  
    setShuttingDown(true);  
}  
  
/* Wait for the service methods to stop. */  
while(numServices() > 0) {  
    try {  
        Thread.sleep(interval);  
    } catch (InterruptedException e) {  
    }  
}  
}
```

2.3.3 Creating Polite Long-running Methods

The final step in providing a clean shutdown is to make any long-running methods behave politely. Methods that might run for a long time should check the value of the field that notifies them of shut downs, and interrupt their work if necessary. For example:

```
public void doPost(...) {  
    ...  
    for(i = 0; ((i < lotsOfStuffToDo) && !isShuttingDown()); i++) {  
        try {  
            partOfLongRunningOperation(i);  
        } catch (InterruptedException e) {  
        }  
    }  
}
```

}

}

CHAPTER 3 - SERVLET-CLIENT INTERACTION

3.1 HANDLING HTTP CLIENTS

An HTTP Servlet handles client requests through its service method. The service method supports standard HTTP client requests by dispatching each request to a method designed to handle that request. For example, the service method calls the `doGet` method shown earlier in the simple example servlet.

3.2 REQUESTS AND RESPONSES

Methods in the `HttpServlet` class that handle client requests take two arguments:

- An `HttpServletRequest` object, which encapsulates the data from the client
- An `HttpServletResponse` object, which encapsulates the response to the client

3.2.1 `HttpServletRequest` Objects

An `HttpServletRequest` object provides access to HTTP header data, such as any cookies found in the request and the HTTP method with which the request was made. The `HttpServletRequest` object also allows you to obtain the arguments that the client sent as part of the request.

To access client data:

- The `getParameter` method returns the value of a named parameter. If your parameter could have more than one value, use `getParameterValues` instead. The `getParameterValues` method returns an array of values for the named parameter. (The method `getParameterNames` provides the names of the parameters.)
- For HTTP GET requests, the `getQueryString` method returns a `String` of raw data from the client. You must parse this data yourself to obtain the parameters and values.
- For HTTP POST, PUT, and DELETE requests,

- If you expect text data, the `getReader` method returns a `BufferedReader` for you to use to read the raw data.
 - If you expect binary data, the `getInputStream` method returns a `ServletInputStream` for you to use to read the raw data
-

Note: Use either a `getParameter[Values]` method or one of the methods that allow you to parse the data yourself. They can not be used together in a single request.

3.3 HTTPSERVLETRESPONSE OBJECTS

An `HttpServletResponse` object provides two ways of returning data to the user:

- The `getWriter` method returns a `Writer`
- The `getOutputStream` method returns a `ServletOutputStream`

Use the `getWriter` method to return text data to the user, and the `getOutputStream` method for binary data.

Closing the `Writer` or `ServletOutputStream` after you send the response allows the server to know when the response is complete.

3.3.1 HTTP Header Data

You must set HTTP header data before you access the `Writer` or `OutputStream`. The `HttpServletResponse` class provides methods to access the header data. For example, the `setContentType` method sets the content type. (This header is often the only one manually set.)

3.4 HANDLING GET AND POST REQUESTS

The methods to which the service method delegates HTTP requests include,

- `doGet`, for handling GET, conditional GET, and HEAD requests
- `doPost`, for handling POST requests
- `doPut`, for handling PUT requests

- `doDelete`, for handling DELETE requests

By default, these methods return a BAD_REQUEST (400) error. Your Servlet should override the method or methods designed to handle the HTTP interactions that it supports. This section shows you how to implement methods that handle the most common HTTP requests: GET and POST.

The `HttpServlet`'s service method also calls the `doOptions` method when the servlet receives an OPTIONS request, and `doTrace` when the servlet receives a TRACE request. The default implementation of `doOptions` automatically determines what HTTP options are supported and returns that information. The default implementation of `doTrace` causes a response with a message containing all of the headers sent in the trace request. These methods are not typically overridden.

3.5 THREADING ISSUES

HTTP servlets are typically capable of serving multiple clients concurrently. If the methods in your servlet that do work for clients access a shared resource, then you can handle the concurrency by creating a servlet that handles only one client request at a time.

To have your servlet handle only one client at a time, have your servlet implement the `SingleThreadModel` interface in addition to extending the `HttpServlet` class.

Implementing the `SingleThreadModel` interface does not involve writing any extra methods. You merely declare that the servlet implements the interface, and the server makes sure that your servlet runs only one service method at a time.

For example, the `ReceiptServlet` accepts a user's name and credit card number, and thanks the user for their order. If this servlet actually updated a database, for example one that kept track of inventory, then the database connection might be a shared resource. The servlet could either synchronize access to that resource, or it could implement the `SingleThreadModel` interface.

```
public class ReceiptServlet extends HttpServlet
    implements SingleThreadModel {

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

3.6 SERVLET DESCRIPTIONS

In addition to handling HTTP client requests, some applications, such as the Java Web Server's Administration Tool, get descriptive information from the servlet and display it. The servlet description is a string that can describe the purpose of the servlet, its author, its version number, or whatever the servlet author deems important.

The method that returns this information is `getServletInfo`, which returns null by default. You are not required to override this method, but applications are unable to supply a description of your servlet unless you do.

The following example shows the description of the `BookStoreServlet`:

```
public class BookStoreServlet extends HttpServlet {
    ...
    public String getServletInfo() {
        return "The BookStore servlet returns the " +
            "main web page for Duke's Bookstore.";
    }
}
```

CHAPTER 4 - WRITING YOUR FIRST SERVLET

As discussed in the servlet Overview, Servlets extend a server's functionality much the same way applets extend a browser's capability. Servlets have the following advantages over other server-side development strategies: they are platform independent, they are written to a standard API, they have network capabilities (can be used with RMI or JDBC), they are faster than CGI, and they are reusable.

Servlets are also easy to develop. This document discusses the following minimum steps needed to create any servlet:

1. Write the servlet

- Import the necessary Java packages
- Inherit from `GenericServlet` or the HTTP convenience class `HttpServlet`
- Override the service method (this is where the actual work is done by the servlet)
- Save the file with a `.java` filename extension

2. Compile the servlet

- Make sure `servlet.jar` is included in your classpath
- Invoke `javac`

3. Install the servlet

- Use the Java Web Server's Administration Tool to install it, and configure it.

4. Test the servlet

- Invoke the servlet from a JDK1.1-compatible browser.

We'll discuss each one of these steps in its simplest form.

4.1 WRITE THE SERVLET

The following class completely defines a servlet:

```
//Import needed Java packages
import java.io.*;
import javax.servlet.*;
```

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

// Create a class which inherits from GenericServlet or HttpServlet.
public class MyFirst extends HttpServlet
{
    /**
     * Override the service method.
     * Here we handle the HTTP GET method by building a simple web page.
     */

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        PrintWriter out;
        String title = "MyFirst Servlet Output";

        // set content type and other response header fields first
        response.setContentType("text/html");

        // then write the data of the response
        out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println(title);
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>This is output from MyFirst servlet.");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

```
}  
}
```

Don't forget to save the file with the name of the class and a .java filename extension -- in this case, we'll use MyFirst.java.

4.2 COMPILE THE SERVLET

Make sure the compiler will be able to find the servlet.jar file. The servlet.jar file contains Java Web Server's implementation of the Servlet API. Ensuring the compiler can find the file is easy to do using the cp option to the compiler, as we do below. (You can also modify the classpath environment variable temporarily from a command line or permanently in your system settings.)

Use javac, located in the Java Web Server's jre/bin directory, to compile the .java file. You'll need to copy the resulting .class file to the servlets directory of the Java Web Server so the web server will automatically find it. To skip the copying step, simply invoke javac with the -d option to direct the resulting .class file to the target directory.

For example, the following command compiles MyFirst.java and stores the resulting MyFirst.class file in the servlets directory:

```
javac -cp server_root/lib/servlet.jar -d server_root/servlets MyFirst.java
```

where server_root is the directory of the installed Java Web Server.

4.3 INSTALL THE SERVLET

The process for installing a servlet into a web server varies from web server to webserver. For the Java Web Server, the procedure is as follows:

1. Start the Java Web Server, if it is not already running
2. Log in to the Java Web Server on the administration port.

By default, the administration port is 9090 and the username/password is admin/admin. If you are successful, you should see the services page of the AdminTool applet.

3. Select the WebPage Service then click the Manage button. This should bring up the Web Service window.
4. Click the Servlets button, then select the Add entry in the list in the lefthand pane. You should now see the Add a New Servlet fields in the righthand pane.
5. Fill in the fields as follows:
 - Servlet Name: Choose any name you like, it need not be related to the filename. This will be the name used to invoke the servlet. We'll use FirstServlet.
 - Servlet Class: Provide the actual name of the servlet class, not file, you created -- for this example the real name is MyFirst.
 - Click the Add button and you will see your servlet name appear in the list of servlets in the lefthand pane under Configure. The righthand pane will have changed to two tabbed panels -- Configuration and Properties. We won't need these for our simple example so simply close the window.

4.4 TEST THE SERVLET

If you successfully compiled MyFirst.java and added it to the Java Web Server as described above, you should be able to invoke it from a JDK1.1-compatible browser. Use a URL of the following form:

`http://host_name:port/servlet/servlet_name`

For example:

`http://schnauzer:8080/servlet/FirstServlet`

CHAPTER 5 - FILTERS AND SERVLET CHAINING

In the Java Web Server, filtering support is provided by the ability to flexibly chain servlets. Filtering is yet another way to load and invoke servlets in the Java Web Server.

Both local and remote servlets can be part of a servlet chain. There are restrictions, however, on chaining the local internal servlets which provide the Java Web Server's own functionality. If an internal servlet is used in a chain, it must be the first servlet in the chain. Internal servlets include: file servlet, pageCompile servlet, ssinclude servlet, and template servlet. (The templatefilter servlet cannot be used as the first servlet--you would use template instead--but can be used elsewhere in the servlet chain.)

5.1 ENABLING FILTERS AND SERVLET CHAINING

In the basic subsection of the setup section, there is a button to indicate whether servlet chaining is enabled. By default, this is disabled; set the button to enable if you want servlet chaining and filtering capabilities in your server.

5.2 SERVLET CHAINS

For some requests, a chain of ordered servlets can be invoked rather than just one servlet. The input from the browser is sent to the first servlet in the chain and the output from the last servlet in the chain is the response sent back to the browser. Each servlet in the chain has the inputs and outputs piped to the servlet before and after it, respectively. A chain of servlets can be triggered for an incoming request by:

- Using Servlet Aliasing to indicate a chain of servlets for a request
- Using MIME types to trigger the next servlet in the chain

5.3 CONFIGURING A SERVLET CHAIN USING SERVLET ALIASING

Using the Servlet Aliasing subsection of the setup section in the Administration Tool, a list of servlets can be named for a particular URL request. In the servlets table, when adding a new mapping, enter a comma-separated list of servlets in the order in which they should be invoked when a request arrives for that particular

URL. This configures a servlet chain to be invoked every time a request that matches the URL arrives.

5.4 MIME TYPES AND SERVLETS

The Java Web Server provides a great deal of flexibility in how servlets can be invoked. One way of invoking servlets is by associating a servlet with a particular mime-type such that the servlet so configured is invoked each time a response with the corresponding mime-type is generated.

When a servlet writes to the OutputStream for the first time, the mime-type is checked. If there is a servlet configured for this particular mime-type, then the OutputStream of the response is piped to the InputStream of the servlet. Thus, the second servlet's request stream is essentially a pipe from the first servlet's response stream. In this way, many such servlets can be chained together.

5.4.1 Configuring a Servlet Chain Using MIME Types

There is no Administrative Tool support for servlet chaining. The administrator will need to manually edit the mimeservlets.properties file. This file resides in the directory

`<server_root>/properties/server/javawebserver/webpageservice/`

This file maps MIME types to servlet names. Servlet names are mapped to actual classes using the servlets section of the Administrative Tool. This flexible approach means that some of the servlets in the servlet chain can actually be configured to be remote servlets. In the case of remote servlets in the chain, they are brought over before being invoked.

5.4.2 Triggering Filter Chains

When servlets generate responses, they set the MIME type of the response. When a particular MIME type is configured to invoke another servlet in the mimeservlets.properties file, the filter chain is triggered. For servlets that get invoked in the chain, their request inputs are the outputs of the previous member in the chain.

5.5 SECURITY IN SERVLET CHAINS

All servlets in a servlet chain are subjected to the same security constraints when it comes to ACL-based protection. Two servlets cannot be protected by two different ACLs and belong in the same chain, since the request is from a single client. If they are configured to be protected by two different ACLs, then a response will not be generated.

Also, if a particular servlet is protected using an ACL in the chain, then the entire chain becomes protected. That is, if the third servlet in a chain is protected and the first two are not, then a response is not generated unless a valid user/password is entered before the chain is invoked.

Servlet sandboxes are preserved as usual for individual servlets. Individual servlets can be sandboxed if they are remote and they behave exactly as if they had not been in a servlet chain.

CHAPTER 6 - PROCESSING FORMS WITH SERVLETS

Up until now, Java technology has largely offered client-side benefits. That's about to change. Now web servers as well as web browser clients can be extended.

In the context of client-server software solutions--including the serving and browsing of Internet web pages--Java applets have enabled client browsers to extend their behavior by downloading compiled code from network servers. Applets have changed the nature and meaning of distributed computing. Applets have opened new worlds by enabling the convenient distribution of compiled code from a central server.

There's no reason that servers can't take advantage of Java technology in the same way that desktop applications. To this end, the Java Web Server is written in 100% Pure Java and is capable of being extended dynamically by loading compiled Java code known as a servlet. While applets provide a way of dynamically extending the functionality of client-side browsers, servlets let you dynamically extend the functionality of network servers.

Programmers should think of servlets as server-side components. Servlets are to servers what applets are to browsers. Servlet code can be downloaded into a running server to extend its behavior to provide new, or temporary, services to network clients.

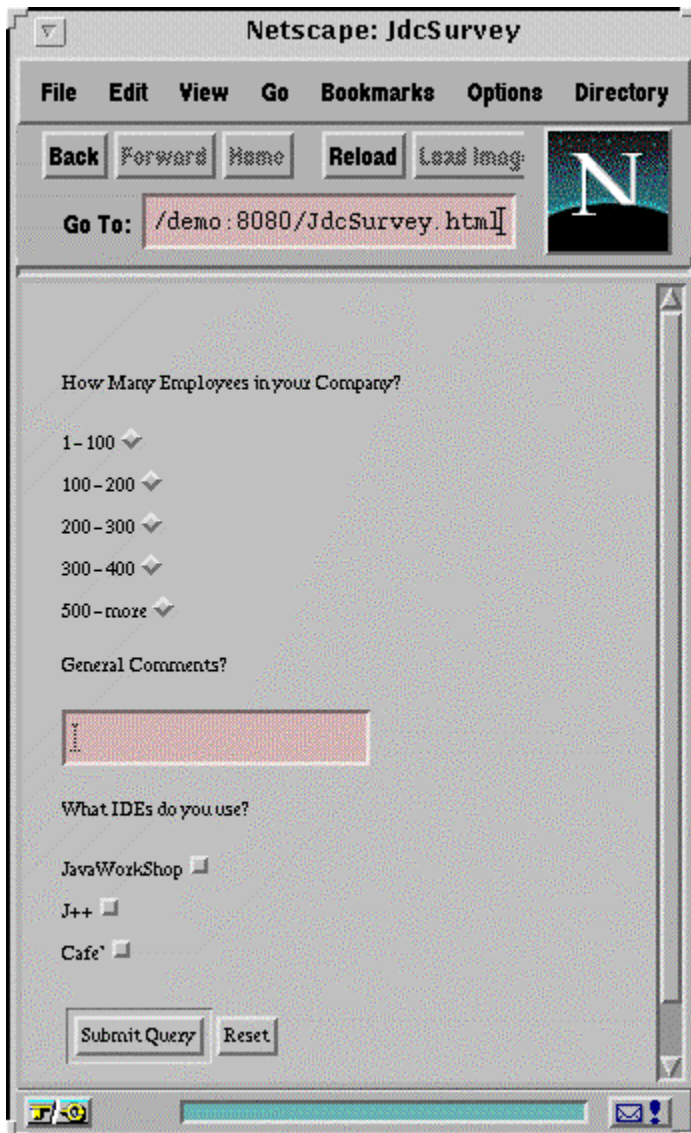
6.1 HELLO, CGI WORLD

Java Web Server can handle your legacy CGI scripts; you can use existing Perl code, shell scripts, or compiled C programs as you always have. To run your CGI scripts, see the Running Existing CGI scripts with Java Web Server online document.

Servlets, however, provide a faster and cleaner way to generate dynamic documents. You'll find servlets are an effective substitute for CGI scripts. This document shows you how simple it is to process an HTML form using a servlet rather than a CGI script.

There are two sides to form processing:

1. Specifying the input fields and widgets to be presented to the users in an HTML file
2. Processing data



The screenshot shows a Netscape browser window with the title 'Netscape: JdcSurvey'. The menu bar includes 'File', 'Edit', 'View', 'Go', 'Bookmarks', 'Options', and 'Directory'. Below the menu bar are buttons for 'Back', 'Forward', 'Home', 'Reload', and 'Load Image', along with a large 'N' logo. The address bar shows 'Go To: /demo:8080/JdcSurvey.html'. The main content area displays a survey form with the following sections:

- How Many Employees in your Company?**
 - 1 - 100
 - 100 - 200
 - 200 - 300
 - 300 - 400
 - 500 - more
- General Comments?**
 - A text input field.
- What IDEs do you use?**
 - JavaWorkShop
 - J++
 - Cafe'
- Buttons:** 'Submit Query' and 'Reset'.

The status bar at the bottom shows a file icon, a progress bar, and a mail icon.

First consider a form that looks like following figure.

Writing the CGI scripts to process this code could be quite complex. However, writing a Java servlet class to handle the task is straightforward.

The following is an example of the HTML used to present the form to user. Note that it uses the POST method in HTTP. This is for two reasons. First, since this data is going to be stored in a database, the GET method is inappropriate. Second, the POST method lets you pass much more data to the server than the GET method would. (GET methods are typically used for queries. Most forms should use POST.)

```
<html>
<head>
  <title>JdcSurvey</title>
</head>

<body>
  <form action=http://demo:8080/servlet/survey method=POST>
    <input type=hidden name=survey value=Survey01Results>

    <BR><BR>How Many Employees in your Company?<BR>
    <BR>1-100<input type=radio name=employee value=1-100>
    <BR>100-200<input type=radio name=employee value=100-200>
    <BR>200-300<input type=radio name=employee value=200-300>
    <BR>300-400<input type=radio name=employee value=300-400>
    <BR>500-more<input type=radio name=employee value=500-more>

    <BR><BR>General Comments?<BR>
    <BR><input type=text name=comment>

    <BR><BR>What IDEs do you use?<BR>
    <BR>JavaWorkShop<input type=checkbox name=ide value=JavaWorkShop>
```

```

<BR>J++<input type=checkbox name=ide value=J++>
<BR>Cafe'<input type=checkbox name=ide value=Cafe'>

<BR><BR><input type=submit><input type=reset>
</form>
</body>
</html>

```

This code file should be placed in the public_html directory immediately below the root directory where you installed your Java Web Server. The form specifies three basic questions to be answered by users. First, the question "How Many Employees in your Company?" offers a set of five radio buttons for potential responses.

```

<BR><BR>How Many Employees in your Company?<BR>
<BR>1-100<input type=radio name=employee value=1-100>
<BR>100-200<input type=radio name=employee value=100-200>
<BR>200-300<input type=radio name=employee value=200-300>
<BR>300-400<input type=radio name=employee value=300-400>
<BR>500-more<input type=radio name=employee value=500-more>

```

Second, the question "General Comments?" provides a single text input field where users can enter a general comment.

```

<BR><BR>General Comments?<BR>
<BR><input type=text name=comment>

```

Third, the question "What IDEs do you use?" provides a set of three check boxes. Unlike the radio buttons, these are not mutually exclusive choices. A user might use all three products: Java WorkShop, J++, and Cafe.

```

<BR><BR>What IDEs do you use?<BR>
<BR>JavaWorkShop<input type=checkbox name=ide value=JavaWorkShop>
<BR>J++<input type=checkbox name=ide value=J++>

```

`
Cafe'<input type=checkbox name=ide value=Cafe'>`

Finally, a Submit button is provided to send and process the form when the user has finished answering questions. A Reset button appears next to the Submit button in case the user wants to reset all fields to the original default values.

`

<input type=submit><input type=reset>`

Once you press the Submit button, the data in the form is sent as a data stream to the servlet specified in the form tag at the top of the HTML file:

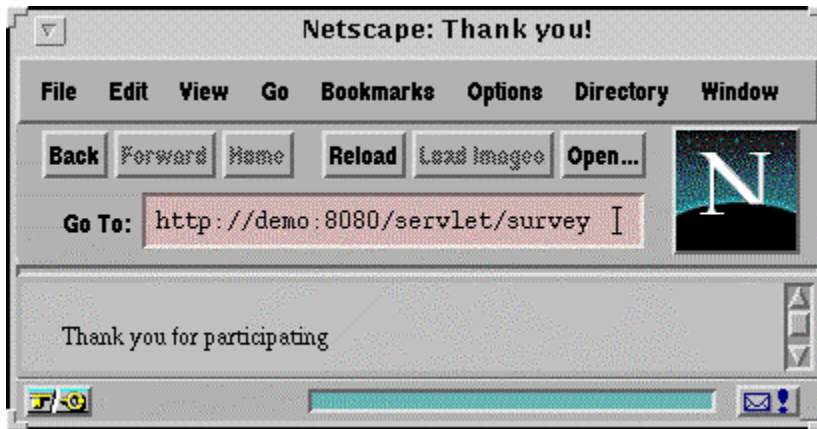
`<form action=http://demo:8080/servlet/survey method=POST>`

This syntax is very similar to the way in which you would specify that data in the HTML form to be processed by a CGI script (whether Perl, shell, or compiled C program).

After submitting the form shown in Figure 1, the text in file /tmp/Survey01Results.txt on host demo should look like this:

```
<BEGIN>
ide: JavaWorkShop
survey: Survey01Results
employee: 300-400
comment: Great Stuff!!!
<END>
```

Also, after submitting the form, the Java Web Server will display a new web page based on the output written to the response argument passed to the servlet's doPost method. (A query form, using the GET method, would call the doGet method of the servlet.) In this case a "thank you" message is displayed.



6.2 SO LONG, CGI

If you are accustomed to CGI script programming, you might notice how much simpler it is to write a servlet class to process the data than it is to write a comparable CGI script.

Here's the Java code required to process the form, once submitted, from the page described by the JdcSurvey.html file:

```
import java.io.*;
import java.util.*;
```

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

```
/**
```

```
 * A sample single-threaded servlet that takes input from a form and writes it out to a
 * file. It * is single threaded to serialize access to the file. After the results are written
 * to the file,
```

- * the servlet returns a "thank you" to the user.
- * You can run the servlet as provided, and only one thread will run a service method at a
- * time. There are no thread synchronization issues with this type of servlet, even though
- * the service method writes to a file. (Writing to a file within a service method requires
- * synchronization in a typical servlet.)

*

- * You can also run the servlet without using the single thread model by removing the `*implements` statement. Because the service method does not synchronize access to the `*file`, multiple threads can write to it at the same time. When multiple threads try to write to `*the file` concurrently, the data from one survey does not follow the data from another `*survey` in an orderly fashion.

*

- * To see interaction (or lack of interaction) between threads, use at least two browser `*windows` and have them access the servlet as close to simultaneously as possible. `*Expect correct results` (that is, expect no interference between threads) only when the `*servlet` implements the `SingleThreadedModel` interface.

`*/`

```
public class SurveyServlet extends HttpServlet
    implements SingleThreadModel
{
    String resultsDir;

    public void init()
        throws ServletException
    {
        resultsDir = getInitParameter("resultsDir");
        if (resultsDir == null) {
```

```
Enumeration initParams = getInitParameterNames();
System.err.println("The init parameters were: ");
while (initParams.hasMoreElements()) {
    System.err.println(initParams.nextElement());
}
System.err.println("Should have seen one parameter name");
throw new UnavailableException (this,
    "Not given a directory to write survey results!");
}
}

/**
 * Write survey results to output file in response to the POSTed
 * form. Write a "thank you" to the client.
 */
public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
{
    // first, set the "content type" header of the response
    res.setContentType("text/html");

    //Get the response's PrintWriter to return text to the client.
    PrintWriter toClient = res.getWriter();

    try {
        //Open the file for writing the survey results.
        String surveyName = req.getParameterValues("survey")[0];
        FileWriter resultsFile = new FileWriter(resultsDir
            + System.getProperty("file.separator")
            + surveyName + ".txt", true);
```

```
PrintWriter toFile = new PrintWriter(resultsFile);

    // Get client's form data & store it in the file
    toFile.println("<BEGIN>");
    Enumeration values = req.getParameterNames();
    while(values.hasMoreElements()) {
        String name = (String)values.nextElement();
        String value = req.getParameterValues(name)[0];
        if(name.compareTo("submit") != 0) {
            toFile.println(name + ": " + value);
        }
    }
    toFile.println("<END>");

    //Close the file.
    resultsFile.close();

    // Respond to client with a thank you
    toClient.println("<html>");
    toClient.println("<title>Thank you!</title>");
    toClient.println("Thank you for participating");
    toClient.println("</html>");

} catch(IOException e) {
    e.printStackTrace();
    toClient.println(
        "A problem occurred while recording your answers. "
        + "Please try again.");
}
```

```
// Close the writer; the response is done.  
    toClient.close();  
}  
}
```

This servlet source file, and the compiled servlet class, should also be placed in the `servlets` directory immediately below the root directory where you installed your Java Web Server. Before clients can make use of the extended server functionality added by the `SurveyServlet` servlet class, you will also have to install the servlet using the Administration Tool.

6.3 SERVING UP SERVLETS

A quick walk through the code should clarify the way in which the form is processed by the `doPost` method of the `SurveyServlet` class.

First, two different writers are opened for writing text. One is opened to record the results of the survey and to give feedback to the users. Note that the writer that returns a response to the user is only accessed after the content type for the response is set.

```
// first, set the "content type" header of the response  
res.setContentType("text/html");  
  
//Get the response's PrintWriter to return text to the client.  
PrintWriter toClient = res.getWriter();  
  
try {  
    //Open the file for writing the survey results.  
    String surveyName = req.getParameterValues("survey")[0];  
    FileWriter resultsFile = new FileWriter(resultsDir  
        + System.getProperty("file.separator")  
        + surveyName + ".txt", true);
```

```
PrintWriter toFile = new PrintWriter(resultsFile);
```

To enable the survey results to be written to a file, a `FileWriter` object is associated with a `PrintWriter` object. The resulting `PrintWriter` is called `toFile`. Anything written to `toFile` will be saved in the survey results file. Similarly, anything written to the `PrintWriter` called `toClient` will be printed on the HTML page generated by the Java Web Server as a result of processing the form.

When looking at the creation of the `FileWriter`'s file name, recall that the survey name is set in the

`JdcSurvey.html`

file:

```
<input type=hidden name=survey value=Survey01Results>
```

and retrieved by the

`doPost`

method:

```
String surveyName = req.getParameterValues("survey")[0];
```

and that the `resultsDir` holds the string that was passed in as the value of the `resultsDir` property. Its value was stored in the `SurveyServlet`'s `resultsDir` field in the `init` method:

```
resultsDir = getInitParameter("resultsDir");
```

The value of the `resultsDir` property was set in the Administration Tool to `resultsDir=/tmp` when the `SurveyServlet` was added to the list of supported servlets. (See the Administration Tool: Adding a Servlet for more information on making the `SurveyServlet` a supported servlet.)

Thus, the name of the file created by:

```
FileWriter resultsFile = new FileWriter(resultsDir  
    + System.getProperty("file.separator")  
    + surveyName + ".txt", true);
```

is /tmp/Survey01Results.txt.

Output that is written to toFile, such as:

```
// Get client's form data & store it in the correct file
toFile.println("<BEGIN>");
...
toFile.println(name + ": " + value);
...
toFile.println("<END>");
```

will be written to the file, /tmp/Survey01Results.txt.

Output that is written to toClient, such as:

```
toClient.println("<html>");
toClient.println("<title>Thank you!</title>");
toClient.println("Thank you for participating");
toClient.println("</html>");
```

will appear in the client's browser after the form has been processed and control returns from JdcSurvey01.doPost.

To understand how a servlet processes form arguments, you must know that input for the form processing is read from the req argument to the doPost method and output is written from the res argument to the doPost method.

```
public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
```

More specifically, the req method contains a list of parameters that can be retrieved with the HttpServletRequest.getParameterNames method.

```
Enumeration values = req.getParameterNames();
```

With the values enumeration object returned by this call, you can now set up a loop to process each of the parameters passed by the HTML form to the servlet doPost routine:

```
while(values.hasMoreElements()) {  
    ...  
}
```

The name of the parameter is retrieved on each iteration by Enumeration.nextElement, while the associated values are retrieved by HttpServletRequest.getParameterValues, which accepts the parameter name as an argument. Note that getParameterValues returns an array of values; merely access the first element in the array if there is only one value for the parameter.

```
String name = (String)values.nextElement();  
String value = req.getParameterValues(name)[0];
```

The only value you don't want to print in the results file is the value of the submit parameter, which is passed because submit was specified as a button in the html file. The servlet doPost routine weeds out this parameter with a simple string comparison:

```
if(name.compareTo("submit") != 0) {  
    toFile.println(name + ": " + value);  
}
```

The only other detail worth mentioning is the exception-handling capabilities provided by servlets. Be sure to specify that your doPost and init methods in your HttpServlet subclass can each throw a ServletException:

```
public void init (ServletConfig config) throws ServletException  
...  
public void doPost (HttpServletRequest req, HttpServletResponse res,  
    ) throws ServletException, IOException  
...
```


Inside the SurveyServlet servlet, the doPost method uses a try/catch block to help provide the user and the programmer with adequate feedback when something goes wrong. The user is warned if a problem occurs while attempting to write the survey results to the file. The programmer is given a stack trace, dumped to the standard error device.

```
try {  
    ...  
} catch(IOException e) {  
    e.printStackTrace();  
    toClient.println(  
        "A problem occurred while recording your answers. "  
        + "Please try again.");  
}
```

CHAPTER 7 - WRITING SERVLETS AS BEANS

One of the important ways that developers can take advantage of JavaBeans technology is to create and use servlets that are themselves beans. Servlets that are beans offer two distinct advantages:

1. If the servlet is a bean, any changes to its configuration take effect immediately.
2. The persistent state of the servlet bean, as well as its configuration, can be stored as a serialized file. For example, if the servlet bean manages a counter, the state of the counter can be made persistent through the Java serialization mechanism.

Thus, if you stop and restart the server, the counter would retain its latest value.

Another important way that developers can take advantage of JavaBeans technology is to use JavaBeans components in servlets and JSP (or JHTML) files.

7.1 INTRODUCTION

A servlet bean is a servlet that adheres to the JavaBeans design pattern for getting and setting properties. This pattern defines a standard method/class combination for getting and setting properties, and uses mandatory type signatures and naming conventions. The design pattern for read/write properties is:

```
void setFoo(Xyz x);  
Xyz getFoo();
```

The JavaServer assumes that all servlets are beans. The server sets all of the servlet bean properties (passed to the servlet as initial arguments) by using JavaBeans introspection.

Servlet beans can be distributed as:

1. Class files
2. Serialized files (servlet_name.ser files)
3. Class and serialized files in JAR archive format

Servlet beans can be serialized into .ser files. A serialized file can be thought of as an instance of the class. The serialized servlet bean files and the class file they reference can be packaged together in a JAR file. JAR files are also referred to as installed files.

Serialized servlet bean files and the class file they reference can also reside outside of a JAR file. These files are referred to as dropped-in files.

The distinction between installed files and dropped-in files is that you cannot use the JavaServer Administration Tool to change the initial arguments specified in dropped-in serialized server bean files (dropped-in serialized files are read-only). Nor can you use the Administration Tool to create serialized files for dropped-in class files. Instead, use the Administration Tool to specify properties and initial values for the dropped-in files.

NOTE: All references to servlet bean files in this document refer to installed files. This document does not describe dropped-in files unless they are mentioned specifically.

7.2 SERVLET BEAN INSTANTIATION

The server provides support for instantiating servlet beans by providing a class loader that deserializes the server beans files from .ser files. These .ser files can be dropped-in files or they can reside in a JAR file. When a JAR or dropped-in file changes on disk, the class loader automatically reloads it.

Servlet beans files can reside in any of these directories:

1. servlet/ directory
2. servletbeans/ directory
3. Any directory included in the CLASSPATH

To take advantage of the automatic reloading feature of the JavaServer, store your servlet beans files (class, .ser, and .jar files) in the servletbeans/ directory and not

include it in the CLASSPATH. (Automatic reloading is not performed on files residing in directories in the CLASSPATH.)

7.3 SERVLET BEAN PROPERTIES

Properties are the initial arguments that are passed to a servlet bean. For example, for a servlet bean named "Counter", there could be an initial argument initial=0. Similarly, for a servlet bean named "CGI", there could be an initial argument bindir=cgi-bin.

7.3.1 Passing Properties to Servlet Beans

Properties are passed to the servlet bean by:

1. Using the ServletConfig interface (the server always passes properties with this interface)
2. Calling the set methods for the property by using JavaBeans introspection

7.3.2 Using the ServerConfig Interface

The ServerConfig interface passes properties to the servlet bean when the init method of the Servlet interface is called. The init method is called by network service when it loads the servlet.

7.3.3 Calling Set Methods for the Property

Properties can also be passed to the servlet bean by calling set methods for the corresponding property. For example, the following code snippet sets the value for the argument PropertyName:

```
void setPropertyName (String);
```

This method will be called for every property that you set by using the Administration Tool. After the servlet is loaded, if you use the Administration Tool to change properties, this method will be called with a new value.

7.4 SERVLET BEAN INSTALLATION

When you have installed the servlet bean, the server knows about the servlet bean's existence. The way in which you install a servlet bean depends on whether or not you have configured the servlet bean (that is, whether or not you have provided a .ser file that contains the appropriate initial arguments for all of the servlet bean's properties). There are two ways in which you can install servlet beans:

1. Installing configured servlet beans
2. Installing servlet beans that are not configured

7.4.1 Installing Configured Servlet Beans

If your servlet bean is configured (that is, if it has a .ser file that contains initial arguments), move the servlet bean file into the appropriate directory. Usually, this will be the servletbeans/ directory. If the directory is not on the CLASSPATH, the server will automatically reload the servlet.

NOTE: If your servlet is a class file or a .ser file, you will not be able to use the Administration Tool to change any of the servlet bean's properties. You can use the Administration Tool to change the properties of only those servlets that are contained in .jar files.

7.5 INSTALLING UNCONFIGURED SERVLET BEANS

If your servlet bean is not configured, use the Administration Tool to specify initial arguments for it.

Invoke the Administration Tool.

Click Web Service.

Click the Manage button.

In the Web Service dialog box, click the Servlets icon.

Click the Add branch of the tree and enter the following information for the servlet:

Servlet Name. The unique name of the servlet you are adding.

Servlet Class. The name of the Java class for the servlet. This consists of the package name without the .class extension. For example, sun.server.http.HttpServlet is a valid class name.

Click Yes and type a name for the .jar file that will contain the servlet bean.

Click Add. The Administration Tool displays the Configuration and Properties tabs.

Enter the following information for the servlet:

Description. A text string describing the servlet. This field is sometimes blank.

Class Name. The name of the associated class file for the servlet.

Load at Startup. Whether the Java Web Server loads the servlet when the server starts.

Loaded Now. Whether the servlet is currently loaded.

Load Remotely. Whether the Java Web Server loads the servlet from a remote location.

Class File URL. The URL that points to the class file for the remote servlet.

Click Save, then click the Properties tab.

Enter the initial arguments for the properties in the Value field.

Click the Add button, then click the Save button to save your changes to the Java Web Server.

7.6 INVOKING SERVLET BEANS

After the servlet bean is loaded, you can invoke it in the same way as you would invoke any other servlet:

1. Call the servlet bean directly by using a URL
2. Embed the servlet bean in a server-side include statement inside an HTML document
3. Invoke an alias for the servlet bean

7.6.1 Invoking Servlet Beans Using a URL

Servlet bean files can be invoked by calling them as a URL address. Note that even though you installed the servlet bean in the servletbeans/ directory, you call it with a

servlet/ URL. The server will search the servletbeans/ directory for the appropriate file.

`http://host-name:port/servlet/ServletName`

ServletName can represent either the name of a servlet bean class file or a .ser file. You invoke a class file or a serialized file with the name of the file without the .class or .ser extension. You can do this regardless of whether the file resides in a directory "as is" (that is, it is a dropped-in file), or it resides in a JAR file.

For example, a servletbeans/ directory contains the files Counter.class, counter1.ser, and a JAR file Hello.jar. The JAR file contains the files Hello.class, hello1.ser, and hello2.ser.

The following table describes how to invoke various servlet beans files.

To Invoke This Servlet Bean File	Use This URL Address
Serialized file in the servletbeans/ directory	Use the file name without the .ser extension. For example: <code>http://host:port/servlet/counter1</code>
Serialized file in a .jar file in the servletbeans/ directory	Use the file name without the .ser extension. For example: <code>http://host:port/servlet/hello1</code>
Class file in the servletbeans/ directory	Use the file name without the .class extension. For example: <code>http://host:port/servlet/Counter</code>
Class file in a .jar file in the servletbeans/ directory	Use the file name without the .class extension. For example: <code>http://host:port/servlet/Hello</code>

7.6.2 An Example

The servlet `Hello.class` creates a web page with a greeting. A code snippet from `Hello.class` appears below.

```
String greeting = "Hello World"
```

```
//code to get, set, and display greetings
```

```
Public String getString {  
    return greeting;  
}  
Public void setSrting(String g){  
    greeting = g;  
}  
public void doGet (HttpServletRequest req, HttpServletResponse res)  
    throws ServletException, IOException  
{  
    PrintWriter        out;  
  
    res.setContentType("text/html");  
    out = res.getWriter();  
  
    out.println("< html >");  
    out.println("< head >< title >Greeting </title >< /head >");  
    out.println("< body >");  
    out.println("< h1 >" + getGreeting() + "< /h1 >");  
    out.println("< /body > < /html >");  
}
```


There are also two .ser files, hello1.ser and hello2.ser, which contain alternate greetings for the servlet. These three files, Hello.class, hello1.ser, and hello2.ser, are archived as a .jar file and stored in the servletbeans/ directory.

The hello1.ser file contains the initial argument Greeting="Good Morning" and hello2.ser contains the initial argument Greeting="Good Evening".

The invocation `http://server-root /servlet/hello1` returns "Good Morning".

The invocation `http://server-root /servlet/hello2` returns "Good Evening".

The invocation `http://server-root /servlet/Hello` returns "Hello World".

7.6.3 Embedding Servlet Beans in a Server-side Include Statement

You can invoke a servlet bean by embedding it in a server-side include statement inside an HTML document. For example, in an .shtml file, you can add a server side include with this statement:

```
< servlet name="servletAliasName" code=ServletBeanExample > < /servlet >
```

In this example, `name="servletAliasName"` represents an alias for the servlet bean and `code=ServletBeanExample` represents the name of the servlet class. Note that the code value can be omitted if an alias has been created for the servlet bean with the Administration Tool.

7.6.4 Invoking an Alias for the Servlet Bean

You can use the Servlet Aliases page of the Administration Tool to create an alias for the servlet bean. The Servlet Aliases page lets you specify the path name mapping rules that the JavaServer uses to invoke servlets. These rules allow you to use a shortcut URL to call a servlet from your browser, to embed the shortcut within HTML documents (using server-side includes), or to embed the shortcut within other Java programs.

CHAPTER 8 - SESSION TRACKING

Session tracking is a mechanism for building a sophisticated state-aware model on top of the web's stateless protocol. With session tracking, the server maintains session state through the use of cookies or URL rewriting. The server creates new sessions, invalidates old sessions, and maintains policies of session persistence.

As a developer, you should read all sections, especially Using Session Tracking from a Servlet where example code is given. As an administrator, refer to Customizing Session Tracking to see how Session Tracking can be tuned.

Note:

When the Java Web Server is installed, it creates an empty sessionSwap directory. When the server is shut down, or if there are too many sessions to be held in physical memory, they are swapped out to that directory to be held on disk. This is invisible to the user but may be a consideration for server performance tuning.

8.1 INTRODUCTION TO SESSION TRACKING

Session tracking is a flexible, lightweight mechanism that enables stateful programming on the web. Its general implementation serves as a basis for more sophisticated state models, such as persistent user profiles or multi-user sessions.

A session is a series of requests from the same user that occur during a time period. This transaction model for sessions has many benefits over the single-hit model. It can maintain state and user identity across multiple page requests. It can also construct a complex overview of user behavior that goes beyond reporting of user hits.

8.2 SERVER-SIDE SESSION OBJECTS AND USERS

Session tracking gives servlets and other server-side applications the ability to keep state information about a user as the user moves through the site. Server-side

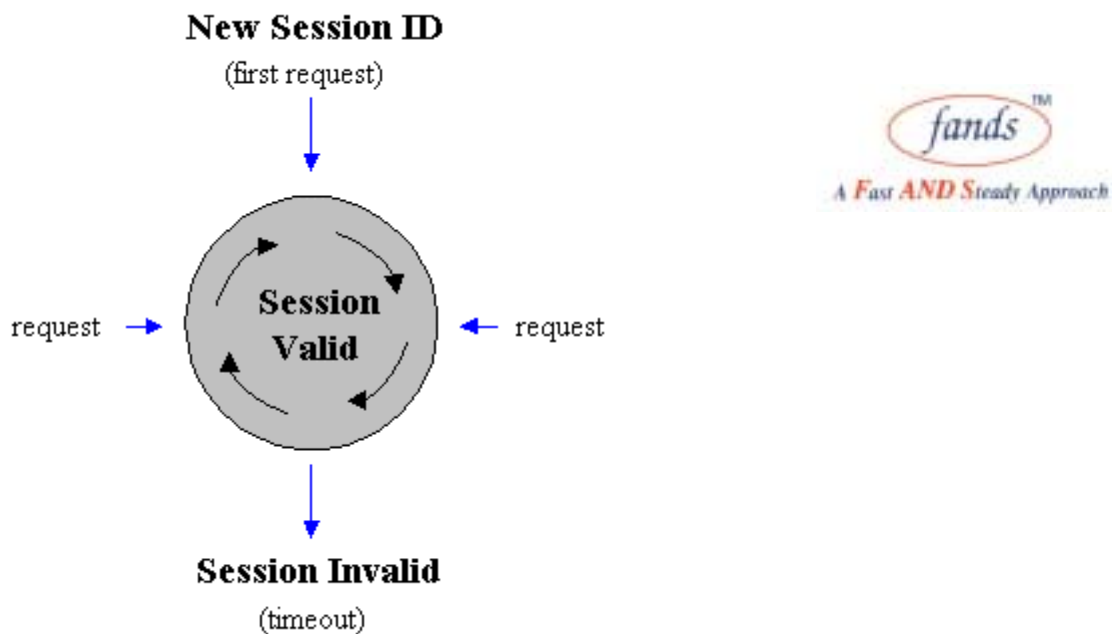
applications can use this facility to create more stateful user experiences and to track who's doing what on the site.

The Java Web Server maintains user state by creating a Session object for each user on the site. These Session objects are stored and maintained on the server. When a user first makes a request to a site, the user is assigned a new Session object and a unique session ID. The session ID matches the user with the Session object in subsequent requests. The Session object is then passed as part of the request to the servlets that handle the request. Servlets can add information to Session objects or read information from them.

8.3 SESSION ENDURANCE

After the user has been idle for more than a certain period of time (30 minutes by default), the user's session becomes invalid, and the corresponding Session object is destroyed.

A session is a set of requests originating from the same browser, going to the same server, bounded by a period of time. Loosely speaking, a session corresponds to a single sitting of a single anonymous user (anonymous because no explicit login or authentication is required to participate in session tracking).



8.4 USING SESSION TRACKING FROM A SERVLET

Session-tracking interfaces are in the `javax.servlet.http` package. The following example uses the `doGet` method from a servlet that prints the number of times users access a particular servlet.

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // Obtain the Session object
    HttpSession session = request.getSession (true);

    // Get the session data value
    Integer ival = (Integer)
        session.getValue ("sessiontest.counter");
    if (ival == null) ival = new Integer (1);
    else ival = new Integer (ival.intValue () + 1);
    session.putValue ("sessiontest.counter", ival);

    // Output the page
    response.setContentType("text/html");
}
```

```
ServletOutputStream out = response.getOutputStream();
out.println("<html>");
out.println("<head><title>Session Tracking Test</title></head>");
out.println("<body>");
out.println("<h1>Session Tracking Test</h1>");
out.println ("You have hit this page " + ival + " times");
out.println("</body></html>");
}
```

The first part of the `doGet` method associates the `Session` object with the user making the request. The second part of the method gets an integer data value from the `Session` object and increments it. The third part outputs the page, including the current value of the counter.

When run, this servlet should output the value of the counter that increments every time you reload the page. You must obtain the `Session` object before you actually write any data to the servlet's output stream. This guarantees that the session tracking headers are sent with the response.

8.5 STORING AND RETRIEVING DATA FROM THE SESSION OBJECT

You obtain the session data value with the following:

```
Integer ival = (Integer)
    session.getValue ("sessiontest.counter");
if (ival == null) ival = new Integer (1);
else ival = new Integer (ival.intValue () + 1);
session.putValue ("sessiontest.counter", ival);
```

The `Session` object has methods similar to `java.util.Dictionary` for adding, retrieving, and removing arbitrary Java objects. In this example, an `Integer` object is read from the `Session` object, incremented, then written back to the `Session` object.

Any name, such as `sessiontest.counter`, may be used to identify values in the Session object. When choosing names, remember that the Session object is shared among any servlets that the user might access. Servlets may access or overwrite each other's values from the Session. Thus, it is good practice to adopt a convention for organizing the namespace to avoid collisions between servlets, such as:

`servletname.name`

8.6 SESSION INVALIDATION

Sessions can be invalidated automatically or manually. Session objects that have no page requests for a period of time (30 minutes by default) are automatically invalidated by the Session Tracker `sessionInvalidationTime` parameter. When a session is invalidated, the Session object and its contained data values are removed from the system.

After invalidation, if the user attempts another request, the Session Tracker detects that the user's session was invalidated and creates a new Session object. However, data from the user's previous session will be lost.

Session objects can be invalidated manually by calling `Session.invalidate()`. This will cause the session to be invalidated immediately, removing it and its data values from the system.

Note: To see how to change the default session invalidation time, refer to **Customizing Session Tracking**.

8.7 HANDLING NON-COOKIE BROWSERS (URL REWRITING)

The Session Tracker uses a session ID to match users with Session objects on the server side. The session ID is a string that is sent as a cookie to the browser when the user first accesses the server. On subsequent requests, the browser sends the session ID back as a cookie, and the server uses this cookie to find the session associated with that request.

There are situations, however, where cookies will not work. Some browsers, for example, do not support cookies. Other browsers allow the user to disable cookie support. In such cases, the Session Tracker must resort to a second method, URL rewriting, to track the user's session.

URL rewriting involves finding all links that will be written back to the browser, and rewriting them to include the session ID. For example, a link that looks like this:

```
<a href="/store/catalog">
```

might be rewritten to look like this:

```
<a href="/store/catalog;$sessionid$DA32242SSGE2">
```

If the user clicks on the link, the rewritten form of the URL will be sent to the server. The server's Session Tracker will be able to recognize the `;$sessionid$DA32242SSGE2` and extract it as the session ID. This is then used to obtain the proper Session object.

Implementing this requires some reworking by the servlet developer. Instead of writing URLs straight to the output stream, the servlet should run the URLs through a special method before sending them to the output stream. For example, a servlet that used to do this:

```
out.println("<a href=\"/store/catalog\">catalog</a>");
```

should now do this:

```
out.print ("<a href=\"");
```

```
out.print (response.encodeUrl ("/store/catalog"));
```

```
out.println ("\">catalog</a>");
```

The `encodeUrl` method performs two functions:

Determine URL Rewriting: The `encodeUrl` method determines if the URL needs to be rewritten. Rules for URL rewriting are somewhat complex, but in general if the server detects that the browser supports cookies, then the URL is not rewritten. The server tracks information indicating whether a particular user's browser supports cookies.

Return URL (modified or the same): If the `encodeUrl` method determined that the URL needs to be rewritten, then the session ID is inserted into the URL and returned. Otherwise, the URL is returned unmodified.

In addition to URLs sent to the browser, the servlet must also encode URLs that would be used in `sendRedirect()` calls. For example, a servlet that used to do this:

```
response.sendRedirect ("http://myhost/store/catalog");
```

should now do this:

```
response.sendRedirect  
    (response.encodeRedirectUrl ("http://myhost/store/catalog"));
```

The methods `encodeUrl` and `encodeRedirectUrl` are distinct because they follow different rules for determining if a URL should be rewritten.

8.7.1 Multiple Servlets

URL conversions are required only if the servlet supports session tracking for browsers that do not support cookies or browsers that reject cookies. The consequences of not doing these conversions is that the user's session will be lost if the user's browser does not support cookies and the user clicks on an un-rewritten URL. Note that this can have consequences for other servlets. If one servlet does not follow these conventions, then a user's session could potentially be lost for all servlets.

8.8 USING SESSION TRACKING WITH THE PAGE COMPILER

Page compilation is a feature of the Java Web Server that allows HTML pages containing Java code to be compiled and run as servlets. Page compilation also simplifies the task of supporting session tracking. To that end, if URL rewriting is enabled, page compilation automatically adds the `encodeUrl` call to links in the HTML page.

For example, the Access Count Example could be rewritten as a `.jhtml` file like this:

```
<html>  
<head><title>Session Tracking Test</title></head>  
<body>  
<h1>Session Tracking Test</h1>
```



```
<java type=import>javax.servlet.http.*</java>
```

```
<java>
```

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

```
    // Get the session data value
```

```
    Integer ival = (Integer)
```

```
        session.getValue ("sessiontest.counter");
```

```
    if (ival == null) ival = new Integer (1);
```

```
    else ival = new Integer (ival.intValue () + 1);
```

```
    session.putValue ("sessiontest.counter", ival);
```

```
</java>
```

You have hit this page <java type=print>ival</java> times.

```
<p>Click here to go to the <a href="/store/catalog">catalog</a>
```

```
</body></html>
```

This example is similar to the servlet code in the previous example, except that the Java code has been inserted directly into the HTML source. In this example, the `/store/catalog` link will be detected by the Page Compiler and will automatically call `encodeUrl`.

Note: The Page Compiler will not detect URLs in the Java code. If the Java code outputs URLs, then the Java code must still run those URLs through `encodeUrl`.

8.9 ADDITIONAL APIs

In addition to the `Session` object, there is another class that may interest the servlet developer.

Description	Class
HttpSessionBinding	HttpSessionBindingListener is an interface that can be implemented by

Listener	objects placed into a Session. When the Session object is invalidated, its contained values are also removed from the system. Some of these values may be active objects that require cleanup operations when their session is invalidated. If a value in a Session object implements HttpSessionBindingListener, then the value is notified when the Session is invalidated, thereby giving the object a chance to perform any necessary cleanup operations.
----------	---

8.10 SESSION SWAPPING AND PERSISTENCE

An Internet site must be prepared to support many valid sessions. A large site, for example, might have hundreds, or even thousands, of simultaneously valid sessions. Because each session can contain arbitrary data objects placed there by the application servlets, the memory requirements for the entire system can grow prohibitively large.

To alleviate some of these problems, the session tracking system places a limit on the number of Session objects that can exist in memory. This limit is set in the session.maxresidents property. When the number of simultaneous sessions exceeds this number, the Session Tracker swaps the least recently-used sessions out to files on disk. Those sessions are not lost: they will be reloaded into memory if further requests come in for those sessions. This system allows for more sessions to remain valid than could exist in memory.

Session invalidation is not affected by session swapping. If a session goes unused for longer than the normal invalidation time, the session is invalidated, whether it is in memory or on disk. Session invalidation is set in the session.invalidationinterval property. See Customizing Session Tracking for more information.

Sessions are written to and read from disk using Java serialization. For this reason, only serializable objects put into the Session object will be written to disk. Any objects put into the Session object that are not serializable will remain in memory, even if the rest of the Session object has been written to disk. This does not affect session tracking, but does reduce the memory savings that the Session Tracker gets

from swapping a session to disk. For this reason, the servlet developer should try to put only serializable objects into the Session object. Serializable objects are those that implement either `java.io.Serializable` or `java.io.Externalizable`.

The session-swapping mechanism is also used to implement session persistence, if the session persistence feature is enabled. When the server is shut down, sessions still in memory are written to the disk as specified in the `session.swapdirectory` property. When the server starts again, sessions that were written to disk will once again become valid. This allows the server to be restarted without losing existing sessions. Only serializable data elements in the session will survive this shutdown/restart operation.

Note: Session persistence is intended for preserving sessions across server restarts. It is not meant to be used as a general long-term session persistence mechanism.

8.11 CUSTOMIZING SESSION TRACKING

8.11.1 Multiple Servers

When running multiple servers on one machine, you may get an error regarding failure of the additional Session Service startup. This is because both server's Session Service are attempting to use the default Session Service port of 9094. To avoid this error, manually edit the `endpoint.main.port` property to change the additional Session Service to some port other than 9094. This property is kept in the `endpoint.properties` files at:

`server_root/sessionsservice/endpoint.properties`

where `server_root` is the directory into which you installed the Java Web Server product.

8.11.2 Server-wide Session Tuning

Property settings are applied server-wide and cannot be tuned for individual sessions. The properties are kept in the `server.properties` files at:

`server_root/properties/server/javawebserver/sessionsservice/session.properties`

where `server_root` is the directory into which you installed the Java Web Server product.

Parameter	Description	Default
session.invalidat ioninterval	Time interval when Java Web Server checks for sessions that have gone unused long enough to be invalidated. Value is an integer, specifying the interval in milliseconds.	10000 (10 seconds)
session.swapinterval	Time interval when Java Web Server checks if too many sessions are in memory, causing the overflow of sessions to be swapped to disk. Value is an integer, specifying the interval in milliseconds.	10000 (10 seconds)
session.persistenc e	Boolean value specifying if Java Web Server keeps session data persistent. If <code>true</code> , sessions are swapped to disk when Java Web Server shuts down and are revalidated from disk when it restarts. If <code>false</code> , Java Web Server removes session swap files every time it starts.	true
session.swapdirect ory	Name of directory that the Java Web Server uses to swap out session data. No other data should be kept in this directory.	sessionSwap
session.maxresiden ts	Number of sessions allowed to remain in memory at once. If the number of sessions exceeds this number, sessions will be swapped out to disk on a least recently used basis to reduce the number of resident sessions.	1024
session.invalidati ontime	Amount of time a session is allowed to go unused before it is invalidated. Value is specified in milliseconds.	1800000 (30 minutes)
enable.sess ions	Boolean value specifying whether Session Tracking is active. If <code>false</code> , then the Java Web Server performs no function for extracting or inserting session IDs into requests.	true
enable.co okies	Boolean value indicating whether Java Web Server uses cookies as a vehicle for carrying session ID. If <code>true</code> , session IDs arriving as cookies are recognized and the Java Web Server tries to use cookies as a means for sending the session ID.	true
enable.ur lrewritin g	Boolean value indicating whether Java Web Server uses rewritten URLs as a vehicle to carry the session ID. If <code>true</code> , then session IDs arriving in the URL are recognized, and the Java Web Server rewrites URLs if necessary to send the session ID.	false
enable.pr otocolswi tchrewrit ing	Boolean value indicating whether the session ID is added to URLs when the URL dictates a switch from "http" to "https" or vice-versa.	false

session.cookie.name	Name of the cookie used to carry the session ID, if cookies are in use.	jwssessionid
session.cookie.comment	Comment of the cookie used to carry the session ID, if cookies are in use.	Java Web Server Session Tracking Cookie
session.cookie.domain	If present, this defines the value of the domain field that is sent for session cookies.	null
session.cookie.maxage	If present, this defines the value of the maximum age of the cookie.	-1
session.cookie.path	If present, this defines the value of the path field that will be sent for session cookies.	"/"
session.cookie.secure	If true, then session cookies will include the secure field.	false

CHAPTER 9 - LOADING AND INVOKING SERVLETS

9.1 LOADING SERVLETS

Servlets can be loaded from any one of these places:

- From a directory that is on the CLASSPATH. The CLASSPATH of the Java Web Server includes `server_root/classes/`, which is where the system classes reside.
- From the `server_root/servlets/` directory. This is not in the server's CLASSPATH. A class loader is used to create servlets from this directory. New servlets can be added or existing servlets can be recompiled and the server will notice these changes.
- From a remote location. For this a codebase like `http://nine.eng/classes/foo/` is required in addition to the servlet's class name.

9.2 LOADING REMOTE SERVLETS

Remote servlets can be loaded by:

- Configuring the servlet in the Administration Tool to set up automatic loading of remote servlets
- Setting up server side include tags in .shtml files
- Defining a filter chain configuration

9.3 IDENTIFYING SERVLETS

Servlets are identified by `servletName` which is either:

- A virtual name that is assigned to the servlet by using the Adding Servlets section of the Administration Tool
- Its own class name if the servlet is dropped into the `server_root/servlets/` directory and invoked by its class name

9.4 INVOKING SERVLETS

A servlet invoker is a servlet that invokes the "service" method on a named servlet. If the servlet is not loaded in the server, the invoker loads the servlet (either from local disk or from the network) and then invokes the "service" method. Like applets, local servlets in the server can be identified by just the class name. In other words, if a servlet name is not absolute, it is treated as local.

Servlets can be invoked by a client in the following ways:

- 1) The client can ask for a document that is served by the servlet.

The server gets a request for a document, looks up the configuration parameters, and finds out that the document is not a static document residing on the disk, but a

live document generated by a servlet object; the server forwards the request to the servlet, which services the request by generating the output. This method is the same one used by traditional web servers for invoking CGI scripts.

2) The client (browser) can invoke the servlet directly using a URL, once it has been mapped to servletName using the Servlet Aliases section of the administration GUI. The client invokes the servlet with a URL of the form:

`http://server_host_name/servlet/servlet URL`

where servlet URL is a regular URL that points to the location of the servlet. The host where the servlet resides might be different from the host where the server is running. In this case the servlet class will be dynamically downloaded to the server, instantiated, and then run. The reason this invocation is useful is that it does not need any special understanding of servlets on the part of the browser.

3) The servlet can be invoked through server side include tags.

Any file ending with .shtml is a server-parsed file. In .shtml, if a servlet tag or server-insert tag is present, the server will run the servlet and insert the output generated by the servlet in the place of the insert tag.

4) The servlet can be invoked by placing it in the servlets directory.

Servlets can be dropped in to the servlets directory relative to the root of the server. Servlets placed in this directory can be invoked using their class names. In this example, if HelloWorld.class was placed in the server_root/servlets directory:

`http://server_host_name:800/servlet/HelloWorld`

A class_file.initArgs file can be placed in the same directory for passing init args to the servlet in question. For example, to pass two initial arguments to the HelloWorld servlet, a HelloWorld.initArgs file must be created in the extensions directory. The syntax would be:

A=B
C=B

When servlets are recompiled in this directory, the new version will be automatically loaded by the server.

5) The servlet can be invoked by using it in a filter chain.

This happens when the servlet's configuration specifies that the servlet should be invoked when a particular MIME type is set as the response. See the Filters and Servlet Chaining section for more on this topic.

CHAPTER 10 - JAVA SERVER PAGES

JavaServer(TM) Pages is a simple, yet powerful technology for creating and maintaining dynamic-content web pages. Based on the Java programming language, JavaServer Pages offers proven portability, open standards, and a mature re-usable component model.

The JavaServer Pages architecture enables the separation of content generation from content presentation. This separation not only eases maintenance headaches, it also allows webteam members to focus on their areas of expertise. Now, webpage designers can concentrate on layout, and web application designers on programming, with minimal concern about impacting each other's work.

10.1 PORTABILITY

JavaServer Pages files can be run on any web server or web-enabled application server that provides support for them. Dubbed the JSP engine, this support involves recognition, translation, and management of the JavaServer Page lifecycle and its interactions with associated components.

The JSP engine for a particular server might be built-in or might be provided through a 3rd-party add-on. As long as the server on which you plan to execute the JavaServer Pages supports the same specification level as that to which the file was written, no changes should be necessary as you move your files from server to server. Note, however, that instructions for the setup and configuration of the files may differ between files.

To date, there has been no upwards- or backwards-compatibility between JavaServer Pages specifications. A JavaServer Pages file written to the 0.92 specification can be run only on a server supporting JavaServer Pages 0.92. The same file could not run on a server supporting only JavaServer Pages 1.0 or JavaServer Pages 0.91.

10.2 COMPOSITION

It was mentioned earlier that the JavaServer Pages architecture can include reusable Java components. The architecture also allows for the embedding of a scripting language directly into the JavaServer Pages file.

The components currently supported include JavaBeans, and Servlets. Support for Enterprise Java Beans components will likely be added in a future release. As the default scripting language, JavaServer Pages use the Java programming language. This means that scripting on the server side can take advantage of the full set of capabilities that the Java programming language offers. Support for other scripting languages might become available in the future.

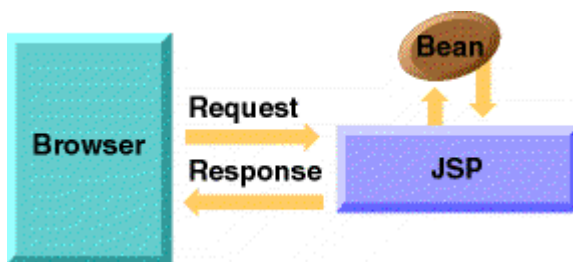
10.3 PROCESSING

A JavaServer Pages file is essentially an HTML document with JSP scripting or tags. It may have associated components in the form of .class, .jar, or .ser files--or it may not. The use of components is not required.

The JavaServer Pages file has a .jsp extension to identify it to the server as a JavaServer Pages file. Before the page is served, the JavaServer Pages syntax is parsed and processed into a servlet on the server side. The servlet that is generated outputs real content in straight HTML for responding to the client. Because it is standard HTML, the dynamically generated response looks no different to the client browser than a static response.

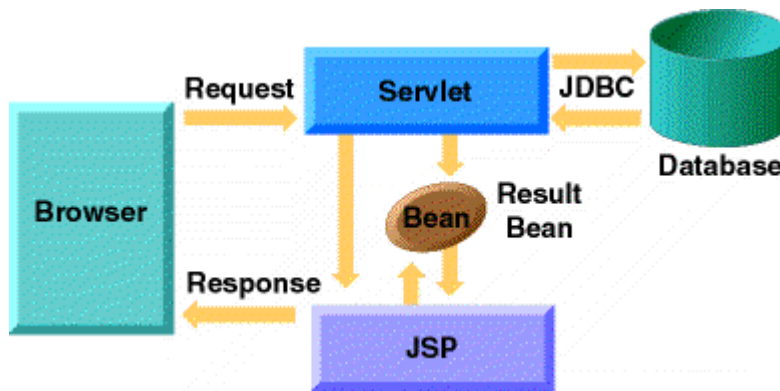
10.4 ACCESS MODELS

A JavaServer Pages file may be accessed in at least two different ways:
A client request comes directly into a JavaServer Page.



In this scenario, suppose the page accesses reusable JavaBean components that perform particular well-defined computations like accessing a database. The result of the Bean's computations, called result sets are stored within the Bean as properties. The page uses such Beans to generate dynamic content and present it back to the client.

A request comes through a servlet.



The servlet generates the dynamic content. To handle the response to the client, the servlet creates a Bean and stores the dynamic content (sometimes called the result set) in the Bean. The servlet then invokes a JavaServer Page that will present the content along with the Bean containing the generated from the servlet.

There are two APIs to support this model of request processing using JavaServer Pages. One API facilitates passing context between the invoking servlet and the JavaServer Page. The other API lets the invoking servlet specify which JavaServer Page to use.

In both of the above cases, the page could also contain any valid Java code. The JavaServer Pages architecture encourages separation of content from presentation--it does not mandate it.

10.4.1 How to Choose Between Access Models

With at least two access models, the question naturally arises "When does it make sense to have a JavaServer Page as the front-end to a servlet, as the back-end to a servlet, or use only the servlet? Here are some possible guidelines:

- If a graphical interface (GUI) is necessary to collect the request data--use a JavaServer Pages file.
- If the request and request parameters are otherwise available to the servlet, but the results of the servlet processing requires a graphical interface to present them--use a JavaServer Pages file.
- If presentation layout is minimal (will not require very many println lines in your servlet code) and you don't need to make that presentation logic available to a customer or your webpage designer, then a Servlet might suffice.

CHAPTER 11 - WRITING YOUR FIRST JAVA SERVER PAGE

The **JavaServer Pages Overview** discussed how the **JavaServer Pages (JSP)** technology eases the development and maintenance of webpages containing dynamic content. In the **Java Web Server**, **JavaServer Pages** are compiled to **servlets**. As a result, they have all the benefits associated with **Java servlets**: platform independence, standard API, network capabilities (can be used with **RMI** or **JDBC**), faster than **CGI**, and reusable. In addition, **JavaServer Pages** have the following benefits: they allow separation of content generation (logic) from content presentation and they leverage reusable components (such as **JavaBeans** components).

Like **Servlets**, **JavaServer Pages** can be as simple or as complex as your needs require-- a **Hello World** demonstration or a multi-tier web application. This document discusses the following minimum steps needed to create a **JavaServer Page**.

1. Write the **JSP** file.
 - Declare any **JavaBeans** components.
 - Use tag-centric syntax to access Bean properties or scripting-centric syntax to provide desired functionality.
 - Save the file with a **.jsp** filename extension.
2. Place the **JSP** file and any associated files.
 - Place the **.jsp** files under your doc root directory (**/public_html** by default).
 - Place associated **.class**, **.jar**, or **.ser** files in a directory on your **CLASSPATH** (**/classes** by default).
3. Test the **JSP** file.
 - Invoke the **JSP** file from a web browser.

Each of these steps in its simplest form is discussed below.

JavaServer Pages samples, complete with working code, are provided with the product. For a listing of samples, see the **Java Web Server Samples** online document.

11.1 WRITE THE JSP FILE

The **JavaServer Pages** technology allows for the use of **Java** scriptlets and the use of **JavaServer Pages** tags to invoke **Java** components, but it does not require either. The possible permutations include:

Only **HTML**

A file containing nothing but standard HTML code can be renamed with an filename extension of .jsp and therefore meet the minimum requirements to be invoked as a JavaServer Page. Such a file would still be parsed and compiled to a servlet. The servlet would still return a response when the page was invoked--its response to the client would simply contain the original HTML.

Here's an example of such a JSP file:

```
<html>
<head>
<title>HTML-only JSP File</title>
</head>
<body>
<h1>Hello World (HTML)</h1>
</body>
</html>
```

HTML and component-centric tags

Of course, the whole point of JavaServer Pages technology is to simply and easily manage dynamic content on the server side. So, if you take your plain HTML file from above, you could add some JavaServer Pages tags to interact with Java components. Your component could be as simple as a Bean which returns the current time. Note that no Java scriptlets are required.

```
<html>
<head>
<title>HTML plus Bean JSP File</title>
</head>
<jsp:useBean id="clock" scope="page" class="sunexamples.beans.JspCalendar"
type="sunexamples.beans.JspCalendar" />
<body>
<h1>Hello World (HTML)</h1>
<p>Today is: <jsp:getProperty name="clock" property="date"/></p>
</body>
</html>
```

HTML and script-centric tags

Although JavaServer Pages architecture encourages the use of componentization for ease of maintenance and reusability, you are not required to use components. This example uses a simple scriptlet containing raw Java code. This usage of the JavaServer Pages tags is reminiscent of earlier dynamic content generation such as Page Compilation.

```
<HTML>
<HEAD>
<TITLE>HTML and scriptlet JSP File</TITLE>
</HEAD>
<BODY>
```

```

<H1>
Hello World !
<P>
<%
out.println("The time now is: " + new java.util.Date());
%>
</H1>
</BODY>
</HTML>

```

HTML and component-centric tag and script-centered tag

Here's the same example with the substitution of 2 lines of code for the original Hello line. By checking for a name parameter in the incoming request (available in the automatic request object), the page can return either a generic or a personalized greeting.

```

<html>
<head>
<title>HTML and Bean and scriptlet JSP File</title>
</head>
<jsp:useBean id="clock" scope="page" class="sunexamples.beans.JspCalendar"
type="sunexamples.beans.JspCalendar" />
<body>
<h1>
<%
if (request.getParameter("name") == null) {
    out.println("Hello World");
} else {
    out.println("Hello" + request.getParameter("name"));
}
%>
</h1>
<p>Today is: <jsp:getProperty name="clock" property="date"/></p>
</body>
</html>

```

Remember to save the file with a .jsp filename extension. This tells the webserver that the file is a JavaServer Pages file and to process it accordingly.

11.2 PLACE THE JSP FILE AND ASSOCIATED FILES

Placing the various parts of a JSP application is straight forward if you follow these rules:

JSP file

Think of the JSP file as an HTML page (which it is when the client receives it). This makes it obvious where the file should be placed--in the document root of the webserver. By default, the document root is:

`server_root/public_html/`

where `server_root` is the location of the installed Java Web Server.

If the JSP file is located outside the Java Web Server's docroot directory (remote), do the following:

On UNIX, create a link from within your document root of the Java Web Server to the .jsp file(s) in the remote directory.

On any platform, create a file alias using the Java Web Server's Administration Tool. For information on setting up file aliases, see the Accessing Files Through Aliases online document.

Note that file aliases are set on a per-service basis. That means if you want the file alias to work on both the Web Page Service and the Secure Web Service, you must set it up separately under each of these services.

Associated JavaBeans components, .class files, and servlets

Locate these files as you always do, the fact that they are being used by a JavaServer Pages file makes no difference to their location.

JavaBeans components: The standard rule is that these components--in .class, .jar, or .ser format--should be placed in a directory on your CLASSPATH.

Remember when adding the location of .jar files to your CLASSPATH, that you must include the actual name of the file, not simply the directory containing the.jar file.

Servlets: The standard rule is that servlets should be placed in a directory on your classpath if they are not to be reloaded automatically when they are changed, or located in the `servlets/` of the Java Web Server if they are to be automatically recompiled and loaded when they are changed.

If the JavaBeans components, .class files, or servlet files associated with the .jsp file are remote to the Java Web Server, you can do one of the following:

Set your CLASSPATH to include the remote location

Use the `-cp` option of your startup command (typically `httpd`) to specify the location of the supporting classes. This will temporarily set the CLASSPATH. If the server is shutdown or restarted, this temporary setting will disappear and need to be reset if still desired.

To be sure that the CLASSPATH has been successfully set, use the Snoop servlet example to see what the CLASSPATH has been set to.

11.3 TEST THE JSP FILE

With the Java Web Server running, type in the URL to the file. This will be relative to the Java Web Server's document root.

`http://server_name:doc port/path_to_jsp_file`

For example, if myfirst.jsp file was located in the default document root (public_html) and on the default document port (8080), you would type:

`http://schnauzer.eng:8080/myfirst.jsp`

JavaServer Pages 1.0 Syntax Summary	
Tag-centric Syntax	
<u><jsp:useBean ≥</u>	Locates or instantiates a Bean with a specific name and scope.
<u><jsp:setProper ty></u>	Sets a property value or values in a Bean.
<u><jsp:getPrope rty></u>	Gets the value of a Bean property so that you can display it in a JSP page.
<u><jsp:include></u>	Sends a request to an object and includes the result in a JSP file.
<u><jsp:forward></u>	Forwards a client request to an HTML file, JSP file, or servlet for processing.
<u><jsp:plugin></u>	Downloads a Java plugin to the client Web browser to execute an applet or Bean.
<u><!-- output comment --></u>	Generates a comment that can be viewed in the HTML source file.
Script-centric Syntax	
<u><%@ directive %></u>	<u>page:</u> Defines attributes that apply to an entire JSP page.
	<u>include:</u> Includes a file of text or code when the JSP page is translated.
	<u>taglib:</u> (Not Supported) Defines a tag library and prefix for the custom tags used in the JSP page.
<u><%! declaration %></u>	Declares a variable or method valid in the scripting language used in the page.
<u><%=</u>	Contains an expression valid in the scripting language used in

<u>expression %></u>	the page.	
<u><% scriptlet %></u>	Contains a code fragment valid in the scripting language used in the page.	
<u><%-- page comment --%></u>	Documents the JSP page but is not sent to the client.	
Implicit Objects		
request	Subclass of <i>javax.servlet.HttpServletRequest</i>	Request scope.
response	Subclass of <i>javax.servlet.HttpServletResponse</i>	Page scope.
session	<i>javax.servlet.http.HttpSession</i>	Session scope.
application	<i>javax.servlet.ServletContext</i>	Application scope.
config	<i>javax.servlet.ServletConfig</i>	Page scope.
out	<i>javax.servlet.jsp.JspWriter</i>	Page scope.
pageContext	<i>javax.servlet.jsp.PageContext</i>	Page scope.

CHAPTER 12 - FREQUENTLY ASKED QUESTIONS

12.1 ABOUT SERVLETS

The Servlets FAQ answers the following questions:

- How do I pass arguments to the servlets?
- How long do servlets last?
- What servlets are currently part of the server?
- Why can't I compile a servlet?
- How do I set multiple properties?
- What is passed as ServletRequest for servlets invoked as server-side include?

12.1.1 How do I pass arguments to the servlets?

Arguments to the servlets can be passed at two levels.

When a client is invoking the servlet, the client can pass the arguments as part of a URL in the form of name/value pairs in the query string portion of the URL. If a tag is used in the HTML language to invoke the servlet, the arguments can be passed through the param name construct:

```
<servlet code="servlet_name" stock_symbol="SUNW">
```

The server administrator/webmaster can pass arguments to the servlet at loading or initialization time by specifying the arguments as part of the server configuration.

12.1.2 How long do servlets last?

Once activated, servlets live until the server exits or until the server calls the destroy method on the servlet. For servlets associated with servlet aliases, unmapping the alias will destroy the servlet, and subsequent access to the servlet will reload the servlet.

12.1.3 What servlets are currently part of the server?

See the **Sample Servlets** page for information on some of the available servlets. Also look in the `server_root/examples` directory of the release for more servlets.

12.1.4 Why can't I compile a servlet?

Make sure that your **CLASSPATH** is set to include the Java Web Server classes. This can be done at the command line (using `java -classpath`) or else in the environment variable.

12.1.5 How do I set multiple properties?

The following is the proper syntax:

```
test.Servlet.initArgs=\nfruit=apple,qty=5
```

12.1.6 What is passed as ServletRequest for servlets invoked as server-side include?

SSIIncludeServlet will clone its ServletRequest, and substitute the parameters embedded in the tag.

12.2 ABOUT JAVA SERVER PAGES

- What are "JavaServer Pages"?
- Why would I want to use them?
- How do I implement them?
- Where can I run them?
- What version of JavaServer Pages does the Java Web Server support?

12.2.1 What are "JavaServer Pages"?

A powerful and easy way to create dynamic web pages, they allow you to cleanly separate dynamically generated content from the presentation of that content. Dynamic content is created by embedding statements (scripts) or re-usable components (such as JavaBeans) directly into your HTML. The default scripting language used by JavaServer Pages is the powerful Java language. (Currently this is the only scripting language supported. Support for other scripting languages might be added in the future.)

12.2.2 Why would I want to use them?

The ability to focus on presentation separately from content.

This allow the tasks of webpage design and content generation to be worked on by different people simultaneously or the same person sequentially.

The opportunity to employ re-usable components.

Within the content generation task, content can be generated by re-usable components such as JavaBeans components. So now the business logic or database access you created for one set of web pages can easily be reused.

The ease of making changes.

Compilation of the Java/HTML page is automatic the first time the page is invoked. If you make changes, it recompiles automatically, otherwise the server continues to use the cached copy providing remarkably fast access.

12.2.3 How do I implement them?

Web Page Design

Determine the nature of the information to be displayed

Create a page design in HTML to display that information (You and the Dynamic Content developer will need to agree on the names and tags you will use to access the data. For instance, by using a jsp:usebean tag and calling the Bean properties by name.)

Save the file with an .jsp extension.

Place the file in your document root directory. (By default, this is the public_html subdirectory of the webserver root.)

Dynamic Content Implementation

Determine how the information to be generated will be created

Write any valid Java code (including SQL calls) or use special tags to declare and access your own (or third-party) JavaBeans components.

If using your own components, write them.

Place any classes or components that need to be accessed into a directory in your CLASSPATH. (The default CLASSPATH when you install the Java Web Server(TM) includes the /classes subdirectory of the webserver root.)

12.2.4 Where can I run them?

JavaServer Pages are cross-platform.

Thanks to the power of the Java programming language, the Java Web Server 2.0 runs on any platform that is at least JDK 1.1.7-compliant. That means, whether your server platform is Windows NT or Solaris or something else, Java Web Server 2.0 can be used to serve pages created using JavaServer Pages technology.

JavaServer Pages can be cross-server.

JavaServer Pages can be run on any web server which implements support for them. Like servlet support, JavaServer Pages support is expected to be available on other servers directly or through the use of 3rd-party plug-ins.

12.2.5 What version of JavaServer Pages does the Java Web Server support?

The Beta 1 version of Java Web Server 2.0 supports the JavaServer Pages Specification 0.92.

The Beta 2 and Preview version of Java Web Server 2.0 support the JavaServer Pages Specification 1.0 (Public Draft 1).

The final release of Java Web Server 2.0 will support the JavaServer Pages Specification 1.0 (final).