THE EXPERT'S VOICE® IN JAVA

JavaServer Faces

Introduction by Example

Josh Juneau

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

CHAPTER 1

Introduction to Servlets

Java servlets were the first technology for producing dynamic Java web applications. Sun Microsystems released the first Java Servlet specification in 1997. Since then it has undergone tremendous change, making it more powerful and easing development more with each release. The 3.0 version was released as part of Java EE 6 in December 2009. Although not always used directly by Java web developers, servlets are at the base of all Java EE applications. Many developers use servlet frameworks such as Java Server Pages (JSP) and Java Server Faces (JSF), both of those technologies compile pages into Java servlets behind the scenes via the servlet container. That said, a fundamental knowledge of Java servlet technology is very useful for any Java web developer.

Servlets are Java classes that conform to the Java Servlet API, which allows a Java class to respond to requests. Although servlets can respond to any type of request, they are most commonly written to respond to HTTP requests. A servlet must be deployed to a Java servlet container in order to become usable. The Servlet API provides a number of objects that are used to enable the functionality of a servlet within a web container. Such objects include the request and response objects, `pageContext`, and a great deal of others, and when these objects are used properly, they enable a Java servlet to perform just about any task a web-based application needs to perform.

As mentioned, servlets can produce not only static content but also dynamic content. Since a servlet is written in Java, any valid Java code can be used within the body of the servlet class. This empowers Java servlets and allows them to interact with other Java classes, the web container, the underlying file server, and much more.

This chapter will get you started developing and deploying servlets, and provide you with foundational knowledge to move forward with other servlet-based web frameworks In this chapter, you will learn how to install Oracle's GlassFish application server, a robust servlet container, which will enable you to deploy sophisticated Java enterprise applications. You will be taught the basics of developing servlets, how to use them with client web sessions, and how to link a servlet to another application. All the while, you will learn to use standards from the latest release of the Java Servlet API (3.2), which modernizes servlet development and makes it much easier and more productive than in years past.

Note You can run the examples within this chapter by deploying the `JSFByExample.war` file (contained in the sources) to a local Java EE application server container such as GlassFish v4.x. You can also set up the NetBeans 8.x project entitled JSFByExample that is contained in the sources, build it, and deploy to GlassFish v4.x. Otherwise, you can run the examples in Chapter 1 stand-alone using the instructions provided in the section "Packaging, Compiling, and Deploying a Servlet". If you deploy the `JSFByExample.war` file to a Java EE application server container, you can visit the following URL to load the examples for this chapter: `http://localhost:8080/JSFByExample/faces/chapter01/index.xhtml`.

Setting Up a Java Enterprise Environment

You'll need an environment in which to experiment with servlets, and then later with JavaServer Faces. Oracle's GlassFish application server is a good choice, as it is the Java EE 7 Reference Impementation. It's easy to set up, and the following example will get you started and ready to run all the subsequent examples in the book.

Example

To get started, ownload and install Oracle's GlassFish application server from the GlassFish web site. The version used for this book is the open source edition, release 4.1, and it can be downloaded from `http://glassfish.java.net/` in the "Download" section. Select the `.zip` or `.tar.gz` download format, and decompress the downloaded files within a directory on your workstation. I will refer to that directory as `/JAVA_DEV/GlassFish`. The GlassFish distribution comes prepackaged with a domain so that developers can get up and running quickly. Once the `.zip` file has been unpacked, you can start the domain by opening a command prompt or terminal and starting GlassFish using the following statement:

```
/PATH_TO_GLASSFISH /GlassFish/bin/asadmin start-domain domain1
```

The domain will start, and it will be ready for use. You will see output from the server that looks similar to the following:

```
Waiting for domain1 to start .....
Successfully started the domain : domain1
domain  Location: /PATH_TO_GLASSFISH/glassfish/domains/domain1
Log File: /PATH_TO_GLASSFISH/glassfish/domains/domain1/logs/server.log
Admin Port: 4848
Command start-domain executed successfully.
```

Explanation

The development of Java EE applications begins with a Java EE–compliant application server. A Java EE–compliant server contains all the essential components to provide a robust environment for deploying and hosting enterprise Java applications. The GlassFish application server is the industry standard for Java EE 7. As of GlassFish 4.0, there is only an open sourced distribution of the server available, meaning that it is not possible to purchase Oracle support for GlassFish. However, in a production environment, you may want to consider purchasing GlassFish 4.x support from a third-party organization so that technical support will be available if needed. An alternative is to utilize a commercially supported server that is Java EE 7 compliant, such as Oracle WebLogic 12.1.x.

Installing GlassFish is easy. It consists of downloading an archive and uncompressing it on your development machine. Once you've completed this, the application server will make use of your locally installed Java development kit (JDK) when it is started. JDK 8 is supported for use with GlassFish as of release 4.1. For GlassFish 4.0, please use JDK 7. Once the server starts, you can open a browser and go to `http://localhost:4848` to gain access to the GlassFish administrative console. Most Java EE developers who deploy on GlassFish use the administrative console often. The administrative console provides developers with the tools needed to deploy web applications, register databases with Java Naming and Directory Interface (JNDI), set up security realms for a domain, and do much more. You should take some time to become familiar with the administrative console because the more you know about it, the easier it will be to maintain your Java EE environment.

Installing the GlassFish application server is the first step toward developing Java applications for the enterprise. While other applications servers such as JBoss WildFly, Apache TomEE, and WebLogic are very well adopted, GlassFish offers developers a solid environment that is suitable for production use and easy to learn. It also has the bonus of being an open source application server and the reference implementation for Java EE 7.

Developing Your First Servlet

Web applications are based upon a series of web views or pages. There is often a requirement to develop a view that has the ability to include content that may change at any given time. For instance, you may be developing a view that contains stock data, and you may wish to have that data updated often. Servlets provide the ability to produce dynamic content, allowing server-side computations and processes to update the data in the servlet at will.

Example

Develop a Java servlet class, and compile it to run within a Java servlet container. In this example, a simple servlet is created that will display some dynamic content to the web page. The The following code is the servlet code that contains the functionality for the servlet:`package org.javaserverfaces.chapter01;`

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Simple Dynamic Servlet
 * @author juneau
 */
public class SimpleServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP
     * <code>GET</code> and
     * <code>POST</code> methods.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            /*
             * TODO output your page here. You may use following sample code.
             */
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet SimpleServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h2>Servlet SimpleServlet at " + request.getContextPath() + "</h2>");
            out.println("<br/>Welcome to JavaServer Faces: Introduction By Example!");
```

```
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the
left to edit the code.">
    /**
     * Handles the HTTP
     * <code>GET</code> method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Handles the HTTP
     * <code>POST</code> method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Returns a short description of the servlet.
     *
     * @return a String containing servlet description
     */
    @Override
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>
}
```

The following code is the web deployment descriptor. This file is required for application deployment to a servlet container. It contains the servlet configuration and mapping that maps the servlet to a URL. Later in this chapter, will learn how to omit the servlet configuration and mapping from the `web.xml` file to make servlet development, deployment, and maintenance easier.

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <servlet>
        <servlet-name>SimpleServlet</servlet-name>
        <servlet-class>org.javaeeexamples.chapter1.example01_02.SimpleServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SimpleServlet</servlet-name>
        <url-pattern>/SimpleServlet</url-pattern>
    </servlet-mapping>
              <welcome-file-list>
        <welcome-file> /SimpleServlet </welcome-file>
    </welcome-file-list>
</web-app>
```

■ **Note** Many web applications use a page named `index.html` or `index.xhtml` as their welcome file. There is nothing wrong with doing that, and as a matter of fact, it is the correct thing to do. The use of `/SimpleServlet` as the welcome file in this example is to make it easier to follow for demonstration purposes.

To compile the Java servlet, use the `javac` command-line utility. The following line was excerpted from the command line, and it compiles the `SimpleServlet.java` file into a class file. First, traverse into the directory containing the `SimpleServlet.java` file; then, execute the following:

```
javac -cp /JAVA_DEV/GlassFish/glassfish/modules/javax.servlet-api.jar SimpleServlet.java
```

Once the servlet code has been compiled into a Java class file, it is ready to package for deployment.

■ **Note** You may want to consider installing a Java integrated development environment (IDE) to increase your development productivity. There are several very good IDEs available to developers, so be sure to choose one that contains the features you find most important and useful for development. As the author of this book on Java EE 7, I recommend installing NetBeans 8.x or newer for development. NetBeans is an open source IDE that is maintained by Oracle, and it includes support for all the cutting-edge features that the Java industry has to offer, including EJB development with Java EE 7, JavaFX 8 support, and more.

Explanation

Java servlets provide developers with the flexibility to design applications using a request-response programming model. Servlets play a key role in the development of service-oriented and web application development on the Java platform. Different types of servlets can be created, and each of them is geared toward providing different functionality. The first type is the `GenericServlet`, which provides services and functionality. The second type, `HttpServlet`, is a subclass of `GenericServlet`, and servlets of this type provide functionality and a response that uses HTTP. The solution to this example demonstrates the latter type of servlet because it displays a result for the user to see within a web browser.

Servlets conform to a life cycle for processing requests and posting results. First, the Java servlet container calls the servlet's constructor. The constructor of every servlet must take no arguments. Next, the container calls the servlet `init` method, which is responsible for initializing the servlet. Once the servlet has been initialized, it is ready for use. At that point, the servlet can begin processing. Each servlet contains a `service` method, which handles the requests being made and dispatches them to the appropriate methods for request handling. Implementing the `service` method is optional. Finally, the container calls the servlet's `destroy` method, which takes care of finalizing the servlet and taking it out of service.

Every servlet class must implement the `javax.servlet.Servlet` interface or extend another class that does. In the solution to this example, the servlet named `SimpleServlet` extends the `HttpServlet` class, which provides methods for handling HTTP processes. In this scenario, a browser client request is sent from the container to the servlet; then the servlet service method dispatches the `HttpServletRequest` object to the appropriate method provided by `HttpServlet`. Namely, the `HttpServlet` class provides the `doGet`, `doPut`, `doPost`, and `doDelete` methods for working with an HTTP request. The HttpServlet class is abstract, so it must be subclassed, and then an implementation can be provided for its methods. In the solution to this example, the `doGet` method is implemented, and the responsibility of processing is passed to the `processRequest` method, which writes a response to the browser using the `PrintWriter`. Table 1-1 describes each of the methods available to an `HttpServlet`.

Table 1-1. *`HttpServlet` Methods*

Method Name	Description
`doGet`	Used to process HTTP `GET` requests. Input sent to the servlet must be included in the URL address. For example: `?myName=Josh&myBook=JSF`.
`doPost`	Used to process HTTP `POST` requests. Input can be sent to the servlet within HTML form fields.
`doPut`	Used to process HTTP `PUT` requests.
`doDelete`	Used to process HTTP `DELETE` requests.
`doHead`	Used to process HTTP `HEAD` requests.
`doOptions`	Called by the container to allow `OPTIONS` request handling.
`doTrace`	Called by the container to handle `TRACE` requests.
`getLastModified`	Returns the time that the `HttpServletRequest` object was last modified.
`init`	Initializes the servlet.
`destroy`	Finalizes the servlet.
`getServletInfo`	Provides information regarding the servlet.

A servlet generally performs some processing within the implementation of its methods and then returns a response to the client. The `HttpServletRequest` object can be used to process arguments that are sent via the request. For instance, if an HTML form contains some input fields that are sent to the server, those fields would be contained within the `HttpServletRequest` object. The `HttpServletResponse` object is used to send responses to the client browser. Both the `doGet` and `doPost` methods within a servlet accept the same arguments, namely, the `HttpServletRequest` and `HttpServletResponse` objects.

■ **Note** The `doGet` method is used to intercept HTTP `GET` requests, and `doPost` is used to intercept HTTP `POST` requests. Generally, the `doGet` method is used to prepare a request before displaying for a client, and the `doPost` method is used to process a request and gather information from an HTML form.

In the solution to this example, both the `doGet` and `doPost` methods pass the `HttpServletRequest` and `HttpServletResponse` objects to the `processRequest` method for further processing. The `HttpServletResponse` object is used to set the content type of the response and to obtain a handle on the `PrintWriter` object in the `processRequest` method. The following lines of code show how this is done, assuming that the identifier referencing the `HttpServletResponse` object is `response`:

```
response.setContentType("text/html;charset=UTF-8");
PrintWriter out = response.getWriter();
```

A `GenericServlet` can be used for providing services to web applications. This type of servlet is oftentimes used for logging events because it implements the `log` method. A `GenericServlet` implements both the Servlet and `ServletConfig` interfaces, and to write a generic servlet, only the `service` method must be overridden.

How to Package, Compile, and Deploy a Servlet

Once a servlet has been developed (and compiled), it needs to be deployed to a servlet container before it can be used. After deployment to the server, the servlet needs to be mapped to a URL for invocation.

Example

Compile the sources, set up a deployable application, and copy the contents into the GlassFish deployment directory. From the command line, use the `javac` command to compile the sources.

```
javac -cp /PATH_TO_GLASSFISH/GlassFish/glassfish/modules/javax.servlet-api.jar SimpleServlet.java
```

After the class has been compiled, deploy it along with the `web.xml` deployment descriptor, conforming to the appropriate directory structure. In web.xml, declare the servlet, and map it to a URL using the following format:

```
<servlet>
    <servlet-name>SimpleServlet</servlet-name>
    <servlet-class>org.javaserverfaces.chapter01.SimpleServlet</servlet-class>
</servlet>
</servlet-mapping>
    <servlet-mapping>
    <servlet-name>SimpleServlet</servlet-name>
    <url-pattern>/SimpleServlet</url-pattern>
</servlet-mapping>
```

QUICK START FOR DEPLOYING WITHOUT AN IDE

To quickly get started with packaging, compiling, and deploying the example application for the servlet examples in this chapter on GlassFish or other servlet containers such as Apache Tomcat without an IDE, follow these steps:

1. Create a single application named `SimpleServlet` by making a directory named `SimpleServlet`.

2. Create a directory at the root of the application, and name it WEB-INF. Create an XML file in the new WEB-INF directory, and name it web.xml. In the web.xml, add the following markup:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/
xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
    <servlet>
        <servlet-name>SimpleServlet</servlet-name>
        <servlet-class>org.javaserverfaces.chapter01.SimpleServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SimpleServlet</servlet-name>
        <url-pattern>/SimpleServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

3. Create “classes”, and “lib” drectories inside the directory that was created in step 2. Drag the Chapter 1 sources into the WEB-INF/classes directory.

4. Set your CLASSSPATH to include any necessary JAR files. For this chapter, the JavaMail API JAR (`mail.jar`) is required. Place it into the `WEB-INF/lib directory` and set your CLASSPATH accordingly.

5. At the command prompt, change directories so that you are within the “classes” directory that was created in Step 3. Compile each class within the `org.javaserverfaces.chapter01` directory with the following command:

```
javac org\javaserverfaces\chapter01\*.java
```

6. Copy your SimpleServlet application into the `/JAVA_DEV/GlassFish/glassfish/domains/domain1/autodeploy` directory for GlassFish, or the `/Tomcat/webapps` directory for Tomcat.

Test the application by launching a browser and going to `http://localhost:8080/SimpleServlet/servlet_name`, where `servlet_name` corresponds to the servlet name in each example. If using Tomcat, you may need to restart the server in order for the application to deploy.

Explanation

To compile the sources, you can use your favorite Java IDE such as NetBeans or Eclipse, or you can use the command line. For the purposes of this example, I will use the command line. Note that in many of the remaining examples for this book, the NetBeans IDE is used. If you're using the command line, you must ensure you are using the `javac` command that is associated with the same Java release that you will be using to run your servlet container. In this example we will assume that GlassFish 4.1 is being used with JDK 7, and therefore assume that the location of the Java SE 7 installation is at the following path:

```
/Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/Home
```

This path may differ in your environment if you are using a different operating system and/or installation location. To ensure you are using the Java runtime that is located at this path, set the `JAVA_HOME` environment variable equal to this path. On OS X and *nix operating systems, you can set the environment variable by opening the terminal and typing the following:

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/Home
```

If you are using Windows, use the `SET` command within the command line to set up the `JAVA_HOME` environment variable.

```
set JAVA_HOME=C:\your-java-se-path\
```

Next, compile your Java servlet sources, and be sure to include the `javax.servlet-api.jar` file that is packaged with your servlet container (use `servlet-api.jar` for Tomcat) in your `CLASSPATH`. You can set the `CLASSPATH` by using the `-cp` flag of the `javac` command. The following command should be executed at the command line from within the same directory that contains the sources. In this case, the source file is named `SimpleServlet.java`.

```
javac -cp /path_to_jar/javax.servlet-api.jar SimpleServlet.java
```

Next, package your application by creating a directory and naming it after your application. In this case, create a directory and name it `SimpleServlet`. Within that directory, create another directory named `WEB-INF`. Traverse into the `WEB-INF` directory, and create another directory named `classes`. Lastly, create directories within the `classes` directory in order to replicate your Java servlet package structure. For this example, the `SimpleServlet.java` class resides within the Java package `org.javaserverfaces.chapter01`, so create a directory for each of those packages within the `classes` directory. Create another directory within `WEB-INF` and name it `lib`; any JAR files containing external libraries should be placed within the `lib` directory. In the end, your directory structure should resemble the following:

```
SimpleServlet
|_WEB-INF
        |_classes
              |_org
                    |_javaserverfaces
                                |_chapter01

        |_lib
```

Place your `web.xml` deployment descriptor within the `WEB-INF` directory, and place the compiled `SimpleServlet.class` file within the `chapter01` directory. The entire contents of the `SimpleServlet` directory can now be copied within the deployment directory for your application server container to deploy the application. Restart the application server if using Tomcat, and visit the URL `http://localhost:8080/SimpleServlet/SimpleServlet` to see the servlet in action.

Registering Servlets Without WEB-XML

Registering servlets in the `web.xml` file is cumbersome. With the later releases of the Servlet specification, it is possible to deploy servlets without the requirement for a `web.xml` file. In this section, we will take a look at how to register servlets without the `web.xml` requirement.

Example

Use the `@WebServlet` annotation to register the servlet, and omit the `web.xml` registration. This will alleviate the need to modify the `web.xml` file each time a servlet is added to your application. The following adaptation of the `SimpleServlet` class that was used in the previous example includes the `@WebServlet` annotation and demonstrates its use:

```
package org.javaserverfaces.chapter01;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Registering Servlets without WEB-XML
 * @author juneau
 */
@WebServlet(name = "SimpleServletNoDescriptor", urlPatterns = {"/SimpleServletNoDescriptor"})
public class SimpleServletNoDescriptor extends HttpServlet {

    /**
     * Processes requests for both HTTP
     * <code>GET</code> and
     * <code>POST</code> methods.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            /*
             * TODO output your page here. You may use following sample code.
             */
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet SimpleServlet</title>");
```

```
            out.println("</head>");
            out.println("<body>");
            out.println("<h2>Servlet SimpleServlet at " + request.getContextPath() + "</h2>");
            out.println("<br/>Look ma, no WEB-XML!");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    /**
     * Handles the HTTP <code>GET</code> method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Handles the HTTP <code>POST</code> method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

}
```

In the end, the servlet will be accessible via a URL in the same way that it would if the servlet were registered within `web.xml`.

Explanation

There are a couple of ways to register servlets with a web container. The first way is to register them using the `web.xml` deployment descriptor, as demonstrated earlier in the chapter. The second way to register them is to use the `@WebServlet` annotation. The Servlet 3.0 API introduced the `@WebServlet` annotation, which provides an easier technique to use for mapping a servlet to a URL. The `@WebServlet` annotation is placed before the declaration of a class, and it accepts the elements listed in Table 1-2.

Table 1-2. *`@WebServlet` Annotation Elements*

Element	Description
`description`	Description of the servlet
`displayName`	The display name of the servlet
`initParams`	Accepts list of `@WebInitParam` annotations
`largeIcon`	The large icon of the servlet
`loadOnStartup`	Load on start-up order of the servlet
`name`	Servlet name
`smallIcon`	The small icon of the servlet
`urlPatterns`	URL patterns that invoke the servlet

In the solution to this example, the `@WebServlet` annotation maps the servlet class named `SimpleServletNoDescriptor` to the URL pattern of `/SimpleServletNoDescriptor`, and it also names the servlet `SimpleServletNoDescriptor`.

```
@WebServlet(name="SimpleServletNoDescriptor", urlPatterns={"/SimpleServletNoDescriptor"})
```

The new `@WebServlet` can be used rather than altering the `web.xml` file to register each servlet in an application. This provides ease of development and manageability. However, in some cases, it may make sense to continue using the deployment descriptor for servlet registration, such as if you do not want to recompile sources when a URL pattern changes. If you look at the `web.xml` file used earlier, you can see the following lines of XML, which map the servlet to a given URL and provide a name for the servlet. These lines of XML perform essentially the same function as the `@WebServlet` annotation in this example.

```
<servlet>
    <servlet-name>SimpleServletNoDescriptor</servlet-name>
    <servlet-class>org.javaserverfaces.chapter01.SimpleServletNoDescriptor</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>SimpleServletNoDescriptor</servlet-name>
    <url-pattern>/SimpleServletNoDescriptor</url-pattern>
</servlet-mapping>
```

Displaying Dynamic Content with a Servlet

As mentioned previously in the chapter, it sometimes makes sense to deliver dynamic content (content that changes frequently), rather than serving static content that never changes. In this example, we will take a look at how to develop a servlet that has the ability to display dynamic content.

Example

Define a field within your servlet to contain the dynamic content that is to be displayed. Post the dynamic content on the page by appending the field containing it using the `PrintWriter println` method. The following example servlet declares a `Date` field and updates it with the current `Date` each time the page is loaded:

```
package org.javaserverfaces.chapter01;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Displaying Dynamic Content with a Servlet
 *
 * @author juneau
 */
@WebServlet(name = "CurrentDateAndTime", urlPatterns = {"/CurrentDateAndTime"})
public class CurrentDateAndTime extends HttpServlet {

    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
     * methods.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet CurrentDateAndTime</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet CurrentDateAndTime at " + request.getContextPath() + "</h1>");
            out.println("<br/>");

            Date currDateAndTime = new Date();
            out.println("The current date and time is: " + currDateAndTime);
```

```
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    /**
     * Handles the HTTP <code>GET</code> method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Handles the HTTP <code>POST</code> method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

The resulting output from this servlet will be the current date and time.

Explanation

One of the reasons why Java servlets are so useful is because they allow dynamic content to be displayed on a web page. The content can be taken from the server itself, a database, another web site, or many other web-accessible resources. Servlets are not static web pages; they are dynamic, and that is arguably their biggest strength.

In the solution to this example, a servlet is used to display the current time and date on the server. When the servlet is processed, the `doGet` method is called, which subsequently makes a call to the `processRequest` method, passing the `request` and `response` objects. Therefore, the `processRequest` method is where the bulk of the work occurs. The `processRequest` method creates a `PrintWriter` by calling the `response.getWriter` method, and the `PrintWriter` is used to display content on the resulting web page. Next, the current date and time are obtained from the server by creating a new `Date` and assigning it to the `currDateAndTime` field. Lastly, the `processRequest` method sends the web content through the `out.println` method, and the contents of the `currDateAndTime` field are concatenated to a `String` and sent to `out.println` as well. Each time the servlet is processed, it will display the current date and time at the time in which the servlet is invoked because a new `Date` is created with each request.

This example just scratches the surface of what is possible with a Java servlet. Although displaying the current date and time is trivial, you could alter that logic to display the contents of any field contained within the servlet. Whether it be an `int` field that displays a calculation that was performed by the servlet container or a `String` field containing some information, the possibilities are endless.

Handling Requests and Responses

Most applications allow forms that accept input, and then produce a response. This is one of the main components of an HTTP application, and servlets are ideal for handling a request-response lifecycle. It can also be useful to develop forms in HTML, and have the form submitted to a processing engine, such as a servlet.

Example

To see a request-response example in action, create a standard HTML-based web form, and when the submit button is clicked, invoke a servlet to process the end-user input and post a response. To examine this technique, you will see two different pieces of code. The following code is HTML that is used to generate the input form. Pay particular attention to the `<form>` and `<input>` tags. You will see that the form's `action` parameter lists a servlet name, `MathServlet`.

```
<html>
    <head>
    <title>Simple Math Servlet</title>
    </head>
    <body>
        <h1>This is a simple Math Servlet</h1>
        <form method="POST" action="MathServlet">
            <label for="numa">Enter Number A: </label>
            <input type="text" id="numa" name="numa"/><br><br>
                                <label for="numb">Enter Number B: </label>
                                <input type="text" id="numb" name="numb"/><br/><br/>
            <input type="submit" value="Submit Form"/>
            <input type="reset" value="Reset Form"/>
        </form>
    </body>
</html>
```

Next, take a look at the following code for a servlet named `MathServlet`. This is the Java code that receives the input from the HTML code listed earlier, processes it accordingly, and posts a response.

```
package org.javaserverfaces.chapter01;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;

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

```
/**
 * Handling Requests and Responses
 */
// Uncomment the following line to run example stand-alone
//@WebServlet(name="SessionServlet", urlPatterns={"/MathServlet"})
// The following will allow the example to run within the context of the JSFByExample example
// enterprise application (JSFByExample.war distro or Netbeans Project)
@WebServlet(name = "MathServlet", urlPatterns = {"/chapter01/MathServlet"})
public class MathServlet extends HttpServlet {

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

        res.setContentType("text/html");

        // Store the input parameter values into Strings
        String numA = req.getParameter("numa");
        String numB = req.getParameter("numb");

        PrintWriter out = res.getWriter();
        out.println("<html><head>");
        out.println("<title>Test Math Servlet</title>");
        out.println("\t<style>body { font-family: 'Lucida Grande', "
                + "'Lucida Sans Unicode';font-size: 13px; }</style>");
        out.println("</head>");
        out.println("<body>");

        try {
            int solution = Integer.valueOf(numA) + Integer.valueOf(numB);

            /*
             * Display some response to the user
             */
            out.println("<p>Solution: "
                    + numA + " + " + numB + " = " + solution + "</p>");

        } catch (java.lang.NumberFormatException ex) {
            // Display error if an exception is raised
            out.println("<p>Please use numbers only...try again.</p>");
        }

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

        out.close();
    }
}
```

■ **Note** To run the example, deploy the JSFByExample application to your application server container, and then enter the following address into your browser: `http://localhost:8080/JSFByExample/chapter01/math.html`. This assumes you are using default port numbers for your application server installation. If using the NetBeans project that was packaged with the sources, you do not need to worry about copying the code as everything is pre-configured.

Explanation

Servlets make it easy to create web applications that adhere to a request and response life cycle. They have the ability to provide HTTP responses and also process business logic within the same body of code. The ability to process business logic makes servlets much more powerful than standard HTML code. The solution to this example demonstrates a standard servlet structure for processing requests and sending responses. An HTML web form contains parameters that are sent to a servlet. The servlet then processes those parameters in some fashion and publishes a response that can be seen by the client. In the case of an `HttpServlet` object, the client is a web browser, and the response is a web page.

Values can be obtained from an HTML form by using HTML `<input>` tags embedded within an HTML `<form>`. In the solution to this example, two values are accepted as input, and they are referenced by their `id` attributes as `numa` and `numb`. There are two more `<input>` tags within the form; one of them is used to submit the values to the form `action`, and the other is used to reset the form fields to blank. The form action is the name of the servlet that the form values will be passed to as parameters. In this case, the action is set to `MathServlet`. The `<form>` tag also accepts a form-processing method, either `GET` or `POST`. In the example, the `POST` method is used because form data is being sent to the action; in this case, data is being sent to `MathServlet`. You could, of course, create an HTML form as detailed as you would like and then have that data sent to any servlet in the same manner. This example is relatively basic; it serves to give you an understanding of how the processing is performed.

The `<form>` action attribute states that the `MathServlet` should be used to process the values that are contained within the form. The `MathServlet` name is mapped back to the `MathServlet` class via the `web.xml` deployment descriptor or the `@WebServlet` annotation. Looking at the `MathServlet` code, you can see that a `doPost` method is implemented to handle the processing of the `POST` form values. The `doPost` method accepts `HttpServletRequest` and `HttpServletResponse` objects as arguments. The values contained with the HTML form are embodied within the `HttpServletRequest` object. To obtain those values, call the request object's `getParameter` method, passing the `id` of the input parameter you want to obtain. In this example, those values are obtained and stored within local `String` fields.

```
String numA = req.getParameter("numa");
String numB = req.getParameter("numb");
```

Once the values are obtained, they can be processed as needed. In this case, those `String` values are converted into `int` values, and then they are added together to generate a sum and stored into an `int` field. That field is then presented as a response on a resulting web page.

```
int solution = Integer.valueOf(numA) + Integer.valueOf(numB);
```

As mentioned, the HTML form could be much more complex, containing any number of `<input>` fields. Likewise, the servlet could perform more complex processing of those field values. This example is merely the tip of the iceberg, and the possibilities are without bounds. Servlet-based web frameworks such as Java Server Pages and Java Server Faces hide many of the complexities of passing form values to a servlet and processing a response. However, the same basic framework is used behind the scenes.

Listening for Servlet Container Events

There are cases when it may be useful for an application to perform some tasks when it is being started up or shut down. In such cases, servlet context event listeners can become useful.

Example

Create a servlet context event listener to alert when the application has started up or when it has been shut down. The following solution demonstrates the code for a context listener, which will log application start-up and shutdown events and send e-mail alerting of such events:

```
package org.javaserverfaces.chapter01;

import java.util.Properties;
import javax.mail.Message;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.annotation.WebListener;

@WebListener
public class StartupShutdownListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent event) {
        System.out.println("Servlet startup...");
        System.out.println(event.getServletContext().getServerInfo());
        System.out.println(System.currentTimeMillis());
        sendEmail("Servlet context has initialized");
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        System.out.println("Servlet shutdown...");
        System.out.println(event.getServletContext().getServerInfo());
        System.out.println(System.currentTimeMillis());
        // See error in server.log file if mail is unsuccessful
        sendEmail("Servlet context has been destroyed...");
    }

    /**
     * This implementation uses the GMail smtp server
     * @param message
     * @return
     */
```

```
    private boolean sendEmail(String message) {
       boolean result = false;
       String smtpHost = "smtp.gmail.com";
       String smtpUsername = "username";
       String smtpPassword = "password";
       String from = "fromaddress";
       String to = "toaddress";
       int smtpPort = 587;
       System.out.println("sending email...");
        try {
            // Send email here

            //Set the host smtp address
            Properties props = new Properties();
            props.put("mail.smtp.host", smtpHost);
            props.put("mail.smtp.auth", "true");
            props.put("mail.smtp.starttls.enable", "true");

            // create some properties and get the default Session
            Session session = Session.getInstance(props);

            // create a message
            Message msg = new MimeMessage(session);

            // set the from and to address
            InternetAddress addressFrom = new InternetAddress(from);
            msg.setFrom(addressFrom);
            InternetAddress[] address = new InternetAddress[1];
            address[0] = new InternetAddress(to);
            msg.setRecipients(Message.RecipientType.TO, address);
            msg.setSubject("Servlet container shutting down");
            // Append Footer
            msg.setContent(message, "text/plain");
            Transport transport = session.getTransport("smtp");
            transport.connect(smtpHost, smtpPort, smtpUsername, smtpPassword);

            Transport.send(msg);

            result = true;
        } catch (javax.mail.MessagingException ex) {
            ex.printStackTrace();
            result = false;
        }
        return result;
    }
}
```

■ **Note** To run this example, you may need additional external JARs in your `CLASSPATH`. Specifically, make sure you have `mail.jar` and `javaee.jar`.

Explanation

Sometimes it is useful to know when certain events occur within the application server container. This concept can be useful under many different circumstances, but most often it would likely be used for initializing an application upon start-up or cleaning up after an application upon shutdown. A servlet listener can be registered with an application to indicate when it has been started up or shut down. Therefore, by listening for such events, the servlet has the opportunity to perform some actions when they occur.

To create a listener that performs actions based upon a container event, you must develop a class that implements the `ServletContextListener` interface. The methods that need to be implemented are `contextInitialized` and `contextDestroyed`. Both of the methods accept a `ServletContextEvent` as an argument, and they are automatically called each time the servlet container is initialized or shut down, respectively. To register the listener with the container, you can use one of the following techniques:

- Utilize the `@WebListener` annotation, as demonstrated by the solution to this example.
- Register the listener within the `web.xml` application deployment descriptor.
- Use the `addListener` methods defined on `ServletContext`.

For example, to register this listener within `web.xml`, you would need to add the following lines of XML:

```
<listener>
    <listener-class> org.javaserverfaces.chapter01.StartupShutdownListener</listener-class>
</listener>
```

Neither way is better than the other. The only time that listener registration within the application deployment descriptor (`web.xml`) would be more helpful is if you had the need to disable the listener in some cases. On the other hand, to disable a listener when it is registered using `@WebListener`, you must remove the annotation and recompile the code. Altering the web deployment descriptor does not require any code to be recompiled.

There are many different listener types, and the interface that the class implements is what determines the listener type. For instance, in this example, the class implements the `ServletContextListener` interface. Doing so creates a listener for servlet context events. If, however, the class implements `HttpSessionListener`, it would be a listener for HTTP session events. The following is a complete listing of listener interfaces:

```
javax.servlet.ServletRequestListener
javax.servlet.ServletRequestAttrbiteListener
javax.servlet.ServletContextListener
javax.servlet.ServletContextAttributeListener
javax.servlet.HttpSessionListener
javax.servlet.HttpSessionAttributeListener
```

It is also possible to create a listener that implements multiple listener interfaces. To learn more about listening for different situations such as attribute changes, please see the section entitled Listening for Attribute Changes.

Setting Initialization Parameters

It is possible to set initialization parameters for servlets as well. Doing so can be handy in cases where you would like to implement a task with default values if none were given.

Example #1

Set the servlet initialization parameters using the `@WebInitParam` annotation. The following code sets an initialization parameter that is equal to a `String` value:

```
package org.javaserverfaces.chapter01;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.*;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet(name="SimpleServletCtx1", urlPatterns={"/SimpleServletCtx1"},
initParams={ @WebInitParam(name="name", value="Duke") })
public class SimpleServletCtx1 extends HttpServlet {

        @Override
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws IOException, ServletException {

        res.setContentType("text/html");

        PrintWriter out = res.getWriter();

        /* Display some response to the user */

        out.println("<html><head>");
        out.println("<title>Simple Servlet Context Example</title>");
        out.println("\t<style>body { font-family: 'Lucida Grande', " +
            "'Lucida Sans Unicode';font-size: 13px; }</style>");
        out.println("</head>");
        out.println("<body>");

        out.println("<p>This is a simple servlet to demonstrate context!  Hello "
                                + getServletConfig().getInitParameter("name") + "</p>");

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

To execute the example using the sources for this book, load the following URL into your web browser: `http://localhost:8080/JSFByExample/SimpleServletCtx1`. The resulting web page will display the following text:

```
This is a simple servlet to demonstrate context! Hello Duke
```

Example #2

Place the `init` parameters inside the `web.xml` deployment descriptor file. The following lines are excerpted from the `web.xml` deployment descriptor for the `SimpleServlet` application. They include the initialization parameter names and values.

```
<web-app>
    <servlet>
        <servlet-name>SimpleServletCtx1</servlet-name>
        <servlet-class> org.javaserverfaces.chapter01.SimpleServletCtx1</servlet-class>

        <init-param>
            <param-name>name</param-name>
            <param-value>Duke</param-value>
        </init-param>
    ...
    </servlet>
    ...
</web-app>
```

Explanation

Oftentimes there is a requirement to set initialization parameters for a servlet in order to initialize certain values. Servlets can accept any number of initialization parameters, and there are a couple of ways in which they can be set. The first example is to annotate the servlet class with the `@WebInitParam` annotation, and the second way to set an initialization parameter is to declare the parameter within the `web.xml` deployment descriptor, as demonstrated in the second example. Either way will work; however, the solution using `@WebInitParam` is based upon the newer Java Servlet 3.0 API. Therefore, Example #1 is the more contemporary approach, but Example #2 remains valid for following an older model or using an older Java servlet release.

To use the `@WebInitParam` annotation, it must be embedded within the `@WebServlet` annotation. Therefore, the servlet must be registered with the web application via the `@WebServlet` annotation rather than within the `web.xml` file. For more information on registering a servlet via the `@WebServlet` annotation, see the section entitled Registering Servlets Without web.xml.

The `@WebInitParam` annotation accepts a name-value pair as an initialization parameter. In the solution to this example, the parameter name is `name`, and the value is `Duke`.

```
@WebInitParam(name="name", value="Duke")
```

Once set, the parameter can be used within code by calling `getServletConfig().getInitializationParameter()` and passing the name of the parameter, as shown in the following line of code:

```
out.println("<p>This is a simple servlet to demonstrate context!  Hello "
                                + getServletConfig().getInitParameter("name") + "</p>");
```

The annotations have the benefit of providing ease of development, and they also make it easier to maintain servlets as a single package rather than jumping back and forth between the servlet and the deployment descriptor. However, those benefits come at the cost of compilation because in order to change the value of an initialization parameter using the `@WebInitParam` annotation, you must recompile the code. Such is not the case when using the `web.xml` deployment descriptor. It is best to evaluate your application circumstances before committing to a standard for naming initialization parameters.

Filtering Web Requests

Another useful technique can be to apply a filter against a specified URL for a servlet. A filter can then invoke custom processing each time the URL is visited, and the filter will be executed prior to the servlet.

Example

Create a servlet filter that will be processed when the specified URL format is used to access the application. In this example, the filter will be executed when a URL conforming to the format of /* is used. This format pertains to any URL in the application. Therefore, any page will cause the servlet to be invoked.

```
package org.javaserverfaces.chapter01;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.*;

/**
 * This filter obtains the IP address of the remote host and logs
 * it.
 *
 * @author juneau
 */
@WebFilter("/*")
public class LoggingFilter implements Filter {

    private FilterConfig filterConf = null;

    public void init(FilterConfig filterConf) {
        this.filterConf = filterConf;
    }

    public void doFilter(ServletRequest request,
            ServletResponse response,
            FilterChain chain)
            throws IOException, ServletException {
        String userAddy = request.getRemoteHost();
        filterConf.getServletContext().log("Vistor User IP: " + userAddy);
        chain.doFilter(request, response);
    }

    @Override
    public void destroy() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

The filter could contain any processing; the important thing to note is that this servlet is processed when a specified URL is used to access the application.

■ **Note** To invoke the filter, load a URL for the application with which the filter is associated. For the purposes of this example, load the following URL (for the previous example) to see the filter add text to the server log: `http://localhost:8080/JSFByExample/SimpleServletCtx1`.

How It Works

Web filters are useful for preprocessing requests and invoking certain functionality when a given URL is visited. Rather than invoking a servlet that exists at a given URL directly, any filter that contains the same URL pattern will be invoked prior to the servlet. This can be helpful in many situations, perhaps the most useful for performing logging, authentication, or other services that occur in the background without user interaction.

Filters must implement the `javax.servlet.Filter` interface. Methods contained within this interface include `init`, `destroy`, and `doFilter`. The `init` and `destroy` methods are invoked by the container. The `doFilter` method is used to implement tasks for the filter class. As you can see from this example, the filter class has access to the `ServletRequest` and `ServletResponse` objects. This means the request can be captured, and information can be obtained from it. This also means the request can be modified if need be. For example, including the user name in the request after an authentication filter has been used.

If you want to chain filters or if more than one filter exists for a given URL pattern, they will be invoked in the order in which they are configured in the `web.xml` deployment descriptor. It is best to manually configure the filters if you are using more than one per URL pattern rather than using the `@WebFilter` annotation. To manually configure the `web.xml` file to include a filter, use the `<filter>` and `<filter-mapping>` XML elements along with their associated child element tags. The following excerpt from a `web.xml` configuration file shows how the filter that has been created for this example may be manually configured within the `web.xml` file:

```
<filter>
    <filter-name>LoggingFilter</filter-name>
    <filter-class>LoggingFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>LogingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Of course, the `@WebFilter` annotation takes care of the configuration for you, so in this case the manual configuration is not required.

■ **Note** As of Servlet 3.1 API, if a filter invokes the next entity in the chain, each of the filter `service` methods must run in the same thread as all filters that apply to the servlet.

Listening for Attribute Changes

Servlets can perform listening event tasks when HTTP session attributes are changed by implementing the `HttpSessionAttributeListener` interface.

Example

This example demonstrates how to generate an attribute listener servlet to listen for such events as attributes being added, removed, or modified. The following class demonstrates this technique by implementing `HttpSessionAttributeListener` and listening for attributes that are added, removed, or replaced within the HTTP session:

```
package org.javaserverfaces.chapter01;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;

/**
 * Attribute Listener
 */
@WebListener
public final class AttributeListener implements ServletContextListener,
        HttpSessionAttributeListener {

    private ServletContext context = null;

    @Override
    public void attributeAdded(HttpSessionBindingEvent se) {
        HttpSession session = se.getSession();
        String id = session.getId();
        String name = se.getName();
        String value = (String) se.getValue();
        String message = new StringBuffer("New attribute has been added to session: \n").
        append("Attribute Name: ").append(name).append("\n").append("Attribute Value:").
        append(value).toString();
        log(message);
    }

    /**
     *
     * @param se
     */
    @Override
    public void attributeRemoved(HttpSessionBindingEvent se) {
        HttpSession session = se.getSession();
        String id = session.getId();
```

```
        String name = se.getName();
        if (name == null) {
            name = "Unknown";
        }
        String value = (String) se.getValue();
        String message = new StringBuffer("Attribute has been removed: \n")
        .append("Attribute Name: ").append(name).append("\n").append("Attribute Value:")
        .append(value).toString();
        log(message);
    }

    @Override
    public void attributeReplaced(HttpSessionBindingEvent se) {
        String name = se.getName();
        if (name == null) {
            name = "Unknown";
        }
        String value = (String) se.getValue();
        String message = new StringBuffer("Attribute has been replaced: \n ").append(name).
        toString();
        log(message);
    }

    private void log(String message) {
        if (context != null) {
            context.log("SessionListener: " + message);
        } else {
            System.out.println("SessionListener: " + message);
        }
    }

    @Override
    public void contextInitialized(ServletContextEvent event) {
        this.context = event.getServletContext();
        log("contextInitialized()");
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
// Do something
    }
}
```

Messages will be displayed within the server log file indicating when attributes have been added, removed, or replaced.

Explanation

In some situations, it can be useful to know when an attribute has been set or what an attribute value has been set to. This example demonstrates how to create an attribute listener in order to determine this information. To create a servlet listener, you must implement one or more of the servlet listener interfaces. To listen for HTTP session attribute changes, implement `HttpSessionAttributeListener`. In doing so, the listener will implement the `attributeAdded`, `attributeRemoved`, and `attributeReplaced` methods. Each of these methods accepts `HttpSessionBindingEvent` as an argument, and their implementation defines what will occur when an HTTP session attribute is added, removed, or changed, respectively.

In this example, you can see that each of the three methods listed in the previous paragraph contains a similar implementation. Within each method, the `HttpSessionBindingEvent` is interrogated and broken down into `String` values, which represent the ID, name, and value of the attribute that caused the listener to react. For instance, in the `attributeAdded` method, the session is obtained from `HttpSessionBindingEvent`, and then the session ID is retrieved from that via the use of `getSession`. The attribute information can be obtained directly from the `HttpSessionBindingEvent` using the `getId` and `getName` methods, as shown in the following lines of code:

```
HttpSession session = se.getSession();
String id = session.getId();
String name = se.getName();
String value = (String) se.getValue();
```

After these values are obtained, the application can do whatever it needs to do with them. In this example, the attribute ID, name, and session ID are simply logged and printed.

```
String message = new StringBuffer("New attribute has been added to session: \n")
.append("Attribute Name: ").append(name).append("\n")
.append("Attribute Value:").append(value).toString();
log(message);
```

The body of the `attributeReplaced` and `attributeRemoved` methods contain similar functionality. In the end, the same routine is used within each to obtain the attribute name and value, and then something is done with those values.

A few different options can be used to register the listener with the container. The `@WebListener` annotation is the easiest way to do so, and the only downfall to using it is that you will need to recompile code in order to remove the listener annotation if you ever need to do so. The listener can be registered within the web deployment descriptor, or it can be registered using one of the `addListener` methods contained in `ServletContext`.

Although the example does not perform any life-changing events, it does demonstrate how to create and use an attribute listener. In the real world, such a listener could become handy if an application needed to capture the user name of everyone who logs in or needed to send an e-mail whenever a specified attribute is set.

Applying a Listener to a Session

In the same way that a listener can be applied to an HTTP session to listen for attribute changes, a listener can be applied for performing tasks when sessions are created and destroyedAssume in the following example that you wish to listen for sessions to be created so that you can count how many active sessions your application currently contains, as well as perform some initialization for each session.