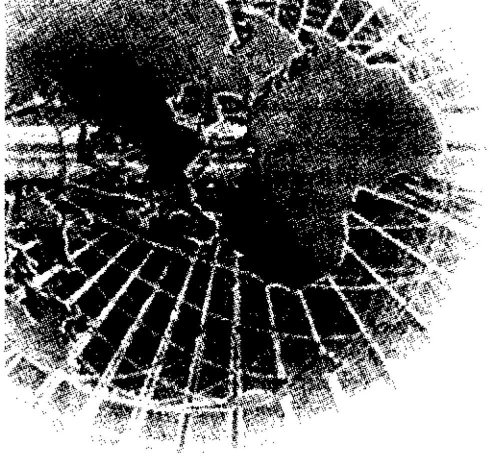


11 Working with Servlet Programming

The Java Servlet technology provides a simple, vendor-independent mechanism to extend the functionality of a Web server. This technology provides high level, component-based, platform-independent, and server-independent standards to develop Web applications in Java. The Java Servlet technology is similar to other scripting languages, such as Common Gateway Interface (CGI) scripts, JavaScript (on the client-side), and Hypertext Preprocessor (PHP). However, servlets are more acceptable since they overcome the limitations of CGI, such as low performance and scalability.

A servlet is a simple Java class, which is dynamically loaded on a Web server and thereby enhances the functionality of the Web server. Servlets are secure and portable as they run on Java Virtual Machine (JVM) embedded with the Web server and cannot operate outside the domain of the Web server. In other words, servlets are objects that generate dynamic content after processing requests that originate from a Web browser. They are Java components that are used to create dynamic Web applications. Servlets can run on any Java-enabled platform and are usually designed to process HyperText Transfer Protocol (HTTP) requests, such as GET, and POST.

In this chapter, you learn about the Java Servlet API, version 2.5, which can be downloaded from the `http://java.sun.com/products/servlet/index.jsp` Uniform Resource Locator (URL). This API includes two packages, `javax.servlet` and `javax.servlet.http`, which provide interfaces and classes to write servlets.

The chapter first introduces the new features introduced in Servlet 2.5 followed by a discussion about general features of a Java Servlet. Next, the chapter explains the classes and packages of the Servlet Application Programming Interface (API) used to develop Web applications. You also learn about the life cycle of a servlet and configuring a servlet in the web.xml file. In addition, you learn to create a sample servlet by mapping it in the web.xml file. Moreover, the chapter also provides a walkthrough to the noteworthy interfaces of the `javax.servlet` and `javax.servlet.http` packages. Toward the end of the chapter, you learn about request delegation, request scope and servlet collaboration. The chapter also explains the concept of session tracking. You also learn about how to implement the concept by creating a sample Web application using session tracking.

Just as Servlet 2.2 introduced the concept of self-contained Web applications, Servlet 2.3 introduced filters, and Servlet 2.4 provided deprecated classes and methods, Servlet 2.5 too includes several new features and enhancements over the previous version. You learn about them later in the chapter.

Let's begin the chapter by first discussing Servlet 2.5.

Introducing Servlet 2.5

Servlet 2.5 introduced several new advancements, such as changes in the web.xml file and in filter mapping. We assume that you are familiar with the classes and methods of the previous versions of Java Servlet. The following features and enhancements have been included in Servlet 2.5:

- Provides support for the new language features of J2SE 5.0, such as generics, the new enum type, and autoboxing. Note that the minimum platform requirement for the Servlet 2.5 specification is JDK 1.5 (J2SE 5.0). This implies that Servlet 2.5 cannot be used in versions below JDK 1.5.
- Introduces changes in the web.xml file that have allowed Java developers to easily configure the resources of an application. In addition, in Servlet 2.5, you can bind all servlets to a filter simultaneously, which was not possible in the earlier versions. Now, it is possible to use an asterisk (*) within the <filter-mapping> element as the value of the <servlet-name> element to represent all servlets. Moreover, you can provide multiple matching criteria in the same entry while writing the <servlet-mapping> or <filter-mapping> elements in Servlet 2.5. Earlier, the <servlet-mapping> element supported a single <url-pattern> element; however, now in Servlet 2.5, the <servlet-mapping> element supports more than one url pattern.
- Removes two major restrictions in error-handling and session tracking. Earlier, there was a rule that the resource configured in the <error-page> element could not call the setStatus() method to alter the code provided to display an error message. However, the Servlet 2.5 specification no longer prevents the error-handling page to produce a non-error response. Therefore, the error-handling page can do far more than

just show an error. Moreover, in the case of session tracking, Servlet 2.5 does not allow you to set response headers for a servlet called by the RequestDispatcher's include() method.

- Clarifies certain options available in the Servlet 2.4 specification. These clarifications are as follows:
  - According to the Servlet 2.4 specification, before calling the request.getReader() method, you need to call the request.setCharacterEncoding() method. However, the specification does not clarify why this needs to be done. The Servlet 2.5 specification describes this properly and states that if you ignore this specification option, the request.getReader() method is not executed.
  - The Servlet 2.4 specification does not define what happens if a session id is not specified. However, the Servlet 2.5 specification states that the `HttpServletRequest` interface will return false if the session id is not specified.
  - The Servlet 2.4 specification states that a response should be committed in most situations. The following code snippet shows a situation where the amount of content specified in the setContentLength() method of the response is not greater than zero and has been returned to the response:

```
res.setHeader("Host","localhost");
res.setHeader("Pragma","no-cache");
res.setHeader("Content-Length","0");
res.setHeader("Location","www.kogentindia.com")
```

In the preceding code snippet, a servlet technically ignores the `Location` header because the response must be committed immediately, as the zero byte content length is satisfied. However, the Servlet 2.5 specification states that the response should be committed when the amount of content specified in the setContentLength() method of the response is greater than zero, and is returned to the response.

  - The Servlet 2.5 specification changes the rules of cross context session management. This feature is used when Servlets dispatch requests from one context to another. The Servlet 2.5 specification states that resources present in a context can refer to the session of that context, irrespective of the origin of a request.

After discussing the new features and enhancements introduced in Servlet 2.5, let's now explore the features of Java Servlets.

Exploring the Features of Java Servlets

Java Servlet provides various key features, such as security, performance, as well as the request and response model. Servlets are considered as a request and response model in which requests are sent by users and their appropriate responses are generated by a Web server. In addition, a key feature of a servlet is that you can create multiple instances of a servlet with different data and each servlet can be configured with different names. A Java Servlet also provides support for the security policy used to control accessibility permissions, such as a user accessing a resource. In addition, scripting languages can be used in servlets to dynamically modify or generate Hypertext Markup Language (HTML) pages. Apart from this, servlets also support various HTTP methods, such as GET and POST, which are used to redirect requests and responses.

Now, let's learn about these features in detail.

Servlet – A Request and Response Model

Servlets are based on the programming model that accepts requests and generates responses accordingly. A developer extends the GenericServlet or HttpServlet class to create a servlet. The `service()` method in a servlet is defined to handle requests and responses. The following code snippet defines the service() method for the `MyServletApplication` servlet class:

```
import javax.servlet.*;
public class MyServletApplication extends GenericServlet {
  public void service (ServletRequest  request, ServletResponse  response)
  throws ServletException, IOException
  {
        ...
  }
```

```
    ...
}
```

The `service()` method is provided with request and response parameters. These parameters encapsulate the data sent by a client, which provides access to the parameters and allows servlets to generate responses. Servlets normally use an input stream to retrieve most of their parameters, and an output stream is used to send responses. The following code snippet shows how the request parameter is used to invoke the `getInputStream()` method:

```
ServletInputStream    input = request.getInputStream();
ServletOutputStream   output = response.getOutputStream();
```

In the preceding code snippet, instances of the input and output streams are created, which may be used to read or write data.

Servlet and Environment State

Servlets are similar to any other Java objects and have instance-specific data. This implies that servlets are also independent applications that run within the server environment and do not require any additional classes (which are required by some alternative server extension APIs).

When servlets are initialized, they have access to some servlet-specific configuration data. This enables the initialization of different instances of the same servlet-class with different data, and their management as differently named servlets. The data, provided with each servlet instance at the time of its initialization, also includes some information about the persistent state of an instance. The `ServletContext` object provides the ability to servlets to interact with other servlets in a Web application.

Next, let's discuss the different modes in which a servlet can be used.

Usage Modes

Servlets can be used in various modes. However, these modes are not supported by all server environments. At the core of a request–response protocol, the basic modes in which servlets can be used are as follows:

- Servlets can be chained together into filter chains by the servers
- Servlets can support protocols, such as HTTP
- Servlets serve as a complete, efficient, and portable replacement for CGI-based extensions in HTTP-based applications
- Servlets can be used with HTML to dynamically generate some parts of a Web document in HTTP-based applications

Now, let's discuss the life cycle of a servlet.

Servlet Life Cycle

When a server loads a servlet, the `init()` method of the servlet is executed. The servlet initialization process is completed before any client request is addressed or before the servlet is destroyed. The server calls the init() method once, when the server loads the servlet, and does not call the method again unless the server reloads the servlet. A server cannot reload a servlet after the servlet is destroyed. After initialization, the servlet handles client requests and generates responses. Finally, the destroy() method is invoked, to destroy the servlet.

Let's discuss various possible sources from where a servlet is loaded. Moreover, you also explore different situations in which a servlet is loaded.

Possible Sources of Servlets

When a client requests for a servlet, the server maps the request of the client and loads the servlet. The server administrator can specify the mapping of client requests to servlets in the following ways:

- Mapping client requests to a particular servlet, for example, client requests made to a specific database.
- Mapping client requests to the servlets found in an administered servlets directory. This servlet directory may be shared among different servers that share the processing load for the clients of a website.

- Configuring some servers to automatically invoke servlets that filter the output of other servlets. For example, a particular type of output generated by a servlet may invoke other servlets to carry out post processing, probably to perform format conversions.
- Invoking the specific servlets without administrative intervention by properly authorized clients.

Figure 11.1 displays the various sources of servlets:

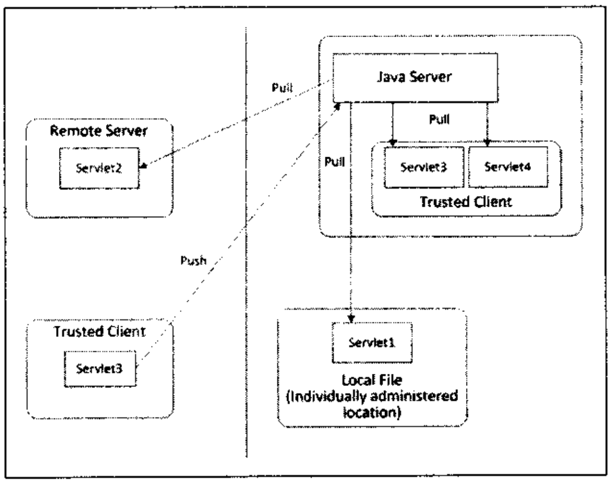


Figure 11.1: Displaying the Possible Sources of Loading a Servlet

Figure 11.1 shows the various possible sources of servlets, which can be as follows:

- Individually administered locations
- Directory of servlets that are shared between servers
- Authorized clients
- After having a brief understanding about life cycle of a servlet and exploring its possible sources, let's describe the primary methods of a servlet, which are init(), service(), and destroy().

Primary Servlet Methods

The following are the methods used in the life cycle of a loaded servlet:

- `init()`–Refers to the method that an application server uses to load and initialize a servlet. The `init()` method is typically used to create or load objects that will be used by the servlet to handle client requests. The application server calls the `init()` method to provide a new servlet with the information related to its context.
- `service()`–Helps servlets to handle client requests. Servlets handle many requests after initialization. One `service()` method call is generated by each client request. These requests may be concurrent. This enables servlets to coordinate activities among multiple clients. To share data between requests, the class-static state may be used.
- `destroy()`–Terminates an executing servlet. Servlets process client requests until they are explicitly terminated by the Web server by calling the `destroy()` method. When a servlet is destroyed, it becomes eligible for garbage collection.

Security Features

Servlets have access to information about their clients. Peer identities can be determined reliably when servlets are used with secure protocols, such as Secure Sockets Layer (SSL). Servlets that rely on HTTP also have access to HTTP-specific authentication data.

Servlets have various advantages of Java. For example, as in Java, memory access violations and strong typing violations are also not possible with servlets. Due to these advantages, faulty servlets do not crash servers, which is common in most C language server extension environments.

Java Servlet provides strong security policy support unlike any other current server extension API. This is because a security manager, provided by all Java environments, can be used to control the permissions for actions such as accessing a network or file. Servlets are not considered trustworthy in general and are not allowed to carry out the actions by default. However, servlets that are either built into the server, or digitally signed servlets that are put into Java ARchive (JAR) files are considered trustworthy and granted permissions by the security manager. A digital signature on any executable code ensures that the organization that created and signed the code takes the guarantee of its trustworthiness. Such digital signatures cannot support answerability for the code by themselves. However, they do provide assurance about the use of that code. For example, all code that accesses network services within a corporate Intranet of the Management Information System (MIS) organization may require having a particular signature, to access those network services. The signature on the code ensures that the code does not violate any security policy. Figure 11.2 displays the approaches to server extensions depicting the Java server security manager layer used by servlets to verify permissions:

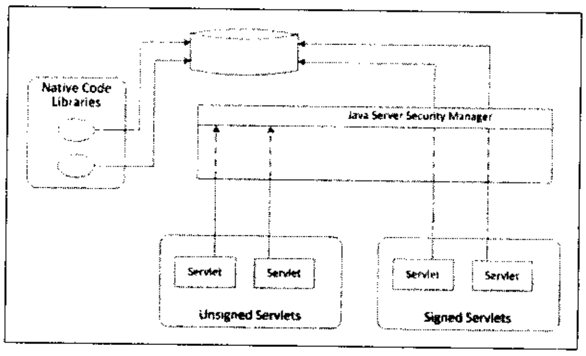


Figure 11.2: Showing the Activities of Signed and Unsigned Servlets

Figure 11.2 compares two approaches to server extensions:

- Activities of servlets in the case of signed servlets, which are monitored at fine granularity by the Java security manager
- Activities of native code extensions in the case of unsigned servlets, which are never monitored

In both cases, the host operating system usually provides very coarse-grained protection. In languages such as C or scripting languages, extension APIs cannot support fine-grained access controls, even if they do allow digital signatures for their code. This explains that Pure Java™ extensions are fundamentally more secure than current competitive solutions including, in particular ActiveX of Microsoft.

There are some immediate commercial applications for such improved security technologies. At present, it is not possible for most Internet Service Providers (ISPs) to accept server extensions from their clients. The reason is that the ISPs do not have proper methods to protect themselves or their clients from the attacks building on extensions, which use native C code or CGI facilities. However, it has been observed that extensions built with pure Java Servlet can prevent modification of data. Along with the use of digitally signed code, ISPs can ensure that they safely extend their Web servers with the servlets provided by their customers.

HTML-Aware Servlets

It has been observed that many servlets directly generate HTML formatted text, because it is easy to do so with standard internationalized Java formatted output classes, such as `java.io.PrintWriter`. To dynamically modify HTML pages or generate HTML pages, you do not have to use scripting languages. You can also use other approaches to generate Java HTML formatted text. For example, some multi-language sites that serve pages in multiple languages, such as English and Japanese, usually maintain language-specific libraries of localized HTML template files. These sites also display the localized HTML templates by using localized message catalogs. Other sites may also have developed HTML generation packages. These packages are particularly well accustomed to other specific needs for dynamic Web page generation, for example, the ones that are closely integrated with other application toolsets.

Servlets may also be invoked by Web servers that provide complete servlet support; to help in preprocessing Web pages by using the server-side include functionality. This kind of preprocessing can be indicated to Web servers by a special HTML syntax. The following code snippet shows the syntax that is used in HTML files to indicate preprocessing of Web pages:

```
<SERVLET NAME=ServletName>
  <PARAM NAME=param1 VALUE=val1>
  <PARAM NAME=param2 VALUE=val2>
  If you see this text, it means that the Web server providing this page
  does not support the SERVLET tag. Ask your Internet Service Provider to
  upgrade!
< /SERVLET>
```

In the preceding code snippet, the invocation style usage of the `SERVLET` tag indicates that a preconfigured servlet should be loaded and initialized in cases where it has not already been done, and then invoked with a specific set of parameters. The output of that servlet is included directly in the HTML-formatted response. Apart from using the SERVLET tag, another invocation style can also be used, which allows the passing of the initialization arguments to the servlet and specifies its `CLASS` and `CODEBASE` values directly.

The `SERVLET` tag could be used to insert formatted data. The formatted data can be the output of a Web or database search, user-targeted advertising, or the individual views of an online magazine. HTML-aware servlets can generate arbitrary dynamic Web pages in which typical servlets accept input parameters from different sources. Some of these sources are as follows:

- The input stream of a request, perhaps from an applet
- The Uniform Resource Identifier (URI) of the request
- Any other servlet or the network service
- Parameters passed from an HTML form

The input parameters are used to generate HTML-formatted responses. A servlet often checks with one or more databases, or other data with which the servlet is configured, to decide the exact data that is to be returned with the response.

HTTP-Specific Servlets

HTTP-specific servlets are those servlets that are used with the HTTP protocol. These servlets can support any HTTP method, such as `GET`, `POST`, and `HEAD`; redirect requests to other locations; and send HTTP-specific error messages. They can also have access to the parameters passed through standard HTML forms. HTTP-specific servlets include the HTTP method to be executed and the Uniform Resource Identifier (URI), which describes the destination of the request. The following code snippet shows some of the methods used in HTTP-specific servlets:

```
String method = request.getMethod();    // e.g. POST
String uri = request.getRequestURI();
String name = request.getParameter("name");
String phone = request.getParameter ("phone");
String card = request.getParameter ("creditcard");
```

In HTTP-specific servlets, request and response data is always provided in the Multipurpose Internet Mail Extensions (MIME) format. This implies that the servlet first specifies the data type and then writes the encoded data. This allows servlets to refer the data format regarding arbitrary sources of input data, and then return the data in the appropriate form for the particular request. Examples of request and response data formats are HTML, graphics formats, such as `Joint Photographic Experts Group (JPEG)` or `Moving Picture Experts Group (MPEG)`, and data formats that are used by some applications.

In most applications, HTTP servlets are considered better than CGI programs in terms of performance, flexibility, portability, and security. Therefore, rather than using CGI or a C language plug-in, write your next server extension by using the Java Servlet API.

Performance Features

Let's discuss the performance feature of servlets. One of the most prominent performance features of Java Servlets is that a new process need not be created for every new request received. In most environments, several servlets run in parallel to the server within the same process. This is a big advantage. When you use servlets in such environments with HTTP, performance is assumed to be much better than what it would be if the CGI or Fast-CGI approach was used.

Figure 11.3 displays the different approaches to depict the functionality and performance of a servlet:

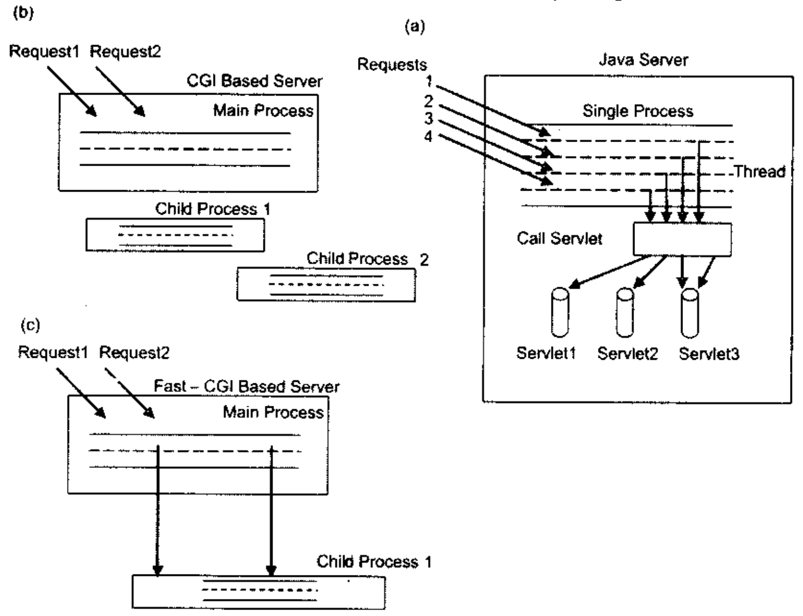


Figure 11.3: Showing Different Servlet Functional Approaches

Figure 11.3 compares the following three server extension approaches:

- The servlet approach, which allows embedding to be supported inside a server, as shown by section (a) of Figure 11.3
- The CGI approach, which involves creating a new child process with every new request, as shown by section (b) of Figure 11.3
- The Fast-CGI approach, which involves creating one child process for multiple requests, as shown by section (c) of Figure 11.3

The difference between these approaches is that the servlet approach require only lightweight thread context switches, whereas Fast-CGI involves heavyweight process context switching on each request, and regular CGI requires even heavier weight process start-up and initialization code on each request. After a servlet is initialized, it can address requests from multiple clients in most environments. This spreads the cost of initialization over many methods. It also enables all client requests made to a service to share data and communication resources as well as use system caches effectively.

Java Servlet can take the advantage of additional processors, with multiple implementations of JVM. The virtual machine (JVM) enables you to scale applications from entry-level servers to the mainframe class multiprocessors. This also helps to provide better throughput and timely response to clients. Pure Java programs are platform-independent; therefore, they can run on any operating system. In other words, you can select any operating system that best addresses your requirements for any given application. The implementation of Java Servlet is beneficial for many large Web-based applications that use Java and other Internet technologies.

3-Tier Applications

Using Java Servlet helps a user to opt for 3-tier applications. Many organizations require you to use multi-tier applications. Many clients and single server models are giving way to a single application, which includes many servers that exchange data between each other.

The first tier of an application may use any number of Java enabled Web browsers. The browsers can include those running on Network Computers (NCs) as well as on personal computers or workstations. Complex tasks related to the user interface are handled in the first tier by Java applets downloaded from second-tier servers. Simpler tasks can be handled by using standard HTML forms.

The second tier of an application involves servlets, which implement the specific business rules and business logic of the application. Such rules can include application-specific access controls for sensitive corporate data.

Figure 11.4 displays the 3-tier structure of servlets:

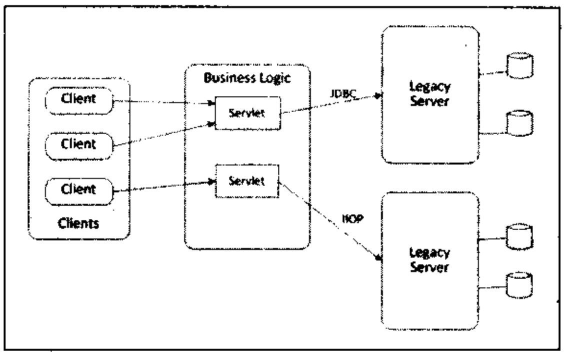


Figure 11.4: Showing 3-Tier Structure of Servlets

Figure 11.4 shows how servlets can be used to connect the second tier of an application to the first tier. A variety of client technologies may be used to connect to other tiers.

The third tier of an application involves data repositories. This tier can be accessed by using relational database interfaces such as Java Database Connectivity (JDBC), or other interfaces supported by legacy data, for example, Remote Procedure Call (RPC)-like protocols such as Open Network Computing (ONC) RPC, Distributed Computing Environment (DCE) RPC, and Common Object Request Broker Architecture (COBRA)/Internet Inter-Orb Protocol (IIOP).

Web Publishing System

Many organizations have large collections of data. They have to manage and publish the data, which is usually flexible and tends to change. For example, an organization may maintain a collection of both historical and real-time weather data that needs to be presented in easily understood formats in the form of a response to the current application.

In this case, you can use a Web publishing system that provides sites to access historical and real-time weather data from a database. In other words, the required data can be accessed from the database by using JDBC. The weather data (including temperature, wind, rainfall, a frame of image data, or a stream of MPEG data) is sent by a Java equipped remote recording station to a site receiving the data. The servlet processing the data at the collecting site may selectively store the data. For example, it may store data of a specific time period, such as the last two weeks, or it may discard some data immediately.

The collecting sites may receive queries from other sites, such as individual browsers or other collecting sites, to return data in a specific format. In that case, servlets process the saved data and return the response of the query in the appropriate format, such as a Web page with current data and historical tables and graphs, to a user. Some servlets can also perform administrative tasks, such as archiving and deleting data, or pulling data from staging areas, as part of an automated data distribution system.

Figure 11.5 shows communication between two servlets:

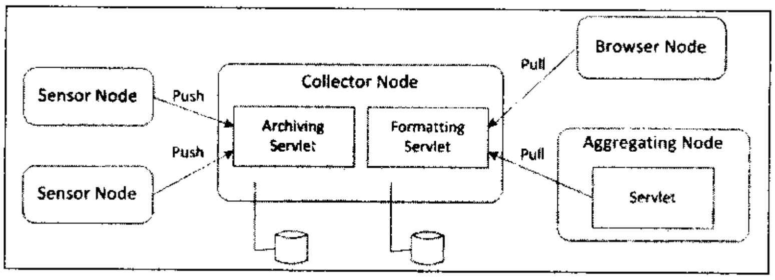


Figure 11.5: Showing Servlet Communication

Figure 11.5 shows two kinds of servlets used by a collecting site. One is used by the remote sensor nodes to push data to the collecting site, and the other is used by clients to pull data from the collecting site in a specific format. Such clients can include tertiary nodes, which assemble data from multiple collecting sites.

After exploring the features of Java Servlets, let's have a discussion about the Servlet API.

Exploring the Servlet API

The Servlet API is a part of the Java Servlet specification designed by the Java Community Process (JCP). This API is supported by all servlet containers, such as Tomcat and WebLogic. The Servlet API contains classes and interfaces that define a standard contract between a servlet class a Servlet container (or Servlet engine). These classes and interfaces are available in the following two packages of the Servlet API:

- javax.servlet
- javax.servlet.http
- Let's learn about these packages in detail in the following sections.

The javax.servlet Package

The `javax.servlet` package contains some interfaces and classes that allow a servlet to access the basic services provided by a Servlet container. The Servlet container provides the implementation of the classes and interfaces packaged under the `javax.servlet` package.

The central abstraction of the Servlet API is the `Servlet` interface. The two classes in the Servlet API that implement the `Servlet` interface are `GenericServlet` and `HttpServlet`. Generally, developers implement the `HttpServlet` interface to create their servlets for Web applications that employ the HTTP protocol.

The basic Servlet interface defines a `service()` method to handle client requests. This method is called for each request that is routed to an instance of a servlet by the Servlet container.

The contract between a Web application and a Web container is provided by the `javax.servlet` package. This allows Servlet container vendors to focus on developing the container in the manner most suited to their requirements (or those of their customers), as long as the package provides the specified implementations of the interfaces used in a servlet. The package provides a standard library to process client requests and develop servlet-based applications, from a developer's perspective.

The Servlet interface that defines the core structure of a servlet is provided in the `javax.servlet` package, which is the basis for all servlet implementations. However, for most servlet implementations, the subclass from a defined implementation of this interface provides the basis for a Web application.

The various interfaces and classes, such as `ServletConfig` and `ServletContext`, provide the additional services to a developer. An example of such a service is the Servlet container that provides a servlet with access to a client request through a standard interface. The `javax.servlet` package, therefore, provides the basis to develop a cross-platform, cross-servlet container Web application, and allows programmers to focus on developing a Web application.

Developers sometimes also use the `javax.servlet.http` package. Additionally, you need the `javax.servlet` package to build servlet implementations that use a non-HTTP protocol. For example, you can extend classes from the `javax.servlet` package to implement a Simple Mail Transfer Protocol (SMTP) servlet that provides an e-mail service to clients.

Let's now discuss the interfaces, classes, and exception classes of the `javax.servlet` package.

Explaining the Interfaces and Classes of the javax.servlet Package

The `javax.servlet` package comprises fourteen interfaces. While building an application, a programmer can implement seven interfaces, such as Servlet, and ServletRequestListener. A Servlet container provides the implementation for the following seven interfaces:

- ServletConfig
- ServletContext
- ServletRequest
- ServletResponse
- RequestDispatcher
- FilterChain
- FilterConfig

The Servlet container must provide an object for the preceding interfaces to a servlet. The `getServletContext()` method is probably the most important method of the `ServletConfig` Interface. This method returns the `ServletContext` object, which communicates with the Servlet container when you want to perform some action, such as writing to a log file or dispatching requests. There is only one `ServletContext` object per Web application per JVM. This object is initialized when the Web application starts and is destroyed only when the Web application shuts down. One useful application of the `ServletContext` object is as a persistence mechanism. A programmer may store attributes in the `ServletContext` interface so that they are available throughout the execution of an application and not just for the duration of a request for a resource.

The Servlet container provides the classes that implement the `ServletRequest` and `ServletResponse` interfaces. These classes provide client request information to a servlet and the object used to send a response to the client.

An object defined by the `RequestDispatcher` interface manages client requests by directing them to an appropriate resource on the server.

The `FilterChain`, `FilterConfig`, and `Filter` interfaces are used to implement the filtering functionality in an application. You can also combine the interfaces into chains, implying that you can chain them such that before being processed by a container, the request is filtered through each filter defined in the application. The response goes down the chain in reverse.

A programmer building a Web application implements the following seven interfaces:

- Servlet
- ServletRequestListener
- ServletRequestAttributeListener
- ServletContextListener
- ServletContextAttributeListener
- SingleThreadModel
- Filter

The preceding interfaces are defined so that a Servlet container can invoke the methods defined in the interfaces. Therefore, the Servlet container needs to know only the methods defined in the interfaces. The details of the implementation of the methods are provided by the developers.

The event classes used to notify the changes made to the `ServletContext` object and its attributes are `ServetContextEvent` and `ServletContextAttributeEvent`, respectively.

Initially, the system creates a single instance of a servlet. If a new request is received while the previous one is being processed, a new thread is created for each new user request, with multiple threads running concurrently. This implies that the `doGet()` and `doPost()` methods need to carefully synchronize the access to fields and other shared data because multiple executing threads may access the data simultaneously. You can implement the `SingleThreadModel` interface in a servlet, if you want to prevent this multithreaded access, as shown by the following code snippet:

```
public class YourServlet extends HttpServlet implements SingleThreadModel {
    ...
}
```

If you implement the `SingleThreadModel` interface, the system ensures that a single instance of a servlet is never accessed by more than one request thread. This is implemented by queuing all the requests and passing them one by one to a single servlet instance. However, the server can create a pool of multiple instances, with each addressing one request at a time. This implies that there is no need to worry about simultaneous access to regular fields (instance variables) of a servlet. However, access to class variables (static fields) or shared data stored outside the servlet still needs to be synchronized.

Synchronous and frequent access to servlets can significantly hurt performance (latency). The server remains idle instead of handling pending requests, when a servlet waits for Input/Output (I/O). Therefore, think twice before using the `SingleThreadModel` approach.

NOTE

The SingleThreadModel interface has been deprecated as of Java Servlet API 2.4, with no direct replacement

Two classes, `ServletRequestEvent` and `ServletRequestAttributeEvent`, are used to indicate the life cycle events and attribute events for a `ServletRequest` object. The `ServletContext` object of a Web application is the source of the event.

To read or send binary data to or from a client, the `ServletInputStream` and `ServletOutputStream` classes provide input and output streams, respectively.

Useful implementations of the `ServletRequest` and `ServletResponse` interfaces are provided by the wrapper classes `ServletRequestWrapper` and `ServletResponseWrapper`, respectively. These implementations can be subclassed to allow programmers to adapt or enhance the functionality of the wrapped object for their own Web application. This can be done to implement a basic protocol agreed between a client and a server or to transparently adapt the requests or responses to a particular format that the Web application requires.

Explaining the Exception Classes of the **javax.servlet** Package

The following two exceptions are present in the `javax.servlet` package:

- ServletException
- UnavailableException

Generally, the `ServletException` exception is thrown by a servlet to indicate a problem with a user request. The problem may be in processing a request or sending of a response.

Whenever the `ServletException` exception is thrown to a Servlet container, an application loses control of the request being processed. It is the responsibility of the Servlet container to clean up the request and return a response to a client. Instead of sending a response, the Servlet container may also return an error page to the client indicating a server problem, depending on implementation and configuration of the container.

A `ServletException` exception should be thrown only as a last resort. The preferred approach to deal with an insuperable problem is to handle the problem and then return the type of the problem to the client.

An application throws the `UnavailableException` exception when a requested resource is not available. The resource can be a servlet, a filter, or any other configuration details required by the servlet to process requests, such as a database, a domain name server, or another servlet.

Exploring the javax.servlet.http Package

The `javax.servlet.http` package contains some interfaces and classes that enhance the basic functionality of a servlet to support HTTP-specific features, such as request and response headers, different request methods, and cookies. As discussed earlier, there are two classes (`GenericServlet` and `HttpServlet`) in the Servlet API, which implement the `Servlet` interface. `HttpServlet` is an abstract class that extends the `GenericServlet` base class and provides a framework to handle the HTTP protocol. The following section discusses the classes and interfaces of the `javax.servlet.http` package.

Explaining the Interfaces of the javax.servlet.http Package

The `javax.servlet.http` package comprises the following eight interfaces:

- HttpServletRequest
- HttpServletResponse
- HttpSession
- HttpSessionBindingListener
- HttpSessionContext
- HttpSessionActivationListener
- HttpSessionAttributeListener
- HttpSessionListener

The `HttpServletRequest` interface retrieves data from a client to a servlet for use in the `HttpServlet.service()` method. This interface allows the `service()` method to access the HTTP protocol specified header information. The `HttpServletResponse` interface allows the `service()` method of a servlet to manipulate the HTTP protocol specified header information. This interface also returns the data to its client. The `HttpSession` interface is used to maintain a session between an HTTP client and the HTTP server. This session is used to maintain the state and user identity across multiple connections or requests during a given period.

The `HttpSessionContext` interface provides a group of the `HttpSessions` objects associated with a single session. The `getSessionContext()` method is used by a servlet to get the `HttpSessionContext` object. The `HttpSessionContext` interface also provides various methods to servlets to list the IDs or retrieve a session based on the ID. The `HttpSessionActivationListener` interface notifies a Web container about the activation or passivation of a session object. The `HttpSessionAttribute` interface is implemented to receive the notifications of changes in the attribute lists of sessions within a Web application. The `HttpSessionListener` interface notifies the changes made in the active sessions in a Web application.

NOTE

The HttpSessionContext interface has been deprecated as of Java™ Servlet API 2.1 for security reasons, with no replacement.

The `HttpSessionBindingListener` interface has the `valueBound()` and `valueUnbound()` methods to notify a listener that it is being bound to a session or unbound from a session.

Next, we discuss the classes of the `javax.servlet.http` package.

Explaining the Classes of the javax.servlet.http Package

Apart from the interfaces, the `javax.servlet.http` package also has various classes, which are as follows:

- Cookie
- HttpServlet
- HttpServletRequestWrapper
- HttpServletResponseWrapper
- HttpSessionBindingEvent
- HttpSessionEvent

- HttpUtils

HttpServlet is an abstract class that simplifies the writing of HTTP servlets. As HttpServlet is an abstract class, servlet programmers must override at least one of the following methods: doGet(), doPost(), doPut(), doDelete() and getServletInfo(). The HttpServletRequestWrapper class provides a convenient implementation of the HttpServletRequest interface to adapt a request to a servlet. Similarly, the HttpServletResponseWrapper class provides a convenient implementation of the HttpServletResponse interface to adapt the response from a servlet. The HttpSessionBindingEvent class communicates to the HttpSessionBindingListener object regarding bounding to or unbounding from the HttpSession value. The HttpSessionEvent class represents event notifications for changes in a session within a Web application. As already discussed, the HttpSession interface maintains a session and manages the session with the help of the Cookie class. The HttpUtils class is a collection of static utility methods useful to HTTP servlets. After learning about the Servlet API, let's discuss the servlet life cycle next.

Introducing the Servlet Life Cycle

Servlets follow a life cycle that governs the multithreaded environment in which the servlets run. It also provides a clear perception about some of the mechanisms available to a developer to share server-side resources. The primary reason why servlets and JavaServer Pages (JSP) outperform traditional CGI is the servlet life cycle. Servlets follow a three-phase life cycle, namely initialization, service, and destruction. This three-phase life cycle is opposed to the single-phase life cycle. Of the three phases, the initialization and destruction phases are performed only once while the service phase is carried out many times.

The first phase of the servlet life cycle is initialization. It represents the creation and initialization of the resources the servlet may need in response to service requests. All servlets must implement the javax.servlet.Servlet interface, which defines the init() method that corresponds to the initialization phase of a servlet life cycle. As soon as a servlet is loaded in a container, the init() method is invoked before servicing any requests.

The second phase of a servlet life cycle is the service phase. This phase of the servlet life cycle represents all the interactions carried out, along with the requests that are handled by the servlet until it is destroyed. The service phase of the servlet life cycle corresponds to the service() method of the Servlet interface. The service() method of a servlet is invoked once for every request. Then, its sole responsibility is to generate the response to that request.

The service() method takes two object parameters, javax.servlet.ServletRequest and javax.servlet.ServletResponse. These two objects represent a request for dynamic resource from a client and a response sent by a servlet to the client, respectively. A servlet is usually multithreaded. This implies that a single instance of a servlet is loaded by a servlet container at a given instance, by default. The initialization of the servlet is done only once, and after that, each request is handled concurrently by threads executing the service() method of the servlet.

The destruction phase is the third and final phase of the servlet life cycle. This phase represents the termination of the servlet execution and its removal from the container. The destruction phase corresponds to the destroy() method of the Servlet interface. The container calls the destroy() method when a servlet is to be removed from the container.

The invocation of the destroy() method enables the servlet to terminate gracefully and clean up any resources held or created by it during execution. To efficiently manage application resources, a servlet should properly use all the three phases of its life cycle. A servlet loads all the required resources during the initialization phase, which may be needed to service client requests. The resources are used during the service phase and then can be given up in the destruction phase.

We have discussed the three events or phases that form the life cycle of a servlet. However, there are many more methods that need to be considered by a Web developer. HTTP is primarily used to access content on the Internet. Though a basic servlet does not know anything about HTTP, a special implementation of the servlet, namely javax.servlet.http.HttpServlet has been specifically designed for this purpose.

When the Servlet container creates a servlet for the first time, the container invokes the `init()` method of the servlet. After this, each user request results in the creation of a thread, which calls the `service()` method of the respective instance. Though the servlet in question can implement a special interface, (`SingleThreadModel`), which stipulates that not only a single thread is permitted to run at a time, but also multiple concurrent requests can be made. The `service()` method then calls the `doGet()`, `doPost()`, or any other `doXXX()` method. However, the calling of the doXXX() method depends on the type of HTTP request received. Finally, when the server decides to unload a servlet, it first calls the servlet's `destroy()` method.

Let's now discuss the various methods used in the life cycle of the servlet.

The *init()* Method

As mentioned earlier, the `init()` method is called when a servlet is created for the first time. It is not called again for other user requests. Therefore, the `init()` method is used only for one-time initializations. A servlet is normally created when a user invokes a URL corresponding to the servlet, for the first time; however, the servlet is loaded on the server when a Servlet container maps the user request to the servlet. The following code snippet shows the `init()` method definition:

```
public void init() throws ServletException {
   // Initialization code...
}
```

Reading server-specific initialization parameters is one of the most common tasks that the `init()` method performs. For example, a servlet might need to know various information details, such as database settings, password files, server-specific performance parameters, hit count files, or serialized cookie data from previous requests.

When you need to read the initialization parameters, you have to first obtain a `ServletConfig` object by using the `getServletConfig()` method, and then call the `getInitParameter()` method on the result. The following code snippet shows how to obtain a `ServletConfig` object:

```
public void init() throws ServletException {
   ServletConfig config = getServletConfig();
   String param1 = config.getInitParameter("parameter1");
}
```

In the preceding code snippet, notice that the `init()` method uses the `getServletConfig()` method to obtain a reference to the `ServletConfig` object. The object has a `getInitParameter()` method, which can be used to look up the initialization parameters associated with the servlet. Similar to the `getParameter()` method used in the `init()` method of applets, both the input (i.e., the name of the parameter) and the output (i.e., the parameter value) are nothing but Strings.

You can read the initialization parameters by calling the `getInitParameter()` method of the `ServletConfig` object. However, setting up these initialization parameters is the job of the `web.xml` file, which is called Deployment Descriptor, which we discuss in the *Understanding Servlet Configuration* section of this chapter.

The *service()* Method

Each time a server receives a request for a servlet, the server spawns a new thread and calls for the `service()` method. It is possible that the server spawns a new thread by reusing an idle thread from a thread pool. The `service()` method verifies the HTTP request type (`GET`, `POST`, `PUT`, `DELETE`) and accordingly calls the `doGet()`, `doPost()`, `doPut()`, `doDelete()` methods. A normal request for a URL or a request from an HTML form that has no `METHOD` specified, results in a `GET` request. Apart from the GET request, an HTML form can also specify `POST` as the request method type. The following code snippet explains the implementation of the POST method:

```
<html>
<form name="greetForm" method="post">
```

Now, if you have a servlet that needs to handle both `POST` and `GET` requests identically, you may be tempted to override the `service()` method directly rather than implementing both the `doGet()` and `doPost()`

methods. However, remember, this is not a good idea. Instead, just you can use the `doPost()` method to call the `doGet()` method (or vice versa), as shown in the following code snippet:

```
@Override
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet code
}
@Override
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
```

In the preceding code snippet, the `@Override` annotation is used. Though this approach takes a couple of extra lines of code, it has several advantages over the approach of directly overriding of the `service()` method. One advantage is that you can add support for other HTTP request methods later by adding the `doPut()`, `doTrace()` methods in a subclass. Another advantage of using this approach is that you can add support to retrieve the date on which modifications on data have been made by adding the `getLastModified()` method. However, overriding the `service()` method eliminates this option because the `getLastModified()` method is invoked by the default `service()` method. Finally, as an added advantage, you can get automatic support for the HEAD, OPTION, and TRACE requests.

NOTE

If a servlet needs to handle both GET and POST identically, the doPost() method should call the doGet() method or vice versa. Remember, you should not override the service() method directly.

During the entire request and response process, most of the time, you only care about the GET or POST requests. Therefore, you override either the `doGet()` method or the `doPost()` method or both. However, if required, you can also override the following methods depending upon the request types:

- The doDelete() method for DELETE requests
- The doPut() method for PUT requests
- The doOptions() method for OPTIONS requests
- The doTrace() method for TRACE requests

Remember, however, that you have automatic support for OPTIONS and TRACE.

The `doHead()` method is not provided in versions 2.1 and 2.2 of the Servlet API, because in those versions the system answers HEAD requests automatically by using the status line and header settings of the `doGet()` method. However, the `doHead()` method is included in version 2.3 to enable the generation of responses to HEAD requests.

The destroy() Method

The destroy () method runs only once during the lifetime of a servlet, and signals the end of the servlet instance. A Servlet container holds a servlet instance till the servlet is active or its `destroy()` method is called. The following code snippet shows the method signature of the `destroy()` method:

```
public void destroy ()
```

As soon as the `destroy()` method is activated, the Servlet container releases the servlet instance.

NOTE

It is not recommended to implement the finalize() method in the servlet object; instead, provide the code for the finalization tasks of an application in the destroy() method.

After learning about the life cycle of a servlet, let's understand servlet configuration.

Configuring Servlet in web.xml

You may sometimes need to provide initial configuration information for a servlet. The configuration information for the servlet may include a `String` or a set of `String` values, listed in the `web.xml` file as initialization parameters required during the initialization of the servlet. Due to this functionality, servlets are allowed to have initial parameters specified outside of the compiled code and changed without requiring recompiling of the servlet. Each servlet has an object associated with it, called `ServletConfig`. This object is created by the container and implements the `javax.servlet.ServletConfig` interface. The `ServletConfig` object contains the initialization parameter and you can retrieve the reference of the `ServletConfig` object by invoking the `getServletConfig()` method. The following code snippet shows the method provided by the `ServletConfig` object to access an initialization parameter:

```
getInitParameter(String name)
```

The `getInitParameter()` method returns a `String` object that contains the value of the named initialization parameter or null, if the parameter does not exist. The following code snippet shows the getInitParameterNames() method of the ServletConfig object:

```
getInitParameterNames()
```

The `getInitParameterNames()` method returns the names of the initialization parameters of a servlet as an enumeration of `String` objects. An empty enumeration is returned by the method if the servlet has no initialization parameters.

A servlet can define initial parameters by using the `init-param`, `param-name`, and `param-value` elements in the `web.xml file`. Each `init-param` element defines one initial parameter. This initial parameter must contain a parameter name and value specified by the `param-name` and `param-value` child elements, respectively. A servlet may have as many initial parameters as needed. However, note that initial parameter information for a specific servlet should be specified within the `<servlet>` element for that particular servlet.

A servlet can be configured with the help of the `web.xml` file, which lies in the `WEB-INF` directory of a Web application. This file controls many behavioral aspects of the servlet and JSP. Many servers provide graphical interfaces that allow you to specify initialization parameters and control various behavioral aspects of servlets and JSP pages.

These graphical interfaces are server-specific. However, these interfaces also use the `web.xml` file, which is completely portable. The `web.xml` file contains an XML header, a `DOCTYPE` declaration, and a `web-app` element. To specify initialization parameters, the `web-app` element must contain a `<servlet>` element with three subelements, which are as follows:

- servlet-name
- servlet-class
- init-param

The <servlet-name> element specifies the name that helps you to access the servlet. The <servlet-class> element specifies the fully qualified (that is, a servlet class name is included with the package name) class name of the servlet, and the `init-param` element specifies names and values for parameter initialization.

The Servlet 2.5 specification has introduced several changes to the web.xml file format to make its use more convenient. For example, in the previous versions of Java Servlet, only one servlet could be bound to a filter at a time, as shown in the following code snippet:

```
<filter-mapping>
  <filter-name>MyFilter</filter-name>
  <servlet-name>MyServlet</servlet-name>
</filter-mapping>
```

In the preceding code snippet, the `MyServlet` servlet is mapped to the `MyFilter` filter; however, in Servlet 2.5, you can bind all servlets at once by using an asterisk. The asterisk is used in the <servlet-name> element to represent all servlets. The following code snippet shows you how to bind all servlets to the `MyFilter` filter at once:

```
<filter-mapping>
  <filter-name>MyFilter</filter-name>
```

```
        <servlet-name>*</servlet-name>
    </filter-mapping>
```

Apart from this, Servlet 2.5 provides support for configuring multiple patterns for a filter. In the earlier versions of Java Servlet, you could use the <filter-mapping> element with just one <url-pattern>element, whereas Servlet 2.5 supports multiple <url-pattern> elements. Consider the following example in which multiple <url-pattern> elements have been provided for the MyServlet servlet:

```
    <servlet-mapping>
        <servlet-name>MyServlet</servlet-name>
        <url-pattern>/servlet/*<url-pattern>
        <url-pattern>/servlet/*<url-pattern>
    </servlet-mapping>
```

Listing 11.1 provides a sample web.xml file, which maps to a single servlet named FirstServlet (you can find this file on the CD in the code\JavaEE\Chapter11\FirstApp\WEB-INF folder):

Listing 11.1: Displaying the Code for the web.xml File

```
    <?xml version="1.0" encoding="UTF-8"?>
    <web-app version="2.5" 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_2_5.xsd">
        <servlet>
            <servlet-name>FirstServlet</servlet-name>
            <servlet-class>FirstServlet</servlet-class>
        </servlet>
        <servlet-mapping>
            <servlet-name>FirstServlet</servlet-name>
            <url-pattern>/FirstServlet</url-pattern>
        </servlet-mapping>
        <session-config>
            <session-timeout>
                30
            </session-timeout>
        </session-config>
    </web-app>
```

In Listing 11.1, the web.xml file contains an XML header, a DOCTYPE declaration, and a web-app element. The <servlet-name> element contains the name of the servlet and <servlet-class> contains the fully qualified class name of the servlet. The <servlet-mapping> element contains two subelements, <servlet-name> and <url-pattern>. The <servlet-name> element contains the name of the servlet as provided in the <servlet> element.

Before you learn to create a servlet, its important to understand the ServletConfig and ServletContext interfaces. So let's explore these interfaces in the next section.

Working with ServletConfig and ServletContext Objects

ServletContext objects help to provide context information in a Servlet container. A ServletContext object is used to communicate with the Servlet container while ServletConfig, which is a servlet configuration object, is passed to the servlet by the container when the servlet is initialized. A ServletConfig object contains a ServletContext object, which specifies the parameters for a particular servlet while the ServletContext object specifies the parameters for an entire Web application. The ServletContext object parameters are available to all the other servlets in that application.

The following code snippet sets an attribute named name with the value Pallavi Sharma:

```
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        config.getServletContext().setAttribute("name", "Pallavi Sharma");
    }
```

In the preceding code snippet, the call to the super.init(config) method ensures that the GenericServlet class receives a reference to the ServletConfig object. The implementation of the

GenericServlet class maintains a reference to the ServletConfig object and requires the invocation of the super.init(config) method in subclasses.

In the preceding code snippet, config is the instance of ServletConfig passed as an argument to the init() method. The setAttribute() method is used to set the value of the name attribute. The value of this attribute is accessed with the help of the getAttribute() method. This attribute is available for all the servlets in the Web application and can be accessed in any servlet.

Now, let's learn how to create a servlet.

Creating a Simple Servlet

Let's create a simple servlet in the FirstApp application, which handles HTTP request and sets the value of the name attribute at the initialization of the servlet, which is displayed on the browser. Moreover, the value of the init-param, greeting, is set in the web.xml file. Listing 11.2 provides the code for FirstServlet servlet (you can find this file on the CD in the code\JavaEE\Chapter11\FirstApp\src\com\kogent folder):

Listing 11.2: Showing the Code for the FirstServlet.java File

```
package com.kogent;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FirstServlet extends HttpServlet {
  @Override
  public void init(ServletConfig config) throws ServletException {
        super.init(config);
        config.getServletContext().setAttribute("name", "Pallavi Sharma");
  }
  @Override
 public void doGet(HttpServletRequest request, HttpServletResponse response) throws
   IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter printwrite = response.getWriter();
        printwrite.println("<html>");
        printwrite.println("<head>");
        String greet;
        String name;
        greet = getServletConfig().getInitParameter("greeting");
        name=getServletContext().getAttribute("name").toString();
        printwrite.println("<title>"+greet+"</title>");
        printwrite.println("</head>");
        printwrite.println("<body>");
        printwrite.println("<h1>"+greet+"</h1>");
        printwrite.println("<h2>"+name+"</h2>");
        printwrite.println("</body>");
        printwrite.println("</html>");
  }
 }
```

The FirstServlet servlet handles the HttpRequest object and so the object is extended with the HttpServlet class. The FirstServlet servlet overrides the init() and doGet() methods. The value for the name attribute is set in the init() method. The doGet() method retrieves the value of the greeting initializing parameter, which will be set in the web.xml file. The doGet() method also displays the value of the name attribute.

In Listing 11.2, the getServletContext() method of the ServletConfig object calls the setAttribute() method, to set the value and the getAttribute() method to retrieve the value of the name attribute. The getInitParameter() method is used to retrieve the value of init-param, which will be set in the web.xml file (Listing 11.3).

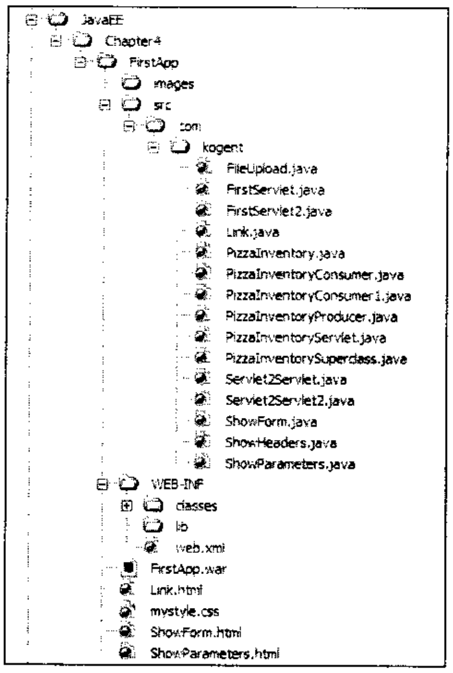
Create a JavaEE folder in the C: drive for the applications that you create in all the chapters of this book. This folder can be found on the CD as well.

Now, let's define directory structure for the `FirstApp` application to store the `FirstServlet` servlet, configure the servlet in the `web.xml` file, and then package, deploy, and run the `FirstApp` application.

Creating Directory Structure

The root directory for all the applications in this book is JavaEE and you will find a folder for each chapter in the CD under this root directory. To run an application on your system, you can either copy the JavaEE folder from CD to the C: drive or create a new JavaEE folder containing the folders for each chapter. For example, for this chapter, the `Chapter11` folder is created under the root directory, JavaEE.

Figure 11.6 shows the directory structure of the FirstApp Web application:



```
JavaEE
  Chapter4
    FirstApp
      images
      src
        com
          kogent
            FileUpload.java
            FirstServlet.java
            FirstServlet2.java
            Link.java
            PizzaInventory.java
            PizzaInventoryConsumer.java
            PizzaInventoryConsumer1.java
            PizzaInventoryProducer.java
            PizzaInventoryServlet.java
            PizzaInventorySuperclass.java
            Servlet2Servlet.java
            Servlet2Servlet2.java
            ShowForm.java
            ShowHeaders.java
            ShowParameters.java
      WEB-INF
        classes
        lib
        web.xml
      FirstApp.war
      Link.html
      mystyle.css
      ShowForm.html
      ShowParameters.html
```

Figure 11.6: Displaying the Directory Structure of the FirstApp Application

In Figure 11.6, FirstApp is the name of the created application. Run the following command from the `C:\JavaEE\Chapter11\FirstApp\src\com\kogent` location to compile the `FirstServlet` servlet:

```
javac -d C:\JavaEE\Chapter11\FirstApp\WEB-INF\classes FirstServlet.java
```

The preceding command compiles the `FirstServlet.java` file and creates the `com.kogent` package that contains the class file of `FirstServlet`. The `com.kogent` package is created under the `classes` folder (Figure 11.6).

Configuring the Servlet

Configuration implies mapping a servlet and providing the initialization parameter values for it. Servlet configuration for any Web application is done in the `web.xml` file. Listing 11.3 shows how to configure the `FirstServlet` servlet (you can find this file on CD in the `code\JavaEE\Chapter11\FirstApp\WEB-INF` folder):

Listing 11.3: Showing the Code for the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" 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_2_5.xsd">
```

```
    <servlet>
        <servlet-name>FirstServlet</servlet-name>
        <servlet-class>com.kogent.FirstServlet</servlet-class>
        <init-param>
            <param-name>greeting</param-name>
            <param-value>Welcome</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>FirstServlet</servlet-name>
        <url-pattern>/FirstServlet</url-pattern>
        <url-pattern>/FServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

The `web.xml` file is saved under the `WEB-INF` folder of the `FirstApp` application (Figure 11.6). In Listing 11.3, the `FirstServlet` servlet is mapped to two url-patterns (`/FirstServlet` and `/FServlet`). In Servlet 2.5, you can provide multiple url patterns for a servlet in the `web.xml` file. In Listing 11.3, the `greeting` initialization parameter is provided `Welcome` as its value. The `FirstServlet` servlet is configured to the `com.kogent.FirstServlet` class.

Packaging, Deploying and Running the Web Application

Before deploying the FirstApp Web application, a Web ARchive (WAR) file is created to package the entire Web application. Further, you need o deploy the application on the Glassfish application server and finally you can run the application. Perform the following steps to package, deploy, and run the FirstApp application.

- Run the following command from the `code\JavaEE\Chapter11\FirstApp` location to create the `FirstApp.war` file:

```
jar -cvf FirstApp.war
```

The preceding command creates the `FirstApp.war` file in the `FirstApp` folder (Figure 11.6). The WAR file contains the entire directory structure shown in Figure 11.6.

- Start the Glassfish application server and open the http://localhost:4848/ URL. The Login window appears.
- Enter admin as the user name and adminadmin as the password. After logging to the Web application, the Index page of the application is displayed.
- Select the Web Applications option under Applications in the directory tree on the left side of the index page, to deploy the FirstApp WAR file, as shown in Figure 11.7:

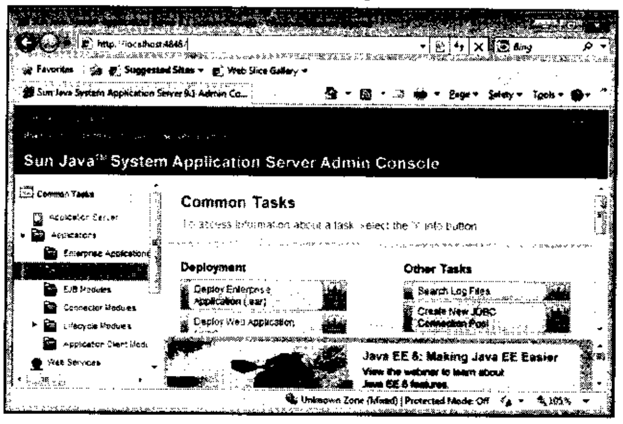

http://localhost:4848/
Favorites
Suggested Sites
Web Slice Gallery
Sun Java System Application Server 9.1 Admin Co...
Page
Safety
Tools
Sun Java™ System Application Server Admin Console
Common Tasks
Application Server
Applications
Enterprise Applications
EJB Modules
Connector Modules
Lifecycle Modules
Application Client Modu
Web Services
Common Tasks
To access information about a task select the info button
Deployment
Deploy Enterprise Application (.ear)
Deploy Web Application
Other Tasks
Search Log Files
Create New JDBC Connection Pool
Java EE 6: Making Java EE Easier
View the webinar to learn about Java EE 6 features
Unknown Zone (Mixed) | Protected Mode: Off
105%

Figure 11.7: Displaying the Admin Console of the Glassfish Server

After selecting the `Web Applications` option, the Web Applications pane is displayed (Figure 11.8).

- Click the `Deploy` button, as shown in Figure 11.8:

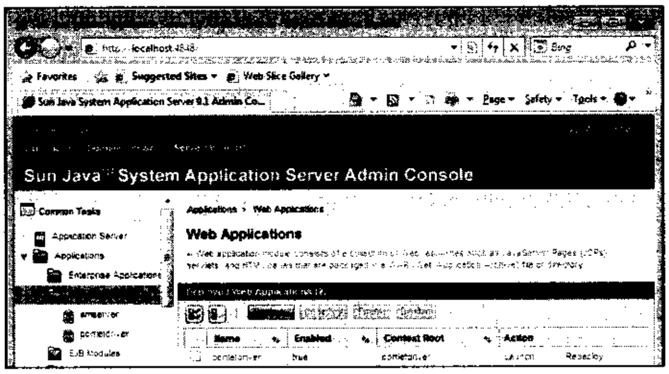


Figure 11.8: Displaying the Web Applications Window

The `Deploy Enterprise Applications/Modules` pane is displayed (Figure 11.9).

- Click the `Browse` button and locate the `FirstApp.war` file. The location for the `FirstApp.war` file to be deployed is shown in Figure 11.9:

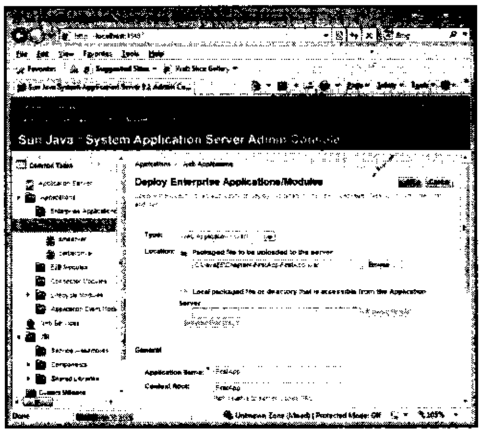
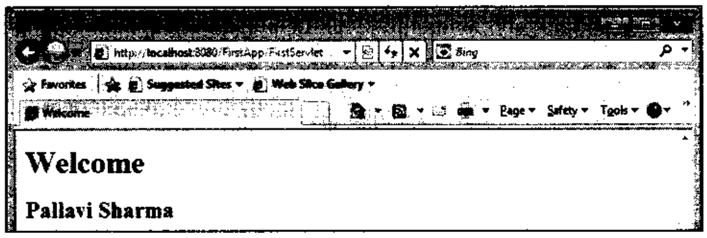

Sun Java System Application Server Admin Console
Deploy Enterprise Applications/Modules
Type:
Location:
Packaged file to be uploaded to the server
Browse...
Local packaged file or directory that is accessible from the Application Server
General
Application Name:
Context Root:

Figure 11.9: Displaying the Details of the Web Application to Deploy

The `FirstApp.war` file is uploaded and the general information of the `FirstApp` Web application automatically updated in the required text boxes of the `Deploy Enterprise Applications/Modules` pane.

- Click the `OK` button to deploy the `FirstApp` Web application, packaged into the WAR file. After the application is deployed, you need to run the application to display the output.
- Open the `http://localhost:8080/FirstApp/FirstServlet` URL. Figure 11.10 displays the output of the `FirstApp` Web application:



Welcome

Pallavi Sharma

Figure 11.10: Displaying the Output of the FirstApp Web Application

Alternatively, you can specify the URL pattern `/FServlet` in the address bar of the Web browser to access the `FirstServlet` servlet. The `http://localhost:8080/FirstApp/FServlet` URL also displays the same output (Figure 11.10).

After creating and configuring a simple servlet by using `Deployment Descriptor` (the `web.xml` file), let's learn about the HttpServletRequest and HttpServletResponse interfaces of the Java Servlet API.

Working with the HttpServletRequest and HttpServletResponse Interfaces

We discussed earlier in this chapter that the most common HTTP requests are the `GET` and the `POST` methods. You must implement these methods to handle different types of requests. The servlet container recognizes the type of HTTP request made and passes the request to the correct servlet method. Accordingly, you do not override the `service()` methods as you do for Servlets that extend the `GenericServlet` class; rather, you override the appropriate request methods.

Let's now learn about the `HttpServletRequest` and `HttpServletResponse` interfaces in detail in the following sections.

HttpServletRequest Interface

An `HttpServletRequest` object always represents a client's HTTP request. `HttpServletRequest` is an interface and a subtype of the `ServletRequest` interface. The Web container provider implements this interface to encapsulate all HTTP-based request information, including headers and request methods.

All properties, such as request parameters and attributes of the `ServletRequest` interface are also accessible through the `HttpServletRequest` interface.

Let's learn about the implementation of the HttpServletRequest interface under the following subheads:

- The role of form data
- Form data and parameters
- Headers
- File uploads

The Role of Form Data

You can understand the role of Form Data better by considering a real-life scenario. When you use a search engine, visit an online bookstore, track stocks on the Internet, or ask a Web-based site for quotes on plane tickets, you may have seen funny-looking URLs such as `http://host/path?user=John+Smith&origin=lon&dest=par`. The part of the URL after the question mark (i.e., `user=John+Smith&origin=lon&dest=par`) is known as Form Data (or query data). This is the most common way by which a server-side program gets information from a Web page. For `GET` requests, Form Data can be attached to the end of the URL after a question mark (as in the proceeding example). For `POST` requests, Form Data can be sent to the server separately.

CGI programming involves a tedious traditional approach of extracting the needed information from Form Data. First, you have to read the data one way for `GET` requests (in traditional CGI, this is usually through the `QUERY_STRING` environment variable) and in a different way for `POST` requests (by reading the standard input in traditional CGI). Second, you have to separate the pairs at the ampersands and then separate the parameter names (left of the equal signs) from the parameter values (right of the equal signs). Third, you have the URL-decode values. All the alphanumeric characters are sent unchanged, but spaces are replaced with plus signs and other characters are replaced with `%XX`, where `XX` implies the American Standard Code for Information Interchange (ASCII) or International Organization for Standardization (ISO) Latin-1 value of the character. The process is reversed for the server-side program. For example, if a user enters the values `~jim`, `~robert`, and `~hall` into text fields with the name users in an HTML form, the data is sent as `users=%7Ejim%2C+%7Erobert%2C+and+%7Ehall`, and the original string is reconstituted by the server-side program. Finally, the fourth reason that makes parsing Form Data in CGI programs a tedious process is that values can be omitted (for example `param1=val1&param2=&param3=val3`) or a parameter can have more than one value (for example param1=val1&param2=val2&param1=val3). Therefore, special cases need to be applied in your parsing code for these situations.

Form Data and Parameters

One of the important features of servlets is that parsing of a form is handled automatically. You only need to call the getParameter() method of the HttpServletRequest object with the case-sensitive parameter name as an argument. The getParameter() method is used in the same way when data is sent by the GET request as when it is sent by the POST request. The servlet can identify which request method is used and automatically returns the String value according to the URL-decoded value of the first occurrence of that parameter name. If the parameter exists but has no value, an empty String is returned. In addition, if no such parameter exists, null is returned. You should call the getParameterValues () method (which returns an array of Strings) instead of the getParameters() method (which returns a single String), if the parameter can potentially have more than one value. The return value of the getParameterValues () method is null for parameter names that do not exist. A single element array is returned when only a single value exists for the parameter.

Parameter names are case-sensitive; therefore, for example, request.getParameter("Param1") and request.getParameter("param1") are not interchangeable.

Finally, for debugging, it is sometimes useful to get a full list of parameters, although most real servlets look for a specific set of parameter names. You can use the getParameterNames() method to get the list of parameter names in the form of an enumeration, each entry of which can be cast to a String and used in the getParameter() or getParameterValues() method. You should note that the order in which the parameter names appear within the enumeration is not specified by the HttpServletRequest interface.

Let's now discuss the role of request parameters. Perhaps the most commonly used methods of the HttpServletRequest object are those that involve getting request parameters: getParameter() and getParameters(). Whenever an HTML form is filled and sent to a server, the fields of the form are passed as parameters. This includes any information sent through input fields, selection lists, combo boxes, check boxes, and hidden fields. However, the form submission excludes file uploads. Any information passed as a query string is also available on the server-side as a request parameter. The HttpServletRequest object includes the following methods to access request parameters:

- getParameter(java.lang.String parameterName)–Takes a parameter name as a parameter and returns a String object representing the corresponding value. This method returns null when it does not find a parameter of the given name.
- getParameters(java.lang.String parameterName)–Allows you to get all the parameter values for the same parameter name returned as an array of Strings. The getParameters() method is similar to the getParameter() method. However, note that the getParameters() method should be used when there are multiple parameters with the same name. Often, an HTML form check box or combo box sends multiple values for the same parameter name.
- getParameterNames()–Returns the parameter names in the form of an enumeration, which are used in a request. This method can be used with the getParameter() and getParameters() methods, to obtain a list of names and values of all the parameters included with a request.

Let's now create a servlet that reads and displays all the parameters sent with a request. You can use such a servlet to get a little more familiar with parameters, and to debug HTML forms by seeing the information being sent. Listing 11.4 provides the code for such a servlet (you can find the ShowParameters.java file on CD in the code\JavaEE\Chapter11\FirstApp\src\com\kogent folder):

Listing 11.4: Displaying the Code for the ShowParameters.java File

```
package com.kogent;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShowParameters extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
  response) throws IOException, ServletException {
```

```
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Request HTTP Parameters Sent</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<p>Parameters sent with request:</p>");
            Enumeration enm = request.getParameterNames();
            while (enm.hasMoreElements()) {
                String pName = (String) enm.nextElement();
                String[] pValues = request.getParameterValues(pName);
                out.print("<b>"+pName + "</b>: ");
                for (int i=0;i<pValues.length;i++) {
                    out.print(pValues[i]);
                }
                out.print("<br>");
            }
            out.println("</body>");
            out.println("</html>");
        }
        public void doPost(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {
            doGet(request, response);
        }
    }
```

Save the `ShowParameters.java` file in the `src\com\kogent` folder (Figure 11.6) and compile and deploy the `ShowParameters` servlet in the `FirstApp` Web application with a mapping to the `/ShowParameters` path. Now, create a few simple HTML forms and use the servlet to see the parameters being sent. In Listing 11.5, `ShowParameters.html` provides a sample HTML form (you can find this file on the CD in the `code/JavaEE/Chapter11/FirstApp` folder):

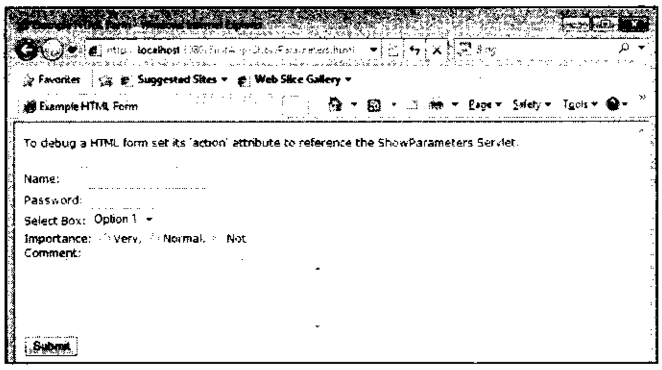
Listing 11.5: Showing the Code for the ShowParameters.html File

```
<html>
<head>
  <title>Example HTML Form</title>
 <link rel="stylesheet" href="mystyle.css" type="text/css"/>
</head>
<body>
  <p>To debug a HTML form set its 'action' attribute to reference the
  ShowParameters Servlet.</p>
  <form action="ShowParameters" method="post">
        Name: <input type="text" name="name"><br>
        Password: <input type="password" name="password"><br>
        Select Box:
              <select name="selectbox">
              <option value="option1">Option 1</option>
              <option value="option2">Option 2</option>
              <option value="option3">Option 3</option>
        </select><br>
        Importance:
        <input type="radio" name="importance" value="very">Very,
        <input type="radio" name="importance" value="normal">Normal,
        <input type="radio" name="importance" value="not">Not<br>
        Comment: <br>
        <textarea name="textarea" cols="40" rows="5"></textarea><br>
        <input value="Submit" type="submit">
  </form>
</body>
</html>
```

Save the `ShowParameters.html` file in the base directory of the `FirstApp` Web application and deploy the new WAR file. After deploying the file, browse the following URL:

http://localhost:8080/FirstApp/ShowParameters.html

The ShowParameters HTML page is displayed, as shown in Figure 11.11:

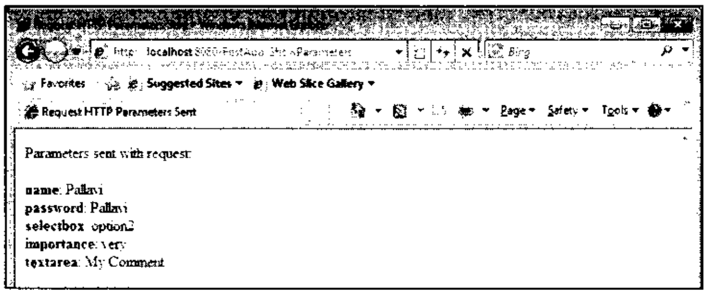


To debug a HTML form set its 'action' attribute to reference the ShowParameters Servlet.

Name:
Password:
Select Box: Option 1
Importance: Very, Normal, Not
Comment:

Figure 11.11: Displaying the ShowParameters Page

After entering the relevant details in the HTML form (Figure 11.11), the parameters are sent to the `ShowParameters` servlet. The parameters sent from the HTML page appear, as shown in Figure 11.12:



Parameters sent with request:

name: Pallavi
password: Pallavi
selectbox: option2
importance: very
textarea: My Comment

Figure 11.12: Displaying the Request HTTP Parameters

On the server-side, each piece of information received from an HTML form is referenced by the same name as defined in the HTML form and is linked to the value that a user has entered for the respective field. The `ShowParameters` servlet calls the `getParameterNames()` method to retrieve a list of all the parameter names and subsequently calls the `getParameters()` method to retrieve the matching value or set of values for each name. The following code snippet shows the segment of code shown in Listing 11.4 that retrieves the list of parameters along with their respective values:

```
Enumeration enm = request.getParameterNames();
while (enm.hasMoreElements()) {
   String pName = (String) enm.nextElement();
   String[] pValues = request.getParameterValues(pName);
   out.print("<b>"+pName + "</b>: ");
   for (int i=0;i<pValues.length;i++) { out.print(pValues[i]);}
   out.print("<br>");
}
```

A servlet can fetch information from HTML clients by using parameters. The `ShowParameters` servlet only takes the parameters from the HTML page and displays them back to a client. However, normally, these parameter values are combined and processed with other code to generate responses. Later on in the book, you learn to use this functionality with servlets and JSP to interact with clients, including sending e-mail messages and user authentication.

HTTP Headers

HTTP headers are set by a client to give information to a server about software that the client is using and how the client wants a server to send back the requested information. HTTP request headers can be accessed from a servlet by calling different methods, which are as follows:

- `getHeader(java.lang.String name)`–Returns the specified request header value as a `String`. This method returns null if the request does not include a header of the specified name. The header name is case insensitive. A user can use this method with any request header.
- `getHeaders(java.lang.String name)`–Returns all the specified request header values as an enumeration of `String` objects. Sometimes, a client can send the header values as an enumeration of String objects, rather than sending the header as a comma-separated list. Each of these headers can have a different value. If the request includes no header of the specified name, this method returns an empty Enumeration object. The header name is case insensitive in this method also as well. The user can use this method with any request header.
- `getHeaderNames()`–Returns an enumeration of the names of all the headers sent by a request. In combination with the `getHeader()` and `getHeaders()` methods, the `getHeaderNames()` method can be used to retrieve the names and values of all the headers sent with a request. Some containers do not allow access to HTTP headers. In that case, null is returned.
- `getIntHeader(java.lang.String name)`–Returns the value of the specified request header as an int type. A value of -1 is returned by this method if the request does not contain a header of the specified name. A `NumberFormatException` exception is thrown if the header cannot be converted to an integer.
- `getDateHeader(java.lang.String name)`–Returns the specified request header value as a long value representing a `Date` object. The returned date is counted as the number of milliseconds since the epoch. The header name is case insensitive. A value of -1 is returned if a request header of the specified name is not found. An `IllegalArgumentException` exception is thrown if the header cannot be converted to a date.

In this way, you can see that HTTP request headers are very helpful to determine diversified information, which can be obtained by calling the preceding listed methods. In the later chapters of the book, HTTP request headers are used as the primary resource to mine data about a client. This includes identifying what language a client would prefer, what type of Web browser is being used, and whether or not the client can support compressed content for efficiency. For now, it is helpful to understand that these headers exist, and to get a general idea about what type of information the headers contain. Listing 11.6 creates a servlet designed to do just that. Save the code of the `ShowHeaders.java` file in the `/src/com/kogent` directory of the FirstApp Web application, created in Figure 11.6. Listing 11.6 shows the code for the `ShowHeaders` class (you can find this file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

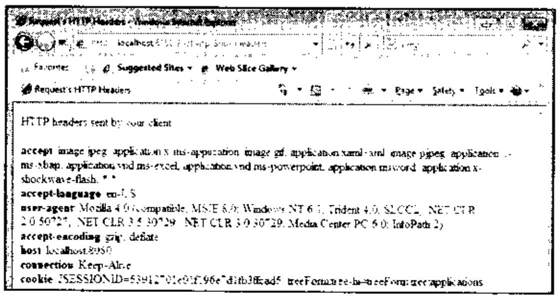
Listing 11.6: Displaying the Code for the ShowHeaders.java File

```
package com.kogent;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShowHeaders extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
  response) throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Request's HTTP Headers</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>HTTP headers sent by your client:</p>");
        Enumeration enm = request.getHeaderNames();
        while (enm.hasMoreElements()) {
                String headerName = (String) enm.nextElement();
                String headerValue = request.getHeader(headerName);
                out.print("<b>"+headerName + "</b>: ");
                out.println(headerValue + "<br>");
        }
        out.println("</body>");
        out.println("</html>");
  }
}
```

Compile the ShowHeaders servlet and configure the servlet to the `/ShowHeaders` path in the web.xml file. The following code snippet provides the configuration of the ShowHeaders servlet in the web.xml file:

```
...
<servlet>
   <servlet-name>ShowHeaders</servlet-name>
   <servlet-class>com.kogent.ShowHeaders</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>ShowHeaders</servlet-name>
   <url-pattern>/ShowHeaders</url-pattern>
</servlet-mapping>
```

Deploy the FirstApp Web application and after reloading the Web application, browse to `http://localhost:8080/FirstApp/ShowHeaders` to view a listing of all the HTTP headers sent by your browser. Figure 11.13 shows the details of the HTTP headers:



```
HTTP headers sent by your client

accept image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, application/x-shockwave-flash, */*
accept-language en-US
user-agent Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; InfoPath.2)
accept-encoding gzip, deflate
host localhost:8080
connection Keep-Alive
cookie JSESSIONID=53912701e01f196e7d3fb3ffead5 treeForm:tree-hi=treeForm:tree:applications
```

Figure 11.13: Displaying the Request's HTTP Headers

Listing 11.6 is a good example to illustrate how headers are normally sent by a Web browser. They are self-descriptive and can therefore be understood easily. You can probably imagine how these headers can be used to infer browser and internationalization information. Table 11.1 lists some of the most relevant request headers:

Table 11.1: Various Request Headers

Request header	Description
`Accept`	Specifies certain media types that can be accepted for a response. Accept headers can be used to specify that the request is limited to a set of desired media types.
`Accept-Charset`	Indicates the character sets that can be accepted for a response. This header accepts clients that can understand comprehensive or special-purpose character sets. Due to this, the server can represent responses in those character sets. All user agents can accept the ISO-8859-1 character set.
Referer (sic)	Allows a client to state the address (URI) of the resource from which the Request URI was obtained.
`Accept-Language`	Restricts the set of natural languages that are preferred as a response to a request. Otherwise, this header is similar to the Accept -Character header.
`Host`	Specifies the Internet host and port number of a requested resource, as obtained from the original URL given by a user or referring resource. This header is mandatory for HTTP 1.1.
`User-Agent`	Contains information about the user agent (or browser) making a request. This header is used for statistical purposes, to trace violations of protocol, and to automatically recognize user agents.

File Uploads

File uploads are simple for HTML developers but difficult for server-side developers. Usually, discussions on Servlets and HTML forms conveniently skip the topic of file uploads. However, a servlet developer must have a good understanding of HTML form file uploads. For example, consider a situation where a client needs to upload something besides a simple string of text, such as a picture. In this case, using the `getParameter()`

method will not work because it produces unpredictable results. Therefore, to read the uploaded picture, the Servlet API provides the MIME type.

There are two primary MIME types for form information, `application/x-www-form-urlencoded` and `multipart/form-data`. In the MIME type `application/x-www-form-urlencoded`, the results in the Servlet API automatically parse out name and value pairs. The information is then available by invoking `HttpServletRequest getParameter()` or any of the other related methods as described earlier. The second MIME type, `multipart/form-data`, is usually considered difficult, because the Servlet API does not provide any support for it. The information is left as it is and you are responsible for parsing the request body by using either the `getInputStream()` or `getReader()` method.

The Request For Comments (RFC) 1867 memo provided at the `http://www.ietf.org/rfc/rfc1867.txt` URL explains the `multipart/form-data` MIME type and the format of the associated HTTP requests. You can determine how to properly and appropriately handle the information posted to a servlet, by using RFC. This is not a difficult task and is usually not needed, because other developers create complementary APIs to handle file uploads. We discuss this later on in this chapter.

A user can actually look at the content of a request when the `multipart/form-data` information is sent. For this, create a file-uploading form and a servlet that reproduces the information obtained from the `ServletInputStream` object. Listing 11.7 provides the code for such a servlet that accepts a `multipart/form-data` request and displays its content as plain text (you can find the ShowForm.java file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

Listing 11.7: Displaying the Code for the ShowForm.java File

```
package com.kogent;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShowForm extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse
  response) throws IOException, ServletException {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();
        ServletInputStream sis = request.getInputStream();
        for (int i = sis.read(); i != -1; i = sis.read()) {
                out.print((char)i);
        }
  }
  public void doGet(HttpServletRequest request, HttpServletResponse
  response) throws IOException, ServletException {
                doPost(request, response);
  }
}
```

Save the preceding code as `ShowForm.java` in the `/src/com/kogent` directory of the `FirstApp` Web application. Configure the `ShowForm` servlet to `/ShowForm` in the `web.xml` file by using the following code snippet:

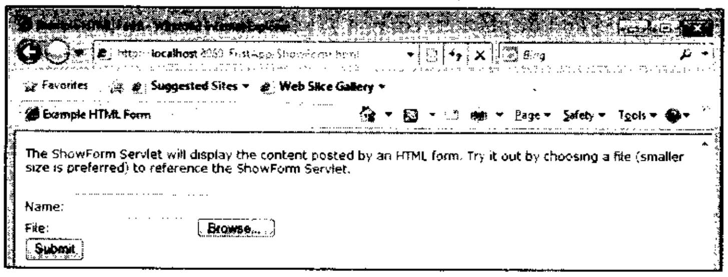
```
...
<servlet>
  <servlet-name>ShowForm</servlet-name>
  <servlet-class>com.kogent.ShowForm</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>ShowForm</servlet-name>
  <url-pattern>/ShowForm</url-pattern>
</servlet-mapping>
```

Now, any information posted by any form can be viewed by directing the request to `http://localhost:8080/FirstApp/ShowForm`. You can run the `ShowForm` servlet only after creating an HTML form used to upload a file. Listing 11.8 provides the code to create an HTML page (you can find the `ShowForm.html` file on the CD in the `code\JavaEE\Chapter11\FirstApp` folder):

Listing 11.8: Displaying the Code for the ShowForm.html File

```
<html>
  <head>
      <title>Example HTML Form</title>
      <link rel="stylesheet" href="mystyle.css" type="text/css"/>
  </head>
  <body>
      <p>The ShowForm Servlet will display the content posted by an HTML
      form. Try it out by choosing a file (smaller size is preferred) to
      reference the ShowForms Servlet.</p>
      <form action="ShowForm" method="post"
            enctype="multipart/form-data">
            Name: <input type="text" name="name"><br>
            File: <input type="file" name="file"><br>
            <input value="Submit" type="submit">
      </form>
  </body>
</html>
```

Save the code shown in Listing 11.8 as `ShowForm.html` in the base directory of the `FirstApp` Web application and browse the `http://localhost:8080/FirstApp/ShowForm.html` URL. A small HTML form with two inputs, a name and a file to upload is displayed, as shown in Figure 11.14:

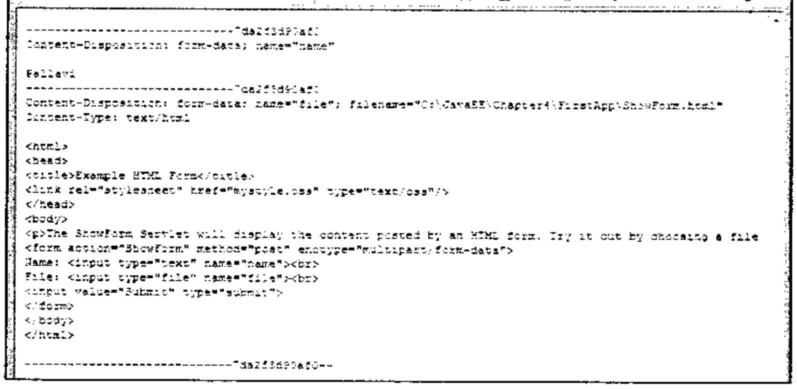


The ShowForm Servlet will display the content posted by an HTML form. Try it out by choosing a file (smaller size is preferred) to reference the ShowForm Servlet.

Name:
File: Browse...
Submit

Figure 11.14: Displaying the ShowForm Page

Enter appropriate values for the `Name` and `File` fields in the form (Figure 11.14). In this case, a small file is preferred because its content is going to be displayed by the `ShowForm` Servlet. A good candidate for a file to be uploaded is `ShowForm.html`. After entering the values, click the `Submit` button to show the content of the uploaded file. For example, using `ShowForm.html` as the file displays a form similar to the one shown in Figure 11.15:



```
-----------------------------7da2f3d97af0
Content-Disposition: form-data; name="name"

Pallavi
-----------------------------7da2f3d97af0
Content-Disposition: form-data; name="file"; filename="C:\JavaEE\Chapter4\FirstApp\ShowForm.html"
Content-Type: text/html

<html>
<head>
<title>Example HTML Form</title>
<link rel="stylesheet" href="mystyle.css" type="text/css"/>
</head>
<body>
<p>The ShowForm Servlet will display the content posted by an HTML form. Try it out by choosing a file
<form action="ShowForm" method="post" enctype="multipart/form-data">
Name: <input type="text" name="name"><br>
File: <input type="file" name="file"><br>
<input value="Submit" type="submit">
</form>
</body>
</html>

-----------------------------7da2f3d97af0--
```

Figure 11.15: Displaying the Output of the ShowForm.html File

The following line is displayed in Figure 11.15 depicting the unique token for the start portion of the multipart of the uploaded file:

```
-----------------------------7da2f3d90af0
```

The preceding line declares the start of the multipart section and concludes at the ending token, which is identical to the start but has -- appended to it. Between the starting and ending tokens are sections of data (possibly nested multiparts) with headers used to describe the content. For example, the Content-Disposition header, which describes a form parameter, is displayed as follows:

```
Content-Disposition: form-data; name="name"
Pallavi
```

The Content-Disposition header defines the information as part of the form and is identified by the name "name". The value of name is the content that follows it. By default, the MIME type of name is text/plain. The uploaded file is described in the second multipart, shown as follows:

```
Content-Disposition:form-data;name="file";
  filename="C:\JavaEE\Chapter11\FirstApp\ShowForm.html"
Content-Type: text/html
<html>
  <head>
        <title>Example HTML Form</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
  </head>
  <body>
        <p>The ShowForm
. . .
```

In the preceding code snippet, the Content-Disposition header specifies the form field name to be filled, which corresponds to the field name listed in the HTML form, and describes Content-Type as text/html, as it is not text/plain. The output after the Content-Type header includes code of the `ShowForm.html` file.

So far, you have learned how the `HttpServletRequest` object is used to retrieve HTML form parameters from a request by using the `getParameter()` method. The `HttpServletRequest` object is also used to retrieve HTTP request header information, upload a file by using the `getInputStream()` method, and display the content of the uploaded file by using the `getWriter()` method of the `HttpServletResponse` object.

After discussing the implementation of the `HttpServletRequest` interface, let's learn about the `HttpServletResponse` interface.

Using the HttpServletResponse Interface

The `HttpServletResponse` object helps to set an HTTP response header, set the content type of the response, or redirect an HTTP request to another URL. Let's discuss the implementation of the `HttpServletResponse` interface.

In the previous section, we discussed how to send information back to a client. The `HttpServletResponse` object is responsible for this functionality, which creates an empty HTTP response. Custom content can be sent back by obtaining an output stream by using either the `getWriter()` or `getOutputStream()` method to write the content. A suitable object is returned by these two methods to send either text or binary content to a client. Only one of the two methods can be used with a given `HttpServletResponse` object. An exception is thrown if you try calling both the methods. The `HttpServletResponse` object is also used to provide a `PrintWriter` instance to print the response on a browser. The following code snippet displays the welcome message on the browser:

```
PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Welcome Message</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>Welcome to the Users</p>");
```

In the preceding code snippet, the `getWriter()` method is used to get an output stream to send the HTML markup. When you use an instance of the `PrintWriter` object, you need to provide a `String` object and call the `print()`, `println()`, or `write()` method.

In this section, you learn about the implementation of the `HttpServletResponse` interface under the following heads:

- Response header
- Response redirection
- Response redirection translation issues
- Auto-refresh/wait pages

Response Header

The `HttpServletResponse` object is used to manipulate the HTTP headers of a response and to send the content back to a client. HTTP response headers inform a client the type and amount of content being sent, and the type of server sending the content.

The `HttpServletResponse` object includes the following methods to manipulate HTTP response headers:

- `addHeader(java.lang.String name, java.lang.String value)`–Adds a response header having the given name and value. This method can be used to create response headers with multiple values.
- `containsHeader(java.lang.String name)`–Returns a `Boolean` value that indicates whether the named response header is already been set or not.
- `setHeader(java.lang.String name, java.lang.String value)`–Sets the name and value of a response header as specified in the arguments. The previous value is overwritten, if the header is already set. The `containsHeader()` method can be used to test whether a header is already present or not, before setting its value.
- `setIntHeader(java.lang.String name, int value)`–Sets the name and integer values for a response header, as specified in the arguments. The previous value is overwritten, if the header is already set. The `containsHeader()` method can be used to test whether a header is already present or not, before setting its value.
- `setDateHeader(java.lang.String name, long date)`–Sets the given name and date values for a response header. The date is provided in terms of milliseconds since the epoch. The previous value is overwritten with the new value, if the header is already set.
- `addIntHeader(java.lang.String name, int value)`–Adds a response header having the name and integer values, as specified in the arguments. Response headers can be assigned multiple values when created by using this method.
- `addDateHeader(java.lang.String name, long date)`–Adds a response header having the name and date values, as specified in the arguments. The previous response header values are not overwritten and response headers can have multiple values.

Table 11.2 provides a description of the HTTP response header fields and their values:

Table 11.2: Response Header Fields and their Values

[illegible]	[illegible] Value
`Age`	Represents the estimated time since the last response generated from the server. The value of this header field is usually a positive integer.
`Content-Length`	Indicates the size of the message body, in decimal number of octets (8-bit bytes), sent to a recipient.
`Content-Type`	Refers to the MIME type corresponding to the content of an HTTP response. A browser can use this value to determine whether the content is rendered internally or launched to be rendered by an external application.
`Date`	Represents the date and time at which a message originated.

Table 11.2: Response Header Fields and their Values

Header Field	Header Value
`Location`	Specifies the location of a new resource in case HTTP response codes redirect a client to such a resource. The location is specified as an absolute address.
`Pragma`	Specifies the implementation-specific directives that may be applied to any recipient along the request-response chain. *No-cache*, which indicates that a resource should not be cached, is the most commonly used value.
`Retry-After`	Indicates the tentative duration for which a service is unavailable to a requesting client. It is used with a 503 (Service Unavailable) response. A date can be returned as a value of this field. The value can also be an integer representing the number of seconds (in decimals) after the time of the response.
`Server`	Represents information about the server that generated the current response, as a String value.

Response Redirection

In this section, the response code of HTTP and its different functions have been discussed. The `setStatus()` method of an `HttpServletResponse` object can be used to send any HTTP response code to a client. The servlet sends back a status code 200, OK if everything works smoothly. A status code of 302 is sent by the servlet displaying the `Resource Temporarily Moved` message, which informs a client that the resource they were looking for is not at the requested URL, but can be found at the URL specified by the `Location` header in the HTTP response. The 302 response code is very helpful because almost every Web browser automatically follows the new link without informing the user. This allows a servlet to take the request of a user and forward it to any other resource on the Internet.

The 302 response code has excellent uses besides its intended purpose. The reason for this is that the 302 response code has a very common implementation. Most websites often track the users who visit their sites to get an idea about their interests so that they can send information related to their interests. The technique for tracking website users requires the extracting of the `referrer` header of an HTTP request. This makes it easy to keep track of the information sent by a site. The problem originates because the link on a site that directs to an external resource also sends a request back to the originating site from where it was sent. To solve the problem, a clever trick can be used that relies on the HTTP 302 response code. In this trick, rather than providing direct links to external resources, all links can be encoded in such a way that they all lead to the same servlet on your site, but at the same time, the real link can be included as a parameter. After that, link tracking can be implemented by providing the intended link through a servlet. In addition, a 302 status code is sent back to the client along with the real link to visit.

HTTP-aware sites commonly use a servlet to track links. The HTTP 302 response code is used very often because it provides the `sendRedirect()` method in the `HttpServletResponse` object. The `sendRedirect()` method takes one parameter, a `String`, representing the new URL, and automatically sets the HTTP 302 status code with appropriate headers. Using the `sendRedirect()` method and the `java.util.Hashtable` class, it is easy to create a servlet to track the links used. Let's create a servlet, named `Link`, to understand the use of the `sendRedirect()` method. Listing 11.9 shows the code for the `Link.java` file (you can find the `Link.java` file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

Listing 11.9: Displaying the Code for the Link.java File

```
package com.kogent;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Link extends HttpServlet {
   static private Hashtable links = new Hashtable();
   String stamp;
   public Link() {
         stamp = new Date().toString();
   }
```

```
        public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {
                String lnk = request.getParameter("link");
                if (lnk != null && !lnk.equals("")) {
                        synchronized (links) {
                                Integer count = (Integer) links.get(lnk);
                                if (count == null) {
                                        links.put(lnk, new Integer(1));
                                }
                                else {
                                        links.put(lnk, new
                                        Integer(1+count.intValue()));
                                }
                        }
                        response.sendRedirect(lnk);
                }
                else {
                        response.setContentType("text/html");
                        PrintWriter out = response.getWriter();
                        request.getSession();
                        out.println("<html>");
                        out.println("<head>");
                        out.println("<title>Links Tracker Servlet</title>");
                        out.println("</head>");
                        out.println("<body>");
                        out.println("<p>Links Tracked Since");
                        out.println(stamp+":</p>");
                        if (links.size() != 0) {
                                Enumeration enm = links.keys();
                                while (enm.hasMoreElements()) {
                                        String key = (String)enm.nextElement();
                                        int count =
                                        ((Integer)links.get(key)).intValue();
                                        out.println(key+" : "+count+" visits<br>");
                                }
                        }
                        else {
                                out.println("No links have been tracked!<br>");
                        }
                        out.println("</body>");
                        out.println("</html>");
                }
        }
        public void doPost(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {
                doGet(request, response);
        }
    }
```

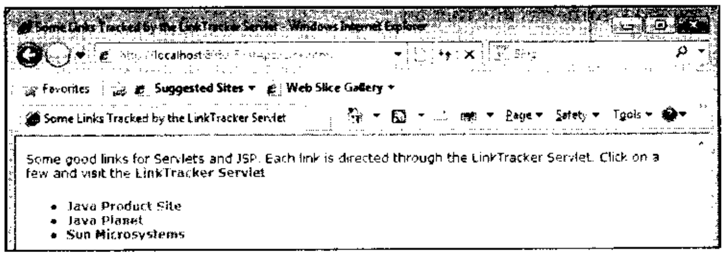
Some links used by the `Link` servlet class are needed to complement the `Link` servlet. The links can be encoded properly to redirect a user to any resource. Encoding the links only requires passing the real link as a link parameter in a query `String`. The code given in Listing 11.10 is that of a simple HTML page that includes a few properly encoded links. Save the HTML code provided in Listing 11.10 as `Link.html` in the base directory of the `FirstApp` Web application. Listing 11.10 shows the code for the `Link.html` file (you can find this file on the CD in the `code\JavaEE\Chapter11\FirstApp` folder):

Listing 11.10: Displaying the Code for the Link.html File

```
    <html>
      <head>
              <title>Some Links Tracked by the LinkTracker Servlet</title>
              <link rel="stylesheet" href="mystyle.css" type="text/css"/>
      </head>
      <body>
              Some good links for Servlets and JSP. Each link is directed through
              the LinkTracker Servlet. Click on a few and visit the
      <a href="Link">LinkTracker Servlet</a>.
              <ul>
                      <li><a href="Link?link=http://www.java.sun.com">
```

```
                    Java Product Site</a></li>
                    <li><a href="Link?link=http://www.javaplanet.com">
                    Java Planet</a></li>
                    <li><a href="Link?link=http://java.sun.com">
                    Sun Microsystems</a></li>
          </ul>
     </body>
</html>
```

After creating the new `FirstApp.war` file and deploying the `FirstApp` Web application, browse the `http://localhost:8080/FirstApp/Link.html` URL. Each link is directed through the `Link` servlet, which in turn redirects a browser to visit the correct link. Before each redirection, the `Link` servlet logs the number of times the link has been visited by keying the link URL to an Integer object in a hashtable. You can browse the `http://localhost:8080/FirstApp/Link` URL, to view information about the links visited. The results are same as of the last reloading of the `Link` servlet. This servlet does not log the information for long-term use. Figure 11.16 displays the `Link.html` page that comprises multiple links:

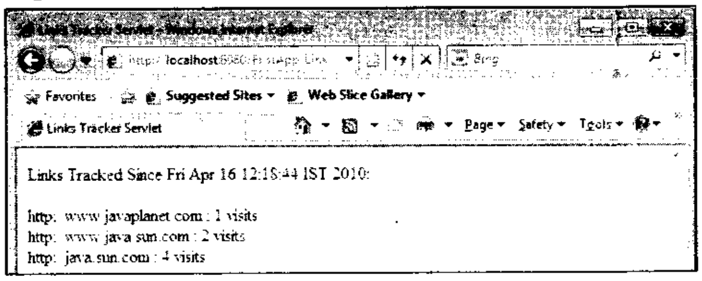

Some Links Tracked by the LinkTracker Servlet - Windows Internet Explorer
Favorites
Suggested Sites
Web Slice Gallery
Some Links Tracked by the LinkTracker Servlet
Page
Safety
Tools

Some good links for Servlets and JSP. Each link is directed through the LinkTracker Servlet. Click on a few and visit the LinkTracker Servlet

- Java Product Site
- Java Planet
- Sun Microsystems

Figure 11.16: Displaying the Link.html Page

After clicking each link multiple times, click the `LinkTracer Servlet` link. You are redirected to the `Link` servlet, as shown in Figure 11.17:



```
Links Tracked Since Fri Apr 16 12:18:44 IST 2010:

http: www javaplanet com : 1 visits
http: www java sun.com : 2 visits
http: java.sun.com : 4 visits
```

Figure 11.17: Displaying the Link Servlet

Figure 11.17 displays the output after clicking the `LinkTracer Servlet` link.

Now, let's discuss the concerns related to response redirection translation.

Response Redirection Translation Issues

Response redirection is a tool that intimates a user about the resource to which a response is to be redirected. It works with any implementation of the Servlet API. However, there is a specific bug that arises while using relative response redirection. Consider the following command, which is used to redirect response:

```
response.sendRedirect("../foo/bar.html");
```

The preceding command works perfectly when used in some servlets but not in others. The problem comes from using the relative back `../` to traverse back a directory. A JSP page can use this command correctly; (you have to assume that the browser translates the URL correctly) but the JSP page can use it only if the requested URL is combined with the correct path and ends on an appropriate resource. For instance, if `http://localhost:8080/foo/bar.html` is a valid URL, then `http://localhost:8080/foo/../foo/bar.html` should also be valid. However, `http://localhost:8080/foo/foo/../foo/bar.html` will not reach the same resource.

This might seem an irrelevant problem. However, request dispatching that we introduce in the next section makes it clear why this is an issue. Request dispatching allows requests to be forwarded on the server-side, which means that the requested URL does not change, but the server-side resource that handles the request can be changed. Relative redirections are not always safe; using `../` can be bad. The solution is to either use absolute redirections or a complete URL, as shown by the following command:

```
response.sendRedirect("http://localhost/foo/bar.html");
```

Alternatively, you can use an absolute URL from the root, "/", of the Web application, as shown by the following command:

```
response.sendRedirect("/foo/bar.html");
```

The `HttpServletRequest getContextPath()` method should also be used, when you can deploy the Web application to a non-root URL, as shown by the following command:

```
response.sendRedirect(request.getContextPath()+"/foo/bar.html");
```

You have already studied about the `HttpServletRequest` object along with the use of the `getContextPath()` method, earlier in this chapter.

Auto-Refresh/Wait Pages

The other useful response header technique is to send a wait page or a page that auto-refreshes to a new page after a given time period to a user. This tactic is helpful in cases where there is a possibility of getting a response which might take an uncontrollable time to generate or for cases where you want to ensure a brief pause in a response. In this case, the entire mechanism involves setting the refresh response header. The header can be set by using the following command:

```
response.setHeader("Refresh", "time; URL=url" );
```

In the preceding command, `time` is replaced with the amount of seconds the page should wait, and `url` is replaced with the URL that the page should eventually load. For instance, if you want to load the `http://localhost/foo.html` URL after waiting for 10 seconds, the header is set by using the following command:

```
response.setHeader("Refresh", "10; URL=http://localhost:8080/foo.html");
```

The technique of sending a page that auto-refreshes is very helpful because it allows a proper message to be conveyed to clients until their requests are being processed. For example, a simple your-request-is-being-processed-page, which automatically refreshes to display the results of the response after a few seconds, can be displayed to the client. Alternatively, the client has to wait until a request is completely processed, before sending back any content. The alternative approach is used in most of the cases. However, this approach requires the client browser to wait for the response. Due to this, sometimes the client may assume that the request may result as timed-out, and may make a time-consuming request twice.

Another practical use for a wait page is to slow down a request. This is done by a developer to get better and more relevant information. For example, a wait page that displays either an advertisement or legal information before redirecting a user to the desired page.

NOTE

In some situations, the Refresh response header can prove to be helpful. Sometimes it can be considered as the de facto standard; however, it is not a standard HTTP 1.1 header.

Now, let's learn how to delegate a request to a resource and discuss request scopes.

Describing Request Delegation and Request Scope

Request delegation refers to the request of a single client passing through many servlets or other resources in a Web application. The entire process is performed entirely on the server-side, unlike response redirection. Request delegation does not require any action from a client or extra information sent between the client and the server. Request delegation is available through the `javax.servlet.RequestDispatcher` object, which can be obtained by calling any of the following methods of the `ServletRequest` object:

- `getRequestDispatcher(java.lang.String path)`–Returns the `RequestDispatcher` object for the path provided as an argument. The path value must start from the base directory / and can direct to any resource in a Web application.
- `getNamedDispatcher(java.lang.String name)`–Returns the `RequestDispatcher` object for the named servlet. The servlet-name elements of the `web.xml` file define the valid names.

The following two methods are provided by a `RequestDispatcher` object to include different resources and to forward a request to a different resource:

- `forward(javax.servlet.ServletRequest, javax.servlet.ServletResponse)`–Delegates a request and response to the resource of the `RequestDispatcher` object. A call to the `forward()` method may be used only if no content is previously sent to a client. After the completion of the processing of the `forward()` method, no further data can be sent to the client.
- `include(javax.servlet.ServletRequest, javax.servlet.ServletResponse)`–Works similar to the `forward()` method, but has some restrictions. Any number of resources can be included by a servlet by using the `include()` method; however, the resource cannot set headers or commit a response.

While a request delegation is often used to break a large servlet into smaller and more relevant parts, a simple case involves separating a common HTML header that is being shared by all the pages of a website. The `include()` method of the `RequestDispacher` object provides a convenient method to include the header in any other servlet that needs the header. In the case of request delegation, any future changes to the header are automatically reflected on all the servlets. A much more elegant solution to this problem is provided by JSPs. In practice, a servlet request delegation is usually used to break large servlets into smaller and relevant parts.

In addition to this, simple server-side components include request delegations, which are a key part of server-side Java implementations of popular design patterns. As far as Java Servlet and JSP are concerned, design patterns are commonly agreed-upon methods to build Web applications that are robust in functionality and easily maintainable.

You may already know that there are well-defined scopes for variables in Java. Local variables are declared inside methods and are by default only available inside the scope of that method. Instance variables, declared in a class but outside a method or Constructor, are available to all methods in a Java class. There are many other possible scopes as well. These scopes help to keep track of objects and to allow JVM to accurately carry out garbage-collection of the memory. Apart from the various Java variable scopes, such as local and global, the servlets also provide some new scopes. The request scope is introduced by a request delegation.

The request scope and the other scopes are not officially marked by the Servlet specification. A set of methods defined by the servlet specification in the `javax.servlet` package allow you to bind objects to and retrieve objects from various containers (that are themselves objects). As an object bound in this manner is referenced by the container it is bound to, the bound object is not destroyed until the reference is removed. Therefore, bound objects are in the same scope as the container they are bound to. For example, the `HttpServletRequest` object is bound to a container and includes the methods of the `HttpServletRequest` interface. The methods of the `HttpServletRequest` object can be used to bind, access, and remove objects to and from the request scope that is shared by all servlets to which a request is delegated. This is an important concept and can easily be shown in the Servlet2Servlet class, created in Listing 11.11.

A request scope can be defined as a method in which an object is passed between two or more servlets with the assurance that the object goes out of scope (that is, it will be garbage-collected) after the last servlet has performed its task. In later chapters, more examples of this concept are provided. Let's first create the `Servlet2Servlet` servlet that passes an object to another servlet, `Servlet2Servlet2`. Listing 11.11 provides the code for the `Servlet2Servlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

Listing 11.11: Displaying the Code for the Servlet2Servlet.java File

```
package com.kogent;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class Servlet2Servlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
  response) throws IOException, ServletException {
       response.setContentType("text/html");
       String param = request.getParameter("value");
       if(param != null && !param.equals("")) {
              request.setAttribute("value", param);
              RequestDispatcher rd =
       request.getRequestDispatcher("/Servlet2Servlet2");
              rd.forward(request, response);
              return;
       }
       PrintWriter out = response.getWriter();
       out.println("<html>");
       out.println("<head>");
       out.println("<title>Servlet #1</title>");
       out.println("</head>");
       out.println("<body>");
       out.println("<h1>A form from Servlet #1</h1>");
       out.println("<form>");
       out.println("Enter a value to send to Servlet #2.");
       out.println("<input name=\"value\"><br>");
       out.print("<input type=\"submit\" ");
       out.println("value=\"Send to Servlet #2\">");
       out.println("</form>");
       out.println("</body>");
       out.println("</html>");
  }
}
```

Compile the Servlet2Servlet servlet and map it to the `/Servlet2Servlet` URL extension in the `web.xml` file by using the following code snippet:

```
<servlet>
  <servlet-name>Servlet2Servlet</servlet-name>
  <servlet-class>com.kogent.Servlet2Servlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Servlet2Servlet</servlet-name>
  <url-pattern>/Servlet2Servlet</url-pattern>
</servlet-mapping>
```

Listing 11.12 shows the code for the `Servlet2Servlet2.java` file (you can find this file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

Listing 11.12: Displaying the Code for the Servlet2Servlet2.java File

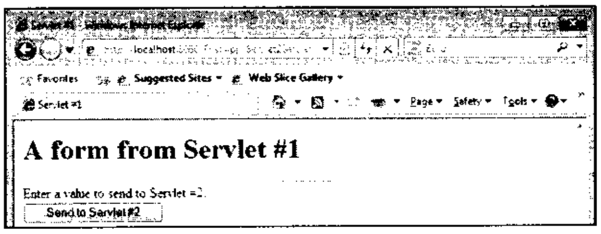
```
package com.kogent;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Servlet2Servlet2 extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
  response) throws IOException, ServletException {
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();
       out.println("<html>");
       out.println("<head>");
       out.println("<title>Servlet #2</title>");
       out.println("</head>");
       out.println("<body>");
       out.println("<h1>Servlet #2</h1>");
       String value = (String)request.getAttribute("value");
       if(value != null && !value.equals("")) {
              out.print("Servlet #1 passed a String object via ");
              out.print("request scope. The value of the String is: ");
              out.println("<b>"+value+"</b>.");
       }
```

```
        else {
            out.println("No value passed!");
        }
        out.println("</body>");
        out.println("</html>");
    }
}
```

Now, save the code given in Listing 11.12, with the name `Servlet2Servlet2.java` in the `/src/com/kogent` directory of the `FirstApp` Web application. In addition, compile the `Servlet2Servlet2` servlet and map it to the `/Servlet2Servlet2` URL extension in the `web.xml` file, as provided in the following code snippet:

```
<servlet>
    <servlet-name>Servlet2Servlet2</servlet-name>
    <servlet-class>com.kogent.Servlet2Servlet2</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Servlet2Servlet2</servlet-name>
    <url-pattern>/Servlet2Servlet2</url-pattern>
</servlet-mapping>
```

Redeploy the FirstApp Web application and the application is ready for execution. Next, browse the `http://localhost:8080/FirstApp/Servlet2Servlet` URL. Figure 11.18 shows the servlet response that appears similar to a simple HTML form asking you to enter a value to pass to the second servlet:



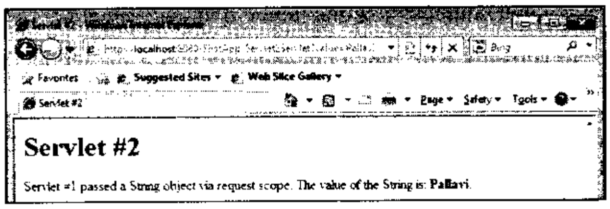
A form from Servlet #1

Enter a value to send to Servlet #2.

Send to Servlet #2

Figure 11.18: Displaying the Output of the `Servlet2Servlet` Servlet

Type a value in the `Enter a value to send Servlet#2` text box to send to the second servlet. In our case, we have entered the value, Pallavi. Now, click the `Send to Servlet #2` button. The second servlet appears, as shown in Figure 11.19:



Servlet #2

Servlet #1 passed a String object via request scope. The value of the String is: **Pallavi**.

Figure 11.19: Displaying Servlet 2 in Progress

The preceding example demonstrates how the value of one servlet is passed to the other servlet. After discussing the concept of request delegation and request scope, let's now learn how to share information among servlets by using the servlet collaboration technique.

Understanding Servlet Collaboration

Servlet sometimes cooperate with each other by sharing information. This sort of cooperation is known as servlet collaboration. The collaborating servlets can pass the shared information between each other through method invocations; however, to do this, each servlet is required to know about the servlets with which it is collaborating. This adds unnecessary burden on the server. Several other techniques can be used to carry out for servlet collaboration. Let's discuss these techniques in the following sections.

Collaboration through the System Properties List

A simple way for servlets in collaboration to share information is by using Java's system-wide `Properties` list. The `Properties` list is found in the `java.lang.System` class and holds standard system properties, such as `java.version` and `path.separator` as well as application-specific properties. Servlets can use the `Properties` list to hold the information they need to share. A servlet can add or change a property by using the `setProperties()` method, as shown in the following code snippet:

```
System.setProperties().put("key", "value");
```

The concerned servlet, or some other servlet running in the same JVM, can later get the value of the property by calling the `getProperties()` method, as shown in the following code snippet:

```
String value = System.getProperty("key");
```

The property can also be removed, by calling the `remove()` method, as shown in the following code snippet:

```
System.getProperties().remove("key");
```

Generally, you should include a prefix while defining the key for a property, which contains the name of the servlet's package and the name of the collaboration group, for example, `com.kogent.Servlet.ShoppingCart`. The `Properties` class is `String`-based, which implies that each key and value is supposed to be a `String`. However, this is not a commonly enforced limitation and can be ignored by servlets if they want to store and retrieve non-`String` objects. Such servlets can use the `Properties` list as a `hashtable` at the time of storing keys and values because the `Properties` class extends the `Hashtable` class; therefore, the `Properties` list can be treated as a `hashtable`. For example, a servlet can add or change a property object by calling the `setProperties()` method, as shown in the following code snippet:

```
System.setProperties().put(keyObject, valueObject);
```

The property object is retrieved by calling the `getProperties()` method, as shown in the following code snippet:

```
SomeObject valueObject = (SomeObject)System.getProperties().get(keyObject);
```

The property object is removed by calling the `remove()` method, as shown in the following code snippet:

```
System.getProperties().remove(keyObject);
```

Due to the misuse of the `Properties` list, the `getProperty()`, `list()` and `save()` methods of the `Properties` class throw the `ClassCastException` exception and it is also assumed that each key and value is of the `String` type. Due to this reason, you should use some other technique for Servlet collaboration. JVM should look for the class files for the `keyObject` and `valueObject` arguments in the server's `CLASSPATH`. The class files should not be looked in the default servlet directory where the servlet-class loaders load and reload the servlets.

Servlet collaboration works correctly with the use of property lists if the servlets use the property lists to share insensitive, non-critical, and easily replaceable information. For example, consider a set of servlets that are used to sell pizzas and share a particular pizza on a special day of the week. To implement collaboration between the servlets, the administrative servlet can use a property list to set a special day and the pizza to be served on that day. The following code snippet shows the implementation of the `setProperties()` method to set the value for the pizza and the special day:

```
System.setProperties().put("com.kogent.ServingPizza.special.pizza", "Cheese
                        pizza");
System.setProperties().put("com.kogent.ServingPizza.special.day", new Date());
```

Now, every other servlet on the server is able to access the `special` properties and display it with the code shown in the following code snippet:

```
String pizza = System.getProperty("com.kogent.ServingPizza.special.pizza");
Date day =
    (Date)System.getProperties().get("com.kogent.ServingPizza.special.day");
DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT);
String today = df.format(day);
prnwriter.println("Our pizza special today (" + today + ") is: " + pizza);
```

In the preceding code snippet, the `System.getProperty()` method is used to retrieve the value of the `com.kogent.ServingPizza.special.pizza` key.

Collaboration through a Shared Object

Sharing information through a shared object is another way for servlets to collaborate with one another. A shared object can hold a pool of shared information and make it available to each servlet as required. In a sense, the system `Properties` list is a special case example of a shared object. To manipulate an object's data, the shared object often incorporates some business logic or rules. By incorporating a rule that the shared object's data be available only through well-defined methods, the rule protects the shared object's data. The rule helps to protect data integrity and triggers events so that they can handle certain conditions. Moreover, various data manipulations can be abstracted into a single method invocation. This capability is not available in the case of the `Properties` list. The garbage collector is an important aspect of collaborating through a shared object. If at any time a loaded servlet does not reference the object, then the servlet can reclaim it. Therefore, every servlet that uses a shared object must save a reference to the object to keep the garbage collector at bay.

Let's consider the previous example in which we used servlets to sell pizzas. Collaboration between servlets to maintain a shared inventory of ingredients can be implemented through a shared object. For this, you first need to create a shared `PizzaInventory` class. The `PizzaInventory` class is defined to maintain the ingredient count and display the count through public methods. An example of the `PizzaInventory` class is shown in Listing 11.13. Notice that this class is a singleton (a class that has just one instance). This makes it easy for every servlet sharing the class to maintain a reference to the same instance.

Now, let's discuss the shared inventory class. Listing 11.13 shows the code for `PizzaInventory.java` (you can find this file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

Listing 11.13: Displaying the Code for the PizzaInventory.java File

```
package com.kogent;
public class PizzaInventory {
  // Protect the constructor, so no other class can call it
  private PizzaInventory() { }
  // Create the only instance, save it to a private static variable
  private static PizzaInventory instance = new PizzaInventory();
  // Make the static instance publicly available
  public static PizzaInventory getInstance() { return instance; }
  // How many servings of each item do we have?
  private int cheese = 0;
  private int wheatflour = 0;
  private int beans = 0;
  private int capsicum = 0;
  // Add to the inventory
  public void addCheese(int added) { cheese += added; }
  public void addWheatflour(int added) { wheatflour += added; }
  public void addBeans(int added) { beans += added; }
  public void addCapsicum(int added) { capsicum += added; }
  // Called when it is time to make a pizza.
  // Returns true if there are enough ingredients to make the pizza,
  // false if not. Decrements the ingredient count when there are enough.
  synchronized public boolean makePizza() {
       // Pizza requires one serving of each item
       if (cheese > 0 && wheatflour > 0 && beans > 0 && capsicum > 0) {
             cheese--; wheatflour--; beans--; capsicum--;
             return true;           // can make the pizza
       }
       else {
             // could order more ingredients
             return false;          // cannot make the pizza
       }
  }
}
```

Save the `PizzaInventory.java` file in the `src\com\kogent` directory of the FirstApp Web application. `PizzaInventory` maintains an inventory count for four pizza ingredients: `cheese`, `wheatflour`, `beans`, and `capsicum`. The `PizzaInventory` class holds the count of the ingredients with the private instance variables. Information of the count should be kept in an external database to maintain a record of the quantity of ingredients used to produce pizzas. Each ingredient's inventory count is increased by using the addCheese(), addWheatflour(), addBeans(), and addCapsicum() methods. These methods may be called from a servlet accessed by the ingredient prepared in the day.

In the makePizza() method, the value of the inventory counts are decreased together. The role of this method is to check whether or not there are enough ingredients to make a full pizza. If there is, then the method decreases the ingredient count and returns true. However, if the ingredients are insufficient, then the makePizza() method returns false (in an improved version, the method may choose to order more ingredients). The makePizza() method may be called by a servlet selling pizzas over the Internet, and perhaps also by a servlet communicating with the check-out cash register. Remember that, similar to all the other non-servlet-class files, the class file for PizzaInventory is placed somewhere in the server's CLASSPATH (such as server_root/classes). Listing 11.14 shows you how a servlet adds ingredients to the inventory.

Let's now create a servlet to add some ingredients to a shared inventory. Listing 11.14 shows the code for the PizzaInventoryProducer.java file (you can find this file on the CD in the code\JavaEE\Chapter11\FirstApp\src\com\kogent folder):

Listing 11.14: Displaying the Code for the PizzaInventoryProducer.java File

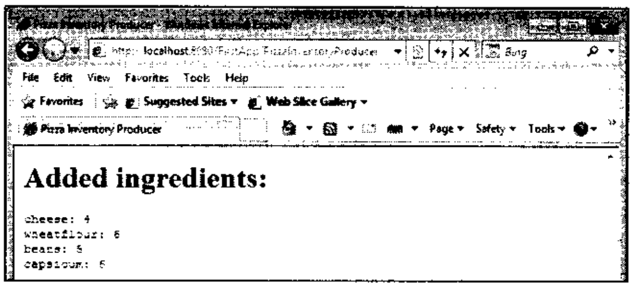
```
package com.kogent;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kogent.* ;
public class PizzaInventoryProducer extends HttpServlet {
  // Get (and keep) a reference to the shared PizzaInventory instance
  // PizzaInventory inventory = PizzaInventory.getInstance();
  PizzaInventory inventory = PizzaInventory.getInstance();
  public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter prnwriter = res.getWriter();
        prnwriter.println("<HTML>");
        prnwriter.println("<HEAD><TITLE>Pizza Inventory Producer
                             </TITLE></HEAD>");
        // Produce random amount of each item
        Random random = new Random();
        int cheese = Math.abs(random.nextInt() % 10);
        int wheatflour = Math.abs(random.nextInt() % 10);
        int bean = Math.abs(random.nextInt() % 10);
        int capsicum = Math.abs(random.nextInt() % 10);
        // Add the item sto the inventory
        inventory.addCheese(cheese);
        inventory.addWheatflour(wheatflour);
        inventory.addBeans(beans);
        inventory.addCapsicum(capsicum);
        // Print the production results
        prnwriter.println("<BODY>");
        prnwriter.println("<H1>Added ingredients:</H1>");
        prnwriter.println("<PRE>");
        prnwriter.println("cheese: " + cheese);
        prnwriter.println("wheatflour: " + wheatflour);
        prnwriter.println("beans: " + beans);
        prnwriter.println("capsicum: " + capsicum);
        prnwriter.println("</PRE>");
        prnwriter.println("</BODY></HTML>");
  }
}
```

Save the PizzaInventoryProducer.java file in the src\com\kogent directory of the FirstApp Web application and compile the PizzaInventory and PizzaInventoryProducer servlets. The following code snippet is used to map the PizzaInventoryProducer servlet to the /PizzaInventoryProducer by using the <url-pattern> element in the web.xml file:

```
<servlet>
   <servlet-name>PizzaInventoryProducer</servlet-name>
   <servlet-class>com.kogent.PizzaInventoryProducer </servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>PizzaInventoryProducer</servlet-name>
   <url-pattern>/PizzaInventoryProducer</url-pattern>
</servlet-mapping>
```

Create a new FirstApp.war file and redeploy the FirstApp Web application. Now, browse the http://localhost:8080/FirstApp/PizzaInventoryProducer URL to see the output.

A random amount of each ingredient (somewhere between zero to nine servings) is produced and added to the inventory, whenever the PizzaInventoryProducer servlet is accessed. Figure 11.20 shows the result of executing the PizzaInventoryProducer servlet:


http://localhost:8080/FirstApp/PizzaInventoryProducer
File Edit View Favorites Tools Help
Favorites
Suggested Sites
Web Slice Gallery
Pizza Inventory Producer
Page
Safety
Tools

Added ingredients:

```
cheese: 4
wheatflour: 6
beans: 5
capsicum: 5
```

Figure 11.20: Displaying the Output of the PizzaInventoryProducer Servlet

The PizzaInventoryProducer servlet plays the important role of maintaining a reference to the shared PizzaInventory instance. This implies that the PizzaInventory instance cannot be reclaimed by the garbage collector until the servlet is loaded. The code for the PizzaInventoryConsumer.java file is provided on the CD.

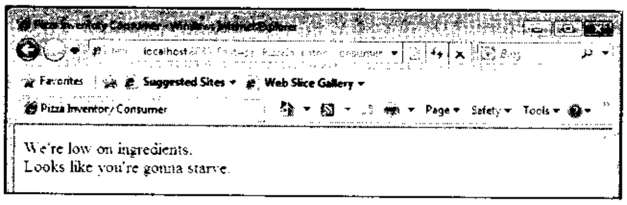
Now, let's create a servlet that calls the makePizza() method, informing the inventory that it wants to make a pizza. The PizzaInventoryConsumer.java file provides the code for the servlet. Save the PizzaInventoryConsumer.java file in the src\com\kogent directory of the FirstApp Web application. Listing 11.15 shows the code for the PizzaInventoryConsumer.java file (you can find this file on the CD in the code\JavaEE\Chapter11\FirstApp\src\com\kogent folder):

Listing 11.15: Displaying the Code for the PizzaInventoryConsumer.java File

```
package com.kogent ;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kogent.* ;
public class PizzaInventoryConsumer extends HttpServlet {
   // Get (and keep) a reference to the shared PizzaInventory instance
   private PizzaInventory inventory = PizzaInventory.getInstance();
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {
         res.setContentType("text/html");
         PrintWriter prnwriter = res.getWriter();
         prnwriter.println("<HTML>");
         prnwriter.println("<HEAD><TITLE>Pizza Inventory Consumer
                                   </TITLE></HEAD>");

         prnwriter.println("<BODY><BIG>");
         if(inventory.makePizza()) {
                 prnwriter.println("Your pizza will be ready
                                      in a few minutes.");
         }
         else
         {
                 prnwriter.println("We're low on ingredients.<BR>");
```

```
                prnwriter.println("Looks like you're gonna starve.");
            }
            prnwriter.println("</BIG></BODY></HTML>");
        }
    }
```

The `PizzaInventoryConsumer` servlet does not need to decrease the ingredients count by itself. It maintains a reference to the `PizzaInventory` instance. This implies that the `PizzaInventory` instance can be referenced even if the `PizzaInventoryProducer` servlet is unloaded. Compile the preceding servlet and redeploy the `FirstApp` Web application. Now, browse the `http://localhost:8080/FirstApp/PizzaInventoryConsumer` URL to see the output. Figure 11.21 shows the output, which is displayed when the `PizzaInventoryConsumer` servlet is executed:



```
We're low on ingredients.
Looks like you're gonna starve.
```

Figure 11.21: Displaying the Output of the PizzaInventoryConsumer Servlet

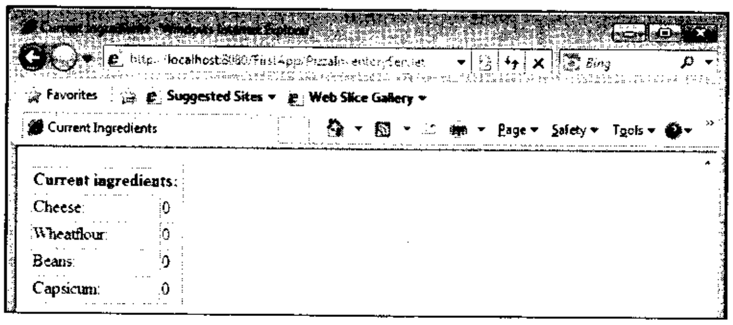
You can also make a servlet act as a shared object. There is an added advantage of using a shared servlet. Sharing allows a servlet to maintain its state by using its `init()` and `destroy()` methods to load and save its state. In addition, each time a shared servlet is accessed, the servlet can print its current state. Let's re-create the `PizzaInventory` servlet to implement the concept of sharing a servlet.

Listing 11.16 shows the code for the re-created `PizzaInventoryServlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

Listing 11.16: Displaying the Code for the PizzaInventoryServlet.java File

```
package com.kogent;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class PizzaInventoryServlet extends HttpServlet {
   // How many "servings" of each item do we have?
   private int cheese = 0;
   private int wheatflour = 0;
   private int beans = 0;
   private int capsicum = 0;
   // Add to the inventory as more servings are prepared.
   public void addCheese(int added) { cheese += added; }
   public void addWheatFlour(int added) { wheatflour += added; }
   public void addBeans(int added) { beans += added; }
   public void addCapsicum(int added) { capsicum += added; }
   // Called when it is time to make a pizza.
   // Returns true if there are enough ingredients to make the pizza,
   // false if not. Decrements the ingredient count when there are enough.
   synchronized public boolean makePizza() {
                // Pizza requires one serving of each item
                if (cheese > 0 && wheatflour > 0 && beans > 0 && capsicum > 0) {
                         cheese--; wheatflour--; beans--; capsicum--;
                         return true;  // can make the pizza
                }
                else {
                         // Could order more ingredients
                         return false;  // cannot make the pizza
                }
   }
   // Display the current inventory count.
   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {
                res.setContentType("text/html");
                PrintWriter prnwriter = res.getWriter();
                prnwriter.println("<HTML><HEAD><TITLE>Current
                    Ingredients</TITLE></HEAD>");
            prnwriter.println("<BODY>");
            prnwriter.println("<TABLE BORDER=1>");
            prnwriter.println("<TR><TH COLSPAN=2> Current ingredients:</TH>
                    </TR>");
            prnwriter.println("<TR><TD>Cheese:</TD><TD>" + cheese + "</TD>
                    </TR>");
            prnwriter.println("<TR><TD>Wheatflour:</TD><TD>" + wheatflour +
                    "</TD></TR>");
            prnwriter.println("<TR><TD>Beans:</TD><TD>" + beans + "</TD></TR>");
            prnwriter.println("<TR><TD>Capsicum:</TD><TD>" + capsicum +
                    "</TD></TR>");
            prnwriter.println("</TABLE>");
            prnwriter.println("</BODY></HTML>");
    }
    // Load the stored inventory count
    public void init(ServletConfig config) throws ServletException {
            super.init(config);
            loadState();
    }
    public void loadState() {
            // Try to load the counts
            FileInputStream file = null;
            try {
            file = new FileInputStream("PizzaInventoryServlet.state");
            DataInputStream in = new DataInputStream(file);
            cheese = in.readInt();
            wheatflour = in.readInt();
            beans = in.readInt();
            capsicum = in.readInt();
            file.close();
            return;
            }
            catch (IOException ignored) {
            // Problem during read
            }
            finally {
            try {
                    if (file != null) file.close();
            }
            catch (IOException ignored) { }
            }
    }
    public void destroy() {
            saveState();
    }
    public void saveState() {
            // Try to save the counts
            FileOutputStream file = null;
            try {
            file = new FileOutputStream("PizzaInventoryServlet.state");
            DataOutputStream prnwriter = new DataOutputStream(file);
            prnwriter.writeInt(cheese);
            prnwriter.writeInt(wheatflour);
            prnwriter.writeInt(beans);
            prnwriter.writeInt(capsicum);
            return;
            }
            catch (IOException ignored) {
            // Problem during write
            }
            finally {
            try {
                    if (file != null) file.close();
            }
            catch (IOException ignored) { }
            }
    }
}
```

Now, save the `PizzaInventoryServlet`.java file in the `src\com\kogent` directory of the `FirstApp` Web application. Then, redeploy the `FirstApp` Web application and view the output by accessing the `http://localhost:8080/FirstApp/PizzaInventoryServlet` URL. The `PizzaInventoryServlet` servlet is no longer a singleton but a normal HTTP servlet, which defines an `init()` method that loads its state as well as a `destroy()` method that saves its state. Figure 11.22 shows the output of the `PizzaInventoryServlet` servlet:



Current ingredients:

Cheese:	0
Wheatflour:	0
Beans:	0
Capsicum:	0

Figure 11.22: Displaying the Output from PizzaInventoryServlet, Showing its State

Remember that even as a servlet, the `PizzaInventoryServlet.class` file should remain in the server's standard `CLASSPATH`. This is required to keep the `PizzaInventoryServlet` servlet from being reloaded. Both the `PizzaInventoryProducer` and `PizzaInventoryConsumer` classes can get a reference to the `PizzaInventoryServlet` servlet. The following code snippet shows how to reuse the `PizzaInventoryServlet` servlet:

```
// Get the inventory Servlet instance if we haven't before
if (inventory == null) {
   inventory = ( PizzaInventoryServlet)
             ServletUtils.getServlet("PizzaInventoryServlet",
             req, getServletContext());
         // If the load was unsuccessful, throw an exception
   if (inventory == null) {
         throw new ServletException("Could not locate PizzaInventoryServlet");
   }
}
```

In the preceding code snippet, instead of calling `PizzaInventory.getInstance()` method, the producer and consumer classes can ask the `PizzaInventoryServlet` instance from the server.

Collaboration through Inheritance

Collaborating through inheritance is perhaps the easiest technique of servlet collaboration. In the inheritance technique, each servlet requiring collaboration can extend the same class and opt for inheriting the same shared information. This simplifies the code of the collaborating servlets and allows only the proper subclasses to access the shared information. Moreover, the common superclass can hold a reference to the shared information or hold the shared information itself.

In the following sections, you learn to collaborate with servlets through inheritance in two ways, by inheriting a shared reference and by inheriting the shared information.

Inheriting a Shared Reference

In case of inheriting a shared reference, a common superclass can hold any number of references to shared business objects, which are easily made available to its subclasses. Such a superclass is shown in Listing 11.17, which we can use for our `PizzaInventory` example.

Listing 11.17 shows the code for `PizzaInventorySuperclass.java` (you can find this file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

Listing 11.17: Displaying the Code for the PizzaInventorySuperclass.java File

```
package com.kogent;
import javax.servlet.*;
import javax.servlet.http.*;
import com.kogent.* ;
public class PizzaInventorySuperclass extends HttpServlet {
   protected static PizzaInventory inventory = PizzaInventory.getInstance();
}
```

In Listing 11.17, the `PizzaInventorySuperclass` servlet creates a new `PizzaInventory` instance. Now, save the `PizzaInventorySuperclass.java` file in the `src\com\kogent` directory of the `FirstApp` Web application. The `PizzaInventoryProducer` and `PizzaInventoryConsumer` classes can now extend the `PizzaInventorySuperclass` class and inherit a reference to the `PizzaInventory` instance. To understand how the `PizzaInventoryConsumer` class can extend and inherit the reference to the `PizzaInventory` instance, the code for the `PizzaInventoryConsumer` servlet is revised in Listing 11.18.

Listing 11.18 shows the revised code for `PizzaInventoryConsumer.java` (you can find this file on the CD in the `code\JavaEE\Chapter11\FirstApp\src\com\kogent` folder):

Listing 11.18: Displaying the Code for the PizzaInventoryConsumer.java File

```
package com.kogent ;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class PizzaInventoryConsumer extends PizzaInventorySuperclass {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
          throws ServletException, IOException {
                 res.setContentType("text/html");
                 PrintWriter prnwriter = res.getWriter();
                 prnwriter.println("<HTML>");
                 prnwriter.println("<HEAD><TITLE>Pizza Inventory Consumer
                                 </TITLE></HEAD>");
                 prnwriter.println("<BODY><BIG>");
                 if (inventory.makePizza()) {
                 prnwriter.println("Your pizza will be ready in a
                                         few minutes.");
          }
          else {
                      prnwriter.println("We're low on ingredients.<BR>");
                      prnwriter.println("Looks like you're gonna starve.");
                 }
                 prnwriter.println("</BIG></BODY></HTML>");
   }
}
```

The `PizzaInventory` class does not have to be a singleton anymore. The reason is that subclasses naturally inherit the same instance. The class file for the `PizzaInventorySuperclass` class should be put in the server's `CLASSPATH`. This is required to keep the `PizzaInventorySuperclass class` from being reloaded.

Inheriting Shared Information

Apart from allowing servlets to hold shared references, you can also inherit the shared information. For this, you can use a common superclass to hold the shared information by itself or optionally make it available through inherited business logic methods. The following code snippet shows how the `PizzaInventorySuperclass` class holds its own shared information:

```
public class PizzaInventorySuperclass extends
          HttpServlet {
   // How many "servings" of each item do we have?
   private static int cheese = 0;
   private static int wheatflour = 0;
   private static int beans = 0;
   private static int capsicum = 0;
   // Add to the inventory as more servings are prepared.
   protected static void addCheese(int added) { cheese += added; }
   protected static void addWheatflour(int added) { wheatflour += added; }
   protected static void addBeans(int added) { beans += added; }
```

```
    protected static void addCapsicum(int added) { capsicum += added; }
    // Called when it is time to make a pizza.
    // Returns true if there are enough ingredients to make the pizza,
    // false if not. Decrements the ingredient count when there are enough.
    synchronized static protected boolean makePizza()      {
            // ...etc...
    }
    // ...The rest matches PizzaInventoryServlet...
```

Now, let's discuss the difference between the `PizzaInventorySuperclass` and `PizzaInventoryServlet` servlets to analyze what changes are required to inherit the shared information in a servlet. There are only two differences between the two servlets, which are as follows:

- All the variables and methods of the `PizzaInventorySuperclass` servlet are static, which is not the case with the `PizzaInventoryServlet` class. This guarantees that only one inventory is maintained for all the subclasses.
- All the methods of the `PizzaInventorySuperclass` servlet are protected. This implies that the methods will be available only to the subclasses.
- This completes the discussion about servlet collaboration. Let's move to the next section and discuss another important mechanism with respect to the Java Servlets termed as session tracking.

Understanding Session Tracking

Before you learn about session tracking, you should understand what does a session refers to, with respect to a Java Servlet or a Web application. A session can be defined as a collection of HTTP requests shared between a client and Web server over a period of time. While creating a session, you require setting its lifetime, which is set to thirty minutes by default. After the set lifetime expires, the session is destroyed and all its resources are returned back to the servlet engine. Session tracking mechanism tracks the details of a user session.

Session tracking is a process of gathering the user information from Web pages, which can be used in an application. Let's cite an example; a shopping cart application can be taken as the most common example of session tracking. In the shopping cart application, a client accesses the server several times from the same browser and visits several Web pages. After browsing the Web pages, the client decides to purchase some of the items offered by the Web site for sale and clicks the BUY ITEM button. In this case, if a stateless server-side object serves each transaction, and the client's side does not provide any identification on each request; it would not be possible to maintain a filled shopping cart over several HTTP requests from the client. If the user visits a Web page multiple times and selects different items to be added to the shopping cart in each visit, the stateless nature of HTTP might not relate each visit to the same session. Therefore, even writing a stateless transaction data to persistent storage would not be a solution in this regard.

Therefore, session tracking involves identifying the user sessions by related ID numbers and tying the requests to their sessions by using the said ID number. Cookies and URL rewriting are the typical mechanisms for session tracking. Depending upon the Servlet specification, session tracking is implemented through HTTP session objects by the servlet container in the application server. These HTTP session objects are instances of a class and implement the `javax.servlet.http.HttpSession` interface. The `getSession()` method of the `HttpSession` interface is used to create the HTTP session object and the stateful client interaction.

The question may arise about the scope of the HTTP session object. Will it be limited to single request or multiple requests or across users? The scope of the HTTP session object is limited to the single client. It is important to note that you cannot use session objects to share the data between different applications and different clients of the same application. There is only one HTTP session object for each client in each application.

To track the session details for a specific user to maintain the session, you need to implement a mechanism in your Web application. You can implement the following session tracking mechanisms to track the session details:

- Cookies
- Hidden form fields

- URL rewriting
- Secure Socket Layer (SSL) sessions

Let's discuss these in detail.

Cookies

While working with session tracking, numerous approaches have been adapted to add a degree of statefulness to the HTTP protocol. Among these approaches, the most widely accepted one is the use of cookies. A cookie is used to transmit an identifier between a server and a client. The transmitting of an identifier is in conjunction with stateful servlets, which can maintain session objects. These session objects are simply the dictionaries that store values (Java objects) together with their associated keys (Java Strings). The following steps describe the usage of cookies:

- After creating a session, the server (container) sends a cookie (as a response from stateful servlet) with a session identifier back to the client. Some other useful information, such as username and password, is also sent with the cookie (all less than 4 KB). The cookie, named `JSESSIONID`, is sent by the container as a response in the HTTP response header.
- Then, whenever any subsequent request is received from the same Web client session (assuming the client supports cookies), the cookie is sent back by the client to the server as part of the request. In this case, the cookie value is used by the server to look for the session state information to be passed to the servlet.
- Finally, with subsequent responses, the container sends the updated cookie back to the client.

As the container handles the process of sending a cookie, the servlet code is not required while using cookies. A Web browser automatically handles the process of sending cookies back to the server unless the user disables cookies.

The cookie is used by the container to maintain a session. Cookies can be retrieved by a servlet by using the `getCookies()` method of the `HttpServletRequest` object. The cookie attributes can be examined by the servlet using the accessor methods of the `javax.servlet.http.Cookie` objects.

The servlet container sends a cookie to the client. Upon each HTTP request, the cookie is returned back to the server. This way, the session id indicated by the cookie is associated with the request. You should note that you can use HTTP cookies to store information about a session and for each subsequent connection the current session can be looked up and then information about that session is extracted from a location on the server machine. For example, in the following code snippet, the code to retrieve the session information is provided that can be implemented in a servlet:

```
String sesID = makeUniqueString();
Hashtable sesInfo = new Hashtable();
Hashtable hashtab = findTableStoringSessions();
hashtab.put(sesID, sesInfo);
Cookie sesCookie = new Cookie("JSESSIONID", sesID);
sesCookie.setPath("/");
response.addCookie(sesCookie);
```

The preceding code snippet requests the server, which uses the `hashtab` hash table to associate a session ID of the `JSESSIONID` cookie with the `sesInfo` hash table of data associated with that particular session. A cookie is the most widely used approach for session handling. The servlet's session tracking API handles sessions and performs the following tasks:

- Extracting the cookie that stores other cookie's session identifier
- Setting an appropriate expiration time for the cookie
- Associating hash tables with each request
- Generating a unique session identifier

Now, let's discuss the Cookies API as it would help you to understand about the classes and interfaces used for session tracking.

In Java Servlet `API`, *`Cookie`* is a class of the `javax.servlet.http` package, which abstracts the notion of a cookie. You can implement session tracking using cookies with the help of the `addCookie()` and getCookies() methods. These methods are provided by the javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse interfaces and are used to add cookies to HTTP responses and to retrieve the cookies from HTTP requests, respectively. A cookie is abstracted by the Cookie class. The following code snippet shows a constructor that instantiates a cookie instance with the given name and value:

```
public Cookie(String name, String value)
```

The Cookie class provides various methods, such as getValue() and setValue(), which simplify working with cookies. For all the cookie parameters, the Cookie class provides getter and setter methods. The following code snippet shows the syntax of some of the methods of the Cookie class:

```
public String getValue()
public void setValue(String newValue)
```

The getter and setter methods can be used to access or set the value of a cookie. Similarly, there exist other methods that can be used to access or change other parameters, such as path and header of a cookie. The following code snippet shows the syntax of the addCookie() method provided by the javax.servlet.http.HttpServletResponse interface to set cookies:

```
public void addCookie(Cookie cookie)
```

To set multiple cookies, you can call this method as many times as you want. The following code snippet shows the syntax of the getCookies() method provided by the javax.servlet.http.HttpServletRequest interface to extract all cookies contained in the HTTP request:

```
public Cookie[] getCookies()
```

Now, you can create your servlet to track the activities of a user using cookies by providing the code for the following actions:

- Check if there is a cookie contained in the incoming request
- Create a cookie and send it with the response, if there is no cookie in the incoming request
- Display the value of the cookies, if there is a cookie in the incoming request

Let's create a new Web application, HandleSession, to implement session tracking using cookies. This application follows the same directory structure as shown in the previous sections of the chapter for the FirstApp application. Let's now create the CookieServlet servlet (in the HandleSession application) to show how to work with a cookie. Listing 11.19 shows the code of the CookieServlet.java file (you can find this file on the CD in the code\JavaEE\Chapter11\HandleSession\src\com\kogent folder):

Listing 11.19: Showing the Source Code of the CookieServlet Servlet

```
package com.kogent;
import java.io.*;
import java.util.Random;
import javax.servlet.http.*;
import javax.servlet.ServletException;
public class CookieServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse
  response) throws ServletException, IOException {
        Cookie[] coki = request.getCookies();
        Cookie tokenCookie = null;
        if(coki !=null) {
              for(int i=0; i<coki.length; i++) {
                    if (coki[i].getName().equals("token")) {
                          tokenCookie = coki[i];
                          break;
                    }
              }
        }
        response.setContentType("text/html");
        PrintWriter prnwriter = response.getWriter();
        prnwriter.println("<html><head><title>Extracting the token
              cookie</title></head><body");
        prnwriter.println("style=\"font-family:arial;font-size:12pt\">");
        String resetParam = request.getParameter("resetParam");
```

```
            if(tokenCookie==null || (resetParam != null &&
                  resetParam.equals("yes"))) {
                  Random rnd = new Random();
                  long cookieid = rnd.nextLong();
                  prnwriter.println("<p>Welcome. A new token " + cookieid + "is
                        now established</p>");
                  tokenCookie = new Cookie("token",
                        Long.toString(cookieid));
                  tokenCookie.setComment("A cookie named token to identity
                        user");
                  tokenCookie.setMaxAge(-1);
                  tokenCookie.setPath("/HandleSession/CookieServlet");
                  response.addCookie(tokenCookie);
            }
            else
            {
                  prnwriter.println("Welcome back.. Your token is " +
                        tokenCookie.getValue() + ".</p>");
            }
            String requestURLSame = request.getRequestURL().toString();
            String requestURLNew = request.getRequestURL() + "?resetParam=yes";
            prnwriter.println("<p>Click <a href=" + requestURLSame +"> here
                  </a>again to continue browsing with the "+ " same
                  identity.</p>");
            prnwriter.println("<p>Otherwise, click <a href = " + requestURLNew +
                  "> here</a> to start browsing with a new identity. </p>");
            prnwriter.println("</body></html>");
            prnwriter.close();
      }
}
```

In Listing 11.19, the `CookieServlet` servlet first retrieves all the cookies that are contained in the `request` object. In case the cookies are present, the `CookieServlet` servlet locates the cookie named `token`. If the servlet does not find a cookie with the name `token`, then the Web container, on the `CookieServlet` servlet request, creates a cookie called `token` and adds it to the response. A random number for the cookie is created by the servlet. The servlet creates a cookie with the help of the parameters, as shown in the following code snippet:

```
Name : token
Value :  A random number
Comment : A cookie named tokens to identify user
Max-Age : -1 (The value of -1 indicates that the cookie will be discarded when the
          Browser exits)
Path : /HandleSession/CookieServlet  (This path enables the browser to send the
          cookie to only those requests  under
          http://localhost:8080/HandleSession/CookieServlet
```

If you are deploying this application on a remote machine (server) and accessing it from another machine (client), you need to set the domain name to be that of the server name; by default, it is `localhost`. Compile the `CookieServlet.java` file using the following command:

```
javac -d C:\JavaEE\Chapter11\HandleSession\WEB-INF\classes CookieServlet.java
```

The execution of the preceding command would compile the `CookieServlet` servlet and store the .class file along with package directory under the `WEB-INF\classes` directory of the `HandleSession` application.

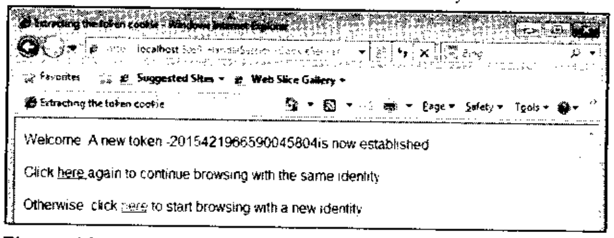
Now, let's create the `web.xml` file to configure and map the `CookieServlet` servlet to the `/CookieServlet` url pattern. Listing 11.20 shows the code of the `web.xml` file (you can find this file on the CD in the `code\JavaEE\Chapter11\HandleSession\WEB-INF` folder):

Listing 11.20: Configuring the CookieServlet Servlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" 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_2_5.xsd">
<servlet>
   <servlet-name>CookieServlet</servlet-name>
```

```
    <servlet-class>com.kogent.CookieServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>CookieServlet</servlet-name>
    <url-pattern>/CookieServlet</url-pattern>
  </servlet-mapping>
  </web-app>
```

Save the code of Listing 11.20 as the `web.xml` file under the WEB-INF directory of the `HandleSession` Web application. Now, package the Web application into `HandleSession.war`, as discussed in the previous sections of the chapter, and deploy it on the Glassfish server. Now, run the servlet in a browser by browsing the `http://localhost:8080/HandleSession/CookieServlet` URL. Figure 11.23 displays the output of `CookieServlet.java`, showing the new token value for the newly created cookie named *token*:

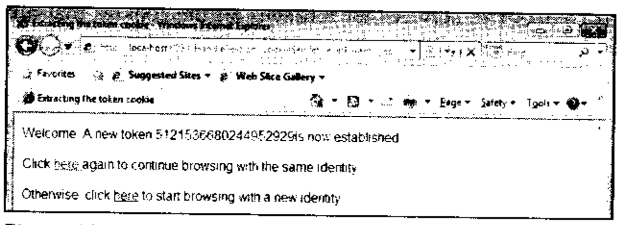


```
Welcome  A new token -2015421966590045804is now established

Click here again to continue browsing with the same identity

Otherwise  click here to start browsing with a new identity
```

Extracting the token cookie - Windows Internet Explorer
Favorites
Suggested Sites
Web Slice Gallery
Extracting the token cookie
Page
Safety
Tools

Figure 11.23: Displaying the Output of CookieServlet Servlet

In Figure 11.23, there are two hyperlinks. Click the first hyperlink to browse through the same identity of the cookie. However, to browse with the new identity, click the second hyperlink. When you click the second hyperlink, a new identity is created, with the new cookie value and it is displayed on the browser. After clicking the second hyperlink, the `resetParam` parameter is set to `yes`, as shown in the address bar in Figure 11.24:

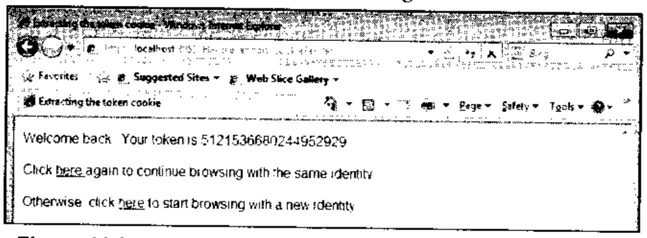


```
Welcome  A new token 5121536680244952929is now established

Click here again to continue browsing with the same identity

Otherwise  click here to start browsing with a new identity
```

Extracting the token cookie - Windows Internet Explorer
Favorites
Suggested Sites
Web Slice Gallery
Extracting the token cookie
Page
Safety
Tools

Figure 11.24: Browsing CookieServlet with New Token Value

Figure 11.24 displays the new token value. Now, to continue with the same token value, as shown in Figure 11.24, click the first hyperlink. After clicking the first hyperlink, the same request URL is retrieved and the browser displays the `Welcome back` message, as shown in Figure 11.25:



```
Welcome back   Your token is 5121536680244952929

Click here again to continue browsing with the same identity

Otherwise  click here to start browsing with a new identity
```

Browsing the token cookie - Windows Internet Explorer
Favorites
Suggested Sites
Web Slice Gallery
Extracting the token cookie
Page
Safety
Tools

Figure 11.25: Browsing CookieServlet with the Same Identity

After discussing how to use cookies, let's now discuss another approach used for session tracking, the hidden form fields.

Hidden Form Fields

The hidden form fields are the fields in a Hypertext Markup Language (HTML) or JavaServer Pages (JSP) form that are not shown to the user and used to store information about a session. You can use the following syntax to use hidden form fields in an HTML page:

```
<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">
```

In the preceding syntax, the `hidden` value is assigned to the `type` attribute, which implies that the input field is a hidden form field. However, using hidden form fields has a major disadvantage, that is, hidden form fields only work when every page is dynamically generated by a form submission. Therefore, hidden form fields cannot support general session tracking and can only support tracking within a specific series of operations.

URL Rewriting

The mechanism of URL rewriting is similar to that of cookies. The URL rewriting mechanism uses the `encodeURL()` method of the response object to encode the session ID into the URL path of a request. The following code snippet shows an example of URL rewriting in which the name of the path parameter is `jsessionid`:

```
http://host:port/myapp/index.html?jsessionid=6789
```

The server uses the value of the rewritten URL to find the session state information, and to pass the information to the servlet. This is similar to the functionality of cookies. Although, cookies are typically enabled; however, to ensure session tracking using URL rewriting, the `encodeURL()` method is used in the servlets. In addition, the `encodeRedirectURL()` method is used in servlets to redirect to a resource.

According to the Servlet specification, if cookies are enabled, then any call to the `encodeURL()` and `encodeRedirectURL()` methods does not result in any action. In case the cookies are disabled, the servlet can call the `encodeURL()` method of the response object to append a session ID to the URL path for each request. Alternatively, the `encodeRedirectURL()` method is used to redirect a Web page to a resource. As a result, URL rewriting helps in associating the request with the session. URL rewriting is the most commonly used mechanism for session tracking in cases when clients do not accept cookies.

Instead of embedding session information within the forms by using the hidden form fields, URL rewriting stores session details as a part of the URL itself. The following code snippet shows various ways in which the information for a servlet can be requested by using URL rewriting:

```
[1]     http://www.acknowledge.co.uk/Servlet/search
[2]     http://www.acknowledge.co.uk/Servlet/search/23434abc
[3]     http://www.acknowledge.co.uk/Servlet/search?sesID=23434abc
```

In [1], URL rewriting has not been done rather the URL requests for the `search` servlet mapped to `/search` in `web.xml`. In [2], URL has been rewritten at the server to add extra path information. This extra information is embedded in the pages returned to the client. The following code snippet shows how to retrieve the extra path information, provided in [2]:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
   ...
   String sesID = request.getPathInfo(); // return 2343abc from [2]
   ...
}
```

The extra path information works for both the `GET` and the `POST` methods in forms and outside the forms with static links. While using technique [3], you can simply rewrite the URL with parameter information, as shown in the following code snippet:

```
request.getParameterValue("sesID");
```

Similar to hidden form fields, URL rewriting provides a means to implement anonymous session tracking. URL rewriting is not limited to forms only. URLs can also be rewritten in static documents to contain the required session information. However, URL rewriting suffers from a disadvantage that the URLs must be dynamically generated and most importantly, the chain of HTML page generation cannot be broken. Therefore, URL rewriting is a tedious and error-prone process.

The mechanism of URL rewriting can be better understood by creating a servlet. Let's create the `TokenServlet` servlet, which performs the following tasks:

- ❑ Checks whether or not the client sends a token with its request
- ❑ Creates a new token, if no token has been sent by the client

In addition, the `TokenServlet` servlet provides two hyperlinks—one that includes the token and other does not.

Listing 11.21 shows the code of the TokenServlet.java file (you can find this file on the CD in the code\JavaEE\Chapter11\HandleSession\src\com\kogent folder):

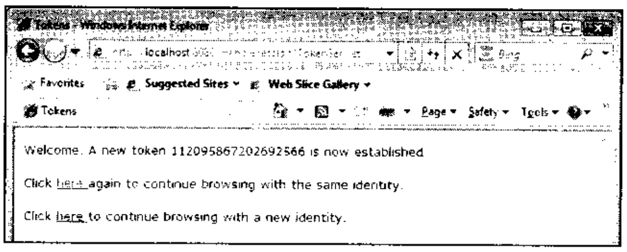
Listing 11.21: Implementing the URL Rewriting Mechanism

```
package com.kogent;
import java.io.*;
import java.util.Random;
import javax.servlet.http.*;
import javax.servlet.ServletException;
public class TokenServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse
   response) throws ServletException, IOException {
            String tokensID = request.getParameter("tokens");
            response.setContentType("text/html");
            PrintWriter prnwriter = response.getWriter();
            prnwriter.println("<html><head><title>Tokens</title></head><body");
            prnwriter.println("style=\"font-family:verdana;font-size:10pt\">");
            if(tokensID==null) {
                     Random rnd = new Random();
                     tokensID = Long.toString(rnd.nextLong());
                     prnwriter.println("<p>Welcome. A new token " + tokensID + "
                     is now established</p>");
            }
            else {
                     prnwriter.println("Welcome back. Your token is " + tokensID
                     + ".</p>");
            }
            String requestURLSame = request.getRequestURL().toString()+"?tokens=
            " + tokensID;
            String requestURLNew = request.getRequestURL().toString();
            prnwriter.println("<p>Click <a href=" + requestURLSame +" > here
            </a> again to continue browsing with the same identity.</p>");
            prnwriter.println("<p>Click <a href=" + requestURLNew +" > here </a>
            to continue browsing with a new identity.</p>");
            prnwriter.println("</body></html>");
            prnwriter.close();
   }
}
```

Save the code of Listing 11.21 as the TokenServlet.java file in the src\com\kogent directory of the HandleSession Web application. When the here hyperlink is clicked in the TokenServlet servlet, the URL path is retrieved using the getRequestURL() method and the URL is rewritten with the parameter information. Compile the TokenServlet servlet and configure it in the web.xml file, created in Listing 11.20 for the HandleSession Web application. The following code snippet shows how to configure and map the TokenServlet servlet to the /TokenServlet url pattern:

```
<servlet>
   <servlet-name>TokenServlet</servlet-name>
   <servlet-class>com.kogent.TokenServlet</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>TokenServlet</servlet-name>
   <url-pattern>/TokenServlet</url-pattern>
</servlet-mapping>
```

You can run the TokenServlet servlet only after packaging the HandleSession Web application in the HandleSession.war file. Then, you need to redeploy the Web application on the Glassfish server and navigate the http://localhost:8080/HandleSession/TokenServlet URL. After navigating this URL, you can notice that the query parameter, tokens, is not included in the initial request. The servlet not only creates a new token but also generates two links. A query string is included in one link, which is passed as a query parameter in the request. Therefore, the servlet can recognize the user from this parameter and display the Welcome back message. This allows the user to continue browsing with the same identity. Another link, that does not include the query parameter, is also generated by the servlet. This link allows the user to continue browsing with a new identity, every time the link is clicked. Figure 11.26 shows the output of the `TokenServlet.java` servlet when `it` is executed for the first time:

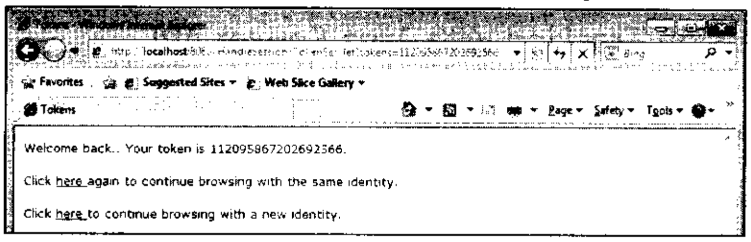

Welcome. A new token 112095867202692566 is now established

Click here again to continue browsing with the same identity.

Click here to continue browsing with a new identity.

Figure 11.26: Displaying the Output of TokenServlet Servlet

After clicking the first link shown in Figure 11.26, the address bar of the browser (Figure 11.27) shows rewritten URL including the `tokens` parameter in the request. The `tokens` parameter added in the URL contains the new token identity based on the user's requests. Figure 11.27 shows the output when the first link is clicked:

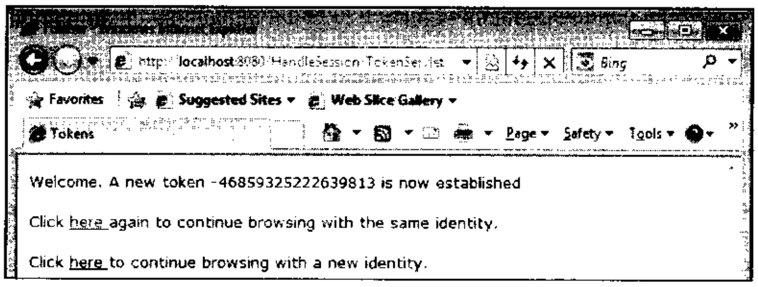

Welcome back.. Your token is 112095867202692566.

Click here again to continue browsing with the same identity.

Click here to continue browsing with a new identity.

Figure 11.27: Displaying the Output of TokenServlet Servlet Using URL Rewriting

Clicking the second link allows the user to continue browsing with a new identity. Figure 11.28 shows the output when the second link is clicked:


Welcome. A new token -46859325222639813 is now established

Click here again to continue browsing with the same identity.

Click here to continue browsing with a new identity.

Figure 11.28: Displaying the Output of Browsing the TokenServlet Servlet with a New Identity

Although, the URL rewriting mechanism can solve the problem of session tracking, it has two main limitations:

- It is not a secure session tracking mechanism as the token or session id is visible in the URL during a session
- It can only be used with servlets or other dynamic pages as the links in static pages are hard coded and cannot change dynamically for every user

You can use cookies as a session tracking mechanism to overcome the preceding problems of the URL rewriting mechanism.

Session using Secure Socket Layer (SSL)

SSL is used to protect the data during transmission that covers all network services. This layer uses Transmission Control Protocol (TCP)/Internet Protocol (IP) to support typical application tasks that require communication between clients and servers. It is an encryption technology that runs on top of TCP/IP and below application level protocols, such as HTTP. SSL ensures the security of data transported and routed through HTTP. SSL is designed to utilize TCP as a communication layer protocol to provide a dependable, uninterrupted, secure, and authenticated connection between two points over a network. It is used mostly in an HTTP server and client applications. Almost each available HTTP server can support an SSL session. Figure 11.29 shows SSL between application protocols and TCP/IP:

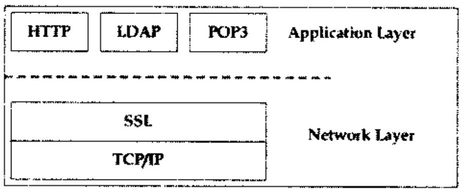


Figure 11.29: Displaying SSL between Application Protocols and TCP/IP

Apart from the cookies, hidden form fields, URL rewriting, and SSL mechanisms, the session details of a user can also be tracked by using Java Servlet API. The `HttpSession` interface in Java Servlet API also helps in implementing session tracking. Let's discuss the `HttpSession` interface in the next section and learn how to use it to implement session tracking.

Describing Session Tracking with Java Servlet API

Session tracking is a mechanism used to maintain the state of a user within the lifetime of a session. In other words, session tracking is a means to keep track of session data, which represents the data being transferred in a session. As the HTTP protocol is a stateless protocol, the Web container needs to manage the session data in a Web application.

Session tracking is used when session data of a user might be required by a Web server to complete specific operations in the current session. For example, suppose you are shopping on an online book store. You access the online book store Web site and add items to the shopping cart. When you proceed for checkout, then due to session tracking the server would be known that the user who added items to the shopping cart is logging out. The following subsections briefly discuss about history of session tracking and provide a description about how to create and track a session.

History of Session Tracking

Developers maintain the session state by providing user information into hidden form fields on an HTML page. In addition, they can maintain user's session by embedding the user activities into URLs with a long String of appended characters. You can find good examples of embedding user activities into URLs mostly in search engine sites, which still depend on CGI. These URLs contain the URL parameter name/value pairs that are appended after the reserved HTTP character `?`. The search engine sites use the URL parameter name/value pairs to track the user choices. However, appending URL parameters can result in a very long URL that needs to be carefully parsed and managed by the CGI script. The URL parameter name/value pairs cannot be passed through URL from session to session. Once the control over the URL is lost, that is once the user leaves one of the pages, the user information is lost forever.

Later, browser cookies were introduced by Netscape, which can be used by each server to store user related information on the client side. However, one of the drawbacks of using cookies is that cookies are not fully supported by some browsers and most browsers limit the amount of data that can be stored with a cookie.

To overcome the shortcomings faced while using cookies and to maintain the user session, the HTTP Servlet specification was introduced. The HTTP Servlet specification protects the code from the complexities of tracking sessions. Servlets may use the HttpSession object to track the input provided by the user over the span of a single session as well as to share the session details with other servlets.

Session Creation and Tracking

An instance of a class that implements the `javax.servlet.http.HttpSession` interface represents each client session in the standard Servlet API. The servlets can use the `HttpSession` object to set or get the information about the session which must be of the application-level scope. A servlet can retrieve or create the `HttpSession` object for the user by calling the `getSession()` method. The `getSession()` method accepts a boolean argument that specifies whether to create a new session object for the client if no session already exists within the application. Let's now learn how to create a session.

Creating a Session

A prospective session that has not yet been established is considered new. As HTTP protocol is request-response based; therefore, the HTTP session is considered as new until it is joined by a client. The client is considered to have joined the session when session tracking information is returned to the server indicating that the session has been successfully established. Any next request from the client is not recognized as a part of the session until the client joins the session. A session is considered to be new, in either of the following cases:

- The client does not have knowledge about the session
- The client opts for not joining the session

In both the cases, the servlet container could not correlate a request with the previous request by any means. Therefore, a servlet developer must design the application in such a manner that it could handle a situation where a client has not yet joined a session or will not join a session.

As explained earlier, the servlet container uses the HTTP session objects that implement the `javax.servlet.http.HttpSession` interface to track and manage the user sessions. The `HttpSession` interface contains public methods, such as `setAttribute()` and `getAttribute()`, to set as well as get the session information, respectively. The following code snippet shows the syntax of the `setAttribute()` method of the `HttpSession` interface:

```
void setAttribute(String name, Object value)
```

The `setAttribute()` method binds the specified object under the specified name, to the session. The following code snippet shows the syntax of the `getAttribute()` method of the `HttpSession` interface:

```
Object getAttribute(String name)
```

The `getAttribute()` method retrieves the object that is attached to the session with the specified name. A `null` value is returned if there is no match. According to the configuration of a servlet as well as the servlet container, sessions may automatically expire after the specified time or the servlet may invalidate the session explicitly. The following methods can be used by servlets to manage the session life-cycle specified by the `HttpSession` interface:

- `void invalidate()` — Invalidates the session instantly and unbinds any bound objects from the session.
- `void setMaxInactiveInterval(int interval)` — Sets a session timeout interval as an integer value in seconds. Timeout cannot be indicated by a negative value. A value of 0 results in immediate timeout.
- `boolean isNew()` — Returns `true` if a new session is created within the request that creates a new session; otherwise, it returns `false`.
- `long getCreationTime()` — Returns the time of creation of the session object. The time is measured in milliseconds and is calculated since midnight, January 1, 1970.
- `long getLastAccessedTime()` — Returns the time associated with the most recent request made by the client during the session. The time is measured in milliseconds and is calculated since midnight, January 1, 1970. Session creation time is returned by the method if the client session has not yet been accessed.

Let's create a servlet to understand the implementation of the HTTP session object. Listing 11.22 provides the code of the `MySessionServlet` servlet, which creates the `HttpSession` object and prints the data held by the request and session objects (you can find the `MySessionServlet.java` file on the CD in the `code\JavaEE\Chapter11\HandleSession\src\com\kogent` folder):

Listing 11.22: Showing the Code of the MySessionServlet.java File

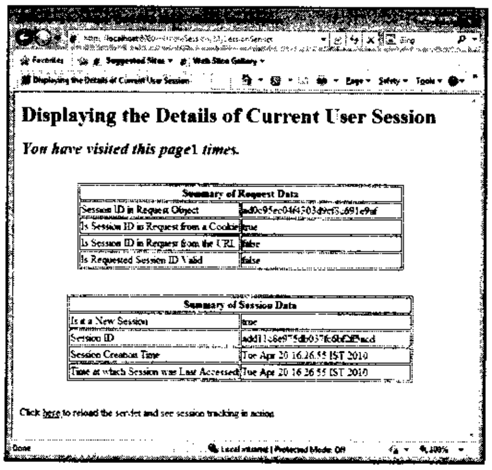
```
package com.kogent ;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;
public class MySessionServlet extends HttpServlet {
  public void doGet (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
```

```
        // Retrieve the session object for the current user session.
// A new session is created if a session object has not been
// created previously.
HttpSession sessionObj = req.getSession(true);
// Create and update the variable that holds a count indicating
// the number of times the page has been visited during the current
// user session.
Integer count = (Integer) sessionObj.getAttribute("count");
        if (count == null) {
        count = new Integer(1);
        }
        else {
                count = new Integer(count.intValue() + 1);
        }
        // Save the updated count value to the current user session
        sessionObj.setAttribute("count", count);
// Displaying the output to the user
        res.setContentType("text/html");
PrintWriter prnwriter = res.getWriter();
prnwriter.println("<head><title> " + "Displaying the Details of
Current User Session" + "</title></head><body>");
prnwriter.println("<h1>Displaying the Details of Current User
Session</h1>");
prnwriter.println("<h2><I>You have visited this page</I><b><font
color=\"blue\">" + count + "</font></b><I> times.</I></h2><p>");
prnwriter.println("<BR>");
// Displaying the request related information from the request
// object
prnwriter.println("<Table ALIGN=CENTER Border=\"1\"
BorderColor=\"Red\">");
prnwriter.println("<TH COLSPAN=2 ALIGN=CENTER>Summary of Request
Data</TH>");
prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Session ID in Request Object");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(req.getRequestedSessionId());
prnwriter.println("</TD></TR>");
prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Is Session ID in Request from a Cookie");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(req.isRequestedSessionIdFromCookie());
prnwriter.println("</TD></TR>");
prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Is Session ID in Request from the URL");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(req.isRequestedSessionIdFromURL());
prnwriter.println("</TD></TR>");
prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Is Requested Session ID Valid");
prnwriter.println("</TD>");
prnwriter.println("<TD WIDTH=\"50%\">");
prnwriter.println(req.isRequestedSessionIdValid());
prnwriter.println("</TD></TR>");
prnwriter.println("</Table>");
prnwriter.println("<BR><BR>");
// Displaying the session related information from the session
// object
prnwriter.println("<Table ALIGN=CENTER Border=\"1\"
BorderColor=\"Red\">");
prnwriter.println("<TH COLSPAN=2 ALIGN=CENTER>Summary of Session
Data</TH>");
prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
VALIGN=CENTER>");
prnwriter.println("Is it a New Session");
prnwriter.println("</TD>");
```

```
            prnwriter.println("<TD WIDTH=\"50%\">");
            prnwriter.println(sessionObj.isNew());
            prnwriter.println("</TD></TR>");
            prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
            VALIGN=CENTER>");
            prnwriter.println("Session ID");
            prnwriter.println("</TD>");
            prnwriter.println("<TD WIDTH=\"50%\">");
            prnwriter.println(sessionObj.getId());
            prnwriter.println("</TD></TR>");
            prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
            VALIGN=CENTER>");
            prnwriter.println("Session Creation Time");
            prnwriter.println("</TD>");
            prnwriter.println("<TD WIDTH=\"50%\">");
            prnwriter.println(new Date(sessionObj.getCreationTime()));
            prnwriter.println("</TD></TR>");
            prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
            VALIGN=CENTER>");
            prnwriter.println("Time at which Session was Last Accessed");
            prnwriter.println("</TD>");
            prnwriter.println("<TD WIDTH=\"50%\">");
            prnwriter.println(new Date(sessionObj.getLastAccessedTime()));
            prnwriter.println("</TD></TR>");
            prnwriter.println("</Table>");
            prnwriter.println("<BR><BR>");
                // User can reaccess the servlet using this hyperlink to see
            // Session Tracking in action. Clicking the hyperlink refreshes
            // the page and increment the session variable count by one every
            // time it is clicked.
            String url = req.getRequestURL().toString();
                prnwriter.println(" Click <a href=" + url + "> here </a> to reload
            the servlet and see session tracking in action. <br>");
            prnwriter.println("</body>");
                prnwriter.close();
    }
}
```

Save `MySessionServlet` in the `src\com\kogent` directory of the `HandleSession` Web application. Now, configure and map the `MySessionServlet` servlet to the `/MySessionServlet` url pattern in the `web.xml` file and then create `HandleSession.war`. Redeploy the `HandleSession` Web application on the `Glassfish` server and browse the URL `http://localhost:8080/HandleSession/MySessionServlet` to see the output of the `MySessionServlet` servlet.

Figure 11.30 shows the output of the `MySessionServlet` servlet, displaying the results of the accessor methods on the request and session objects as well as listing request and session data:



Displaying the Details of Current User Session

*You have visited this page*1 *times.*

Summary of Request Data	
Session ID in Request Object	ad0c95ec04f4303d9ef5c691e9af
Is Session ID in Request from a Cookie	true
Is Session ID in Request from the URL	false
Is Requested Session ID Valid	false

Summary of Session Data	
Is it a New Session	true
Session ID	add11c6e975db037fc6bf2f?acd
Session Creation Time	Tue Apr 20 16:26:55 IST 2010
Time at which Session was Last Accessed	Tue Apr 20 16:26:55 IST 2010

Click here to reload the servlet and see session tracking in action.

Favorites
Suggested Sites
Web Slice Gallery
Displaying the Details of Current User Session
Page
Safety
Tools
Done
Local intranet | Protected Mode: Off
100%

Figure 11.30: Displaying the Output of MySessionServlet Servlet

In Figure 11.30, if a user clicks the here link and the cookies are enabled in the browser settings, then the request and session data get changed. In such a case, change the Web browser settings, for example, disable cookies and click the `here link` that causes URL rewriting.

Tracking a Session Using Servlet API

The Servlet API provides various methods and classes, such as `getSession()` and `HttpSession` that are particularly designed to handle session tracking.

The session tracking API, which is the part of the Servlet API and provides session tracking functionality, can be used in any Web server that supports servlets. However, the level of support depends on the server.

For example, session objects can be written to Java Web Server even if the server disk memory fills up or the server shuts down. However, to take advantage of this option; the items that are placed in the session are required to implement the `Serializable` interface.

Exploring Session Tracking Basics

In session tracking, the user identity is linked with the `javax.servlet.http.HttpSession` object that can be used by the servlets to store or retrieve information about that user. Any set of arbitrary Java objects can be saved in a session object. For example, to store the user's shopping cart content in a database, a user's current HttpSession instance is retrieved by using the `getSession()` method of the `HttpServletRequest` interface, as shown in the following code snippet:

```
public HttpSession HttpServletRequest.getSession(boolean create)
```

The current session that is associated with the user who makes the request, is returned by the `getSession()` method. If the `create Boolean` variable has a value `true`, the `getSession()` method creates the session; otherwise, the method returns null. The `getSession()` method must be called at least once before writing any output to the response to ensure that the session is maintained properly.

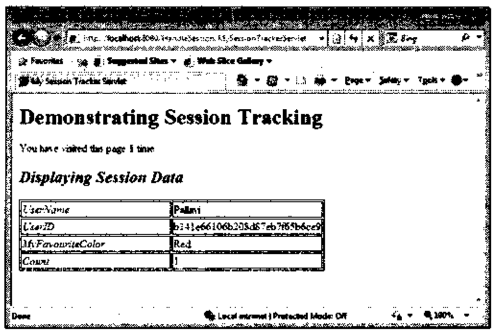
Let's create another servlet, `MySessionTrackerServlet`, to have a better understanding of the concept of session tracking. The `MySessionTrackerServlet` servlet fetches and prints all the attributes associated with the current user session along with a count for the number of times the servlet has been visited by the user during the session. Listing 11.23 shows the code of the `MySessionTrackerServlet.java` file (you can find this file on the CD in the `code\JavaEE\Chapter11\HandleSession\src\com\kogent` folder):

Listing 11.23: Showing the Code of the `MySessionTrackerServlet.java File`

```
package com.kogent ;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class MySessionTrackerServlet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {
        // Retrieve the session object for the current user session.
        // A new session is created if a session object has not been
        // created previously.
        HttpSession sessionObj = req.getSession(true);
        // Create and update the variable that holds a count indicating
        // the number of times the page has been visited during the current
        // user session.
        Integer count = (Integer) sessionObj.getAttribute("Count");
                if (count == null) {
                count = new Integer(1);
                }
                else {
                count = new Integer(count.intValue() + 1);
                }
                // Save the updated count value to the current user session
                sessionObj.setAttribute("Count", count);
        // Add some more attributes to the current user session
        sessionObj.setAttribute("UserName", "Pallavi");
        sessionObj.setAttribute("UserID", sessionObj.getId());
        sessionObj.setAttribute("MyFavouriteColor", "Red");
        // Displaying the output to the user
            res.setContentType("text/html");
            PrintWriter prnwriter = res.getWriter();
            prnwriter.println("<HTML><HEAD><TITLE>My Session Tracker
        Servlet</TITLE></HEAD>");
            prnwriter.println("<BODY><H1>Demonstrating Session Tracking</H1>");
            // Display the hit cnt for this page for the current user
            prnwriter.println("You have visited this page <b><font
        color=\"blue\">" + count + ((count.intValue()==1)?"</font></b>
        time." : "</font></b> times."));
            prnwriter.println("<P>");
            prnwriter.println("<H2><I>Displaying Session Data</I></H2>");
        prnwriter.println("<Table Border=\"1\" BorderColor=\"Blue\">");
            Enumeration names = sessionObj.getAttributeNames();
            while(names.hasMoreElements()) {
            String name = (String) names.nextElement();
            prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT
            VALIGN=CENTER><I>");
            prnwriter.println(name);
            prnwriter.println("</I></TD><TD>");
            prnwriter.println(sessionObj.getAttribute(name));
            prnwriter.println("</TD></TR>");
            }
        prnwriter.println("</TABLE>");
            prnwriter.println("</BODY></HTML>");
    }
  }
```

The `MySessionTrackerServlet` servlet tracks the number of times the client accesses it. When the client accesses the `MySessionTrackerServlet` servlet for the first time, the new session is created and each time the client accesses it, the value of `count` variable is incremented by one and the browser displays the number of times the client has accessed the servlet. The other variables related to the current client session are also displayed by the servlet.

Save the `MySessionTrackerServlet` servlet in the `src\com\kogent` directory of the `HandleSession` Web application. Configure the `MySessionTrackerServlet` servlet in the `web.xml` file. Then, compile the `MySessionTrackerServlet` servlet, package the `HandleSession application`, and redeploy the Web application on the `Glassfish` server. Browse the `http://localhost:8080/HandleSession/MySessionTrackerServlet` URL to view the output, as shown in Figure 11.31:



Demonstrating Session Tracking

You have visited this page 1 time

Displaying Session Data

UserName	Pallavi
UserID	b141e66106b205d67eb7f65b6ce9
MyFavouriteColor	Red
Count	1

Figure 11.31: Displaying the Output of the MySessionTrackerServlet Servlet

The `MySessionTrackerServlet` servlet first retrieves the `HttpSession` object linked with the current client session. The `getSession()` method, with a `Boolean` value of `true` as an argument, creates a session if it does not exist for the user. The servlet then fetches and sets the value of the `Integer` object `count`, which is bound to the current user session with a name `Count`. The servlet initializes the `Count` variable, if it is previously `null`. Otherwise, the servlet increments the value of the `Count` variable by one and resets it for the user session. The servlet also fetches the values of other variables associated with the current user session. Finally, the servlet displays the value for the count variable and all the name/value pairs for the variables associated with the current user session.

Demonstrating Session Life-Cycle with Cookies

Let's now explore the life-cycle of HttpSession objects, by creating a servlet, TrackSessionLifeCycle. The servlet examines certain session attributes and provides link to invalidate the existing session. We will also examine the behavior of the servlet in the absence of cookies, and then discuss the ways to generate Web pages that work as desired, irrespective of whether cookies are accepted by the client browser or not.

The TrackSessionLifeCycle servlet generates a page that displays session object data including session ID, creation time of the session object, last accessed time, and max inactive interval for the session. The TrackSessionLifeCycle servlet also provides links to reload the page and invalidate the current user session. Listing 11.24 shows the code of the TrackSessionLifeCycle.java file (you can find this file on the CD in the code\JavaEE\Chapter11\HandleSession\src\com\kogent folder):

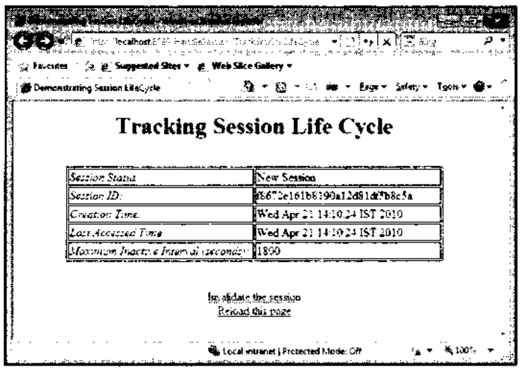
Listing 11.24: Displaying the Code of the TrackSessionLifeCycle Class

```
package com.kogent ;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class TrackSessionLifeCycle extends HttpServlet {
  protected void doGet (HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {
        String url = "/HandleSession/TrackSessionLifeCycle";
        String reqAction = req.getParameter("requestAction");
        HttpSession sessionObj = req.getSession();
        res.setContentType("text/html");
        PrintWriter prnwriter = res.getWriter();
        prnwriter.println("<html>");
        prnwriter.println("<head> <title>Demonstrating Session
        LifeCycle</title></head>");
        prnwriter.println("<body>");
        if (reqAction != null && reqAction.equals("invalidate")) {
               sessionObj.invalidate();
               prnwriter.println("<center><p>Your session has been
               invalidated.</p>");
               prnwriter.println("Would you like to <a href=\"" + url +
               "?requestAction=createNewSession\">");
               prnwriter.println("create a new session</a>");
        }
        else {
               prnwriter.println("<h1><center>Tracking Session Life
               Cycle</center></h1>");
               prnwriter.println("<br><Table ALIGN=CENTER Border=\"1\"
               BorderColor=\"Blue\">");
               prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
               prnwriter.println("Session Status");
               prnwriter.println("</I></TD><TD>");
               if (sessionObj.isNew()) {
                      prnwriter.println("New Session");
               }
               else {
                      prnwriter.println("Old Session");
               }
               prnwriter.println("</TD></TR>");
               prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
               prnwriter.println("Session ID:");
               prnwriter.println("</I></TD><TD>");
               prnwriter.println(sessionObj.getId());
               prnwriter.println("</TD></TR>");
               prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
```

```
                    prnwriter.println("Creation Time:");
                    prnwriter.println("</I></TD><TD>");
                    prnwriter.println(new Date(sessionObj.getCreationTime()));
                    prnwriter.println("</TD></TR>");
                    prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
                    prnwriter.println("Last Accessed Time:");
                    prnwriter.println("</I></TD><TD>");
                    prnwriter.println(new
                    Date(sessionObj.getLastAccessedTime()));
                    prnwriter.println("</TD></TR>");
                    prnwriter.println("<TR><TD WIDTH=\"50%\" ALIGN=LEFT><I>");
                    prnwriter.println("Maximum Inactive Interval
                    (seconds):");
                    prnwriter.println("</I></TD><TD>");
                    prnwriter.println(sessionObj.getMaxInactiveInterval());
                    prnwriter.println("</TD></TR>");
                    prnwriter.println("</TABLE>");
                    prnwriter.println("<br><br><center><a href =\"" + url +
                    "?requestAction=invalidate\">");
                    prnwriter.println("Invalidate the session</a>");
                    prnwriter.println("<br><center><a href =\"" + url + "\">");
                    prnwriter.println("Reload this page</a>");
                }
                prnwriter.println("</body></html>");
                prnwriter.close();
        }
}
```

Save the code of Listing 11.24 as the TrackSessionLifeCycle.java file in the src\com\kogent directory of the HandleSession application. Configure the TrackSessionLifeCycle servlet in the web.xml file and map it to the /TrackSessionLifeCycle url pattern. After configuring, compile the TrackSessionLifeCycle servlet, package the application into HandleSession.war, and redeploy the HandleSession application. To run the servlet, browse http://localhost:8080/HandleSession/TrackSessionLifeCycle.

Figure 11.32 displays the output of TrackSessionLifeCycle showing the new session, since the user is accessing this page for the first time:


Favorites
Suggested Sites
Web Slice Gallery
Demonstrating Session LifeCycle
Page
Safety
Tools
Local intranet | Protected Mode: Off
100%

Tracking Session Life Cycle

Session Status	New Session
Session ID:	8672e161b8190a12d81df7b8c5a
Creation Time.	Wed Apr 21 14:10:24 IST 2010
Last Accessed Time	Wed Apr 21 14:10:24 IST 2010
Maximum Inactive Interval (seconds)	1800

Invalidate the session
Reload this page

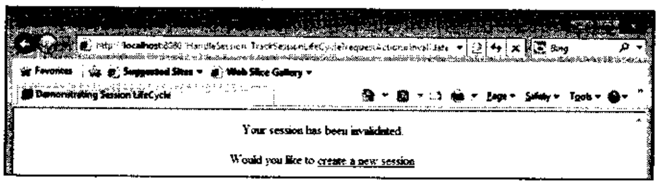
Figure 11.32: Displaying the Output of TrackSessionLifeCycle Servlet

The else block provided in Listing 11.24 is executed when the servlet is invoked without any parameters and performs the following steps:

- Calls the isNew() method to check for the status of the session that whether it is new or old.
- Calls the getSession() method on the HttpSession object with the true Boolean value passed as an argument.
- Calls the getId() method to get the session ID.

- Calls the `getCreationTime()` method to get the creation time of the session. When this method returns the creation time as milliseconds since January 1, 1970 00:00:00 GMT, the returned value needs to be converted into a `Date` object by using the new constructor.
- Calls the `getLastAccessedTime()` method to get the time the session was last accessed.
- Calls the `getMaxInactiveInterval()` method to get the current max-inactive setting.

After printing the details, such as `sessionID`, creation time of the session, last accessed time of the session, the servlet generates two links, one to invalidate the session and the other to reload the page. The first link has query string as `requestAction=invalidate` that is appended to the URL. When you click the `Invalidate the session` link, the `if` block of the `doGet()` method is executed. However, the second link simply points to the same page. Ensure that cookies are enabled in your browser configuration so that cookies can be stored on your system. Figure 11.33 displays the browser's output, when the `Invalidate the session` link is clicked:

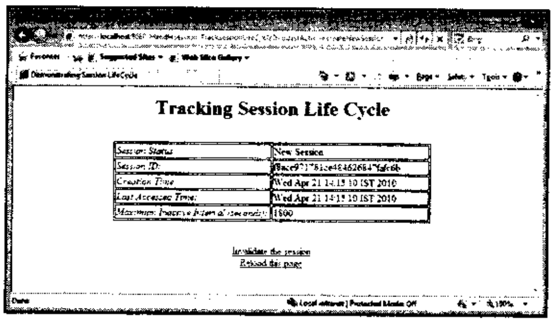


Your session has been invalidated.

Would you like to create a new session

Figure 11.33: Displaying a Message Depicting the Expiration of a Session

Now, when you click the `create a new session` link, a new session is created, as shown in Figure 11.34:


Favorites
Suggested Sites
Web Slice Gallery
Demonstrating Session LifeCycle
Page
Safety
Tools
Local intranet | Protected Mode: Off
100%

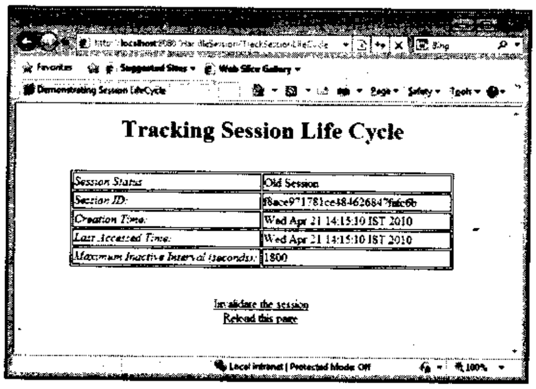
Tracking Session Life Cycle

Session Status	New Session
Session ID:	f8ace971781ce48462684?fafc6b
Creation Time:	Wed Apr 21 14:15:10 IST 2010
Last Accessed Time:	Wed Apr 21 14:15:10 IST 2010
Maximum Inactive Interval (seconds):	1800

Invalidate the session
Reload this page

Figure 11.34: Creating a new session

Now, let's see what happens if the code in the `else` block is executed again. Click the `Reload this page` link. You find that the session status displays that the session is old. Figure 11.35 displays the output when the `Reload this page` link is clicked:


Favorites
Suggested Sites
Web Slice Gallery
Demonstrating Session LifeCycle
Page
Safety
Tools
Local intranet | Protected Mode: Off
100%

Tracking Session Life Cycle

Session Status	Old Session
Session ID:	f8ace971781ce48462684?fafc6b
Creation Time:	Wed Apr 21 14:15:10 IST 2010
Last Accessed Time:	Wed Apr 21 14:15:10 IST 2010
Maximum Inactive Interval (seconds):	1800

Invalidate the session
Reload this page

Figure 11.35: Reloading the Old Session

You may notice in Figure 11.35 that the session ID and the session creation time are same as they were before reloading the page, as shown in Figure 11.34. Therefore, every time you click the `Reload this page` link, only the last accessed time changes and not the session ID and the session creation time.

This illustrates how simple it is to create and keep track of sessions. Now, let's see what happens if you click the `Invalidate the session` link. As this URL has a query parameter `action=invalidate`, the `if` block of the `doGet()` method will be executed to invalidate the created session.

Now, click the `Create new session` link. The resulting page should be similar to Figure 11.34. Examine the `if` block of the code of Listing 11.24. This part of the `TrackSessionLifeCycle` servlet gets the session from the request, calls the `invalidate()` method, and generates a new link back to the previous page.

Now let's create a login application using Session Tracking API. In this application, a user logs on with the username and password and after logging into the application, the user browses the session details. Finally, the user logs out and the session is terminated.

Working with Login Application using Session Tracking

Let's create the login Web application, in which the user logs on with the username and password, and then creates a session. Till the session is active, the user can browse the session details; however, once the user logs out, the session becomes invalid.

Let's name the Web application as `LoginApp` to demonstrate session tracking. This application has an HTML page and two servlets. These will be configured in the web.xml file. You should know the directory structure of the application before creating the application. Therefore, let's first discuss the directory structure of the `LoginApp` application. The directory structure of the LoginApp Web application would make it easy to understand where to save the Java, HTML, and configuration files.

Exploring the Directory Structure of Login Application

The `servlets`, `HTML` pages, and the `Cascading Style Sheet (CSS)` files of the LoginApp application are stored under a base directory of the application. Create a folder for your application say, `LoginApp`, in the C:\JavaEE\Chapter11 folder. Create some more folders, such as `WEB-INF`, `WEB-INF\classes`, `WEB-INF\lib`, and `src` (Figure 11.36). Store different types of files at the proper location in the directory structure, as described in the following statements:

- All packages containing class files are stored in the WEB-INF\classes folder.
- The configuration file, such as web.xml, is stored in the WEB-INF folder.
- All source files (.java files) can be stored in the src\com\kogent folder. This folder is optional in your application and you can store your source files at any other location.

Figure 11.36 displays the directory structure of the LoginApp application:

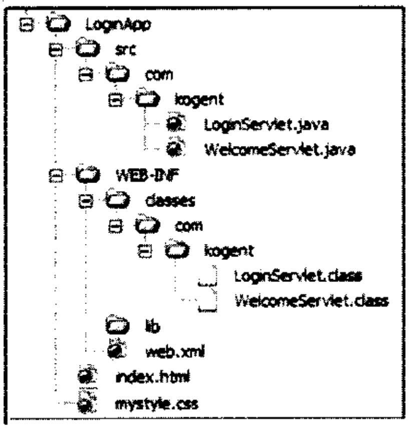


Figure 11.36: Displaying the Root Directory Structure for LoginApp Web application

As shown in Figure 11.36, LoginApp is the root folder containing the WEB-INF folder, src folder, and index.html file. The WEB-INF folder has two folders, classes and lib, and a file web.xml. As discussed earlier, the package containing LoginServlet and WelcomeServlet class files is stored at the WEB-INF\classes location under the LoginApp directory. The src\com\kogent is the optional folder containing source files (LoginServlet.java and WelcomeServlet.java). The configuration file, web.xml, is stored in the WEB-INF folder.

Building the Front-End

The HTML page acts as a front-end of the LoginApp Web application. The index.html page displays a login page for the users. On the basis of the user details entered in index.html, a session is maintained. The user details include the username and password. Listing 11.25 shows the code of the index.html file (you can find this file on the CD in the code\JavaEE\Chapter11\LoginApp folder):

Listing 11.25: Showing the Code of the Front-End

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
        <TITLE>Login Application</TITLE>
        <link rel="text/css" href="mystyle.css"/>
  </HEAD>
<BODY>
  <FORM METHOD="POST" ACTION="LoginServlet">
  <H1>Login Application using Session Tracking</H1>
  <TABLE ALIGN=CENTER BORDER="0" >
  <TR>
  <TD VALIGN=TOP ALIGN=RIGHT>
        <B>User ID:</B>
  </TD>
  <TD VALIGN=TOP>
  <B><INPUT NAME="username" TYPE="TEXT" MAXLENGTH= "20" SIZE = "20"></B>
  </TD>
  </TR>
  <TR>
  <TD VALIGN=TOP ALIGN=RIGHT>
  <B>Password:</B>
  </TD>
  <TD VALIGN=TOP>
  <B><INPUT NAME="password" TYPE="Password" MAXLENGTH="20" SIZE = "20"></B>
  </TD>
  </TR>
  <TR><TD VALIGN=CENTER>
  <B><INPUT VALUE = "Log In" TYPE= "SUBMIT"></B>
  </TD>
  </TR>
  </TABLE>
  </FORM>
</BODY>
</HTML>
```

Save the code of Listing 11.25 as the index.html file in the base LoginApp directory and link the mystyle CSS stylesheet to index.html. The style that you want to apply to your HTML page can be stored in the mystyle.css file, which is saved in the base LoginApp directory. The code of mystyle.css is provided on the CD. After entering the login details, the LoginServlet servlet is requested.

Creating and Managing a Session

In the LoginApp Web application, two servlets are created, LoginServlet and WelcomeServlet. The LoginServlet servlet creates a new session for each user and retrieves the data from index.html. Based on the username and password entered by the user in index.html, the LoginServlet servlet sets two new attributes, username and password, in the session. Listing 11.26 provides the code of the LoginServlet.java file (you can find this file on the CD in the code\JavaEE\Chapter11\LoginApp\src\com\kogent folder):

Listing 11.26: Setting the Values of the username and password Attributes

```
package com.kogent;
import javax.servlet.http.*;
import java.io.*;
public class LoginServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,HttpServletResponse response) {
        try {
                String username = request.getParameter("username");
                String password = request.getParameter("password");
                HttpSession session = request.getSession(true);
                PrintWriter writer = response.getWriter();
                if (session.isNew() != true)
                {
                        writer.println("<h1>Session is Active</h1>");
writer.println("<p><a href=\"index.html\">HomePage" + "</a> and return to login page");
                }
                else
                {
                        session.setAttribute("username", username);
                        session.setAttribute("password", password);
                        response.setContentType("text/html");
writer.println("<html><body style=\"font-
family:verdana;font-size:10pt\">");
writer.println("<h1>Login Application using Session Tracking</h1>");
writer.println("<p>Thank you, " + username + ".<p> You are now logged in");
                        String newURL = response.encodeURL("WelcomeServlet");
writer.println("Click <a href=\"" + newURL + "\">here</a> for another servlet");
                        writer.println("</body></html>");
                        writer.close();
                }
        }
        catch (Exception e) {
                e.printStackTrace();
        }
  }
}
```

The LoginServlet servlet checks whether the session is new or not. If the session is not new, the user is prompted to go to the home page as a session already exists. However, if the session is new, the HTML page is designed, which provides a link to the user to browse and the request is forwarded to `WelcomeServlet`. The `WelcomeServlet servlet` displays the session details of the logged in user. Listing 11.27 provides the code of `WelcomeServlet` (you can find this file on the CD in the `code\JavaEE\Chapter11\LoginApp\src\com\kogent` folder):

Listing 11.27: Showing the Code of the WelcomeServlet Servlet

```
package com.kogent;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class WelcomeServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,HttpServletResponse response) {
        HttpSession session = request.getSession();
        try {
                PrintWriter writer = response.getWriter();
                if (session == null || session.isNew()) {
                        writer.println("You are not logged in");
                }
                else {
                        response.setContentType("text/html");
                writer.println("<html><body style=\"font-
                family:verdana;font-size:10pt\">");
                writer.println("<h1>Login Application using Session Tracking</h1>");
                writer.println("<p>Thank you, you are already logged in");
                writer.println("<p>Here is the data in your session");
```

```
                        Enumeration names = session.getAttributeNames();
                        while (names.hasMoreElements()) {
                                String name = (String) names.nextElement();
                                Object value = session.getAttribute(name);
                    writer.println("<p>name=" + name + " value=" + value);
                        }
                    }
                    session.invalidate();
    writer.println("<p><a href=\"index.html\">Logout" + "</a> and return to login page");
                    writer.println("</body></html>");
                    writer.close();
                }
                catch (Exception e) {
                    e.printStackTrace();
                }
        }
}
```

When the request is forwarded to the WelcomeServlet servlet, the servlet retrieves the session and checks whether the session is new or not. If the session is new, the logout message is displayed to the user. However, if the same session continues, the session details of the logged in user are displayed on the browser.

The getAttribute() method is used to retrieve the values of username and password attributes. The getAttributeNames() method is used to retrieve the values of the attributes that are set during the user session. In the following code snippet, the while loop is used to retrieve the attribute name and its value, which is displayed on the browser:

```
Enumeration names = session.getAttributeNames();
while (names.hasMoreElements()) {
    String name = (String) names.nextElement();
    Object value = session.getAttribute(name);
    writer.println("<p>name=" + name + " value=" + value);
}
```

After displaying the session details, the session is explicitly terminated using the invalidate() method. The user can end up the session by clicking the logout link.

The LoginApp Web application created a session for the user who had logged in with the username and password. After logging into the application, the user browses to get the session details. During the session, username and password have been set as the attributes in the user session, and the values of these attributes are retrieved by the WelcomeServlet servlet and are displayed on the browser.

Now, compile the two servlets from the src\com\kogent directory by using the following command:

```
javac -d C:\JavaEE\Chapter11\LoginApp\WEB-INF\classes *.java
```

The execution of the preceding command creates the package directory under the WEB-INF\classes folder. Prior to packaging, deploying, and running the LoginApp Web application, these servlets need to be configured in web.xml.

Configuring the Login Application

To configure the LoginApp application, the configuration file used is web.xml, which maps the URL path, forwarded by index.html to the LoginServlet servlet and also defines the url pattern for the WelcomeServlet servlet. Listing 11.28 shows the code of the web.xml file (you can find this file on the CD in the code\Java EE\Chapter11\LoginApp\WEB-INF folder):

Listing 11.28: Configuring the Servlets and HTML Page of the LoginApp Application

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" 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_2_5.xsd">
<servlet>
  <servlet-name>LoginServlet</servlet-name>
  <servlet-class>com.kogent.LoginServlet</servlet-class>
</servlet>
```

```
   <servlet-mapping>
      <servlet-name>LoginServlet</servlet-name>
      <url-pattern>/LoginServlet</url-pattern>
   </servlet-mapping>
   <servlet>
      <servlet-name>WelcomeServlet</servlet-name>
      <servlet-class>com.kogent.WelcomeServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>WelcomeServlet</servlet-name>
      <url-pattern>/WelcomeServlet</url-pattern>
   </servlet-mapping>
   <session-config>
      <session-timeout>30</session-timeout>
   </session-config>
   <welcome-file-list>
      <welcome-file>index.html</welcome-file>
   </welcome-file-list>
   </web-app>
```

In Listing 11.28, web.xml configures the `LoginServlet` and `WelcomeServlet` servlets and also provides the url pattern for these servlets. The full package structure has been provided for each servlet class, as shown in Figure 11.36. Moreover, the session timeout has also been set to 30 minutes and the welcome file which will be displayed when the `LoginApp` Web application runs, is `index.html`.

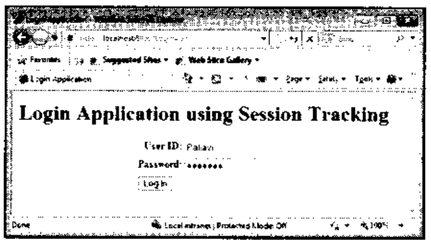
Now, before running the `LoginApp` Web application, it is packaged in the WAR file (`LoginApp.war`) by using the following command:

```
   jar -cvf LoginApp.war .
```

The preceding command creates the `LoginApp.war` file containing all the files of the `LoginApp` directory.

Running the Login Application

After packaging the `LoginApp` Web application, start the Glassfish server and deploy `LoginApp.war`. Now, browse `http://localhost:8080/LoginApp` to see the output of the LoginApp application. Figure 11.37 displays the output of index.html, which serve as the Login page:

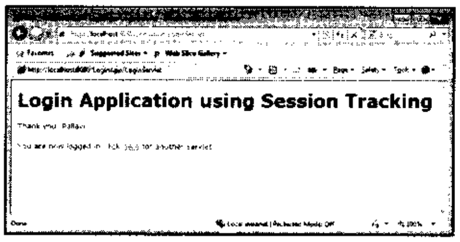


Login Application using Session Tracking

User ID: Pallavi
Password: •••••••
Log In

Figure 11.37: Displaying the Output of index.html

In Figure 11.37, enter the User ID and Password and click the Log In button. In our case, we have entered Pallavi in the User ID text box and pallavi as password in the Password text box. Then, click the Log In button to forward the request to the `LoginServlet servlet`. The `LoginServlet` servlet creates a new session for the pallavi user and displays a welcome message, as shown in Figure 11.38:



Login Application using Session Tracking

Thank you, Pallavi

You are now logged in. Click here for another servlet

Figure 11.38: Displaying the Output of the LoginServlet Servlet

The `LoginServlet` servlet creates a session for the logged in user. To view the session details and value of the attributes set in the session, the user can click the `here` link shown in Figure 11.38. After clicking the link, the request is sent to `WelcomeServlet` and the session details are displayed, as shown in Figure 11.39:

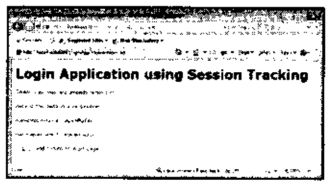

Login Application using Session Tracking

Figure 11.39: Displaying the Output of the WelcomeServlet Servlet

By clicking the Logout link, shown in Figure 11.39, the user can explicitly end the session and return to index.html. Therefore, the `LoginApp` application maintains a session for the user who logs in with the username and password. Based on the username and password, the username and password attributes are set in the session and till the user is in the same session, the session details can be viewed. After logging out, the session is explicitly terminated and the user is no longer in the session and a new session is created for the new user. This ends up the discussion about session tracking. With this, we come to the end of the chapter. Let's now recap the main points of the chapter in a short summary.

Summary

This chapter has discussed the Java Servlet API, version 2.5. It has first explored the features of Servlet 2.5 after which the general features of a Java Servlet have been discussed. The chapter has then explained the classes and packages of the Servlet API that are used to develop Web applications. You have also learned about the life cycle of a servlet and configuring a servlet in the web.xml file. Apart from this, you have learned to create a sample servlet by mapping it in the web.xml file. The chapter has also listed the noteworthy interfaces of the Servlet 2.5 API. You have also learned about request delegation, request scope and servlet collaboration. At the end of the chapter, you have learned about session tracking; the Java Servlet API used for implementing session tracking; and also learned about how to implement session tracking by creating the Login application.

Quick Revise

Q1. **The two arguments passed to the `forward()` method of the `RequestDispatcher` class are objects of the ………………………………………………………… classes.**

A) HttpServletRequest and HttpServletResponse

B) HttpServletResponse and PrintWriter

C) ServletContext and ServletConfig

D) HttpServletRequest and ServletContext

Ans: The correct option is A.

Q2. **To get an object of the `PrintWriter` class, we use the `getWriter()` method of the ………………………… class.**

A) HttpServletRequest B) HttpServletResponse

C) SessionContext D) HttpSession

Ans: The correct option is B.

Q3. **A life cycle method of a servlet is …………………………………… .**

A) init() B) service()

C) `destroy()` D) Above All

Ans: The correct option is D.

Q4. **We can get an object of the `RequestDispatcher` class by using the `getRequestDispatcher()` method on the …………………………… class.**

A) ServletRequest B) ServletContext

C) Both A and B D) None

Ans: The correct option is C.

Q5. Initialization parameters can be fetched by using the .... method.

A) getAttribute() B) getParameter()

C) getInitParameter() D) getServletContext()

Ans: The correct option is C.

Q6. The number of `ServletContext` objects present for an application is ....

A) 2 B) 1

C) Not fixed D) Each for a Servlet

Ans: The correct option is B.

Q7. `HttpServlet` extends ....

A) The javax.servlet.GenericServlet class

B) The javax.servlet.http.GenericServlet class

C) The javax.servlet.http.HttpServletRequest interface

D) The javax.servlet.ServletInputStream class

Ans: The correct option is A.

Q8. The method used to fetch the value of an object in the request scope is ....

A) getAttribute() B) getParameter()

C) getInitParameter() D) None

Ans: The correct option is A.

Q9. If cookies are disabled on client-side, the alternate session mechanism that can be used is ....

A) Either Cookies or URL rewriting B) URL rewriting

C) Cookies and URL rewriting D) None

Ans: B

Q10. The interface that defines the `getSession()` method is ....

A) HttpServletRequest B) ServletRequest

C) ServletResponse D) HttpServletResponse

Ans: A

Q11. The .... method of `HttpServletRequest` returns null if a session does not exist.

A) getSession() B) getSession(true)

C) getSession(false) D) getNewSession()

Ans: C

Q12. The .... method on the session object is used to remove a set attribute.

A) removeAllValues() B) removeAttribute("attributeName")

C) removeAttributes() D) removeAllAttributes()

Ans: B

Q13. Which statement is false:

A) URL rewriting can be used to track a session

B) SSL has a built in mechanism to obtain the data to define a session

C) The name of session tracking cookies must be `JSESSIONID`

D) There is no restriction for name of the cookie tracking the session

Ans: D

Q14. What is a servlet?

Ans: A servlet is a simple Java class working on the request-response model. Various interfaces and classes to handle common HTTP-specific services are defined in the Java Servlet API. Each servlet implements the Servlet interface by providing implementation of the life cycle methods, init(), service(), and destroy().

Q15. Differentiate between the ServletContext and ServletConfig objects.

Ans: A `ServletContext` object is used to communicate with a Servlet container while `ServletConfig`, which is a Servlet configuration object, is passed to the servlet by a container when the servlet is initialized. A `ServletContext` object is contained within a `ServletConfig` object.

Q16. What is the difference between the `getRequestDispatcher()` methods of the `ServletRequest` and `ServletContext interfaces`?

Ans: Both `getRequestDispatcher()` methods take a String parameter, which is a path to the location where a user's request would be forwarded. The `getRequestDispatcher()` method of the ServletRequest interface can accept both types of paths, i.e. the relative path from the requesting servlet and the path relative to the context root. However, the `getRequestDispatcher()` method of the `ServletContext` interface cannot accept the relative path.

Q17. Define the `init()` method of a servlet.

Ans: The `init()` method of the HttpServlet class is called before a servlet handles the first request and is used to initialize the servlet. This `init()` method also saves the `ServletConfig` object by using the super.`init()` method and stores the initialization details of a servlet. This method is called once only.

Q18. How is the `GET` method different from the `Post` method?

Ans: In the case of the GET method, all data submitted with an HTML form is attached with a URL. This method is faster and easier to use than the POST method but not as secure, and the upper limit of the URL length limits the amount of data transferred. The Post method, on the other hand, puts name/value combinations inside the HTTP request body and is therefore more secure. In addition, there is no limit for the amount of data that can be sent in this method.

Q19. What is a session?

Ans: A session can be defined as a collection of HTTP requests, over a period of time, between a client and a Web server. When a session is created the lifetime of the session is also set. The session object is destroyed on the expiration of session and all the resources are returned back to the servlet engine.

Q20. List the session tracking techniques.

Ans: There are basically the following four session tracking techniques:

- Cookies
- Hidden Form Fields
- URL Rewriting
- Secure Socket Layer (SSL) sessions

Q21. How cookies are used to track a session?

Ans: Using cookies is the simplest and easiest way to track a session. A unique session id (stored in the form of a cookie) is sent by the server to the client as a part of the response and the same session id saved with the client is sent to the server as a part of the request which helps the server to recognize the unique client session.

Q22. List the methods of the HttpSession interface which help a servlet to manage session life-cycle.

Ans: The HttpSession interface provides the following methods to manage session life-cycle:

- Invalidate()
- setMaxInactiveInterval(int interval)
- isNew()
- getCreationTime()
- getLastAccessedTime()

Q23. Cookie is a class of the .... package.

Ans: javax.servlet.http