**Overview of Web Applications**

**A** web application is a dynamic extension of a web or application server. There are two types of web applications:

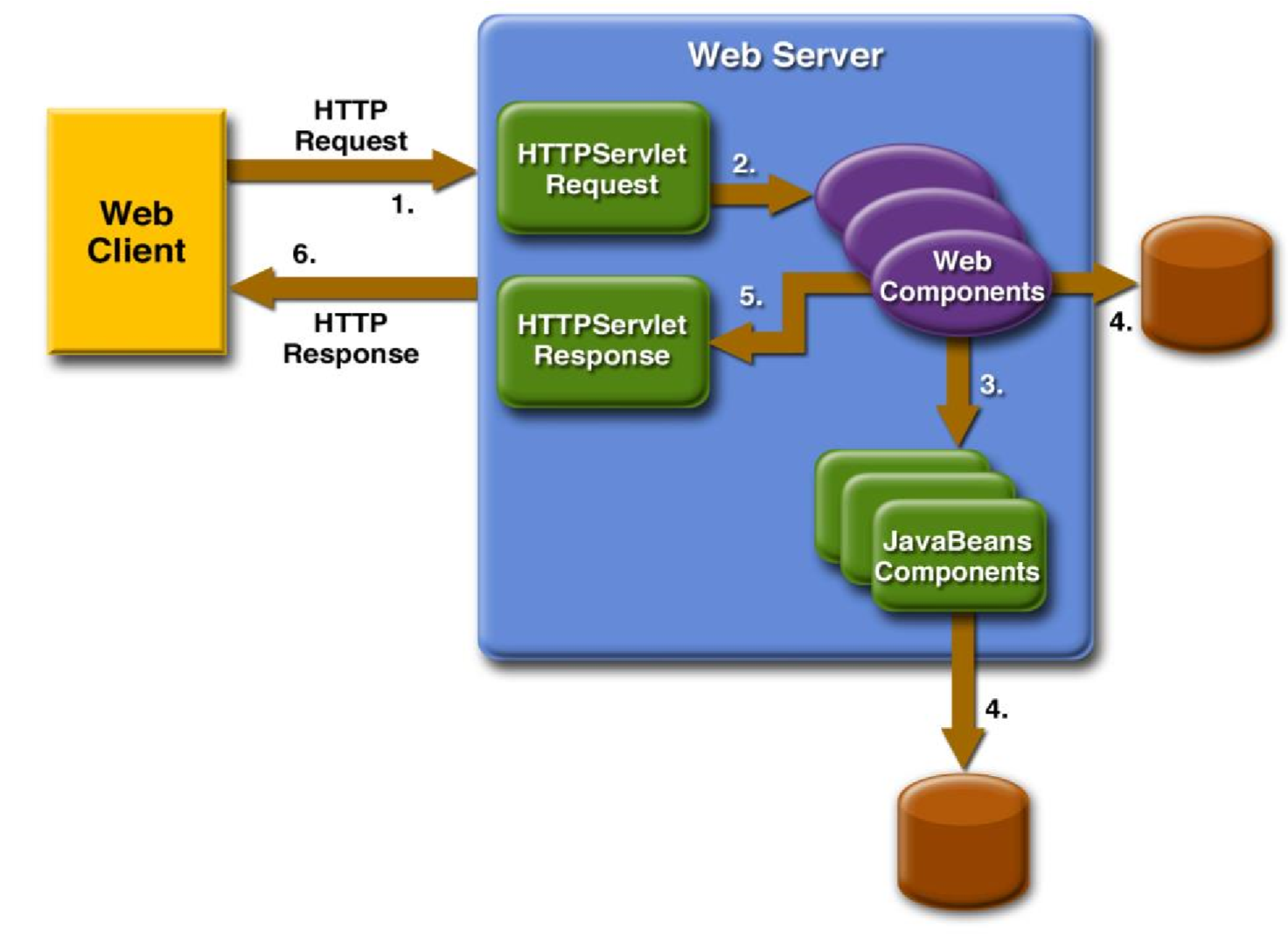
* ***Presentation-oriented*:** A presentation-oriented web application generates interactive web pages containing various types of markup language (HTML, XML, and so on) and dynamic content in response to requests.

* ***Service-oriented*:** A service-oriented web application implements the endpoint of a web service.

Presentation-oriented applications are often clients of service-oriented web applications.

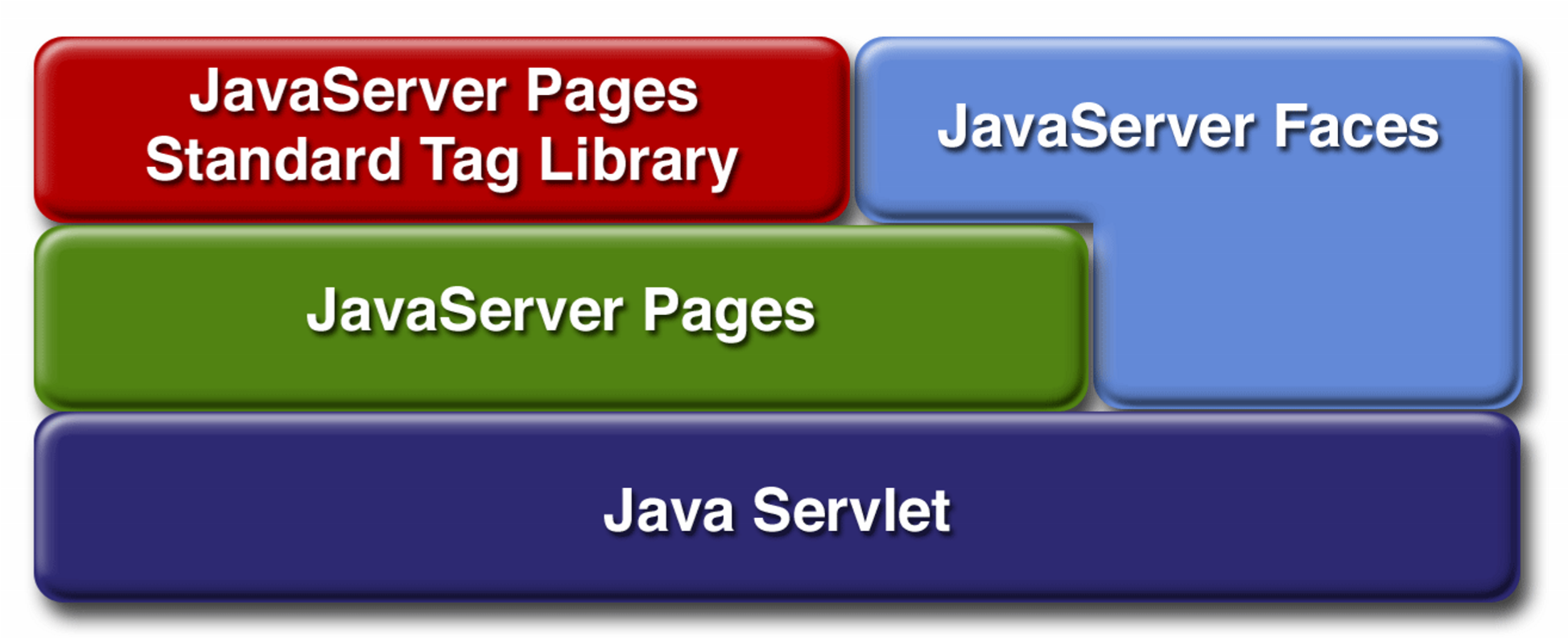
In the Java 2 platform, *web components* provide the dynamic extension capabilities for a web server. Web components are either Java servlets, JSP pages, or web service endpoints. The interaction between a web client and a web application is illustrated in Figure below.

The client sends an HTTP request to the web server. A web server that implements Java Servlet and JavaServer Pages technology converts the request into an HTTPServletRequest object. This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic content. The web component can then generate an HTTPServletResponse or it can pass the request to another web component. Eventually a web component generates an HTTPServletResponse object. The web server converts this object to an HTTP response and returns it to the client.



*Servlets* are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content.Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited for service-oriented applications (web service endpoints are implemented as servlets) and the control functions of a presentation- oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, Scalable Vector Graphics (SVG), Wireless Markup Language (WML), and XML.

Since the introduction of Java Servlet and JSP technology, additional Java technologies and frameworks for building interactive web applications have been developed. These technologies and their relationships are illustrated in Figure below.



Java Servlet technology is the foundation of all the web application technologies. Each technology adds a level of abstraction that makes web application prototyping and development faster and the web applications themselves more maintainable, scalable, and robust.Web components are supported by the services of a runtime platform called a *web container*. A web container provides services such as request dispatching,security, concurrency, and life-cycle management. It also gives web components access to APIs such as naming, transactions, and email.

Certain aspects of web application behavior can be configured when the application is installed, or *deployed*, to the web container. The configuration information is maintained in a text file in XML format called a *web application deployment descriptor* (DD). A DD must conform to the schema described in the Java Servlet Specification.

Most web applications use the HTTP protocol, and support for HTTP is a major aspect of web components

# Web Application Life Cycle

A web application consists of web components, static resource files such as images, and helper classes and libraries. The web container provides many supporting services that enhance the capabilities of web components and make them easier to develop. However, because a web application must take these services into account, the process for creating and running a web application is different from that of traditional stand-alone Java classes. The process for creating, deploying, and executing a web application can be summarized as follows:

1. Develop the web component code.
2. Develop the web application deployment descriptor.
3. Compile the web application components and helper classes referenced by the components.
4. Optionally package the application into a deployable unit.
5. Deploy the application into a web container.
6. Access a URL that references the web application.

# Web Modules

In the J2EE architecture, web components and static web content files such as images are called *web resources*. A *web module* is the smallest deployable and usable unit of web resources. A J2EE web module corresponds to a *web application* as defined in the Java Servlet specification.

In addition to web components and web resources, a web module can contain other files:

* Server-side utility classes (database beans, shopping carts, and so on).

Often these classes conform to the JavaBeans component architecture.

* Client-side classes (applets and utility classes).

A web module has a specific structure. The top-level directory of a web module is the *document root* of the application. The document root is where JSP pages, *client-side* classes and archives, and static web resources, such as images, are stored.

The document root contains a subdirectory named /WEB-INF/, which contains the following files and directories:

* web.xml: The web application deployment descriptor
* classes: A directory that contains *server-side classes*: servlets, utility classes, and JavaBeans components
* lib: A directory that contains JAR archives of libraries called by server side Classes

A web module can be deployed as an unpacked file structure or can be packaged in a JAR file known as a web archive (WAR) file. Because the contents and use of WAR files differ from those of JAR files, WAR file names use a .war extension. The web module just described is portable; you can deploy it into any web container that conforms to the Java Servlet Specification.

# Packaging Web Modules

A web module must be packaged into a WAR in certain deployment scenarios and whenever you want to distribute the web module. You package a web module into a WAR using the

Application Server deploytool utility, by executing the jar command in a directory laid out in the format of a web module, or by using the asant utility

# 2. Introduction to Servlets

AS soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java servlet technology was created as a portable way to provide dynamic, user-oriented content.

**2.1 What Is a Servlet?**

A servlet is a Java programming language class that is used to extend the capabilities of servers that host applications access via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods. When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API.

The HttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.

## HTTP Overview

MOST Web clients use the HTTP protocol to communicate with a J2EE server. HTTP defines the requests that a client can send to a server and responses that the server can send in reply. Each request contains a URL, which is a string that identifies a Web component or a static object such as an HTML page or mage file.

A J2EE server converts an HTTP request to an HTTP request object and delivers it to the Web component identified by the request URL. The Web component fills in an HTTP response object, which the server converts to an HTTP response and sends to the client.

## HTTP Requests

An HTTP request consists of a request method, a request URL, header fields,and a body. HTTP 1.1 defines the following request methods:

* GET: Retrieves the resource identified by the request URL
* HEAD: Returns the headers identified by the request URL
* POST: Sends data of unlimited length to the Web server
* PUT: Stores a resource under the request URL
* DELETE: Removes the resource identified by the request URL
* OPTIONS: Returns the HTTP methods the server supports
* TRACE: Returns the header fields sent with the TRACE request

HTTP 1.0 includes only the GET, HEAD, and POST methods. Although J2EE servers are required to support only HTTP 1.0, in practice many servers, including the Application Server, support HTTP 1.1.

## HTTP Responses

An HTTP response contains a result code, header fields, and a body.

The HTTP protocol expects the result code and all header fields to be returned before any body content.

Some commonly used status codes include:

* 404: Indicates that the requested resource is not available
* 401: Indicates that the request requires HTTP authentication
* 500: Indicates that an error occurred inside the HTTP server that prevented it from fulfilling the request
* 503: Indicates that the HTTP server is temporarily overloaded and unable to handle the request

## Servlet Container

A servlet container is nothing but a compiled, executable program. The main function of the container is to load, initialize and execute servlets. The servlet container is the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed by Sun under the Java Community Process. A container handles large number of requests as it can hold many active servlets, listeners etc. It is interesting to note here that the container and the objects in a container are multithreaded. So each object must be thread safe in a container as the multiple requests are being handled by the container due to the entrance of more than one thread to an object at a time.

Note : A Servlet container may run stand alone i.e. without a web server or even on another host.

We can categorize the servlet containers as:

1. A simple servlet container is not fully functional and therefore it can only run very simple servlets and does the following :
   * Wait for HTTP request.
   * Construct a ServletRequest object and a ServletResponse object.
   * If the request is for a static resource, invoke the process method of the

StaticResourceProcessor instance, passing the ServletRequest and ServletResponse objects.

* + If the request is for a servlet, load the servlet class and invoke its service method, passing the ServletRequest and ServletResponse objects. Note that in this servlet container, the servlet class is loaded every time the servlet is requested.

1. A fully functional servlet container additionally does the following for each HTTP request for a servlet:
   * When the servlet is called for the first time, load the servlet class and call its init method (once only).
   * For each request, construct an instance of javax.servlet.ServletRequest and an instance of javax.servlet.ServletResponse.
   * Invoke the servlet's service method, passing the ServletRequest and ServletResponse objects.
   * When the servlet class is shut down, call the servlet's destroy method and unload the servlet class.

Now lets see what a servlet container does for each HTTP request for a servlet, in general :

* + The servlet container loads the servlet class and calls the init method of the servlet as soon as the servlet is called for the first time.
  + Then this container makes an instance of javax.servlet.ServletRequest and javax.servlet.ServletResponse for each request.
  + Then it passes the ServletRequest and ServletResponse objects by invoking the servlet's service method.
  + Finally, it calls the destroy method and unload the servlet class when the servlet class is to be shut down.

## Servlet API

|  |  |
| --- | --- |
| **Packages** |  |
| [**javax.servlet**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/package-summary.html) | The javax.servlet package contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container. |
| [**javax.servlet.http**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/package-summary.html) | The javax.servlet.http package contains a number of classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment provided for an instance of such a class by a conforming servlet container. |

## Package javax.servlet

The javax.servlet package contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.

**See:**

[**Description**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/package-summary.html#package_description)

|  |  |
| --- | --- |
| **Interface Summary** | |
| [**Filter**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/Filter.html) | A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both. |
| [**FilterChain**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/FilterChain.html) | A FilterChain is an object provided by the servlet container to the developer giving a view into the invocation chain of a filtered request for a resource. |
| [**FilterConfig**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/FilterConfig.html) | A filter configuration object used by a servlet container to pass information to a filter during initialization. |
| [**RequestDispatcher**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/RequestDispatcher.html) | Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. |
| [**Servlet**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/Servlet.html) | Defines methods that all servlets must implement. |
| [**ServletConfig**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletConfig.html) | A servlet configuration object used by a servlet container to pass information to a servlet during initialization. |
| [**ServletContext**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletContext.html) | Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file. |
| [**ServletContextAttributeListener**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletContextAttributeListener.html) | Implementations of this interface receive notifications of changes to the attribute list on the servlet context of a web application. |
| [**ServletContextListener**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletContextListener.html) | Implementations of this interface receive notifications about changes to the servlet context of the web application they are part of. |
| [**ServletRequest**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequest.html) | Defines an object to provide client request information to a servlet. |
| [**ServletRequestAttributeListener**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequestAttributeListener.html) | A ServletRequestAttributeListener can be implemented by the developer interested in being notified of request attribute changes. |
| [**ServletRequestListener**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequestListener.html) | A ServletRequestListener can be implemented by the developer interested in being notified of requests coming in and out of scope in a web component. |
| [**ServletResponse**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletResponse.html) | Defines an object to assist a servlet in sending a response to the client. |
| [**SingleThreadModel**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/SingleThreadModel.html) | **Deprecated.** *As of Java Servlet API 2.4, with no direct replacement.* |

|  |  |
| --- | --- |
| **Class Summary** | |
| [**GenericServlet**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/GenericServlet.html) | Defines a generic, protocol-independent servlet. |
| [**ServletContextAttributeEvent**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletContextAttributeEvent.html) | This is the event class for notifications about changes to the attributes of the servlet context of a web application. |
| [**ServletContextEvent**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletContextEvent.html) | This is the event class for notifications about changes to the servlet context of a web application. |
| [**ServletInputStream**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletInputStream.html) | Provides an input stream for reading binary data from a client request, including an efficient readLine method for reading data one line at a time. |
| [**ServletOutputStream**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletOutputStream.html) | Provides an output stream for sending binary data to the client. |
| [**ServletRequestAttributeEvent**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequestAttributeEvent.html) | This is the event class for notifications of changes to the attributes of the servlet request in an application. |
| [**ServletRequestEvent**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequestEvent.html) | Events of this kind indicate lifecycle events for a ServletRequest. |
| [**ServletRequestWrapper**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequestWrapper.html) | Provides a convenient implementation of the ServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet. |
| [**ServletResponseWrapper**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletResponseWrapper.html) | Provides a convenient implementation of the ServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet. |

|  |  |
| --- | --- |
| **Exception Summary** | |
| [**ServletException**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletException.html) | Defines a general exception a servlet can throw when it encounters difficulty. |
| [**UnavailableException**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/UnavailableException.html) | Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable. |

## Package javax.servlet.http

The javax.servlet.http package contains a number of classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment provided for an instance of such a class by a conforming servlet container.

|  |  |
| --- | --- |
| **Interface Summary** | |
| [**HttpServletRequest**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletRequest.html) | Extends the [ServletRequest](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletRequest.html) interface to provide request information for HTTP servlets. |
| [**HttpServletResponse**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletResponse.html) | Extends the [ServletResponse](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletResponse.html) interface to provide HTTPspecific functionality in sending a response. |
| [**HttpSession**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSession.html) | Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user. |
| [**HttpSessionActivationListener**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionActivationListener.html) | Objects that are bound to a session may listen to container events notifying them that sessions will be passivated and that session will be activated. |
| [**HttpSessionAttributeListener**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionAttributeListener.html) | This listener interface can be implemented in order to get notifications of changes to the attribute lists of sessions within this web application. |
| [**HttpSessionBindingListener**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionBindingListener.html) | Causes an object to be notified when it is bound to or unbound from a session. |
| [**HttpSessionContext**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionContext.html) | **Deprecated.** *As of Java(tm) Servlet API 2.1 for security reasons, with no replacement.* |
| [**HttpSessionListener**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionListener.html) | Implementations of this interface are notified of changes to the list of active sessions in a web application. |

|  |  |
| --- | --- |
| **Class Summary** |  |
| [**Cookie**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/Cookie.html) | Creates a cookie, a small amount of information sent by a |
|  | servlet to a Web browser, saved by the browser, and later sent back to the server. |
| [**HttpServlet**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServlet.html) | Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. |
| [**HttpServletRequestWrapper**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletRequestWrapper.html) | Provides a convenient implementation of the  HttpServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet. |
| [**HttpServletResponseWrapper**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletResponseWrapper.html) | Provides a convenient implementation of the  HttpServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet. |
| [**HttpSessionBindingEvent**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionBindingEvent.html) | Events of this type are either sent to an object that implements [HttpSessionBindingListener](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionBindingListener.html) when it is bound or unbound from a session, or to a [HttpSessionAttributeListener](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionAttributeListener.html) that has been configured in the deployment descriptor when any attribute is bound, unbound or replaced in a session. |
| [**HttpSessionEvent**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpSessionEvent.html) | This is the class representing event notifications for changes to sessions within a web application. |
| [**HttpUtils**](http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpUtils.html) | **Deprecated.** *As of Java(tm) Servlet API 2.3.* |

**GenericServlet class**

**GenericServlet** class implements **Servlet**, **ServletConfig** and **Serializable**interfaces. It provides the implementation of all the methods of these interfaces except the service method.

GenericServlet class can handle any type of request so it is protocol-independent.

You may create a generic servlet by inheriting the GenericServlet class and providing the implementation of the service method.

### Methods of GenericServlet class

There are many methods in GenericServlet class. They are as follows:

**public void init(ServletConfig config)** is used to initialize the servlet.

**public abstract void service(ServletRequest request, ServletResponse response)** provides service for the incoming request. It is invoked at each time when user requests for a servlet.

**public void destroy()** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.

**public ServletConfig getServletConfig()** returns the object of ServletConfig.

**public String getServletInfo()** returns information about servlet such as writer, copyright, version etc.

**public void init()** it is a convenient method for the servlet programmers, now there is no need to call super.init(config)

**public ServletContext getServletContext()** returns the object of ServletContext.

**public String getInitParameter(String name)** returns the parameter value for the given parameter name.

**public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.

**public String getServletName()** returns the name of the servlet object.

**public void log(String msg)** writes the given message in the servlet log file.

**public void log(String msg,Throwable t)** writes the explanatory message in the servlet log file and a stack trace.

**import** java.io.\*;

**import** javax.servlet.\*;

**public** **class** First **extends** GenericServlet{

**public** **void** service(ServletRequest req,ServletResponse res)

**throws** IOException,ServletException{

res.setContentType("text/html");

PrintWriter out=res.getWriter();

out.print("<html><body>");

out.print("<b>hello generic servlet</b>");

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

}

}

**HttpServlet Class**

We know that a Servlet is a simple Java class which must implement javax.servlet.Servlet interface. GenericServlet class provides a default implementation of this interface so that we don't have to implement every method of it. HttpServlet class extends GenericServlet to provide an HTTP protocol specific implementation of Servlet interface. If your Servlet has to work with HTTP protocol you should simply extend HttpServlet class and override the methods you need.

Examining Methods of HttpServlet class

Let's examine the methods which this class provides one by one :

* init()Called only once during the initialization of the Servlet.
* destroy()Called only once when Servlet instance is about to be destroyed.
* service()Do not override this method.
* doGet(), doPost(), doPut(), doDelete(), doOptions, doTrace()These methods are called according to the type of HTTP request received. Override them to generate your own response.
* log()Writes messages to the Servlet's log files.
* getLastModified()Override this method to return your Servlet's last modified date.
* getServletInfo()Override this method to provide a String of general info about your Servlet such author, version, copyright etc.
* getServletName()Override this method to return name of the Servlet.
* getInitParameter(), getInitParameterNames()First one returns value of given initialization parameter, second one returns an Enumeration object containing names of all initialization parameters provided.
* getServletConfig()Returns a reference to ServletConfig object. More on it later in this article.
* getServletContext()Returns reference to ServletContext object. More on it later in this article.

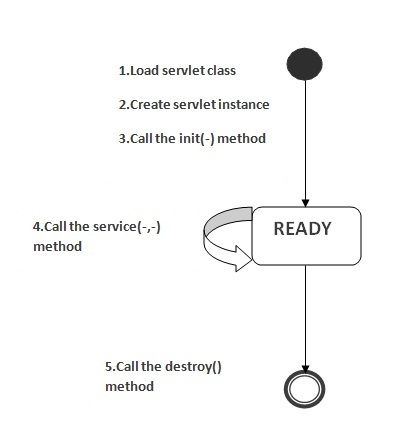
You have now seen all of the methods provided to you by the HttpServlet class. You can override all of them except service(), getServletConfig() and getServletContext() methods to provide your own implementation of them. The three methods which stated above should not be overriden as their default implementation is more than enough to suffice.

**Servlet Life Cycle**

The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the web container
   1. Loads the servlet class.
   2. Creates an instance of the servlet class.
   3. Initializes the servlet instance by calling the init() method.
2. Invokes the service() method, passing request and response objects.

3.If the container needs to remove the servlet, it finalizes the servlet by calling the servlet’s destroy() method



A servlet is basically a small Java program that runs within a Web server. It can receive requests from clients and return responses.

The life cycle of a servlet is controlled by the container in which the servlet has been deployed. More specifically, the behavior of a servlet is described in javax.servlet.Servlet interface, in which the following methods are defined:

**The init Method**

Most of the time, your servlets deal only with per-request data, and doGet or doPost are the only life-cycle methods you need. Occasionally, however, you want to perform complex setup tasks when the servlet is first loaded, but not repeat those tasks for each request. The init method is designed for this case; it is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started. The init method performs two varieties of initializations: general initializations and initializations controlled by initialization parameters.

* public void init(ServletConfig config) throws ServletException

The init() method has a ServletConfig parameter. The servlet can read its initialization arguments through the ServletConfig object. How the initialization arguments are set is servlet engine dependent but they are usually defined in a configuration file.

**The service Method**

* public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException

This method is called to process a request. It can be called zero, one or many times until the servlet is unloaded.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc., as appropriate. A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified. A POST request results from an HTML form that specifically lists POST as the METHOD. Other HTTP requests are generated only by custom clients. Now, if you have a servlet that needs to handle both POST

and GET requests identically, you may be tempted to override service directly rather than implementing both doGet and doPost. This is not a good idea. Instead, just have doPost call doGet (or vice versa).

**The doGet, doPost, and doXxx Methods**

These methods contain the real meat of your servlet. Ninety-nine percent of the time, you only care about GET or POST requests, so you override doGet and/or doPost. However, if you want to, you can also override doDelete for DELETE requests, doPut for PUT, doOptions for OPTIONS, and doTrace for TRACE. Recall, however, that you have automatic support for OPTIONS and TRACE. Normally, you do not need to implement doHead in order to handle HEAD requests (HEAD requests stipulate that the server should return the normal HTTP headers, but no associated document). You don’t normally need to implement doHead because the system automatically calls doGet and uses the resultant status line and header settings to answer HEAD requests. However, it is occasionally useful to implement doHead so that you can generate responses to HEAD requests (i.e., requests from custom clients that want just the HTTP headers, not the actual document) more quickly—without building the actual document output.

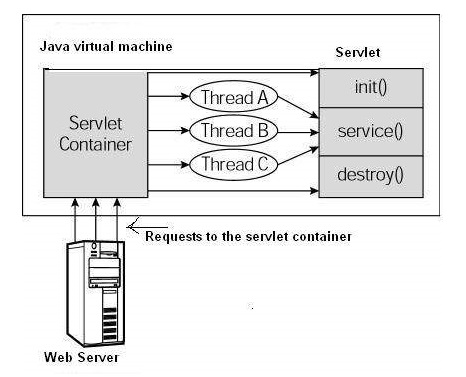
**The destroy Method**

* public void destroy()

This method is called once just before the servlet is unloaded and taken out of service.

The server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator or perhaps because the servlet is idle for a long time. Before it does, however, it calls the servlet’s destroy method. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities. Be aware, however, that it is possible for the Web server to crash. So, don’t count on destroy as the only mechanism for saving state to disk. If your servlet performs activities like counting hits or accumulating lists of cookie values that indicate special access, you should also proactively write the data to disk periodically.

The servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.



## Example Servlet Program

Program to print the system date at client browser import java.io.\*;

import java.util.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class MyServlet 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("Today's Date"); out.println("</Head>"); out.println("<BODY>"); Date today=new Date(); out.println("<p>" +today +"</p>"); out.println("</BODY>"); out.println("</HTML>");

out.flush();

}

}

**READING FORM DATA FROM SERVLETS**

**Reading Single Values: getParameter**

To read a request (form) parameter, you simply call the getParameter method of HttpServletRequest, supplying the case-sensitive parameter name as an argument. You supply the parameter name exactly as it appeared in the HTML source code, and you get the result exactly as the end user entered it; any necessary URL-decoding is done automatically. An empty String is returned if the parameter exists but has no value (i.e., the user left the corresponding textfield empty when submitting the form), and null is returned if there was no such parameter. Parameter names are case sensitive so, for example, request getParameter ("Param1") and request. getParameter ("param1") are not interchangeable.

**Reading Multiple Values: getParameterValues**

If the same parameter name might appear in the form data more than once, you should call getParameterValues (which returns an array of strings) instead of getParameter (which returns a single string corresponding to the first occurrence of the parameter). The return value of getParameterValues is null for nonexistent parameter names and is a one element array when the parameter has only a single value. Now, if you are the author of the HTML form, it is usually best to ensure that each textfield, checkbox, or other user interface element has a unique name. That way, you can just stick with the simpler getParameter method and avoid getParameterValues altogether. Besides, multiselectable list boxes repeat the parameter name for each selected element in the list. So, you cannot always avoid multiple values.

**Looking Up Parameter Names: getParameterNames**

Use getParameterNames to get this list in the form of an Enumeration, each entry of which can be cast to a String and used in a getParameter or getParameterValues call. If there are no parameters in the current request, getParameterNames returns an empty Enumeration (not null). Note that Enumeration is an interface that merely guarantees that the actual class will have hasMoreElements and nextElement methods: there is no guarantee that any particular underlying data structure will be used. And, since some common data structures (hash tables, in particular) scramble the order of the elements, you should not count on getParameterNames returning the parameters in the order in which they appeared in the HTML form.

**Example: Write a Servlet that accepts name and age of student sent from an HTML document and displays them on screen. //Student.html file**

<html>

<head><title>Student Information</title> </head>

<form name=frm method=get action=http:\\localhost:8080\Servlet\Student.class> </form>

<body>

<table>

<tr><td>Student Name</td><td><input type=text name=txtName></td></tr>

<tr><td>Student Age</td><td><input type=text name=txtAge></td></tr>

</table>

<input type=submit name=submit> </body>

</html>

**//Student.java file**

import java.io.\*; import javax.servlet.\*;

import javax.servlet.http.\*;

public class Student extends HttpServlet

{

public void doGet(HttpServletRequest req, HttpServletResponse res)

throws ServletException,IOException

{

res.setContentType("text/html");

String name=(String)req.getParameter("txtName"); String age=(String)req.getParameter("txtAge"); PrintWriter out=res.getWriter();

out.println(“Name = ”+name); out.println(“Age= ”+age);

}

}//class

## Sharing Information

Web components, like most objects, usually work with other objects to accomplish their tasks. There are several ways they can do this. They can use private helper objects (for example, JavaBeans components), they can share objects that are attributes of a public scope, they can use a database, and they can invoke other Web resources.

### Using Scope Objects

Collaborating Web components share information via objects maintained as attributes of four scope objects. These attributes are accessed with the [get|set]Attribute methods of the class representing the scope.

|  |  |  |
| --- | --- | --- |
| Scope Objects | |  |
| Scope Object | Class | Accessible From |
| Web context | [javax.servlet.ServletConte xt](http://java.sun.com/j2ee/tutorial/api/javax/servlet/ServletContext.html) | Web components within a Web context |
| session | [javax.servlet.http.HttpSession](http://java.sun.com/j2ee/tutorial/api/javax/servlet/http/HttpSession.html) | Web components handling a request that belongs to the session |
| request | Subtype of [javax.servlet.ServletRequest](http://java.sun.com/j2ee/tutorial/api/javax/servlet/ServletRequest.html) | Web components handling the request |
| Page | [javax.servlet.jsp.PageContext](http://java.sun.com/j2ee/tutorial/api/javax/servlet/jsp/PageContext.html) | The JSP page that creates the object |

### Accessing Databases

Data that is shared between Web components and is persistent between invocations of a J2EE application is usually maintained by a database. Web components use the JDBC 2.0 API to access relational databases.

Consider the example below with a servlet generating client form to enter the data which will be inserted in to database through another servlet.

import java.io.\*; import javax.servlet.\*;

import javax.servlet.http.\*;

public class DeptForm extends HttpServlet

{

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

{

try{

res.setContentType("text/html"); PrintWriter out=res.getWriter();

out.println("<Html><Head><Title>Department Info</Title></Head>");

out.println("<Form name=frm method="+"POST"+"

action=DeptEntry.class>");

out.println("DepartmentNo: <input type=text name=txtNo><br>"); out.println("DepartmentName: <input type=text name=txtName><br>"); out.println("Location: <input type=text name=txtLoc><br>"); out.println("<input type=submit name=Submit>"); out.println("<input type=reset name=Reset>"); out.println("</Form></Html>");

}

catch (Exception e){

System.out.println(e.getMessage());

}

}

} import java.io.\*; import javax.servlet.\*; import javax.servlet.http.\*; import java.sql.\*;

public class DeptEntry extends HttpServlet

{

public void doPost(HttpServletRequest req,HttpServletResponse res)throws

IOException,ServletException

{ String a,b,c,d,e,f;

int i;

Connection con;

try

{

res.setContentType("text/html"); Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); con=DriverManager.getConnection("jdbc:odbc:emp");

String Query="insert into dept Values(?,?,?)";

Statement st=con.createStatement(); PreparedStatement ps; ps=con.prepareStatement(Query); a=(String)req.getParameter("txtNo"); b=(String)req.getParameter("txtName"); c=(String)req.getParameter("txtLoc");

ps.setString(1,a); ps.setString(2,b); ps.setString(3,c);

ps.executeUpdate();

PrintWriter out=res.getWriter();

ResultSet rs=st.executeQuery("select \* from dept"); ResultSetMetaData md=rs.getMetaData(); int num=md.getColumnCount(); out.println("<html><body><table border=1><tr>");

for(i=1;i<=num;i++)

{

out.print("<th>"+md.getColumnName(i)+"</th>");

}

out.println("</tr>"); while(rs.next()) { d=rs.getString(1); e=rs.getString(2); f=rs.getString(3); out.println("<tr><td>"); out.println(d); out.println("</td><td>"); out.println(e); out.println("</td><td>"); out.println(f); out.println("</td></tr>");

}

out.println("</table>"); con.commit();

out.println("<a href=DeptForm.class>BACK</a>");

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

}

catch (Exception ae){

System.out.println(ae.getMessage());

}

}

}

### Invoking Other Web Resources

Web components can invoke other web resources in two ways: indirectly and directly. A web component indirectly invokes another web resource when it embeds a URL that points to another web component in content returned to a client.

To invoke a resource available on the server that is running a web component, you must first obtain a RequestDispatcher object using the getRequestDispatcher("URL") method.

You can get a RequestDispatcher object from either a request or the web context; however, the two methods have slightly different behavior. The method takes the path to the requested resource as an argument. A request can take a relative path (that is, one that does not begin with a /), but the web context requires an absolute path. If the resource is not available or if the server has not implemented a RequestDispatcher object for that type of resource,getRequestDispatcher will return null. Your servlet should be prepared to deal with this condition.

### Including Other Resources in the Response

It is often useful to include another web resource (for example, banner content or copyright information) in the response returned from a web component. To include another resource,invoke the include method of a RequestDispatcher object: include(request, response);

If the resource is static, the include() method enables programmatic server-side includes. If the resource is a web component, the effect of the method is to send the request to the included web component, execute the web component, and then include the result of the execution in the response from the containing servlet. An included web component has access to the request object, but it is limited in what it can do with the response object: ■ It can write to the body of the response and commit a response.

■ It cannot set headers or call any method (for example, setCookie) that affects the headers of the response.

public class BannerServlet extends HttpServlet { public void doGet (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

output(request, response);

}

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

output(request, response);

}

private void output(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { PrintWriter out = response.getWriter(); out.println("<body bgcolor=\"#ffffff\">" + "<center>" + "<hr> <br> &nbsp;" + "<h1>" +

"<font size=\"+3\" color=\"#CC0066\">Duke’s </font>" +

<img src=\"" + request.getContextPath() +

"/duke.books.gif\">" +

"<font size=\"+3\" color=\"black\">Bookstore</font>" +

"</h1>" + "</center>" + "<br> &nbsp; <hr> <br> ");

}

}

Any servlet can include the result from BannerServlet using the following code: RequestDispatcher dispatcher =

getServletContext().getRequestDispatcher("/banner"); if (dispatcher != null)

dispatcher.include(request, response);

}

### Transferring Control to Another Web Component

In some applications, you might want to have one web component do preliminary processing of a request and have another component generate the response. For example, you might want to partially process a request and then transfer to another component depending on the nature of the request.

To transfer control to another web component, you invoke the forward method of a RequestDispatcher. When a request is forwarded, the request URL is set to the path of the forwarded page.

public class Dispatcher extends HttpServlet { public void doGet(HttpServletRequest request, HttpServletResponse response) { RequestDispatcher dispatcher = request. getRequestDispatcher("/template.jsp");

if (dispatcher != null)

dispatcher.forward(request, response);

} public void doPost(HttpServletRequest request,

...

}

The forward method should be used to give another resource responsibility for replying to the user. If you have already accessed a ServletOutputStream or PrintWriter object within the servlet, you cannot use this method; doing so throws an IllegalStateException.

## Accessing the Web Context

The context in which web components execute is an object that implements the ServletContext interface. You retrieve the web context by using the getServletContext method. The web context provides methods for accessing

* Initialization parameters
* Resources associated with the web context
* Object-valued attributes
* Logging capabilities

The counter’s access methods are synchronized to prevent incompatible operations by servlets that are running concurrently. A filter retrieves the counter object by using the context’s getAttribute method. The incremented value of the counter is recorded in the log.

Web-based applications are responsible for maintaining such state, called a session, because HTTP is stateless.

### 2.3 Init() method of a Servlet

After the Web container loads and instantiates the servlet class and before it delivers requests from clients, the Web container initializes the servlet. You can customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities by overriding the init method of the [Servlet](http://java.sun.com/j2ee/tutorial/api/javax/servlet/Servlet.html) interface. A servlet that cannot complete its initialization process should throw UnavailableException.

In the below program we are going to make a such a servlet which will count and displays the number of times it has been accessed and by reading the init parameter to know from where the counting will begin.

In this program we are going to make use of the init method of the Servlet interface which takes one argument of ServletConfig. Firstly declare a variable counter which will have the initial value of the counter. The init() method accepts an object which implements ServletConfig interface. It uses the method getInitParameter() method of the ServletConfig interface to the value of the init parameter initial which we have defined in the deployment descriptor file. You need to parse the String value which you will get from the getInitParameter() method to a Integer.

The code of the program is given below:

|  |
| --- |
| import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;    public class CounterInInit extends HttpServlet { int counter;  public void init(ServletConfig config) throws ServletException{ super.init(config);  String initValue = config.getInitParameter("initial"); try{  counter = Integer.parseInt(initValue);  }  catch(NumberFormatException e){ counter = 0;  }  }  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { response.setContentType("text/html"); PrintWriter pw = response.getWriter(); counter++;  pw.println("Since loading this servlet has been accessed" + counter + "times");  }  } |

web.xml file for this program:

<?xml version="1.0" encoding="ISO-8859-1"?> <!--<!DOCTYPE web-app

|  |
| --- |
| PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app\_2\_3.dtd"> -->    <web-app>  <init-param> <param-name>  initial  </param-name>  <param-value>  5  </param-value>  </init-param>  <servlet>  <servlet-name>Hello</servlet-name>  <servlet-class>CounterInInit</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>Hello</servlet-name>  <url-pattern>/CounterInInit</url-pattern>  </servlet-mapping>  </web-app> |

### 2.4 Writing Service Methods

The service provided by a servlet is implemented in the service method of a GenericServlet,in the doMethod methods (whereMethod can take the value Get, Delete, Options, Post, Put, orTrace) of an HttpServlet object, or in any other protocol-specific methods defined by a class that implements the Servlet interface.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response based on that information.

For HTTP servlets, the correct procedure for populating the response is to first retrieve an output stream from the response, then fill in the response headers, and finally write any body content to the output stream. Response headers must always be set before the response has been committed. Any attempt to set or add headers after the response has been committed will be ignored by the web container. The next two sections describe how to get information from requests and generate responses.

#### Getting Information from Requests

A request contains data passed between a client and the servlet. All requests implement the ServletRequest interface. This interface defines methods for accessing the following information:

■ Parameters, which are typically used to convey information between clients and servlets

■ Object-valued attributes, which are typically used to pass information between the servlet container and a servlet or between collaborating servlets

■ Information about the protocol used to communicate the request and about the client and server involved in the request

■ Information relevant to localization

For example, in CatalogServlet the identifier of the book that a customer wishes to purchase is included as a parameter to the request.

The following code fragment illustrates how to use the getParameter method to extract the identifier: String bookId = request.getParameter("Add");

if (bookId != null) {

Book book = bookDB.getBook(bookId);

You can also retrieve an input stream from the request and manually parse the data. To read character data, use the BufferedReader object returned by the request’s getReader method. To read binary data, use the ServletInputStream returned by getInputStream.

HTTP servlets are passed an HTTP request object,HttpServletRequest, which contains the request URL, HTTP headers, query string, and so on.

An HTTP request URL contains the following parts:

http://[host]:[port][request-path]?[query-string]

The request path is further composed of the following elements:

■ Context path: A concatenation of a forward slash (/) with the context root of the servlet’s web application.

■ Servlet path: The path section that corresponds to the component alias that activated this request. This path starts with a forward slash (/).

■ Path info: The part of the request path that is not part of the context path or the servlet path.

Query strings are composed of a set of parameters and values. Individual parameters are retrieved from a request by using the getParameter method. There are two ways to generate query strings:

* A query string can explicitly appear in a web page.

* A query string is appended to a URL when a form with a GET HTTP method is submitted.

#### Constructing Responses

A response contains data passed between a server and the client. All responses implement the ServletResponse interface. This interface defines methods that allow you to:

■ Retrieve an output stream to use to send data to the client. To send character data, use the PrintWriter returned by the response’s getWriter method. To send binary data in a MIME body response, use the ServletOutputStream returned by getOutputStream. To mix binaryand text data (as in a multipart response), use a ServletOutputStream and manage the character sections manually.

■ Indicate the content type (for example, text/html) being returned by the response with the setContentType(String) method. This method must be called before the response is committed. A registry of content type names is kept by the Internet AssignedNumbers Authority (IANA) at http://www.iana.org/assignments/media-types/.

■ Indicate whether to buffer output with the setBufferSize(int) method. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is actually sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another web resource. The method must be called before any content is written or before the response is committed.

■ Set localization information such as locale and character encoding.

HTTP response objects,HttpServletResponse, have fields representing HTTP headers such as the following:

■ Status codes, which are used to indicate the reason a request is not satisfied or that a request has been redirected.

■ Cookies, which are used to store application-specific information at the client. Sometimes cookies are used to maintain an identifier for tracking a user’s session .

### Maintaining Client State

Many applications require that a series of requests from a client be associated with one another.

Web-based applications are responsible for maintaining such state, called a session, because HTTP is stateless. To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

#### Accessing a Session

Sessions are represented by an HttpSession object. You access a session by calling the getSession method of a request object. This method returns the current session associated with this request, or, if the request does not have a session, it creates one.

#### Associating Objects with a Session

You can associate object-valued attributes with a session by name. Such attributes are accessible by any web component that belongs to the same web context and is handling a request that is part of the same session.

#### Session Management

Because there is no way for an HTTP client to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed. The timeout period can be accessed by using a session’s [get|set]MaxInactiveInterval methods.

#### SessionTracking

A web container can use several methods to associate a session with a user, all of which involve passing an identifier between the client and the server. The identifier can be maintained on the client as a cookie, or the web component can include the identifier in every URL that is returned to the client.

If your application uses session objects, you must ensure that session tracking is enabled by having the application rewrite URLs whenever the client turns off cookies. You do this by calling the response’s encodeURL(URL) method on all URLs returned by a servlet. This method includes the session ID in the URL only if cookies are disabled; otherwise, it returns the URL unchanged.

* **Here is some session related methods from the HttpServletRequest interface:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| public HttpSession getSession() | Will cause one session to be created. |
| public HttpSession getSession(boolean) | true = will cause one to be created; false = will return null (no session) |
| public String getRequestedSessionId() | Gets the ID assigned by the server to the session |
| public Boolean  isRequestedSessionIdValid() | Returns true if the request contains a valid session ID |
| public Boolean  isRequestedSessionIdFromCookie() | Returns true if the session ID was sent as part of a cookie |
| public Boolean  isRequestedSessionIdFromURL() | Returns true if the session ID was sent through URL rewriting |

* Default technique for session tracking is to use cookies.
* Cookies are sent in the header part of an HTTP message, so they must be set in the response prior to writing any data to the response.

* **HttpServletResponse interface:**

|  |  |  |
| --- | --- | --- |
|  | **Method** | **Description** |
|  | public String encodeURL(String) | Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. |
|  | public String  encodeRedirectURL(String) | Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. |
|  | **HttpSession interface:** |  |
|  | **Method (Get & Set types)** | **Description** |

public Object getAttribute(String name) Returns the object bound with the specified name in this session, or null if no object is bound under the name.

public Enumeration getAttributeNames() Returns an Enumeration of String objects containing the names of all the objects bound to this session.

|  |  |  |
| --- | --- | --- |
| public void setAttribute(String name, Object value) | Binds an object to this session, using the name specified. | |
| public void removeAttribute(String name) | Removes the object bound with the specified name from this session. | |
| **Method (lifecycle types)** | **Description** | |
| public long getCreationTime() | Returns the time when this session was created. | |
| public String getId() | Returns a string containing the unique identifier assigned to this session. | |
| public long getLastAccessedTime() | Returns the last time the client sent a request associated with this session. | |
| public boolean isNew() | | Returns true if the client does not yet know about the session or if the client chooses not to join the session. |
| public void setMaxInactiveInterval(int interval) | | Specifies the time, in seconds, between client requests before the servlet container will invalidate this session. |
| public int getMaxInactiveInterval() | | Returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses. |
| public void invalidate() | | Invalidates this session then unbinds any objects bound to it. |

**A simple Servlet Session program:**

public class LoginSES extends HttpServlet {

@Override

public void doPost(HttpServletRequest request,

HttpServletResponse response)

{

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

String password = request.getParameter("password");

// Get the session - if no session exists create one

HttpSession session = request.getSession(true);

// Set some attribute values to the session

// In this case user and password from the request and client session.setAttribute("username", username);

session.setAttribute("password", password);

try {

response.setContentType("text/html"); PrintWriter writer = response.getWriter(); writer.println("<html><body>"); writer.println("Thank you, " + username + ". You are now logged into the system");

// Encodes the specified URL by including the session ID in it,

// or, if encoding is not needed, returns the URL unchanged

String newURL = response.encodeURL("/ServletSession/GetSession");

// Return a <a> tag with the new url writer.println("Click <a href=\"" + newURL +

"\">here</a> for another servlet");

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

writer.close(); } catch (Exception e) { e.printStackTrace();

}

} }

The client input form in this example is:

Please enter your username and password

<form action="/ServletSession/Login" method="POST">

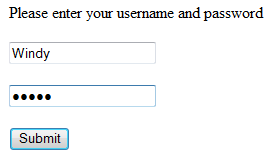
<p><input type="text" name="username" length="40">

<p><input type="password" name="password" length="40">

<p><input type="submit" value="Submit">

</form>

The form will look like:



* When the user press the submit button he get a response like: "Thank you, Windy. You are now logged into the system Click here for another servlet"
* If you click the here link you will access another servlet, GetSession.

Here is source code for the GetSession servlet:

public class GetSession extends HttpServlet {

@Override

public void doGet(HttpServletRequest request, HttpServletResponse response) {

// gets the session if it exists

HttpSession session = request.getSession(false);

try {

response.setContentType("text/html"); PrintWriter writer = response.getWriter(); writer.println("<html><body>");

// If you are not in a session - you are not logged in if (session == null) {

writer.println("<p>You are not logged in</p>");

} else {

writer.println("Thank you, you are already logged in"); writer.println("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 + "</p>");

}

}

// Write html for a new login

writer.println("<p><a href=\"/ServletSession/login.html\">Return" +

"</a> to login page</p>"); writer.println("</body></html>");

writer.close(); } catch (Exception e) { e.printStackTrace();

}

}

}

Java Servlets Cookies

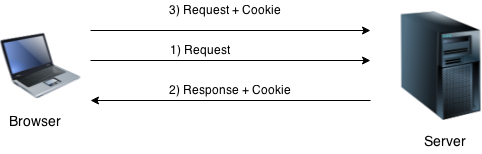
**What is Cookies?**

A **cookie** is a small piece of information that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.



Types of Cookie

There are 2 types of cookies in servlets.

1. Non-persistent cookie
2. Persistent cookie

**Non-persistent cookie**

It is **valid for single session** only. It is removed each time when user closes the browser.

### Persistent cookie

It is **valid for multiple session** . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

Cookie class

**javax.servlet.http.Cookie** class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

### Constructor of Cookie class Constructor

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| Cookie() | constructs a cookie. |
| Cookie(String name, String value) | constructs a cookie with a specified name and value. |

### Useful Methods of Cookie class

|  |  |
| --- | --- |
| **Method Description** | |
| public void setMaxAge(int expiry) | Sets the maximum age of the cookie in seconds. |
| public String getName() | Returns the name of the cookie. The name cannot be changed after creation. |
| public String getValue() | Returns the value of the cookie. |
| public void setName(String name) | changes the name of the cookie. |
| public void setValue(String value) | changes the value of the cookie. |

Other methods required for using Cookies

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:

1. **public void addCookie(Cookie ck):**method of HttpServletResponse interface is used to add cookie in response object.
2. **public Cookie[] getCookies():**method of HttpServletRequest interface is used to return all the cookies from the browser.

How to create Cookie?

Let's see the simple code to create cookie.

1. Cookie ck=**new** Cookie("user","ABCD");//creating cookie object

2. response.addCookie(ck);//adding cookie in the response

How to delete Cookie?

Let's see the simple code to delete cookie. It is mainly used to logout or signout the user.

1. Cookie ck=**new** Cookie("user","");//deleting value of cookie
2. ck.setMaxAge(0);//changing the maximum age to 0 seconds
3. response.addCookie(ck);//adding cookie in the response

How to get Cookies?

Let's see the simple code to get all the cookies.

Cookie ck[]=request.getCookies();

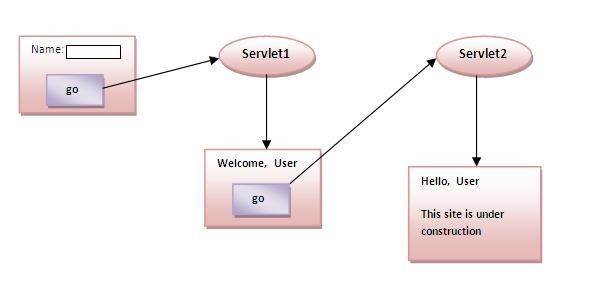
**for**(**int** i=0;i<ck.length;i++){

out.print("<br>"+ck[i].getName()+" "+ck[i].getValue()); //printing name and value of cookie

}

Simple example of Servlet Cookies

In this example, we are storing the name of the user in the cookie object and accessing it in another servlet. As we know well that session corresponds to the particular user. So if you access it from too many browsers with different values, you will get the different value.



### index.html

<form action="Servlet1" method="post">

Name:<input type="text" name="userName"/><br/>

<input type="submit" value="go"/>

</form>

### Servlet1.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response){

**try**{

response.setContentType("text/html"); PrintWriter out = response.getWriter();

String n=request.getParameter("userName");

out.print("Welcome "+n);

Cookie ck=**new** Cookie("uname",n);//creating cookie object

response.addCookie(ck);//adding cookie in the response

//creating submit button

out.print("<form action='Servlet2'>");

out.print("<input type='submit' value='go'>"); out.print("</form>");

out.close();

}**catch**(Exception e){System.out.println(e);}

}

}

### Servlet2.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response){

**try**{ response.setContentType("text/html"); PrintWriter out = response.getWriter();

Cookie ck[]=request.getCookies();

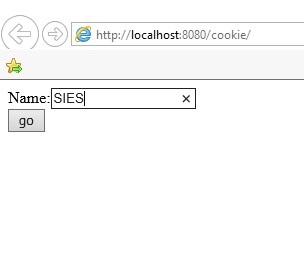
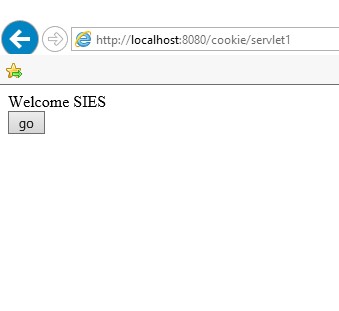
out.print("Hello "+ck[0].getValue());

out.close();

}**catch**(Exception e)

{System.out.println(e);} }

}



### 2.9 Finalizing a Servlet

When a servlet container determines that a servlet should be removed from service (for example, when a container wants to reclaim memory resources or when it is being shut down), the container calls the destroy method of the Servlet interface. In this method, you release any resources the servlet is using and save any persistent state.

public void destroy() { bookDB = null; }

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

■ Keep track of how many threads are currently running the service method.

■ Provide a clean shutdown by having the destroy method notify long-running threads of the shutdown and wait for them to complete.

■ Have the long-running methods poll periodically to check for shutdown and, if necessary,stop working, clean up, and return.