Servlets and JSP

Q: - What are Servlets?

Ans: - Servlets are the Java platform technology of choice for extending and enhancing Web servers. Servlets provide a component-based, platform-independent method for building Web-based applications, without the performance limitations of CGI programs. And unlike proprietary server extension mechanisms (such as the Netscape Server API or Apache modules), servlets are server- and platform-independent. This leaves you free to select a "best of breed" strategy for your servers, platforms, and tools.

Servlets have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. Servlets can also access a library of HTTP-specific calls and receive all the benefits of the mature Java language, including portability, performance, reusability, and crash protection.

Today servlets are a popular choice for building interactive Web applications. Third-party servlet containers are available for Apache Web Server, Microsoft IIS, and others. Servlet containers are usually a component of Web and application servers, such as BEA WebLogic Application Server, IBM WebSphere, Sun Java System Web Server, Sun Java System Application Server, and others.

The Java Servlet API allows a software developer to add dynamic content to a Web server using the Java platform. The generated content is commonly HTML, but may be other data such as XML. Servlets are the Java counterpart to non-Java dynamic Web content technologies such as PHP, CGI and ASP.NET. Servlets can maintain state across many server transactions by using HTTP cookies, session variables or URL rewriting.

The Servlet API, contained in the Java package hierarchy javax.servlet, defines the expected interactions of a Web container and a servlet. A Web container is essentially the component of a Web server that interacts with the servlets. The Web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights.

A Servlet is an object that receives a request and generates a response based on that request. The basic servlet package defines Java objects to represent servlet requests and responses, as well as objects to reflect the servlet's configuration parameters and execution environment. The package javax.servlet.http defines HTTP-specific subclasses of the generic servlet elements, including session management objects that track multiple requests and responses between the Web server and a client. Servlets may be packaged in a WAR file as a Web application.

Servlets can be generated automatically by JavaServer Pages (JSP), or alternately by template engines such as WebMacro. Often servlets are used in conjunction with JSPs in a pattern called "Model 2", which is a flavor of the model-view-controller pattern.

**ServletConfig and ServletContext**

There is only one ServletContext in every application. This object can be used by all the servlets to obtain application level information or container details. Every servlet, on the other hand, gets its own ServletConfig object. This object provides initialization parameters for a servlet. A developer can obtain the reference to ServletContext using either the ServletConfig object or ServletRequest object.

A Servlet container is a specialized web server that supports Servlet execution. It combines the basic functionality of a web server with certain Java/Servlet specific optimizations and extensions – such as an integrated Java runtime environment, and the ability to automatically translate specific URLs into Servlet requests. Individual Servlets are registered with a Servlet container, providing the container with information about what functionality they provide, and what URL or other resource locater they will use to identify themselves. The Servlet container is then able to initialize the Servlet as necessary and deliver requests to the Servlet as they arrive. Many containers have the ability to dynamically add and remove Servlets from the system, allowing new Servlets to quickly be deployed or removed without affecting other Servlets running from the same container. Servlet containers are also referred to as web containers or web engines.

Like the other Java APIs, different vendors provide their own implementation of the Servlet container standard. Below is a list of some of the free and commercial web containers. (Note that 'free' means that non-commercial use is free. Some of the commercial containers, e.g. Resin and Orion, are free to use in a server environment for non-profit organizations).

Q: - What are advantages of servlets over CGI?

Ans: - Servlets are Java technology's answer to CGI programming. They are programs that run on a Web server and build Web pages. Building Web pages on the fly is useful (and commonly done) for a number of reasons:

* **The Web page is based on data submitted by the user.** For example the results pages from search engines are generated this way, and programs that process orders for e-commerce sites do this as well.
* **The data changes frequently.** For example, a weather-report or news headlines page might build the page dynamically, perhaps returning a previously built page if it is still up to date.
* **The Web page uses information from corporate databases or other such sources.** For example, you would use this for making a Web page at an on-line store that lists current prices and number of items in stock.

What are the Advantage of Servlets Over "TraditionalCGI? Java servlets are more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI and than many alternative CGI-like technologies. (More importantly, servlet developers get paid more than Perl programmers :-).

* **Efficient.** With traditional CGI, a new process is started for each HTTP request. If the CGI program does a relatively fast operation, the overhead of starting the process can dominate the execution time. With servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous request to the same CGI program, then the code for the CGI program is loaded into memory N times. With servlets, however, there are *N* threads but only a single copy of the servlet class. Servlets also have more alternatives than do regular CGI programs for optimizations such as caching previous computations, keeping database connections open, and the like.
* **Convenient.** Hey, you already know Java. Why learn Perl too? Besides the convenience of being able to use a familiar language, servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.
* **Powerful.** Java servlets let you easily do several things that are difficult or impossible with regular CGI. For one thing, servlets can talk directly to the Web server (regular CGI programs can't). This simplifies operations that need to look up images and other data stored in standard places. Servlets can also share data among each other, making useful things like database connection pools easy to implement. They can also maintain information from request to request, simplifying things like session tracking and caching of previous computations.
* **Portable.** Servlets are written in Java and follow a well-standardized API. Consequently, servlets written for, say I-Planet Enterprise Server can run virtually unchanged on Apache, Microsoft IIS, or WebStar. Servlets are supported directly or via a plugin on almost every major Web server.
* **Inexpensive.** There are a number of free or very inexpensive Web servers available that are good for "personaluse or low-volume Web sites. However, with the major exception of Apache, which is free, most commercial-quality Web servers are relatively expensive. Nevertheless, once you have a Web server, no matter the cost of that server, adding servlet support to it (if it doesn't come preconfigured to support servlets) is generally free or cheap.

Can you explain Servlet life cycle?

Ans: - **Lifecycle of a Servlet**

The Servlet lifecycle consists of the following steps:

 1. The Servlet class is loaded by the container during start-up.

 2. The container calls the init() method. This method initializes the servlet and must be called before the servlet can service any requests. In the entire life of a servlet, the init() method is called only once.

 3. After initialization, the servlet can service client-requests. Each request is serviced in its own separate thread. The container calls the service() method of the servlet for every request. The service() method determines the kind of request being made and dispatches it to an appropriate method to handle the request. The developer of the servlet must provide an implementation for these methods. If a request for a method that is not implemented by the servlet is made, the method of the parent class is called, typically resulting in an error being returned to the requester.

 4. Finally, the container calls the destroy() method which takes the servlet out of service. The destroy() method like init() is called only once in the lifecycle of a Servlet.



Q: - What are the two important API's in for Servlets?

Ans: -

javax.servlet

javax.servlet.http

javax.servlet.jsp

javax.servlet.jsp.tagext

Q:- Can you explain in detail "javax.servlet" package?

Ans: -

|  |
| --- |
| **Class Summary** |
| [**GenericServlet**](http://tomcat.apache.org/tomcat-4.1-doc/servletapi/javax/servlet/GenericServlet.html) | Defines a generic, protocol-independent servlet. |
| [**ServletContextAttributeEvent**](http://tomcat.apache.org/tomcat-4.1-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-4.1-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-4.1-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-4.1-doc/servletapi/javax/servlet/ServletOutputStream.html) | Provides an output stream for sending binary data to the client. |
| [**ServletRequestWrapper**](http://tomcat.apache.org/tomcat-4.1-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-4.1-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. |

|  |
| --- |
|  |

Q: - What's the use of ServletContext?

Ans: - 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.

Q: - How do we define an application level scope for servlet?

Ans: -

You can set default variables and application-level constants on the Application.cfm page. For example, you can specify the following values:

* A data source
* A domain name
* Style settings, such as fonts or colors
* Other important application-level variables

In the Servlets 2.1 spec, you can set a Servlet Context Attribute. For example, getServletContext().setAttribute("counter", new foo.Counter()); . You can also access (or initialize) these Attributes as Application Scope Beans in a JSP, using <jsp:useBean scope="application">.

Q: - What's the difference between GenericServlet and HttpServlet?

Ans: -

HttpServlet Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. A subclass of HttpServlet must override at least one method, usually one of these:

* doGet, if the servlet supports HTTP GET requests
* doPost, for HTTP POST requests
* doPut, for HTTP PUT requests
* doDelete, for HTTP DELETE requests
* init and destroy, to manage resources that are held for the life of the servlet
* getServletInfo, which the servlet uses to provide information about itself

There’s almost no reason to override the service method. service handles standard HTTP requests by dispatching them to the handler methods for each HTTP request type (the do*XXX* methods listed above). Likewise, there’s almost no reason to override the doOptions and doTrace methods.

GenericServlet defines a generic, protocol-independent servlet. To write an HTTP servlet for use on the Web, extend HttpServlet instead.

GenericServlet implements the Servlet and ServletConfig interfaces. GenericServlet may be directly extended by a servlet, although it’s more common to extend a protocol-specific subclass such as HttpServlet.

GenericServlet makes writing servlets easier. It provides simple versions of the lifecycle methods init and destroy and of the methods in the ServletConfig interface. GenericServlet also implements the log method, declared in the ServletContext interface.

To write a generic servlet, you need only override the abstract service method

Q: - Can you explain in detail javax.servlet.http package?

Ans: -

**Package javax.servlet.http**

|  |
| --- |
| **Interface Summary** |
| [***HttpServletRequest***](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpServletRequest.html) | Extends the [ServletRequest](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/ServletRequest.html) interface to provide request information for HTTP servlets. |
| [***HttpServletResponse***](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpServletResponse.html) | Extends the [ServletResponse](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/ServletResponse.html) interface to provide HTTP-specific functionality in sending a response. |
| [***HttpSession***](http://java.sun.com/products/servlet/2.2/javadoc/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. |
| [***HttpSessionBindingListener***](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpSessionBindingListener.html) | Causes an object to be notified when it is bound to or unbound from a session. |
| [***HttpSessionContext***](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpSessionContext.html) | **Deprecated.** *As of Java(tm) Servlet API 2.1 for security reasons, with no replacement.* |

|  |
| --- |
| **Class Summary** |
| [**Cookie**](http://java.sun.com/products/servlet/2.2/javadoc/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://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpServlet.html) | Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. |
| [**HttpSessionBindingEvent**](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpSessionBindingEvent.html) | Sent to an object that implements [HttpSessionBindingListener](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpSessionBindingListener.html) when the object is bound to or unbound from the session. |
| [**HttpUtils**](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpUtils.html) | Provides a collection of methods that are useful in writing HTTP servlets. |

Q: - What's the architecture of a Servlet package?

Ans: -

## Architecture of the Servlet Package

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

### The Servlet Interface

### The central abstraction in the Servlet API is the [Servlet](http://java.sun.com/products/servlet/2.1/api/javax.servlet.Servlet.html) (in the API reference documentation)interface. All servlets implement this interface, either directly or, more commonly, by extending a class that implements it such as [HttpServlet](http://java.sun.com/products/servlet/2.1/api/javax.servlet.http.HttpServlet.html)(in the API reference documentation)



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

### Client Interaction

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

* A [ServletRequest](http://java.sun.com/products/servlet/2.1/api/javax.servlet.ServletRequest.html), which encapsulates the communication from the client to the server.
* A [ServletResponse](http://java.sun.com/products/servlet/2.1/api/javax.servlet.ServletResponse.html), which encapsulates the communication from the servlet back to the client.

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

### The ServletRequest Interface

The ServletRequest interface allows the servlet access to:

* Information such as the names of the parameters passed in by the client, the protocol (scheme) being used by the client, and the names of the remote host that made the request and the server that received it.
* The input stream, [ServletInputStream](http://java.sun.com/products/servlet/2.1/api/javax.servlet.ServletInputStream.html). Servlets use the input stream to get data from clients that use application protocols such as the HTTP POST and PUT methods.

Interfaces that extend ServletRequest interface allow the servlet to retrieve more protocol-specific data. For example, the [HttpServletRequest](http://java.sun.com/products/servlet/2.1/api/javax.servlet.http.HttpServletRequest.html)interface contains methods for accessing HTTP-specific header information.

### The ServletResponse Interface

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

* Allows the servlet to set the content length and MIME type of the reply.
* Provides an output stream, [ServletOutputStream](http://java.sun.com/products/servlet/2.1/api/javax.servlet.ServletOutputStream.html), and a Writer through which the servlet can send the reply data.

Interfaces that extend the ServletResponse interface give the servlet more protocol-specific capabilities. For example, the [HttpServletResponse](http://java.sun.com/products/servlet/2.1/api/javax.servlet.http.HttpServletResponse.html)interface contains methods that allow the servlet to manipulate HTTP-specific header information.

Q: - Why is HTTP protocol called as a stateless protocol?

Ans: -

HTTP is a [stateless](http://en.wikipedia.org/wiki/Stateless_server) protocol. The advantage of a stateless protocol is that hosts do not need to retain information about users between requests, but this forces [web developers](http://en.wikipedia.org/wiki/Web_developer) to use alternative methods for maintaining users' states. For example, when a host would like to customize content for a user while visiting a [website](http://en.wikipedia.org/wiki/Website), the [web application](http://en.wikipedia.org/wiki/Web_application) must be written to track the user's progress from page to page. A common method for solving this problem involves sending and requesting [cookies](http://en.wikipedia.org/wiki/HTTP_cookie). Other methods include server side sessions, hidden variables (when current page is a [form](http://en.wikipedia.org/wiki/Form_%28web%29)), and [URL](http://en.wikipedia.org/wiki/Uniform_Resource_Locator) encoded parameters (such as /index.php?userid=3).

Q: - What are the different ways we can maintain state between requests?

Ans: -

Developers are an inventive bunch and quickly found solutions. The simplest of all was to encode hidden parameters inside an HTML form, such as the username and type of transaction being made. Though not evident to the user, additional state information has been added to the form, which will be passed when the form is submitted. If every form included a hidden field indicating the user, then it would be possible to track user actions across HTTP requests. Of course, additional information can also be included, such as the type of operation being performed, as well as the focus of the operation. The following HTML code illustrates the use of hidden fields:

<form action="/servlet/order" method=post>

**<input type="hidden" name=userid value="5532211">**

**<input type="hidden" name=operation value="add">**

**<input type="hidden" name=product\_id value="4432A">**

Product ID : 4432A

Quantity : <input type="quantity" value=1>

<input type=submit>

</form>

A variation on this theme is to use hyperlinks, which generate HTTP GET requests. Often a hyperlink is better than an HTML form, as most users are more comfortable with hyperlinks than buttons. Parameters can be added to the end of each URL, so that state information is passed when the next connection is made. For example, every hyperlink on the page could contain a userID code to allow user activity to be tracked. This information can be fetched by a servlet using

javax.servlet.http.HttpServletRequest.getParameter(String)

just as a standard CGI parameter would.

<a href="/servlet/shopping**?userID=5532211**>View Shopping Cart</a>

Whether a hyperlink or an HTML form is used, this state information must be included every time a request is made, so that subsequent requests can gain access to it. If your servlet misses a URL or a hidden variable is missed from a form, the state information is lost. Such solutions are crude but effective. They do the job but are an inelegant and overly complex way of maintaining state across HTTP requests. A better way of maintaining state information is to use persistent client state cookies.

Cookies

To solve the problem of maintaining persistence across HTTP connections, Netscape Communications developed a specification for "cookies". Cookies are state objects stored by a Web browser (or other HTTP client) and can be used by server-side applications to store and retrieve information. Cookies can be created by servlets (or CGI scripts) and sent to the browser. For every subsequent request made by the browser, the cookie is sent as part of the HTTP request. This allows server-side applications to access state information, without the effort of encoding it in a hyperlink or HTML form. Figure 1 shows an example transaction between a server-side application and a browser where a cookie is stored and then returned.

To understand how cookies work, let's consider the analogy of a bank. When signing up for an account, most customers will be given a card that contains identification information. Every time the customer uses an ATM, he or she presents the card. State information (the user's account number) persists across each transaction, even though transactions may be days, weeks, or even months apart. Cookies are no different — the state information issued by a server-side application is presented later, when the next HTTP request is made.

Each cookie is named and contains a single value. The cookie is effectively a mapping between a single key and a single value. If developers need to store multiple values, they can use more than one cookie or can encode multiple values using some sort of separator character for the cookie data. Remember, however, that the number of cookies per domain name varies from browser to browser. Netscape imposes a limit of 20 cookies per domain, and a maximum size of 4096 bytes per cookie. As a general rule, cookies should only contain small pieces of data (to identify a user or a session). You have the option of using an identifier stored in a cookie to look up larger pieces of data stored server-side.

#### Storing cookies from a Java servlet

Support for cookies has been included in the Servlet API and provides an extremely easy interface for storing and retrieving cookies. Cookies are represented by the

|  |
| --- |
| javax.servlet.http.Cookie |

class. The constructor takes two strings as parameters — the name of the cookie (which is fixed) and the value (which can be changed at a later date).

// Create a new cookie

Cookie myCookie = new Cookie ( "accountID" , "212994234");

Servlets that wish to set cookies must add their cookie to the response sent back to the browser.

|  |
| --- |
| HttpServletResponse |

offers an

|  |
| --- |
| addCookie(Cookie) |

method, which can be invoked once or multiple times to add additional cookies.

public void doGet(HttpServletRequest req, HttpServletResponse res)

{

 // Store state information in browser cookies

 res.addCookie (new Cookie ("thecounter", "1");

 // Additional servlet code would go here.....

}

#### Reading cookies from a Java servlet

Accessing stored cookies from a servlet is also easy. Cookies are sent each time a request is made, so that if a cookie is already stored in the browser, it can be accessed by invoking the

|  |
| --- |
| Cookies[] getCookies() |

method of

|  |
| --- |
| javax.servlet.http.HttpServletRequest |

. This returns an array of Cookie objects, or

|  |
| --- |
| null |

if no cookies are present.

// Get cookie array from HttpServletRequest

Cookie[] cookieArray = request.getCookies();

// Guard statement to check for missing cookies

if (cookieArray != null)

{

 // Print a list of all cookies sent by browser

 for (int i =0; i< cookieArray.length; i++)

 {

 Cookie c = cookieArray[i];

 pout.println ("Name : " + c.getName());

 pout.println ("Value: " + c.getValue());

 }

}

else

pout.println ("No cookies present, or browser does not support cookies");

#### Putting cookies to work

To demonstrate cookies in action, let's look at a simple example that tracks the number of times a user has visited a site. Though no unique identifier is to be stored as a cookie, the number of visits made by the user will persist over time. This shows that state has been maintained across connections.

We start by checking for the presence of any cookies, and then a specific cookie named "count." If the cookie exists, we read the value and then display it to the user. Since the idea of a counter is to increase on every visit, we increment the counter value and send it back to the user as a cookie. As you reload the page, the cookie is incremented, and state information persists across connections. Note too that, by default, the cookie will persist only while the browser remains open, unless a specific time period is assigned to the cookie (see Listing 1).

Q: - What is URL rewriting?

Ans: -

**URL Rewriting** is a technique for saving state information on the user's browser between page hits. It's sort of like cookies, only the information gets stored inside the URL, as an additional parameter. The HttpSession API, which is part of the Servlet API, sometimes uses URL Rewriting when cookies are unavailable.

**Example:** changing <A HREF="nextpage.html"> into
<A HREF="nextpage.html;$sessionid$=DSJFSDKFSLDFEEKOE"> (or whatever the actual syntax is; I forget offhand)

(Unfortunately, the method in the Servlet API for doing URL rewriting for session management is called encodeURL(). Sigh...)

**So once again** assume that a user has searched for some books and he has been presented with a search result that has 2 books listed. It is basically a Form with 2 checkboxes, each for one book and a Submit button to add any of these book to his Cart.

|  |
| --- |
| <b>Search results for books</b><form method="post" action="serverprogram.jsp"><input type="checkbox" name="bookID" value="100">Java Servlet Programming<br><input type="checkbox" name="bookID" value="101">Professional JSP<br><input type="submit" name="Submit" value="Add to Cart"><br></form> |

Now once again suppose the user selects the book named 'Java Servlet Programming' and then clicks on the Submit button. This would pass the contents of the form to the server side program called serverprogram.jsp which should read the selected checkboxes and do the necessary (i.e.. make some arrangements to keep a track of the selected books, which basically means implement session tracking). Now suppose the user continues browsing and searches for more books and is presented with a new search result just like in the previous example. For better understanding I shall once again give you the same 2 results as shown in hidden fields method. The 2 books named 'Teach yourself WML Programming' and 'Teach yourself C++'

|  |
| --- |
| <b>Search results for books</b><form method="post" action="serverprogram.jsp?bookID=100"><input type="checkbox" name="bookID" value="150">Teach yourself WML Programming<br><input type="checkbox" name="bookID" value="160">Teach yourself C++<br><input type="submit" name="Submit" value="Add to Cart"><br></form> |

You should be able to guess by now what URL rewriting is all about. In the above html source, the target for the form has been changed from serverprogram.jsp to serverprogram.jsp?bookID=100 . This is exactly what URL Rewriting means. The original URL which was only serverprogram.jsp has now been rewritten as serverprogram.jsp?bookID=100 . The effect of this is that the any part of the URL after the ? (question mark) is treated as extra parameters that are passed to the server side program. They are known as GET parameters. GET method of submitting forms always uses URL Rewriting. Now when the serverprogram.jsp fetches the parameters by the name bookID it would be presented with the one that was present after the ? in the URL as well as the newly selected checkboxes by the user in that Form.

Consider a general example where a user has selected 2 values, then whenever a program generates a new Form the target for that form should look something like

|  |
| --- |
| <form method="post" action="serversideprogram.jsp?name1=value1+name2=value2"> |

This sort of URL would keep on increasing as more and more values have to be carried on from one page to another.

Q: - What are cookies?

Q: - What are sessions in Servlets?

Ans: -

## 1. What is Session Tracking?

There are a number of problems that arise from the fact that HTTP is a "stateless" protocol. In particular, when you are doing on-line shopping, it is a real annoyance that the Web server can't easily remember previous transactions. This makes applications like shopping carts very problematic: when you add an entry to your cart, how does the server know what's already in your cart? Even if servers did retain contextual information, you'd still have problems with e-commerce. When you move from the page where you specify what you want to buy (hosted on the regular Web server) to the page that takes your credit card number and shipping address (hosted on the secure server that uses SSL), how does the server remember what you were buying?

There are three typical solutions to this problem.

1. **Cookies.** You can use HTTP cookies to store information about a shopping session, and each subsequent connection can look up the current session and then extract information about that session from some location on the server machine. This is an excellent alternative, and is the most widely used approach. However, even though servlets have a [high-level and easy-to-use interface to cookies](http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/Servlet-Tutorial-Cookies.html), there are still a number of relatively tedious details that need to be handled:
	* Extracting the cookie that stores the session identifier from the other cookies (there may be many, after all),
	* Setting an appropriate expiration time for the cookie (sessions interrupted by 24 hours probably should be reset), and
	* Associating information on the server with the session identifier (there may be far too much information to actually store it in the cookie, plus sensitive data like credit card numbers should *never* go in cookies).
2. **URL Rewriting.** You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session. This is also an excellent solution, and even has the advantage that it works with browsers that don't support cookies or where the user has disabled cookies. However, it has most of the same problems as cookies, namely that the server-side program has a lot of straightforward but tedious processing to do. In addition, you have to be very careful that every URL returned to the user (even via indirect means like Location fields in server redirects) has the extra information appended. And, if the user leaves the session and comes back via a bookmark or link, the session information can be lost.
3. **Hidden form fields.** HTML forms have an entry that looks like the following: <INPUT TYPE="HIDDEN" NAME="session" VALUE="...">. This means that, when the form is submitted, the specified name and value are included in the GET or POST data. This can be used to store information about the session. However, it has the major disadvantage that it only works if every page is dynamically generated, since the whole point is that each session has a unique identifier.

Servlets provide an outstanding technical solution: the HttpSession API. This is a high-level interface built on top of cookies or URL-rewriting. In fact, on many servers, they use cookies if the browser supports them, but automatically revert to URL-rewriting when cookies are unsupported or explicitly disabled. But the servlet author doesn't need to bother with many of the details, doesn't have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store data that is associated with each session.

## 2. The Session Tracking API

Using sessions in servlets is quite straightforward, and involves looking up the session object associated with the current request, creating a new session object when necessary, looking up information associated with a session, storing information in a session, and discarding completed or abandoned sessions.

### 2.1 Looking up the [HttpSession](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpSession.html) object associated with the current request.

This is done by calling the getSession method of HttpServletRequest. If this returns null, you can create a new session, but this is so commonly done that there is an option to automatically create a new session if there isn't one already. Just pass true to getSession. Thus, your first step usually looks like this:

 HttpSession session = request.getSession(true);

### 2.2 Looking up Information Associated with a Session.

HttpSession objects live on the server; they're just automatically associated with the requester by a behind-the-scenes mechanism like cookies or URL-rewriting. These session objects have a builtin data structure that let you store any number of keys and associated values. In version 2.1 and earlier of the servlet API, you use getValue("key") to look up a previously stored value. The return type is Object, so you have to do a typecast to whatever more specific type of data was associated with that key in the session. The return value is null if there is no such attribute. In version 2.2, getValue is deprecated in favor of getAttribute, both because of the better naming match with setAttribute (the match for getValue is putValue, not setValue), and because setAttribute lets you use an attached [HttpSessionBindingListener](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpSessionBindingListener.html) to monitor values, while putValue doesn't. Nevertheless, since few commercial servlet engines yet support version 2.2, I'll use getValue in my examples. Here's one representative example, assuming ShoppingCart is some class you've defined yourself that stores information on items being purchased.

 HttpSession session = request.getSession(true);

 **ShoppingCart previousItems =**

 **(ShoppingCart)session.getValue("previousItems");**

 if (previousItems != null) {

 doSomethingWith(previousItems);

 } else {

 previousItems = new ShoppingCart(...);

 doSomethingElseWith(previousItems);

 }

In most cases, you have a specific attribute name in mind, and want to find the value (if any) already associated with it. However, you can also discover all the attribute names in a given session by calling getValueNames, which returns a String array. In version 2.2, use getAttributeNames, which has a better name and which is more consistent in that it returns an Enumeration, just like the getHeaders and getParameterNames methods of HttpServletRequest.

Although the data that was explicitly associated with a session is the part you care most about, there are some other pieces of information that are sometimes useful as well.

* **getId.** This method returns the unique identifier generated for each session. It is sometimes used as the key name when there is only a single value associated with a session, or when logging information about previous sessions.
* **isNew.** This returns true if the client (browser) has never seen the session, usually because it was just created rather than being referenced by an incoming client request. It returns false for preexisting sessions.
* **getCreationTime.** This returns the time, in milliseconds since the epoch, at which the session was made. To get a value useful for printing out, pass the value to the Date constructor or the setTimeInMillis method of GregorianCalendar.
* **getLastAccessedTime.** This returns the time, in milliseconds since the epoch, at which the session was last sent from the client.
* **getMaxInactiveInterval.** This returns the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never timeout.

### 2.3 Associating Information with a Session

As discussed in the previous section, you read information associated with a session by using getValue (or getAttribute in version 2.2 of the servlet spec). To specify information, you use putValue (or setAttribute in version 2.2), supplying a key and a value. Note that putValue replaces any previous values. Sometimes that's what you want (as with the referringPage entry in the example below), but other times you want to retrieve a previous value and augment it (as with the previousItems entry below). Here's an example:

 HttpSession session = request.getSession(true);

 **session.putValue("referringPage", request.getHeader("Referer"));**

 ShoppingCart previousItems =

 (ShoppingCart)session.getValue("previousItems");

 if (previousItems == null) {

 previousItems = new ShoppingCart(...);

 }

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

 previousItems.addEntry(Catalog.getEntry(itemID));

 // You still have to do putValue, not just modify the cart, since

 // the cart may be new and thus not already stored in the session.

 **session.putValue("previousItems", previousItems);**

## 3. Example: Showing Session Information

Here is a simple example that generates a Web page showing some information about the current session. You can also [download the source](http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/hall/ShowSession.java) or [try it on-line](http://webdev.apl.jhu.edu/servlet/hall.ShowSession).

package hall;

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.net.\*;

import java.util.\*;

/\*\* Simple example of session tracking. See the shopping

 \* cart example for a more detailed one.

 \* <P>

 \* Part of tutorial on servlets and JSP that appears at

 \* http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/

 \* 1999 Marty Hall; may be freely used or adapted.

 \*/

public class ShowSession extends HttpServlet {

 public void doGet(HttpServletRequest request,

 HttpServletResponse response)

 throws ServletException, IOException {

 HttpSession session = request.getSession(true);

 response.setContentType("text/html");

 PrintWriter out = response.getWriter();

 String title = "Searching the Web";

 String heading;

 Integer accessCount = new Integer(0);;

 if (session.isNew()) {

 heading = "Welcome, Newcomer";

 } else {

 heading = "Welcome Back";

 Integer oldAccessCount =

 // Use getAttribute, not getValue, in version

 // 2.2 of servlet API.

 (Integer)session.getValue("accessCount");

 if (oldAccessCount != null) {

 accessCount =

 new Integer(oldAccessCount.intValue() + 1);

 }

 }

 // Use putAttribute in version 2.2 of servlet API.

 session.putValue("accessCount", accessCount);

 out.println(ServletUtilities.headWithTitle(title) +

 "<BODY BGCOLOR=\"#FDF5E6\">\n" +

 "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +

 "<H2>Information on Your Session:</H2>\n" +

 "<TABLE BORDER=1 ALIGN=CENTER>\n" +

 "<TR BGCOLOR=\"#FFAD00\">\n" +

 " <TH>Info Type<TH>Value\n" +

 "<TR>\n" +

 " <TD>ID\n" +

 " <TD>" + session.getId() + "\n" +

 "<TR>\n" +

 " <TD>Creation Time\n" +

 " <TD>" + new Date(session.getCreationTime()) + "\n" +

 "<TR>\n" +

 " <TD>Time of Last Access\n" +

 " <TD>" + new Date(session.getLastAccessedTime()) + "\n" +

 "<TR>\n" +

 " <TD>Number of Previous Accesses\n" +

 " <TD>" + accessCount + "\n" +

 "</TABLE>\n" +

 "</BODY></HTML>");

 }

 public void doPost(HttpServletRequest request,

 HttpServletResponse response)

 throws ServletException, IOException {

 doGet(request, response);

 }

}

Here's a typical result, shown after visiting the page several without quitting the browser in between:



Q: - What's the difference between getSession(true) and getSession(false)?

Ans: -

public interface **HttpServletRequest** extends [ServletRequest](http://java.sun.com/j2ee/sdk_1.2.1/techdocs/api/javax/servlet/ServletRequest.html)

**Method: -**

### getSession

public [HttpSession](http://java.sun.com/j2ee/sdk_1.2.1/techdocs/api/javax/servlet/http/HttpSession.html) **getSession**(boolean create)

Returns the current HttpSession associated with this request or, if if there is no current session and create is true, returns a new session.

If create is false and the request has no valid HttpSession, this method returns null.

To make sure the session is properly maintained, you must call this method before the response is committed.

**Parameters:**

true - to create a new session for this request if necessary; false to return null if there's no current session

**Returns:**

the HttpSession associated with this request or null if create is false and the request has no valid session

Q: - What's the difference between "doPost" and "doGet" methods?

Ans: -

**Servlet Backend**

[ExampServlet.java](http://java.sun.com/developer/onlineTraining/Programming/BasicJava1/Code/ExampServlet.java) builds an HTML page to return to the end user. This means the servlet code does not use any Project Swing or Abstract Window Toolkit (AWT) components or have event handling code. For this simple servlet, you only need to import these packages:

* java.io for system input and output. The HttpServlet class uses the IOException class in this package to signal that an input or output exception of some kind has occurred.
* javax.servlet, which contains generic (protocol-independent) servlet classes. The HttpServlet class uses the ServletException class in this package to indicate a servlet problem.
* javax.servlet.http, which contains HTTP servlet classes. The HttpServlet class is in this package.

|  |
| --- |
| import java.io.\*;import javax.servlet.\*;import javax.servlet.http.\*;public class ExampServlet extends HttpServlet { public void doPost(HttpServletRequest request,  HttpServletResponse response) throws ServletException, IOException { response.setContentType("text/html"); PrintWriter out = response.getWriter(); out.println("<title>Example</title>" + "<body bgcolor=FFFFFF>"); out.println("<h2>Button Clicked</h2>"); String DATA = request.getParameter("DATA"); if(DATA != null){ out.println(DATA); } else { out.println("No text entered."); } out.println("<P>Return to  <A HREF="../simpleHTML.html">Form</A>"); out.close(); }} |

## 1. Basic Servlet Structure

Here's the outline of a basic servlet that handles GET requests. GET requests, for those unfamiliar with HTTP, are requests made by browsers when the user types in a URL on the address line, follows a link from a Web page, or makes an HTML form that does not specify a METHOD. Servlets can also very easily handle POST requests, which are generated when someone creates an HTML form that specifies METHOD="POST". We'll discuss that in later sections.

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class SomeServlet extends HttpServlet {

 public void doGet(HttpServletRequest request,

 HttpServletResponse response)

 throws ServletException, IOException {

 *// Use "request" to read incoming HTTP headers (e.g. cookies)*

 *// and HTML form data (e.g. data the user entered and submitted)*

 *// Use "response" to specify the HTTP response line and headers*

 *// (e.g. specifying the content type, setting cookies).*

 PrintWriter out = response.getWriter();

 *// Use "out" to send content to browser*

 }

}

(Download [template source code](http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/hall/SomeServlet.java) -- click with the right mouse on the link or hold down SHIFT while clicking on the link.)

To be a servlet, a class should extend [HttpServlet](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpServlet.html) and override doGet or doPost (or both), depending on whether the data is being sent by GET or by POST. These methods take two arguments: an [HttpServletRequest](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpServletRequest.html) and an [HttpServletResponse](http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpServletResponse.html). The HttpServletRequest has methods that let you find out about incoming information such as FORM data, HTTP request headers, and the like. The HttpServletResponse has methods that lets you specify the HTTP response line (200, 404, etc.), response headers (Content-Type, Set-Cookie, etc.), and, most importantly, lets you obtain a [PrintWriter](http://java.sun.com/products/jdk/1.2/docs/api/java/io/PrintWriter.html) used to send output back to the client. For simple servlets, most of the effort is spent in println statements that generate the desired page. Note that doGet and doPost throw two exceptions, so you are required to include them in the declaration. Also note that you have to import classes in java.io (for PrintWriter, etc.), javax.servlet (for HttpServlet, etc.), and javax.servlet.http (for HttpServletRequest and HttpServletResponse). Finally, note that doGet and doPost are called by the service method, and sometimes you may want to override service directly, e.g. for a servlet that handles both GET and POST request.

## 2. A Simple Servlet Generating Plain Text

Here is a simple servlet that just generates plain text. The following section will show the more usual case where HTML is generated.

### 2.1 HelloWorld.java

You can also [download the source](http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/hall/HelloWorld.java) or [try it on-line](http://webdev.apl.jhu.edu/servlet/hall.HelloWorld).

package hall;

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

**public class HelloWorld extends HttpServlet {**

 **public void doGet(HttpServletRequest request,**

 **HttpServletResponse response)**

 **throws ServletException, IOException {**

 **PrintWriter out = response.getWriter();**

 **out.println("Hello World");**

 **}**

**}**

Q: - Which are the different ways you can communicate between servlets?

Ans: -

Java has brought much in the way of programming advancement within easy reach of developers. Powerful constructs such as multithreading and advance communication are now relatively easy for you to use in your programs. Java has also brought a strong sense of object orientation. The ability to reuse code and not reinvent the wheel each time you program is a great improvement over alternative programming languages.

But for a developer to reuse code, the language has to support the mechanisms that make it easy to build complex systems by simply plugging objects together. Java has these mechanisms - the servlet API is no stranger to object reuse. This article outlines the different methods by which servlets can communicate data to one another. This communication promotes the concept of solving a problem only once and then applying the solution to many different areas.

As this article will illustrate, you, as a servlet developer, have a lot of choices on how you can best reuse your servlet classes.

**Common Base Class**
A servlet is run in response to a client connection. It is designed to be lightweight and to process the client request as quickly as possible, thus freeing up the server so it can immediately process the next request that comes in. A servlet should never try to do everything itself. In a large system it is better to produce many smaller servlets, as opposed to a small number of large servlets. The longer a servlet spends servicing a request, the fewer clients the servlet can process in a given time.

If smaller servlets are to be built, rather than using the one-servlet-solution-fits-all approach, the ability to share information between the servlets is paramount.

One of the easiest ways for a servlet to share information (this works for classes as well) is to share common objects through inheritance - where the servlets are extended from a class other than HttpServlet or GenericServlet. To illustrate this concept, consider the example shown in Figure 1.

Suppose we had two servlets in our system: one to log a user onto our system and the other to log him. Let's assume that the reason we want a user to log on and log off again is to enable us to connect to a specific database given the username and password. Now, let us assume that they will share various methods, such as a simple check to see if the user is already logged on. We can develop methods for this simple check and place a copy of them in each servlet. However, since it's a database connection, let's also assume that we will be sharing database connections between the servlets.

Many different methods are available to facilitate method sharing and data pooling. For this section, we will use the simplest method. Instead of each servlet extending the HttpServlet, each will extend the capabilities of ServletBase, which is a custom class we will build. This class will itself extend the HttpServlet class, but it will also add other methods to allow the servlets to intelligently handle the database connections.

Listing 1 shows the basic skeleton implementation for the ServletBase class. You'll notice that it looks very similar to a normal servlet, except it doesn't override the service(...) method or any of the doXXX(...) methods. It does, however, override the init(...) method. In this instance it needs to open up a connection to the database which will be shared through the static Connection variable. This will be performed when the servlet first initializes.

However, please note a very important fact. The init(...) method will be called multiple times if multiple servlets inherit this class because each particular servlet needs to be initialized and needs to go through the complete servlet life cycle. Since we don't want to have additional database connections open, we'll check to see if it has been opened or not, and if so, we skip over that section.

The two additional methods, isLoggedOn(...) and logUserOn(...), are support functions that are accessible by all servlets that extend this class.

So now that we have developed the new base class, let's look at how we can use it. Listing 2 shows the logonServlet that will log a user onto the system. This class, instead of extending the HttpServlet class, extends the ServletBase class. Notice that this class does not override the init(...) method because the base class is taking care of this for us.

The API doesn't require the init(...) method to be overridden like the previous versions insisted it should be, so it may be left out without any adverse effects.

The logonServlet can now call methods from the underlying base class without having to wrestle with any object references. This allows class to easily share common methods and data without keeping a reference to a collection class.

This ability is not hard, and it's not new. It is used throughout Java to create groups of logical units, and it can easily be used in the world of servlets, where the need to share data is even more important.

**Sharing Data**
If the previous method seems a little clunky and too much of a hassle, the 2.1 version of the servlet API introduces a way to share objects among servlets. It operates along the same lines as the session management functionality, but it is done completely on the server side.

In the session management, you associated an object with a given client session. The session manager attempted to return the same object to you each time that particular client made a request. This was done using cookies or URL rewriting. Objects were stored using a unique string to reference them.

Servlets can now benefit from this technology through the ServletContext interface. Each servlet runs within some sort of environment. This environment describes various parameters associated with the servlet environment, such as the document root and any mappings that may exist. This is known as the ServletContext. A servlet belongs to only one ServletContext and the server's Web administrator controls this.

For example, a server running virtual hosts would have a different ServletContext for each virtual host. This has the advantage of logically grouping servlets while maintaining different security models for each type.

The ServletContext can store objects on behalf of servlets running within its context. There are methods that allow a servlet to add and remove objects and to determine what the objects are that are stored. These objects can be retrieved and modified by other servlets running within the context.

Let's now look at how we could share a database connection among servlets using the ServletContext mechanisms. Listing 3 shows a simple example where we create a new database connection from some method and ask the ServletContext to store it for us under the name of "database.connection" using the setAttribute(...) method.

Retrieving the database connection again is just as easy as setting it. Listing 4 shows another servlet, running within the same context, which attempts to retrieve the object back.

If the object has not been stored, the getAttribute(...) method will return null. As you can see, the whole storage and retrieval system operates very much like the cookie system - so this can be thought of as server-side cookies!

Listing 5 illustrates another useful method from the ServletContext interface. The ability to discover which objects are in existence is a very handy feature; for example, it enables you to discover whether any initialization has been performed. The getAttributeNames() method returns an Enumeration to the list of stored objects, which can then be returned using the getAttribute() method.

We have now seen methods for storing and retrieval of data. One additional method allows you to remove a given object from the ServletContext. This method, removeAttribute(...), ensures that the ServletContext no longer holds a reference to the object.

So the ability to share data among servlets running within the same context is straightforward enough. But what if the servlet isn't running within the same context? What if a servlet running on another machine or another virtual host has data that the current servlet needs? Have no fear; help is at hand.

The 2.1 servlet API introduces a new interface that allows servlets to get an external reference to a context. This is done through an overloaded version of getServletContext(...) which takes in the URL of the servlet you wish to retrieve the context for. An example of this call is shown in Listing 6.

Assuming the remote host allows this to be performed, the ServletContext will be returned. However, if it is not, then the method will return null. In calling this method, it will not invoke the servlet running in the other context, nor will it interpret any processing that that remote servlet may be doing at that point.

Using this new mechanism from the servlet API, you can easily share objects with other developers, including other contexts.

**Servlets Working Together**
In the previous section, we looked at a number of ways to share objects between servlets. In most instances this is used to share common data resources such as a database connection. This can allow for more efficient operation within the server. Now, thanks to the new addition to the servlet API, servlets can work together in yet another way.

In past version releases, a technique known as servlet chaining was possible. In servlet chaining, the output of one servlet fed the input to another servlet. The second servlet, or the next one in the chain, could perform additional processing before sending the result out to the client. Although this was useful, it was a bit clunky to operate. To begin with, the chaining mechanism had to be set up by the Web administrator, and it could not be performed dynamically. Also, the servlet had no control over whether it should be in a chain. From the developer's point of view, no API control was possible.

The new API has made this feature much more accessible. You have the ability to insert the contents of one servlet within the output of another, and you can pass a request on to another servlet for processing.

The following sections will show you how you can use these features in your servlets.

**Forwarding a Request**
You may find a time when it's better to pass a request on to another servlet for processing. For example, consider a servlet that acts as a front-end processor for a search engine. Depending on the result of the query, different servlets could be used to generate the actual output. The first servlet would take in all the search parameters, run the query, and pass the results on to a specific servlet for output. The advantage over the conventional servlet-chaining model is that the servlet can decide at runtime which output servlet will be used as opposed to relying on a Web administrator to set up the mapping. This is particularly useful in an ISP environment where the developer has to rely on a third party to get this right.

To handle this scenario and the one in the next section, a new interface was developed. The RequestDispatcher interface is the glue that controls the flow between servlets and output streams such as HTTP clients, files, or even JavaServer Pages (JSP).

Before you can send control to another servlet, you must first get the necessary RequestDispatcher for that particular servlet. This is controlled through the method call getRequestDispatcher(...) from the ServletContext interface. You specify the URI to the servlet, and assuming no security violations incur, the necessary RequestDispatcher reference is returned.

From here, you can make a call to the forward(...) method, which passes on the request to the servlet represented by theRequestDispatcher. Let's look at an example of this in action.

Listing 7 shows a typical use of this mechanism. Assume that the forwardServlet shown takes in a search query from the user. It first retrieves an instance to the context it is running within, then using this reference a call to retrieve the RequestDispatcher for the xyzServlet that will be used to generate our output for us.

Before we pass on the request for processing, we run the query. The next servlet will be used to process the results for this query, so we need some mechanism to pass the results to the next servlet.

As demonstrated in the previous section, sharing objects between servlets is not that big a problem; we simply ask the ServletContext to store the data for us. But this is not satisfactory, as these objects take no account of the client request. Storing the results this way would be very dangerous since only one copy per ServletContext would exist, as opposed to one copy per client request.

You could use session management to get around this problem, but since the data hasn't actually made it back to the client, it's not an ideal solution.

Thanks to the ServletRequest object, help is available. There are methods that allow servlets to ask that the ServletRequest object store objects for it on its behalf, ensuring that the object remains intact per client session. The first servlet can then pass data to the second servlet without worrying about multiple-client sessions.

As you'll see in Listing 7, the method for storing data is exactly the same as was demonstrated for storing objects using the ServletContext. The equivalent getAttribute(...) methods exist to allow the servlet to retrieve the objects again.

When you are forwarding a request to another servlet for processing, the first servlet is not permitted to do anything to the output, including setting any HTTP status fields. As soon as the first servlet makes an attempt to retrieve an output stream, then the call to forward(...) will fail. The calling servlet is permitted to continue running after the call, but it must not make any attempt to communicate back to the original client.

**Inserting a Request**
The previous section looked at when the complete request is passed on to another servlet for processing. Although this procedure is very useful, it is restrictive - only one servlet can produce the final output. There are times where it would be nice if servlets could cooperatively work together to create the output. For example, you could have one servlet conduct or control the flow calling other servlets to insert content as and when.

Fortunately the API supports this very type of use. The previous section introduced the forward(...) method from the RequestDispatcher interface. To insert content into the current output stream, the insert(...) method is available. This method operates in exactly the same way as forward(...) in that it accepts a ServletRequest and ServletResponse as parameters. Only this time, the called servlet is not meant to set any headers. If it does (for example, any cookie setting), then it is not guaranteed to be successful. As before, data can be passed to the servlet using the ServletRequest interface that was shown in the previous section.

Listing 8 shows an example of the insert(...) method. As before, it retrieves a reference to the RequestDispatcher of the servlet we wish to use. We create some output from the original servlet and then call the secondary servlet 10 times so it can insert its output to the original client request.

There is no limit to the number of times or number of servlets you can ask to insert output for. Using this technique, you have the ability to create sophisticated servlet systems that can all work together closely, thus reducing the total number of servlets that the server must develop and handle.

**Summary**
This article introduced the features that are available to you for sharing data between servlets. This functionality, as well as the ability to have servlets share the responsibility of processing the client request by collaborating on the output generation, was a major step forward in servlet ideology.

As the name says it, it is communication between servlets. Servlets talking to each other. [There are many ways to communicate between servlets, including

* Request Dispatching
* HTTP Redirect
* Servlet Chaining
* HTTP request (using sockets or the URLConnection class)
* Shared session, request, or application objects (beans)
* Direct method invocation (deprecated)
* Shared static or instance variables (deprecated)

Search the FAQ, especially topic [Message Passing (including Request Dispatching)](http://www.jguru.com/jguru/faq/faqindex.jsp?title=FAQ+Entries+in+Topic+Java%3AAPI%3AServlets%3AMessage+Passing+%28including+Request+Dispatching%29&topic=52022) for information on each of these techniques. -Alex]

Basically interServlet communication is acheived through servlet chaining. Which is a process in which you pass the output of one servlet as the input to other. These servlets should be running in the same server.

e.g. ServletContext.getRequestDispatcher(HttpRequest, HttpResponse).forward("NextServlet") ; You can pass in the current request and response object from the latest form submission to the next servlet/JSP. You can modify these objects and pass them so that the next servlet/JSP can use the results of this servlet.

There are some Servlet engine specific configurations for servlet chaining.

Servlets can also call public functions of other servlets running in the same server. This can be done by obtaining a handle to the desired servlet through the ServletContext Object by passing it the servlet name ( this object can return any servlets running in the server). And then calling the function on the returned Servlet object.

e.g. TestServlet test= (TestServlet)getServletConfig().getServletContext().getServlet("OtherServlet"); otherServletDetails= Test.getServletDetails();

You must be careful when you call another servlet's methods. If the servlet that you want to call implements the SingleThreadModel interface, your call could conflict with the servlet's single threaded nature. (The server cannot intervene and make sure your call happens when the servlet is not interacting with another client.) In this case, your servlet should make an HTTP request to the other servlet instead of direct calls.

Servlets could also invoke other servlets programmatically by sending an HTTP request. This could be done by opening a URL connection to the desired Servlet.

Q: - What is functionality of "RequestDispatcher" object?

Ans: -

## javax.servlet Interface RequestDispatcher

public interface **RequestDispatcher**

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. The servlet container creates the RequestDispatcher object, which is used as a wrapper around a server resource located at a particular path or given by a particular name.

This interface is intended to wrap servlets, but a servlet container can create RequestDispatcher objects to wrap any type of resource.

**See Also:**

[ServletContext.getRequestDispatcher(java.lang.String)](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/ServletContext.html#getRequestDispatcher%28java.lang.String%29), [ServletContext.getNamedDispatcher(java.lang.String)](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/ServletContext.html#getNamedDispatcher%28java.lang.String%29), [ServletRequest.getRequestDispatcher(java.lang.String)](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/ServletRequest.html#getRequestDispatcher%28java.lang.String%29)

|  |
| --- |
| **Method Summary** |
|  void | [**forward**](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/RequestDispatcher.html#forward%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse%29)([ServletRequest](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/ServletRequest.html) request, [ServletResponse](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/ServletResponse.html) response)           Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server. |
|  void | [**include**](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/RequestDispatcher.html#include%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse%29)([ServletRequest](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/ServletRequest.html) request, [ServletResponse](http://java.sun.com/webservices/docs/1.6/api/javax/servlet/ServletResponse.html) response)           Includes the content of a resource (servlet, JSP page, HTML file) in the response. |

public void **forward**([ServletRequest](http://kickjava.com/2992.htm) request,

 [ServletResponse](http://kickjava.com/2980.htm) response)

 throws [ServletException](http://kickjava.com/3054.htm),

 [IOException](http://kickjava.com/2848.htm)

**See Also:**

[IllegalStateException](http://kickjava.com/2792.htm), [ServletResponseWrapper](http://kickjava.com/2981.htm), [ServletRequestWrapper](http://kickjava.com/2993.htm)

public void doPost  ( HttpServletRequest req, HttpServletResponse res )
       throws ServletException, IOException  {
     HttpSession session = req.getSession ( false ) ;
     if  ( session == null )   {
       res.sendRedirect ( "http://localhost:8080/error.html" ) ;
      }

 ...

      String url="/jsp/KickJava.jsp";
       ServletContext sc = getServletContext (  ) ;
       RequestDispatcher rd = sc.getRequestDispatcher ( url ) ;
       rd.forward ( req, res ) ;
  }

Q: - How do we share data using "getServletContext ()"?

Ans: -

Q: - Explain the concept of SSI?

Ans: -

**S**erver **S**ide **I**ncludes: - A technique that makes it much easier to maintain boilerplate text that appears in many different web pages.

SSI (Server Side Includes) are directives that are placed in HTML pages, and evaluated on the server while the pages are being served. They let you add dynamically generated content to an existing HTML page, without having to serve the entire page via a CGI program, or other dynamic technology.

The decision of when to use SSI, and when to have your page entirely generated by some program, is usually a matter of how much of the page is static, and how much needs to be recalculated every time the page is served. SSI is a great way to add small pieces of information, such as the current time. But if a majority of your page is being generated at the time that it is served, you need to look for some other solution.

### Most common directives

|  |  |  |  |
| --- | --- | --- | --- |
| **Directive** | **Parameters** | **Description** | **Example** |
| include | file, direct or virtual | This is probably the most used SSI directive, allowing the content of one document to be included in another. The file or virtual parameters specify the file ([HTML](http://en.wikipedia.org/wiki/HTML) page, text file, script, etc) to be included. The file parameter defines the included file as relative to the document path; the virtual parameter defines the included file as relative to the document root. | <!--#include virtual="header.html"--> |
| exec | cgi or cmd | This directive executes a program, script, or shell command on the server. the cmd parameter specifies a server-side command; the cgi parameter specifies the path to a CGI script. The PATH\_INFO and QUERY\_STRING of the current SSI script will be passed to the CGI script. "include virtual" should be used instead of "exec cgi". | <!--#exec cgi="/cgi-bin/foo.cgi"-->or<!--#exec cmd="ls -l"--> |
| echo | var | This directive displays the contents of a specified [HTTP](http://en.wikipedia.org/wiki/HTTP) environment variable. Variables include HTTP\_USER\_AGENT, LAST\_MODIFIED, and HTTP\_ACCEPT. | <!--#echo var="REMOTE\_ADDR" --> |
| config | timefmt, sizefmt, or errmsg | This directive configures the display formats for the date, time, filesize, and error message (returned when an SSI command fails). | <!--#config timefmt="%y %m %d" -->or<!--#config sizefmt="bytes" -->or<!--#config errmsg="SSI command failed!" --> |
| flastmod or fsize | file or virtual | These directives display the date when the specified document was last modified, or the specified document's size. The file or virtual parameters specify the document to use. The file parameter defines the document as relative to the document path; the virtual parameter defines the document as relative to the document root. | <!--#flastmod virtual="index.html"-->or<!--#fsize file="script.pl"--> |
| printenv |   | This directive outputs a list of all variables and their values, including environmental and user-defined variables. It has no attributes. | <!--#printenv --> |

### Advanced directives

|  |  |  |  |
| --- | --- | --- | --- |
| **Directive** | **Parameters** | **Description** | **Example** |
| if | expr | Used for condition tests that may determine and generate multiple logical pages from one single physical page. | <!--#if expr="${Sec\_Nav}" --><!--#include virtual="secondary\_nav.txt" --><!--#endif --> |
| elif | expr | Serves the same purpose as further conditioning in programming languages. | <!--#if expr="${Sec\_Nav}" --><!--#include virtual="secondary\_nav.txt" --><!--#elif expr="${Pri\_Nav}}"><!--#include virtual="primary\_nav.txt" --><!--#endif --> |
| else |  | If none of the if and elif directive catches the present condition, things in here should happen. | <!--#if expr="${Sec\_Nav}" --><!--#include virtual="secondary\_nav.txt" --><!--#else --><!--#include virtual="article.txt" --><!--#endif --> |
| endif |  | Termination of a conditional construct. | See above for example. |

## Client Side Includes

**Client Side Includes** are HTML includes achieved on the [client-side](http://en.wikipedia.org/wiki/Client-side) through use of frames, [IFrames](http://en.wikipedia.org/wiki/IFrame), [JavaScript](http://en.wikipedia.org/wiki/JavaScript), or JavaScript with [Ajax](http://en.wikipedia.org/wiki/Ajax_%28programming%29) requests. These methods suffer from shortcomings not present in server-side includes: they rely on the client's support of their respective technologies and, in the case of frames and iframes, are less accessible.

Q: - What are filters in JAVA?

Ans: -

# The Essentials of Filters

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |   |

|  |  |
| --- | --- |
|  |  |

 |

The Java Servlet specification version 2.3 introduces a new component type, called a filter. A *filter* dynamically intercepts requests and responses to transform or use the information contained in the requests or responses. Filters typically do not themselves create responses, but instead provide universal functions that can be "attached" to any type of servlet or JSP page. Filters are important for a number of reasons. First, they provide the ability to encapsulate recurring tasks in reusable units. Organized developers are constantly on the lookout for ways to modularize their code. Modular code is more manageable and documentable, is easier to debug, and if done well, can be reused in another setting. Second, filters can be used to transform the response from a servlet or a JSP page. A common task for the web application is to format data sent back to the client. Increasingly the clients require formats (for example, WML) other than just HTML. To accommodate these clients, there is usually a strong component of transformation or filtering in a fully featured web application. Many servlet and JSP containers have introduced proprietary filter mechanisms, resulting in a gain for the developer that deploys on that container, but reducing the reusability of such code. With the introduction of filters as part of the Java Servlet specification, developers now have the opportunity to write reusable transformation components that are portable across containers. Filters can perform many different types of functions. We'll discuss examples of the italicized items in this paper: * Authentication-Blocking requests based on user identity.
* Logging and auditing-Tracking users of a web application.
* Image conversion-Scaling maps, and so on.
* Data compression-Making downloads smaller.
* Localization-Targeting the request and response to a particular locale.
* XSL/T transformations of XML content-Targeting web application responses to more that one type of client.

These are just a few of the applications of filters. There are many more, such as encryption, tokenizing, triggering resource access events, mime-type chaining, and caching. In this paper we'll first discuss how to program filters to perform the following types of tasks: * Querying the request and acting accordingly
* Blocking the request and response pair from passing any further.
* Modifying the request headers and data. You do this by providing a customized version of the request.
* Modifying the response headers and data. You do this by providing a customized version of the response.

We'll outline the filter API, and describe how to develop customized requests and responses. Programming the filter is only half the job of using filters-you also need to configure how they are mapped to servlets when the application is deployed in a web container. This decoupling of programming and configuration is a prime benefit of the filter mechanism: * You don't have to recompile anything to change the input or output of your web application. You just edit a text file or use a tool to change the configuration. For example, adding compression to a PDF download is just a matter of mapping a compression filter to the download servlet.
* You can experiment with filters easily because they are so easy to configure.

The last section of this paper shows how to use the very flexible filter configuration mechanism. Once you have read this paper, you will be armed with the knowledge to implement your own filters and have a handy bag of tricks based on some common filter types. Programming Filters The filter API is defined by the Filter, FilterChain, and FilterConfig interfaces in the javax.servlet package. You define a filter by implementing the [Filter](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/Filter.html) interface. A filter chain, passed to a filter by the container, provides a mechanism for invoking a series of filters. A filter config contains initialization data. The most important method in the Filter interface is the doFilter method, which is the heart of the filter. This method usually performs some of the following actions: * Examines the request headers
* Customizes the request object if it wishes to modify request headers or data or block the request entirely
* Customizes the response object if it wishes to modify response headers or data
* Invokes the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target servlet, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. It invokes the next entity by calling the doFilter method on the chain object (passing in the request and response it was called with, or the wrapped versions it may have created). Alternatively, it can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
* Examines response headers after it has invoked the next filter in the chain
* Throws an exception to indicate an error in processing

In addition to doFilter, you must implement the init and destroy methods. The init method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter you retrieve them from the FilterConfig object passed to init. Example: Logging Servlet Access Now that you know what the main elements of the filter API are, let's take a look at a very simple filter that does not block requests, transform responses, or anything fancy-a good place to start learning the basic concepts of the API. Consider web sites that track the number of users. To add this capability to an existing web application without changing any servlets you could use a logging filter. HitCounterFilter increments and logs the value of a counter when a servlet is accessed. In the doFilter method, HitCounterFilter first retrieves the servlet context from the filter configuration object so that it can access the counter, which is stored as a context attribute. After the filter retrieves, increments, and writes the counter to a log, it invokes doFilter on the filter chain object passed into the original doFilter method. The elided code is discussed in [Programming Customized Requests and Responses](http://java.sun.com/products/servlet/Filters.html#72674).

|  |
| --- |
| public final class HitCounterFilter implements Filter {   private FilterConfig filterConfig = null;   public void init(FilterConfig filterConfig)       throws ServletException {      this.filterConfig = filterConfig;   }   public void destroy() {      this.filterConfig = null;   }   public void doFilter(ServletRequest request,      ServletResponse response, FilterChain chain)       throws IOException, ServletException {      if (filterConfig == null)         return;      StringWriter sw = new StringWriter();      PrintWriter writer = new PrintWriter(sw);      Counter counter = (Counter)filterConfig.         getServletContext().         getAttribute("hitCounter");      writer.println();      writer.println("===============");      writer.println("The number of hits is: " +         counter.incCounter());      writer.println("===============");      // Log the resulting string      writer.flush();      filterConfig.getServletContext().         log(sw.getBuffer().toString());      ...      chain.doFilter(request, wrapper);      ...   }}  |

Example: Modifying the Request Character Encoding Currently, many browsers do not send character encoding information in the Content-Type header of an HTTP request. If an encoding has not been specified by the client request, the container uses a default encoding to parse request parameters. If the client hasn't set character encoding and the request parameters are encoded with a different encoding than the default, the parameters will be parsed incorrectly. You can use the method setCharacterEncoding in the ServletRequest interface to set the encoding. Since this method must be called prior to parsing any post data or reading any input from the request, this function is a prime application for filters. Such a filter is contained in the examples distributed with the [Tomcat 4.0](http://jakarta.apache.org/tomcat) web container. The filter sets the character encoding from a filter initialization parameter. This filter could easily be extended to set the encoding based on characteristics of the incoming request, such as the values of the Accept-Language and User-Agent headers, or a value saved in the current user's session.

|  |
| --- |
| public void doFilter(ServletRequest request,    ServletResponse response, FilterChain chain) throws   IOException, ServletException {   String encoding = selectEncoding(request);   if (encoding != null)      request.setCharacterEncoding(encoding);   chain.doFilter(request, response);}public void init(FilterConfig filterConfig) throws   ServletException {   this.filterConfig = filterConfig;   this.encoding = filterConfig.getInitParameter("encoding");}protected String selectEncoding(ServletRequest request) {   return (this.encoding);}  |

Programming Customized Requests and Responses So far we have looked at some simple examples. Now let's get a bit more sophisticated and look at a filter that modifies the request from or response back to the client. There are many ways for a filter to modify a request or response. For example, a filter could add an attribute to the request or it could insert data in or otherwise transform the response. A filter that modifies a response must usually capture the response before it is returned to the client. The way to do this is to pass the servlet that generates the response a stand-in stream. The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet's response. In order to pass this stand-in stream to the servlet, the filter creates a response "wrapper" that overrides the getWriter or getOutputStream method to return this stand-in stream. The wrapper is passed to the doFilter method of the filter chain. Wrapper methods default to calling through to the wrapped request or response object. This approach follows the well-known Wrapper or Decorator pattern described in Design Patterns, Elements of Reusable Object-Oriented Software. The following sections describe how the hit counter filter described earlier and other types of filters use wrappers. To override request methods, you wrap the request in an object that extends [ServletRequestWrapper](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/ServletRequestWrapper.html) or [HttpServletRequestWrapper](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpServletRequestWrapper.html). To override response methods, you wrap the response in an object that extends [ServletResponseWrapper](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/ServletResponseWrapper.html) or [HttpServletResponseWrapper](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpServletResponseWrapper.html). The hit counter filter described in [Programming Filters](http://java.sun.com/products/servlet/Filters.html#70176) inserts the value of the counter into the response. The elided code from HitCounterFilter is:

|  |
| --- |
| PrintWriter out = response.getWriter();CharResponseWrapper wrapper = new CharResponseWrapper(   (HttpServletResponse)response);chain.doFilter(request, wrapper);if(wrapper.getContentType().equals("text/html")) {   CharArrayWriter caw = new CharArrayWriter();   caw.write(wrapper.toString().substring(0,      wrapper.toString().indexOf("</body>")-1));   caw.write("<p>\nYou are visitor number    <font color='red'>" + counter.getCounter() + "</font>");   caw.write("\n</body></html>");   response.setContentLength(caw.toString().length());   out.write(caw.toString());} else    out.write(wrapper.toString());out.close();  |

HitCounterFilter wraps the response in a CharResponseWrapper. CharResponseWrapper overrides the getWriter method to return a stand-in stream into which the servlet at the end of the filter chain writes its response. When chain.doFilter returns, HitCounterFilter retrieves the servlet's response from PrintWriter and writes it to a buffer if it is an HTML response. The filter inserts the value of the counter into the buffer, resets the content length header of the response, and finally writes the contents of the buffer to the response stream.

|  |
| --- |
| public class CharResponseWrapper extends   HttpServletResponseWrapper {   private CharArrayWriter output;   public String toString() {      return output.toString();   }   public CharResponseWrapper(HttpServletResponse response){      super(response);      output = new CharArrayWriter();   }   public PrintWriter getWriter(){      return new PrintWriter(output);   }}  |

Example: Compressing the Response Another example of a filter that modifies the response is the compression filter contained in the examples distributed with the Tomcat servlet engine. Although high-speed Internet connections are becoming more commonplace, there is still a need to use bandwidth effectively. A compression filter is handy because you can attach it to any servlet to reduce the size of a response. Like the hit counter filter, the compression filter creates a stand-in stream, in this case CompressionResponseStream, for the servlet to write to and wraps the response passed to the servlet. The filter creates the wrapper and stand-in stream only if the client can accept a compressed response. The servlet writes its response to the compression stream it retrieves from the wrapper. CompressionResponseStream overrides the write method to write response data to a GZIPOutputStream once the data is larger than a threshold passed as an initialization parameter to the filter:

|  |
| --- |
| public void write(int b) throws IOException {   ...   if ((bufferCount >= buffer.length) ||       (count>=compressionThreshold)) {      compressionThresholdReached = true;   }   if (compressionThresholdReached) {      writeToGZip(b);   } else {      buffer[bufferCount++] = (byte) b;      count++;   }}  |

Example: Transforming the Response The last filter we'll discuss is an XSLT filter. XSLT is a language for transforming XML data. You can use XSLT to transform an XML document to end user-oriented formats such as HTML or PDF, or to another XML format. Some example applications include: * Converting an XML document in a format required by one company to the format required by another company.
* Customizing the look and feel of a web page based on user preferences.
* Enabling the same web application to respond to different types of clients, for example, WML phones and cHTML phones, by looking at the User-Agent header and choosing a stylesheet.

Consider a web service that responds to requests for product inventory. The following XML document is an example of such a response:

|  |
| --- |
| <book>   <isbn>123</isbn>   <title>Web Servers for Fun and Profit</title>   <quantity>10</quantity>   <price>$17.95</price></book>  |

The following XSL stylesheets render this XML document as a user-oriented description of the inventory in HTML format and as a machine-oriented version in XML format.

|  |
| --- |
| <?xml version="1.0" ?> <xsl:stylesheet version="1.0"    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:output method="html" omit-xml-declaration="yes"/><xsl:template match="/">   <xsl:apply-templates/></xsl:template>**<xsl:template match="book">**<html><body>There are <xsl:value-of select="quantity"/> copies of <i><xsl:value-of select="title"/></i> available.</body></html></xsl:template></xsl:stylesheet> <?xml version="1.0" ?> <xsl:stylesheet version="1.0"    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:output method="xml" omit-xml-declaration="no"/><xsl:template match="/">   <xsl:apply-templates/></xsl:template><xsl:template match="book"><xsl:element name="book"><xsl:attribute name="isbn"><xsl:value-of select="isbn"/></xsl:attribute><xsl:element name="quantity"><xsl:value-of select="quantity"/></xsl:element></xsl:element></xsl:template></xsl:stylesheet>  |

The following XSLT filter uses the stylesheets to transform the response depending on the value of a request parameter. The filter sets content type of the response according to the request parameter. The response is then wrapped in a CharResponseWrapper and passed to the doFilter method of the filter chain. The last element in the filter chain is a servlet that returns the inventory response described earlier. When doFilter returns, the filter retrieves the response data from the wrapper and transforms it using the stylesheet.

|  |
| --- |
| public void doFilter(ServletRequest request,    ServletResponse response, FilterChain chain)   throws IOException, ServletException {   String contentType;   String styleSheet;   String type = request.getParameter("type");   if (type == null || type.equals("")) {      contentType = "text/html";      styleSheet = "/xml/html.xsl";   } else {      if (type.equals("xml")) {         contentType = "text/plain";         styleSheet = "/xml/xml.xsl";      } else {         contentType = "text/html";         styleSheet = "/xml/html.xsl";      }   }   response.setContentType(contentType);   String stylepath=filterConfig.getServletContext().      getRealPath(styleSheet);   Source styleSource = new StreamSource(stylePath);   PrintWriter out = response.getWriter();   CharResponseWrapper responseWrapper =       new CharResponseWrapper(         (HttpServletResponse)response);   chain.doFilter(request, wrapper);   // Get response from servlet   StringReader sr = new StringReader(      new String(wrapper.getData()));   Source xmlSource = new StreamSource((Reader)sr);   try {      TransformerFactory transformerFactory =         TransformerFactory.newInstance();      Transformer transformer = transformerFactory.         newTransformer(styleSource);      CharArrayWriter caw = new CharArrayWriter();      StreamResult result = new StreamResult(caw);      transformer.transform(xmlSource, result);      response.setContentLength(caw.toString().length());      out.write(caw.toString());   } catch(Exception ex) {      out.println(ex.toString());      out.write(wrapper.toString());   }}  |

Specifying Filter Configuration Now that we have seen how to program a filter, the last step is to specify how to apply it to a web component or a set of web components. To map a filter to a servlet you: * Declare the filter using the <filter> element in the web application deployment descriptor. This element creates a name for the filter and declares the filter's implementation class and initialization parameters.
* Map the filter to a servlet by defining a <filter-mapping> element in the deployment descriptor. This element maps a filter name to a servlet by name or by URL pattern.

The following elements show how to specify the elements needed for the compression filter. To define the compression filter you provide a name for the filter, the class that implements the filter, and name and value of the threshold initialization parameter.

|  |
| --- |
| <filter>   <filter-name>Compression Filter</filter-name>   <filter-class>CompressionFilter</filter-class>   <init-param>      <param-name>compressionThreshold</param-name>      <param-value>10</param-value>   </init-param></filter>  |

The filter-mapping element maps the compression filter to the servlet CompressionTest. The mapping could also have specified the URL pattern /CompressionTest. Note that the filter, filter-mapping, servlet, and servlet-mapping elements must appear in the web application deployment descriptor in that order.

|  |
| --- |
| <filter-mapping>   <filter-name>Compression Filter</filter-name>   <servlet-name>CompressionTest</servlet-name></filter-mapping><servlet>   <servlet-name>CompressionTest</servlet-name>   <servlet-class>CompressionTest</servlet-class></servlet><servlet-mapping>   <servlet-name>CompressionTest</servlet-name>   <url-pattern>/CompressionTest</url-pattern></servlet-mapping>  |

Note that this mapping causes the filter to be called for all requests to the CompressionTest servlet and to any servlet JSP or static content mapped to the URL pattern /CompressionTest. If you want to log every request to a web application, you would map the hit counter filter to the URL pattern /\*. Here's the deployment descriptor distributed with the examples:

|  |
| --- |
| ?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>   <filter>      <filter-name>XSLTFilter</filter-name>      <filter-class>XSLTFilter</filter-class>   </filter>   <filter>      <filter-name>HitCounterFilter</filter-name>      <filter-class>HitCounterFilter</filter-class>   </filter>   <filter-mapping>      <filter-name>HitCounterFilter</filter-name>      <url-pattern>/\*</url-pattern>   </filter-mapping>    <filter-mapping>      <filter-name>XSLTFilter</filter-name>      <servlet-name>FilteredFileServlet</servlet-name>   </filter-mapping>    <servlet>      <servlet-name>FilteredFileServlet</servlet-name>      <servlet-class>FileServlet</servlet-class>   </servlet>   <servlet-mapping>      <servlet-name>FilteredFileServlet</servlet-name>      <url-pattern>/ffs</url-pattern>   </servlet-mapping></web-app>  |

As you can see, you can map a filter to one or more servlets and you can map more than one filter to a servlet. This is illustrated in [Figure 1](http://java.sun.com/products/servlet/Filters.html#73808), where filter F1is mapped to servlets S1, S2, and S3, filter F2 is mapped to servlet S2, and filter F3 is mapped to servlets S1 and S2. http://java.sun.com/products/servlet/images/chain.gif**Figure 1 Filter to Servlet Mapping** Recall that a filter chain is one of the objects passed to the doFilter method of a filter. This chain is formed indirectly via filter mappings. The order of the filters in the chain is the same as the order that filter mappings appear in the web application deployment descriptor. When a URL is mapped to servlet S1, the web container invokes the doFilter method of F1. The doFilter method of each filter in S1's filter chain is invoked by the preceding filter in the chain via the chain.doFilter method. Since servlet S1's filter chain contains filters F1 and F3, F1's call to chain.doFilter invokes the doFilter method of filter F3. When F3's doFilter method completes, control returns to F1's doFilter method. The deployment descriptor just discussed puts the hit counter and XSLT filter in the filter chain of FilteredFileServlet. The hit counter filter logs access whenever FilteredFileServlet is invoked, but inserts the value of the counter into the response after the XSLT transformation only if the response type is HTML: http://java.sun.com/products/servlet/images/filterHTML.gifhttp://java.sun.com/products/servlet/images/filterXML.gifConclusion The filter mechanism provides a way to encapsulate common functionality in a component that can reused in many different contexts. Filters are easy to write and configure as well as being portable and reusable. In summary, filters are an essential element in a web developer's toolkit. Acknowledgments The character encoding filter was developed by Craig McClanahan of Sun Microsystems, the compression filter was developed by Amy Roh of Sun Microsystems, and the XSLT filter was provided by Alan Canon of National Processing Company. Resources You can get the character encoding and compression filters by downloading [Tomcat 4.0](http://jakarta.apache.org/tomcat). The character encoding filter is located in the TOMCAT\_HOME/webapps/examples/WEB-INF/classes/filters directory. The compression filter is located in the TOMCAT\_HOME/webapps/examples/WEB-INF/classes/compressionFilters directory. You can download a zip file containing the other filters described in this paper from [here](http://java.sun.com/products/servlet/filters.zip). To compile and run the XSLT filter, you'll need to obtain an XML parser and transformation engine from <http://java.sun.com/xml/downloads/> For further information about the technologies described in this paper, see the following resources: * [http://java.sun.com/products/servlet](http://java.sun.com/products/servlet/index.jsp)

* [http://www.w3.org/XML](http://www.w3.org/Style/XSL)

* <http://www.w3.org/Style/XSL>

* [http://java.sun.com/xml](http://java.sun.com/xml/index.jsp)
 |

Q: - Can you explain in short how do you go about implementing filters using Apache Tomcat?

Twist: - Explain step by step of how to implement filters?

Ans: -

Page 29

Q: - What's the difference between Authentication and authorization?

Ans: -

Implementing security for Web applications is a mandatory task for architects and Web application developers. In J2EE, the Web containers have support for built-in security mechanisms for their applications.

|  |
| --- |
|  |

There are two major components of Web application security: authentication and authorization. J2EE-based Web containers offer three types of authentication mechanisms: basic, form-based, and mutual authentication. Most Web applications use the form-based authentication mechanism, since it allows applications to customize the authentication user interface. Web containers implement authorization on Web resources of applications, using security roles defined in the deployment descriptor of the Web application.

There are three issues that software architects and software developers come across using the form-based authentication mechanism:

* How the form-based authentication works with other security realms, such as database and LDAP (this is necessary because most organizations may already have authentication information in database or LDAP form).
* How to add or delete authorization roles declared in the deployment descriptor (web.xml) of the Web application.
* Web containers enforce authorization at Web resource level; however, an application needs to enforce authorization at its functionality level within a single Web resource.

Despite the fact that there is plenty of documentation and many examples available for form-based authentication, none of them clarifies these issues. Hence, most applications have implemented security in their own way.

This article demonstrates how the form-based authentication mechanism works with other security realms, especially the database. It also explains how Web containers use the security roles to implement authorization, and how applications can extend these security roles to protect functionalities within single resource.

### Form-Based Authentication

Form-based authentication lets developers customize the authentication user interface. The login-config section of web.xml defines the type of authentication mechanism, and the URIs to login and error pages.

<login-config>

<auth-method>FORM</auth-method>

<form-login-config>

<form-login-page>/login.jsp</form-login-page>

<form-error-page>/fail\_login.html</form-error-page>

</form-login-config>

</login-config>

The login form must contain fields for entering username and password. These fields must be named j\_username and j\_password, respectively. This form should post these values to j\_security\_check logical name.

Here is an example showing how the form should be coded into an HTML page:

<form method="POST" action="j\_security\_check">

<input type="text" name="j\_username">

<input type="password" name="j\_password">

</form>

This form of authentication, which uses base64 encoding, can expose username and password unless all connections are over SSL.

The Web containers activate the authentication mechanism that has been configured for that resource, when protected Web resources are accessed.

Web containers perform the following steps to implement security of a Web application.

1. Determines whether the user has been authenticated when the protected Web resources are accessed.
2. If the user has not been authenticated yet, requests that the user provide security credentials by redirecting to the login page defined in the deployment descriptor.
3. Validates the user's credentials against the security realm configured for the container.
4. Determines whether the authenticated user is authorized to access Web resources defined in the deployment descriptor (web.xml).

In the deployment descriptor of a Web application, form-based authentication does not specify the security realm, as the basic authentication mechanism does. In other words, it does not explicitly declare the type of security realm it should use to authenticate a user. This causes some confusion as to what security realm it actually uses to authenticate the user.

Web containers perform the following steps to validate a user:

1. Determines the configured security realm for the container.
2. Uses that security realm for authentication.

Since database and LDAP provide greater flexibility in maintaining information, most organizations may want this persistence for security authentication and authorization information.

Many Web containers have support for different types of security realms: database, LDAP, and custom realm.

For example, in the Tomcat Web container, server.xml configures database as its security realm.

<Realm

className="org.apache.catalina.realm.JDBCRealm"

debug="99"

driverName="oracle.jdbc.driver.OracleDriver"

connectionURL="jdbc:oracle:thin:@{IPAddress}:{Port}:{Servicename}"

 connectionName="{DB Username}"

 connectionPassword="{Password}"

 userTable="users"

userNameCol="username"

 userCredCol="password"

userRoleTable="user\_roles"

roleNameCol="rolename"

/>

The <Realm> tag of Tomcat's server.xml defines the type of security realm that the container should use to authenticate a user. Note that the container uses this realm for Web applications whose authentication mechanism is form-based.

**Note:** Please see the sections on deploying database security realm support for Tomcat and WebLogic below.

|  |  |
| --- | --- |
|  |  |

#### Authorization

Once the user has been authenticated, the container retrieves security roles for the authenticated user and checks to see if the user belongs to one of the roles defined in the <auth-constraint> tag of the deployment descriptor. The Web container displays an error message if the user does not belong to one of roles.

The <security-constraint> tag of the deployment descriptor (web.xml) defines protected Web resources and a list of roles that have access to these Web resources.

<security-constraint>

<web-resource-collection>

<web-resource-name>AdminPages</web-resource-name>

<description> accessible by authorised users </description>

<url-pattern>/admin/\*</url-pattern>

<http-method>GET</http-method>

</web-resource-collection>

<auth-constraint>

<description>These are the roles who have access</description>

<role-name>manager</role-name>

</auth-constraint>

</security-constraint>

Web containers enforce authorization at a page level. However, commercial applications may also want to enforce authorization on functionalities within a single page. This may require that some additional new application-specific security roles be defined in the application. To control access to functionalities, the application needs to understand the notion of privileges for a role. The Web container specification does not address privileges.

Since authorization roles are dynamic, developers often get confused as to whether these security roles need to be added to the deployment descriptor. Web containers just need one role defined in the deployment descriptor for the application to take advantage of security support. So, the application can define one high-level role and assign all users to this role. This allows all users in this role to have access to the Web resources.

In addition, the application can define additional roles to enforce authorization on low-level functionalities within a single Web resource. These low-level security roles do not need to be in the deployment descriptor, since the application has been configured with a high-level security role that contains all of the users in the application. This allows Web applications to take advantage of container authorization support, and to implement application-specific authorization.

For example, you can define an administrator role for all of the users at a high level in the deployment descriptor to protect administration Web resources. This allows all of the users in the administrator role to access the administration pages. To control other functionalities within the administration pages, you can create new roles, such as sysadmin or appadmin, within the application.

The application can extend these security roles to have privileges. Then, the application can use these privileges to control access to its functionalities.

Despite the fact that application-specific security roles are not defined in the deployment descriptors, these roles can still be used in the isUserInRole method to determine whether the user in these security roles.

#### Advantages

* Web applications do not need to implement the authentication mechanism by themselves. It is simply a matter of configuring the Web application.
* Web applications can implement programmatic security using getRemoteUser, IsUserInRole, and getUserPrincipal methods.
* The Web container is capable of propagating authentication information to EJB containers.

### Configuring the Database Security Realm in Tomcat

1. Create User Table

This table needs the username and password columns.

create table users (username varchar(20) not null, password(20) not null)

1. Create Role Table

This table maintains the list of roles in application. It just needs role name column in the table.

create table roles (rolename varchar(20) not null)

1. Create User-Role Association Table

This table maintains association between a user and roles. A user can belong to one or more roles.

create table user\_roles (username varchar(20) not null, rolename varchar(20) not null)

1. Insert data into tables

insert into users values('user1', 'password')

insert into role values('manager')
insert into user\_roles values('user1', 'manager')

1. Create User Table

This table needs the username and password columns.

create table users (username varchar(20) not null, password(20) not null)

1. Create Role Table

This table maintains the list of roles in the application. It just needs the role name column in the table.

create table roles (rolename varchar(20) not null)

1. Create User-Role Association Table

This table maintains association between a user and roles. A user can belong to one or more roles.

create table user\_roles (username varchar(20) not null, rolename varchar(20) not null)

1. Insert data into tables

insert into users values('user1', 'password')

insert into role values('manager')

insert into user\_roles values('user1', 'manager')

1. Configure Tomcat (This example uses the Oracle thin driver) by copying this information in server.xml in the {tomcat}\conf\ folder. (Tomcat uses memory realm as default realm. Comment the memory realm.)
2. <Realm
3. className="org.apache.catalina.realm.JDBCRealm"
4. debug="99"
5. driverName="oracle.jdbc.driver.OracleDriver"
6. connectionURL="jdbc:oracle:thin:@{IP address}:{Port}:{Servicename}"
7. connectionName="{DB Username}"
8. connectionPassword="{Password}"
9. userTable="users"
10. userNameCol="username"
11. userCredCol="password"
12. userRoleTable="user\_roles"
13. roleNameCol="rolename"

/>

Replace the following values with environment values

{IP Address} - IP address of database server

{Port} - Port number

{Servicename} - Service name

{DB Username} - database login

{Password} - password to database login

**Note**: You need to have the JDBC driver for your database. (The Oracle thin driver is used in this example.)

1. Copy the Oracle thin driver JAR file or JDBC driver for your database (oracle\_thin\_driver.jar) into the {tomcat\_home}/server/lib directory.
2. Configure the Web application deployment descriptor with security constraints.
3. <security-constraint>
4. <web-resource-collection>
5. <web-resource-name>Protected Area</web-resource-name>
6. <!-- Define the context-relative URL(s) to be protected -->
7. <url-pattern>/\*</url-pattern>
8. <http-method>DELETE</http-method>
9. <http-method>GET</http-method>
10. <http-method>POST</http-method>
11. <http-method>PUT</http-method>
12. </web-resource-collection>
13. <auth-constraint>
14. <role-name>manager</role-name>
15. </auth-constraint>
16. <user-data-constraint><transport-guarantee>
17. NONE</transport-guarantee></user-data-constraint>
18. </security-constraint>
19. <!-- Default login configuration uses form-based authentication -->
20. <login-config>
21. <auth-method>FORM</auth-method>
22. <realm-name>Example Form-Based Authentication Area</realm-name>
23. <form-login-config>
24. <form-login-page>/jsp/login.jsp</form-login-page>
25. <form-error-page>/jsp/error.jsp</form-error-page>
26. </form-login-config>

</login-config>

Note that the <role-name> value of this tag in <auth-constraint> should be one of the role names in the user-roles table.

### Deploying The Sample Program Into Tomcat

1. Configure Tomcat using the instructions defined in the previous section.
2. Download the [security-form-based.war](http://www.onjava.com/onjava/2002/06/12/examples/security-form-based.war) file and copy it into Tomcat's webapps directory.
3. Start the Tomcat server.
4. Open a Web browser and go to http://{ip address:port no}/security-form-based/protected/index.jsp.
5. Type in the username and the password (user1, password).

### Configuring the Database Security Realm in WebLogic

There is [very good documentation](http://e-docs.bea.com/wls/docs61/adminguide/cnfgsec.html%20-%201052867) on configuring the database security realm. Please refer to this link.

Configure your Web application deployment descriptor, similar to the one shown in the configuration steps for Tomcat.

One difference between Tomcat and WebLogic deployment descriptors is that the deployment descriptor for WebLogic requires the following section, while Tomcat does not require this section.

<security-role>

<description>

Manager security role

</description>

<role-name>

manager

</role-name>

</security-role>

### Conclusion

This article has provided in-depth understanding of the form-based mechanism and how it works with database security realms to handle authentication. Web applications can take advantage of the form-based mechanism to protect their resources while at the same time using legacy security realms for authentication information.

In addition, this article has described the level of authorization support provided by J2EE Web containers and how you can define new security roles without modifying the deployment descriptor of a Web application.

Q: - Explain in brief the directory structure of a web application?

Ans: -

# About the web application directory structure

The following figure shows the directory structure of a web application:



The application root directory, *web\_app* in the figure, functions as the document root for serving application files. The *web\_app* directory is often the WAR filename. This directory includes JSPs and HTML pages that you develop as part of a web application. For example, for a web application located at *jrun\_root*/servers/*jrun\_server*/*web\_app,* the default welcome file is typically located at *jrun\_root*/servers/*jrun\_server*/*web\_app*/index.html. *jrun\_server* is the directory name of a JRun server and *web\_app* is the name of a directory corresponding to the name of a web application hosted by the server.

Web applications must conform to the following directory structure:

|  |  |
| --- | --- |
| **Directory** | **Description** |
| Web\_app | Contains the WEB-INF directory and all files that must be accessible to the client request, such as JSPs, HTML pages, cascading style sheets, images, and JavaScript files. You can place these files directly in the web application root directory or in subdirectories that do not use the reserved name WEB-INF. |
| WEB-INF | (Required) Contains the classes and lib directories, the standard web application deployment descriptor (web.xml), and possibly a JRun-specific deployment descriptor (jrun-web.xml).This directory is not part of the public document tree of the application; that is, you cannot directly serve any file contained in this directory, or any of its subdirectories, to a client. |
| WEB-INF/classes | Contains servlets, other Java classes, and associated resources that are not packaged in Java ARchive (JAR) files. |
| WEB-INF/lib | Contains servlets, other Java classes, and associated resources that are contained within JAR or ZIP files. This subdirectory includes any JAR files that contain tag libraries. |
| WEB-INF/jsp | Contains the files generated by JRun when translating a JSP. |
| WEB-INF/sessions | Contains serialized JRun sessions used by the default session persistence mechanism. |

Enterprise applications conform to a similar directory structure as web applications. The application root directory, *web\_app*, is often the EAR filename. The enterprise application deployment descriptor, application.xml, is located in the *jrun\_root*/servers/*jrun\_server*/*web\_app*/META-INF directory.

Your web application is not limited to the directories and subdirectories listed above. You can add directories to the application for items such as Flash SWF files, HTML files, images, and other application resources. These directories constitute the public document tree for the web application for resources accessed directly by a client. For more information on adding directories to the application, see ["Adding directories"](http://livedocs.adobe.com/jrun/4/Getting_Started_with_JRun/develapps5.htm#1113196).

## The deployment descriptor (web.xml)

A web application is defined by the contents of the WEB-INF/web.xml file, which is also called the deployment descriptor. A **deployment descriptor** is an XML file that contains configuration information used by the application for execution on an application server. The Java Servlet specification defines the contents of web.xml, so it is not specific to JRun. All platforms that support J2EE-compliant web applications, recognize and interpret the contents of a web.xml file.

You use the web.xml file to define the following types of configuration and deployment information for a web application:

* Servlet initialization parameters
* Session configuration
* Servlet and JSP definitions
* Servlet and JSP URL mappings
* MIME-type mappings
* Welcome file list
* Error pages
* Filters
* Event handlers
* Tag libraries
* Security information
* Resources, such as data sources and JMS connection factories

In addition to containing information about a web application, a web.xml file can also contain reference information about EJBs accessed by the web application. The standard deployment descriptors for web applications and EJBs have common elements for environment entries, EJB references, resource references, and security roles.

In addition to the standard deployment descriptors, you can use the following JRun-specific deployment descriptor files:

* **WEB-INF/jrun-web.xml**  Contains web application elements that are specific to the JRun application server. This deployment descriptor lets you set a context-root for the web application, enable and disable automatic servlet compilation and reloading, set virtual paths, configure servlet session persistence and cookies, and set values for JNDI names.
* **SERVER-INF/default-web.xml**  Has the same structure as the standard web.xml file, but it applies settings to all the web applications in a JRun server. Contains default values for each web application web.xml file. This file includes definitions and mappings for global servlets, used internally by JRun.

You can edit the web.xml file using a standard text editor or an XML editor. JRun lets you specify additional elements in the jrun-web.xml and default-web.xml files. For a complete list of all properties in the standard web.xml file, see the Java Servlet 2.3 specification. For more information on JRun-specific deployment descriptors, see *JRun Assembly and Deployment Guide*.

Directory Structure

Develop your Web Application within a specified directory structure so that it can be archived and deployed on WebLogic Server or another J2EE-compliant server. All servlets, classes, static files, and other resources belonging to a Web Application are organized under a directory hierarchy. The root of this hierarchy defines the document root of your Web Application. All files under this root directory can be served to the client, except for files under the special directory WEB-INF, located under the root directory. The name of your Web Application is used to resolve requests for components of the Web Application.

Place private files in the WEB-INF directory, under the root directory. All files under WEB-INF are private, and are not served to a client.

**DefaultWebApp/**

Place your static files, such as HTML files and JSP files in the directory that is the document root of your Web Application. In the default installation of WebLogic Server, this directory is called DefaultWebApp, under user\_domains/mydomain/applications.

**DefaultWebApp/WEB-INF/web.xml**

The Web Application deployment descriptor that configures the Web Application.

**DefaultWebApp/WEB-INF/weblogic.xml**

The WebLogic-specific deployment descriptor file that defines how named resources in the web.xml file are mapped to resources residing elsewhere in WebLogic Server. This file is also used to define JSP and HTTP session attributes.

**DefaultWebApp/WEB-INF/classes**

Contains server-side classes such as HTTP servlets and utility classes.

**DefaultWebApp/WEB-INF/lib**

Contains JAR files used by the Web Application, including JSP tag libraries.

Q: - Can you explain JSP page life cycle?

Ans: -

Q: - What is EL?

Ans: -

One major component of JSP 2.0 is the new expression language named EL. EL is used extensively in JSTL. However, it is important to remember that EL is a feature of JSP and not of JSTL. JSP scriptlet code used with JSP 2.0 can contain EL expressions. The following lines of code demonstrate using EL inside of JSP scriptlet code.

<p>

 Your total, including shipping is ${total+shipping}

</p>

As you can see form the preceding code, the values "total" and "shipping" are added and displayed as the HTML is generated. These expressions can be used inside of JSTL tags as well. One important requirement of JSTL 1.0 was that JSTL could be used with JSP 1.2. Because JSP 1.2 does not support EL, it is necessary to provide a few additional JSTL tags that facilitate the use of EL. For example, if you wanted to use JSTL to display the above expression, you would use the following code.

<p>

 Your total, including shipping is <c:out var="${total+shipping"/>

</p>

One of the requirements of JSTL was that it not require JSP 2.0 to run. By providing a tag that is capable of displaying EL expressions, this requirement is met.

A primary feature of JSP technology version 2.0 is its support for an expression language (EL). An expression language makes it possible to easily access application data stored in JavaBeans components. For example, the JSP expression language allows a page author to access a bean using simple syntax such as ${name} for a simple variable or ${name.foo.bar} for a nested property.

The test attribute of the following conditional tag is supplied with an EL expression that compares the number of items in the session-scoped bean named cart with 0:

<c:if test="${sessionScope.cart.numberOfItems > 0}">

  ...

</c:if>

The JSP expression evaluator is responsible for handling EL expressions, which are enclosed by the ${ } characters and can include literals. Here's an example:

<c:if test="${bean1.a < 3}" >

  ...

</c:if>

Any value that does not begin with ${ is treated as a literal and is parsed to the expected type using the PropertyEditor for the type:

<c:if test="true" >

...

</c:if>

Literal values that contain the ${ characters must be escaped as follows:

<mytags:example attr1="an expression is ${'${'}true}" />

### Deactivating Expression Evaluation

Because the pattern that identifies EL expressions--${ }--was not reserved in the JSP specifications before JSP 2.0, there may be applications where such a pattern is intended to pass through verbatim. To prevent the pattern from being evaluated, you can deactivate EL evaluation.

To deactivate the evaluation of EL expressions, you specify the isELIgnored attribute of the page directive:

<%@ page isELIgnored ="true|false" %>

The valid values of this attribute are true and false. If it is true, EL expressions are ignored when they appear in static text or tag attributes. If it is false, EL expressions are evaluated by the container.

The default value varies depending on the version of the web application deployment descriptor. The default mode for JSP pages delivered using a Servlet 2.3 or earlier descriptor is to ignore EL expressions; this provides backward compatibility. The default mode for JSP pages delivered with a Servlet 2.4 descriptor is to evaluate EL expressions; this automatically provides the default that most applications want. You can also deactivate EL expression evaluation for a group of JSP pages (see [Deactivating EL Expression Evaluation](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro13.html#wp83214)).

### Using Expressions

EL expressions can be used:

* In static text
* In any standard or custom tag attribute that can accept an expression

The value of an expression in static text is computed and inserted into the current output. If the static text appears in a tag body, note that an expression will not be evaluated if the body is declared to be tagdependent (see [body-content Attribute](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPTags5.html#wp89848)).

There are three ways to set a tag attribute value:

* With a single expression construct:

<some:tag value="${expr}"/>

The expression is evaluated and the result is coerced to the attribute's expected type.

* With one or more expressions separated or surrounded by text:

<some:tag value="some${expr}${expr}text${expr}"/>

The expressions are evaluated from left to right. Each expression is coerced to a String and then concatenated with any intervening text. The resulting String is then coerced to the attribute's expected type.

* With text only:

<some:tag value="sometext"/>

In this case, the attribute's String value is coerced to the attribute's expected type.

Expressions used to set attribute values are evaluated in the context of an expected type. If the result of the expression evaluation does not match the expected type exactly, a type conversion will be performed. For example, the expression ${1.2E4} provided as the value of an attribute of type float will result in the following conversion:

Float.valueOf("1.2E4").floatValue()

See section JSP2.8 of the [JSP 2.0 specification](http://java.sun.com/products/jsp/download.html#specs) for the complete type conversion rules.

### Variables

The web container evaluates a variable that appears in an expression by looking up its value according to the behavior of PageContext.findAttribute(String). For example, when evaluating the expression ${product}, the container will look for product in the page, request, session, and application scopes and will return its value. If product is not found, null is returned. A variable that matches one of the implicit objects described in [Implicit Objects](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html#wp71043) will return that implicit object instead of the variable's value.

Properties of variables are accessed using the . operator and can be nested arbitrarily.

The JSP expression language unifies the treatment of the . and [] operators. expr-a.identifier-b is equivalent to expr-a["identifier-b"]; that is, the expression expr-b is used to construct a literal whose value is the identifier, and then the [] operator is used with that value.

To evaluate expr-a[expr-b], evaluate expr-a into value-a and evaluate expr-b into value-b. If either value-a or value-b is null, return null.

* If value-a is a Map, return value-a.get(value-b). If !value-a.containsKey(value-b), then return null.
* If value-a is a List or array, coerce value-b to int and return value-a.get(value-b) or Array.get(value-a, value-b), as appropriate. If the coercion couldn't be performed, an error is returned. If the get call returns an IndexOutOfBoundsException, null is returned. If the get call returns another exception, an error is returned.
* If value-a is a JavaBeans object, coerce value-b to String. If value-b is a readable property of value-a, then return the result of a get call. If the get method throws an exception, an error is returned.

### Implicit Objects

The JSP expression language defines a set of implicit objects:

* pageContext: The context for the JSP page. Provides access to various objects including:
	+ servletContext: The context for the JSP page's servlet and any web components contained in the same application. See [Accessing the Web Context](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets10.html#wp64724).
	+ session: The session object for the client. See [Maintaining Client State](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets11.html#wp64744).
	+ request: The request triggering the execution of the JSP page. See [Getting Information from Requests](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets7.html#wp64433).
	+ response: The response returned by the JSP page. See [Constructing Responses](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets7.html#wp64531).

In addition, several implicit objects are available that allow easy access to the following objects:

* param: Maps a request parameter name to a single value
* paramValues: Maps a request parameter name to an array of values
* header: Maps a request header name to a single value
* headerValues: Maps a request header name to an array of values
* cookie: Maps a cookie name to a single cookie
* initParam: Maps a context initialization parameter name to a single value

Finally, there are objects that allow access to the various scoped variables described in [Using Scope Objects](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets5.html#wp64315).

* pageScope: Maps page-scoped variable names to their values
* requestScope: Maps request-scoped variable names to their values
* sessionScope: Maps session-scoped variable names to their values
* applicationScope: Maps application-scoped variable names to their values

When an expression references one of these objects by name, the appropriate object is returned instead of the corresponding attribute. For example, ${pageContext} returns the PageContext object, even if there is an existing pageContext attribute containing some other value.

### Literals

The JSP expression language defines the following literals:

* Boolean: true and false
* Integer: as in Java
* Floating point: as in Java
* String: with single and double quotes; " is escaped as \", ' is escaped as \', and \ is escaped as \\.
* Null: null

### Operators

In addition to the . and [] operators discussed in [Variables](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html#wp84857), the JSP expression language provides the following operators:

* Arithmetic: +, - (binary), \*, / and div, % and mod, - (unary)
* Logical: and, &&, or, ||, not, !
* Relational: ==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le. Comparisons can be made against other values, or against boolean, string, integer, or floating point literals.
* Empty: The empty operator is a prefix operation that can be used to determine whether a value is null or empty.
* Conditional: A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.

The precedence of operators highest to lowest, left to right is as follows:

* [] .
* () - Used to change the precedence of operators.
* - (unary) not ! empty
* \* / div % mod
* + - (binary)
* < > <= >= lt gt le ge
* == != eq ne
* && and
* || or
* ? :

### Reserved Words

The following words are reserved for the JSP expression language and should not be used as identifiers.

and   eq   gt   true   instanceof

or    ne   le   false  empty

not   lt   ge   null   div   mod

Note that many of these words are not in the language now, but they may be in the future, so you should avoid using them.

### Examples

[Table 12-2](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html%22%20%5Cl%20%22wp77083) contains example EL expressions and the result of evaluating them.

|  |
| --- |
| Table 12-2 Example Expressions   |
| **EL Expression**  | **Result**  |
| ${1 > (4/2)}  | false  |
| ${4.0 >= 3}  | true  |
| ${100.0 == 100}  | true  |
| ${(10\*10) ne 100}  | false  |
| ${'a' < 'b'}  | true  |
| ${'hip' gt 'hit'}  | false  |
| ${4 > 3}  | true  |
| ${1.2E4 + 1.4}  | 12001.4  |
| ${3 div 4}  | 0.75  |
| ${10 mod 4}  | 2  |
| ${empty param.Add}  | True if the request parameter named Add is null or an empty string  |
| ${pageContext.request.contextPath}  | The context path  |
| ${sessionScope.cart.numberOfItems}  | The value of the numberOfItems property of the session-scoped attribute named cart  |
| ${param['mycom.productId']}  | The value of the request parameter named mycom.productId  |
| ${header["host"]}  | The host  |
| ${departments[deptName]}  | The value of the entry named deptName in the departments map  |
| ${requestScope['javax.servlet.forward.servlet\_path']}  | The value of the request-scoped attribute named javax.servlet.forward.servlet\_path  |

### Functions

The JSP expression language allows you to define a function that can be invoked in an expression. Functions are defined using the same mechanisms as custom tags (See [Using Custom Tags](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro9.html#wp73314) and Chapter [15](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPTags.html#wp74641)).

#### Using Functions

Functions can appear in static text and tag attribute values.

To use a function in a JSP page, you use a taglib directive to import the tag library containing the function. Then you preface the function invocation with the prefix declared in the directive.

For example, the date example page index.jsp imports the /functions library and invokes the function equals in an expression:

<%@ taglib prefix="f" uri="/functions"%>

...

    <c:when

      test="${f:equals(selectedLocaleString,

        localeString)}" >

#### Defining Functions

To define a function you program it as a public static method in a public class. The mypkg.MyLocales class in the date example defines a function that tests the equality of two Strings as follows:

package mypkg;

public class MyLocales {

  ...

  public static boolean equals( String l1, String l2 ) {

    return l1.equals(l2);

  }

}

Then you map the function name as used in the EL expression to the defining class and function signature in a TLD. The following functions.tld file in the date example maps the equals function to the class containing the implementation of the function equals and the signature of the function:

<function>

  <name>equals</name>

  <function-class>mypkg.MyLocales</function-class>

  <function-signature>boolean equals( java.lang.String,

    java.lang.String )</function-signature>

</function>

A tag library can have only one function element that has any given name element.

Q: - How does EL search for an attribute?

Ans: -

Q: - What are the implicit EL objects in JSP?

Ans: -

The JSP expression language defines a set of implicit objects:

* pageContext: The context for the JSP page. Provides access to various objects including:
	+ servletContext: The context for the JSP page's servlet and any web components contained in the same application. See [Accessing the Web Context](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets10.html#wp64724).
	+ session: The session object for the client. See [Maintaining Client State](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets11.html#wp64744).
	+ request: The request triggering the execution of the JSP page. See [Getting Information from Requests](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets7.html#wp64433).
	+ response: The response returned by the JSP page. See [Constructing Responses](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets7.html#wp64531).

In addition, several implicit objects are available that allow easy access to the following objects:

* param: Maps a request parameter name to a single value
* paramValues: Maps a request parameter name to an array of values
* header: Maps a request header name to a single value
* headerValues: Maps a request header name to an array of values
* cookie: Maps a cookie name to a single cookie
* initParam: Maps a context initialization parameter name to a single value

Finally, there are objects that allow access to the various scoped variables described in [Using Scope Objects](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets5.html#wp64315).

* pageScope: Maps page-scoped variable names to their values
* requestScope: Maps request-scoped variable names to their values
* sessionScope: Maps session-scoped variable names to their values
* applicationScope: Maps application-scoped variable names to their values

When an expression references one of these objects by name, the appropriate object is returned instead of the corresponding attribute. For example, ${pageContext} returns the PageContext object, even if there is an existing pageContext attribute containing some other value.

Q: - How can we disable EL?

Ans: -

Because the pattern that identifies EL expressions--${ }--was not reserved in the JSP specifications before JSP 2.0, there may be applications where such a pattern is intended to pass through verbatim. To prevent the pattern from being evaluated, you can deactivate EL evaluation.

To deactivate the evaluation of EL expressions, you specify the isELIgnored attribute of the page directive:

<%@ page isELIgnored ="true|false" %>

The valid values of this attribute are true and false. If it is true, EL expressions are ignored when they appear in static text or tag attributes. If it is false, EL expressions are evaluated by the container.

The default value varies depending on the version of the web application deployment descriptor. The default mode for JSP pages delivered using a Servlet 2.3 or earlier descriptor is to ignore EL expressions; this provides backward compatibility. The default mode for JSP pages delivered with a Servlet 2.4 descriptor is to evaluate EL expressions; this automatically provides the default that most applications want. You can also deactivate EL expression evaluation for a group of JSP pages (see [Deactivating EL Expression Evaluation](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro13.html#wp83214)).

Each JSP page has a default mode for EL expression evaluation. The default value varies depending on the version of the web application deployment descriptor. The default mode for JSP pages delivered using a Servlet 2.3 or earlier descriptor is to ignore EL expressions; this provides backward compatibility. The default mode for JSP pages delivered with a Servlet 2.4 descriptor is to evaluate EL expressions; this automatically provides the default that most applications want. For tag files (see [Encapsulating Reusable Content Using Tag Files](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPTags5.html#wp89664)), the default is to always evaluate expressions.

You can override the default mode through the isELIgnored attribute of the page directive in JSP pages and through the isELIgnored attribute of the tag directive in tag files. You can also explicitly change the default mode by setting the value of the EL Evaluation Ignored checkbox in the JSP Properties tab. [Table 12-5](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro13.html#wp78166) summarizes the EL evaluation settings for JSP pages and their meanings.

|  |
| --- |
| Table 12-5 EL Evaluation Settings for JSP Pages   |
| **JSP Configuration**  | **Page DirectiveisELIgnored**  | **EL Encountered**  |
| Unspecified  | Unspecified  | Evaluated if 2.4 web.xml Ignored if <= 2.3 web.xml  |
| false  | Unspecified  | Evaluated  |
| true  | Unspecified  | Ignored  |
| Overridden by page directive  | false  | Evaluated  |
| Overridden by page directive  | true  | Ignored  |

[Table 12-6](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro13.html%22%20%5Cl%20%22wp78216) summarizes the EL evaluation settings for tag files and their meanings.

|  |
| --- |
| Table 12-6 EL Evaluation Settings for Tag Files   |
| **Tag Directive isELIgnored**  | **EL Encountered**  |
| Unspecified  | Evaluated  |
| false  | Evaluated  |
| true  | Ignored  |

Q: - What is JSTL?

Ans: -

The JSP Standard Template Library (JSTL) is a very new component released by Sun for JSP programming. JSTL allows you to program your JSP pages using tags, rather than the scriptlet code that most JSP programmers are already accustomed to. JSTL can do nearly everything that regular JSP scriptlet code can do. You may be wondering why we need yet another HTML generation programming language.

JSTL was introduced was to allow JSP programmers to program using tags rather than Java code. To show why this is preferable, a quick example is in order. We will examine a very simple JSP page that counts to ten. We will examine this page both as regular scriptlet-based JSP, and then as JSTL. When the count to ten example is programmed using scriptlet based JSP, the JSP page appears as follows.

<html>

 <head>

 <title>Count to 10 in JSP scriptlet</title>

 </head>

 <body>

<%

 for(int i=1;i<=10;i++)

{%>

<%=i%><br/>

<%

}

%>

 </body>

</html>

As you can see from the preceding example, using scriptlet code produces page source code the contains a mix of HTML tags and Java statements. There are several reasons why this mixing of programming styles is not optimal.

The primary reason that it is not optimal to mix scriptlet and tag-based code is readability. This readability apples both to humans and computers. JSTL allows the human programmer to look at a program that consists entirely of HTML and HTML-like tags.

The readability of JSP scriptlet code does not just apply to human beings. The mixing of scriptlet and HTML code is also hard for computers to read. This is particularly true of HTML authoring tools such as Someone's Dreamweaver and Microsoft FrontPage. Currently, most HTML authoring tools will segregate JSP scriptlet code as non-editable blocks. The HTML authoring tools usually do not modify the JSP scriptlet code directly.

The following code shows how the count to ten example would be written using JSTL. As you can see, this code listing is much more constant, as only tags are used. HTML and JSTL tags are mixed to produce the example.

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<html>

 <head>

 <title>Count to 10 Example (using JSTL)</title>

 </head>

 <body>

 <c:forEach var="i" begin="1" end="10" step="1">

 <c:out value="${i}" />

 <br />

 </c:forEach>

 </body>

</html>

When you examine the preceding source code, you can see that the JSP page consists entirely of tags. The above code makes use of HTML tags such <head> and <br>. The use of tags is not confined just to HTML tags. This code also makes use of JSTL tags such as <c:forEach> and <c:out>. In this article you will be introduced to some of the basics of JSTL.

Q: - Can you explain in short what the different types of JSTL tags are?

Ans: -

### The JSTL Tag Libraries

JSTL is often spoken of as a single-tag library. JSTL is actually four tag libraries. These tag libraries are summarized as follows.

* **Core Tag Library**—Contains tags that are essential to nearly any Web application. Examples of core tag libraries include looping, expression evaluation, and basic input and output.
* **Formatting/Internationalization Tag Library**—Contains tags that are used to and parse data. Some of these tags will parse data, such as dates, differently based on the current locale.
* **Database Tag Library**—Contains tags that can be used to access SQL databases. These tags are normally used only to create prototype programs. This is because most programs will not handle database access directly from JSP pages. Database access should be embedded in EJBs that are accessed by the JSP pages.
* **XML Tag Library**—Contains tags that can be used to access XML elements. Because XML is used in many Web applications, XML processing is an important feature of JSTL.

In this article, we will only take a brief look at a few of the core tags. We will examine a simple example that shows how to process data that a user enters into a form. Before we examine this program, we must first see how JSTL handles expressions. Expression handling in JSTL is accomplished by using the EL expression language, just as it is done in JSP 2.0. In the next section, we will examine the EL expression language.

### JSTL Example

We will now examine a simple example that uses JSTL. For this example, we will examine a common procedure that is done by many Web applications. We will see how to POST a form and process the results from that POST. A simple program that is capable of doing this is shown below.

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<html>

 <head>

 <title>If with Body</title>

 </head>

 <body>

 <c:if test="${pageContext.request.method=='POST'}">

 <c:if test="${param.guess=='Java'}">You guessed it!

 <br />

 <br />

 <br />

 </c:if>

 <c:if test="${param.guess!='Java'}">You are wrong

 <br />

 <br />

 <br />

 </c:if>

 </c:if>

 <form method="post">Guess what computer language

 I am thinking of?

 <input type="text" name="guess" />

 <input type="submit" value="Try!" />

 <br />

 </form>

 </body>

</html>

This simple Web page will display a form and ask the user to guess what computer language the program is thinking of. Of course, the computer is thinking of "Java." This page begins by checking to see if a POST was done. This allows both the form, and the code that handles the form, to be placed on one single page. This is done with the following JSTL if statement.

 <c:if test="${pageContext.request.method=='POST'}">

Here you can see that the <c:if> tag uses an EL expression to evaluate whether the request method is POST. If data was posted to the page, the value that the user entered for their guess is stored in a parameter named "guess". This is because "guess" was specified as the name of the form input item. We must now check to see whether this parameter is equal to the word "Java". This is done with the following <c:if> tag.

 <c:if test="${param.guess=='Java'}">

 You guessed it!

 </c:if>

As you can see, the body of the <c:if> tag is executed if the statement evaluates to true. In this article, we began to examine the basics of how JSTL is installed and how it works. There is much more to JSTL than the small example we examined in this article.

The core tags of JSTL also include tags for looping, iteration, and variable handling. By using these tags, you can iterate through collections, access user session data, and perform other core tasks that all Web applications perform. In addition to the core tag library, the XML, database, and formatting tag libraries are also provided for more advanced uses.

Q: - How can we use beans in JSP?

Ans: -

**Java Beans**

Java Beans are reusable components. They are used to separate Business logic from the Presentation logic. Internally, a bean is just an instance of a class.

 JSP’s provide three basic tags for working with Beans.

* **<jsp:useBean id=“*bean name*” class=“bean class”  scope = “page | request | session |application ”/>**

 bean name = the name that refers to the bean.

Bean class = name of the java class that defines the bean.

* **<jsp:setProperty name = “id” property = “someProperty”    value = “someValue” />**

      id = the name of the bean as specified in the useBean tag.

property = name of the property to be passed to the bean.

value = value of that particular property .

An variant for this tag is the property attribute can be replaced by an  “ \* ”. What this does is that it accepts all the form parameters and thus reduces the need for writing multiple setProperty tags. The only consideration is that the form parameter names should be the same as that of the bean property names.

* **<jsp:getProperty name = “id” property = “someProperty” />**

Here the property is the name of the property whose value is to be obtained from the bean.

 **BEAN SCOPES :**

          These defines the range and lifespan of the bean.

          The different options are :

* + Page scope :

Any object whose scope is the page will disappear as soon as the current page finishes generating. The object with a page scope may be modified as often as desired within the particular page but the changes are lost as soon as the page exists.

By default all beans have page scope.

* + Request scope :

Any objects created in the request scope will be available as long as the request object is.  For example if the JSP page uses an jsp:forward tag, then the bean should be applicable in the forwarded JSP also, if the scope defined is of Request scope.

* + The Session scope :

In JSP terms, the data associated with the user has session scope. A session does not correspond directly to the user; rather, it corresponds with a particular period of time the user spends at a site. Typically, this period is defined as all the visits a user makes to a site between starting and existing his browser.

 **The BEAN structure :**

The most basic kind of bean simply exposes a number of properties by following a few simple rules regarding method names. The Java BEAN is not much different from an java program. The main differences are the signature methods being used in a bean. For passing parameters to a bean, there has to be a corresponding get/set method for every parameter. Together these methods are known as *accessors*.

 **Eg.** Suppose we want to pass a parameter “name” to the bean and then return it in the capital form. In the bean, there has to be an setName() method and an corresponding getProperty() method.  A point to be noted is that the first letter of the property name is capitalized.(Here, N is in capital)

          Also, it is possible to have either get or set in a bean, depending on the requirement for a read only or a write only property.

An example for a Database connection bean is as shown :

|  |
| --- |
| package SQLBean; import java.sql.\*;  import java.io.\*;  public class DbBean {   String dbURL = "jdbc:db2:sample";  String dbDriver = "COM.ibm.db2.jdbc.app.DB2Driver";   private Connection dbCon;   public DbBean(){         super();               }   public boolean connect() throws ClassNotFoundException,SQLException{           Class.forName(dbDriver);           dbCon = DriverManager.getConnection(dbURL);           return true;         }     public void close() throws SQLException{         dbCon.close();        }   public ResultSet execSQL(String sql) throws SQLException{                     Statement s = dbCon.createStatement();                     ResultSet r = s.executeQuery(sql);                     return (r == null) ? null : r;                     }   public int updateSQL(String sql) throws SQLException{                                        Statement s = dbCon.createStatement();                   int r = s.executeUpdate(sql);                   return (r == 0) ? 0 : r;                 } }      |

The description is as follows :

          This bean is packaged in a folder called as “SQLBean”. The name of the class file of the bean is **DbBean**. For this bean we have hardcoded the Database Driver and the URL. All the statements such as connecting to the database, fetching the driver etc are encapsulated in the bean.

There are two methods involved in this particular bean :

Executing a particular query.

Updating a database.

The *execSQL(String sql)* method accepts the SQL query in the form of a string from the JSP file in which this bean is implemented.

Then the *createStatement()* method initiates the connection with the *dbCon*  connection object.

Further the *executeQuery(sql)* method executes the query which is passed on as a string.

*return (r == null) ? null : r ;*

 What this statement does is that, if the value of *r* is *null*, it returns a null value and if it is a non null value, it returns the value of *r*. Though this statement seems redundant, it is useful for preventing any errors that might occur due to improper value being set in *r.*

The JSP Program is as shows :

|  |
| --- |
| <HTML>  <HEAD><TITLE>DataBase Search</TITLE></HEAD>  <BODY> <%@ page language="Java" import="java.sql.\*" %>   <jsp:useBean id="db" scope="request" class="SQLBean.DbBean" /> <jsp:setProperty name="db" property="\*" />  <%!           ResultSet rs = null ;          ResultSetMetaData rsmd = null ;          int numColumns ;          int i;  %> <center>  <h2> Results from </h2>  <hr>  <br><br>  <table> <%    db.connect(); try {           rs = db.execSQL("select \* from EMPLOYEE");            i = db.updateSQL("UPDATE employee set FIRSTNME = 'hello world' where EMPNO='000010'");           out.println(i);         }catch(SQLException e) {               throw new ServletException("Your query is not working", e);                      }             rsmd = rs.getMetaData();           numColumns = rsmd.getColumnCount();           for(int column=1; column <= numColumns; column++){                      out.println(rsmd.getColumnName(column));            }  %> <%               while(rs.next()) {  %>      <%= rs.getString("EMPNO") %> <BR>  <%                 }  %>  <BR>  <%    db.close(); %> Done </table> </body> </HTML>     |

The corresponding tags used in the JSP are as follows :

**<jsp:useBean id="db" scope="request" class="SQLBean.DbBean" />**

This tag specifies that the id of this bean is “db”. This id is used throughout the page to refer to this particular bean. The scope of this bean is limited to the request scope only. The class attribute points to the class of the bean.

Here the class file is stored in the SQLBean folder.

**<jsp:setProperty name="db" property="\*" />**

This property is used for passing on all the values which are obtained from the form. In this program, the SQL query can be passed on to the program as a part of the *request.getParameter* so that the query can be modified according to the requests.

**rs = db.execSQL("select \* from EMPLOYEE");**

We can access the *execSQL()* method by using the bean id. Also the SQL is passed on to this method.

i = db.updateSQL("UPDATE employee set FIRSTNME = 'hello world' where EMPNO='000010'");

The *updateSQL()*method can also be used in the same JSP program. Here we are updating the employee table and resetting the FIRSTNME field where the EMPNO is 000010.

The major difference between an executeQuery and executeUpdate is that, an executeQuery returns the result set and an executeUpdate returns an integer value corresponding to the number of rows updated by the current query.

As can be seen, it is very easy to connect to databases using beans rather than writing the whole code over and over again in every JSP which requires to talk to the database.

Q: - What is <jsp:forward> tag for ?

Ans: -

The <jsp:forward> element forwards the request object containing the client request information from one JSP file to another file. The target file can be an HTML file, another JSP file, or a servlet, as long as it is in the same application context as the forwarding JSP file. The lines in the source JSP file the <jsp:forward> element are not processed.

This sample code shows the use of tag. This checks the percentage of free memory and based on that opens new page using this tag.

**DemoForward.jsp**

|  |
| --- |
| <html><%  **double**freeMemory = Runtime.getRuntime().freeMemory();  **double**totalMemory = Runtime.getRuntime().totalMemory();  **double**percent = freeMemory/totalMemory;  **if**(percent<0.5){%>  <jsp:forward page="one.jsp"/><%}**else**{%><jsp:forward page="two.html"/><%}%></html>  |

**one.jsp**

|  |
| --- |
| <html><body><font color=”red”>VM Memory usage<50%</html>  |

**two.html**

|  |
| --- |
| <html><body bgcolor= “white”><font color=”red”>VM Memory usage>50%</body><html> |

Forwards a client request to an HTML file, JSP file, or servlet for processing.

#### JSP Syntax

<jsp:forward page="{ relativeURL | <%= expression %> }" />

#### Examples

<jsp:forward page="/servlet/login" />

#### Description

The <jsp:forward> tag forwards the request object sent to the JSP file to another file for processing. The JSP engine does not process any of the remainder of the JSP file.

If the output from the compiled JSP file is buffered (by using a page directive with the default value or an explicit size set for buffer), the buffer is cleared before the request is forwarded. If the output is not buffered (if you used a page directive with buffer=none), and if anything has been written to the buffer, using <jsp:forward> results in an IllegalStateException.

Q: - What are JSP directives?

Ans:-

JSP Directives control the processing of an entire page. Directive examples include setting a scripting language, setting an error page, including other sections, and setting a character encoding.

|  |
| --- |
| 1. [Index](http://www.caucho.com/resin-3.0/jsp/directives.xtp#Index)
2. [JSP Directives](http://www.caucho.com/resin-3.0/jsp/directives.xtp#jsp)
 |

|  |
| --- |
| **Index** |

|  |  |
| --- | --- |
| [<%@ include file="path" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#include) | Includes the raw file path at translation time |
| [<%@ page autoFlush="true" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#autoFlush) | Tells JSP to flush the page buffer when it fills |
| [<%@ page buffer=sizekb %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#buffer) | Gives the size of the page buffer in kb or none for no buffer |
| [<%@ page contentType="description" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#contentType) | Sets the content type and character encoding of the page |
| [<%@ page errorPage="path" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#errorPage) | Defines a page to display if an error occurs in the JSP page |
| [<%@ page extends="Java class" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#extends) | Changes the generated servlet's class |
| [<%@ page import="package" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#import) | Adds to the Java package import list for the generated Java file |
| [<%@ page info="description" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#info) | Gives a brief description for the page |
| [<%@ page isErrorPage="true" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#isErrorPage) | Gives an error page access to the exception implicit variable |
| [<%@ page isThreadSafe="true" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#isThreadSafe) | Tells the JSP that multiple pages can execute in parallel |
| [<%@ page language="lang" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#language) | Sets the JSP script language to lang |
| [<%@ page session="true" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#pagesession) | Tells JSP that the page participates in a session |
| [<%@ taglib prefix="x" uri="foo" %>](http://www.caucho.com/resin-3.0/jsp/directives.xtp#taglib) | Configures tags with prefix x to use the tag library foo |

|  |
| --- |
| **JSP Directives** |

#### <%@ page language="lang" %>

Sets the JSP script language to lang. Defaults to Java. All JSP 1.0 implementations must support Java. Some implementations, like Resin, may support other scripting languages, e.g. JavaScript.

#### <%@ page import="package" %>

Adds to the Java package import list for the generated Java file.

**Note:** Only relevant when using Java.

#### <%@ page errorPage="path" %>

Defines a page to display if an error occurs in the JSP page.

Robust applications can return informative error pages when something goes wrong in a file, for example if a database is overloaded. path is returned as the response file. The error page can use the additional implicit variable [exception](http://www.caucho.com/resin-3.0/jsp/directives.xtp#exception) containing the thrown exception.

path is relative to the current page. Its root is relative to the application root.

The error page itself can be a JSP page. If the error page is a JSP page, it can use the implicit variable exception to get information about the thrown exception.

#### <%@ page isErrorPage="true" %>

Gives an error page access to the exception implicit variable. Default to false.

|  |
| --- |
| errorpage.jsp  |
|  <%@ page isErrorPage="true" %> <h1>Received error <%= exception.message %></h1>  |

#### <%@ include file="path" %>

Includes the raw file path at translation time.

The include directive is a replacement for an SSI include (or the C '#include'). It includes the contents of the file at path into the JSP file. The included file is parsed as JSP, so it can have active elements like expressions, declarations and scriptlets.

path is relative to the current page, and its root is the application root.

|  |
| --- |
| header.jsp  |
|  <html><head> <title><%= title %></title> </head> <body color=white>  |

|  |
| --- |
| page.jsp  |
|  <% var title = "Hello, World"; %> <%@ include file='header.jsp' %> <h1><%= title %></h1>  |

#### <%@ page buffer=sizekb %>

Gives the size of the page buffer in kb or none for no buffer. Default 8kb. If buffer is none, all output is immediately flushed.

JSP 1.0 gives page writers flexibility by buffering its output before sending the response to HTTP. The buffering allows [error recovery](http://www.caucho.com/resin-3.0/jsp/directives.xtp#errorpage) and forwarding, even after generating some content. Once the buffer has filled, it will be flushed. So applications must still detect their errors early.

The following example generates an XML section (for variety). If the form's query is missing the 'name' parameter, it will redirect the results.

|  |
| --- |
|  <?xml version='1.0'?> <form> <% if (request.form["name"] == null) pageContext.forward("redo-form.jsp"); for (var name in request.form) { out.print("<" + name + ">"); out.print(request.form[name]); out.println("</" + name + ">"); } %> </form>  |

#### <%@ page autoFlush="true" %>

Tells JSP to flush the page buffer when it fills. Default is true.

If autoFlush is false, the JSP engine will throw an exception if the buffer overflows.

#### <%@ page session="true" %>

Tells JSP that the page participates in a session. Defaults to true.

The session declaration makes the session implicit variable available to a JSP page.

If the page doesn't use sessions, it should set session to false.

|  |
| --- |
|  <%@ page session="true" %> <%  session.value.count++; %> <h1>Welcome, visitor <%= count %></h1;>  |

#### <%@ page isThreadSafe="true" %>

Tells the JSP that multiple pages can execute in parallel. Defaults to true.

JSP pages are always responsible for synchronization of shared variables, such as the session and application variables. In some rare cases, a page may use servlet variables (created with a declaration), and be too lazy to handle the synchronization.

Even with isThreadSafe=false, the JSP engine may create multiple instances of the JSP servlet. So the page author can never absolve herself of synchronization issues.

In the following example, a JSP engine might create 3 servlet instances of the page. So three calls to the same page may return counts of 17, 3 and 398. In addition, the JSP engine is free to destroy and recreate the servlet at any time, essentially resetting the counter to 0.

|  |
| --- |
|  <%@ page isThreadSafe="false" %> <%! var count = 0; %> <h1>Welcome, visitor <%= count++ %>  |

#### <%@ page info="description" %>

Gives a brief description for the page.

#### <%@ page contentType="description" %>

Sets the content type and character encoding of the page.

contentType can also set the character encoding, for example to utf-8.

|  |
| --- |
|  <%@ page contentType="text/plain; charset=utf-8" %> <%! var count = 0; %> <h1>Welcome, visitor <%= count++ %>  |

#### <%@ page extends="Java class" %>

Changes the generated servlet's class.

In general, a filter is a better solution than using the extends directive.

#### <%@ taglib prefix="x" uri="foo" %>

Configures tags with prefix x to use the tag library foo.

|  |
| --- |
|  <%@ taglib prefix='x' uri='http://www.caucho.com/mytag/test' %> <x:mytag/>  |

Q: - What are Page directives?

Ans: -

Defines attributes that apply to an entire JSP page.

#### JSP Syntax

<%@ page
          [ language="**java**" ]
          [ extends="package.class" ]
          [ import="{package.class | package.\*}, ..." ]
          [ session="**true** | false" ]
          [ buffer="none | **8kb** | sizekb" ]
          [ autoFlush="**true** | false" ]
          [ isThreadSafe="**true** | false" ]
          [ info="text" ]
          [ errorPage="relativeURL" ]
          [ contentType="mimeType [ ;charset=characterSet ]"   |   "**text/html ; charset=ISO-8859-1**" ]
          [ isErrorPage="true | **false**" ]
%>

#### Examples

<%@ page import="java.util.\*, java.lang.\*" %>
<%@ page buffer="5kb" autoFlush="false" %>
<%@ page errorPage="error.jsp" %>

#### Description

The <%@ page %> directive applies to an entire JSP file and any of its static include files, which together are called a translation unit. A static include file is a file whose content becomes part of the calling JSP file. The <%@ page %> directive does not apply to any dynamic include files; see [<jsp:include>](http://java.sun.com/products/jsp/tags/11/syntaxref11.fm11.html#8828) for more information.

You can use the <%@ page %> directive more than once in a translation unit, but you can only use each attribute, except import, once. Because the import attribute is similar to the import statement in the Java programming language, you can use a <%@ page %> directive with import more than once in a JSP file or translation unit.

No matter where you position the <%@ page %> directive in a JSP file or included files, it applies to the entire translation unit. However, it is often good programming style to place it at the top of the JSP file.

#### Attributes

* language="**java**"

The scripting language used in scriptlets, declarations, and expressions in the JSP file and any included files. In this release, the only allowed value is java.

* extends="package.class"

The fully qualified name of the superclass of the Java class file this JSP file will be compiled to. Use this attribute cautiously, as it can limit the JSP container's ability to provide a specialized superclass that improves the quality of the compiled file.

* import="{package.class | package.\* }, ..."

A comma-separated list of Java packages that the JSP file should import. The packages (and their classes) are available to scriptlets, expressions, and declarations within the JSP file. If you want to import more than one package, you can specify a comma-separated list after import or you can use import more than once in a JSP file.

The following packages are implicitly imported, so you don't need to specify them with the import attribute:

java.lang.\*
javax.servlet.\*
javax.servlet.jsp.\*
javax.servlet.http.\*

You must place the import attribute before the element that calls the imported class.

* session="**true** | false"

Whether the client must join an HTTP session in order to use the JSP page. If the value is true, the session object refers to the current or new session.

If the value is false, you cannot use the session object or a <jsp:useBean> element with scope=session in the JSP file. Either of these usages would cause a translation-time error.

The default value is true.

* buffer="none | **8kb** | sizekb"

The buffer size in kilobytes used by the out object to handle output sent from the compiled JSP page to the client Web browser. The default value is 8kb. If you specify a buffer size, the output is buffered with at least the size you specified.

* autoFlush="**true** | false"

Whether the buffered output should be flushed automatically when the buffer is full. If set to true (the default value), the buffer will be flushed. If set to false, an exception will be raised when the buffer overflows. You cannot set autoFlush to false when buffer is set to none.

* isThreadSafe="**true** | false"

Whether thread safety is implemented in the JSP file. The default value is true, which means that the JSP container can send multiple, concurrent client requests to the JSP page. You must write code in the JSP page to synchronize the multiple client threads. If you use false, the JSP container sends client requests one at a time to the JSP page.

* info="text"

A text string that is incorporated verbatim into the compiled JSP page. You can later retrieve the string with the Servlet.getServletInfo() method.

* errorPage="relativeURL"

A pathname to a JSP file that this JSP file sends exceptions to. If the pathname begins with a /, the path is relative to the JSP application's document root directory and is resolved by the Web server. If not, the pathname is relative to the current JSP file.

* isErrorPage="true | **false**"

Whether the JSP file displays an error page. If set to true, you can use the exception object in the JSP file. If set to false (the default value), you cannot use the exception object in the JSP file.

* contentType="mimeType [ ;charset=characterSet ]" | "**text/html;charset=ISO-8859-1**"

The MIME type and character encoding the JSP file uses for the response it sends to the client. You can use any MIME type or character set that are valid for the JSP container. The default MIME type is text/html, and the default character set is ISO-8859-1.

#### Tip

* If you need to include a long list of packages or classes in more than one JSP file, you can create a separate JSP file with a <%@ page %> directive that contains the import list and include that file in the main JSP file.

Q: - What are include directives?

Ans: -

**<%@ include%>**

The include directive enables you to include other JSPs (or static pages) when the JSP is compiled into a servlet. The resource is treated as if it were a part of the JSP.

There is another method for including other resources: the [<jsp:include>](http://www.javacommerce.com/displaypage.jsp?name=/home/downloads/neszip/iplanet_application_server_6_0_welcome_letter_release_notes_and_documentation/javaprogguide/jpgjsp.sql&id=18226#13660) action, which includes the resource at request-time. For more information on file inclusion, see .

**Valid Parameters**
<%@ include file="file" %>

**Example**

<%@include file="/portal/banner.jsp"%>

Most of the HTML part, can be created using any standard WYSIWYG page editor. The only time you would want to use these Tags , would most probably be when you want to have Text Displayed Dynamically every time.  That's when you can use these tags and directly display the values in a particular variable etc.JSP tags use the form <jsp:*tag*>, a form taken from XML. Some tags (particularly scripting tags) have a shortcut for use in HTML files, generally starting with <% and ending with %>.

Note that these shortcuts are not valid for XML files.

Empty elements, tag constructs that have nothing between the start and end tags, can be shortened to one tag ending with />, as in this example:

<!-- include tag with no body: -->
<jsp:include page="/portal/header.jsp"></jsp:include>

<!-- include tag can also be written like this: -->
<jsp:include page="/portal/header.jsp" />

White space is not usually significant, although you should make sure to put a space character between the opening tag and any attributes. For example: <%= myExpression %> is valid, <%=myExpression %> is not.

**What is JSP?**

With JSP, fragments of Java code embedded into special HTML tags are presented to a JSP engine. The JSP engine automatically creates, compiles, and executes servlets to implement the behavior of the combined HTML and embedded Java code.

Previous lessons have explained the use of the following syntax elements:

* Output comments
* Hidden comments
* Declarations
* Expressions
* Scriptlets

This lesson will discuss the *include directive*.

### What is the Include Directive?

The include directive is used to insert a file into a JSP page when the JSP page is compiled.  The text of the included file is added to the JSP page (see the description of a *static* file later in this lesson).

**What kinds of files are eligible for inclusion?**

The included file can be a JSP file, an HTML file, or a text file.  The included file can also be a code file written in the Java programming language.

According to Sun:

|  |
| --- |
| *The included file can be an HTML file, a JSP file, a text file, or a code file written in the Java programming language.* *Be careful, though, that the included file does not contain <html>, </html>, <body>, or </body> tags. Because the entire content of the included file is added at that location in the JSP file, these tags would conflict with similar tags in the JSP file.* |

**Oops!**

Actually, the sample JSP page that I am going to show you later violates this caution and ends up with nested <html> and <body> tags in the HTML source that the JSP page sends to the browser.

I have tested the JSP page with Netscape 4.7 and IE 5.0, and it seems to work OK with both of those browsers.  However, it is certainly possible that this could cause problems with other browsers.

**Including JSP files**

Sun tells us:

|  |
| --- |
| *If the included file is a JSP file, its JSP tags are parsed and their results included (along with any other text) in the JSP file.* |

A sample program later in this lesson illustrates the inclusion of a JSP file and an HTML file into a primary JSP page.

**Include only static files**

According to Sun:

|  |
| --- |
| *You can only use include to include static files. This means that the parsed result of the included file is added to the JSP file where the include directive is placed. Once the included file is parsed and included, processing resumes with the next line of the calling JSP file.* |

**So, what is a *static* file?**

According to Sun:

|  |
| --- |
| *A static include means that the text of the included file is added to the JSP file.* |

Elsewhere in conjunction with another JSP tag, *<jsp:include>, Sun* tells us:

|  |
| --- |
| *The <jsp:include> tag allows you to include either a static or dynamic file.* *A static file is parsed and its content included in the calling JSP page.* *A dynamic file acts on the request and sends back a result that is included in the JSP page.* |

**What is the syntax for including a file?**

You can include another file at a specific location in the JSP file using an include directive with the following syntax:

|  |
| --- |
| **<%@ include file="relativeURL" %>** |

### Sample Include Directives

The following unrelated samples of include directives are used in the sample program discussed later in this lesson.

|  |
| --- |
| **<%@ include file="jsp005a.jsp" %>** **<%@ include file="jsp005a.htm" %>** |

The first include directive causes another JSP file to be included at a specific location in the primary JSP file.

The second include directive causes an ordinary HTML file to be included at the appropriate location.

**More on *relativeURL***

In the above syntax description, I showed an attribute named *file* with a value of *relativeURL*.  The relativeURL value is simply the pathname to the included file.

The value of the file attribute is always a relative URL.  According to Sun,

|  |
| --- |
| *a relative URL is just the path segment of an URL, without the protocol, port, or domain.* |

Also according to Sun,

|  |
| --- |
| *If the relative URL starts with /, the path is relative to the JSP application's context, which is a javax.servlet.ServletContext object that is in turn stored in the application object.* *If the relative URL starts with a directory or file name, the path is relative to the JSP file.* |

If you don't know about the *ServletContext*, you will need to take a look at some information on servlets.  Several lessons on servlets are contained in my [online tutorials](http://www.geocities.com/Athens/7077/scoop/onjava.html).

In the two examples that I showed above, the relative URLs both begin with a file name.

Q: - Can you explain taglib directives?

Ans: -

Defines a tag library and prefix for the custom tags used in the JSP page.

#### JSP Syntax

<%@ taglib uri="URIToTagLibrary" prefix="tagPrefix" %>

#### Examples

<%@ taglib uri="http://www.jspcentral.com/tags" prefix="public" %>
<public:loop>
.
.
</public:loop>

#### Description

The <%@ taglib %> directive declares that the JSP file uses custom tags, names the tag library that defines them, and specifies their tag prefix.

Here, the term custom tag refers to both tags and elements. Because JSP files can be converted to XML, it is important to understand the relationship of tags and elements. A tag is simply a short piece of markup that is part of a JSP element. A JSP element is a unit of JSP syntax that has an XML equivalent with a start tag and an end tag. An element can also contain other text, tags, or elements. For example, a jsp:plugin element always has a <jsp:plugin> start tag and a </jsp:plugin> end tag and may have a <jsp:params> element and a <jsp:fallback> element.

You must use a <%@ taglib %> directive before you use the custom tag in a JSP file. You can use more than one <%@ taglib %> directive in a JSP file, but the prefix defined in each must be unique.

The technique for creating custom tags is described in the JavaServer Pages Specification for version 1.1.

#### Attributes

* uri="URIToTagLibrary"

The Uniform Resource Identifier (URI) that uniquely names the set of custom tags associated with the named tag prefix. A URI can be any of the following:

* + A Uniform Resource Locator (URL), as defined in RFC 2396, available at http://www.hut.fi/u/jkorpela/rfc/2396/full.html
	+ A Uniform Resource Name (URN), as defined in RFC 2396
	+ An absolute or relative pathname
* prefix="tagPrefix"

The prefix that precedes the custom tag name, for example, public in <public:loop>. Empty prefixes are illegal. If you are developing or using custom tags, you cannot use the tag prefixes jsp, jspx, java, javax, servlet, sun, and sunw, as they are reserved by Sun Microsystems.

|  |
| --- |
| Tag libraries allow you to create custom actions and encapsulate functionality. Custom tags can clearly separate the presentation layer from the business logic. They are easy to maintain reusable components that have access to all the objects available to JSP pages. Please consult the [JavaServer Pages Specification, version 1.2](http://www.jcp.org/aboutJava/communityprocess/final/jsr053) for more details. Servers that implement JSP, v1.1+ support tag libraries. You can find a description of various servers and what they currently support at the [JavaServer Pages Industry Momentum](http://java.sun.com/products/jsp/industry.html) page. There is also has a very good [Java Web Services Tutorial](http://java.sun.com/webservices/docs/ea1/tutorial/index.html) available on Sun's website that includes sections on [Custom Tags](http://java.sun.com/webservices/docs/ea1/tutorial/doc/JSPTags.html) and the [JSP Standard Tag Library (JSTL)](http://java.sun.com/webservices/docs/ea1/tutorial/doc/JSTL.html).[Tag Handler](http://jakarta.apache.org/taglibs/tutorial.html#tag_handler) server-side object that helps evaluate actions during the execution of a JSP page [Tag Library Descriptor](http://jakarta.apache.org/taglibs/tutorial.html#tag_library) xml file that defines tag names and tag attributes [JSP](http://jakarta.apache.org/taglibs/tutorial.html#tag_jsp) tag library is made available to the JSP page using the taglib directive [Installation and Deployment](http://jakarta.apache.org/taglibs/tutorial.html#tag_install) how to install a pre-existing tag library [Examples](http://jakarta.apache.org/taglibs/tutorial.html#tag_examples) code illustrations  |

|  |
| --- |
| **Tag Handler** |
| The Tag Handler is responsible for the interaction between the JSP page and additional server-side objects. The handler is invoked during the execution of a JSP page when a custom tag is encountered. The doStartTag() and doEndTag() methods are invoked when the start and end custom tags, respectively, are encountered. The release() method releases resources allocated by the tag handler.There are two interfaces that describe a tag handler:

|  |  |
| --- | --- |
| Tag | used for simple tag handlers not interested in manipulating their body content |
| BodyTag | an extension of Tag and gives the handler access to its body |

The Tag Handler has two main action methods:

|  |  |
| --- | --- |
| doStartTag() | process the start tag of this action. |
| doEndTag() | process the end tag of this action. Called after returning from doStartTag. |
| release() | release resources |

doStartTag() returns the following: * EVAL\_BODY\_INCLUDE
	+ process the body of the action but do not create a new BodyContent. Pass the body through without manipulating it. Only valid if you **DON'T** implement the BodyTag interface.
* EVAL\_BODY\_TAG
	+ process the body of the action and create a new BodyContent. Only valid if you **DO** implement the BodyTag interface.
* SKIP\_BODY
	+ do not evaluate the body of the tag

doEndTag() returns the following: * EVAL\_PAGE
	+ the rest of the JSP page will be evaluated
* SKIP\_PAGE
	+ the rest of the JSP page will not be evaluated

The return values direct the JSP container on how to evaluate the rest of the JSP page. The release() method releases resources allocated by the tag handler.TagSupport and BodyTagSupport are subclasses of Tag and can be used as base classes when creating new tag handlers. The TagSupport class is a utility class that implements the Tag interface and adds additional convenience methods including:* getter method for Tag properties

If the tag handler manipulates the body of an action, it must implement the BodyTag interface. doStartTag() must return EVAL\_BODY\_TAG in order for the body of the tag to be evaluated. If SKIP\_BODY is returned, the body will be ignored. Methods that interact with the body content include:

|  |  |
| --- | --- |
| doInitBody() | invoked before the body of the tag is evaluated but after body content is set |
| doAfterBody() | invoked after body content is evaluated |

The BodyTagSupport class implements the BodyTag interface and adds additional convenience methods. Some of these methods include:* getter for the bodyContent property
* getter for the previous out JSPWriter

In a Web Application handlers must reside in one of the following standard locations for Java classes:* in a JAR file in the /WEB-INF/lib directory
* in a directory in the /WEB-INF/classes directory

A tag handler has access to some properties that are set by the JSP container using setter methods. This includes the pageContext and parent objects. The tag handler also has access to server-side objects and enclosing actions. If the tag is nested, the parent handler of the enclosing tag can be accessed by using either:* TagSupport.getParent()
* TagSupport.findAncestorWithClass(from, class)

The parent handler's statically and dynamically created objects can be obtained once the parent object is retrieved. |

|  |
| --- |
| **Tag Library Descriptor** |
| The Tag Library Descriptor (TLD) is used by the JSP container to interpret pages that include the taglib directives referring to that tag library. It is an XML document that maps action tags to tag handler classes. You can locate a TLD in two ways:* web.xml taglib element
	+ taglib-uri
		- uri identifying a Tag Library
	+ taglib-location
		- location, as a resource, where the TLD file can be found
* default mapping

You can find more information about the web.xml taglib element in the Servlet 2.2 and JSP 1.1 specifications.You will need to explicitly reference the external DOCTYPE because of a recent change to call the validating parser:  <!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary\_1\_1.dtd">The TLD taglib element is the document root. It has the following subelements:

|  |  |
| --- | --- |
| tlibversion | version of the tag library implementation |
| jspversion | version of the JSP specification the tag library requires |
| shortname | name that could be used to reference the tag library from a JSP page |
| uri | uri uniquely identifying the tag library - info string describing the "use" of the tag library  |
| info | string describing the "use" of the tag library  |

The tag element defines an action in the tag library. It may have several subelements that define the action:

|  |  |
| --- | --- |
| name | unique action name |
| tagclass | tag handler class implementing javax.servlet.jsp.tagext.Tag |
| teiclass | optional subclass of javax.servlet.jsp.tagext.TagExtraInfo |
| bodycontent | one of three body content types |
| info | optional tag-specific information |
| attribute | all attributes of the action |

bodycontent is included if the tag has a body. It is used by page composition tools so it does not affect the composition of the body. It can be one of the three following types:* JSP (default)
	+ the JSP container should evaluate any body of the tag, but it can also be empty
* tagdependent
	+ any body of the tag would be handled by the tag itself, but it can also be empty
* empty
	+ body must be empty

the teiclass defines the scripting variable and includes the following information:* name
* type
* whether variable needs to be created or not
* scope

attributes can have the following fields:* name (required)
	+ attribute name
* required
	+ if attribute is required or optional
* rtexprvalue
	+ if attribute value may be dynamically calculated at runtime by a scriptlet. NOTE: default value is "false", meaning that the attribute has a static value. Make sure you set it to "true" if the attribute value is determined at request time.

For every attribute you must have a JavaBeans style get and set methods in the Tag Handler. If your attribute is named id, the TagSupport class defines the setId() and getId() methods for you. |

|  |
| --- |
| **JavaServer Pages** |
| JavaServer Pages can handle XML content encapsulated in Tag Library actions. <%@ taglib uri="identifier" prefix="prefix" %>To use a Tag Library, you need to tell the JSP container where it is located using a taglib directive. The directive must come **before** any actions.* The "identifier" will need to match the one used in <taglib-uri> in the web.xml file.
* The "prefix" distinguishes which tag library will be used.
 |

|  |
| --- |
| **Installation and Deployment** |
| a) Creating a Generic Tag Library To Install a tag library you need to take the following steps:1. bundle the tag classes in a jar file. Make sure to include the taglib {library}.tld file located in the /WEB-INF directory
2. add the tag {library}.jar file to the CLASSPATH
3. copy the {library}.jar file to the /WEB-INF/lib directory
4. define the taglib element in the /WEB-INF/web.xml file. For example:
5. <taglib>
6. <taglib-uri>http://jakarta.apache.org/taglibs/{library}</taglib-uri>
7. <taglib-location>/WEB-INF/{library}.tld</taglib-location>
8. </taglib>

 1. define the tag extension in the jsp page. The <taglib-uri> and the uri directive must match. The prefix identifies the tags in the tag library within the jsp page. For example:

 <%@ taglib uri="http://jakarta.apache.org/taglibs/{library}" prefix="x" %> b) Adding a Jakarta-Taglibs Library To add a tag library subproject to Jakarta-Taglibs you need to do the following:1. create a top level directory for the project.
2. copy the following top-level files from one of the existing subprojects:
	* build.sh
	* build.bat
	* build.xml
3. change the taglib.name property to the new custom tag library subproject name
4. duplicate the directory structure from one of the existing subprojects
5. modify the top-level Jakarta-Taglibs build.xml file to include the new library

c) Deploying a Tag LibraryUse the build scripts in the jakarta-taglibs project to create the war files. Once you have a war file built you can simply place that file in the $TOMCAT\_HOME/webapps directory. Tomcat will load your classes and create the new context.The war file should have the following structure: META-INF/ META-INF/MANIFEST.MF WEB-INF/ WEB-INF/classes/ WEB-INF/lib/ WEB-INF/lib/{tagLibrary}.jar WEB-INF/web.xml WEB-INF/{tagLibrary}.tldIf you do not want to use a jar file, you can place all the class files in the /WEB-INF/classes directory.Consult the [Java Servlet Specification, v2.2](http://java.sun.com/products/servlet/download.html) for more information on war files. |

|  |
| --- |
| **Examples** |
| Some examples include:* [Basic Tag](http://jakarta.apache.org/taglibs/tutorial.html#tag_basic)
* [Simple Nested Tag](http://jakarta.apache.org/taglibs/tutorial.html#tag_nested)
* [UtilityTags Library Documentation](http://jakarta.apache.org/taglibs/tutorial.html#tag_UtilityTags)
 |

|  |
| --- |
| **Basic Tag** |
| This basic tag is the "Hello World" example. The text "Hello World" will print whenever the tag is encountered.* [Tag Handler](http://jakarta.apache.org/taglibs/tutorial.html#basic_tag_handler)
* [Tag Library Descriptor](http://jakarta.apache.org/taglibs/tutorial.html#basic_tag_library)
* [web.xml file](http://jakarta.apache.org/taglibs/tutorial.html#basic_tag_webxml)
* [JSP](http://jakarta.apache.org/taglibs/tutorial.html#basic_tag_jsp)

Hello World Tag HandlerYou can find the Tag Handler for the Hello World tag in the /WEB-INF/classes/basic directory since it is a part of the basic package package basic;Import the jsp and tag classes: import javax.servlet.jsp.\*; import javax.servlet.jsp.tagext.\*;The Hello World Tag Handler implements the doStartTag() method which is invoked when the start tag is encountered. public int doStartTag() throws JspException { try { pageContext.getOut().print("Hello World"); } catch (Exception ex) { throw new JspException("IO problems"); } return SKIP\_BODY; }The pageContext is set by the JSP container and is available to the Tag Handler. The SKIP\_BODY value makes sure that no evaluation of the tag body takes place.Hello World Tag Library Descriptor <?xml version="1.0" encoding="ISO-8859-1" ?>XML header describing the deployment descriptor DOCTYPE. The deployment descriptor includes the elements and configuration information of a web application. <!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"  "http://java.sun.com/j2ee/dtds/web-jsptaglib\_1\_1.dtd">Initial taglibrary description <taglib> <!-- The version number of this tag library --> <tlibversion>1.0</tlibversion> <!-- The JSP specification version required to function --> <jspversion>1.1</jspversion> <!-- The short name of this tag library --> <shortname>utility</shortname> <!-- Public URI that uniquely identifies this version of the tag library --> <uri>http://jakarta.apache.org/taglibs/utilitytags</uri> <!-- General information about this tag library --> <info> A simple tag library for the examples </info>Hello World tag description.* tagclass element associates the Hello World tag handler with the Hello World tag
* bodycontent tag tells us that the tag will not contain any body

 <!-- Hello tag --> <tag> <name>Hello</name> <tagclass>basic.Hello</tagclass> <bodycontent>empty</bodycontent> <info> Print Hello World </info> </tag>web.xml fileThe web.xml file describes the mapping between the taglib uri and the location of the Tag Library Descriptor.Here the unique taglib-uri "http://jakarta.apache.org/taglibs/utilitytags" is associated with the Tag Library Descriptor in /WEB-INF/tld/utilitytags.tld. <web-app> <taglib> <taglib-uri> http://jakarta.apache.org/taglibs/utilitytags </taglib-uri> <taglib-location> /WEB-INF/tld/utilitytags.tld </taglib-location> </taglib> </web-app>Hello World jspThe following directive tells the JSP container to use the "http://jakarta.apache.org/taglibs/utilitytags" uri defined in web.xml. "jLib" is defined as the prefix value for the tag.  <%@ taglib uri="http://jakarta.apache.org/taglibs/utilitytags" prefix="jLib" %>The Hello World tag is called. The tag name "Hello" is defined in the the Tag Library Descriptor. <jLib:Hello/> |

|  |
| --- |
| **Simple Nested Action** |
| This nested tag is an example of an "If" conditional tag. Based on the value of the attribute the included scriptlet will be evaluated or skipped.* [Tag Handler](http://jakarta.apache.org/taglibs/tutorial.html#nested_tag_handler)
* [Tag Library Descriptor](http://jakarta.apache.org/taglibs/tutorial.html#nested_tag_library)
* [web.xml file](http://jakarta.apache.org/taglibs/tutorial.html#nested_tag_webxml)
* [JSP](http://jakarta.apache.org/taglibs/tutorial.html#nested_tag_jsp)

If Tag HandlerThe BodyTagSupport class implements the BodyTag interface and has getter methods for the bodyContent property.  public class IfTag extends BodyTagSupport {The doStartTag() method which is invoked when the start tag is encountered and calls the local getPredicate() method. If the return value is true, the rest of the the tag body is evaluated, otherwise it is skipped.  public int doStartTag() { if (getPredicate()) return EVAL\_BODY\_TAG; else return SKIP\_BODY; }doAfterBody() is called after some body has been evaluated. It is not invoked in empty tags or in tags returning SKIP\_BODY in doStartTag(). public int doAfterBody() throws JspException { try { bodyContent.writeOut(bodyContent.getEnclosingWriter()); return SKIP\_BODY; } catch (IOException ex) { throw new JspTagException(ex.toString()); } }If Tag Library Descriptor <!-- IF tag --> <tag> <name>If</name> <tagclass>lang.IfTag</tagclass>The If tag has one required attribute. Since the rtexprvalue is set to true, the attribute can have scriptlet expressions as a value. The value can be dynamically calculated at request time. <attribute> <name>predicate</name> <required>true</required> <rtexprvalue>true</rtexprvalue> </attribute> <info> Conditional Tag. </info> </tag> web.xml fileThe web.xml file describes the mapping between the taglib uri and the location of the Tag Library Descriptor.Here the unique taglib-uri "http://jakarta.apache.org/taglibs/utilitytags" is associated with the Tag Library Descriptor in /WEB-INF/tld/utilitytags.tld.If JSPThe If tag requires one attribute. The predicate attribute includes a scriptlet which will be evaluated at runtime. Based on the predicate attribute value, the jLib:Hello tag will be evaluated or skipped. <jlib:if predicate="<%= x==5 %>"> <jLib:Hello/>  </jlib:if> |

|  |
| --- |
| **Documentation for the UtilityTags Tag Library** |
| 1. INTRODUCTIONThe utilitytags custom tag library contains examples of some basic tags. It illustrates several straightforward custom tag library code techniques.2. PREREQUISITE SOFTWAREThis custom tag library requires no software other than a servlet container that supports the JavaServer Pages Specification, version 1.1.3. CONFIGURATION INFORMATIONFollow these steps to configure your web application with this tag library:* Copy the tag library descriptor file (utilitytags/utilitytags.tld) to the /WEB-INF subdirectory of your web application.
* Copy the tag library JAR file (utilitytags/utilitytags.jar) to the /WEB-INF/lib subdirectory of your web application.
* Add a <taglib> element to your web application deployment descriptor in /WEB-INF/web.xml like this:
* <taglib>
* <taglib-uri>http://jakarta.apache.org/taglibs/utilitytags</taglib-uri>
* <taglib-location>/WEB-INF/utilitytags.tld</taglib-location>
* </taglib>

 To use the tags from this library in your JSP pages, add the following directive at the top of each page: <%@ taglib uri="http://jakarta.apache.org/taglibs/utilitytags" prefix="x" %>where "x" is the tag name prefix you wish to use for tags from this library. You can change this value to any prefix you like. 4. TAG DOCUMENTATIONThe utilitytags Tag Library contains the following tags:Hello TagThe Hello tag prints out the text "Hello World". It does not have any attributes.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| - | - | - |

Copy TagThe MacroCopy tag copies the attribute text to a Writer.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| name | Name associated with the text to be copied. Any string value. | yes |

Paste TagThe MacroPaste tag pastes the text specified by a Writer.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| name | Name associated with the text to be pasted. Any string value. | yes |

ShowSource TagThe ShowSource tag takes a jspFile and copies the contents to a Writer.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| jspFile | The filename and relative path of the jsp file. Any string value. | yes |

Include TagThe Include tag includes in-line the output of the specified url.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| url | Any valid url. | yes |

If TagThe If tag is a basic conditional tag.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| predicate | Any string value. | yes |

For TagThe For tag is a basic looping tag.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| iterations | Number of loop iterations to be completed. Any string integer value. | yes |
| varName | Variable name associated with the For loop. Any string value. | no |
| begin | Loop starting value. Any string integer value. | no |

useBean TagThe useBean tag associates an instance of a Java object with the given id.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| id | Uniquely identifies the bean to the JSP container and page. Any string value. | yes |
| scope | page|request|session|application | no |
| classname | name of class that defines the implementation of the object. | no |
| type | type of the scripting variable defined | no |
| beanName | The name of the bean as expected by the instantiate() method of the java.beans.Beans class | yes |
| processRequest | true|false. JSP 0.92 compatibility. | no |

Validate TagThe Validate tag generates Javascript to validate the HTML form.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| name | Name of the form. Any string value. | yes |
| method | Name of the Javascript function to be generated. Any string value. | yes |
| reqdFields | Comma separated mandatory field list. Any string value. | yes |

 |

Q: - How does JSP engines instantiate tag handler classes instances?

Ans: -

## Defining Tags

To define a tag, you need to:

* Develop a tag handler and helper classes for the tag
* Declare the tag in a tag library descriptor

This section describes the properties of tag handlers and TLDs and explains how to develop tag handlers and library descriptor elements for each type of tag introduced in the previous section.

### Tag Handlers

A *tag handler* is an object invoked by a Web container to evaluate a custom tag during the execution of the JSP page that references the tag. Tag handlers must implement either the [Tag](http://java.sun.com/j2ee/tutorial/api/javax/servlet/jsp/tagext/Tag.html) or [BodyTag](http://java.sun.com/j2ee/tutorial/api/javax/servlet/jsp/tagext/BodyTag.html) interface. Interfaces can be used to take an existing Java object and make it a tag handler. For newly created handlers, you can use the [TagSupport](http://java.sun.com/j2ee/tutorial/api/javax/servlet/jsp/tagext/TagSupport.html) and [BodyTagSupport](http://java.sun.com/j2ee/tutorial/api/javax/servlet/jsp/tagext/BodyTagSupport.html) classes as base classes. These classes and interfaces are contained in the [javax.servlet.jsp.tagext](http://java.sun.com/j2ee/tutorial/api/javax/servlet/jsp/tagext/package-summary.html) package.

Tag handler methods defined by the Tag and BodyTag interfaces are called by the JSP page's servlet at various points during the evaluation of the tag. When the start tag of a custom tag is encountered, the JSP page's servlet calls methods to initialize the appropriate handler and then invokes the handler's doStartTag method. When the end tag of a custom tag is encountered, the handler's doEndTag method is invoked. Additional methods are invoked in between when a tag handler needs to interact with the body of the tag. For further information, see [How Is a Tag Handler Invoked?](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags7.html#68637). In order to provide a tag handler implementation, you must implement the methods, summarized in [Table 13-1](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#94002), that are invoked at various stages of processing the tag.

A tag handler has access to an API that allows it to communicate with the JSP page. The entry point to the API is the page context object ([javax.servlet.jsp.PageContext](http://java.sun.com/j2ee/tutorial/api/javax/servlet/jsp/PageContext.html)), through which a tag handler can retrieve all the other implicit objects (request, session, and application) accessible from a JSP page.

Implicit objects can have named attributes associated with them. Such attributes are accessed using [set|get]Attribute methods.

If the tag is nested, a tag handler also has access to the handler (called the *parent*) associated with the enclosing tag.

|  |
| --- |
| **Table 13-1 Tag Handler Methods** |
| **Tag Handler Type** | **Methods** |
| Simple | doStartTag, doEndTag, release |
| Attributes | doStartTag, doEndTag, set/getAttribute1...N, release |
| Body, evaluation and no interaction | doStartTag, doEndTag, release |
| Body, iterative evaluation | doStartTag, doAfterBody, doEndTag, release |
| Body, interaction | doStartTag, doEndTag, release, doInitBody, doAfterBody, release |

A set of related tag handler classes (a tag library) is usually packaged and deployed as a JAR archive.

### Tag Library Descriptors

A *tag library descriptor* (TLD) is an XML document that describes a tag library. A TLD contains information about a library as a whole and about each tag contained in the library. TLDs are used by a Web container to validate the tags and by JSP page development tools.

TLD file names must have the extension .tld. TLD files are stored in the WEB-INF directory of the WAR file or in a subdirectory of WEB-INF. When you add a TLD to a WAR using deploytool, it automatically puts it into WEB-INF.

A TLD must begin with an XML document prolog that specifies the version of XML and the document type definition (DTD):

<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag

Library 1.2//EN"

"http://java.sun.com/dtd/web-jsptaglibrary\_1\_2.dtd">

The J2EE SDK version 1.3 can understand version 1.1 and 1.2 DTDs. However, this chapter documents the 1.2 version because you should use the newer version in any tag libraries that you develop. The template library TLD, [tutorial-template.tld](http://java.sun.com/j2ee/tutorial/1_3-fcs/examples/src/web/bookstore3/tutorial-template.tld), conforms to the 1.2 version. The Struts library TLDs conform to the 1.1 version of the DTD, which has fewer elements and uses slightly different names for some of the elements.

The root of a TLD is the taglib element. The subelements of taglib are listed in [Table 13-2](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#75678):

|  |
| --- |
| **Table 13-2 taglib Subelements** |
| **Element** | **Description** |
| tlib-version | The tag library's version |
| jsp-version | The JSP specification version that the tag library requires |
| short-name | Optional name that could be used by a JSP page authoring tool to create names with a mnemonic value |
| uri | A URI that uniquely identifies the tag library |
| display-name | Optional name intended to be displayed by tools |
| small-icon | Optional small icon that can be used by tools |
| large-icon | Optional large icon that can be used by tools |
| description | Optional tag-specific information |
| listener | See [listener Element](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#74734) |
| tag | See [tag Element](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#74741) |

#### listener Element

A tag library can specify some classes that are event listeners (see [Handling Servlet Life-Cycle Events](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets4.html#75825)). The listeners are listed in the TLD as listener elements, and the Web container will instantiate the listener classes and register them in a way analogous to listeners defined at the WAR level. Unlike WAR-level listeners, the order in which the tag library listeners are registered is undefined. The only subelement of the listener element is the listener-class element, which must contain the fully qualified name of the listener class.

#### tag Element

Each tag in the library is described by giving its name and the class of its tag handler, information on the scripting variables created by the tag, and information on the tag's attributes. Scripting variable information can be given directly in the TLD or through a tag extra info class (see [Tags That Define Scripting Variables](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#68067)). Each attribute declaration contains an indication of whether the attribute is required, whether its value can be determined by request-time expressions, and the type of the attribute (see [Tags with Attributes](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#76223)).

A tag is specified in a TLD in a tag element. The subelements of tag are listed in [Table 13-3](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#75877):

|  |
| --- |
| **Table 13-3 tag Subelements** |
| **Element** | **Description** |
| name | The unique tag name. |
| tag-class | The fully-qualified name of the tag handler class. |
| tei-class | Optional subclass of javax.servlet.jsp.tagext.TagExtraInfo. See [TagExtraInfo Class](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#94122). |
| body-content | The body content type. See [body-content Element](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#67947) and [body-content Element](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#68059). |
| display-name | Optional name intended to be displayed by tools. |
| small-icon | Optional small-icon that can be used by tools. |
| large-icon | Optional large-icon that can be used by tools. |
| description | Optional tag-specific information. |
| variable | Optional scripting variable information. See [variable Element](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#76020). |
| attribute | Tag attribute information. See [attribute Element](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#67968). |

The following sections describe the methods and TLD elements that you need to develop for each type of tag introduced in [Using Tags](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags4.html#69766).

### Simple Tags

#### Tag Handlers

The handler for a simple tag must implement the doStartTag and doEndTag methods of the Tag interface. The doStartTag method is invoked when the start tag is encountered. This method returns SKIP\_BODY because a simple tag has no body. The doEndTag method is invoked when the end tag is encountered. The doEndTag method needs to return EVAL\_PAGE if the rest of the page needs to be evaluated; otherwise, it should return SKIP\_PAGE.

The simple tag discussed in the first section,

<tt:simple />

would be implemented by the following tag handler:

public SimpleTag extends TagSupport {

 public int doStartTag() throws JspException {

 try {

 pageContext.getOut().print("Hello.");

 } catch (Exception ex) {

 throw new JspTagException("SimpleTag: " +

 ex.getMessage());

 }

 return SKIP\_BODY;

 }

 public int doEndTag() {

 return EVAL\_PAGE;

 }

}

#### body-content Element

Tags without bodies must declare that their body content is empty using the body-content element:

<body-content>empty</body-content>

### Tags with Attributes

#### Defining Attributes in a Tag Handler

For each tag attribute, you must define a property and get and set methods that conform to the JavaBeans architecture conventions in the tag handler. For example, the tag handler for the Struts logic:present tag,

<logic:present parameter="Clear">

contains the following declaration and methods:

protected String parameter = null;

public String getParameter() {

 return (this.parameter);

}

public void setParameter(String parameter) {

 this.parameter = parameter;

}

Note that if your attribute is named id and your tag handler inherits from the TagSupport class, you do not need to define the property and set and get methods because these are already defined by TagSupport.

A tag attribute whose value is a String can name an attribute of one of the implicit objects available to tag handlers. An implicit object attribute would be accessed by passing the tag attribute value to the [set|get]Attribute method of the implicit object. This is a good way to pass scripting variable names to a tag handler where they are associated with objects stored in the page context (see [Tags That Define Scripting Variables](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#68067)).

#### attribute Element

For each tag attribute, you must specify whether the attribute is required, whether the value can be determined by an expression, and, optionally, the type of the attribute in an attribute element. For static values the type is always java.lang.String. If the rtexprvalue element is true or yes, then the type element defines the return type expected from any expression specified as the value of the attribute.

<attribute>

 <name>*attr1*</name>

 <required>true|false|yes|no</required>

 <rtexprvalue>true|false|yes|no</rtexprvalue>

 <type>*fully\_qualified\_type*</type>

</attribute>

If a tag attribute is not required, a tag handler should provide a default value.

The tag element for the logic:present tag declares that the parameter attribute is not required (because the tag can also test for the presence of other entities such as bean properties) and that its value can be set by a runtime expression.

<tag>

 <name>present</name>

 <tag-class>org.apache.struts.taglib.

 logic.PresentTag</tag-class>

 <body-content>JSP</body-content>

 ...

 <attribute>

 <name>parameter</name>

 <required>false</required>

 <rtexprvalue>true</rtexprvalue>

 </attribute>

 ...

</tag>

#### Attribute Validation

The documentation for a tag library should describe valid values for tag attributes. When a JSP page is translated, a Web container will enforce any constraints contained in the TLD element for each attribute.

The attributes passed to a tag can also be validated at translation time with the isValid method of a class derived from TagExtraInfo. This class is also used to provide information about scripting variables defined by the tag (see [Tags That Define Scripting Variables](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#68067)).

The isValid method is passed the attribute information in a TagData object, which contains attribute-value tuples for each of the tag's attributes. Since the validation occurs at translation time, the value of an attribute that is computed at request time will be set to TagData.REQUEST\_TIME\_VALUE.

The tag <tt:twa attr1="value1"/> has the following TLD attribute element:

<attribute>

 <name>attr1</name>

 <required>true</required>

 <rtexprvalue>true</a>

</attribute>

This declaration indicates that the value of attr1 can be determined at runtime.

The following isValid method checks that the value of attr1 is a valid Boolean value. Note that since the value of attr1 can be computed at runtime, isValid must check whether the tag user has chosen to provide a runtime value.

public class TwaTEI extends TagExtraInfo {

 public boolean isValid(Tagdata data) {

 Object o = data.getAttribute("attr1");

 if (o != null && o != TagData.REQUEST\_TIME\_VALUE) {

 if (o.toLowerCase().equals("true") ||

 o.toLowerCase().equals("false") )

 return true;

 else

 return false;

 }

 else

 return true;

 }

}

### Tags With Bodies

#### Tag Handlers

A tag handler for a tag with a body is implemented differently depending on whether the tag handler needs to interact with the body or not. By *interact*, we mean that the tag handler reads or modifies the contents of the body.

##### *Tag Handler Does Not Interact with the Body*

If the tag handler does not need to interact with the body, the tag handler should implement the Tag interface (or be derived from TagSupport). If the body of the tag needs to be evaluated, the doStartTag method needs to return EVAL\_BODY\_INCLUDE; otherwise, it should return SKIP\_BODY.

If a tag handler needs to iteratively evaluate the body, it should implement the IterationTag interface or be derived from TagSupport. It should return EVAL\_BODY\_AGAIN from the doStartTag and doAfterBody methods if it determines that the body needs to be evaluated again.

##### *Tag Handler Interacts with the Body*

If the tag handler needs to interact with the body, the tag handler must implement BodyTag (or be derived from BodyTagSupport). Such handlers typically implement the doInitBody and the doAfterBody methods. These methods interact with body content passed to the tag handler by the JSP page's servlet.

Body content supports several methods to read and write its contents. A tag handler can use the body content's getString or getReader methods to extract information from the body, and the writeOut(out) method to write the body contents to an out stream. The writer supplied to the writeOut method is obtained using the tag handler's getPreviousOut method. This method is used to ensure that a tag handler's results are available to an enclosing tag handler.

If the body of the tag needs to be evaluated, the doStartTag method needs to return EVAL\_BODY\_BUFFERED; otherwise, it should return SKIP\_BODY.

doInitBody **Method**

The doInitBody method is called after the body content is set but before it is evaluated. You generally use this method to perform any initialization that depends on the body content.

doAfterBody **Method**

The doAfterBody method is called *after* the body content is evaluated. Like the doStartTag method, doAfterBody must return an indication of whether to continue evaluating the body. Thus, if the body should be evaluated again, as would be the case if you were implementing an iteration tag, doAfterBody should return EVAL\_BODY\_BUFFERED; otherwise doAfterBody should return SKIP\_BODY.

release **Method**

A tag handler should reset its state and release any private resources in the release method.

The following example reads the content of the body (which contains a SQL query) and passes it to an object that executes the query. Since the body does not need to be reevaluated, doAfterBody returns SKIP\_BODY.

public class QueryTag extends BodyTagSupport {

 public int doAfterBody() throws JspTagException {

 BodyContent bc = getBodyContent();

 // get the bc as string

 String query = bc.getString();

 // clean up

 bc.clearBody();

 try {

 Statement stmt = connection.createStatement();

 result = stmt.executeQuery(query);

 } catch (SQLException e) {

 throw new JspTagException("QueryTag: " +

 e.getMessage());

 }

 return SKIP\_BODY;

 }

}

#### body-content Element

For tags that have a body, you must specify the type of the body content using the body-content element:

<body-content>JSP|tagdependent</body-content>

Body content containing custom and core tags, scripting elements, and HTML text is categorized as JSP. This is the value declared for the Struts logic:present tag. All other types of body content--for example, SQL statements passed to the query tag--would be labeled tagdependent.

Note that the value of the body-content element does not affect the interpretation of the body by the tag handler; the element is only intended to be used by an authoring tool for rendering the body content.

### Tags That Define Scripting Variables

#### Tag Handlers

A tag handler is responsible for creating and setting the object referred to by the scripting variable into a context accessible from the page. It does this by using the pageContext.setAttribute(name, value, scope) or pageContext.setAttribute(name, value) methods. Typically, an attribute passed to the custom tag specifies the name of the scripting variable object; this name can be retrieved by invoking the attribute's get method described in [Defining Attributes in a Tag Handler](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#67955).

If the value of the scripting variable is dependent on an object present in the tag handler's context, it can retrieve the object using the pageContext.getAttribute(name, scope) method.

The usual procedure is that the tag handler retrieves a scripting variable, performs some processing on the object, and then sets the scripting variable's value using the pageContext.setAttribute(name, object) method.

The scope that an object can have is summarized in [Table 13-4](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#68085). The scope constrains the accessibility and lifetime of the object.

|  |
| --- |
| **Table 13-4 Scope of Objects** |
| **Name** | **Accessible From** | **Lifetime** |
| page | Current page | Until the response has been sent back to the user or the request is passed to a new page |
| request | Current page and any included or forwarded pages | Until the response has been sent back to the user |
| session | Current request and any subsequent request from the same browser (subject to session lifetime) | The life of the user's session |
| application | Current and any future request from the same Web application | The life of the application |

#### Providing Information about the Scripting Variable

The example described in [Tags That Define Scripting Variables](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags4.html#67842) defines a scripting variable book that is used for accessing book information:

<bean:define id="book" name="bookDB" property="bookDetails"

 type="database.BookDetails"/>

<font color="red" size="+2">

 <%=messages.getString("CartRemoved")%>

 <strong><jsp:getProperty name="book"

 property="title"/></strong>

<br>&nbsp;<br>

</font>

When the JSP page containing this tag is translated, the Web container generates code to synchronize the scripting variable with the object referenced by the variable. To generate the code, the Web container requires certain information about the scripting variable:

* Variable name
* Variable class
* Whether the variable refers to a new or existing object
* The availability of the variable

There are two ways to provide this information: by specifying the variable TLD subelement or by defining a tag extra info class and including the tei-class element in the TLD. Using the variable element is simpler, but slightly less flexible.

##### *variable Element*

The variable element has the following subelements:

* name-given: The variable name as a constant.
* name-from-attribute: The name of an attribute whose translation-time value will give the name of the variable.

One of name-given or name-from-attribute is required. The following subelements are optional:

* variable-class: The fully qualified name of the class of the variable. java.lang.String is the default.
* declare: Whether the variable refers to a new object. True is the default.
* scope: The scope of the scripting variable defined. NESTED is default. [Table 13-5](http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags5.html#94133) describes the availability of the scripting variable and the methods in which the value of the variable must be set or reset.

The implementation of the Struts bean:define tag conforms to the JSP specification version 1.1, which requires you to define a tag extra info class. The JSP specification version 1.2 adds the variable element. You could define the following variable element for the bean:define tag:

<tag>

 <variable>

 <name-from-attribute>id</name-from-attribute>

 <variable-class>database.BookDetails</variable-class>

 <declare>true</declare>

 <scope>AT\_BEGIN</scope>

 </variable>

</tag>

|  |
| --- |
| **Table 13-5 Scripting Variable Availability** |
| **Value**  | **Availability**  | **Methods**  |
| NESTED  | Between the start tag and the end tag  | In doInitBody and doAfterBody for a tag handler implementing BodyTag; otherwise, in doStartTag  |
| AT\_BEGIN  | From the start tag until the end of the page  | In doInitBody, doAfterBody, and doEndTag for a tag handler implementing BodyTag; otherwise, in doStartTag and doEndTag  |
| AT\_END  | After the end tag until the end of the page  | In doEndTag  |

##### *TagExtraInfo Class*

You define a tag extra info class by extending the class javax.servlet.jsp.TagExtraInfo. A TagExtraInfo must implement the getVariableInfo method to return an array of VariableInfo objects containing the following information:

* Variable name
* Variable class
* Whether the variable refers to a new object
* The availability of the variable

The Web container passes a parameter called data to the getVariableInfo method that contains attribute-value tuples for each of the tag's attributes. These attributes can be used to provide the VariableInfo object with a scripting variable's name and class.

The Struts tag library provides information about the scripting variable created by the bean:define tag in the DefineTei tag extra info class. Since the name (book) and class (database.BookDetails) of the scripting variable are passed in as tag attributes, they can be retrieved with the data.getAttributeString method and used to fill in the VariableInfo constructor. To allow the scripting variable book to be used in the rest of the page, the scope of book is set to be AT\_BEGIN.

public class DefineTei extends TagExtraInfo {

 public VariableInfo[] getVariableInfo(TagData data) {

 String type = data.getAttributeString("type");

 if (type == null)

 type = "java.lang.Object";

 return new VariableInfo[] {

 new VariableInfo(data.getAttributeString("id"),

 type,

 true,

 VariableInfo.AT\_BEGIN)

 };

 }

}

The fully qualified name of the tag extra info class defined for a scripting variable must be declared in the TLD in the tei-class subelement of the tag element. Thus, the tei-class element for DefineTei would be as follows:

<tei-class>org.apache.struts.taglib.bean.DefineTagTei

</tei-class>

### Cooperating Tags

Tags cooperate by sharing objects. JSP technology supports two styles of object sharing. The first style requires that a shared object be named and stored in the page context (one of the implicit objects accessible to both JSP pages and tag handlers). To access objects created and named by another tag, a tag handler uses the pageContext.getAttribute(name, scope) method.

In the second style of object sharing, an object created by the enclosing tag handler of a group of nested tags is available to all inner tag handlers. This form of object sharing has the advantage that it uses a private namespace for the objects, thus reducing the potential for naming conflicts.

To access an object created by an enclosing tag, a tag handler must first obtain its enclosing tag with the static method TagSupport.findAncestorWithClass(from, class) or the TagSupport.getParent method. The former method should be used when a specific nesting of tag handlers cannot be guaranteed. Once the ancestor has been retrieved, a tag handler can access any statically or dynamically created objects. Statically created objects are members of the parent. Private objects can also be created dynamically. Such objects can be stored in a tag handler with the setValue method and retrieved with the getValue method.

The following example illustrates a tag handler that supports both the named and private object approaches to sharing objects. In the example, the handler for a query tag checks whether an attribute named connection has been set in the doStartTag method. If the connection attribute has been set, the handler retrieves the connection object from the page context. Otherwise, the tag handler first retrieves the tag handler for the enclosing tag and then retrieves the connection object from that handler.

public class QueryTag extends BodyTagSupport {

 private String connectionId;

 public int doStartTag() throws JspException {

 String cid = getConnection();

 if (cid != null) {

 // there is a connection id, use it

 connection =(Connection)pageContext.

 getAttribute(cid);

 } else {

 ConnectionTag ancestorTag =

 (ConnectionTag)findAncestorWithClass(this,

 ConnectionTag.class);

 if (ancestorTag == null) {

 throw new JspTagException("A query without

 a connection attribute must be nested

 within a connection tag.");

 }

 connection = ancestorTag.getConnection();

 }

 }

}

The query tag implemented by this tag handler could be used in either of the following ways:

<tt:connection id="con01" ....> ... </tt:connection>

<tt:query id="balances" connection="con01">

 SELECT account, balance FROM acct\_table

 where customer\_number = <%= request.getCustno()%>

</tt:query>

<tt:connection ...>

 <x:query id="balances">

 SELECT account, balance FROM acct\_table

 where customer\_number = <%= request.getCustno()%>

 </x:query>

</tt:connection>

The TLD for the tag handler must indicate that the connection attribute is optional with the following declaration:

<tag>

 ...

 <attribute>

 <name>connection</name>

 <required>false</required>

 </attribute>

</tag>

Q: - What's the difference between JavaBeans and taglib directives?

Ans: -

## <jsp:useBean>

Locates or instantiates a Bean with a specific name and scope.

#### JSP Syntax

<jsp:useBean
        id="beanInstanceName"
        scope="**page** | request | session | application"
        {
            class="package.class" |
            type="package.class" |
            class="package.class" type="package.class" |
            beanName="{package.class | <%= expression %>}" type="package.class"
        }
        {
            /> |
            > other elements </jsp:useBean>
        }

#### Examples

<jsp:useBean id="cart" scope="session" class="session.Carts" />
<jsp:setProperty name="cart" property="\*" />

<jsp:useBean id="checking" scope="session" class="bank.Checking" >
<jsp:setProperty name="checking" property="balance" value="0.0" />
</jsp:useBean>

#### Description

The <jsp:useBean> element locates or instantiates a JavaBeans component. <jsp:useBean> first attempts to locate an instance of the Bean. If the Bean does not exist, <jsp:useBean> instantiates it from a class or serialized template.

To locate or instantiate the Bean, <jsp:useBean> takes the following steps, in this order:

1. Attempts to locate a Bean with the scope and name you specify.
2. Defines an object reference variable with the name you specify.
3. If it finds the Bean, stores a reference to it in the variable. If you specified type, gives the Bean that type.
4. If it does not find the Bean, instantiates it from the class you specify, storing a reference to it in the new variable. If the class name represents a serialized template, the Bean is instantiated by java.beans.Beans.instantiate.
5. If <jsp:useBean> has instantiated (rather than located) the Bean, and if it has body tags or elements (between <jsp:useBean> and </jsp:useBean>), executes the body tags.

The body of a <jsp:useBean> element often contains a <jsp:setProperty> element that sets property values in the Bean. As described in [Step 5](http://java.sun.com/products/jsp/tags/11/syntaxref11.fm14.html#9843), the body tags are only processed if <jsp:useBean> instantiates the Bean. If the Bean already exists and <jsp:useBean> locates it, the body tags have no effect.

In this release, you can use a <jsp:useBean> element to locate or instantiate a Bean, but not an enterprise bean. To create enterprise beans, you can write a <jsp:useBean> element that calls a Bean that in turn calls the enterprise bean, or you can write a custom tag that calls an enterprise bean directly.

#### Attributes and Usage

* id="beanInstanceName"

A variable that identifies the Bean in the scope you specify. You can use the variable name in expressions or scriptlets in the JSP file.

The name is case sensitive and must conform to the naming conventions of the scripting language used in the JSP page. If you use the Java programming language, the conventions in the Java Language Specification. If the Bean has already been created by another <jsp:useBean> element, the value of id must match the value of id used in the original <jsp:useBean> element.

* scope="**page** | request | session | application"

The scope in which the Bean exists and the variable named in id is available. The default value is page. The meanings of the different scopes are shown below:

* + page - You can use the Bean within the JSP page with the <jsp:useBean> element or any of the page's static include files, until the page sends a response back to the client or forwards a request to another file.
	+ request - You can use the Bean from any JSP page processing the same request, until a JSP page sends a response to the client or forwards the request to another file. You can use the request object to access the Bean, for example, request.getAttribute(beanInstanceName).
	+ session - You can use the Bean from any JSP page in the same session as the JSP page that created the Bean. The Bean exists across the entire session, and any page that participates in the session can use it. The page in which you create the Bean must have a <%@ page %> directive with session=true.
	+ application - You can use the Bean from any JSP page in the same application as the JSP page that created the Bean. The Bean exists across an entire JSP application, and any page in the application can use the Bean.
* class="package.class"

Instantiates a Bean from a class, using the new keyword and the class constructor. The class must not be abstract and must have a public, no-argument constructor. The package and class name are case sensitive.

* type="package.class"

If the Bean already exists in the scope, gives the Bean a data type other than the class from which it was instantiated. If you use type without class or beanName, no Bean is instantiated. The package and class name are case sensitive.

* class="package.class" type="package.class"

Instantiates a Bean from the class named in class and assigns the Bean the data type you specify in type. The value of type can be the same as class, a superclass of class, or an interface implemented by class.

The class you specify in class must not be abstract and must have a public, no-argument constructor. The package and class names you use with both class and type are case sensitive.

* beanName="{package.class | <%= expression %>}" type="package.class"

Instantiates a Bean from either a class or a serialized template, using the java.beans.Beans.instantiate method, and gives the Bean the type specified in type. The Beans.instantiate method checks whether a name represents a class or a serialized template. If the Bean is serialized, Beans.instantiate reads the serialized form (with a name like package.class.ser) using a class loader. For more information, see the JavaBeans API Specification.

The value of beanName is either a package and class name or an [Expression](http://java.sun.com/products/jsp/tags/11/syntaxref11.fm4.html#11258) that evaluates to a package and class name, and is passed to Beans.instantiate. The value of type can be the same as beanName, a superclass of beanName, or an interface implemented by beanName.

The package and class names you use with both beanName and type are case sensitive.

## Taglib Directive

Defines a tag library and prefix for the custom tags used in the JSP page.

#### JSP Syntax

<%@ taglib uri="URIToTagLibrary" prefix="tagPrefix" %>

#### Examples

<%@ taglib uri="http://www.jspcentral.com/tags" prefix="public" %>
<public:loop>
.
.
</public:loop>

#### Description

The <%@ taglib %> directive declares that the JSP file uses custom tags, names the tag library that defines them, and specifies their tag prefix.

Here, the term custom tag refers to both tags and elements. Because JSP files can be converted to XML, it is important to understand the relationship of tags and elements. A tag is simply a short piece of markup that is part of a JSP element. A JSP element is a unit of JSP syntax that has an XML equivalent with a start tag and an end tag. An element can also contain other text, tags, or elements. For example, a jsp:plugin element always has a <jsp:plugin> start tag and a </jsp:plugin> end tag and may have a <jsp:params> element and a <jsp:fallback> element.

You must use a <%@ taglib %> directive before you use the custom tag in a JSP file. You can use more than one <%@ taglib %> directive in a JSP file, but the prefix defined in each must be unique.

The technique for creating custom tags is described in the JavaServer Pages Specification for version 1.1.

#### Attributes

* uri="URIToTagLibrary"

The Uniform Resource Identifier (URI) that uniquely names the set of custom tags associated with the named tag prefix. A URI can be any of the following:

* + A Uniform Resource Locator (URL), as defined in RFC 2396, available at http://www.hut.fi/u/jkorpela/rfc/2396/full.html
	+ A Uniform Resource Name (URN), as defined in RFC 2396
	+ An absolute or relative pathname
* prefix="tagPrefix"

The prefix that precedes the custom tag name, for example, public in <public:loop>. Empty prefixes are illegal. If you are developing or using custom tags, you cannot use the tag prefixes jsp, jspx, java, javax, servlet, sun, and sunw, as they are reserved by Sun Microsystems.

Q: - What are the different scopes an object can have in a JSP page?

Ans: -

## Object Scopes

When a JSP page needs to save **data** for its processing, it must specify a location,called the ***scope*** or ***visibility***.

There are four scopes available - ***page***, ***request***, ***session***, and ***application***.

The following diagram indicates the various scopes that can be associated with a newly created **data** :

|  |  |
| --- | --- |
| **application** | Objects accesible from pages belong to the same application |
| **session** | Objects accessible from pages belonging to the same session as the one in which theye were created |
| **request** | Object accessible from pages processing the request where they were created |
| **page** | Objects accessible only within pages where they were created |

***Page*** - scoped data is accessible only within the JSP page - where the data was created - and it is destroyed

when the page has finished.

***Request***-scoped data is associated with the request and destroyed when the request is completed.

***Session***-scoped data is associated with a session and destroyed when the session is destroyed.

***Application***-scoped data is associated with the web application and destroyed when the web application is destroyed.

*Application*-*scoped* data is not accessible to other web applications.

###

### Using Objects(data) within JSP Pages

You can access a variety of objects or data, including enterprise beans and JavaBeans components, within a JSP page. JSP technology automatically makes some objects available, and you can also create and access application-specific objects.

These **Objects**may be created implicitly using **JSP directives**, explicitly through **actions**, or, in rare cases, directly using **scripting code**. The instantiated **object** can be associated with a ***scope attribute*** defining where there is a reference to the object and when that reference is removed.

**JSP Implicit Objects**

The ***JSP container*** makes available nine implicit **objects** that can be used within **scriptlets** and **expressions**, to create or save data to use within the JSP's. These objects act as **wrappers** around underlying Java classes or interfaces typically defined within the Servlet API.

|  |  |  |  |
| --- | --- | --- | --- |
| **Objects** | **Scope** | **Class** | **Servlet description** |
| **application** | Application  | [javax.servlet.ServletContext](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/ServletContext.html)  | The context for the JSP page's servlet and any Web components contained in the same application. |
| **session** | Session | [javax.servlet.http.HttpSession](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpSession.html) | The session object for the client. See [Maintaining Client State](http://java.sun.com/j2ee/1.3/docs/tutorial/doc/Servlets11.html#wp64744).  |
| **request** | Request | subtype of [javax.servlet.ServletRequest](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/ServletRequest.html) | Represents the HttpServletRequest triggering the service invocation. The request triggering the execution of the JSP page. For example: *request.getParameter("Add");* |
| **response** | Page | subtype of [javax.servlet.ServletResponse](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/ServletResponse.html) | The response to be returned to the client. Not typically used by JSP page authors. |
| **pageContext** | Page | [javax.servlet.jsp.PageContext](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/jsp/PageContext.html) | The context for the JSP page. Provides a single API to manage the various **scoped objects** described before. Encapsulates implementation-dependent features in PageContext. |
| **out** | Page | [javax.servlet.jsp.JspWriter](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/jsp/JspWriter.html)  | A JspWriter object that writes into the output stream. |
| **config** | Page | [javax.servlet.ServletConfig](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/ServletConfig.html) | Initialization information for the JSP page's servlet.  |
| **page** | Page | javax.jsp.HttpJspPage | Synonym for the "this" operator, as an HttpJspPage. Not used often. The instance of the JSP page's servlet processing the current request. |
| **exception** | Page | [java.lang.Throwable](http://java.sun.com/products/jdk/1.3.1/docs/api/java/lang/Throwable.html)  | The uncaught Throwable object that resulted in the error page being invoked. Accessible only from an error page. See [Handling Errors](http://java.sun.com/j2ee/1.3/docs/tutorial/doc/JSPIntro4.html#wp65997).  |

For Example:

|  |  |
| --- | --- |
| **JSTL**  | **JSP 1.x** |
| pageScope.*name* | **pageContext**.getAttribute("*name*") |
| requestScope.*name* | **request**.getAttribute("*name*") |
| sessionScope.*name* | **session**.getAttribute("*name*") |
| applicationScope.*name* | **application**.getAttribute("*name*") |
| param.*name* | **request**.getParameter("*name*") |
| initParam.*name* | **application**.getInitParameter("*name*") |
| *bean*.*name* | *bean*.get*Name*() |
| *bean*.*subBean*.*name* | *bean*.get*SubBean*().get*Name*() |

#### Application-Specific Objects

When possible, application behavior should be encapsulated in objects so that page designers can focus on presentation issues. Objects can be created by developers who are proficient in the Java programming language and in accessing databases and other services. There are four ways to create and use objects within a JSP page:

* Instance and class variables of the JSP page's servlet class are created in declarations and accessed in scriptlets and expressions.
* Local variables of the JSP page's servlet class are created and used in scriptlets and expressions.
* Attributes of scope objects see Objects Scope are created and used in scriptlets and expressions.
* JavaBeans components can be created and accessed using streamlined JSP elements. These elements are discussed in [JavaBeans Components in JSP Pages](http://java.sun.com/j2ee/1.3/docs/tutorial/doc/JSPIntro11.html#wp70915). You can also create a JavaBeans component in a declaration or scriptlet and invoke the methods of a JavaBeans component in a scriptlet or expression.

Declarations, scriptlets, and expressions are described in [**JSP Scripting Elements**](http://www.jprofil.de/Dokumentation/J2EE/JSP/scopeVariables/scopeVariables.htm#JSP_Scripting_Elements)

#### Shared Objects

The conditions affecting concurrent access to shared objects described in [Controlling Concurrent Access to Shared Resources](http://java.sun.com/j2ee/1.3/docs/tutorial/doc/Servlets5.html#wp64386) apply to objects accessed from JSP pages that run as multithreaded servlets. You can indicate how a Web container should dispatch multiple client requests with the following page directive:

<%@ page isThreadSafe="true|false" %>

When isThreadSafe is set to true, the Web container may choose to dispatch multiple concurrent client requests to the JSP page. This is the default setting. If using true, you must ensure that you properly synchronize access to any shared objects defined at the page level. This includes objects created within declarations, JavaBeans components with page scope, and attributes of the page scope object.

If isThreadSafe is set to false, requests are dispatched one at a time, in the order they were received, and access to page level objects does not have to be controlled. However, you still must ensure that access to attributes of the application or session scope objects and to JavaBeans components with application or session scope is properly synchronized.

----------------------------------Counter.jsp---------------------------------------------------------------------------

 <%@ page import="com.jguru.CounterBean" %>

<jsp:useBean id="session\_counter" class="com.jguru.CounterBean" scope="session" />

<jsp:useBean id="app\_counter" class="com.jguru.CounterBean" scope="application" />

<% session\_counter.increaseCount();

 synchronized(page) {

 app\_counter.increaseCount();

 }

%>

<h3>

Number of accesses within this session:

<jsp:getProperty name="session\_counter" property="count" />

</h3>

<p>

<h3>

Total number of accesses:

<% synchronized(page) { %>

<jsp:getProperty name="app\_counter" property="count" />

<% } %>

</h3>

-----------------------------------CounterBean.java--------------------------

package com.jguru;

public class CounterBean {

 int count;

 public int getCount() {

 return count;

 }

 public void increaseCount() {

 count++;

 }

}

Q: - What are different implicit objects of JSP?

Ans: -

Certain objects that are available for the use in JSP documents without being declared first. These objects are parsed by the JSP engine and inserted into the generated servlet. The implicit objects re listed below

|  |
| --- |
| * request
* response
* pageContext
* session
* application
* out
* config
* page
* exception
 |

Implicit Objects in JSP are objects that are automatically available in JSP. Implicit Objects are Java objects that the JSP Container provides to a developer to access them in their program using JavaBeans and Servlets. These objects are called implicit objects because they are automatically instantiated.

There are many implicit objects available. Some of them are:

**request:** The class or the interface name of the object request is http.httpservletrequest. The object request is of type Javax.servlet.http.httpservletrequest. This denotes the data included with the HTTP Request. The client first makes a request that is then passed to the server. The requested object is used to take the value from client’s web browser and pass it to the server. This is performed using HTTP request like headers, cookies and arguments.

**response:** This denotes the HTTP Response data. The result or the information from a request is denoted by this object. This is in contrast to the request object. The class or the interface name of the object response is http.HttpServletResponse. The object response is of type Javax.servlet.http. >httpservletresponse. Generally, the object response is used with cookies. The response object is also used with HTTP Headers.

**Session:** This denotes the data associated with a specific session of user. The class or the interface name of the object Session is http.HttpSession. The object Session is of type Javax.servlet.http.httpsession. The previous two objects, request and response, are used to pass information from web browser to server and from server to web browser respectively. The Session Object provides the connection or association between the client and the server. The main use of Session Objects is for maintaining states when there are multiple page requests. This will be explained in further detail in following sections.

**Out:** This denotes the Output stream in the context of page. The class or the interface name of the Out object is jsp.JspWriter. The Out object is written: Javax.servlet.jsp.JspWriter

**PageContext:** This is used to access page attributes and also to access all the namespaces associated with a JSP page. The class or the interface name of the object PageContext is jsp.pageContext. The object PageContext is written: Javax.servlet.jsp.pagecontext

**Application:** This is used to share the data with all application pages. The class or the interface name of the Application object is ServletContext. The Application object is written: Javax.servlet.http.ServletContext

**Config:** This is used to get information regarding the Servlet configuration, stored in the Config object. The class or the interface name of the Config object is ServletConfig. The object Config is written Javax.servlet.http.ServletConfig

**Page:** The Page object denotes the JSP page, used for calling any instance of a Page's servlet. The class or the interface name of the Page object is jsp.HttpJspPage. The Page object is written: Java.lang.Object

The most commonly used implicit objects are request, response and session objects

Q: - What are different Authentication Options available in servlets?

Ans: -

Q: - Can you explain how do we practically implement security on a resource?

Ans: -

com.sun.identity.policy.jaas**.**Class ISPermission

java.lang.Object

 java.security.Permission

 **com.sun.identity.policy.jaas.ISPermission**

**All Implemented Interfaces:**

java.security.Guard, java.io.Serializable

public class **ISPermission**

extends java.security.Permission

This class provides the support for JAAS Authorization service Its a new JAAS Permission which extends the Permission class. This is the only API which gets used by an application/container to evaluate policy against the Access Manager Policy framework. This class provides implementations of all the required abstract methods of java.security.Permission, in a way that the policy evaluation is made against the Access Manager's Policy service.

For example, one would use this class as follows to evaluate policy permissions:

 ISPermission perm = new ISPermission("iPlanetAMWebAgentService",

 "http://www.sun.com:80","GET");

 AccessController.checkPermission(perm);

If Access Manager has the policy service iPlanetAMWebAgentService which has a Rule defined for resource http://www.sun.com:80 with action "GET" with allow privilege, this call will return quietly, if such a policy is not found then access is denied and Exception thrown accordingly. Also these ISPermission co-exist with the permissions specified in the JDK policy store ( by default file com.sun.security.auth.PolicyFile or defined on the command line using the -D option.

**See Also:**

Permission, Subject,

, [Serialized Form](http://docs.sun.com/source/819-4682/serialized-form.html#com.sun.identity.policy.jaas.ISPermission)

|  |
| --- |
| **Constructor Summary** |
| [**ISPermission**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#ISPermission%28java.lang.String,%20java.lang.String,%20java.lang.String%29)(java.lang.String serviceName, java.lang.String resourceName, java.lang.String actions)           Constructs an ISPermission instance, with the specified service name, resource name and action name. |  |
| [**ISPermission**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#ISPermission%28java.lang.String,%20java.lang.String,%20java.lang.String,%20java.util.Map%29)(java.lang.String serviceName, java.lang.String resourceName, java.lang.String actions, java.util.Map envParams)           Constructs an ISPermission instance, with the specified service name, resource name and action name. |  |
| [**ISPermission**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#ISPermission%28javax.security.auth.Subject,%20java.security.CodeSource%29)(javax.security.auth.Subject subject, java.security.CodeSource codesource)           Constructs an ISPermission instance, with the specified Subject and the CodeSource. |  |

|  |
| --- |
| **Method Summary** |
|  boolean | [**equals**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#equals%28java.lang.Object%29)(java.lang.Object obj)           Returns true if two ISPermission objects for equality. |
|  java.lang.String | [**getActions**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#getActions%28%29)()           returns a comma separated list of actions associated with this ISPermission. |
|  java.security.CodeSource | [**getCodeSource**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#getCodeSource%28%29)()           returns the CodeSourceassociated with this ISPermission. |
|  java.util.Map | [**getEnvParams**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#getEnvParams%28%29)()           returns environment parameters and their values associated with this ISPermission. |
|  java.lang.String | [**getResourceName**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#getResourceName%28%29)()           returns the name of the resource associated with this ISPermission . |
|  java.lang.String | [**getServiceName**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#getServiceName%28%29)()           returns the name of the service associated with this ISPermission . |
|  javax.security.auth.Subject | [**getSubject**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#getSubject%28%29)()           returns the Subjectassociated with this ISPermission . |
|  int | [**hashCode**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#hashCode%28%29)()           Returns the hash code value for this Permission object. |
|  boolean | [**implies**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#implies%28java.security.Permission%29)(java.security.Permission perm)           Checks if the specified permission's actions are "implied by" this object's actions. |
|  java.security.PermissionCollection | [**newPermissionCollection**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#newPermissionCollection%28%29)()           Returns a java.security.PermissionCollection to store this kind of Permission. |
|  java.lang.String | [**toString**](http://docs.sun.com/source/819-4682/com/sun/identity/policy/jaas/ISPermission.html#toString%28%29)()           Returns a string describing this Permission. |

Q: - How do we practically implement form based authentication?

Ans: -

### Use forms to authenticate clients

A common way for servlet-based systems to perform authentication is to use the session to store information indicating that a user has logged into the system. In this scheme, the authentication logic uses the HttpSession object maintained by the servlet engine in the Web server.

A base servlet with knowledge of authentication is helpful in this case. Using the service method of the BaseServlet, the extending servlets can reuse the security check functionality. All code used in this example can be found in [Resources](http://www.javaworld.com/javaworld/jw-04-2000/jw-0428-websecurity.html?page=2#resources).

The service method is shown in the following code snippet:

 public void service(HttpServletRequest request, HttpServletResponse

response)

 throws IOException, ServletException

 {

 // check to see if a session has already been created for this user

 // don't create a new session yet.

 HttpSession session = request.getSession( false);

 String requestedPage = request.getParameter(Constants.REQUEST);

 if ( session != null)

 {

 // retrieve authentication parameter from the session

 Boolean isAuthenticated = (Boolean)

session.getValue(Constants.AUTHENTICATION);

 // if the user is not authenticated

 if ( !isAuthenticated.booleanValue() )

 {

 // process the unauthenticated request

 unauthenticatedUser(response, requestedPage);

 }

 }

 else // the session does not exist

 {

 // therefore the user is not authenticated

 // process the unauthenticated request

 unauthenticatedUser(response, requestedPage);

 }

 }

Notice that you can expand this method to perform other generic functions as well. In this example, I developed only the security aspects of this class.

The BaseServlet attempts to retrieve the session from the servlet engine. On retrieval, the servlet verifies that the user has been granted access to the system. Should either of these checks fail, the servlet redirects the browser to the login screen.

On the login screen, the user is prompted for a username and password. Note that the data passed from the browser to the Web server is unencrypted unless you use Secure Socket Layer (SSL).

The LoginServlet uses the username/password combination to query the database to ensure that this user does indeed have access to the system. If the check fails to return a record for that user, the login screen is redisplayed. If the check is successful, the following code stores the user authentication information inside a session variable.

 // create a session

 session = request.getSession( true);

 // convert the boolean to a Boolean

 Boolean booleanIsAuthenticated = new Boolean( isAuthenticated);

 // store the boolean value to the session

 session.putValue(

 Constants.AUTHENTICATION,

 booleanIsAuthenticated);

In this example, it's assumed that any user who successfully authenticates to the system has access to the pages displayed prior to login. However, there are cases in which the application development team may require a more refined security approach to satisfy its requirements.

Q: - How do we authenticate using JDBC?

Ans: -

This section describes the authentication framework of Access Manager, the encapsulated [Java Authentication and Authorization Service (JAAS)](http://java.sun.com/products/jaas/overview.html) API, and their benefits.

**Authentication Framework**

The authentication framework in Access Manager is based on JAAS. Specifically, Access Manager

* Implements the authentication portion of JAAS-JAAS PAM modules;
* Allows the server and the PAM modules it exposes to be interfaced through JAAS authentication APIs; and by the way,
* Supports JAAS authorization wrappers around the policy framework.

You can wrap proprietary authentication interfaces around a custom authentication module and then plug that module into the Access Manager authentication framework from the Administration Console. Afterwards, Access Manager can manage the new module just as it does other modules on Access Manager, such as the Lightweight Directory Access Protocol (LDAP), UNIX, and RADIUS. You adopt the new module through the authentication service just like other preinstalled module service programming interfaces (SPIs).

Customizing the authentication module involves two major steps:

1. **Implementing an interface by applying the package com.iplanet.authentication.spi.**

That package contains a series of APIs for creating authentication services--a process that requires the implementation of the JAAS LoginModule and methods for accessing the authentication service and module configuration files. As soon as that architecture is in place, the authentication framework can manage all custom JAAS authentication modules.
2. **Adding the module to the list of available authentication modules.**

**Encapsulated (Simplified) JAAS API**

Although the authentication framework supports pure JAAS pluggable authentication modules, it's simpler and more straightforward to customize the authentication module with the Access Manager authentication SPI. As a rule, you would implement the following four classes:

* **Class 1** -- For extending the javax.security.auth.login.Configuration class
* **Class 2** -- For implementing the javax.security.auth.spi.LoginModule interface
* **Class 3** -- For implementing the java.security.Principal interface
* **Class 4** -- For implementing the java.security.PrivilegedAction interface

But to develop pluggable authentication modules with the Access Manager authentication SPI, you can significantly save development cycles because you need implement only two classes:

* One class for implementing com.sun.identity.authentication.spi.AMLoginModule
* One class for implementing the java.security.Principal interface

**Benefits**

The authentication framework offers diverse authentication benefits for a variety of applications:

* For Web applications, the authentication service simplifies the client-side authentication logic by performing almost all authentication tasks.
* For stand-alone applications written in either C or the Java programming language, the authentication service offers authentication APIs.
* For applications written in languages other than Java or C, the authentication service offers the XML interface. By employing the URL http://*server\_name.domain\_name:port\_number/service\_deploy\_URI/*authservice, an application can open a connection with the HTTP POST method and exchange XML messages with the authentication service. The format is an XML representation of JAAS.

In addition, the authentication service complies with JAAS standards and supports SSO, chain-of-authentication modules, policy-based authorization, the Liberty Alliance specifications, SAML, centralized audit trails, and other advanced security features. On Access Manager, therefore, you can switch to other authentication methods, such as LDAP, UNIX, RADIUS, or certificates. The change is transparent to clients--of course, the new authentication method must use the corresponding authentication information, usually user name and password--and requires only a few clicks in the Access Manager Administration Console to take effect.

The Administration Console, a Web-based GUI, is another benefit offered by Access Manager. From the console, you can customize an authentication module by integrating that module as an authentication service into the [Access Manager service management framework](http://docs.sun.com/source/816-6773-10/service.html). To define the custom service, you need create only two files.

**Implementation of the JDBC Authentication Module**

|  |
| --- |
|  |

In this section are the business case for adopting the JDBC authentication module and the procedure for implementing that module.

**Business Case**

Absent an integration of the Access Manager authentication service into Web applications, businesses usually opt for ad hoc alternatives for authenticating users, such as saving user IDs and passwords into a table in a relational database or other similar approaches. For such applications, businesses need a way to authenticate users against an existing relational database through the Access Manager authentication service. Customers can then benefit from Access Manager features (such as SSO and standards-based, centralized authentication and authorization), and businesses can obviate the need to immediately switch the user database to an LDAP server (Sun Java Enterprise System Directory Server, in the case of Access Manager). **Note:** You still need an LDAP directory, though, because authorization and profile information tend to require it in many cases in the current version of Access Manager.

The answer to these needs is simple: Develop a custom authentication module with the SPI in the Access Manager authentication SDK and plug this module into the Access Manager authentication framework. All you need to change in the application is the login logic. Instead of a direct authentication to the database with the JDBC API, the Access Manager authentication API completes the login logic.

**Procedure**

FIGURE 1 shows the relationship between the JDBC authentication module and the architecture of the Access Manager authentication framework. The user information resides in a PointBase database. A Web-based authentication client calls the module through the Access Manager authentication service.

|  |
| --- |
| FIGURE 1: Relationship Between the JDBC Authentication Module and the Architecture of the Access Manager Authentication Framework**FIGURE 1**: Relationship Between the JDBC Authentication Module and the Architecture of the Access Manager Authentication Framework |

The following subsections explain the three steps for implementing the module:

* Customizing a module
* Configuring the authentication module
* Defining a custom service for the module

**Step I. Customize Your Module.**

The abstract class com.sun.identity.authentication.spi.AMLoginModule implements the JAAS login module (LoginModule). Extend that class to implement three methods of AMLoginModule in order to access the Access Manager authentication service, as follows.

**The init() method.** This is an abstract method that initializes LoginModule with the relevant information. Subsequent to an instantiation of LoginModule and before any calls to LoginModule's other public methods, AMLoginModule calls init(), which may traverse the options to determine the configurations that affect LoginModule's behavior and save the option values in variables for future use. In this example, init() includes JDBC connection parameters, such as the database URL, the table name, the user name and password in the database, and the JDBC driver.

Here is the code for init().

|  |
| --- |
| public void init(Subject subject, Map sharedState, Map options) { // The return value of options.get() is type of "HashSet" dbDriver = (String)((HashSet)(options.get("iplanet-am-auth-jdbc-db-driver"))).iterator().next(); dbUrl = (String)((HashSet)(options.get("iplanet-am-auth-jdbc-db-url"))).iterator().next(); dbUserName = (String)((HashSet)(options.get("iplanet-am-auth-jdbc-db-user"))).iterator().next(); dbPassword = (String)((HashSet)(options.get("iplanet-am-auth-jdbc-db-password"))).iterator().next(); if (dbUserName == null) dbUserName = ""; if (dbPassword == null) dbPassword = ""; if (dbDriver != null) loadDBDriver(this.getClass(), dbDriver); //generate the user credential query out of the options String dbUserTable = (String)((HashSet)(options.get("iplanet-am-auth-jdbc-auth-table"))).iterator().next(); String dbUserTableUserField = (String)((HashSet)(options.get("iplanet-am-auth-jdbc-auth-ufield"))).iterator().next(); String dbUserTableCredentialField = (String)((HashSet)(options.get("iplanet-am-auth-jdbc-auth-cfield"))).iterator().next(); queryStr = "select "+dbUserTableCredentialField+" from "+dbUserTable+" where"+dbUserTableUserField+"=?"; } |



**The process() method.** This method implements the actual login logic for authentication. In this example, process() prompts for a user name and password and attempts to verify those credentials against a database. Typically, the process proceeds this way:

1. Perform the authentication.
2. This next step varies according to the result:
	* If authentication succeeds, save the principal data and return -1.
	* If authentication fails, as in the case of an incorrect password, process() throws the InvalidPasswordException error and, if the number of login attempts reaches the maximum allowed, locks out the account.

	If the failure results from a faulty database connection and other similar problems, process() throws the AuthLoginException error.

Here is the code for process().

|  |
| --- |
| public int process(Callback[] callbacks, int state) throws AuthLoginException { int currentState = state; String userName; String passwd; try{ if (currentState == 1) { userName = ((NameCallback) callbacks[0]).getName(); passwd = new String(((PasswordCallback)callbacks[1]).getPassword()); if (userName.equals("") || passwd.equals("")) { throw new AuthLoginException("names must not be empty"); } //load user from database if ( dbLogin(userName,passwd) ) { userTokenId = userName; return -1;//return -1 for login successful; }else{ throw new InvalidPasswordException ("InvalidPassword"); } }else{ throw new AuthLoginException("Login failed!"); } } catch (SQLException e){ throw new AuthLoginException("Login failed with SQLException"); } catch (ClassNotFoundException e){ throw new AuthLoginException("Login failed with ClassNotFoundException"); } } |



**The getPrincipal() method.** Call this method once only at the end of a successful authentication session. A login session is considered successful when the module has sent all login pages and has thrown no exceptions. This method retrieves the authenticated token string--the user ID in the Access Manager environment.

Here is the code for getPrincipal().

|  |
| --- |
| public java.security.Principal getPrincipal() { if (userPrincipal != null) { return userPrincipal; } else if (userTokenId != null) { userPrincipal = new SamplePrincipal(userTokenId); return userPrincipal; } else { return null; } } |



**Step II. Configure the Authentication Module.**

Each authentication module has its own configuration file in XML format, located in *Access\_Manager\_dir*/SUNWam/web-apps/services/config/auth/default. That configuration file, named *module\_name*.xml, for the authentication module defines the information required for authentication. Modifying the elements in this file automatically and dynamically customizes the authentication interface. In this example, the user name and password used by the callback method are in the configuration file called JDBC.xml.

|  |
| --- |
|  |
| **Note** - Access Manager also uses this configuration file to automatically generate login Web pages for users. In this example, a page based on the JavaServer Pages (JSP) technology collects the authentication information; the page generated by Access Manager is not in use here.  |
|  |

Here is the code for ModuleProperties().

|  |
| --- |
| <ModuleProperties moduleName="JDBC" version="1.0" >  <Callbacks length="2" order="1" timeout="60"  header="This is a DB login module sample page" >  <NameCallback>  <Prompt> User Name </Prompt>  </NameCallback>  <PasswordCallback echoPassword="false" >  <Prompt> Password </Prompt>  </PasswordCallback>  </Callbacks> </ModuleProperties>  |



**Step III. Define a Custom Service for the Module.**

In Access Manager, a service is a means of managing groups of attributes with a particular focus and scope. To define a JDBC service managed by Access Manager, you create an XML service file. A localization properties file specifies the screen text and messages that an administrator or user sees when directed to the service attribute configuration page of a JDBC authentication module.

The XML service file will be imported into Access Manager. Once the service is imported, you can register it to any organization and manage or customize its attributes through the Access Manager Administration Console. You must copy the localization properties file to the A*ccess\_Manager\_dir*/SUNWam/locale directory. For detailed steps of defining a custom service, refer to the Access Manager [documentation](http://docs.sun.com/source/816-6774-10/prog_service.html#wp19647).

|  |
| --- |
|  |
| **Note** - The name of a service (other than the name of an authentication module) as defined in the XML service file can be any unique string. The name of an authentication module service as defined in the XML service file must be in the form iPlanet*AMAuth\_module\_name*Service. In our example, the name is iPlanetAMAuthJDBCService.  |
|  |

The content in the localization file amAuthJDBC.properties. is shown below.

|  |
| --- |
| iplanet-am-auth-jdbc-service-description=JDBCa101=Database Driver Classnamea102=Database URLa103=Database usera104=Database passworda105=UserInfo Table Namea106=UserID fielda107=Credential field |



The content in the file amAuthJDBC.xml is shown below.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?><!-- Copyright (c) 2004 Sun Microsystems, Inc. All rights reserved Use is subject to license terms.--><!DOCTYPE ServicesConfiguration PUBLIC "=//iPlanet//Service Management Services (SMS) 1.0 DTD//EN" "jar://com/sun/identity/sm/sms.dtd"><ServicesConfiguration> <Service name="iPlanetAMAuthJDBCService" version="1.0"> <Schema serviceHierarchy="/DSAMEConfig/authentication/iPlanetAMAuthJDBCService" i18nFileName="amAuthJDBC" i18nKey="iplanet-am-auth-jdbc-service-description">  <Organization> <AttributeSchema name="iplanet-am-auth-jdbc-db-driver" type="single" syntax="string" i18nKey="a101"> <DefaultValues> <Value>com.pointbase.jdbc.jdbcUniversalDriver</Value> </DefaultValues> </AttributeSchema> <AttributeSchema name="iplanet-am-auth-jdbc-db-url" type="single" syntax="string" i18nKey="a102"> <DefaultValues> <Value>jdbc:pointbase://127.0.0.1:9092/sample</Value> </DefaultValues> </AttributeSchema> <AttributeSchema name="iplanet-am-auth-jdbc-db-user" type="single" syntax="string" i18nKey="a103"> <DefaultValues> <Value>PBPUBLIC</Value> </DefaultValues> </AttributeSchema> <AttributeSchema name="iplanet-am-auth-jdbc-db-password" type="single" syntax="password" i18nKey="a104"> <DefaultValues> <Value>PBPUBLIC</Value> </DefaultValues> </AttributeSchema> <AttributeSchema name="iplanet-am-auth-jdbc-auth-table" type="single" syntax="string" i18nKey="a105"> <DefaultValues> <Value>user\_t</Value> </DefaultValues> </AttributeSchema> <AttributeSchema name="iplanet-am-auth-jdbc-auth-ufield" type="single" syntax="string" i18nKey="a106"> <DefaultValues> <Value>uid</Value> </DefaultValues> </AttributeSchema> <AttributeSchema name="iplanet-am-auth-jdbc-auth-cfield" type="single" syntax="string" i18nKey="a107"> <DefaultValues> <Value>passwd</Value> </DefaultValues> </AttributeSchema> </Organization> </Schema> </Service></ServicesConfiguration> |



**Implementation of the Authentication Client**

|  |
| --- |
|  |

You can use the Web-based authentication service user interface in Access Manager for all out-of-the-box authentication modules installed in the Access Manager deployment. This interface consists of one or more login pages that are based on the requirements of the invoked authentication module. You can customize these pages so that their look and feel accords with other Web pages in your application.

To minimize the modifications to an existing application, our example uses "0-page login." That way, the existing login page remains; it forwards the authentication request to Access Manager by encoding the login information into the URL by means of the login URL parameters provided by Access Manager.

|  |
| --- |
|  |
| **Caution** - The example technique could represent a fairly major security issue. Since you're encoding these as http GET strings, the password value will be logged in clear text in the deployment containers access logs.  |
|  |

FIGURE 2 illustrates the authentication and SSO flow when a user accesses the example Web application through the Access Manager authentication service.

|  |
| --- |
| FIGURE 2: Authentication and SSO Flow of Access Manager Authentication Service**FIGURE 2**: Authentication and SSO Flow of Access Manager Authentication Service |

**Login and Logout**

In the login process, the developer of the authentication client assembles the URL from a login request. A typical URL used for login is (all on one line)

http://*server\_name.domain\_name:port\_number*/amserver/UI/
Login?IDToken1=*userid*&IDToken2=*password*&org=*orgName*
&module=JDBC&goto=*success\_login\_page*&gotoOnFail=*login\_fail\_page*

where

* IDToken1 and IDToken2 parameters are used by the JDBC authentication module for user ID and password, respectively.
* org parameter enables a user to authenticate as a user in the organization specified by *orgName*.
* module parameter allows authentication by the authentication module specified by *moduleName* (here, JDBC).

Note: Typically, org and module are not needed; at most, you might use one or the other.
* goto parameter links to the URL specified by *success\_login\_page* after successful authentication against the database.
* gotoOnFail parameter links to the URL specified by *login\_fail\_page* if a user fails authentication.

In the logout process, the goto=*logout\_URL* parameter links users to a desired Web page after logout. For our example, the logout URL is (all on one line) http://*server\_name.domain\_name:port\_number*/amserver/UI/
Logout?goto=http://*web\_servername:port\_number*/authsample/index.html.

**Subsequent Access to SSO**

After authentication, a user can access other authentication-required pages without logging in again. The SSO token API checks the SSO token in the request to determine whether the user has already been authenticated. To extract the SSO token from the request in order to validate the token, add the following code to each authentication-required page.

|  |
| --- |
| . . . SSOTokenManager ssoManager = SSOTokenManager.getInstance(); SSOToken ssoToken = null; String orgName = ""; try{ ssoToken = ssoManager.createSSOToken(request); . . . }catch(Exception e){ // Exception handling } if (ssoToken != null && ssoManager.isValidToken(ssoToken)) { // Allow access to the requested page } else { // Forward to an error page, which prompts user to  // log in } . . . |



To minimize the modifications to an existing application, this piece of code mimics the behavior of the Access Manager policy agent.

|  |
| --- |
|  |

Q: - Can you explain JDBCRealm?

Ans: -

JDBCRealm in GlassFish

JDBC realm has a lot of attention in recent months. This blog summarizes the evolution of the JDBC realm implementation in GlassFish and explains how the latest implementation works. I would like to thank Jean-Baptiste, and Richter for their contributions and comments. The participation from the open source community definitely helps everyone. I encourage all of you to give feedback, participate, and help evolve this feature further.

GlassFish always had the capability for anyone to plug-in a realm. Implementing a custom realm in the Sun Java System Application Server EE 8.0 is described in the article [Authentication Using Custom Realms in Sun Java System Application Server](http://developers.sun.com/prodtech/appserver/reference/techart/as8_authentication/index.html). In S1AS 7.x, there is a JDBC Realm bundled in sample. Jean-Baptiste formally filed an [enhancement](https://glassfish.dev.java.net/issues/show_bug.cgi?id=171) and provided a clear text version of JDBCRealm for GlassFish. Richter wrote another [implementation](http://jroller.com/page/brviking?entry=glassfish_for_tomcat_users_jdbc) because the GlassFish JDBCRealm at that time not compatible with Tomcat.

The latest GlassFish implemtation started with the Jean-Baptiste's implementation and includes the following modifications:

* add message digest configuration
* add encoding configuration for digest
* add charset configuration
* add database user and password configuration in realm
* fix the issue with acquiring a connection for EJB

For the GlassFish implementation of the JDBCRealm, the following design descisions were made:

* Should clear text passwords be supported? The decision was made to not to have clear text passwords as default storage mechanism. Under normal circumstances, passwords should not be stored as clear text.
* Should this implementation be compatible with Tomcat and JBoss? The decision was made to attempt to implement this compatibility, but not require this compatibility.

I would appreciate hearing from you on both of these topics.

**GlassFish JDBCRealm**

The following table summarizes configurable properties for GlassFish JDBCRealm:

|  |  |  |  |
| --- | --- | --- | --- |
| **Properties** | **Mandatory** | **Default** | **Remark**  |
| datasource-jndi | Y |   |    |
| user-table | Y |   |    |
| user-name-column | Y |   | as in Tomcat, we assume that both user-table and group-table has this column  |
| password-column | Y |   |    |
| group-table | Y |   |    |
| group-name-column | Y |   |    |
| jaas-context | Y |   | jdbcRealm  |
| db-user | N | from connection pool configuration | If the connection pool doesn't have username or password, define the db-user and db-password here. This allows us to specify a secure resource that is accessible by an realm with user and password specified.  |
| db-password  |
| digest-algorithm | N | MD5 | support all algorithms in JDK, and also "none"  |
| encoding | N | Hex if there is a digest-algorithm, no encoding if digest is not defined | Hex, Base64, not specified (no encoding)  |
| charset | N |   | charset for digest  |

In GlassFish, a security feature allows you to specify a username and password in the JDBCRealm instead of in the connection pool. By doing this, other applications cannot look up the datasource and get a connection to browse the user-name, password database tables.

**How to use JDBCRealm in a JavaEE application?**

1. Create database tables for the JDBCRealm.
For example, if you are using Derby, you would create database tables as shown in the following code example:
2. create table usertable(userid varchar(10) not null, password varchar(32) not null, primary key(userid));
3. create table grouptable(userid varchar(10) not null, groupid varchar(20) not null, primary key(userid));
4. alter table grouptable add constraint FK\_USERID foreign key(userid) references usertable(userid);

1. Create a JDBCRealm using the Admin Console.
For instance, one can create a JDBCRealm with the following properties:

|  |  |
| --- | --- |
| **Property Name** | **Property Value**  |
| datasource-jndi | jdbc/security  |
| user-table | usertable  |
| user-name-column | userid  |
| password-column | password  |
| group-table | grouptable  |
| group-name-column | groupid  |
| jaas-context | jdbcRealm  |

1. where jdbc/security is a datasource can access the tables above.
2. Specify that the JDBCRealm is the realm to be used as the realm for the application. This is specified in the deployment descriptor: sun-application.xml (for EAR) or, web.xml (for WAR) or sun-ejb-jar.xml (for EJB JAR).
3. Make sure that you have <security-role-mapping> in sun-\*.xml. For instance,
  <security-role-mapping>
    <role-name>Employee</role-name>
    <principal-name>Calvin</principal-name>
  </security-role-mapping>
4. Create a user in database. There is no administration tools for creating users for JDBCRealm. A snapshot of a program that can be used to create users in JDBCRealm is as follows:
5. ...
6. private static final String userSql = "insert into usertable values(?, ?)";
7. private static final String groupSql = "insert into grouptable values(?, ?)";
8. private static String hashPassword(String password) throws Exception {
9. ...
10. //compute digest
11. ...
12. //encode the bytes
13. ...
14. }
15. public static void main(String args[]) throws Exception {
16. ...
17. Class.forName(driver);
18. String hPassword = hashPassword(password);
19. Connection conn = DriverManager.getConnection(
20. jdbcUrl, dbUser, dbPassword);
21. PreparedStatement userStmt = conn.prepareStatement(userSql);
22. userStmt.setString(1, user); userStmt.setString(2, hPassword);
23. userStmt.executeUpdate();
24. userStmt.close();
25. ...
26. }

The sample program can be found in [here](http://blogs.sun.com/roller/resources/swchan/CreateJDBCRealmUser.java). When we run the program, please make sure that the JDBC driver is included in CLASSPATH, for instance, $GLASSFISH\_HOME/javadb/lib/derbyclient.jar

Q: - Can you explain how do you configure JNDIRealm?

Ans: -

Q: - How did you implement caching in JSP?

Ans: - JSP cache tags allow you to cache JSP page fragments within the Java engine. Each can be cached using different cache criteria. For example, suppose you have page fragments to view stock quotes, weather information, and so on. The stock quote fragment can be cached for 10 minutes, the weather report fragment for 30 minutes, and so on.

For more information about response caching as it pertains to servlets, see ["Caching Servlet Results."](http://docs.sun.com/source/817-1833-10/pwasrvlt.html#wp34772)

JSP caching uses the custom tag library support provided by JSP 1.2. JSP caching is implemented by a tag library packaged into the install\_dir/bin/https/jar/webserv-rt.jar file, which is always on the server classpath. The sun-web-cache.tld tag description file can be found in install\_dir/bin/https/tlds directory and can be copied into the WEB-INF directory of your web application.

You can add a taglib mapping in the web.xml of your application as follows:

<taglib>
  <taglib-uri>/com/sun/web/taglibs/cache</taglib-uri>
  <taglib-location>/WEB-INF/sun-web-cache.tld</taglib-location>
</taglib>

You can then refer to these tags in your JSP files as follows:

<%@ taglib prefix="mypfx" uri="/com/sun/web/taglibs/cache" %>

Subsequently, the cache tags are available as <mypfx:cache> and <mypfx:flush>.

The tags are as follows:

* [cache](http://docs.sun.com/source/817-1833-10/pwajsp.html%22%20%5Cl%20%22wp41901)

* [flush](http://docs.sun.com/source/817-1833-10/pwajsp.html%22%20%5Cl%20%22wp43264)

### cache

The cache tag allows you to cache fragments of your JSP pages. It caches the body between the beginning and ending tags according to the attributes specified. The first time the tag is encountered, the body content is executed and cached. Each subsequent time it is run, the cached content is checked to see if it needs to be refreshed and if so, it is executed again, and the cached data is refreshed. Otherwise, the cached data is served.

##### Attributes

The following table describes attributes for the cache tag. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

|  |
| --- |
| Table 3-4  cache Attributes  |
| **Attribute**  | **Default**  | **Description**  |
| key  | ServletPath\_Suffix  | (optional) The name used by the container to access the cached entry. The cache key is suffixed to the servlet path to generate a key to access the cached entry. If no key is specified, a number is generated according to the position of the tag in the page.  |
| timeout  | 60s  | (optional) The time in seconds after which the body of the tag is executed and the cache is refreshed. By default, this value is interpreted in seconds. To specify a different unit of time, add a suffix to the timeout value as follows: s for seconds, m for minutes, h for hours, d for days. For example, 2h specifies two hours.  |
| nocache  | false  | (optional) If set to true, the body content is executed and served as if there were no cache tag. This offers a way to programmatically decide whether the cached response should be sent or whether the body must be executed, though the response is not cached.  |
| refresh  | false  | (optional) If set to true, the body content is executed and the response is cached again. This lets you programmatically refresh the cache immediately, regardless of the timeout setting.  |

##### Example

The following example represents a cached JSP page:

<%@ taglib prefix="mypfx" uri="/com/sun/web/taglibs/cache" %>

<%
   String cacheKey = null;
   if (session != null)
      cacheKey = (String)session.getAttribute("loginId");

   // check for nocache
   boolean noCache = false;
   String nc = request.getParameter("nocache");
   if (nc != null)
      noCache = "true";

   // force reload
   boolean reload=false;
   String refresh = request.getParameter("refresh");
   if (refresh != null)
      reload = true;
%>

<mypfx:cache key="<%= cacheKey %>" nocache="<%= noCache %>" refresh="<%= reload %>" timeout="10m">
<%
   String page = request.getParameter("page");
   if (page.equals("frontPage") {
      // get headlines from database
   } else {
      .....
%>
</mypfx:cache>

<mypfx:cache timeout="1h">
<h2> Local News </h2>
<%
   // get the headline news and cache them
%>
</mypfx:cache>

### flush

Forces the cache to be flushed. If a key is specified, only the entry with that key is flushed. If no key is specified, the entire cache is flushed.

##### Attributes

The following table describes attributes for the flush tag. The left column lists the attribute name, the middle column indicates the default value, and the right column describes what the attribute does.

|  |
| --- |
| Table 3-5  flush Attributes  |
| **Attribute**  | **Default**  | **Description**  |
| key  | ServletPath\_Suffix  | (optional) The name used by the container to access the cached entry. The cache key is suffixed to the servlet path to generate a key to access the cached entry. If no key is specified, a number is generated according to the position of the tag in the page.  |

##### Examples

To flush the entry with key="foobar":

|  |  |
| --- | --- |
|

|  |
| --- |
| <mypfx:flush key="foobar"/>  |

 |

To flush the entire cache:

|  |  |
| --- | --- |
|

|  |
| --- |
| <% if (session != null && session.getAttribute("clearCache") != null) { %>     <mypfx:flush /> <% } %>  |

 |