

**Servlet Life Cycle:**

**When the Web Container initializes a servlet, it creates a ServletConfig object for the servlet**

**init ():**  The servlet is **initialized** by calling the **init**() method. The init method is called **only once**. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

## **service ()**

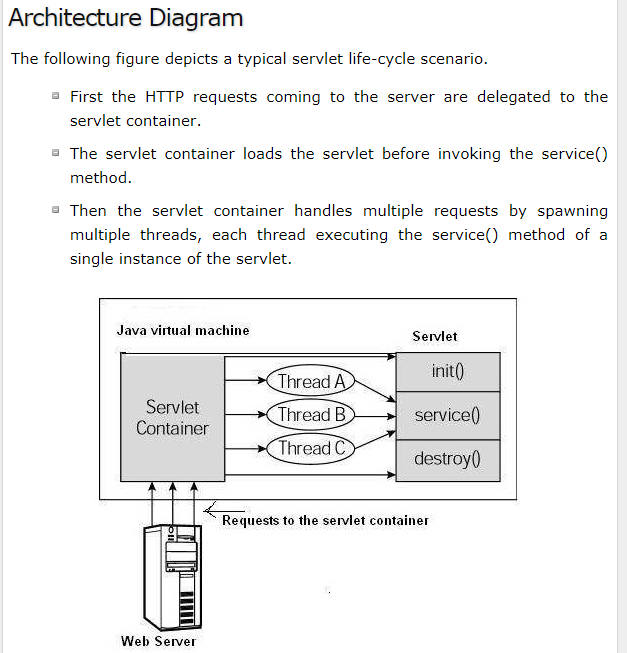
servlet container (i.e. web server) calls the service () method to handle requests coming from the client(browsers) and to write the formatted response back to the client. Each time the server receives a request for a servlet, the server spawns a new thread and calls service.

HttpServlet class provides implementation for service () method

**destroy**

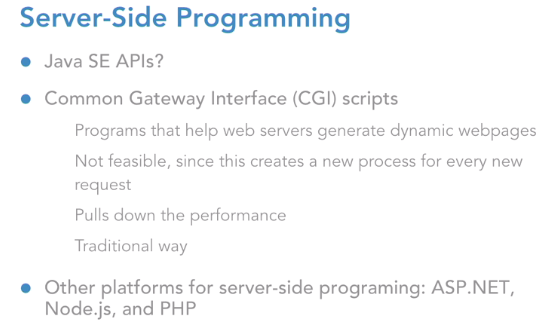
The destroy () method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

**After the destroy () method is called, the servlet object is marked for garbage collection.**

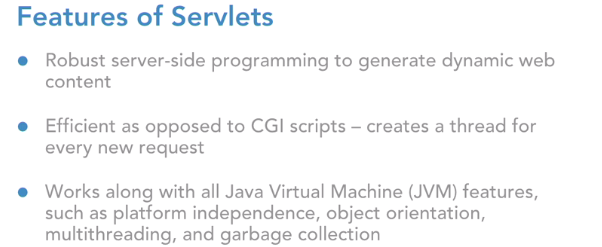


When Servlet is initialized init parameters are read by container.

Entire chain of requests and response that gets exchanged between the web client and the web server is nothing but over the H-T-T-P protocol. Hypertext transfer protocol is the only communication strategy between the web client and the server. They do not understand any other language except H-T-T-P.



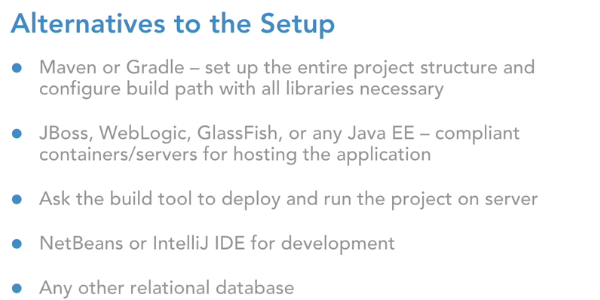
we are into the enterprise world of Java, and we need an A-P-I so that we can code server-side programming features. And that is exactly where servlets pitch in. So, let us look at the features of servlets. Servlets offer us a robust server-side programming option to generate dynamic web content.

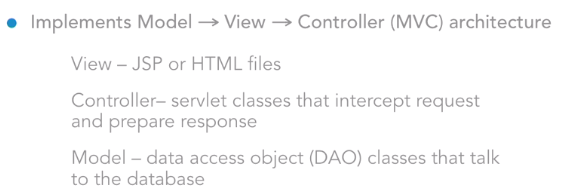


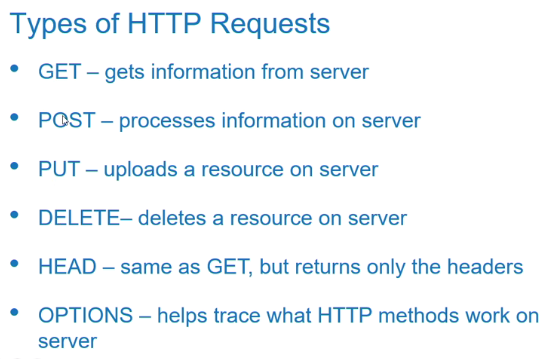


J-S-R, which is the Java Specification Request, the number 315 was released by Java community for version 3.0 as a part of Java Enterprise Edition number six, which works with J-D-K 1.6 and then there is J-S-R 340, which is for version 3.1, which was released as a part of Java E-E seven. Please understand one more point that servlets can work with the entire Java Standard-Edition A-P-Is. Be it object orientation, exception handling, collections, Generix, multi-threading, inner classes and even J-D-B-C.

The servlets are going to work in conjunction with our code Java A-P-Is to help us build a wonderful web application.

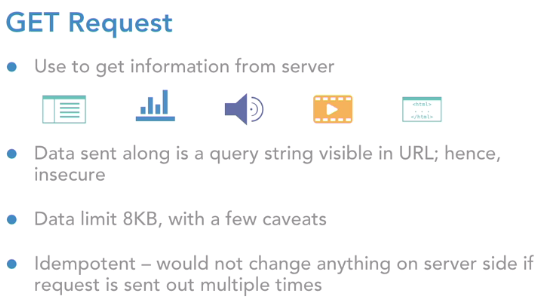






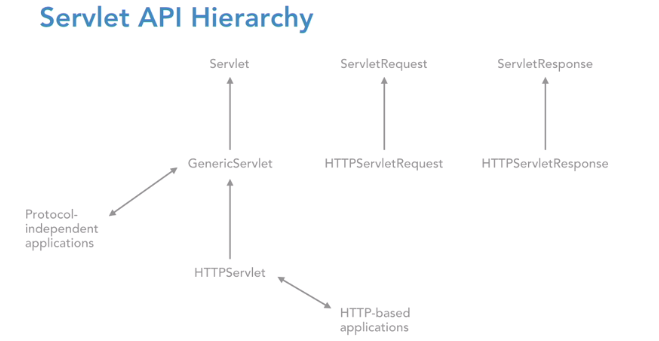
The **Multipurpose Internet Mail Extensions (MIME) type** is a standardized way to indicate the nature and format of a document:

* text/plain
* text/html
* image/jpeg
* image/png
* audio/mpeg
* audio/ogg
* audio/\*
* video/mp4
* application/\*
* application/json
* application/javascript
* application/ecmascript
* application/octet-stream



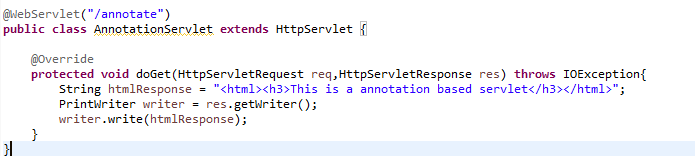
**The GET method has size limitation: only 1024 characters can be used in a request string.**

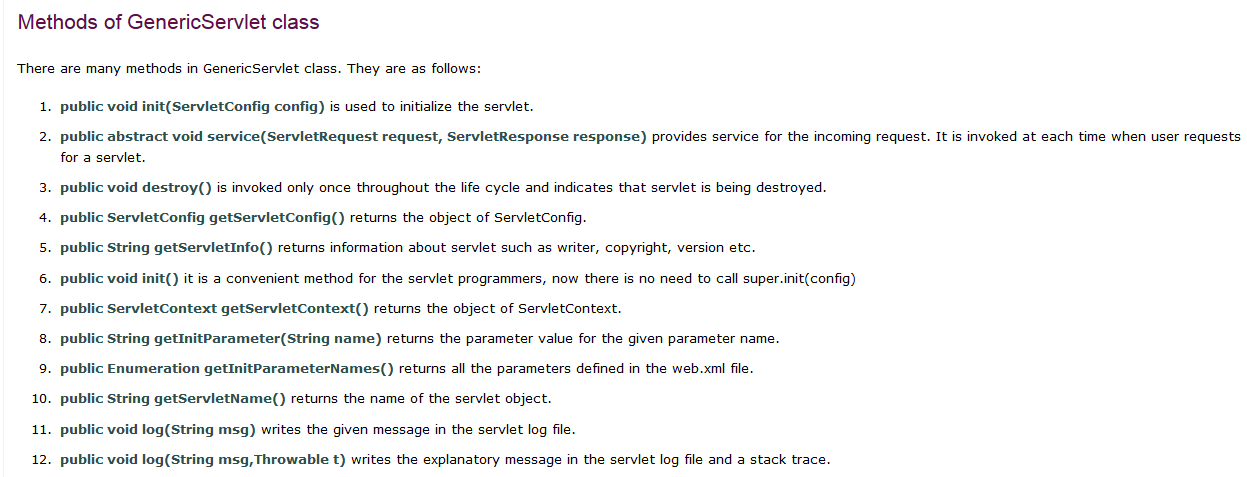
1. **getParameter ()** − request.getParameter() method to get the value of a form parameter.
2. **getParameterValues ()** − If the parameter appears more than once and returns multiple values, for example checkbox.
3. **getParameterNames ()** – Returns complete list of all parameters in the current request.

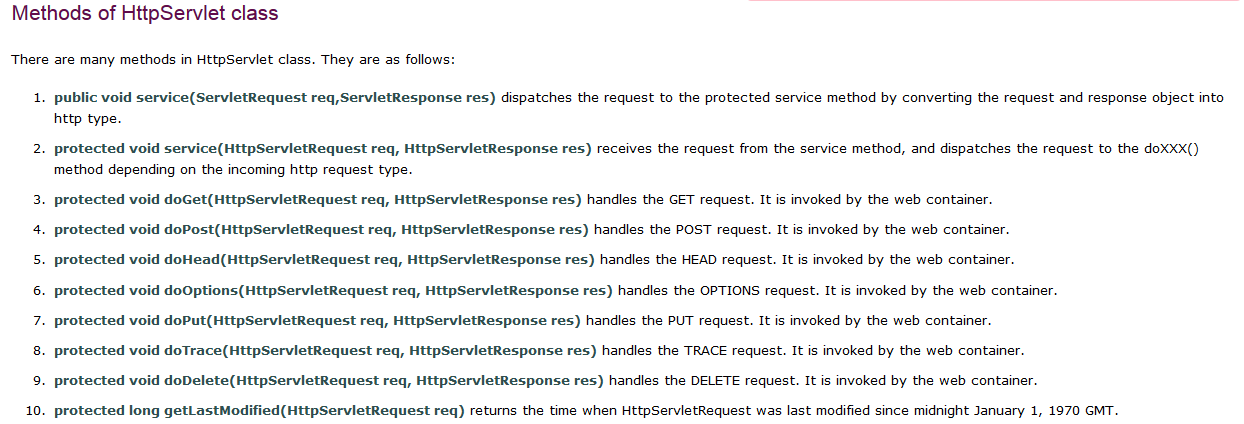


So, on the response object I have an API which is called **getWriter**. This is going to return me a print writer reference. Print Writer is an API under the Java IO package which will help us to exclusively write response on a webpage.

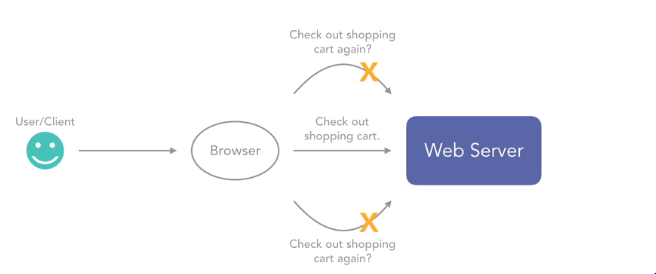
web module 3.0 version allows us to do the configuration via annotations. Annotations are always placed inside a class, so this is how it look like:

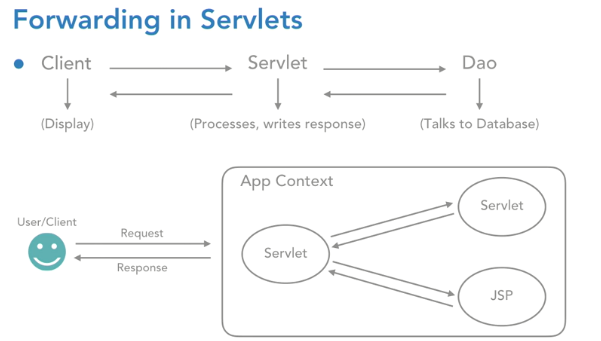




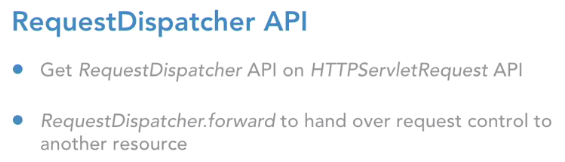






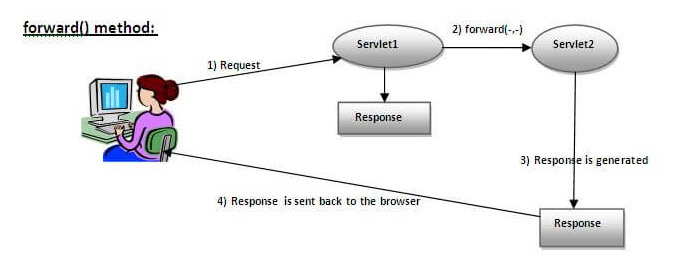


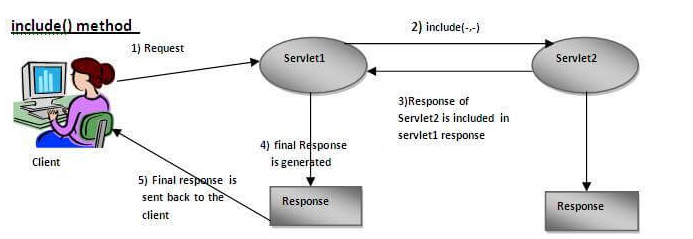
Request forwarding does not reveal any kind of implementation details in URL.



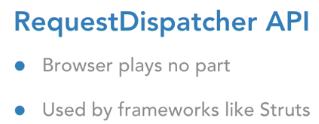


**Include** API combine the response of servlet and the response of jsp file together and render it to the client.

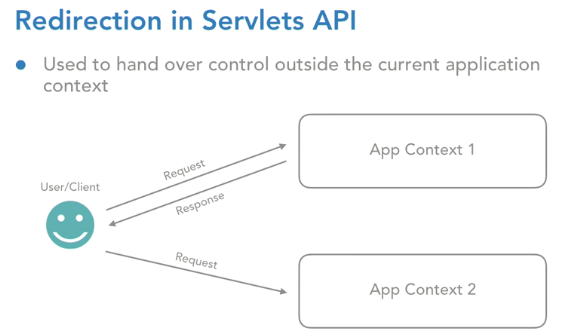




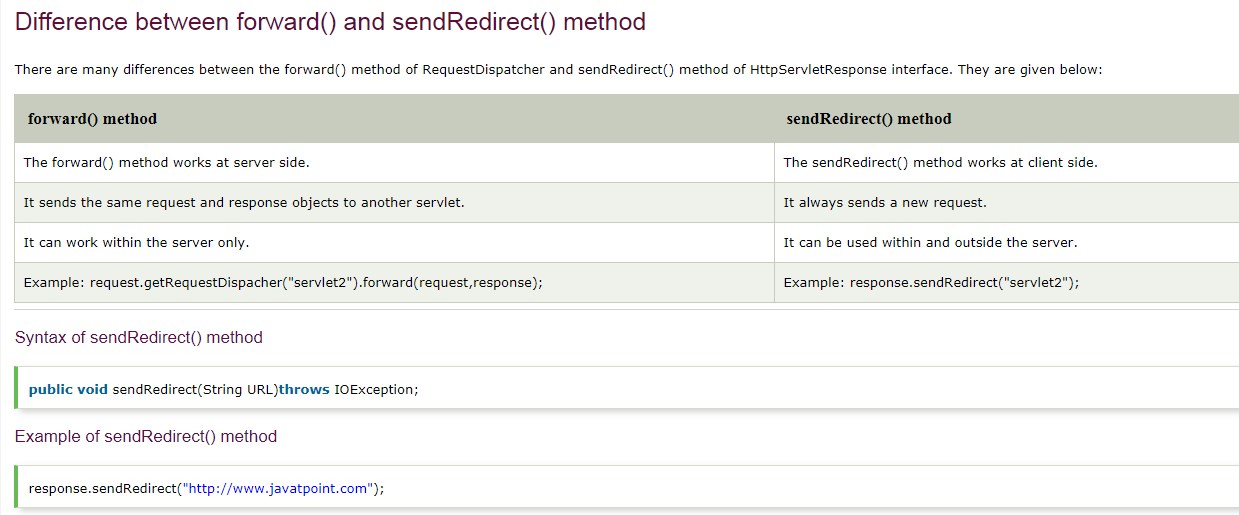




**RequestDispatcher** used to forward the control from one resource to another resource, inside the same application. And in this entire process, the browser will play no part. Entirely, it's managed by the server site. This approach is very commonly used in a framework called **Struts**, where whenever we click on a hyperlink, we always go to an action class and from the action class, the request is dispatched to a JSP. That way, we are always safe, and we never reveal any of our folder details in the URL.

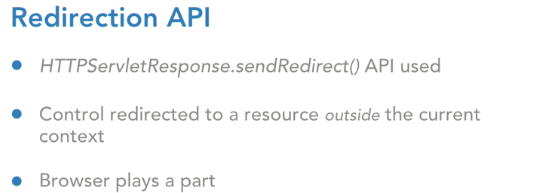




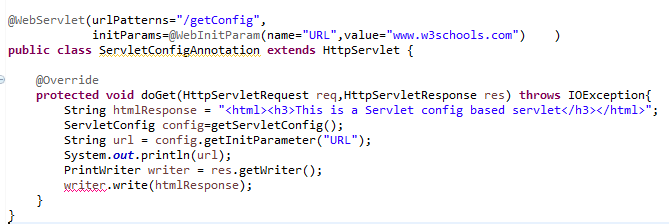


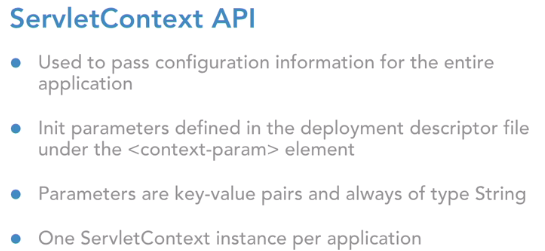
**Signature of init, service and destroy methods:**



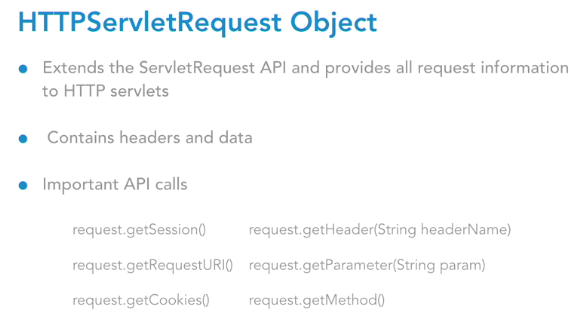


**Servlet Config: Configuring the Init parameter through annotation**

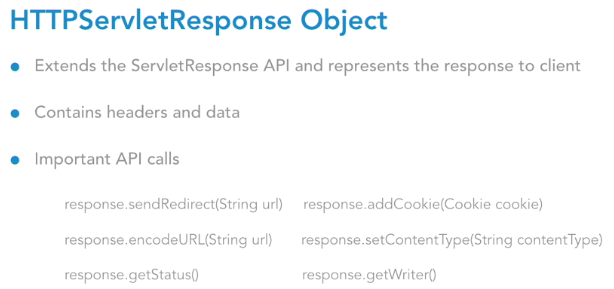




To get the Servlet environment information ServletContext is used



1. Method defined in HttpServletRequest returns the object of RequestDispatcher – getRequestDispatcher()



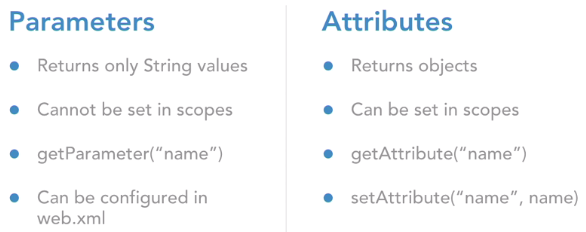


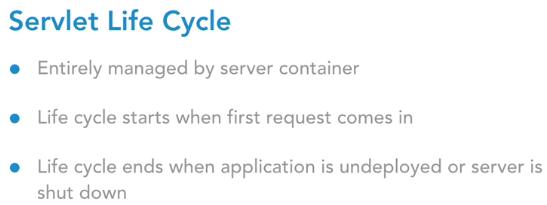
**Request** scope is alive until the response of this request goes back to client.

**Session**: This scope is going to be alive till the user either quits the browser, or the user clicks on the logout button, or maybe the session itself times out on the user's machine.

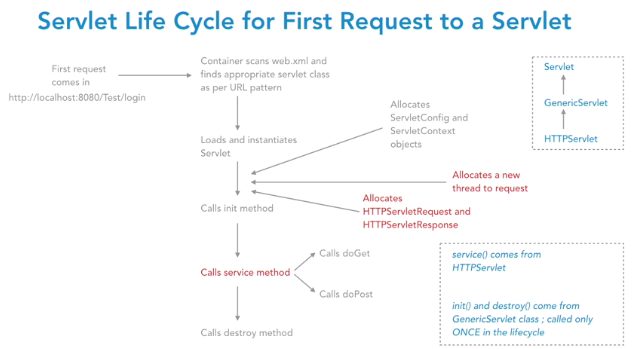
**Context**: context scope, which is of course sometimes known as the application scope, represented by the ServletContext object.

**Page scope**, which is represented by the JSPContext object, and it is going to be accessible from any JSP page that creates that object.





The servlet life cycle firstly is entirely managed by the server container. In our case it is Tomcat, or it could be any other JEE-compliant container source.





The life cycle of a servlet will start off when the first request comes in for that servlet. And it will end when the application is either undeployed or maybe the server is shut down. Now we need to look at the servlet life cycle in two phases. The first phase is when the first request comes in to the servlet. And the second phase is when repeated requests come for the same servlet.

Now we know that every servlet is identified by a URL title. So, let's say we either hit the URL title through the address bar or we hit the URL title via our form element. When the first request comes to any servlet, the container is going to scan that web XML file and find the appropriate servlet class as per URL pattern. Of course, if we have chosen the annotation way of configuration, it'll scan the class accordingly. Once it finds that class, it's going to **load and instantiate** that servlet class, based on the conventional class loader information that is has.

So, there's a ClassLoader API in core Java which it uses to load any instantiate that servlet. Now a very important thing to note here is, that whenever we have **n number of requests coming to the same servlet, there is only one instance of a servlet that gets created**. No matter if there are millions of requests coming from millions of users for the same servlet. But all those requests will have only one servlet object.

Once the instance of the servlet is created, there's a lot of initialization activity that happens after that. The first one is that servlet context and the servlet context objects will get created. Next step is, it allocates a new thread to the request. Now this is important again. We may have our application being accessed by multiple users at the same point in time.

This entire set of concurrent requests is going to be handled by the multithreading capability of Java platform. So, **every request is going to be allocated a new thread, and hence, those threads are going to run in parallel**. So that's exactly what happens at this stage. The incoming request is going to get allocated a new thread. Once this is done, it also allocates a pair of HTTPServletRequest and HTTPServletResponse objects and attaches it to this incoming request.

Once done, then it is going to call a series of APIs. It calls to the init method, the service method, and the destroy method. The **service method comes from the HTTPServlet** class. And the **init** and **destroy** methods come from the **GenericServlet** class.

There's a Servlet interface at the top then there's a GenericServlet, which is the abstract class, the protocol-independent class, and then we have the HTTPServlet class, which is specifically designed for HTTP applications. So, the **service method comes from the HTTPServlet** and the **init and destroy come from the GenericServlet.**

Init method is the place to accommodate any kind of initialization activity for our servlet.

Let's say our servlet wants to interact with the database. So, we can set up the database connection inside the init method. Or let's say our servlet wants to display the weather information all the time so we can connect to that weather service inside the init method. Any kind of one-time job; that happens in init method. Because init method is going to get executed only once during the entire life cycle of the servlet.

The logic that goes inside the service method is: depending on the kind of request that was made from the client, let's say it's a GET or a POST or anything like that, the service method is accordingly going to delegate the call to either the doGet or the doPost methods. And this is the place exactly where our request is being served for the user.

Now, once our application is about to get undeployed, or maybe say our server is absolutely shutting down altogether, that's the point when the destroy method of the servlet is called. Destroy method is going to hold any kind of clean up activity that we want to do for the servlet. So, maybe we have some object created inside the init method where we can clean them up in destroy method so that they're eligible for the garbage collection process. So, this is how the life cycle looks like for the first request.

Now, the second phase, where there are multiple requests coming in for the same servlet, do we see some points highlighted in red in this slide? Those are the steps that happen again for repeated requests. The instance is only one. That does not happen again. Directly, a new thread will be allocated to that request, a new pair of HTTPServlet request and response objects will be created and attached with the request, and then, directly, the service method is called. Again, depending on what request was made, the service will go over and call either the doGet or doPost, so on and so forth.

Like init, destroy is also called only once during the life cycle. So, they will never happen again and again. So, this is how the life cycle looks like overall. So, if we summarize the methods of the servlet life cycle, there are three life cycle methods: init, service, and destroy, out of which init and destroy can definitely be overridden by the developer. Service method, however, there's no logical reason to override it. It has already been implemented in the HTTPServlet class and if we choose to override service, it means we take up the responsibility of delegating the chores to the respective doGet or doPost methods.

A very crucial point is, all the servlet requests that are going to come in the application are multithreaded.

# Event and Listener in Servlet

Events are basically occurrence of something. Changing the state of an object is known as an event. There are many Event classes and Listener interfaces in the javax.servlet and javax.servlet.http packages.

The event classes are as follows:

1. ServletRequestEvent
2. ServletContextEvent
3. ServletRequestAttributeEvent
4. ServletContextAttributeEvent
5. HttpSessionEvent
6. HttpSessionBindingEvent

## **Event interfaces**

The event interfaces are as follows:

1. ServletRequestListener
2. ServletRequestAttributeListener
3. ServletContextListener
4. ServletContextAttributeListener
5. HttpSessionListener
6. HttpSessionAttributeListener
7. HttpSessionBindingListener
8. HttpSessionActivationListener

# ServletContextEvent and ServletContextListener

The ServletContextEvent is notified when web application is deployed on the server.

If you want to perform some action at the time of deploying the web application such as creating database connection, creating all the tables of the project etc, you need to implement ServletContextListener interface and provide the implementation of its methods.

### **Method of ServletContextEvent class**

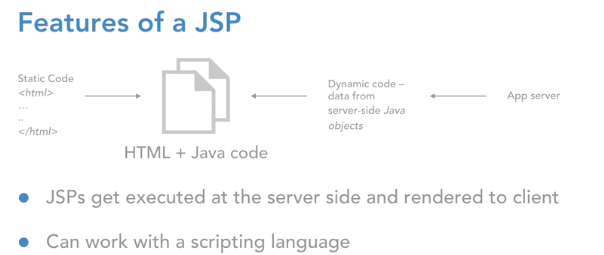
1. **public ServletContext getServletContext()**: returns the instance of ServletContext.

### **Methods of ServletContextListener interface**

There are two methods declared in the ServletContextListener interface which must be implemented by the

servlet programmer to perform some action such as creating database connection etc.

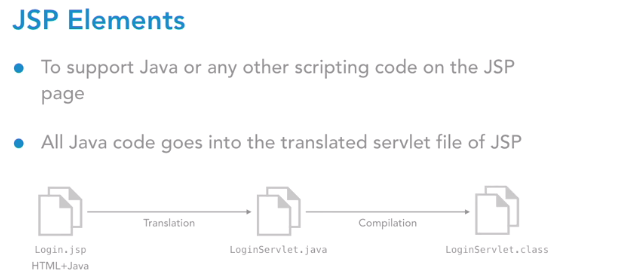
1. **public void contextInitialized(ServletContextEvent e)**: is invoked when application is deployed on the server.
2. **public void contextDestroyed(ServletContextEvent e):** is invoked when application is undeployed from the server.

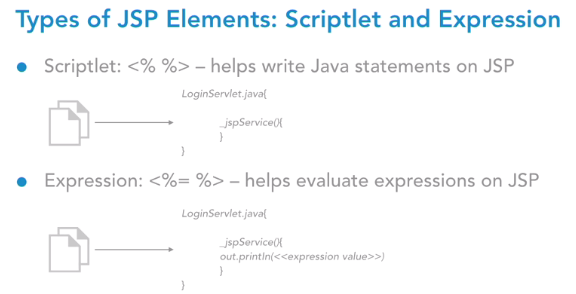


JSP pages are easier to maintain than a servlet.

JSP is built on Java Technology so it is platform independent.

A JSP page separates presentation and business logic.





**Scriptlet:**

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.

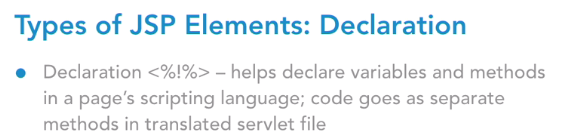
<% code fragment %>

XML equivalent of Scriptlet code:

<jsp:scriptlet> code fragment </jsp:scriptlet>

**Jsp Expression:** The expression element can contain any expression that is valid according to the Java Language Specification, but you cannot use a semicolon to end an expression.

<jsp:expression> expression </jsp:expression>



**Declaration:**

A declaration declares one or more variables or methods that you can use in Java code later in the JSP file. You must declare the variable or method before you use it in the JSP file.

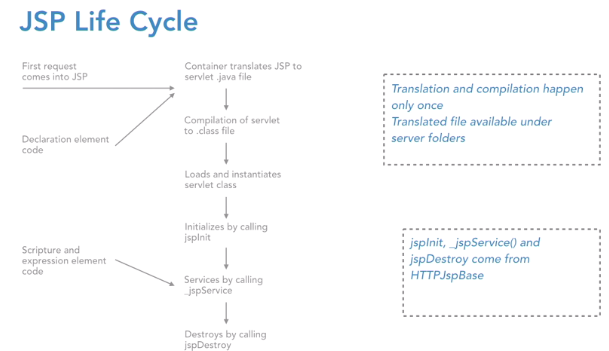
<%! declaration; [ declaration;] + ... %>

XML equivalent of the above syntax as follows −

<jsp:declaration> code fragment </jsp:declaration>

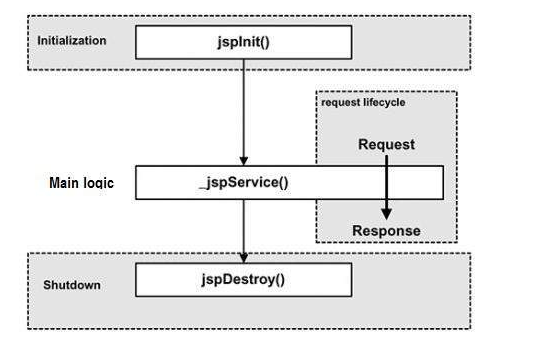
Implicit means it is by default available to the JSP page. We don’t have to create or import explicitly.

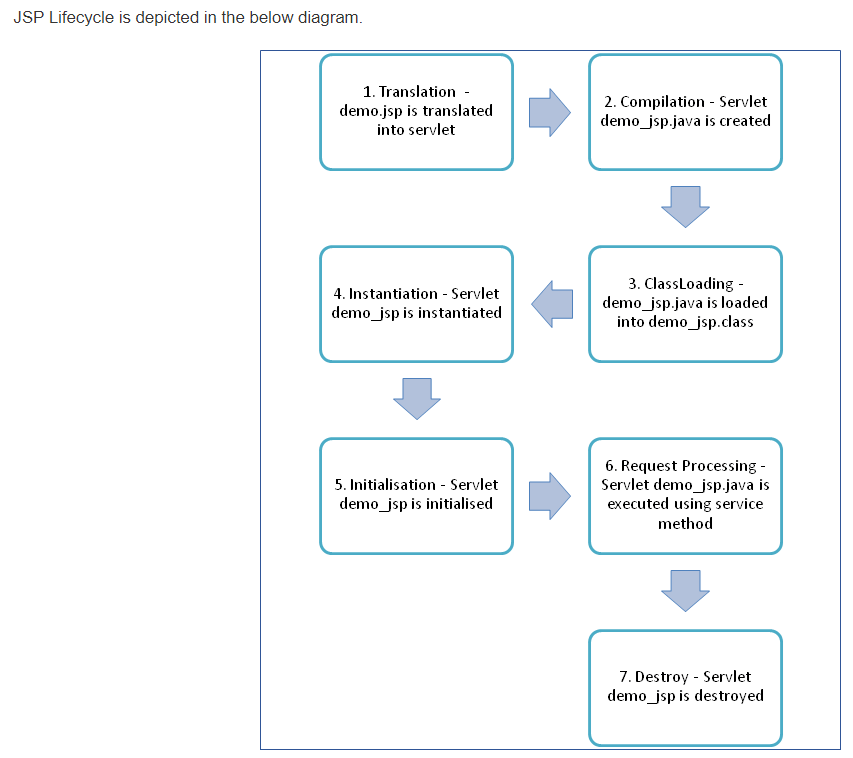
Declaration element allows you to introduce new variables or new methods in the translated file of jsp.



The four major phases of a JSP life cycle are very similar to the Servlet Life Cycle.

1. Compilation
2. Initialization
3. Execution
4. Cleanup





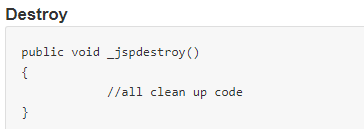
Following steps explain the JSP life cycle:

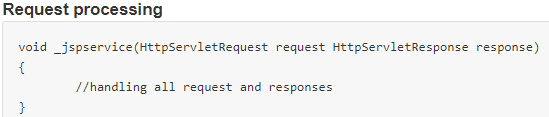
1. Translation of JSP page
2. Compilation of JSP page (Compilation of JSP page into \_jsp.java)
3. Classloading (\_jsp.java is converted to class file \_jsp.class)
4. Instantiation (Object of generated servlet is created)
5. Initialization (jspinit() method is invoked by container)
6. Request Processing(\_jspservice() method is invoked by the container)
7. Destroy (jspDestroy() method invoked by the container)

The JspPage interface describes the generic interaction that a JSP Page Implementation class must satisfy; pages that use the HTTP protocol are described by the HttpJspPage interface.

The interface defines a protocol with 3 methods; only two of them: **jspInit**() and **jspDestroy**() are part of this interface as the signature of the third method: **\_jspService**() depends on the specific protocol used and cannot be expressed in a generic way in Java.

**The jspInit() and jspDestroy() methods can be defined by a JSP author, but the \_jspService() method is defined automatically by the JSP processor based on the contents of the JSP page.**



## **JSP Compilation**

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

The compilation process involves three steps –

* Parsing the JSP.
* Turning the JSP into a servlet.
* Compiling the servlet.

## **JSP Initialization**

When a container loads a JSP it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit() method −

public void jspInit(){

// Initialization code...}

initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

## **JSP Execution**

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the \_jspService() method in the JSP.

The \_jspService() method takes an HttpServletRequest and an HttpServletResponse as its parameters as follows −

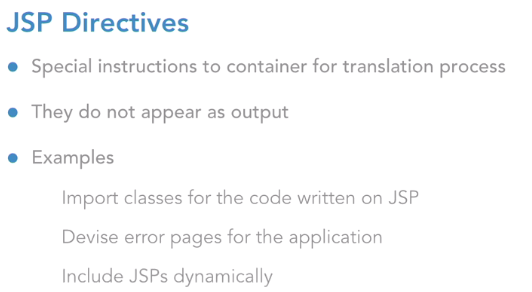
void \_jspService(HttpServletRequest request, HttpServletResponse response) {

// Service handling code...

}

## **JSP Cleanup**

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container. The jspDestroy() method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.



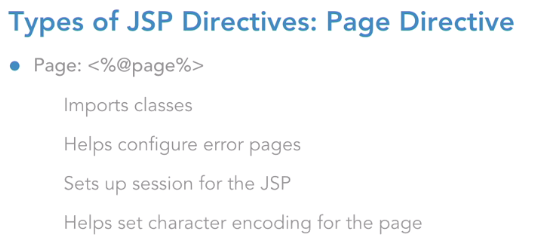
There are 3 types of directive tag:

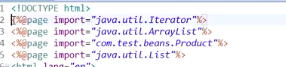
**<%@ page ... %>** Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.

**<%@ include ... %>** Includes a file during the translation phase.

**<%@ taglib ... %>**

Declares a tag library, containing custom actions, used in the page





The **page** directive is used to provide instructions to the container. These instructions pertain to the current JSP page

Basic syntax of the page directive −

<%@ page attribute = "value" %>

You can write the XML equivalent of the above syntax as follows −

<jsp:directive.page attribute = "value" />

Include directive is used to include a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code the include directives anywhere in your JSP page.

The general usage form of this directive is as follows −

**<%@ include file = "relative url" >**

XML equivalent of the above syntax as follows −

<jsp:directive.include file = "relative url" />

**Legal attribute of Page directive:** import, contentType, extends, info, buffer, language, isELIgnored, isThreadSafe, autoFlush, session, pageEncoding, errorPage, isErrorPage

Example:

1. **Import:**



1. **contentType:**

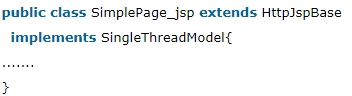


1. **isThreadSafe:**

|  |
| --- |
| Servlet and JSP both are multithreaded.If you want to control this behaviour of JSP page, you can use isThreadSafe attribute of page directive.The value of isThreadSafe value is true.If you make it false, the web container will serialize the multiple requests, i.e. it will wait until the JSP finishes responding to a request before passing another request to it.If you make the value of isThreadSafe attribute like: |

**<%@ page isThreadSafe="false" %>**

The web container in such a case, will generate the servlet as:

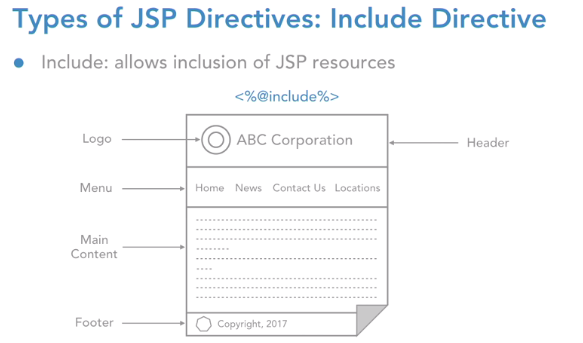


**Except the import attribute of page directive, all the other attributes of page directive cannot be**

**specified more than once.**

**Difference between include directive and include action in JSP:**

|  |  |
| --- | --- |
| **Include Directive (@include)** | **Include Action (<jsp:include>)** |
| Resource is included at translation time | Resource is included at request time |
| Uses file attribute to specify the resource | Uses page attribute to specify the resource |
| Used to include static resource e.g. HTML or Images | Used to include dynamic resource e.g. JSP and Servlet |
| Doesn’t allow us to pass parameters | Allow us to pass parameter using <jsp:param> action |
| Doesn’t pass request and response object | Allow us to pass request and response object |
| Can use both absolute and relative path | Always use relative path to include resource |



Includes a file during the translation phase.

## **The taglib Directive**

The JavaServer Pages API allow you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.

The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides means for identifying the custom tags in our JSP page.

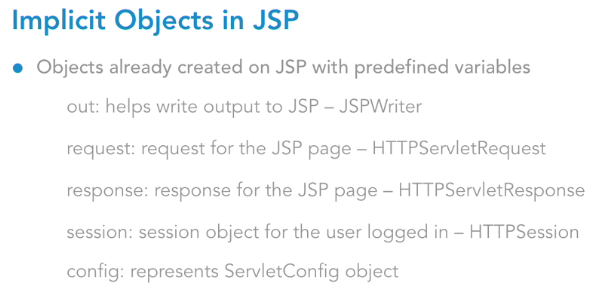
The taglib directive follows the syntax given below −

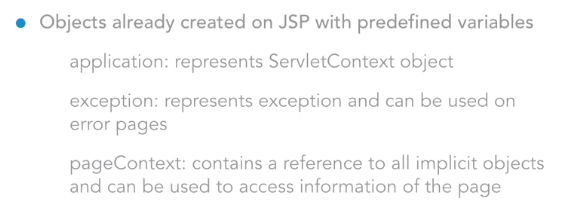
<%@ taglib uri="uri" prefix = "prefixOfTag" >

Here, the uri attribute value resolves to a location the container understands, and the prefix attribute informs a container what bits of markup are custom actions.

You can write the XML equivalent of the above syntax as follows −

<jsp:directive.taglib uri = "uri" prefix = "prefixOfTag" />

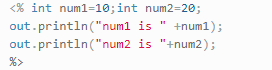




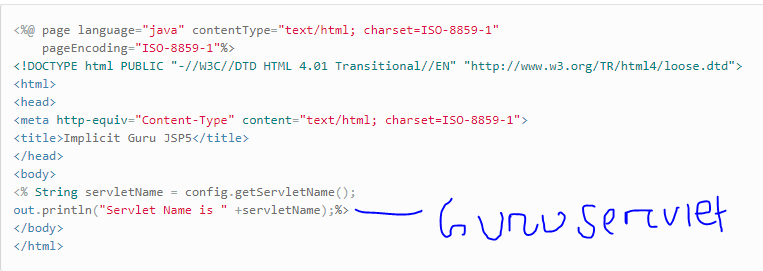
## What is JSP Implicit object?

* JSP implicit objects are created during the **translation** phase of JSP to the servlet.
* These objects can be directly used in scriplets that goes in the **service** method.
* They are created by the container automatically, and they can be accessed using objects

Out:





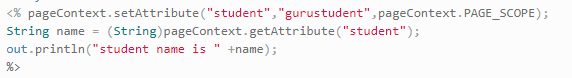


## **pageContext**:

* This object is of the type of pagecontext.
* It is used to get, set and remove the attributes from a particular scope

Scopes are of 4 types:

* 1. Page, 2. Request, 3. Session, 4. Application



we are setting the attribute using pageContext object, and it has three parameters:

* Key
* Value
* Scope

In the above code, the key is student and value: "gurustudent" while the scope is the page scope. Here the scope is "page" and it can get using page scope only.

## **Page**

* Page implicit variable holds the currently executed servlet object for the corresponding jsp.
* Acts as this object for current jsp page.



 In this example, we are trying to use the method toString () of the page object and trying to get the string name of theJSP Page

**What is difference between page and pagecontext?**

Page and Pagecontext are implicit objects in jsp. These are created at JSP translated time. The page object represents the generated servlet instance itself and is used as a scope with in one jsp. Pagecontext is used to initialize all implicit objects for example: - page attributes, access to the request, response and session objects, as well as the JspWriter referenced by out.

**Page**: Implicit object of JSP is equivalent to this variable of Java programming language.

## **The pageContext Object**

The pageContext object is an instance of a **javax.servlet.jsp.PageContext**object. The pageContext object is used to represent the entire JSP page.

This object is intended to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The **application, config, session**, and **out** objects are derived by accessing attributes of this object.

The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the errorPageURL, and page scope.

The PageContext class defines several fields, including **PAGE\_SCOPE, REQUEST\_SCOPE, SESSION\_SCOPE,** and **APPLICATION\_SCOPE**, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the **javax.servlet.jsp.JspContext class**.

One of the important methods is **removeAttribute**. This method accepts either one or two arguments. For example, **pageContext.removeAttribute ("attrName")** removes the attribute from all scopes, while the following code only removes it from the page scope −

pageContext.removeAttribute("attrName", PAGE\_SCOPE);

**JSP implicit objects are defined in \_jspService()**

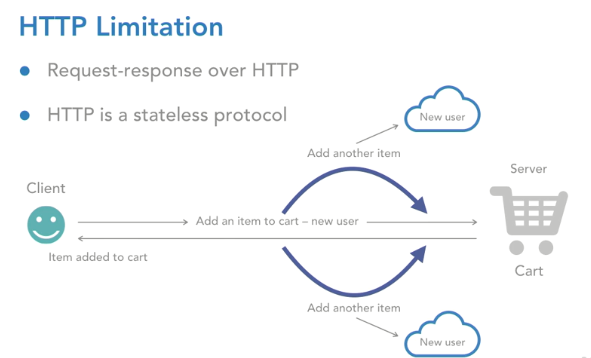
**Important Points**-

1. Implicit variables **application** and **config** are always available to a jsp page.
2. Implicit variable **session** is available if the value of the page directives session attribute is set to true. since it is set to true by default, the implicit variable session is also available by default.
3. **exception** is available only if the value of the page directives isErrorpage attribute is set to true. it is set to false by default, so the implicit variable exception is not available by default. we must explicitly set it to true.

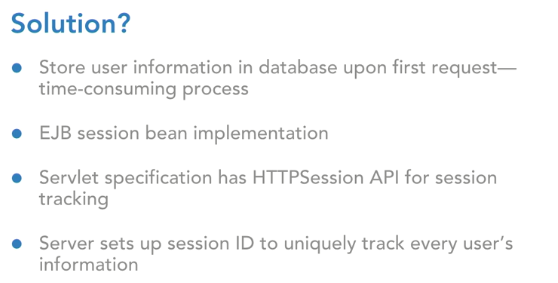
### **JSP Literals**

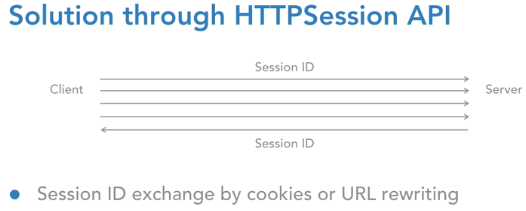
The JSP expression language defines the following literals −

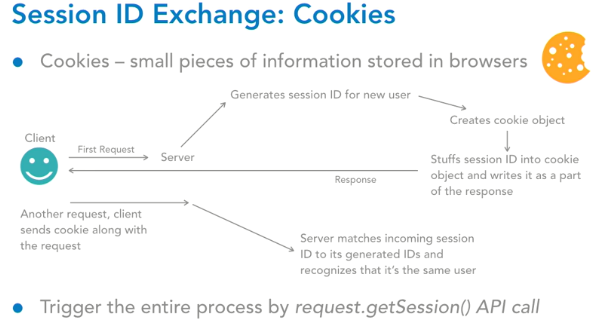
1. **Boolean** − true and false
2. **Integer** − as in Java
3. **Floating point** − as in Java
4. **String** − with single and double quotes; " is escaped as \", ' is escaped as \', and \ is escaped as \\.
5. **Null** − null



HTTP is a "stateless" protocol which means each time a client retrieves a Web page; the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.







Cookies are small pieces of bits of information that are stored in the browser software. They could be information pertaining to the user or the application. So, let's understand how a session ID is generated and then how is it exchanged between the client and server via these cookies. Let's say the client comes in with the first request. Now the server will understand that is a new user because it did not receive any session ID from the client.

So, it will generate a new session ID for this new user. After that, it creates an object of the cookie class. Cookie is an API available to you under the servlet's specification. Once it creates an instance of the cookie, the server is going to stuff that session ID into the cookie object and write it as a part of the response**. response.addCookie** add a cookie object as a part of the response.

So that is exactly what the server does, it writes that entire cookie object stuffed with the session ID back to response and the response is sent to the client. Now let's say the client comes in with another request. The client while making the request ensures that the cookie is always going to travel along with the request so when it reaches the server side, the server can extract the session ID out of the cookie. It will match it with one of its already generated IDs and if it finds one, it recognizes that it is the same user.

Now fortunately for a developer, you do not have to do this entire tedious process of the getting the session ID exchanged. The client and the server are smart enough to do that exchange between them automatically. However, for this entire process to be set up for our application, we must write one line of code which is this: the entire process is triggered by using the API call of request. getSession. This primarily returns us an HTTP session object and that's exactly where this entire process is set up by the server.

* HttpSession object is used to store entire session with a specific client.
* Any servlet can have access to HttpSession object through the getSession () method
* Attribute can be stored, retrieved and removed from HttpSession object
* **request. getSession ()** create a session if one does not exist
* **request. getSession (). getId ()** - returns a string containing the unique identifier assigned to this session. The identifier is assigned by the servlet container and is implementation dependent.

# Difference between HttpSession’s getSession(), getSession(true) and getSession(false) methods

* **getSession**() : Returns the current session associated with this request, or if the request does not have a session, creates one.
* **getSession**(true): Returns the current HttpSession associated with this request, if there is no current session, returns a new session
* **getSession(false):** Returns the current HttpSession associated with this request, if there is no current session, returns null**.**

**JSP – Cookies Handling**

There are three steps involved in identifying and returning users −

1. Server script sends a set of cookies to the browser. For example, name, age, or identification number, etc.
2. Browser stores this information on the local machine for future use.
3. When the next time the browser sends any request to the web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A JSP that sets a cookie might send headers that look something like this −

HTTP/1.1 200 OK

Date: Fri, 04 Feb 2000 21:03:38 GMT

Server: Apache/1.3.9 (UNIX) PHP/4.0b3

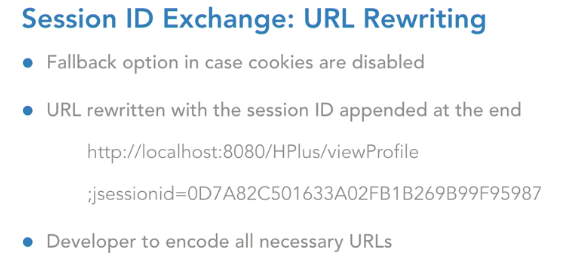
Set-Cookie: name = xyz; expires = Friday, 04-Feb-07 22:03:38 GMT;

path = /; domain = tutorialspoint.com

Connection: close

Content-Type: text/html

**Set-Cookie** header contains a **name** **value** pair, a **GMT** date, a **path** and a **domain**. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.



Now this separator of semicolon and the j session name of the parameter, are very specific to the tomcat container software that we are using. However, if you go ahead and choose any other JEE compliant server.

But the point is that until we encode these URLs, the server would not enable the URL rewriting process for you.

**URL encoding:**

Response.encodeURL(“aaaa”)



Session.invalidate () will invalidate the session.

**Once a session is invalidated it is garbage collected.**

Change in Java files require redeployment and restart.

Change in web.xml requires restart of the server.

Configure the session timeout in web.xml

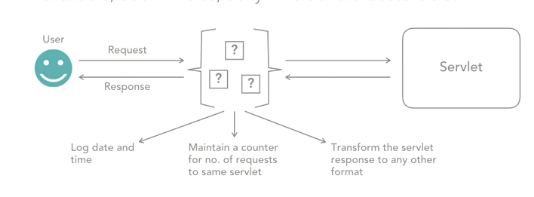


### **URL Rewriting**

We can append some extra data at the end of each URL. This data identifies the session; the server can associate that session identifier with the data it has stored about that session.

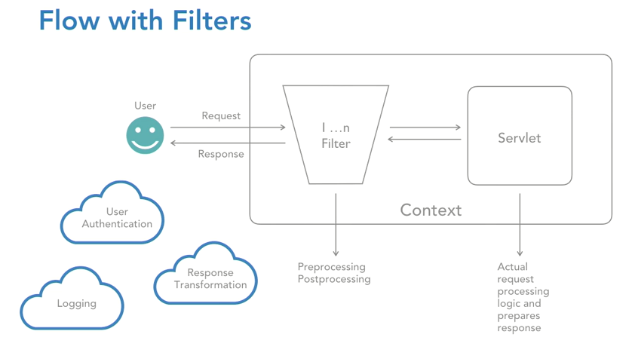
For example, with **http://tutorialspoint.com/file.htm;sessionid=12345**, the session identifier is attached as **sessionid = 12345** which can be accessed at the web server to identify the client.

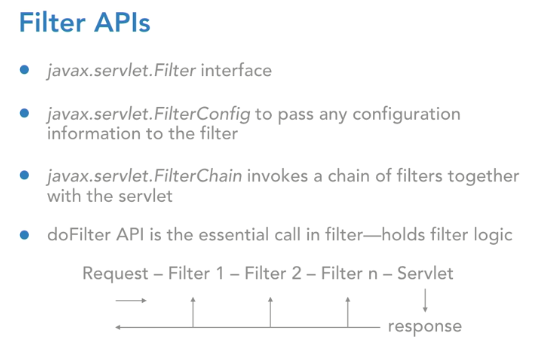
URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies. **The drawback here is that you will have to generate every URL dynamically** to assign a session ID though page is a simple static HTML page.





Filters are extremely crucial components in a web application, because for one, they promote modularity. What does that mean? They give you a way to separate out those administrative jobs from your actual request processing logic jobs which means you're making your code modular. Moreover, the code that you may write in a filter can be applied across various use cases of your application, which essentially means, that you're promoting reusability and when there's modularity, reusability, of course the maintainability of your application is going to increase.





So, any kind of initialization and cleanup activity that you want to put in for this filter can go into the init and destroy methods respectively. If you look at the method of init, the signature looks like this, it's init, public, does not return anything and has got a parameter of FilterConfig. So, FilterConfig is that API, which is used to pass configuration information to this filter. So, let's say your filter wants to do any kind of pre-processing, post-processing logic, for which it needs configuration information.

**FilterChain.doFilter(req,res)---This line is going to take control to next servlet**

The filters that you can configure in your application and the servlet together form a filter chain. And the do filter call is going to ensure that you make those hops from filter number one to filter number two, to filter number N and then finally, the servlet.



Annotating the Filter class:





**public void doFilter (ServletRequest, ServletResponse, FilterChain)**

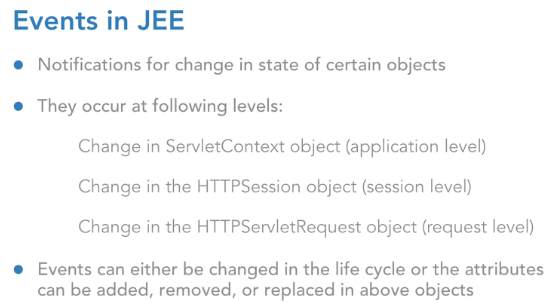
This method is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain

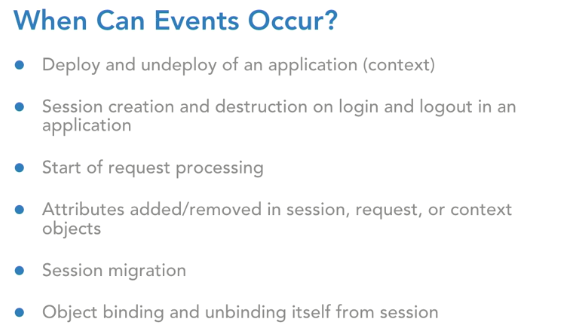
**public void init (FilterConfig filterConfig)**

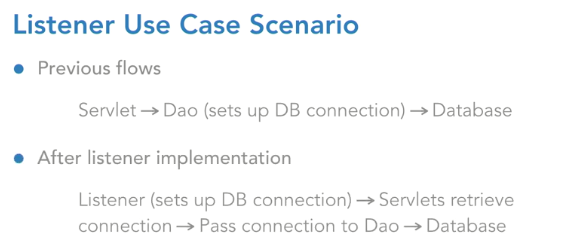
**public void destroy ()**

List of request attributes that an error-handling servlet can access to analyze the nature of error/exception.

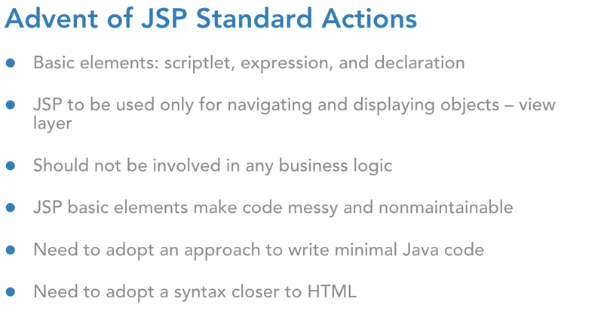
1. javax.servlet.error.status\_code
2. javax.servlet.error.exception\_type
3. javax.servlet.error.message
4. javax.servlet.error.request\_uri
5. javax.servlet.error.exception
6. javax.servlet.error.servlet\_name

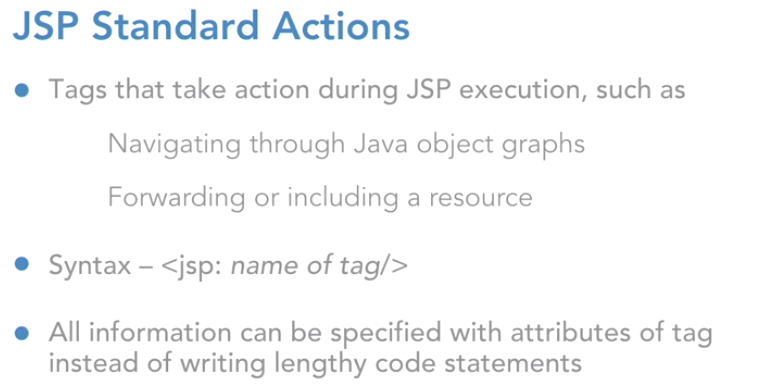


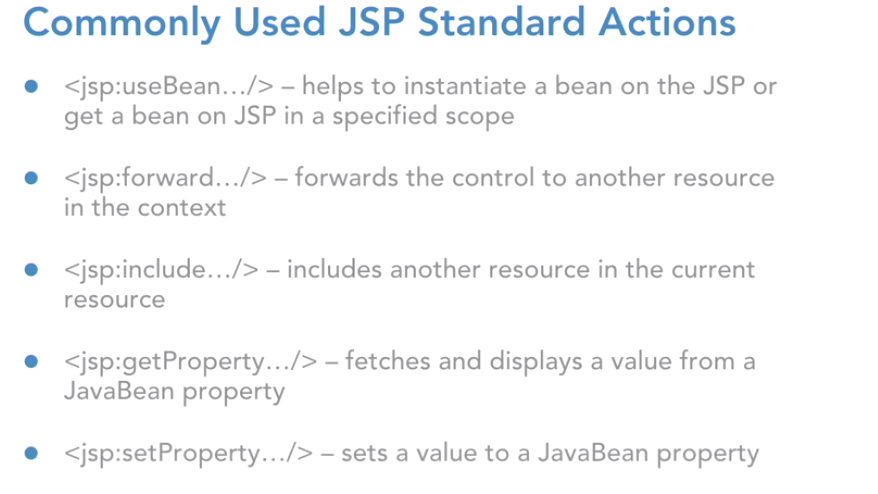












## **Common Attributes**

There are two attributes that are common to all Action elements: the **id** attribute and the **scope** attribute.

### **Id attribute**

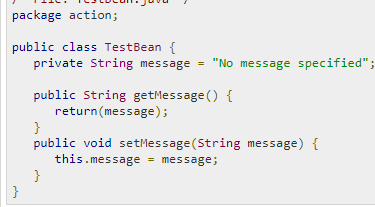
The id attribute uniquely identifies the Action element and allows the action to be referenced inside the JSP page. If the Action creates an instance of an object, the id value can be used to reference it through the implicit object PageContext.

### **Scope attribute**

This attribute identifies the lifecycle of the Action element. The id attribute and the scope attribute are directly related, as the scope attribute determines the lifespan of the object associated with the id. The scope attribute has four possible values: **(a) page, (b)request, (c)session**, and **(d) application**

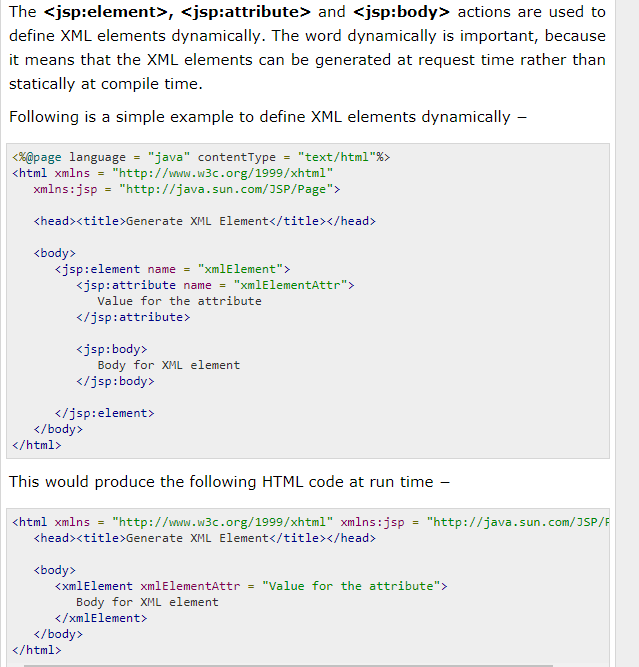
**Examples of JSP standard actions:**

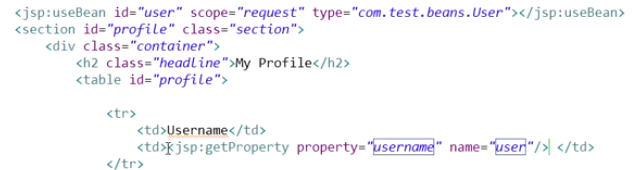










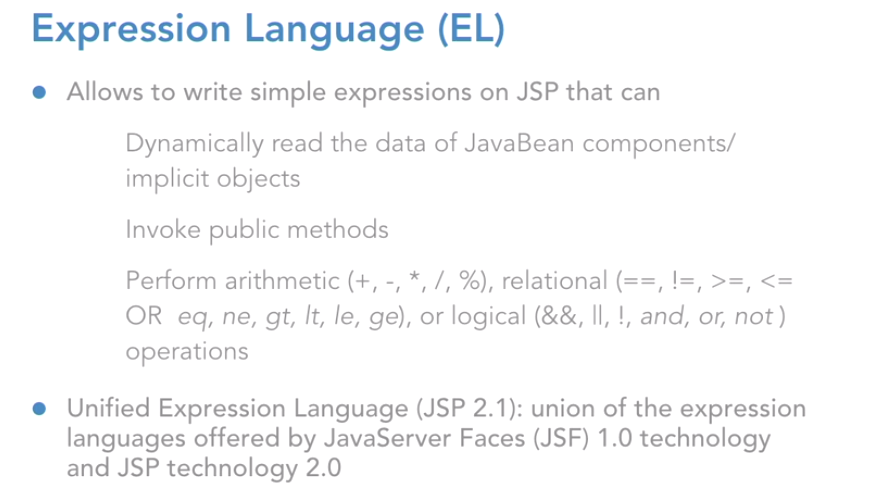


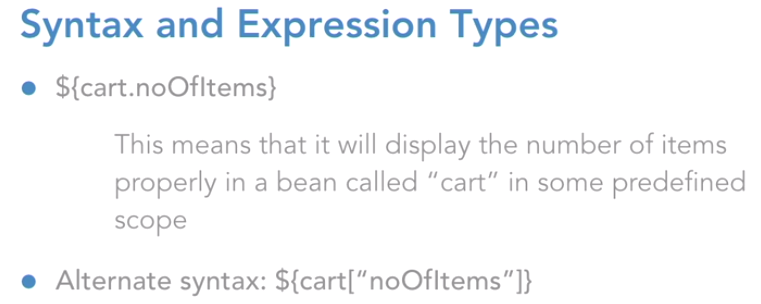
Had you not used JSP standard actions and had you stuck to expressions and scriptlets then, you would have to write down the code of request dot get attribute in a scriptlet to replace this line. But, now do you see that we don't have to write code at all and we can stick to a simple tag which will do the entire job same as what was being done by the scriptlet. So, now that we are ready with the JSP use bean standard action tag, let's go ahead and start displaying the data out of this.

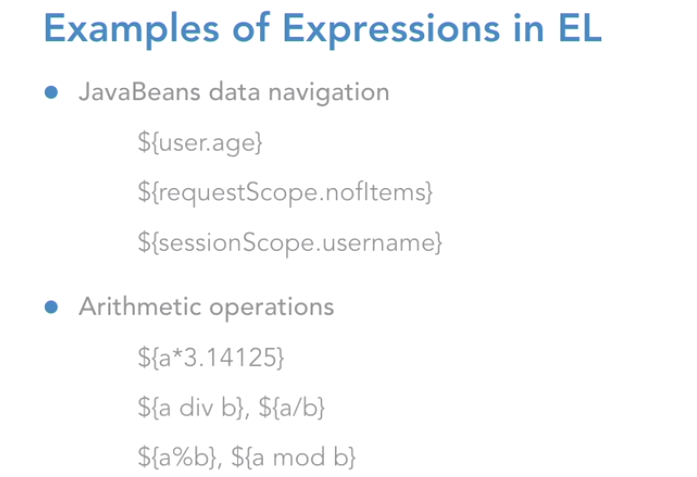


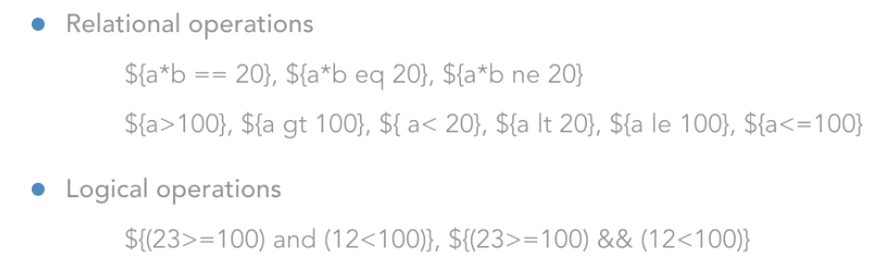
What's the difference between the include directive and the JSP include standard action. So, the difference essentially is that when you use the include directive then the included resource and the current resource, they're both going to be combined and then, it is going to be translated into a servlet file.

However, when you use a JSP include standard action then, the current JSP will be translated separately and this one, header JSP will be translated again separately. And then, at run time, those responses will be combined, and they'll be rendered to the user. But, for the include directive, the translation itself involves the combination and then entirely that thing is going to be translated to the servlet file. So, you are going to use JSP include in your projects. If you think that you want to execute some code in the included resource. Dynamic execution if it's required in your included resource. Stick to JSP include standard action. However, if you're just trying to include static resources, like for example header dot JSP in this case is static.



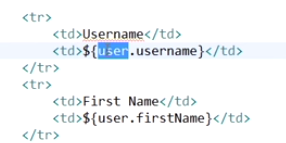






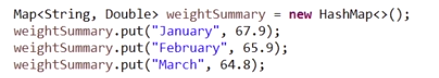
Now you must be wondering how it understood that, it must pick up this user object from the request scope.

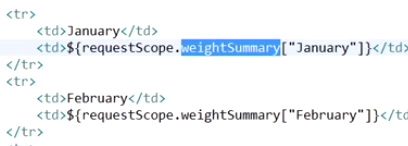
Because when we specified the JSP standard actions mechanism we are explicitly mention that you're supposed to pick up this object from the request scope. However, here, in expression language, you don't have to do that. Because expression language is smart enough. What it does, is, if you have not mentioned any scope here in your expression itself, then it is going to search this object by the key user, by the name user, in those four scopes. The page scope, the request, the session, and then the application, in that order.

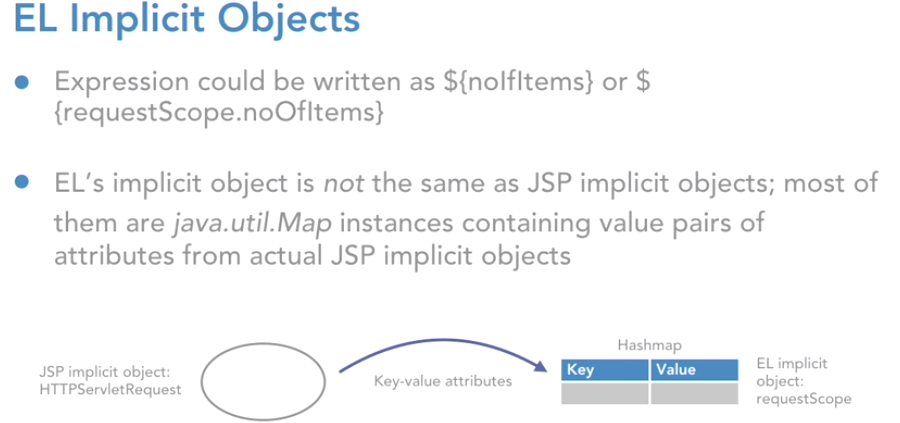


You must also be wondering as to where is this request scope name coming from? How did I write request scope? That is because, if you look at a JSP implicit objects, we have request as the implicit object for the request. Right? But in expression language, the implicit object, which can help you access the data of the request object, the name for it is request scope

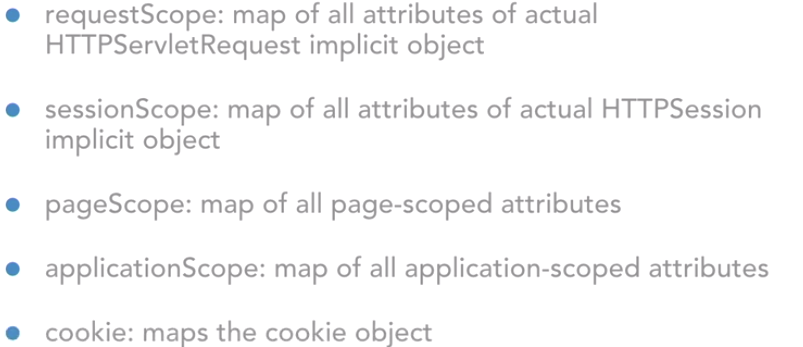
**How to access a Map in Jsp:**

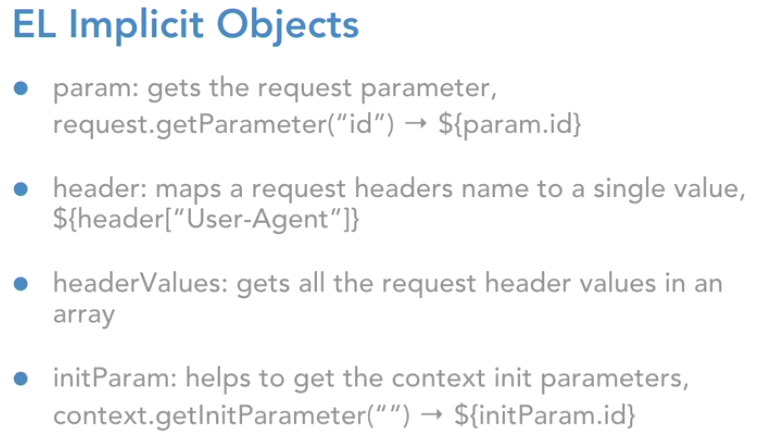


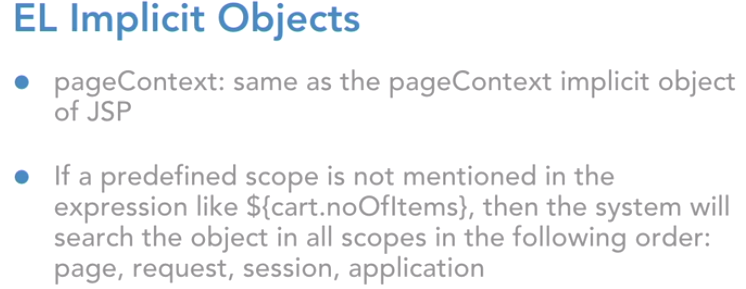




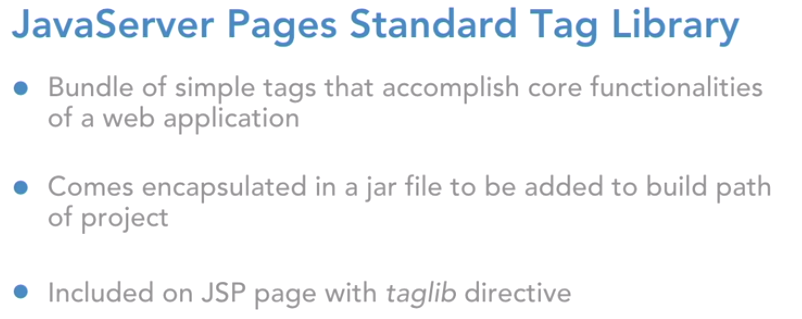
Let's understand that. When you talk about EL's implicit objects, please understand that they are not the same as our jsp implicit objects. There were nine implicit objects in jsp that we had talked about, and those are the ones that are of specific type. HTTP servlet request, HTTP servlet response, then you had servlet context, servlet config, then you had HTTP session etc, right? But when you talk about expression language, the implicit objects are not the same as you have in jsp.

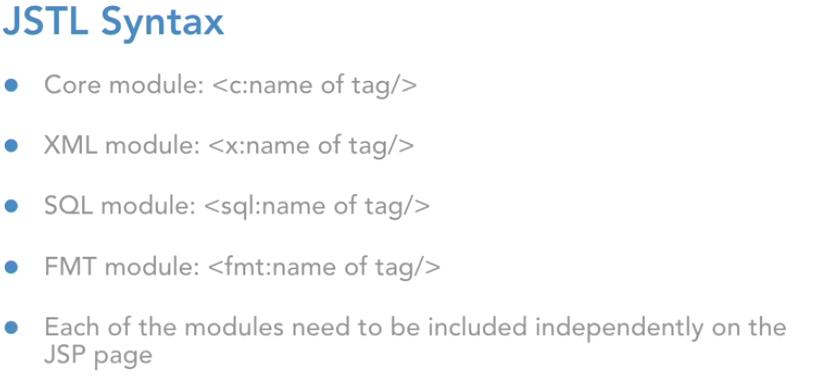






|  |  |
| --- | --- |
| pageScope | Scoped variables from page scope |
| requestScope | Scoped variables from request scope |
| sessionScope | Scoped variables from session scope |
| applicationScope | Scoped variables from application scope |
| param | Request parameters as strings |
| paramValues | Request parameters as collections of strings |
| header | HTTP request headers as strings |
| headerValues | HTTP request headers as collections of strings |
| initParam | Context-initialization parameters |
| cookie | Cookie values |
| pageContext | The JSP PageContext object for the current page |



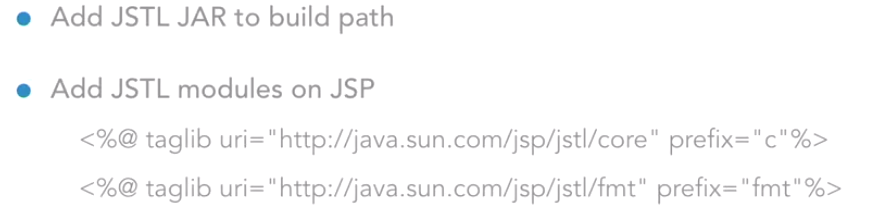


How to use the tag in Jsp:

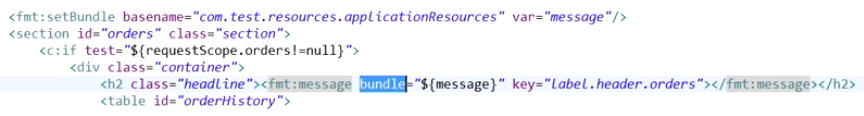


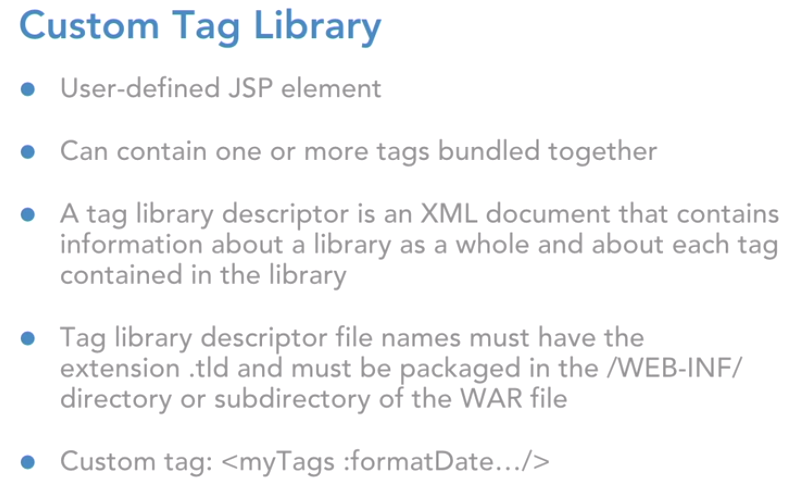
Iterating the list using jstl in Jsp:

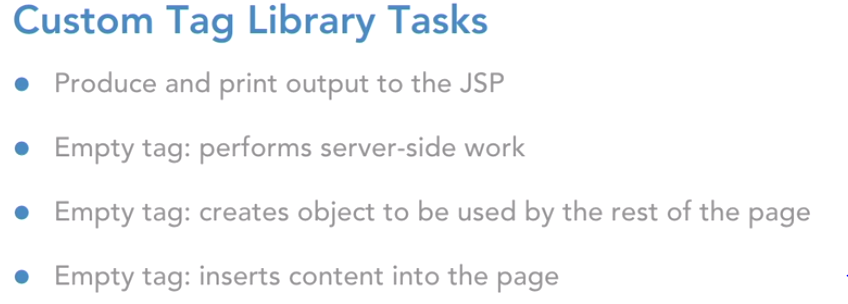


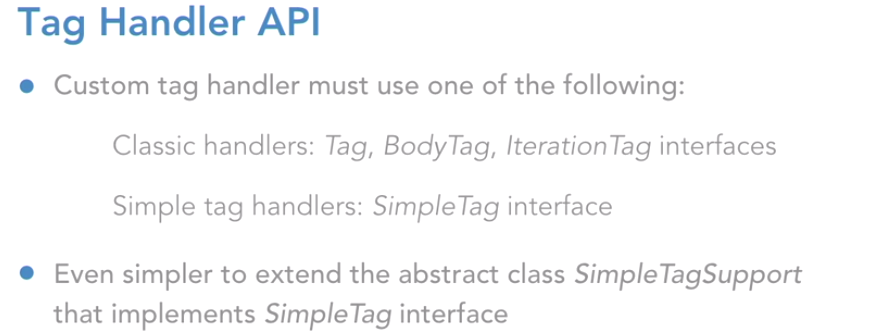


Internalization concept:

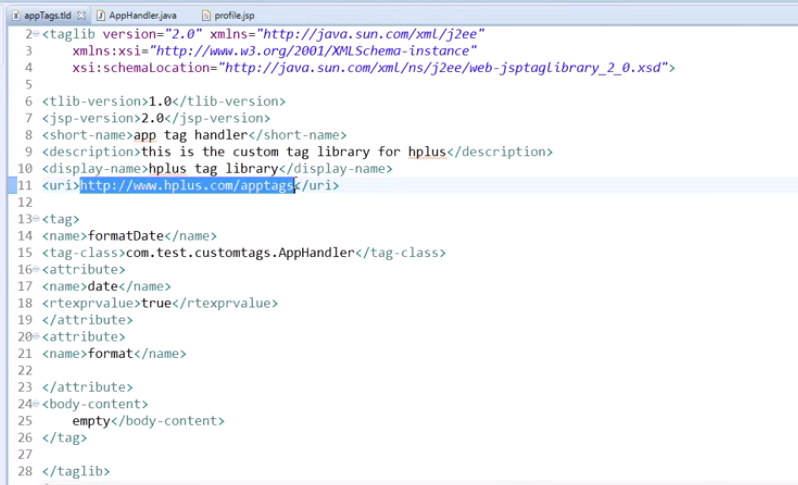




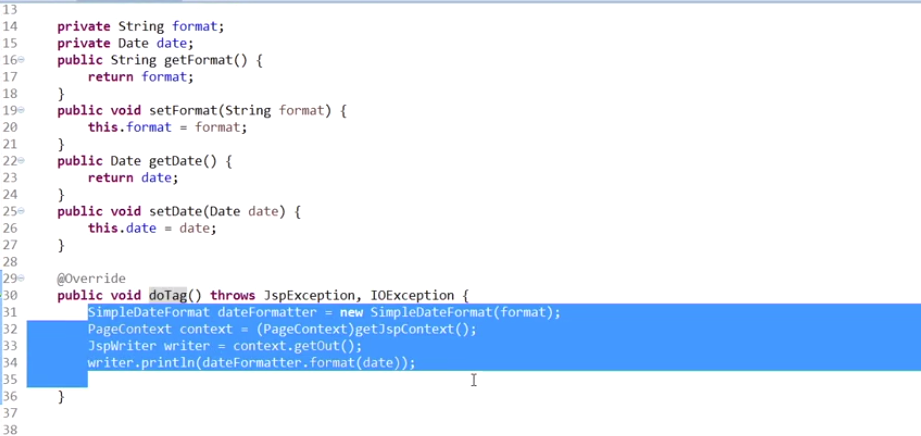




1. **Create a tld file**



1. **Create the Tag handler class**



1. Use the custom tag in the jsp file:

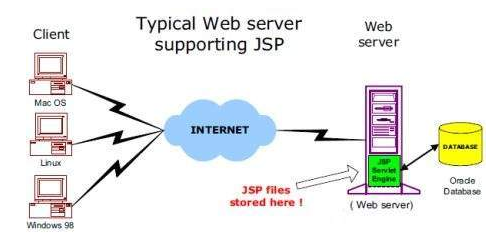




JavaBeans and taglib fundamentals were introduced for reusability. But following are the major differences between them −

* Taglibs are for generating presentation elements while JavaBeans are good for storing information and state.
* Use custom tags to implement actions and JavaBeans to present information.

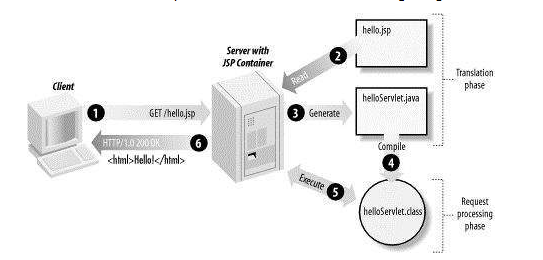
**JSP – Architecture**



The web server needs a JSP engine, i.e. a container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. This tutorial makes use of Apache which has built-in JSP container to support JSP pages development.

A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs.

**JSP Processing:**



1. Browser sends an HTTP request to the web server.
2. The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with “.jsp” instead of “**.html**”
3. The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println() statements and all JSP elements are converted to Java code. This code implements the corresponding dynamic behavior of the page.
4. The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
5. A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format. The output is further passed on to the web server by the servlet engine inside an HTTP response.
6. The web server forwards the HTTP response to your browser in terms of static HTML content
7. The JSP engine checks to see whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with the other scripting languages (such as PHP) and therefore faster.
8. So, in a way, a JSP page is just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet.

**JSP - Auto Refresh**

The simplest way of refreshing a Webpage is by using the **setIntHeader()**method of the response object. Following is the signature of this method −

public void setIntHeader(String header, int headerValue)

This method sends back the header "Refresh" to the browser along with an integer value which indicates time interval in seconds.

## **Classification of The JSTL Tags**

## **Core Tags**

The core group of tags are the most commonly used JSTL tags. Following is the syntax to include the JSTL Core library in your JSP −

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

## **Formatting Tags**

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Websites. Following is the syntax to include Formatting library in your JSP −

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

## **SQL Tags**

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as Oracle, mySQL, or Microsoft SQL Server. Following is the syntax to include JSTL SQL library in your JSP −

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

## **XML tags**

The JSTL XML tags provide a JSP-centric way of creating and manipulating the XML documents.

The JSTL XML tag library has custom tags for interacting with the XML data. This includes parsing the XML, transforming the XML data, and the flow control based on the XPath expressions.

<%@ taglib prefix = "x" uri = "http://java.sun.com/jsp/jstl/xml" %>

## **JSTL Functions**

JSTL includes several standard functions, most of which are common string manipulation functions. Following is the syntax to include JSTL Functions library in your JSP −

<%@ taglib prefix = "fn" uri = "http://java.sun.com/jsp/jstl/functions" %>

## **Accessing JavaBeans**

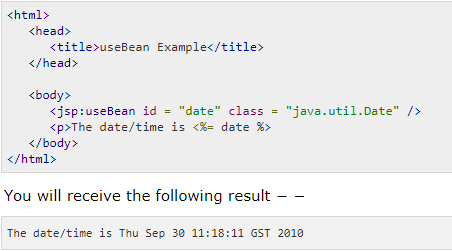
The useBean action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP.

**Syntax**:

<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>

Here values for the scope attribute can be a **page, request, session** or **application based** on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other **useBean declarations** in the same JSP.

**Default scope attribute of <jsp:useBean> is page**



**Ex:**



# JSP - Custom Tags

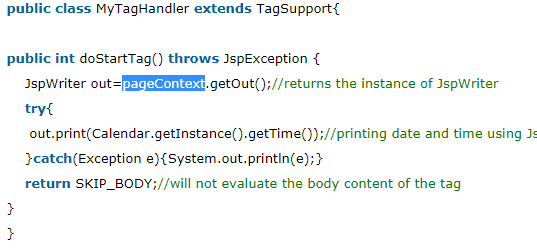
# A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag i translated into a servlet, the tag is converted to operations on an object called a tag handler. The Web container then invokes those operations when the JSP page's servlet is executed.

To write a custom tag, you can simply extend SimpleTagSupport class and override the doTag() method.

Create “Hello” tag

**Step1**: Create a class and extend SimpleTagSupport class



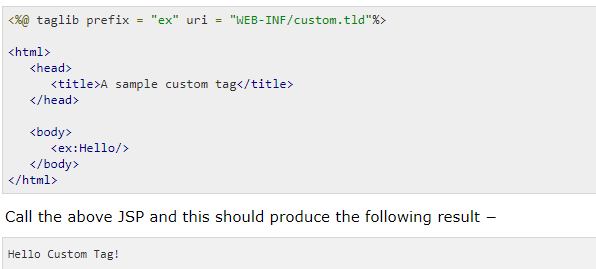


The above code has simple coding where the doTag() method takes the current JspContext object using the getJspContext() method and uses it to send "Hello Custom Tag!" to the current JspWriter object

**Step2: Create a tld file inside WEB-INF\custom.tld**



**Step3: Use it in the JSP:**



## **Custom Tag Attributes**

|  |  |
| --- | --- |
| **Property** | **Purpose** |
| **name** | The name element defines the name of an attribute. Each attribute name must be unique for a tag. |
| **required** | This specifies if this attribute is required or is an optional one. It would be false for optional. |
| **rtexprvalue** | Declares if a runtime expression value for a tag attribute is valid. |
| **type** | Defines the Java class-type of this attribute. By default, it is assumed as **String.** |
| **description** | Informational description can be provided. |
| **fragment** | Declares if this attribute value should be treated as a **JspFragment**. |

# 

### **The doStartTag() Method**

### Implementing the Tag interface implies defining a doStartTag() method. A Tag Handler class can either do that or extend the TagSupport class and override its doStartTag() method to begin the action.

public int doStartTag() throws JspException {

return SKIP\_BODY/ EVAL\_BODY\_INCLUDE

}

### **The doInitBody() Method**

The doInitBody() method can be used to do some processing before anybody content is evaluated (into the BodyContent output stream). After the invocation of doInitBody(), the tag body content is evaluated and the doAfterBody() method is invoked.

### **The doAfterBody() Method**

The doAfterBody() method is invoked after the first evaluation of the body content (if it is not empty). By returning SKIP\_BODY, the doAfterBody() method can tell the JSP container that the processing of the body content is finished:

public int doAfterBody() throws JspException {

// do something, or nothing

return SKIP\_BODY;

}

Sometimes the tag handler class must continually process the body content of a custom tag in a loop. You can tell the JSP container to repeatedly evaluate the body content and doAfterBody() invocation by returning EVAL\_BODY\_TAG from doAfterBody(). Note that because processing can change the body content or its context, each body content evaluation can have differing results.

public int doAfterBody() throws JspException {

//do something, or nothing

return EVAL\_BODY\_TAG;

}

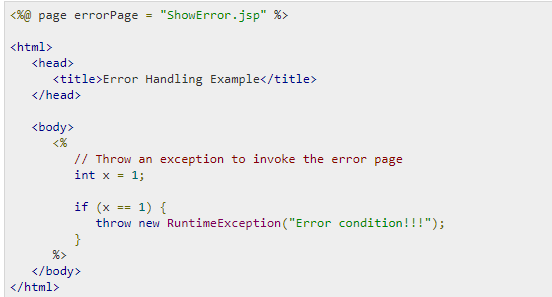
### **The doEndTag() Method**

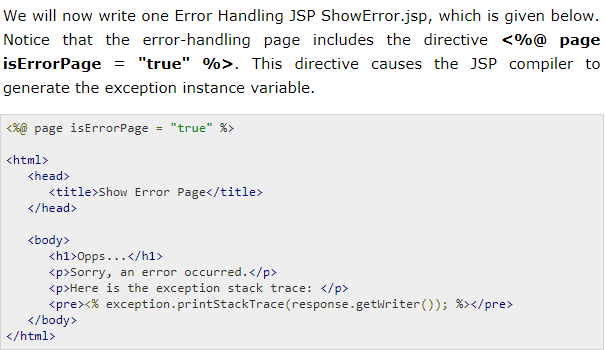
Whether only the Tag interface or the BodyTag interface is implemented by a Tag Handler class, the doEndTag() method is invoked by the container. It can be used for any final processing in the action, whether body content has been evaluated or not. The container will call the release() method to release tag state.

# JSP - Exception Handling

JSP gives you an option to specify **Error Page** for each JSP. Whenever the page throws an exception, the JSP container automatically invokes the error page.

Following is an example to specifiy an error page for a **main.jsp**. To set up an error page, use the **<%@ page errorPage = "xxx" %>** directive.





### **JSP Comments**

### JSP comment marks text or statements that the JSP container should ignore. A JSP comment is useful when you want to hide or "comment out", a part of your JSP page.

Following is the syntax of the JSP comments −

<%-- This is JSP comment --%>

Following example shows the JSP Comments −

<html>

<head><title>A Comment Test</title></head>

<body>

<h2>A Test of Comments</h2>

<%-- This comment will not be visible in the page source --%>

</body>

</html>

**<%-- comment --%>** A JSP comment. Ignored by the JSP engine.

**<! -- comment -->** An HTML comments. Ignored by the browser.

**<\%** Represents static <% literal.

**%\>** Represents static %> literal.

**\'** A single quote in an attribute that uses single quotes

**\"** A double quote in an attribute that uses double quotes.