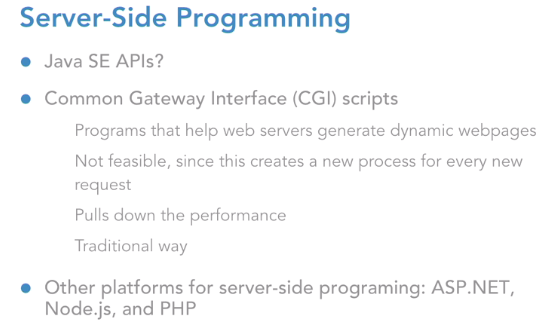
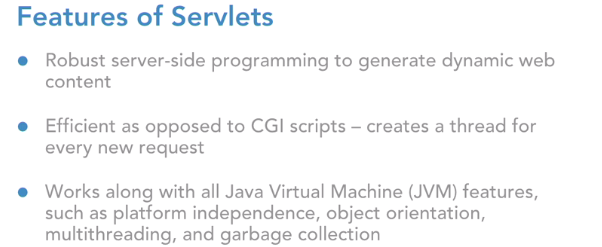


Entire chain of requests and response that gets exchanged between the web client and the web server is nothing but over the H-T-T-P protocol. Hypertext transfer protocol is the only communication strategy between the web client and the server. They do not understand any other language except H-T-T-P.

if our web server needs to act and process a request, it needs to be fed a set of instructions. Let's understand, what option do we have for server-side programming?



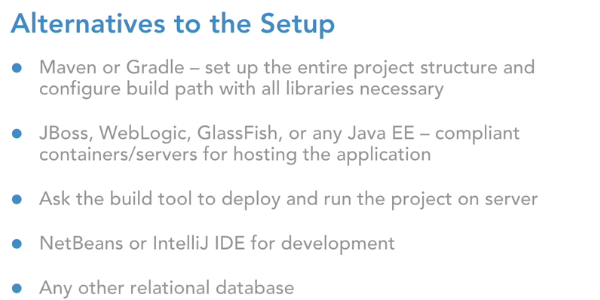
we are into the enterprise world of Java, and we need an A-P-I so that we can code server-side programming features. And that is exactly where servlets pitch in. So, let us look at the features of servlets. Servlets offer us a robust server-side programming option to generate dynamic web content.

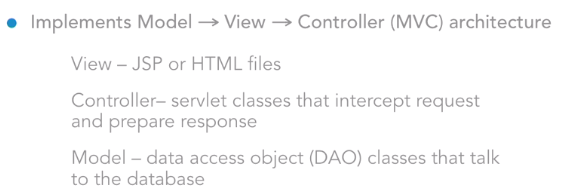


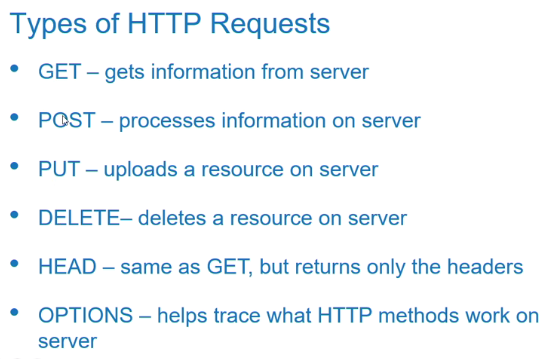


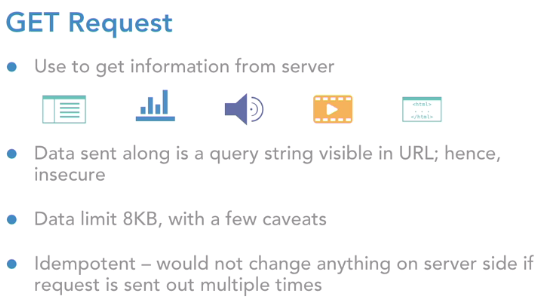
J-S-R, which is the Java Specification Request, the number 315 was released by Java community for version 3.0 as a part of Java Enterprise Edition number six, which works with J-D-K 1.6 and then there is J-S-R 340, which is for version 3.1, which was released as a part of Java E-E seven. Please understand 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.

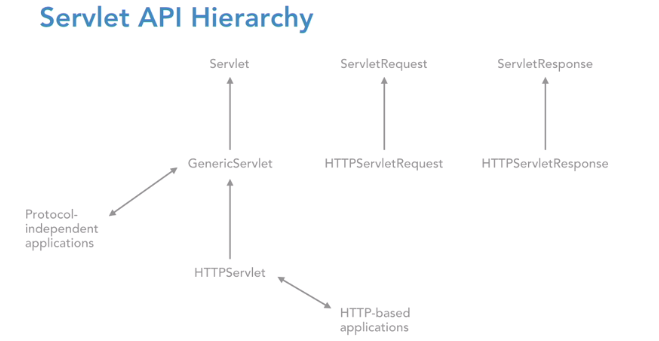
The servlets are going to work in conjunction with our code Java A-P-Is to help us build a wonderful web application.





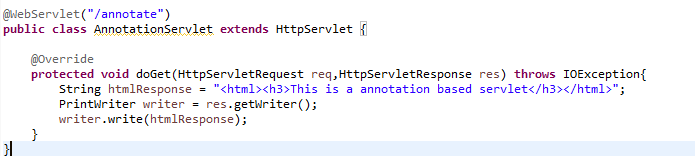




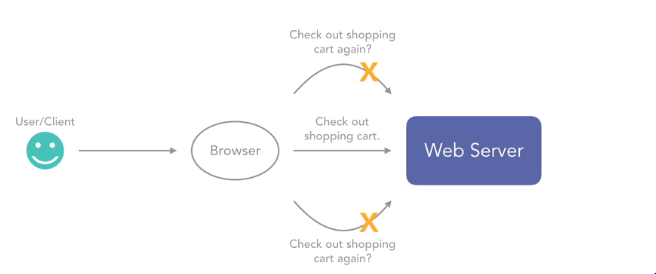


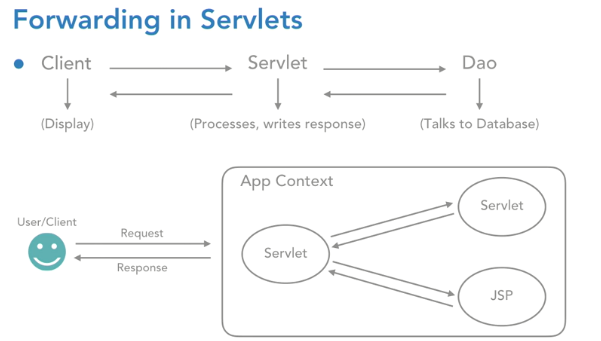
So, on the response object I have an API which is called **getWriter**. This is going to return me a print writer reference. Print Writer is an API under the Java IO package which will help us to exclusively write response on a webpage.

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:

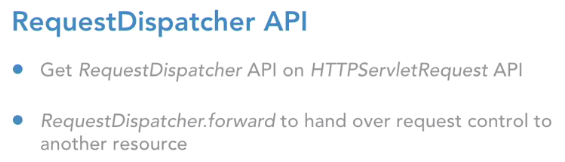








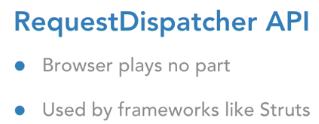
Request forwarding does not reveal any kind of implementation details in URL.



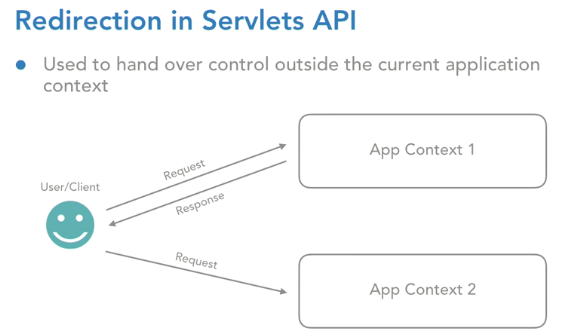


**Include** API combine the response of servlet and the response of jsp file together and render it to the client.

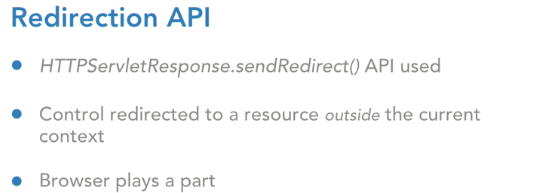




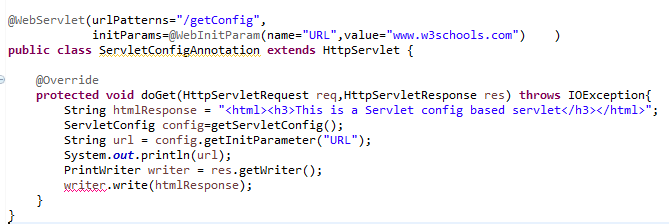
**RequestDispatcher** used to forward the control from one resource to another resource, inside the same application. And in this entire process, the browser will play no part. Entirely, it's managed by the server site. This approach is very commonly used in a framework called **Struts**, where whenever 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.

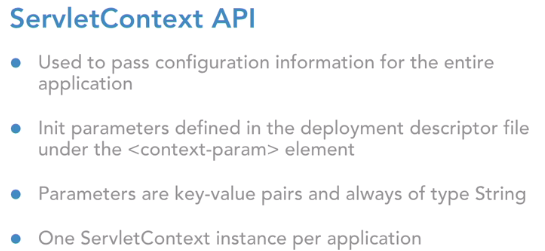


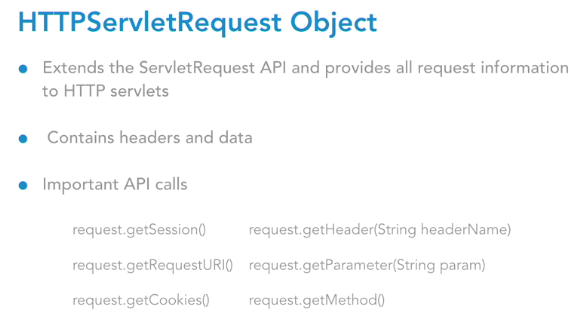


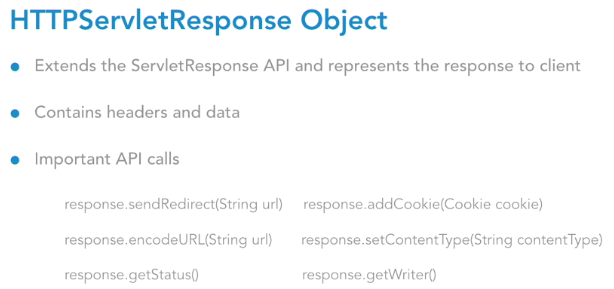


**Servlet Config: Configuring the Init parameter through annotation**









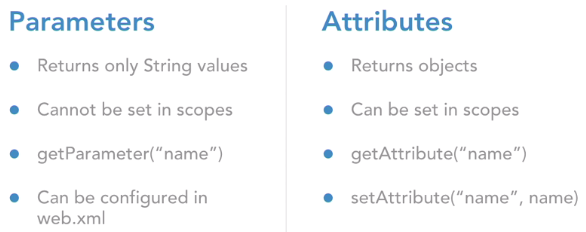


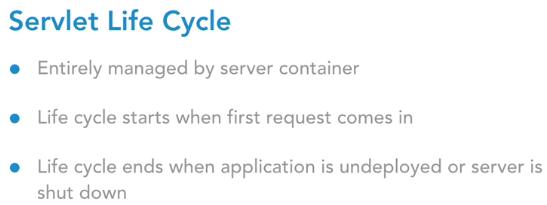
**Request** scope is alive until the response of this request goes back to client.

**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.

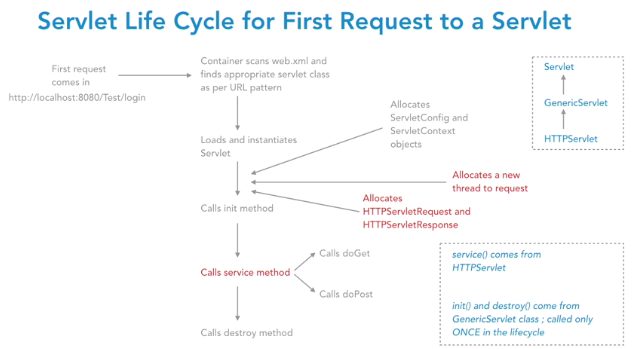
**Context**: context scope, which is of course sometimes known as the application scope, represented by the ServletContext object.

**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.

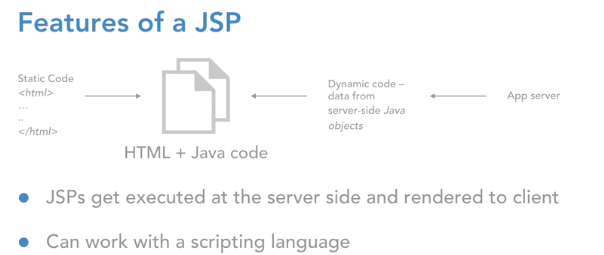
The logic that goes inside the service method is: depending on the kind of request that was made from the client, let's say it's a GET or a POST or anything like that, the service method is accordingly going to delegate the call to either the doGet or the doPost methods. And this is the place exactly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

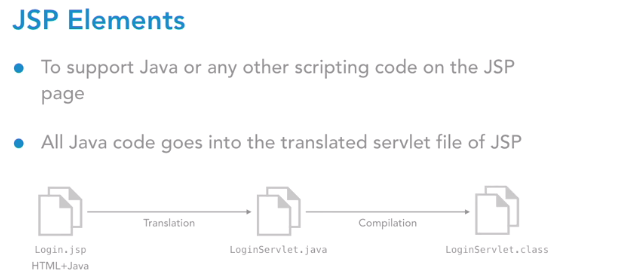
Now, once our application is about to get undeployed, or maybe say our server is absolutely shutting down altogether, that's the point when the destroy method of the servlet is called. Destroy method is going to hold any kind of clean up activity that we want to do for the servlet. So, maybe we have 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.

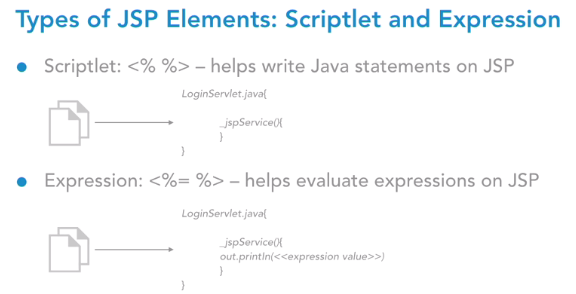
Now, the second phase, where there are multiple requests coming in for the same servlet, do we see some points highlighted in red in this slide? Those are the steps that happen again for repeated requests. The instance is only one. That does not happen again. Directly, a new thread will be allocated 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.

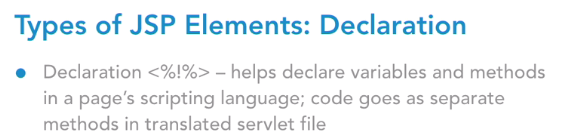
Like init, destroy is also called only once during the life cycle. So, they will never happen again and again. So, this is how the life cycle looks like overall. So, if we summarize the methods of the servlet life cycle, there are three life cycle methods: init, service, and destroy, out of which init 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.



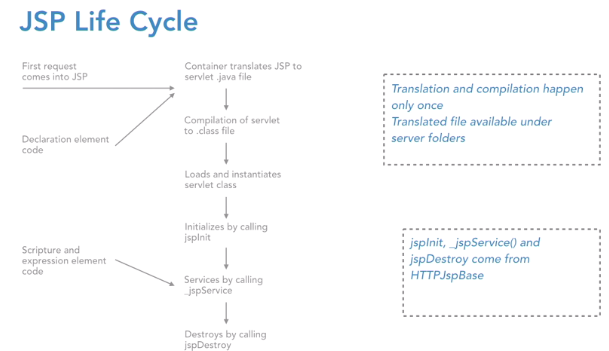


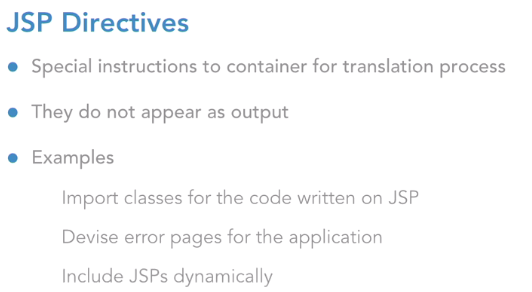


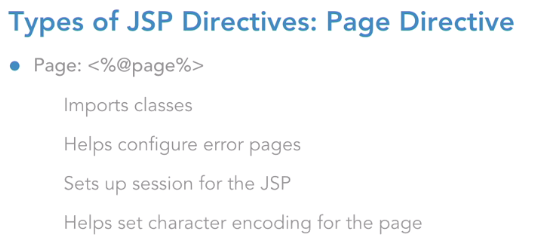


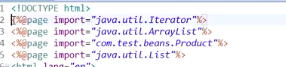
Implicit means it is by default available to the JSP page. We don’t have to create or import explicitly.

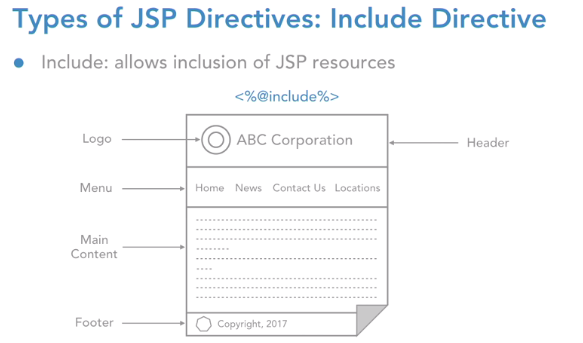
Declaration element allows you to introduce new variables or new methods in the translated file of jsp.

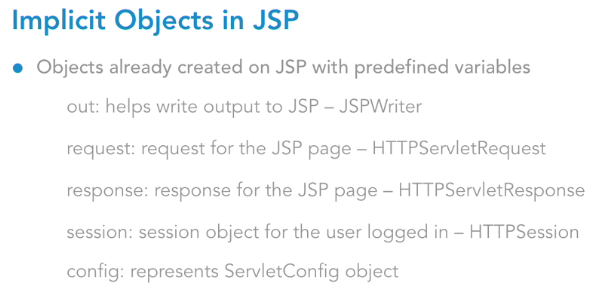


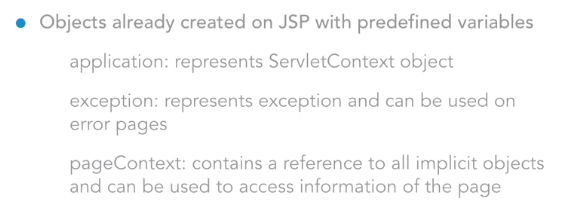


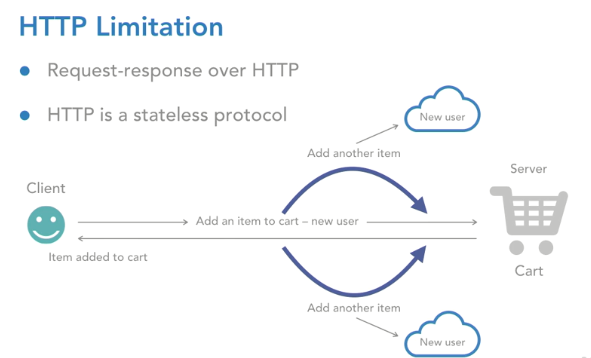


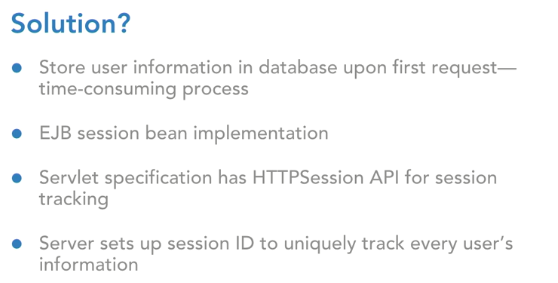


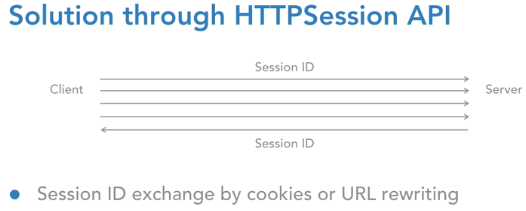


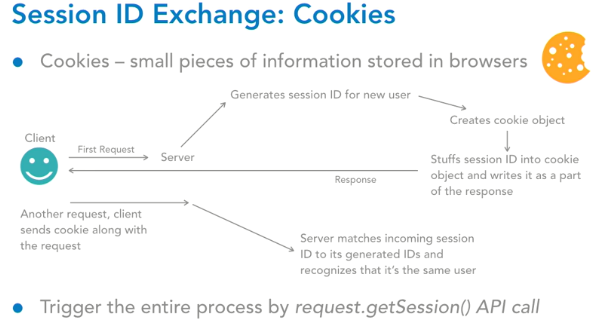










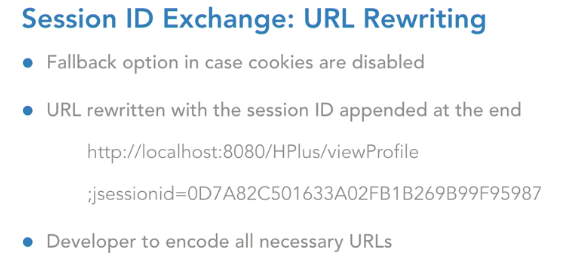


Cookies are small pieces of bits of information that are stored in the browser software. They could be information pertaining to the user or the application. So, let's understand how a session ID is generated and then how is it exchanged between the client and server via these cookies. Let's say the 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 have to 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.



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”)



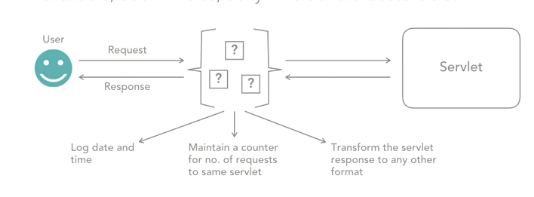
Session.invalidate () will invalidate the session.

Change in Java files require redeployment and restart.

Change in web.xml requires restart of the server.

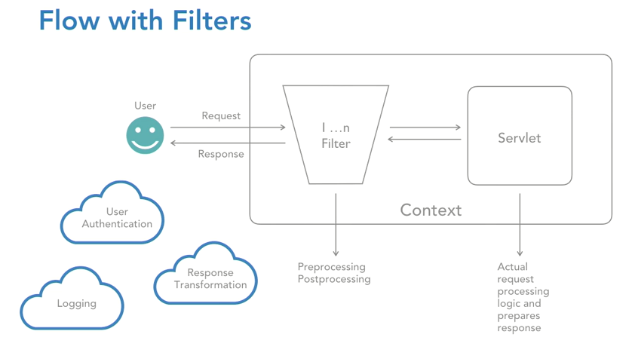
Configure the session timeout in web.xml

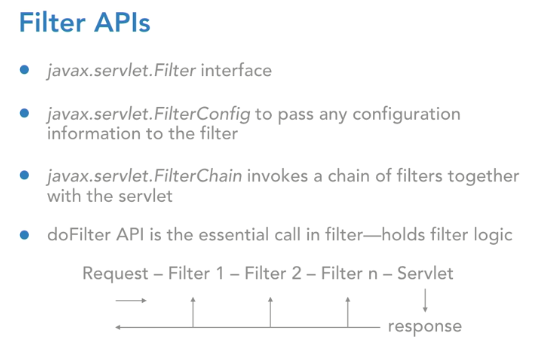






Filters are extremely crucial components in a web application, because for one, they promote modularity. What does that mean? They give you a way to separate out those administrative jobs from your actual request processing logic jobs which means you're making your code modular. Moreover, the code that 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**

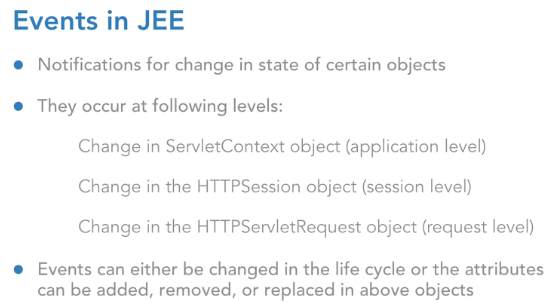
The filters that you can configure in your application and the servlet together form a filter chain. And the the do filter call is going to ensure that you make those hops from filter number one to filter number two, to filter number N and then finally, the servlet.

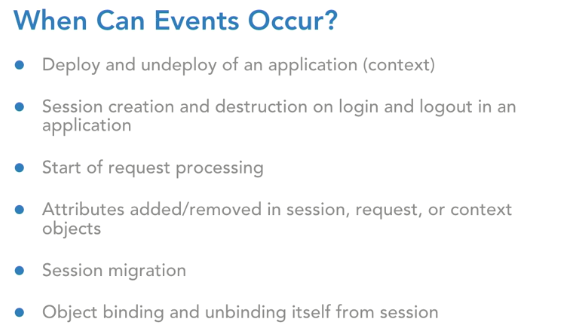


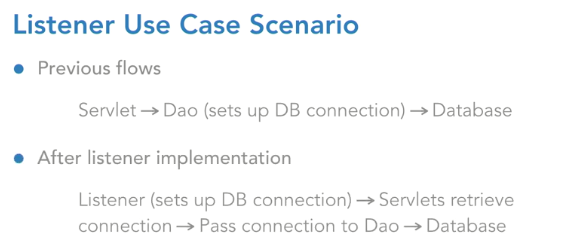
Annotating the Filter class:



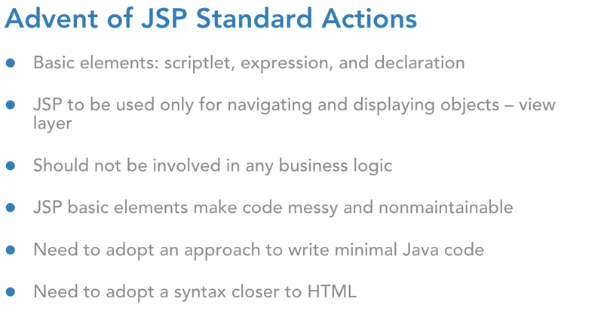


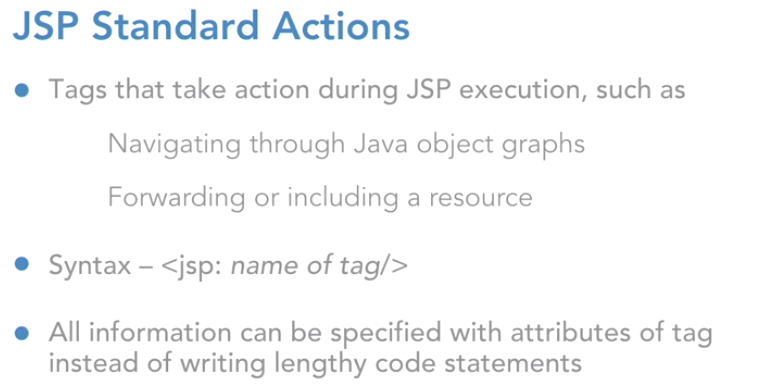


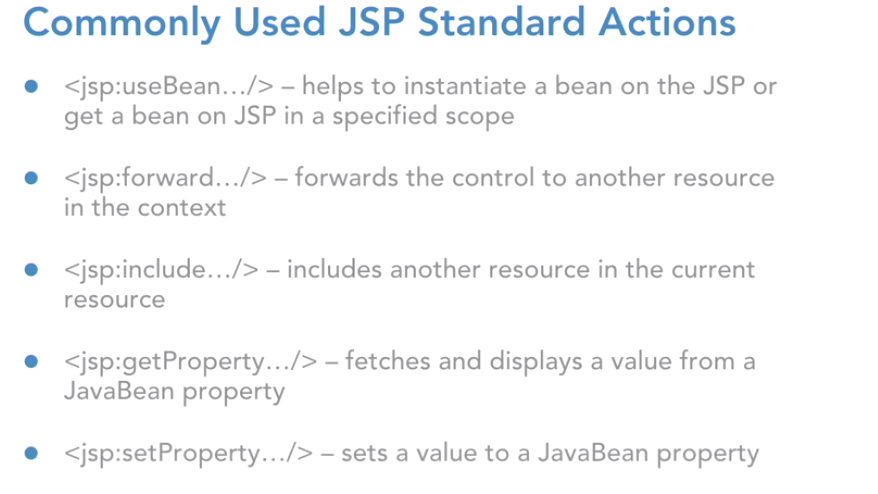


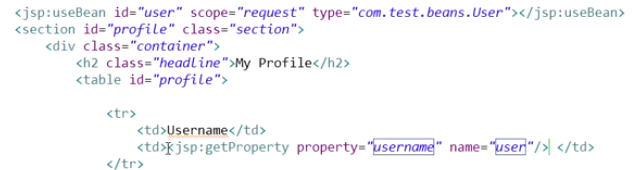










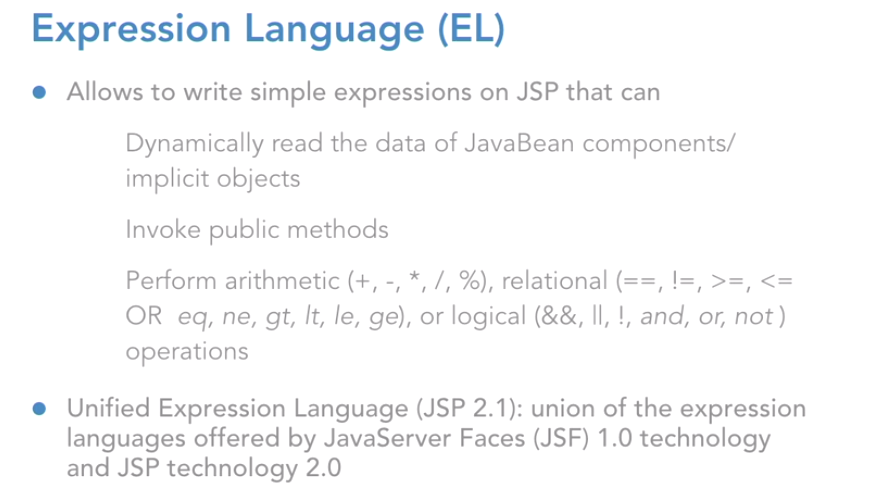


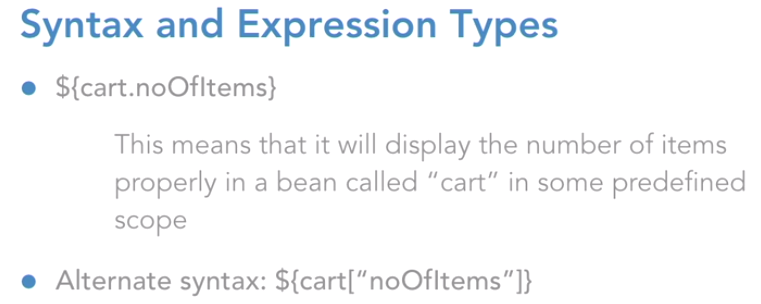
Had you not used JSP standard actions and had you stuck to expressions and scriptlets then, you would have to write down the code of request dot get attribute in a scriptlet to replace this line. But, now do you see that we don't have to write code at all and we can stick to a simple tag which will do 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.

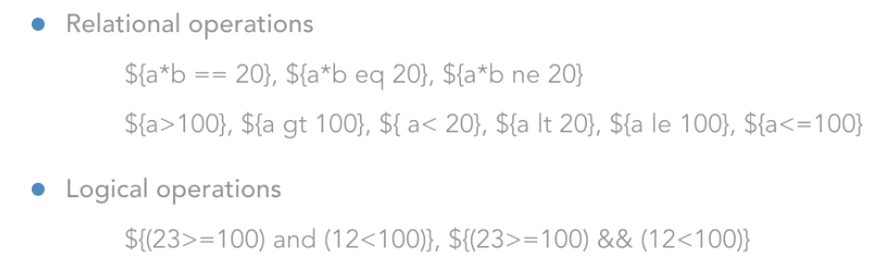
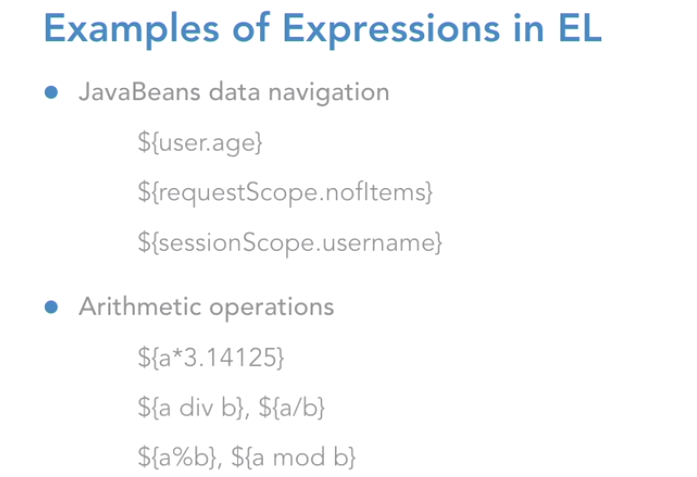


What's the difference between the include directive and the JSP include standard action. So, the difference essentially is that when you use the include directive then the included resource and the current resource, they're both going to be combined and then, it is going to be translated into a servlet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

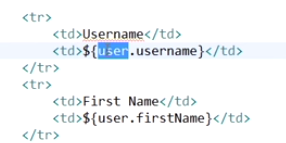






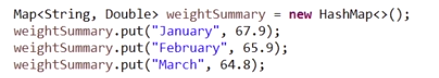
Now you must be wondering how it understood that, it must pick up this user object from the request scope.

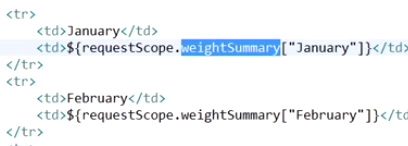
Because when we specified the JSP standard actions mechanism we are explicitly mention that you're supposed to pick up this object from the request scope. However, here, in expression language, you don't have to do that. Because expression language is smart enough. What it does, is, if you have not mentioned 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.

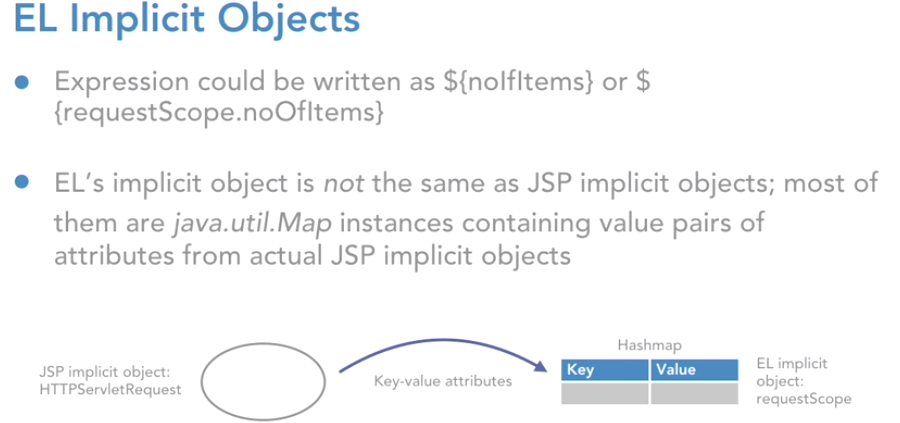


You must also be wondering as to where is this request scope name coming from? How did I write request scope? That is because, if you look at a JSP implicit objects, we have request as the implicit object for the request. Right? But in expression language, the implicit object, which can help you access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

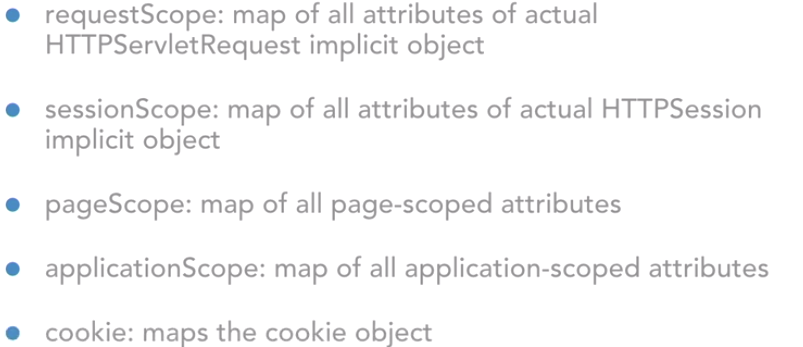
**How to access a Map in Jsp:**

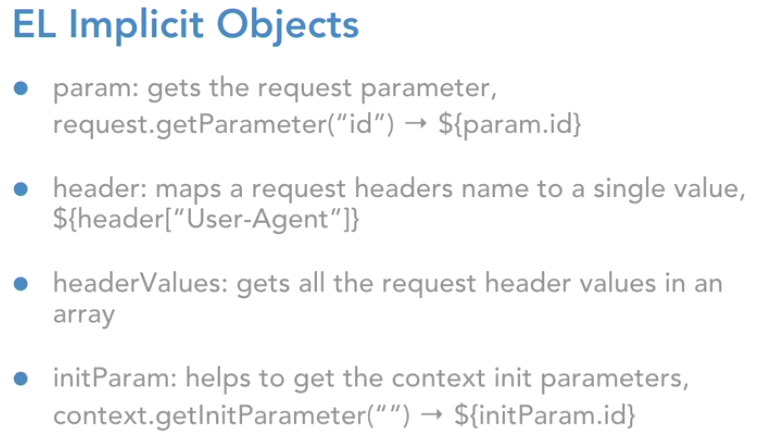


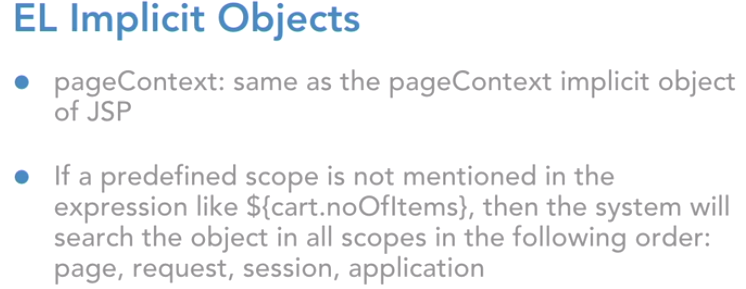


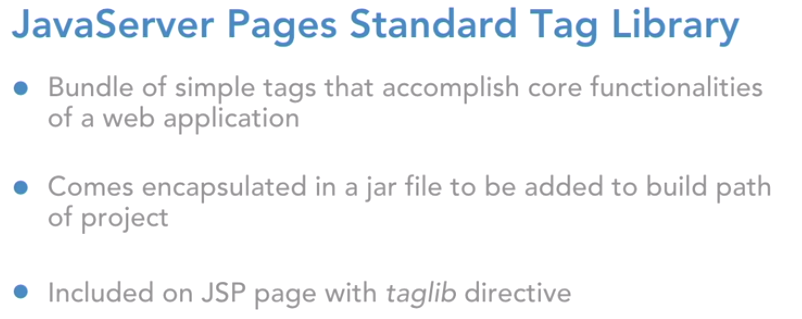


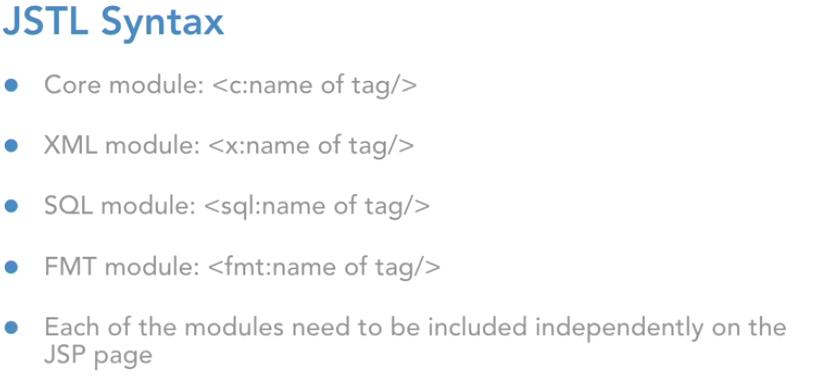
Let's understand that. When you talk about EL's implicit objects, please understand that they are not the same as our jsp implicit objects. There were nine implicit objects in jsp that we had talked about, and those are the ones that are of specific type. HTTP servlet request, HTTP servlet response, 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.









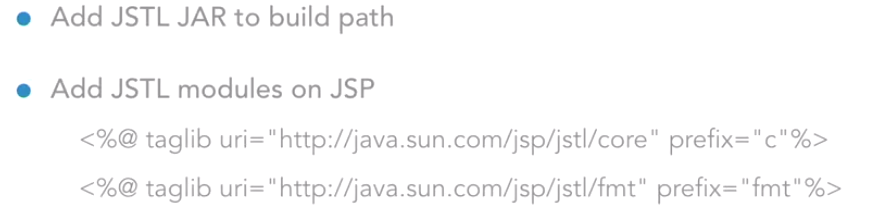


How to use the tag in Jsp:

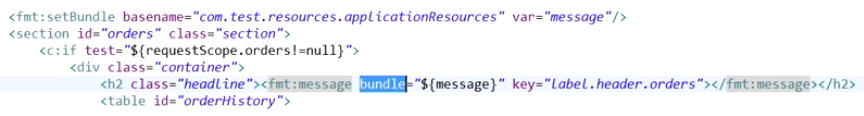


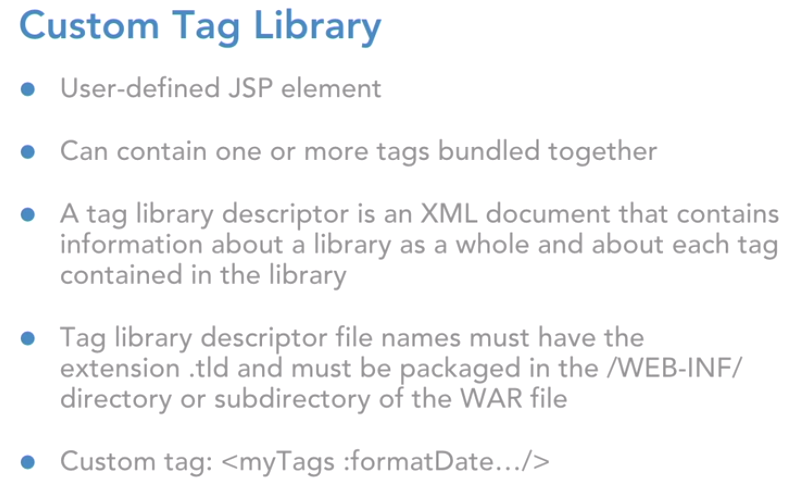
Iterating the list using jstl in Jsp:

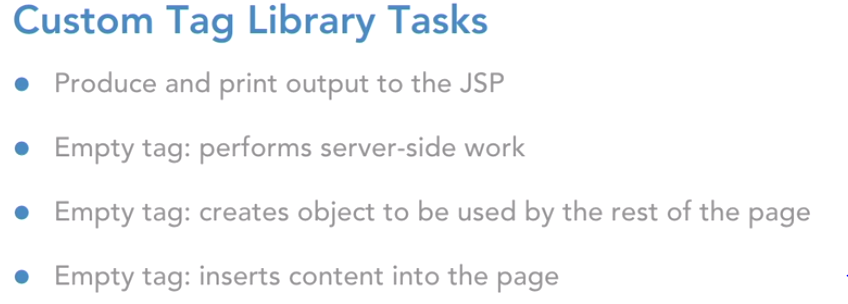


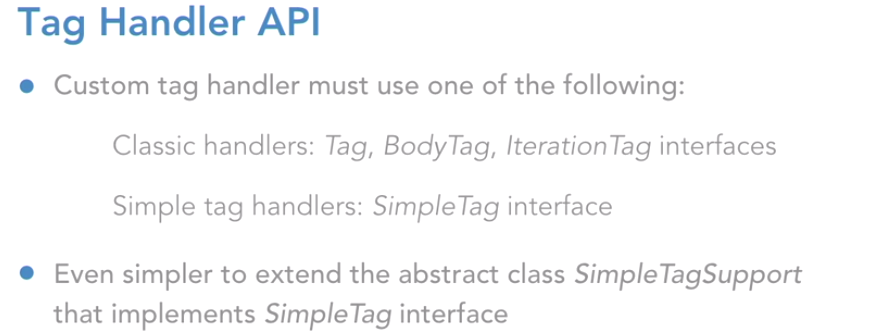


Internalization concept:

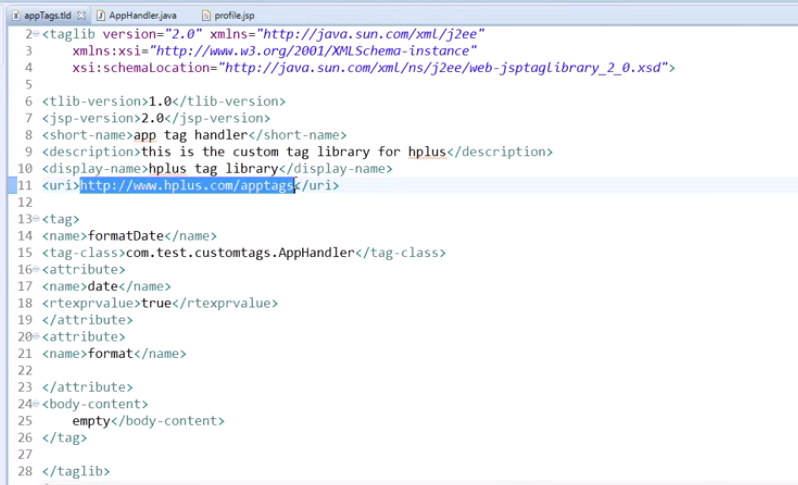




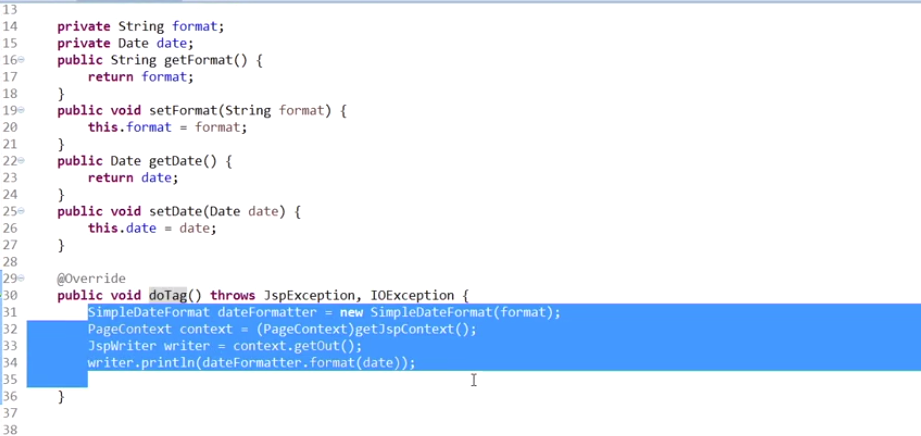




1. **Create a tld file**



1. **Create the Tag handler class**



1. Use the custom tag in the jsp file:



