# Introduction to Java Server Pages – JSP Tutorial

**JSP** is a server side technology that does all the processing at server. It is used for creating dynamic web applications, using java as programming language.

Basically, any html file can be converted to JSP file by just changing the file extension from “.html” to “.jsp”, it would run just fine. What differentiates JSP from HTML is the ability to use java code inside HTML. In JSP, you can embed Java code in HTML using JSP tags. for e.g. run the code below, every time you run this, it would display the current time. That is what makes this code dynamic.

****

The above JSP generates the following output:  
**Hello, Sample JSP code**.

**Explanation of above code**  
1) The line **<%–JSP Comment–%>** represents the JSP element called JSP Comment, While adding comments to a JSP page you can use this tag, we will discuss this in detail in coming posts.  
**Note:** JSP Comments must starts with a tag **<%–** and ends with **–%>**

**2) Head, Title and Body tags are HTML tags** – They are HTML tags, frequently used for static web pages. Whatever content they have is delivered to client(Web browser) as such.

3) **<%out.print(“ Hello, Sample JSP code ”);%>** is a JSP element, which is known as Scriptlet. Scriptlets can contain Java codes.  **syntax of scriptlet is:** <%Executable java code%>. As the code in Scriptlets is java statement, they must end with a semicolon(;). out.print(“ Hello, Sample JSP code ”) is a java statement, which prints“ Hello, Sample JSP code”.

As discussed, JSP is used for creating dynamic webpages. Dynamic webpages are usually a mix of static & dynamic content.

The **static content** can have text-based formats such as HTML,  XML etc and the **dynamic content** is generated by JSP tags using java code inside HTML .

## Servlet Vs JSP

Like JSP, Servlets are also used for generating dynamic webpages. Here is the comparison between them.

The major difference between them is that servlet adds HTML code inside java while JSP adds java code inside HTML. There are few other noticeable points that are as follows:  
**Servlets** –

1. Servlet is a Java program which supports HTML tags too.
2. Generally used for developing business layer(the complex computational code) of an enterprise application.
3. Servlets are created and maintained by Java developers.

**JSP** –

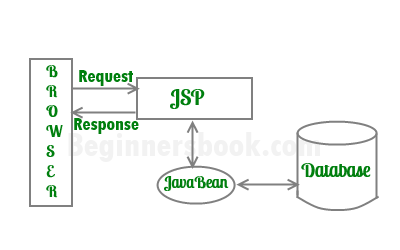
1. JSP program is a HTML code which supports java statements too.To be more precise, JSP embed java in html using JSP tags.
2. Used for developing  presentation layer of an enterprise application
3. Frequently used for designing websites and used by web developers.

## Advantages of JSP

1. JSP has all the advantages of servlet,  like: Better performance than CGI Built in session features,  it also inherits the the features of java technology like – multithreading, exception handling, Database connectivity,etc.
2. JSP Enables the separation of content generation from content presentation. Which makes it more flexible.
3. With the JSP, it is now easy for web designers to show case the information what is needed.
4. Web Application Programmers can concentrate on how to process/build the information.

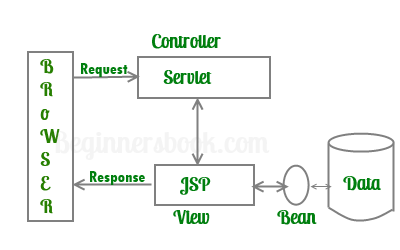
## Architecture of a JSP Application

Before we start developing web application, we should have a basic idea of architectures. Based on the location where request processing happens (Servlet OR JSP(java server pages)) there are two architectures for JSP. They are – Model1 Architecture & Model2 Architecture.  
**1) Model1 Architecture**: In this Model, JSP plays a key role and it is responsible for of processing the request made by client. Client (Web browser) makes a request, JSP then creates a bean object which then fulfils the request and pass the response to JSP. JSP then sends the response back to client. Unlike Model2 architecture, in this Model most of the processing is done by JSP itself.

****

**MODEL1**

**2) Model2 Architecture**: In this Model, Servlet plays a major role and it is responsible for processing the client’s(web browser) request. Presentation part (GUI part) will be handled by JSP and it is done with the help of  bean as shown in image below. The servlet acts as controller and in charge of request processing. It creates the bean objects if required by the jsp page and calls the respective jsp page. The jsp handles the presentation part by using the bean object. In this Model, JSP doesn’t do any processing, Servlet creates the bean Object and calls the JSP program as per the request made by client.

****

**MODEL2**

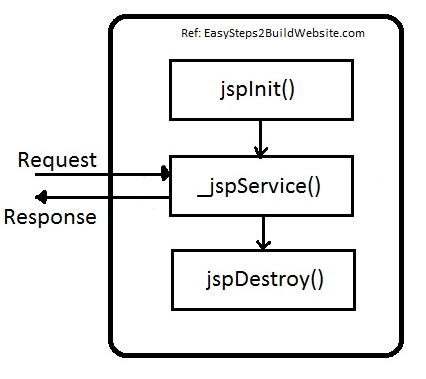
# Java Server Pages (JSP) Life Cycle

JSP pages are saved with .jsp extension which lets the server know that this is a JSP page and needs to go through JSP life cycle stages.  
In my previous post about JSP introduction, I explained that JSP is not processed as such, they first gets converted into Servelts and then the corresponding servlet gets processed by Server.

When client makes a request to Server, it first goes to container. Then container checks whether the servlet class is older than jsp page(  To ensure that the JSP file got modified). If this is the case then container does the translation again (converts JSP to Servlet) otherwise it skips the translation phase (i.e. if JSP webpage is not modified then it doesn’t do the translation to improve the performance as this phase takes time and to repeat this step every time is not time feasible)

**The steps in the life cycle of jsp page are:**

1. Translation
2. Compilation
3. Loading
4. Instantiation
5. Initialization
6. RequestProcessing
7. Destruction



**Let see the Life cycle of JSP in more detail –**  
1) As stated above whenever container receives request from client, it does translation only when servlet class is older than JSP page otherwsie it skips this phase (reason I explained above).  
2) Then the container –

* compiles the corresponding servlet program
* Loads the corresponding servlet class
* Instantiates the servlet class
* Calls the **jspInit() method** to initialize the servlet instance( Jsp container will do this job only when the instance of servlet file is not running or if it is older than the jsp file.)



**What are Web Applications?**

A web site where the HTML pages are generated dynamically.

Based on the user actions

**WEB** -----> **WEB**  ----> **DATA**

**BROWSER** <----- **SERVER**  <---- **BASE**

**What are JSP and Servlets?**

> Java Code that runs on web server

> Reads user's actions(inputs) ...normally from HTML Form

> Performs the work

> Returns an HTML page that is generated dynamically.

Basically, JSP AND SERVLETS run on WEB SERVER

**What types of Web Apps can we create?**

* Any industry
* E-Commerce
* Student/Employee Tracking
* Restaurant/Hotel Reservation
* Social Media
* Basically, any type of app

Note -

* JSPs and Servlets are key components of the Java Enterprise Edition (Java EE)
* Popular MVC frameworks are actually built on top of JSP and Servlets
  + - Spring
    - JSF
    - Struts

**What is a JSP?**

* An html page with some Java code sprinkled in.
* Include dynamic content from java code.

Where is the JSP processed?

* JSP is processed on the server.
* Results of java code included in HTML returned to browser.

Where to place JSP file?

* JSP file goes into **WebContent folder**.
* Must have .jsp extension.



<%=new java.util.Date%> is an JSP Expression. The output is included in the HTML. Calls toString() method on the object. A date object is basically created here. The browser doesn’t know how the code has been generated. All it does is simply renders the response.

**JSP Scripting Elements**

|  |  |
| --- | --- |
| Elements | Syntax |
| JSP Expression | **<%= some java expression %>** |
| JSP Scriptlet | **<% some java code: 1 to many lines %>** |
| JSP Declaration | **<%! Variable or method declaration %>** |

**JSP Expression**

* Compute an expression
* Result included in HTML to browser.

**Syntax: <%= some java expression %>**

**Examples:**

**JSP File**

**File used here:** helloworld.jsp

The time on the server is

<%=**new** java. util.Date()%>

**Generated Output**

The time on the server is Thu Jul 23 01:49:22 IST 2020

**JSP File**

**File used here:** jspExpressions.jsp



**Generated Output**

****

**JSP Scriptlets**

A scriptlet is a JSP construct. It allows you to add one to many lines of Java code.

**Syntax:**

**<%**

**//*Some lines of java code***

**%>**

**Note: Code will be executed top down when the page is processed.**

**Best Practices**

* Minimize the amount of scriptlet code in a JSP
* Avoid dumping thousands of lines of code in a JSP
* Refactor the code into a separate Java class or make use of MVC

**Examples:**

**JSP File**

**File used here:** scriptlet.jsp

****

**Generated Output**

****

**JSP Declarations**

* It helps us to declare a method in a JSP Page.
* And then it also helps to call the method in a JSP Page.

**Syntax**:

<%!

//***Declare a method***

%>

**Best Practices**

* Minimize the amount of declarations code in a JSP
* Avoid dumping thousands of lines of code in a JSP
* Refactor the code into a separate Java class or make use of MVC

**Examples:**

**JSP File**

**File used here:** jspDeclarations.jsp

****

**Generated Output**



**Call a Java Class from JSP *(Refactoring code into a separate JAVA class)***

JSP file is going to actually call a separate Java class. So, the Java class will have all of our code, all of our business logic, and so on, and the JSP can simply make a call, let the Java code or the Java class do the heavy lifting, and then the JSP can get the results and continue on with its processing.

**Steps:**

New 🡪 Dynamic Web Project 🡪 Java Resources 🡪 src 🡪 Rt Click 🡪 New 🡪 Package

Package 🡪 Rt Click 🡪 New 🡪 Class 🡪 Finish

**File used here: FunUtils.java**



Importing the class file in a JSP file.

File used here: fun-test.jsp



In the above code, the package is being imported first. Then using a JSP Expression the method has been used which is previously defined in the class file (FunUtils.java) of the package(com.sayantan).

**JSP Built-In Server Objects**

* Given for free
* Use them directly in JSP Page.

**Commonly used Built-In server objects**

|  |  |
| --- | --- |
| Object | Description |
| Request | Contains HTTPS request headers and form data |
| Response | Provides HTTP support for sending response |
| Out | JspWriter for including content in HTML Page |
| Session | Unique session for each user of the web application |
| Application | Shared data for all users of the web application |

**Request / Response**

We have the browser communicating with JSP. Behind the scenes, the HTTP protocol, they actually send over a request object. This request object contains header information and body information. The JSP can perform some work on that information and then send back a response. So, we have the whole request/response protocol going.

**JSP File**

**File Used here: builtin-test.jsp**



**Generated Output**

****

**Including files in JSP**

Say for example, we're building a website and we want to have the same header and footer information displayed on every page. So, we can simply create the header and footer in separate files and include them in your JSP files.

**Example:**

**Files used here: my-header.jsp; my-footer.jsp; homepage.jsp**

**my-header.jsp**



**my-footer.jsp**

****

**homepage.jsp**

****

**Generated Output**

****

**Reading HTML form details with JSP**

**NOTE: I will be using this below form to read data with JSP and will add more to it while covering other aspects.**

**HTML FORM**



To read the ***First Name***and ***Last Name***, we will read the form using



However, the second syntax is mainly used for displaying data. Incase of reading the data we need to revert back to *request.getParameter(“formFieldName”)*

The action in the form tag specifies where we want to send the data to.

**Generated Output**

****

**DROPDOWN LIST**

**HTML**



**JSP**

****

**Generated Output**

****

**RADIOBUTTONS**

**HTML**



**JSP**



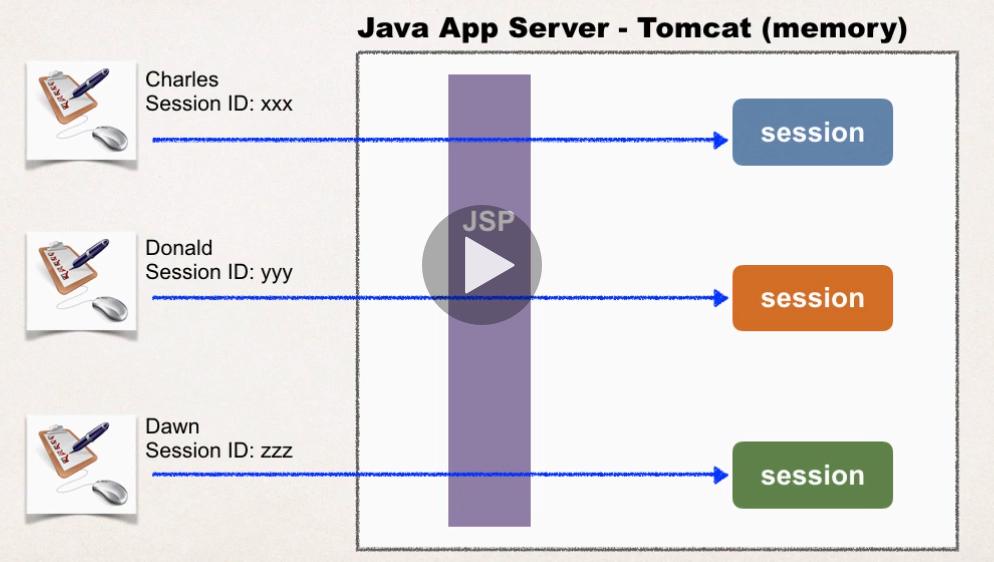
**Generated Output**



**State Management with JSP**

* **Tracking user actions with sessions**
  + JSP session is create once for user’s browser session. Unique for this user.
  + Commonly used when you need to keep track of user’s action.
    - **Examples –** Shopping Cart, Online Banking, Online Exam

If you can think of like an online banking application or a shopping cart application, or maybe like an online exam application, for every user that's accessing the web application, you need to keep track of those unique actions and those actions specific to that given user. That’s the idea of a session.



Note:

* A session object is kept in memory. So, it’s not stored on a database or a file system. Only in our server’s memory.
* Each user has their own session ID.

**Add data to session object**

**Syntax:**

Session.setAttribute(String name, Object value)

**Retrieve data from Session Object**

**Syntax:**

Object session.getAttribute(String name)

**Other useful methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| isNew() : boolean | Returns true if session |
| getId() : String | Returns the session ID |
| invalidate():void | Invalidates this session and unbinds any object associated with it |
| setMaxInactiveInterval(long mills): void | Sets the idle time for a session to expire. The value is supplied in milliseconds |

**TO DO LIST**

ArrayList

items

myToDoList

Goes into

Session Object Name

P.T.O



**Session Tracking - How Does the TO DO LIST demo work?**

<%@ page import="java.util.\*" %>

This code performs an import of the package java.util. We make use of the List and ArrayList from this package.

--

1. <html>
2. <body>

basic HTML set up code

---

**<!-- Step 1: Create HTML form -->**

1. <form action="toDoList.jsp">
2. Add new item: <input type="text" name="theItem" />
4. <input type="submit" value="Submit" />
5. </form>

This code creates an HTML form. The action will point back to the same JSP. So effectively, we are submitting form data back to ourselves. This form will read a text input from the user. The field is named "theItem". We'll read this field later to add it to your list. The form also includes a submit button.

---

This is a big section, so let's break down in detail even further.

At a high-level, we are going to add a new item to our to do list.

The high-level steps include

1. Get the TO DO items from the session

2. If the TO DO items doesn't exist in the session, then create a new list and add it to the session

3. Check to see if there is any form data to add.

Okay, let's break it down in detail with code examples.

----

**<!-- Step 2: Add new item to "To Do" list -->**

1. <%
2. // get the TO DO items from the session
3. List<String> items = (List<String>) session.getAttribute("myToDoList");

This section of code access the JSP session object. The session object is unique for each web user. We attempt to get the TO DO items from the session. We make use of the attribute name "myToDoList". This is basically the key/label to look up data from the session.

The session.getAttribute method will always return something of type java.lang.Object. We downcast this to List<String> because we are making use of strings to keep track of our to do items.

This is assigned to the variable "items". This variable holds an object reference to the data that we retrieved from the session object. We can use this variable later in the program to add items and also display items.

----

**// if the TO DO items doesn't exist, then create a new one**

1. if (items == null) {
2. items = new ArrayList<String>();
3. session.setAttribute("myToDoList", items);
4. }

This section of code checks to see if the TO DO items doesn't exist. If checks the variable "items" to see if it is null. If "items" is null then that means the TO DO items do not exist. Think of this as like a shopping cart .... your cart doesn't exist.

As a result, we need to create a new list and assign to items.

Then we place the items in the session. We make use of the name, value pair.  The attribute name is "myToDoList" and the object is the "items" variable.

---

**// see if there is form data to add**

1. String theItem = request.getParameter("theItem");
2. if (theItem != null) {
3. items.add(theItem);
4. }
5. %>

This section of code checks to see if there is form data to add. It reads the form data with the request.getParameter("theItem").  This is assigned to a variable. If the theItem variable is not null, then that means the user entered some data. Then we add theItem to our "items" array list.

Since we're using object references, remember that "items" is a variable that holds a reference to an object. Then it points to the same area of memory that is used by the session. So in effect, the users's session has now been updated with this new entry.

---

**<!-- Step 3: Display all "To Do" item from session -->**

1. <hr>
2. <b>To List Items:</b> <br/>
3. <ol>
4. <%
5. for (String temp : items) {
6. out.println("<li>" + temp + "</li>");
7. }
8. %>
9. </ol>

This section of code displays all of the "To Do" items from the session.

We make use of our variable "items" because it is our object reference to the data.

We use the "items" variable in the for loop to display the contents of each string in our array list.

To display data, we make use of "out.println".

---

1. </body>
2. </html>

This code just wraps up the HTML page.

* **For TO DO list example, I'm able to enter empty items. How to fix this?**

**Answer:** We basically need to check for null or empty input string. if null/empty then don't add to the TODO list.

Here's the snippet

---

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

if ( (theItem != null) && (!theItem.trim().equals("")) ) {  
   items.add(theItem);  
}

---

And here's the complete code:

<%@ page import="java.util.\*" %>

<html>

<body>

<!-- Step 1: Create HTML form -->  
<form action="toDoList.jsp">  
    Add new item: <input type="text" name="theItem" />  
      
    <input type="submit" value="Submit" />  
</form>

<!-- Step 2: Add new item to "To Do" list -->  
<%  
    // get the TO DO items from the session  
    List<String> items = (List<String>) session.getAttribute("myToDoList");

    // if the TO DO items doesn't exist, then create a new one  
    if (items == null) {  
        items = new ArrayList<String>();  
        session.setAttribute("myToDoList", items);  
    }  
      
    // see if there is form data to add  
**String theItem = request.getParameter("theItem");  
      
    if ( (theItem != null) && (!theItem.trim().equals("")) ) {  
       items.add(theItem);  
    }**  
%>

<!-- Step 3: Display all "To Do" item from session -->  
<hr>  
<b>To List Items:</b> <br/>

<ol>  
<%  
    for (String temp : items) {  
        out.println("<li>" + temp + "</li>");  
    }  
%>  
</ol>

</body>

</html>

**FAQ:**

If I refresh/reload the browser, it is automatically adding previously added item. Any solution for this problem?

**Answer:**

Yes, we can perform a "redirect" in your code to solve this problem.

In your code make the following update. Make note of the new code for: response.sendRedirect("toDoList.jsp");

1. response.sendRedirect("toDoList.jsp"); // UPDATE: NEW CODE

---

for completeness, here is the complete code for toDoList.jsp. Make note of the UPDATED CODE BLOCK FOR booleans and if/then statement.

Details on PageContext and Session objects

**PageContext**

**>> Please define clearly what pageContext is and what it does.**

A PageContext instance provides access to all the namespaces associated with a JSP page, provides access to several page attributes, as well as a layer above the implementation details. Implicit objects are added to the pageContext automatically.

The PageContext provides a number of facilities to the developer, including:

- a single API to manage the various scoped namespaces  
- a number of convenience API's to access various public objects  
- a mechanism to obtain the JspWriter for output  
- a mechanism to manage session usage by the page  
- a mechanism to expose page directive attributes to the scripting environment  
- mechanisms to forward or include the current request to other active components in the application  
- a mechanism to handle errorpage exception processing

**>> And how is pageContext.setAttribute("name", value) different from session.setAttribute("name", value).**

PageContext has a set of attributes that are different from the Session object.

The attributes set on PageContext are only available for a given page. The attributes are not available to other pages or servlets in the application.

Session attributes are created per each user's session. The session attributes are unique to a given session id. Session attributes are available to other pages and servlets in the application for a given session id.

**>> Your session ID is: ${pageContext.session.id}, why do this if we could get the session ID doing this: session.getId().**

As stated above, PageContext has a handle to the session object. You can access the session object via the pageContext or you can access the session object directly. Two different mechanisms for accessing the same object.

**>> Please also include how it's instantiated and where it comes from besides its purpose.**

The PageContext is implicitly instantiated by the application server.

The JSP page will always have access to the PageContext.

====

The following information is from student Jose M. (thanks!)

**Session Object**

*By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows:*

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

*The JSP engine exposes the HttpSession object to the JSP author through the implicit****session****object. Since****session****object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or getSession().*

*Just thought it may be useful to somebody that keeps wondering where the session object comes from.*

===

**Question**

When does the object we store in the session get updated? How does it work behind the scenes?

**Answer**

When you use the statement:

List<String> items = (List<String>) session.getAttribute("myToDoList");

The "items" variable is pointing to the SAME object in the session. You don't get a local copy. You are pointing to the same object.

Any changes to the "items" variable such as calling "items.add(...)" performs the update on the session object since we are pointing to the same object.

As a result, there is no need to explicitly update the session object.

You are not required to use the code below

session.setAttribute("myToDoList", items);

---

The key here is that we are pointing to the same object in the session.

**Cookies**

Cookies are small files which are stored on a user's computer. They are designed to hold a modest amount of data specific to a particular client and website, and can be accessed either by the web server or the client computer. This allows the server to deliver a page tailored to a particular user, or the page itself can contain some script which is aware of the data in the cookie and so is able to carry information from one visit to the website (or related site) to the next.

Basically, text data exchanged between browser and web server.

**Cookie contents**

* Name/Value Pair.
* Max is about 20 cookies per site per path

**How cookies are passed?**

Browser will only send cookies that match the server’s domain name.

**Cookie API – Package**

* Cookie class defined in package: javax.servlet.http
* Package imported for free in all JSP pages.

**Cookie API – Constructor**

Cookie (String name, String value)

Constructs a cookie with specified name and value.

Files used here: **cookies-homepage.jsp, cookies-personalized-form.html, cookies-personalize-response.jsp**

The homepage opens with the default value of the cookie set (in this following example, *Java* is the default value).

****



How to Handle White-Space in Cookie Values?

**Answer:**

To resolve this issue, we need to URL encode the cookie values.

Java provides two classes for URL encoding and decoding: java.net.URLEncoder, java.net.URLDecoder.

As an overview, these are the changes that need to be made.

In the file: **cookies-homepage.jsp**:  add code to URL decode the cookie value

1. // decode cookie data ... handle case of languages with spaces in them
2. favLang = URLDecoder.decode(tempCookie.getValue(), "UTF-8");

In the file: **cookies-personalize-response.jsp**: add code to URL encode the cookie value

1. // encode cookie data ... handle case of languages with spaces in them
2. favLang = URLEncoder.encode(favLang, "UTF-8");

Here is the complete code for this example. Make note of the import statements in the JSP pages.

====

**File: cookies-homepage.jsp**

1. <%@ page import="java.net.URLDecoder" %>
3. <html>
4. <body>
6. <h3>Training Portal</h3>
7. <!-- read the favorite programming language cookie -->
9. <%
10. // the default ... if there are no cookies
11. String favLang = "Java";
13. // get the cookies from the browser request
14. Cookie[] theCookies = request.getCookies();
16. // find our favorite language cookie
17. if (theCookies != null) {
19. for (Cookie tempCookie : theCookies) {
21. if ("myApp.favoriteLanguage".equals(tempCookie.getName())) {
23. // decode cookie data ... handle case of languages with spaces in them
24. favLang = URLDecoder.decode(tempCookie.getValue(), "UTF-8");
26. break;
27. }
28. }
29. }
30. %>
32. <!-- now show a personalized page ... use the "favLang" variable -->
33. <!-- show new books for this lang -->
34. <h4>New Books for <%= favLang %></h4>
35. <ul>
36. <li>blah blah blah</li>
37. <li>blah blah blah</li>
38. </ul>
40. <h4>Latest News Reports for <%= favLang %></h4>
41. <ul>
42. <li>blah blah blah</li>
43. <li>blah blah blah</li>
44. </ul>
46. <h4>Hot Jobs for <%= favLang %></h4>
47. <ul>
48. <li>blah blah blah</li>
49. <li>blah blah blah</li>
50. </ul>
52. <hr>
53. <a href="cookies-personalize-form.html">Personalize this page</a>
55. </body>
56. </html>

====

**File: cookies-personalize-response.jsp**

1. <%@ page import="java.net.URLEncoder" %>
3. <html>
4. <head><title>Confirmation</title></head>
6. <%
7. // read form data
8. String favLang = request.getParameter("favoriteLanguage");
10. // encode cookie data ... handle case of languages with spaces in them
11. favLang = URLEncoder.encode(favLang, "UTF-8");
13. // create the cookie
14. Cookie theCookie = new Cookie("myApp.favoriteLanguage", favLang);
16. // set the life span ... total number of seconds (yuk!)
17. theCookie.setMaxAge(60\*60\*24\*365); // <-- for one year
19. // send cookie to browser
20. response.addCookie(theCookie);
21. %>
23. <body>
24. Thanks! We set your favorite language to: ${param.favoriteLanguage}
26. <br/><br/>
28. <a href="cookies-homepage.jsp">Return to homepage.</a>
30. </body>
31. </html>

====

**File: cookies-personalize-form.html**  *(added languages with multiple words and spaces)*

1. <html>
3. <head>
4. <title>Personalize The Site</title>
5. </head>
7. <body>
9. <form action="cookies-personalize-response.jsp">
10. Select your Favorite Programming Language
12. <select name="favoriteLanguage">
13. <option>Active Server Pages</option>
14. <option>Java</option>
15. <option>C#</option>
16. <option> Common Business-Oriented Language</option>
17. <option>PHP</option>
18. <option>Ruby</option>
19. <option>Hypertext Markup Language</option>
20. </select>
22. <br/><br/>
24. <input type="submit" value="Submit" />
25. </form>
27. </body>
29. </html>

**JSP Scopes Example**

JSP provides very useful features. One of them is maintaining the user defined variable. As you all know that in JAVA, each variable has a scope. Scope decides the accessibility of an object or variable. For example, some variables are accessible within for loop, if-else block or within specific method or class or package.

Same way, in JSP some variables needs to have different scopes than others. JSP provides the capability to user to define the scope of these variables.

## Type of Scopes in JSP:

JSP provides 4 scopes to a variable. Developer can assign any one of them to a variable.

1. Page Scope.  
2. Request Scope.  
3. Session Scope.  
4. Application Scope.

## JSP *Page Scope* Example

Page Scope makes variable available to the developer for the current page only. Once the current page is closed by user or forwarded internally by application or redirected by application, than the variables having page scope will not be accessible on next page.



**Output :**   
Local Variable: Dinesh  
Test Page

**Second JSP File(test.jsp):**



**Output:** Variable from previous page:

As you can see above, we created 2 JSPs. On First JSP, we created local variable ‘name’ using JSTL Core tag c:set. This variable has a value ‘Dinesh’ and assigned ‘Page’ scope using scope attribute of c:set tag. We also provided a link on the same JSP that points to another JSP.

When we run the first JSP, it printed variable name successfully and also provided link on the browser.

On Second JSP, we again tried to print the same variable from the first page. So when user click on the link provided on first JSP, it goes to second JSP. However, when second JSP compiled and run, it did not print the variable value defined in first JSP.

This example result of 2 JSPs confirms that, if a variable has Page Scope, it is accessible in only that page and not on any other page.

## JSP *Request Scop*e Example

Request Scope makes variable available to the developer for the current request only. Once the current request is over, than the variables having request scope will not be accessible on next request. Single request may include multiple pages using forward.

**First JSP File:**



**Second JSP File(test.jsp):**



**Output:** Variable from previous page: Dinesh

As you can see above, we created 2 JSPs. On First JSP, we created local variable ‘name’ using JSTL Core tag c:set. This variable has a value ‘Dinesh’ and assigned ‘request’ scope using scope attribute of c:set tag. We also use jsp:forward tag to forward the same request to another JSP.

When we run the first JSP, it does not print anything on the browser and forwards the request to another JSP. However, it creates a variable before forwarding.

On Second JSP, it prints the variable value defined in first JSP.

This example result of 2 JSPs confirms that, if a variable has Request Scope, it is accessible in current request and on any page as long as request remains same.

## JSP *Session Scope* Example

Session Scope makes variable available to the developer for the current session only. Once the current session is over or timed out, than the variables having session scope will not be accessible on next session.

**First JSP File:**



**Output:**   
Local Variable: Dinesh  
Test Page  
**Second JSP File(test.jsp):**



**Output:** Variable from previous page: Dinesh

As you can see above, we created 2 JSPs. On First JSP, we created local variable ‘name’ using JSTL Core tag c:set. This variable has a value ‘Dinesh’ and assigned ‘Session’ scope using scope attribute of c:set tag. We also provided a link on the same JSP that points to another JSP.

When we run the first JSP, it printed variable name successfully and also provided link on the browser.

On Second JSP, we again tried to print the same variable from the first page. So when user click on the link provided on first JSP, it goes to second JSP. When second JSP compiled and run, it also printed the variable value defined in first JSP.

This example result of 2 JSPs confirms that, if a variable has Session Scope, it is accessible in only that session. During the current session, variable can be accessed from any JSPs.

## JSP *Application Scope* Example

Application Scope makes variable available to the developer for the full application. It remains available till application is running on server.

**First JSP File:**



**Output:**   
Local Variable: Dinesh  
Test Page

**Second JSP File(test.jsp):**



**Output:** Variable from previous page: Dinesh

As you can see above, we created 2 JSPs. On First JSP, we created local variable ‘name’ using JSTL Core tag c:set. This variable has a value ‘Dinesh’ and assigned ‘Application’ scope using scope attribute of c:set tag. We also provided a link on the same JSP that points to another JSP.

When we run the first JSP, it printed variable name successfully and also provided link on the browser.

On Second JSP, we again tried to print the same variable from the first page. So when user click on the link provided on first JSP, it goes to second JSP. When second JSP compiled and run, it also printed the variable value defined in first JSP.

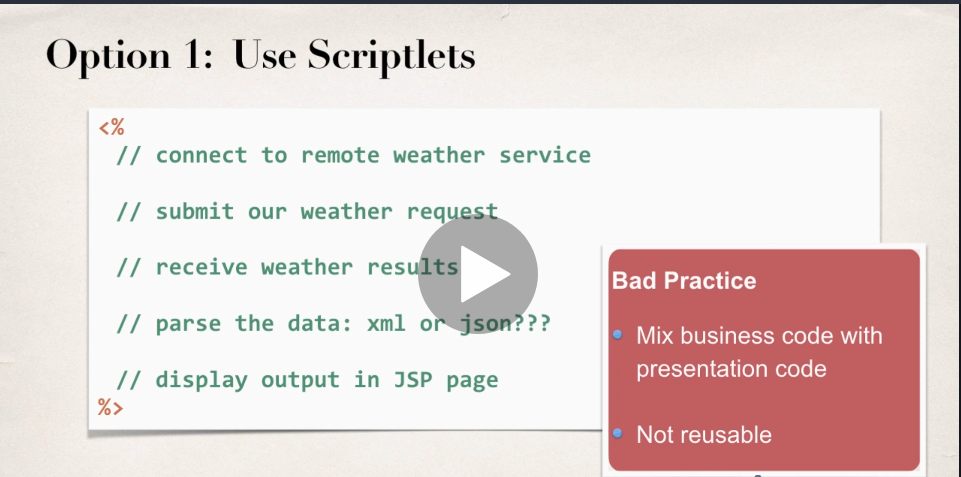
This example result of 2 JSPs confirms that, if a variable has Application Scope, it remains accessible in any JSP during the full application as long as it is running on server.

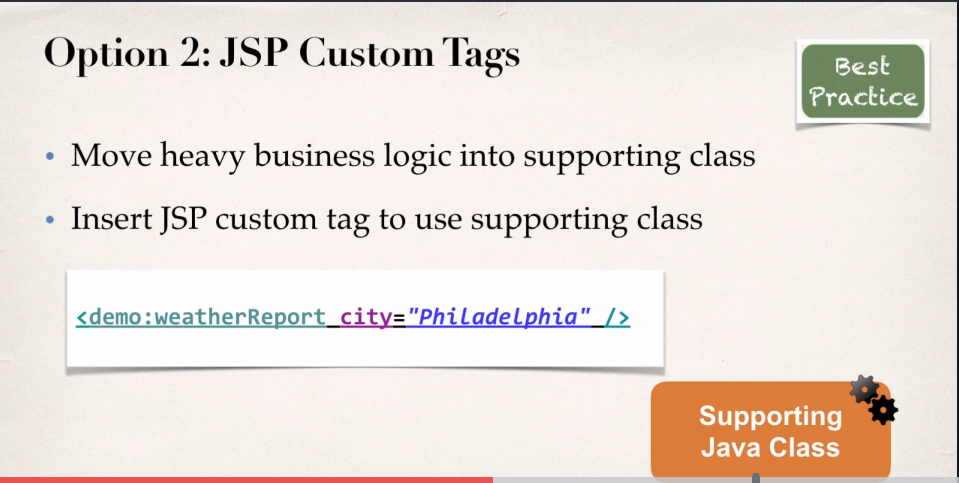
**JSP Standard Library Tag Library**

There’re actually two categories of JSP tags.

* **JSP Custom Tags** - That's where we can write our own, custom code and implement that code and use it as a tag.
* **JSP Standard Library Tags** – The folks at Oracle, they've created a common set of tags that you can make use of in your JSP environment.

**JSP Custom Tags** - We get the email from the boss, and he says, we want to include a weather report in our JSP pages. And we need it by the end of today .What we're going to do here is look at some solutions and how we can implement this for the boss, as far as making use of weather reports in our JSP pages. One option, is that we can make use of scriptlets. With a scriptlet, you simply have an angle bracket percent, and then you simply start writing Java code. So here we could write codes that connect to the remote service, submit our request, get the results, parse the data, maybe xml or json, and finally display the output in a JSP page.That's possible, and we can do it.However, I wouldn't recommend it.And it's actually bad practice.Because one, we'll have a lot of Java code in our JSP pages. It's all getting woven together, so it's gonna look really ugly at the end.And then finally, it's not reusable and not really maintainable either.Ideally, what you'd want to do is make use of JSP custom tags.Basically, you're gonna move your heavy, business logic into supporting classes, and then you simply insert a custom tag to make use of that supporting class. As an example, on our JSP page we can drop in this custom tag called weather report. Let's say city equals, insert your favorite city. And then the real heavy lifting is implement it by a supporting JAVA class. So note here how a JSP page is really simple. We simply have this one tag here, and all the real heavy coding is in a backend JAVA class.And using all of that information together,this will allow us to get a weather report and make use of it on our JSP page.So that's the idea.You want to make use of custom tags to implement some of your custom functionality. Now, what are the benefits of using custom tags? Well, first off, it minimizes the amount of scriptlet code in your JSP.Also avoids dumping thousands of lines of code in your JSP.And then the page is simple.The main focus of the page is only the presentation.And the tag itself is reusable.You can easily reuse that tag in other JSP pages in your project, or on other projects.



****

**Benefits of JSP Custom Tags**

* Minimize the amount of scriptlet code in a JSP
* Avoids dumping thousands of lines of code in a JSP
* JSP page is simple…main focus of JSP is only the presentation.
* **Tag is REUSABLE.**

**JSP Standard Library Tags** – Oracle created a common set of tags that you can use on any project.

* Core Tags - **for handling variables and looping and conditionals**
* Message Tags - **for handling internationalization and formatting**.
* Function Tags - **for doing string manipulation,getting the sizes of a collection**.
* XML tags - **for parsing and setting XML data**
* SQL tags - **set of tags called SQL,for accessing a database**.

**The SQL tags in general are considered bad practice. So, in the industry, they say that those SQL tags are good for prototyping, but not for real-world production applications.**

**LINKS**

* **Custom Tags -** [**https://docs.oracle.com/javaee/5/tutorial/doc/bnalj.html**](https://docs.oracle.com/javaee/5/tutorial/doc/bnalj.html)
* **JSTL -** [**https://docs.oracle.com/javaee/5/tutorial/doc/bnakc.html**](https://docs.oracle.com/javaee/5/tutorial/doc/bnakc.html)
* **JSP Standard Library Tags -** [**https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/**](https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/)
* **JSTL Specification PDF -** [**https://download.oracle.com/otndocs/jcp/jstl-1.2-mrel2-oth-JSpec/**](https://download.oracle.com/otndocs/jcp/jstl-1.2-mrel2-oth-JSpec/)

**Steps for installation of JSTL Jar files.**

Download the two jar files and move them into **Project/WebContent/Web-INF/lib**

* **Javax.servlet.jsp.jstl-1.2.1.jar – IMPLEMENTATION -** This contains an implementation of the JSTL API. This code implements all of the interface from the API above.
* **Javax.servlet.jsp.jstl-api-1.2.1.jar – API -** This contains the JSTL API interfaces and support classes. However, a large number of the interfaces do not have implementation classes.

**Files used here:** Test.jsp



**Note:** Set up a reference to that tag library. And, make use of it by using angle bracket percent with an @ symbol and then the prefix equals c. So, the prefix c is simply short for core. So, we're going to make use of the core tag library here.

**JSTL Core Tags**

|  |  |
| --- | --- |
| Tag | Description |
| catch | Catches any throwable to occurs in the boy |
| chose | Conditional tag that can be used for exclusive operations, similar to switch statement |
| if | Simple if/then conditional |
| import | Retrieves a URL and exposes its contents on the page or a variable. |
| forEach | Iterates over a collection of values |
| forTokens | Iterates over a collection of tokens |
| out | Used in scriptlets to display output, similar to <%=…….%> |
| otherwise | Used with the <choose> tag to handle the else clause |
| param | Adds a parameter to the URL |
| redirect | Redirects the browser to a new URL |
| remove | Removes a scoped variable |
| set | Assigns an expression value to a variable |
| url | Defines a URL with query parameters |
| when | Used with the <choose> tag, when a condition Is true. |

**JSP Core Taglib Reference.**

* Every page that uses the Core tags must include this reference:



One thing about the uri is that it's a bit misleading. There’s no real connection to the internet to make use of this tag library. It’s simply a unique identifier. They’ll actually make use of this unique identifier when they scan our local jar file that's part of our project. So again, no connection to the internet. Simply a unique identifier that associates your tag in the jar filed locally on your file system.

**Looping with forEach**



**Reading Cookies with JSTL**

**Answer:**Yes. The standard syntax to access HTTP Cookie value in JSP is:

1. ${cookie.<cookie name>.value}

So if you want to print value of cookie named “foobar” on JSP page, you would use:

1. ${cookie.foobar.value}

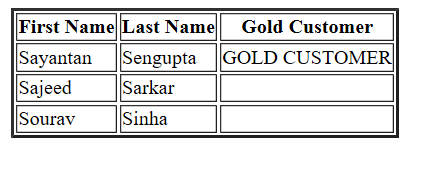
You can also loop through all of the cookies using this

<c:forEach items="${cookie}" var="currentCookie">    
    Cookie name as map entry key: ${currentCookie.key} <br/>  
    Cookie object as map entry value: ${currentCookie.value} <br/>  
    Name property of Cookie object: ${currentCookie.value.name} <br/>  
    Value property of Cookie object: ${currentCookie.value.value} <br/>  
</c:forEach>

**Testing conditionals with IF tag**

****

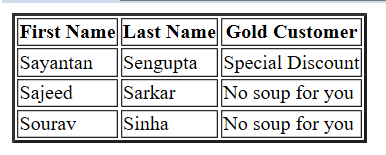
**Output:**

****

**Choose Tag**

****

**Output:**

****

**JSTL Function tags**

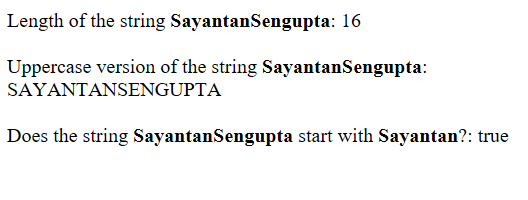
* Collection length
  + Length
* String manipulation
  + toUpperCase, toLowerCase
  + substring, substringAfter, substringBefore
  + trim, replace, indexOf, startsWith, endsWith
  + contains, containsIgnoreCase, split, join, escapeXML
* Every page using Function tags must include this tablib



**Files Used: FunctionsTag.jsp (only few are used, there are many more)**

****

**Output:**



**Split Function**

****

**Output:**



**JSTL Message Tags (i18n Tags)**

**JSTL Internationalization** - The process of designing an application where you can adapt it to different languages without having to make changes to your source code.

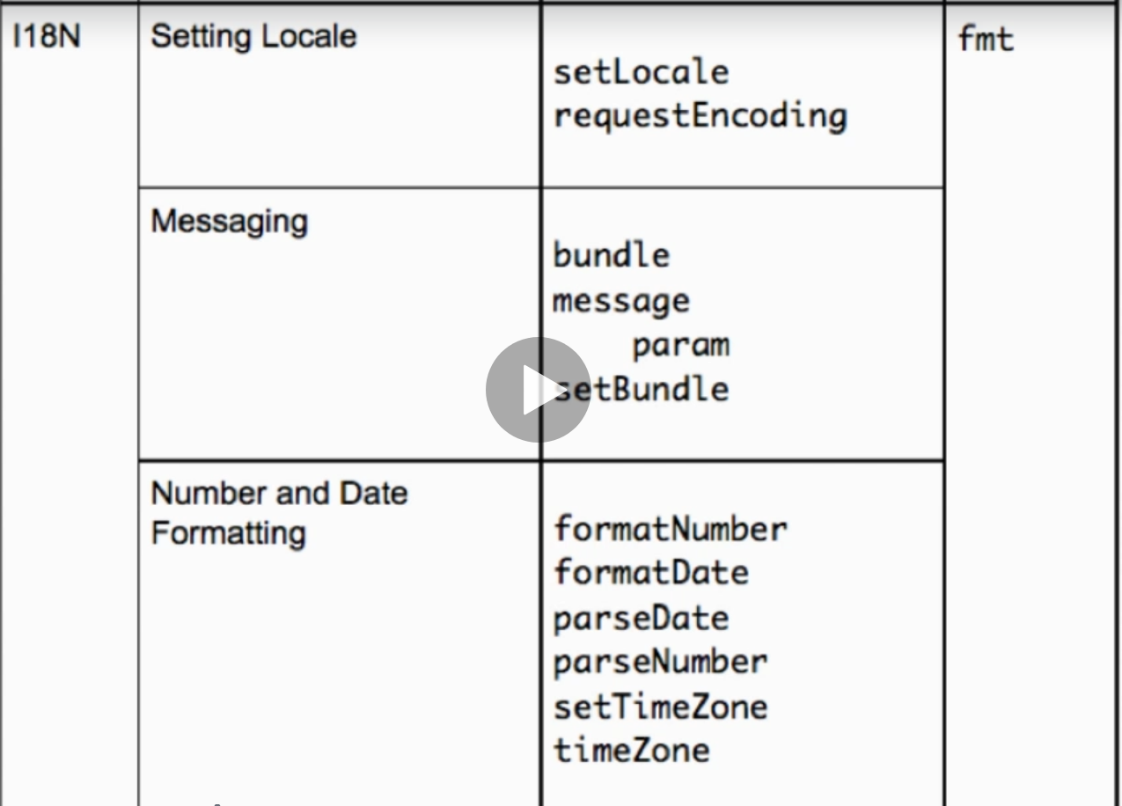
Note - the term internationalization is frequently abbreviated as I18N, and you may wonder, well why did they come up with that? Well, there's 18 letters between the first I and the last n in the word.

**How do you do internationalization?**

Well basically, instead of hard-coding text and messages in your application, instead what you'll do is you'll make use of labels and placeholders. So, for greeting the user, you'll simply have a label for greeting. To reference their first name and last name, you'll use labels for that. To give them a welcome message, you'll have a label for that. So, it's simply just a placeholder, and then you'll supply the appropriate values and data to insert there, based on the language that the user has selected. Now what you'll have to do is for each language you're site's going to support, you need to create translated versions of each label. So there's no magic here. Java won’t automatically create everything for you. You'll have to create this special file that will have the appropriate translations for a greeting, or first name/last name, or the welcome message. You'll have to create that file.

Instead of hard-coding the messages in your application, we make use of these labels, basically placeholders for a locale, and a locale is basically a combination of a language and a region. So, there's a number of different locales. So, here we have the first locale, en\_US, that's English that's based in the US, en\_GB, that's English based in the UK, and there are differences between the two, even though it's the same language, depending on which country or region that you're working with, some things are different. So, as far as some of the differences here, is formatting dates, numbers, and currency. So, in the US, when we display a date, short dates, we simply use month, day, year,however in the United Kingdom or Great Britain,when they make use of short date, it's day, month, year,so it's really confusing.So those are just some of the differences, formatting dates.Also, formatting numbers,depending on what delimiters you'll use there, and also formatting currency, as far as using the actual currency symbol for that given area. One caveat here on the currency. Formatting the currency is only for giving the currency symbol, like a dollar sign, or pound symbol, or euro symbol. However, it will not do any currency conversion for you, so it won't automatically connect, and get the currency exchange, and convert it for you, it’s simply displaying formatting data. You’ll still have to convert the currency in the background with your own Java code.

**AREA FUNCTION TAG PREFIX**



**Steps for Internationalization**

1. Create Resource Files
   1. Translated version of our labels
   2. Must Name the file using the locale.

**Locale = Language Code + Country Code**

File naming syntax: <project file\_name>\_LANGUAGECODE\_COUNTRYCODE.properties

Examples: mylabels\_es\_ES.properties; mylabels\_eb\_GB.properties; mylabels\_de\_DE.properties;

Language Codes - <https://en.wikipedia.org/wiki/ISO_639-1>

Country Codes – <https://en.wikipedia.org/wiki/ISO_3166-2>

1. Create JSP Page with labels
2. Update JSP Page to change locale based on user selection.

**MULTILINGUAL APP**

****

**Files used: Java Resources/src/com.sayantan.i18N.resources/**

**mylabel.properties; mylabel\_es\_ES.properties; mylabel\_de\_DE.properties;**

**i18n-messages-test.jsp**

**Default resource file – No reference to a specific locale**

**mylabels.properties**

label.greeting=Howdy

label.firstName=FIRST NAME

label.lastName =LAST NAME

label.welcome =WELCOME TO SAYANTAN TUTORIALS

**mylabel\_es\_ES.properties**

label.greeting=Hola

label.firstName= Nombre de pila

label.lastName = Apellido

label.welcome = BIENVENIDA A TUTORIALES SAYANTAN

**mylabel\_de\_DE.properties**

label.greeting=Hallo

label.firstName= Vorname

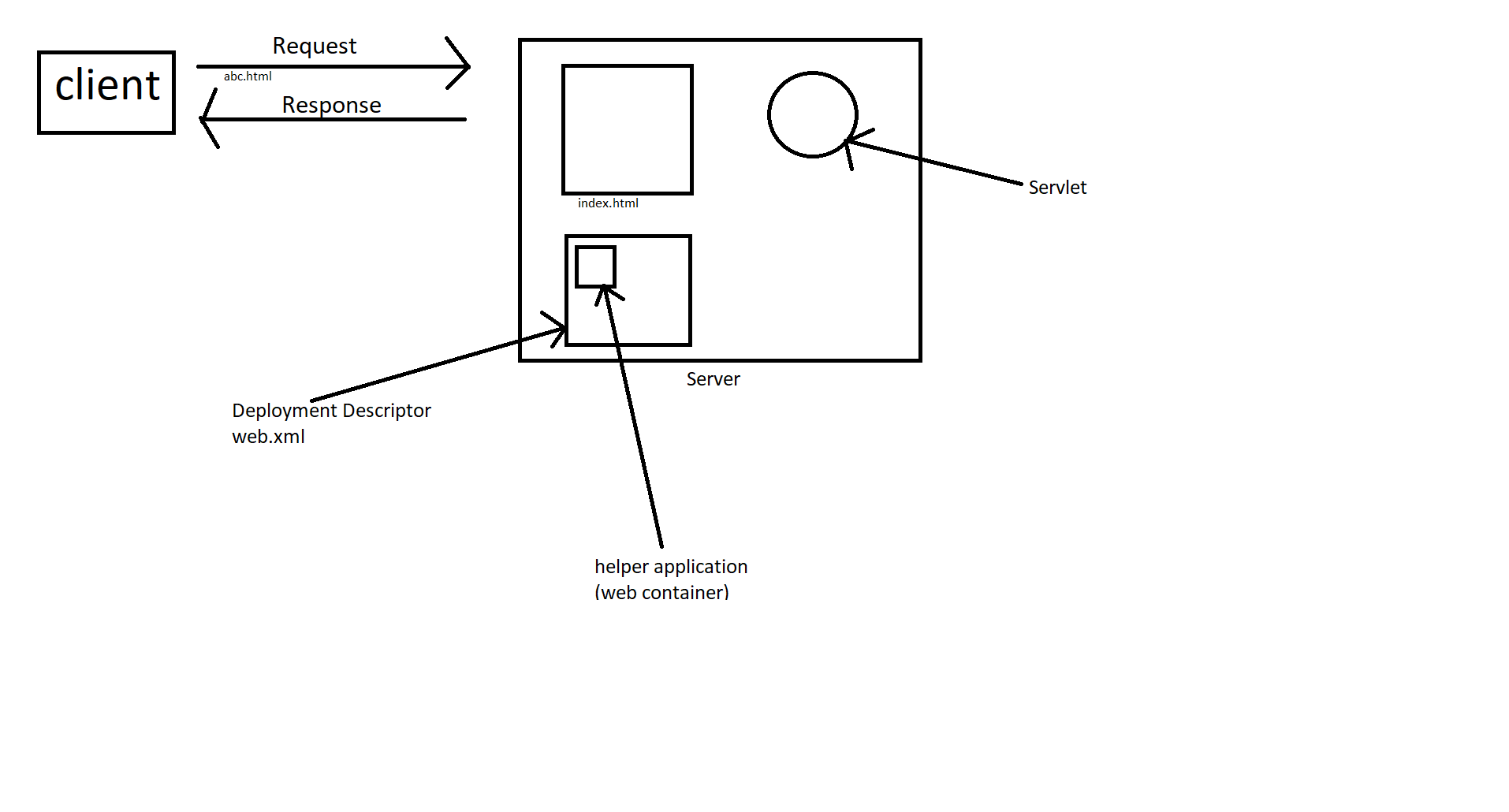
label.lastName = Nachname

label.welcome = WILLKOMMEN BEI SAYANTAN TUTORIALS

**i18n-messages-test.jsp**



**Introduction to Servlets**

* Java class that is processed on the servers,
* Java class generates HTML that is returned to the browser
* Can read HTML form data, use cookies and sessions etc.
* At a higher level, similar functionality to JSPs

**EXPLANATION** - <https://www.youtube.com/watch?v=7TOmdDJc14s&list=PLsyeobzWxl7pUPF2xjjJiG4BKC9x_GY46&index=1>

**THEORY** - <https://www.tutorialspoint.com/servlets/index.htm>

**Comparison between JSP and Servlets**

|  |  |
| --- | --- |
| **JSPs** | **Servlets** |
| HTML file with .jsp extension | Java class file |
| Contains static HTML | Generate all HTML |
| JSP to generate HTML | More steps to access web objects |
| Has built-in JSP objects | - |

**Which one to use?**

The best practice is to

* Integrate them both together
* Servlet does the business logic
* JSP handles the presentation view

**This design pattern is known as MODEL-VIEW-CONTROLLER (MVC) DESIGN PATTERN.**

**Used by frameworks like Spring MVC, JSF, Struts and so on.**

**Reading HTML form data with Servlets**

**Step 1:**  Build HTML Form



**Step 2:** Reading form Data with Servlet



**Difference between GET and POST**

**Sending data with GET method**

<form action=”StudentServlet” method=”GET”>

……..

</form>

* Form data is added to the end of the URL as name/value pairs
  + theURL?field1=value1&field2=value2

**Sending data with GET method**

<form action=”StudentServlet” method=”POST”>

……..

</form>

* **Form data is passed in the body of HTTP request message**

**So, which one should we use?**

|  |  |
| --- | --- |
| **GET** | **POST** |
| Good for debugging | Can’t bookmark or email URL |
| Bookmark or email URL | No limitations on data length |
| Limitations on data length | Can also send binary data |
| Send Non-sensitive Data | Send Sensitive Data |

**Reading Servlet Parameters**

**Servlet configuration Parameters**

* Web app can make use of configuration parameters
* Located in standard file: WEB-INF/web.xml

ServletContext context = getServletContext(): 🡨 Inherited from HttpServlet

The servletContext is actually a helper class to read/parse web.xml.

getInitParameter(..) will always return a string and will return null if no parameter found.

**Can you define per servlet parameters?**

Yes, you can define servlet parameters on a per servlet basis.

These params can only be read by the given servlet.

Here's an example for your web.xml.



IMPORTANT: Your servlet can not use Annotations for @WebServlet. It has to be manually configured in web.xml with <servlet> and <servlet-mapping> tags, as shown above

---

Then in your servlet, you can read this servlet's params using this code

String theGreeting = getInitParameter("greeting");

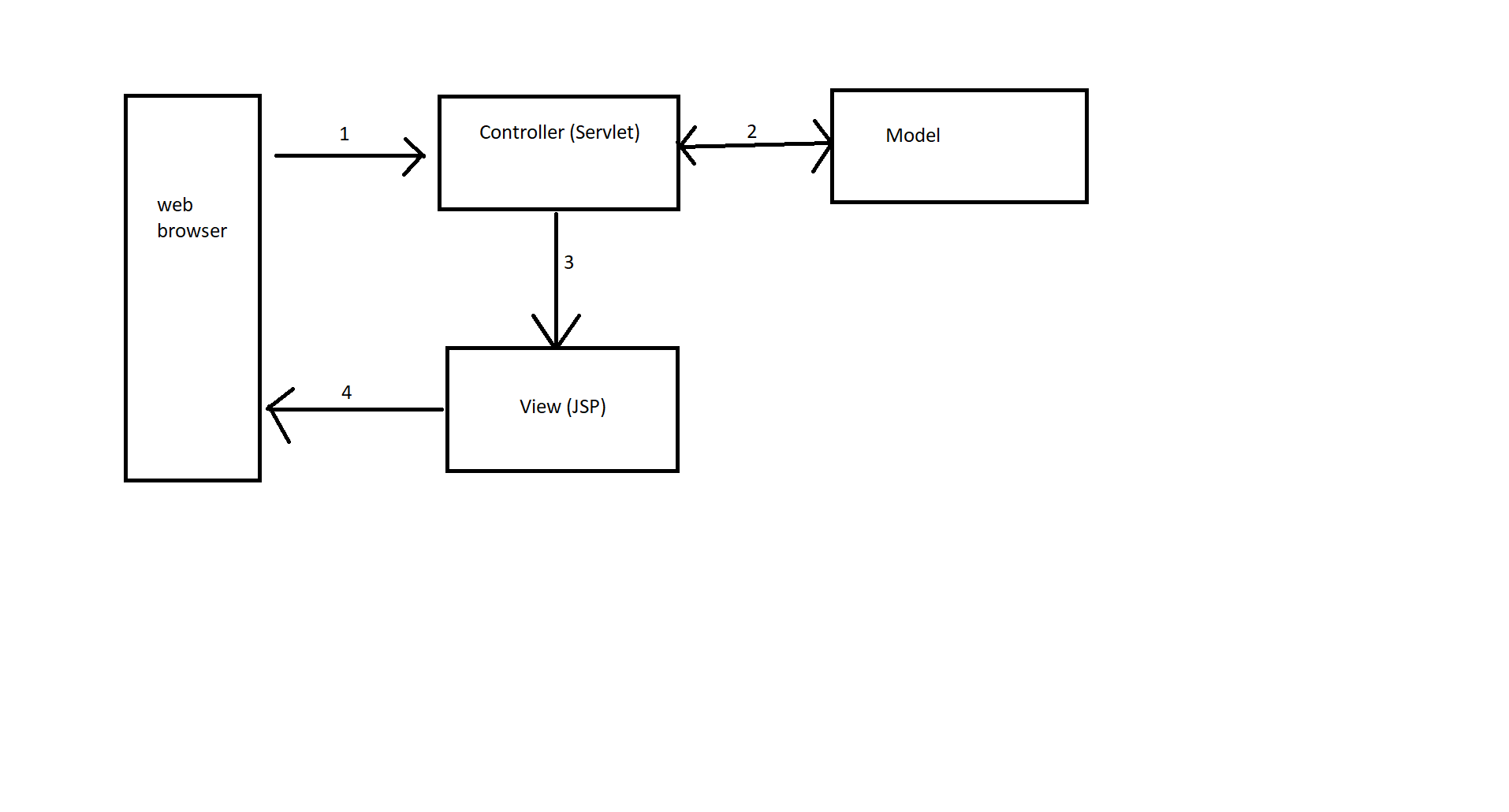
String theInfo = getInitParameter("serviceLevel");

Note: These params can only be read by the given servlet. If another servlet attempts to read the params then they will get null.

Here is the full source code for the servlet.



**MVC with Servlets and JSP**



**Explanation**

The web browser will send them the request, the request initially comes into the controller servlet. The purpose of the controller servlet is to hold the business logic. Now, it may delegate a call out to the model to get additional data, or additional information. Then, once it has the data, then it can send information over to the view page and the view page is actually the JSP. Once the JSP has the information, then the JSP can render an HTML view and the JSP will send that view back to the web browser. So that's kind of the big picture here with the Model-View-Controller.

**Benefits**

* Minimize HTML code in servlet
  + No more: out.println(..) in Servlet code
* Minimize Java business logic in JSP
  + No larger Scriptlets in JSP code

**Servlet calling JSP**

* Servlet can call JSP using a *request dispatcher.*

**Sending Data to JSP**

* Servlet can add data to request object

**JSP Page to view data**

* JSP use JSTL to access data

**Source Code**

**Controller (Servlet):**



**Continued**

****

**View (JSP)**

****

**Now the step 2 i.e. the Model will be created. Basically, a helper class will be created which will provide the list of students. And recreate the whole MVC project.**

1. **Controller (Servlet)**



1. **Model**



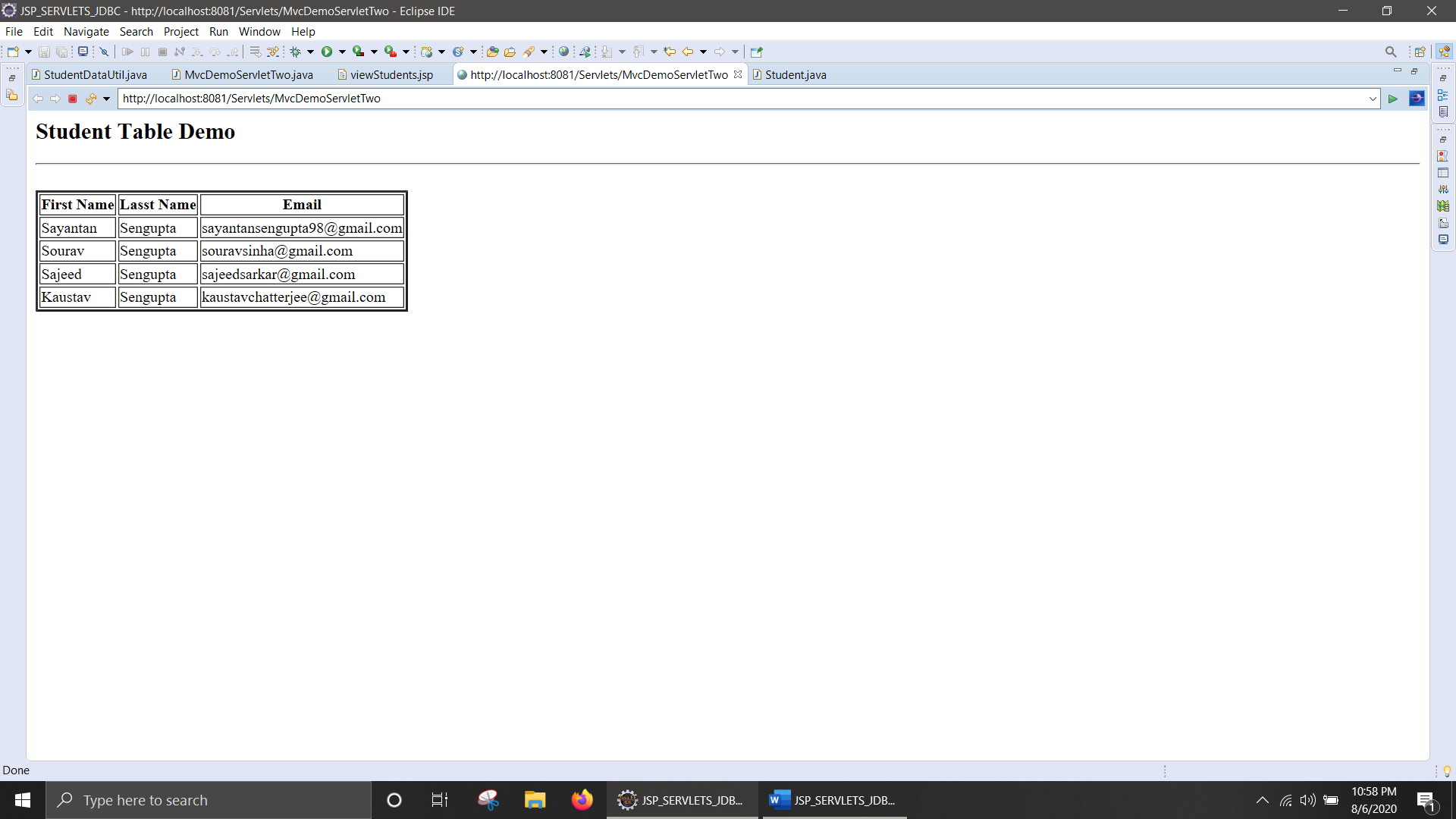


1. **View(JSP)**

**P.T.O**



**Output**



**Database Connection Pool**

* Best practice is to use database connection pools
  + Allows app to scale and handle multiple users quickly.

**Database Connection in Tomcat**

* **JDBC Driver JAR file**
  + **Place it in WEB-INF/lib**
* **Define connection pool in META-INF/context.xml**
* **Get connection pool reference in java code**

**Get Connection pool in Java code**

* Leverage *resource injection* with servlets
* This means that TOMCAT will automatically
  + Set the connection pool/ datasource on your servlet