# Java Servlet Tutorial

JournalDev Pankaj Kumar

# Table of Content

# 1. Understanding Java Web Applications

**Web Applications** are used to create dynamic websites. Java provides support for web application through **Servlets** and **JSPs**. We can create a website with static HTML pages but when we want information to be dynamic, we need web application.

The aim of this article is to provide basic details of different components in Web Application and how can we use Servlet and JSP to create our first java web application.

1. Web Server and Client
2. HTML and HTTP
3. Understanding URL
4. Why we need Servlet and JSPs?
5. First Web Application with Servlet and JSP
6. Web Container
7. Web Application Directory Structure
8. Deployment Descriptor

## A. Web Server and Client

Web Server is a software that can process the client request and send the response back to the client. For example, Apache is one of the most widely used web server. Web Server runs on some physical machine and listens to client request on specific port.

A web client is a software that helps in communicating with the server. Some of the most widely used web clients are Firefox, Google Chrome, and Safari etc. When we request something from server (through URL), web client takes care of creating a request and sending it to server and then parsing the server response and present it to the user.

# B. HTML and HTTP

Web Server and Web Client are two separate software's, so there should be some common language for communication. HTML is the common language between server and client and stands for **H**yper**T**ext **M**arkup **L**anguage.

Web server and client needs a common communication protocol, HTTP (**H**yper**T**ext **T**ransfer **P**rotocol) is the communication protocol between server and client. HTTP runs on top of TCP/IP communication protocol.

Some of the important parts of HTTP Request are:

- **HTTP Method** – action to be performed, usually GET, POST, PUT etc.
- **URL** – Page to access
- **Form Parameters** – similar to arguments in a java method, for example user,password details from login page.

Sample HTTP Request:

1GET /FirstServletProject/jsps/hello.jsp HTTP/1.1
2Host: localhost:8080
3Cache-Control: no-cache

Some of the important parts of HTTP Response are:

- **Status Code** – an integer to indicate whether the request was success or not. Some of the well-known status codes are 200 for success, 404 for Not Found and 403 for Access Forbidden.
- **Content Type** – text, html, image, pdf etc. Also known as MIME type
- **Content** – actual data that is rendered by client and shown to user.

Sample HTTP Response:

```
200 OK
Date: Wed, 07 Aug 2013 19:55:50 GMT
Server: Apache-Coyote/1.1
Content-Length: 309
Content-Type: text/html;charset=US-ASCII
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Hello</title>
</head>
<body>
<h2>Hi There!</h2>
<br>
<h3>Date=Wed Aug 07 12:57:55 PDT 2013
</h3>
</body>
</html>
```

**MIME Type or Content Type**: If you see above sample HTTP response header, it contains tag "Content-Type". It's also called MIME type and server sends it to client to let them know the kind of data it's sending. It helps client in rendering the data for user. Some of the mostly used mime types are text/html, text/xml, application/xml etc.

# C. Understanding URL

URL is acronym of Universal Resource Locator and it's used to locate the server and resource. Every resource on the web has its own unique address. Let's see parts of URL with an example.

**http://localhost:8080/FirstServletProject/jsps/hello.jsp**

**http://** – This is the first part of URL and provides the communication protocol to be used in server-client communication.

**localhost** – The unique address of the server, most of the times it's the hostname of the server that maps to unique IP address. Sometimes multiple hostnames point to same IP addresses and web server virtual host takes care of sending request to the particular server instance.

**8080** – This is the port on which server is listening, it's optional and if we don't provide it in URL then request goes to the default port of the protocol. Port numbers 0 to 1023 are reserved ports for well-known services, for example 80 for HTTP, 443 for HTTPS, 21 for FTP etc.

**FirstServletProject/jsps/hello.jsp** – Resource requested from server. It can be static html, pdf, JSP, servlets, PHP etc.

# D. Why we need Servlet and JSPs?

Web servers are good for static contents HTML pages but they don't know how to generate dynamic content or how to save data into databases, so we need another tool that we can use to generate dynamic content. There are several programming languages for dynamic content like PHP, Python and Ruby on Rails, Java Servlets and JSPs.

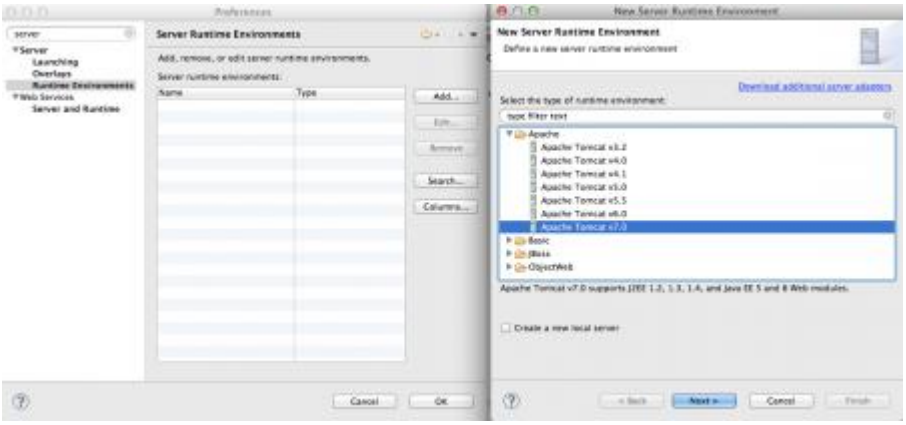
Java Servlet and JSPs are server side technologies to extend the capability of web servers by providing support for dynamic response and data persistence.

# E. First Web Application with Servlet and JSP

We will use "Eclipse IDE for Java EE Developers" for creating our first servlet application. Since servlet is a server side technology, we will need a web container that supports Servlet technology, so we will use Apache Tomcat server. It's very easy to setup and I am leaving that part to yourself.

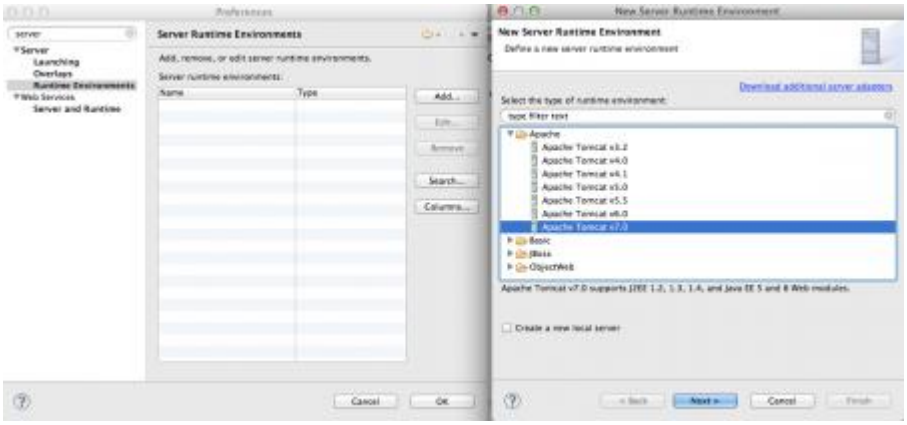
For ease of development, we can add configure Tomcat with Eclipse, it helps in easy deployment and running applications.

Go to Eclipse Preference and select Server Runtime Environments and select the version of your tomcat server, mine is Tomcat 7.



Provide the apache tomcat directory location and JRE information to add the runtime environment.

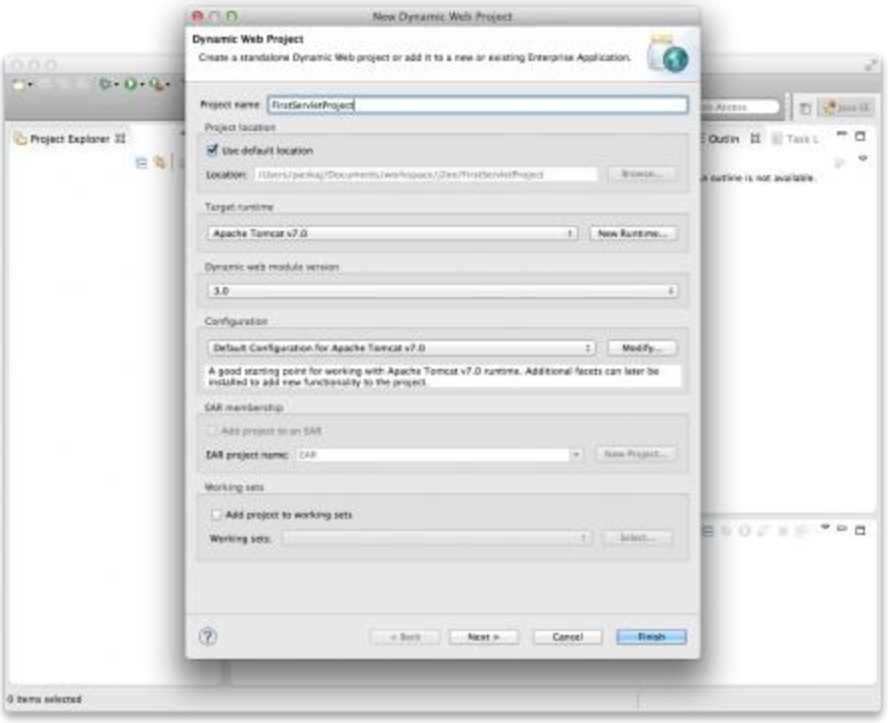
Now go to the Servers view and create a new server like below image pointing to the above added runtime environment.




**Note**: If Servers tab is not visible, then you can select Window > Show View > Servers so that it will be visible in Eclipse window. Try stopping and starting server to make sure it's working fine. If you have already started the server from terminal, then you will have to stop if from terminal and then start it from Eclipse else it won't work perfectly.

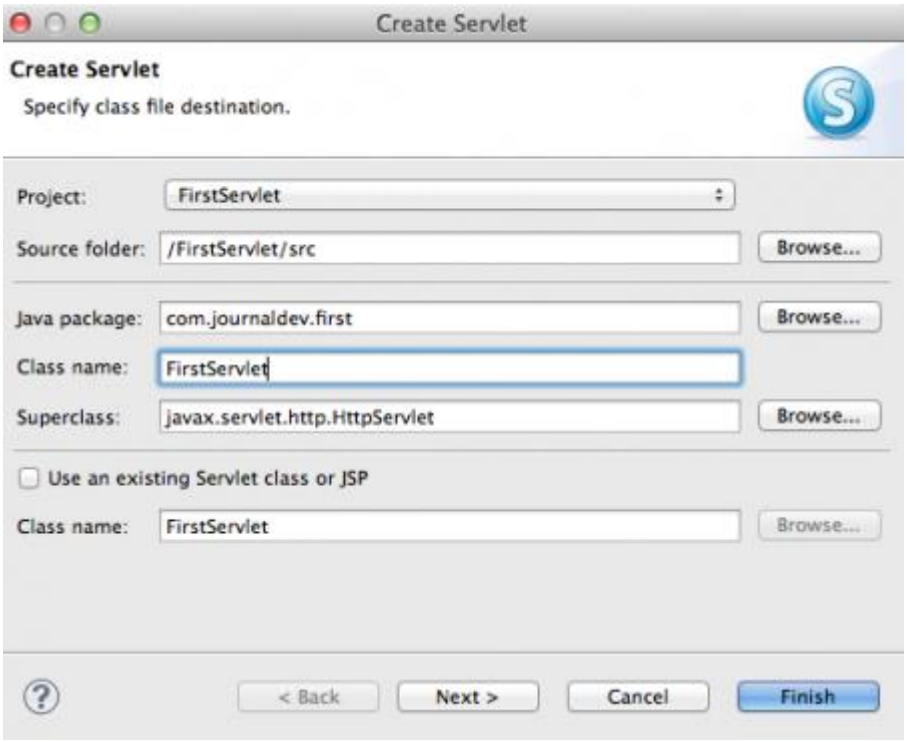
Now we are ready with our setup to create first servlet and run it on tomcat server.

Select File > New > Dynamic Web Project and use below image to provide runtime as the server we added in last step and module version as 3.0 to create our servlet using Servlet 3.0 specs.




You can directly click Finish button to create the project or you can click on Next buttons to check for other options.

Now select File > New > Servlet and use below image to create our first servlet. Again we can click finish or we can check other options through next button.




When we click on Finish button, it generates our Servlet skeleton code, so we don't need to type in all the different methods and imports in servlet and saves us time.

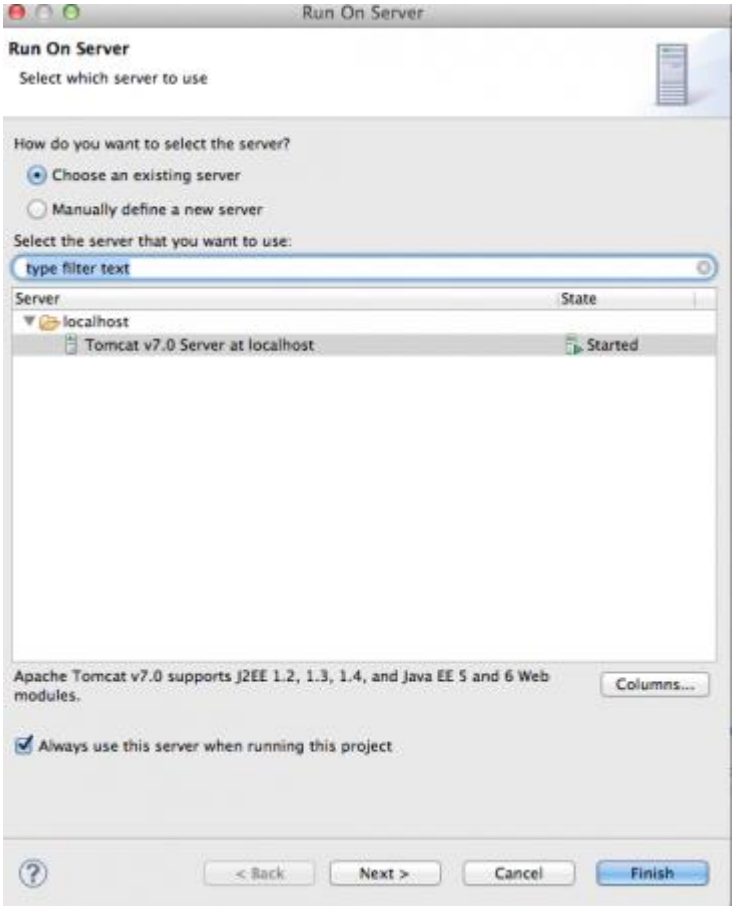
Now we will add some HTML with dynamic data code in **doGet()** method that will be invoked for HTTP GET request. Our first servlet looks like below.

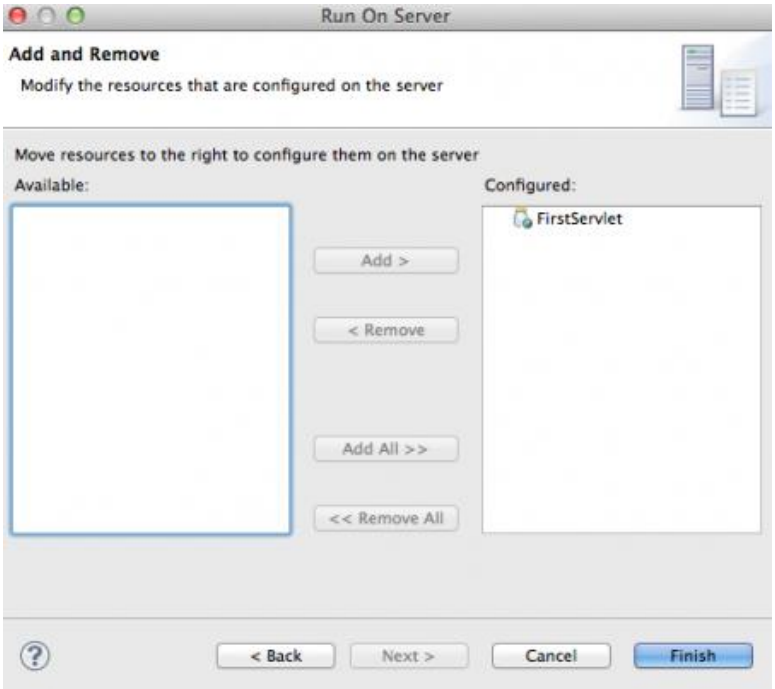
```
package com.journaldev.first;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class FirstServlet
 */
@WebServlet(description = "My First Servlet", urlPatterns = {
"/FirstServlet" , "/FirstServlet.do"}, initParams =
{@WebInitParam(name="id",value="1"),@WebInitParam(name="name",value="pa
nkaj")})
```

```
public class FirstServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public static final String HTML_START="<html><body>";
    public static final String HTML_END="</body></html>";

    /**
     * @see HttpServlet#HttpServlet()
     */
    public FirstServlet() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request,
HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        Date date = new Date();
        out.println(HTML_START + "<h2>Hi
There!</h2><br/><h3>Date="+date +"</h3>"+HTML_END);
    }

    /**
     * @see HttpServlet#doPost(HttpServletRequest request,
HttpServletResponse response)
     */
    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
    }

}
```

Before Servlet 3, we need to provide the url pattern information in web application deployment descriptor but servlet 3.0 uses **java annotations** that is easy to understand and chances of errors are less.

Now chose Run > Run on Server option from servlet editor window and use below images for the options.


Run On Server
Run On Server
Select which server to use
How do you want to select the server?
Choose an existing server
Manually define a new server
Select the server that you want to use:
type filter text
Server
State
localhost
Tomcat v7.0 Server at localhost
Started
Apache Tomcat v7.0 supports J2EE 1.2, 1.3, 1.4, and Java EE 5 and 6 Web modules.
Columns...
Always use this server when running this project
< Back
Next >
Cancel
Finish


Run On Server
Add and Remove
Modify the resources that are configured on the server
Move resources to the right to configure them on the server
Available:
Configured:
FirstServlet
Add >
< Remove
Add All >>
<< Remove All
< Back
Next >
Cancel
Finish

After clicking finish, browser will open in Eclipse and we get following HTML page.


You can refresh it to check that Date is dynamic and keeps on changing, you can open it outside of Eclipse also in any other browser.

So servlet is used to generate HTML and send it in response, if you will look into the doGet() implementation, we are actually creating an HTML document as writing it in response PrintWriter object and we are adding dynamic information where we need it.

It's good for start but if the response is huge with lot of dynamic data, it's error prone and hard to read and maintain. This is the primary reason for introduction of JSPs.

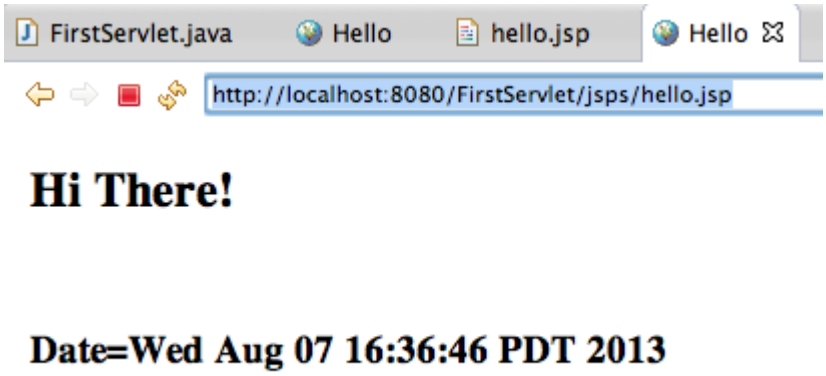
JSP is also server side technology and it's like HTML with additional features to add dynamic content where we need it.

JSPs are good for presentation because it's easy to write because it's like HTML. Here is our first JSP program that does the same thing like above servlet.

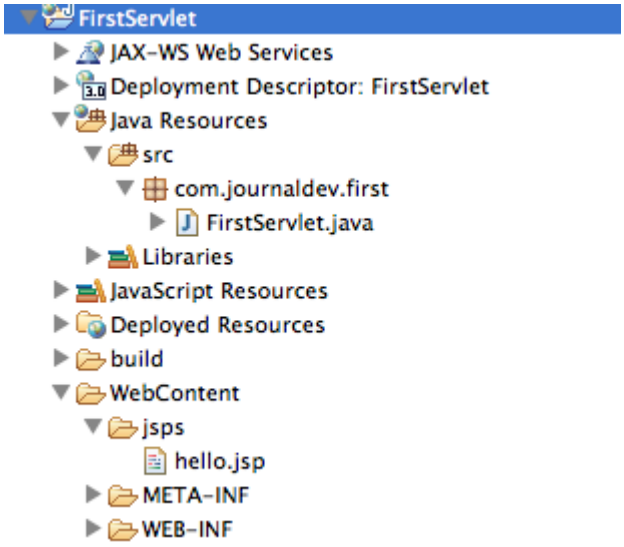
```
<%@page import="java.util.Date"%>
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Hello</title>
</head>
<body>
```

```
<h2>Hi There!</h2>
<br>
<h3>Date=<%= new Date() %>
</h3>
</body>
</html>
```

If we run above JSP, we get output like below image.



The final project hierarchy looks like below image in Eclipse.




**You can download the source code of "First Servlet Project" from here.**

We will look into Servlets and JSPs in more detail in future posts but before concluding this post, we should have good understanding of some of the aspects of java web applications.

# F. Web Container

Tomcat is a web container, when a request is made from Client to web server, it passes the request to web container and its web container job to find the correct resource to handle the request (servlet or JSP) and then use the response from the resource to generate the response and provide it to web server. Then web server sends the response back to the client.

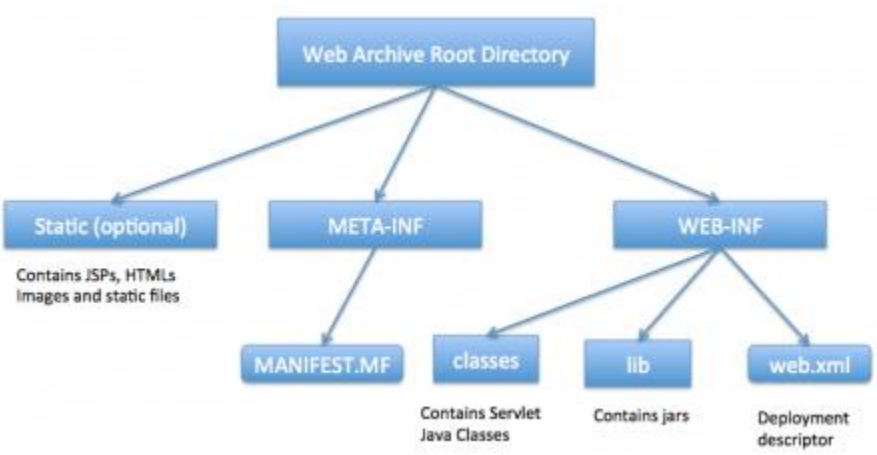
When web container gets the request and if it's for servlet then container creates two Objects HTTPServletRequest and HTTPServletResponse. Then it finds the correct servlet based on the URL and creates a thread for the request. Then it invokes the servlet service() method and based on the HTTP method service() method invokes doGet() or doPost() methods. Servlet methods generate the dynamic page and write it to response. Once servlet thread is complete, container converts the response to HTTP response and send it back to client.

Some of the important work done by web container are:

- **Communication Support** – Container provides easy way of communication between web server and the servlets and JSPs. Because of container, we don't need to build a server socket to listen for any request from web server, parse the request and generate response. All these important and complex tasks are done by container and all we need to focus is on our business logic for our applications.
- **Lifecycle and Resource Management** – Container takes care of managing the life cycle of servlet. Container takes care of loading the servlets into memory, initializing servlets, invoking servlet methods and destroying them. Container also provides utility like JNDI for resource pooling and management.
- **Multithreading Support** – Container creates new thread for every request to the servlet and when it's processed the thread dies. So servlets are not initialized for each request and saves time and memory.
- **JSP Support** – JSPs doesn't look like normal java classes and web container provides support for JSP. Every JSP in the application is compiled by container and converted to Servlet and then container manages them like other servlets.
- **Miscellaneous Task** – Web container manages the resource pool, does memory optimizations, run garbage collector and provides security configurations, support for multiple applications, hot deployment and several other tasks behind the scene that makes our life easier.

# G. Web Application Directory Structure

Java Web Applications are packaged as Web Archive (WAR) and it has a defined structure. You can export above dynamic web project as WAR file and unzip it to check the hierarchy. It will be something like below image.




# H. Deployment Descriptor

**web.xml** file is the deployment descriptor of the web application and contains mapping for servlets (prior to 3.0), welcome pages, security configurations, session timeout settings etc.

# 2. Java Servlet Tutorial

In the earlier section, we learned about **Java Web Application** and looked into core concepts of Web Applications such as **Web Server**, **Web Client**, **HTTP** and **HTML**, **Web Container** and how we can use **Servlets** and **JSPs** to create web application. We also created our first Servlet and JSP web application and executed it on tomcat server.

This tutorial is aimed to provide more details about servlet, core interfaces in Servlet API, Servlet 3.0 annotations, life cycle of Servlet and at the end we will create a simple login servlet application.

1. Servlet Overview
2. Common Gateway Interface (CGI)
3. CGI vs Servlet
4. Servlet API Hierarchy
    1. Servlet Interface
    2. ServletConfig Interface
    3. ServletContext interface
    4. ServletRequest interface
    5. ServletResponse interface
    6. RequestDispatcher interface
    7. GenericServlet class
    8. HTTPServlet class
5. Servlet Attributes
6. Annotations in Servlet 3
7. Servlet Login Example

# A. Servlet Overview

Servlet is J2EE server driven technology to create web applications in java. The *javax.servlet* and *javax.servlet.http* packages provide interfaces and classes for writing our own servlets.

All servlets must implement the *javax.servlet.Servlet* interface, which defines servlet lifecycle methods. When implementing a generic service, we can extend the *GenericServlet* class provided with the Java Servlet API. The *HttpServlet* class provides methods, such as *doGet()* and *doPost()*, for handling HTTP-specific services.

Most of the times, web applications are accessed using HTTP protocol and that's why we mostly extend *HttpServlet* class.

# B. Common Gateway Interface (CGI)

Before introduction of Servlet API, CGI technology was used to create dynamic web applications. CGI technology has many drawbacks such as creating separate process for each request, platform dependent code (C, C++), high memory usage and slow performance.

# C. CGI vs Servlet

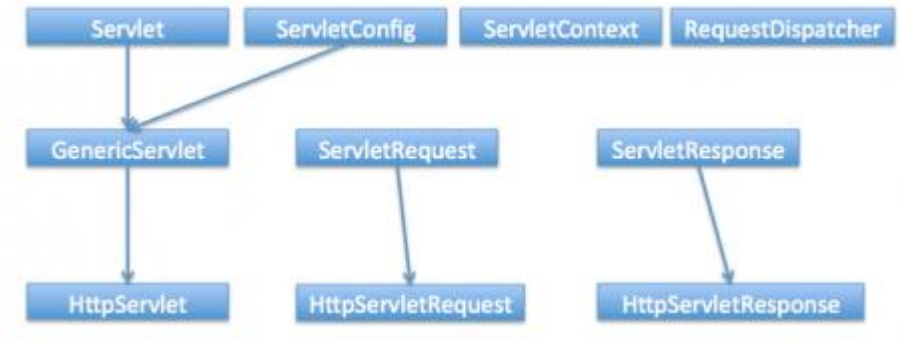
Servlet technology was introduced to overcome the shortcomings of CGI technology.

- Servlets provide better performance that CGI in terms of processing time, memory utilization because servlets uses benefits of multithreading and for each request a new thread is created, that is faster than loading creating new Object for each request with CGI.
- Servlets and platform and system independent, the web application developed with Servlet can be run on any standard web container such as Tomcat, JBoss, and Glassfish servers and on operating systems such as Windows, Linux, UNIX, Solaris, and Mac etc.

- Servlets are robust because container takes care of life cycle of servlet and we don't need to worry about memory leaks, security, garbage collection etc.
- Servlets are maintainable and learning curve is small because all we need to take care is business logic for our application.

# D. Servlet API Hierarchy

javax.servlet.Servlet is the base **interface** of Servlet API. There are some other interfaces and classes that we should be aware of when working with Servlets. Also with Servlet 3.0 specs, servlet API introduced use of annotations rather than having all the servlet configuration in deployment descriptor. In this section, we will look into important Servlet API interfaces, classes and annotations that we will use further in developing our application. The below diagram shows servlet API hierarchy.




### 1. Servlet Interface

javax.servlet.Servlet is the base interface of **Java Servlet API**. Servlet interface declares the life cycle methods of servlet. All the servlet classes are required to implement this interface. The methods declared in this interface are:

i. **public abstract void init(ServletConfig paramServletConfig) throws ServletException** – This is the very important method that is invoked by servlet container to initialized the servlet and ServletConfig parameters. The servlet is not ready to process client request until unless **init()** method is finished executing. This method

is called only once in servlet lifecycle and make Servlet class different from normal java objects. We can extend this method in our servlet classes to initialize resources such as DB Connection, Socket connection etc.

ii. **public abstract ServletConfig getServletConfig()** – This method returns a servlet config object, which contains any initialization parameters and startup configuration for this servlet. We can use this method to get the init parameters of servlet defines in deployment descriptor (web.xml) or through annotation in Servlet 3. We will look into ServletConfig interface later on.

iii. **public abstract void service(ServletRequest req, ServletResponse res) throws ServletException, IOException** – This method is responsible for processing the client request. Whenever servlet container receives any request, it creates a new thread and execute the service() method by passing request and response as argument. Servlets usually run in multi-threaded environment, so it's developer responsibility to keep shared resources thread-safe using **synchronization**.

iv. **public abstract String getServletInfo()** – This method returns string containing information about the servlet, such as its author, version, and copyright. The string returned should be plain text and can't have markups.

v. **public abstract void destroy()** – This method can be called only once in servlet life cycle and used to close any open resources. This is like finalize method of a java class.

## 2. ServletConfig Interface

*javax.servlet.ServletConfig* is used to pass configuration information to Servlet. Every servlet has it's own ServletConfig object and servlet container is responsible for instantiating this object. We can provide servlet init parameters in **web.xml** file or through use of *WebInitParam* annotation. We can use **getServletConfig()** method to get the ServletConfig object of the servlet.

The important methods of ServletConfig interface are:

i. **public abstract ServletContext getServletContext()** – This method returns the ServletContext object for the servlet. We will look into ServletContext interface in next section.
ii. **public abstract Enumeration getInitParameterNames()** – This method returns the Enumeration of name of init parameters defined for the servlet. If there are no init parameters defined, this method returns empty enumeration.
iii. **public abstract String getInitParameter(String paramString)** – This method can be used to get the specific init parameter value by name. If parameter is not present with the name, it returns null.

## 3. ServletContext interface

javax.servlet.ServletContext interface provides access to web application variables to the servlet. The ServletContext is unique object and available to all the servlets in the web application. When we want some init parameters to be available to multiple or all of the servlets in the web application, we can use ServletContext object and define parameters in web.xml using **<context-param>** element. We can get the ServletContext object via the getServletContext() method of ServletConfig. Servlet engines may also provide context objects that are unique to a group of servlets and which is tied to a specific portion of the URL path namespace of the host.

Some of the important methods of ServletContext are:

i. **public abstract ServletContext getContext(String uripath)** – This method returns ServletContext object for a particular uripath or null if not available or not visible to the servlet.
ii. **public abstract URL getResource(String path) throws MalformedURLException** – This method return URL object allowing access to any content resource requested. We can access items whether they reside on the local file system, a remote file system, a database, or a remote network site without knowing the specific details of how to obtain the resources.
iii. **public abstract InputStream getResourceAsStream(String path)** – This method returns an input stream to the given resource path or null if not found.

iv. **public abstract RequestDispatcher getRequestDispatcher(String urlpath)** – This method is mostly used to obtain a reference to another servlet. After obtaining a RequestDispatcher, the servlet programmer forward a request to the target component or include content from it.
v. **public abstract void log(String msg)** – This method is used to write given message string to the servlet log file.
vi. **public abstract Object getAttribute(String name)** – Return the object attribute for the given name. We can get enumeration of all the attributes using **public abstract Enumeration getAttributeNames()** method.
vii. **public abstract void setAttribute(String paramString, Object paramObject)** – This method is used to set the attribute with application scope. The attribute will be accessible to all the other servlets having access to this ServletContext. We can remove an attribute using **public abstract void removeAttribute(String paramString)** method.
viii. **String getInitParameter(String name)** – This method returns the String value for the init parameter defined with name in web.xml, returns null if parameter name doesn't exist. We can use **Enumeration getInitParameterNames()** to get enumeration of all the init parameter names.
ix. **boolean setInitParameter(String paramString1, String paramString2)** – We can use this method to set init parameters to the application.

**Note**: Ideally the name of this interface should be ApplicationContext because it's for the application and not specific to any servlet. Also don't get confused it with the servlet context passed in the URL to access the web application.

### 4. ServletRequest interface

ServletRequest interface is used to provide client request information to the servlet. Servlet container creates ServletRequest object from client request and pass it to the servlet service() method for processing.

Some of the important methods of ServletRequest interface are:

i. **Object getAttribute(String name)** – This method returns the value of named attribute as Object and null if it's not present. We can use

getAttributeNames() method to get the enumeration of attribute names for the request. This interface also provide methods for setting and removing attributes.

ii. **String getParameter(String name)** – This method returns the request parameter as String. We can use getParameterNames() method to get the enumeration of parameter names for the request.
iii. **String getServerName()** – returns the hostname of the server.
iv. **int getServerPort()** – returns the port number of the server on which it's listening.

The child interface of ServletRequest is HttpServletRequest that contains some other methods for session management, cookies and authorization of request.

### 5. ServletResponse interface

ServletResponse interface is used by servlet in sending response to the client. Servlet container creates the ServletResponse object and pass it to servlet service() method and later use the response object to generate the HTML response for client.

Some of the important methods in HttpServletResponse are:

i. **void addCookie(Cookie cookie)** – Used to add cookie to the response.
ii. **void addHeader(String name, String value)** – used to add a response header with the given name and value.
iii. **String encodeURL(java.lang.String url)** – encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.
iv. **String getHeader(String name)** – return the value for the specified header, or null if this header has not been set.
v. **void sendRedirect(String location)** – used to send a temporary redirect response to the client using the specified redirect location URL.
vi. **void setStatus(int sc)** – used to set the status code for the response.

## 6. RequestDispatcher interface

RequestDispatcher interface is used to forward the request to another resource that can be HTML, JSP or another servlet in the same context. We can also use this to include the content of another resource to the response. This interface is used for servlet communication within the same context.

There are two methods defined in this interface:

1. **void forward(ServletRequest request, ServletResponse response)** – forwards the request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
2. **void include(ServletRequest request, ServletResponse response)** – includes the content of a resource (servlet, JSP page, HTML file) in the response.

We can get RequestDispatcher in a servlet using ServletContext *getRequestDispatcher(String path)* method. The path must begin with a / and is interpreted as relative to the current context root.

## 7. GenericServlet class

GenericServlet is an **abstract class** that implements Servlet, ServletConfig and Serializable interface. GenericServlet provide default implementation of all the Servlet life cycle methods and ServletConfig methods and makes our life easier when we extend this class, we need to override only the methods we want and rest of them we can work with the default implementation. Most of the methods defined in this class are only for easy access to common methods defined in Servlet and ServletConfig interfaces.

One of the important method in GenericServlet class is no-argument init() method and we should override this method in our servlet program if we have to initialize some resources before processing any request from servlet.

## 8. HTTPServlet class

HTTPServlet is an abstract class that extends GenericServlet and provides base for creating HTTP based web applications. There are methods defined to be overridden by subclasses for different HTTP methods.

i. doGet(), for HTTP GET requests
ii. doPost(), for HTTP POST requests
iii. doPut(), for HTTP PUT requests
iv. doDelete(), for HTTP DELETE requests

# D. Servlet Attributes

Servlet attributes are used for inter-servlet communication, we can set, get and remove attributes in web application. There are three scopes for servlet attributes – **request scope**, **session scope** and **application scope**.

**ServletRequest**, **HttpSession** and **ServletContext** interfaces provide methods to get/set/remove attributes from request, session and application scope respectively.

Servlet attributes are different from init parameters defined in **web.xml** for ServletConfig or ServletContext.

# E. Annotations in Servlet 3

Prior to Servlet 3, all the servlet mapping and it's init parameters were used to defined in web.xml, this was not convenient and more error prone when number of servlets are huge in an application.

Servlet 3 introduced use of **java annotations** to define a servlet, filter and listener servlets and init parameters.

Some of the important Servlet API annotations are:

1. **WebServlet** – We can use this annotation with Servlet classes to define init parameters, loadOnStartup value, description and url patterns etc. At least one URL pattern MUST be declared in either the value or urlPattern attribute of the annotation, but not both. The class on which this annotation is declared MUST extend HttpServlet.
2. **WebInitParam** – This annotation is used to define init parameters for servlet or filter, it contains name, value pair and we can provide description also. This annotation can be used within a WebFilter or WebServlet annotation.

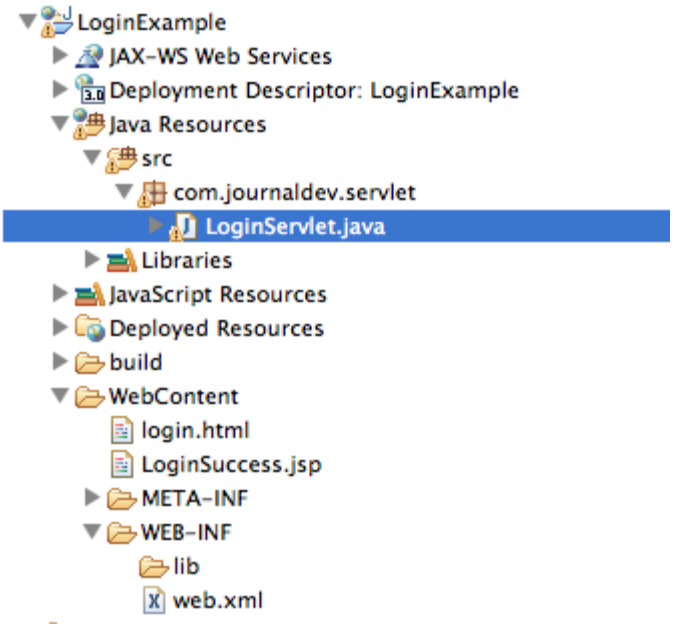
3. **WebFilter** – This annotation is used to declare a servlet filter. This annotation is processed by the container during deployment, the Filter class in which it is found will be created as per the configuration and applied to the URL patterns, Servlets and DispatcherTypes. The annotated class MUST implement javax.servlet.Filter interface.
4. **WebListener** – The annotation used to declare a listener for various types of event, in a given web application context.

Note: We will look into Servlet Filters and Listeners in future articles, in this article our focus is to learn about base interfaces and classes of Servlet API.

# F. Servlet Login Example

Now we are ready to create our login servlet example, in this example I will use simple HTML, JSP and servlet that will authenticate the user credentials. We will also see the use of ServletContext init parameters, attributes, ServletConfig init parameters and RequestDispatcher include() and response sendRedirect() usage.

Our Final Dynamic Web Project will look like below image. I am using Eclipse and Tomcat for the application, the process to create dynamic web project is provided in **Java Web Applications** tutorial.



Here is our login HTML page, we will put it in the welcome files list in the web.xml so that when we launch the application it will open the login page.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="US-ASCII">
<title>Login Page</title>
</head>
<body>

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

Username: <input type="text" name="user">
<br>
Password: <input type="password" name="pwd">
<br>
<input type="submit" value="Login">
</form>
</body>
</html>
```

If the login will be successful, the user will be presented with new JSP page with login successful message. JSP page code is like below.

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Login Success Page</title>
</head>
<body>
<h3>Hi Pankaj, Login successful.</h3>
<a href="login.html">Login Page</a>
</body>
</html>
```

Here is the web.xml deployment descriptor file where we have defined servlet context init parameters and welcome page.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID"
version="3.0">
  <display-name>LoginExample</display-name>
```

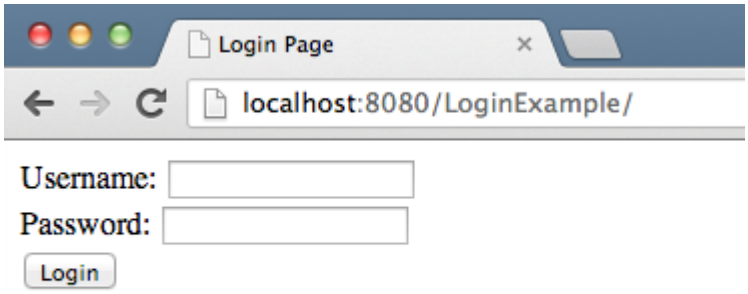
```
  <welcome-file-list>
    <welcome-file>login.html</welcome-file>
  </welcome-file-list>

  <context-param>
  <param-name>dbURL</param-name>
  <param-value>jdbc:mysql://localhost/mysql_db</param-value>
  </context-param>
  <context-param>
  <param-name>dbUser</param-name>
  <param-value>mysql_user</param-value>
  </context-param>
  <context-param>
  <param-name>dbUserPwd</param-name>
  <param-value>mysql_pwd</param-value>
  </context-param>
</web-app>
```

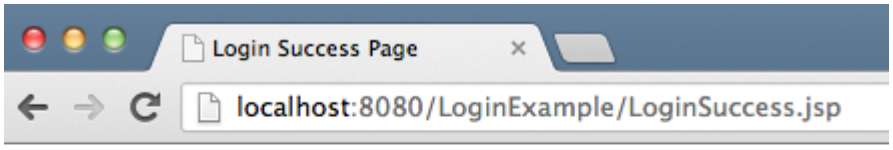
Here is our final Servlet class for authenticating the user credentials, notice the use of annotations for Servlet configuration and ServletConfig init parameters.

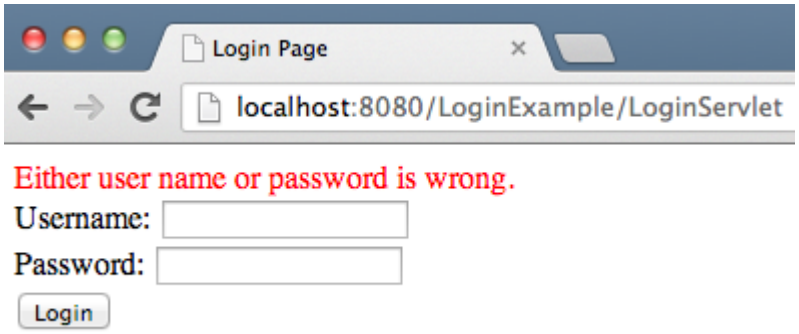
```
package com.journaldev.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class LoginServlet
 */
@WebServlet(
        description = "Login Servlet",
        urlPatterns = { "/LoginServlet" },
        initParams = {
                @WebInitParam(name = "user", value = "Pankaj"),
                @WebInitParam(name = "password", value = "journaldev")
        })
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;


    public void init() throws ServletException {
```

```
        //we can create DB connection resource here and set it to
Servlet context

if(getServletContext().getInitParameter("dbURL").equals("jdbc:mysql://l
ocalhost/mysql_db") &&

getServletContext().getInitParameter("dbUser").equals("mysql_user") &&

getServletContext().getInitParameter("dbUserPwd").equals("mysql_pwd"))
        getServletContext().setAttribute("DB_Success", "True");
        else throw new ServletException("DB Connection error");
    }


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

        //get request parameters for userID and password
        String user = request.getParameter("user");
        String pwd = request.getParameter("pwd");

        //get servlet config init params
        String userID = getServletConfig().getInitParameter("user");
        String password =
getServletConfig().getInitParameter("password");
        //logging example
        log("User="+user+"::password="+pwd);

        if(userID.equals(user) && password.equals(pwd)){
            response.sendRedirect("LoginSuccess.jsp");
        }else{
            RequestDispatcher rd =
getServletContext().getRequestDispatcher("/login.html");
            PrintWriter out= response.getWriter();
            out.println("<font color=red>Either user name or password
is wrong.</font>");
            rd.include(request, response);

        }

    }

}
```

Below screenshots shows the different pages based on the user password combinations for successful login and failed logins.




**You can download the source code of "Servlet Login Example Project" from here.**

# 3. Servlet Filters

In this section, we will learn about the Filter in servlet. We will look into various usage of filter, how can we create filter and learn it's usage with a simple web application.

In this article, we will lean about the Filter in servlet. We will look into various usage of filter, how can we create filter and learn it's usage with a simple web application.

1. Why do we have Servlet Filter?
2. Filter interface
3. WebFilter annotation
4. Filter configuration in web.xml
5. Servlet Filter Example for Logging and session validation

## A. Why do we have Servlet Filter?

In the last article, we learned how we can manage session in web application and if we want to make sure that a resource is accessible only when user session is valid, we can achieve this using servlet session attributes. The approach is simple but if we have a lot of servlets and jsps, then it will become hard to maintain because of redundant code. If we want to change the attribute name in future, we will have to change all the places where we have session authentication.

That's why we have servlet filter. Servlet Filters are **pluggable** java components that we can use to intercept and process requests *before* they are sent to servlets and response *after* servlet code is finished and before container sends the response back to the client.

Some common tasks that we can do with filters are:

- Logging request parameters to log files.
- Authentication and autherization of request for resources.
- Formatting of request body or header before sending it to servlet.
- Compressing the response data sent to the client.

- Alter response by adding some cookies, header information etc.

As I mentioned earlier, **servlet filters are pluggable** and configured in deployment descriptor (web.xml) file. Servlets and filters both are unaware of each other and we can add or remove a filter just by editing web.xml.

We can have multiple filters for a single resource and we can create a chain of filters for a single resource in web.xml. We can create a Servlet Filter by implementing *javax.servlet.Filter* interface.

# B. Filter interface

Filter interface is similar to Servlet interface and we need to implement it to create our own servlet filter. Filter interface contains lifecycle methods of a Filter and it's managed by servlet container.

Filter interface lifecycle methods are:

1. **void init(FilterConfig paramFilterConfig)** – When container initializes the Filter, this is the method that gets invoked. This method is called only once in the lifecycle of filter and we should initialize any resources in this method. **FilterConfig** is used by container to provide init parameters and servlet context object to the Filter. We can throw ServletException in this method.
2. **doFilter(ServletRequest paramServletRequest, ServletResponse paramServletResponse, FilterChain paramFilterChain)** – This is the method invoked every time by container when it has to apply filter to a resource. Container provides request and response object references to filter as argument. **FilterChain** is used to invoke the next filter in the chain. This is a great example of **Chain of Responsibility Pattern**.
3. **void destroy()** – When container offloads the Filter instance, it invokes the destroy() method. This is the method where we can close any resources opened by filter. This method is called only once in the lifetime of filter.

# C. WebFilter annotation

*javax.servlet.annotation.WebFilter* was introduced in Servlet 3.0 and we can use this annotation to declare a servlet filter. We can use this annotation to define init parameters, filter name and description, servlets, url patterns and dispatcher types to apply the filter. If you make frequent changes to the filter configurations, it's better to use web.xml because that will not require you to recompile the filter class.

**Read**: **Java Annotations Tutorial**

# D. Filter configuration in web.xml

We can declare a filter in web.xml like below.

```
<filter>
  <filter-name>RequestLoggingFilter</filter-name> <!-- mandatory -->
  <filter-
class>com.journaldev.servlet.filters.RequestLoggingFilter</filter-
class> <!-- mandatory -->
  <init-param> <!-- optional -->
  <param-name>test</param-name>
  <param-value>testValue</param-value>
  </init-param>
</filter>
```

We can map a Filter to servlet classes or url-patterns like below.

```
<filter-mapping>
  <filter-name>RequestLoggingFilter</filter-name> <!-- mandatory -->
  <url-pattern>/*</url-pattern> <!-- either url-pattern or servlet-name
is mandatory -->
  <servlet-name>LoginServlet</servlet-name>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

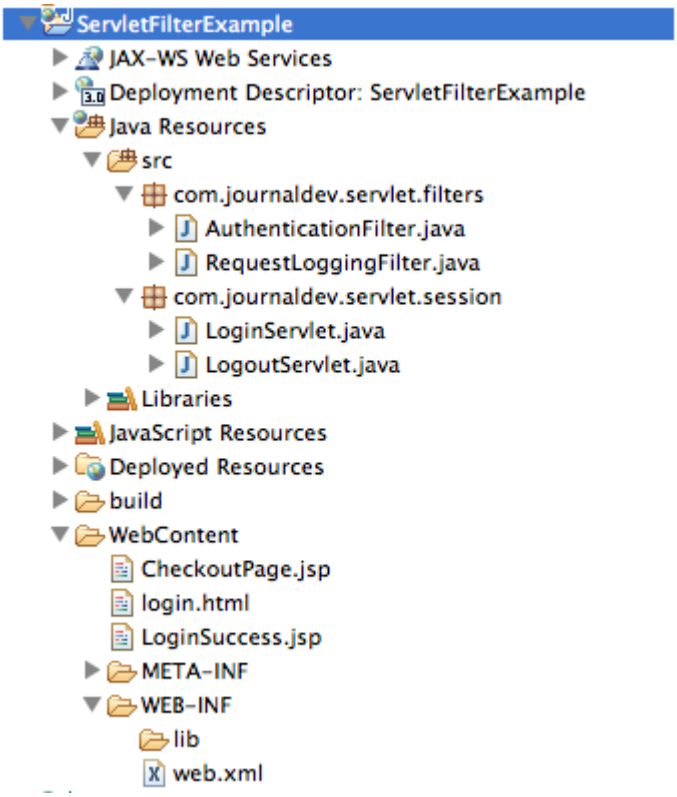
Note: While creating the filter chain for a servlet, container first processes the url-patterns and then servlet-names, so if you have to make sure that filters are getting executed in a particular order, give extra attention while defining the filter mapping.

Servlet Filters are generally used for client requests but sometimes we want to apply filters with **RequestDispatcher** also, we can use dispatcher element in this case, the possible values are REQUEST, FORWARD, INCLUDE, ERROR and ASYNC. If no dispatcher is defined then it's applied only to client requests.

# E. Servlet Filter Example for Logging and session validation

In our **servlet filter example**, we will create filters to log request cookies and parameters and validate session to all the resources except static HTMLs and LoginServlet because it will not have a session.

We will create a dynamic web project **ServletFilterExample** whose project structure will look like below image.



login.html is the entry point of our application where user will provide login id and password for authentication.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="US-ASCII">
<title>Login Page</title>
</head>
<body>

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

Username: <input type="text" name="user">
<br>
Password: <input type="password" name="pwd">
<br>
<input type="submit" value="Login">
</form>
</body>
</html>
```

LoginServlet is used to authenticate the request from client for login.

```
package com.journaldev.servlet.session;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 * Servlet implementation class LoginServlet
 */
@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final String userID = "admin";
    private final String password = "password";

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

```
        // get request parameters for userID and password
        String user = request.getParameter("user");
        String pwd = request.getParameter("pwd");

        if(userID.equals(user) && password.equals(pwd)){
            HttpSession session = request.getSession();
            session.setAttribute("user", "Pankaj");
            //setting session to expiry in 30 mins
            session.setMaxInactiveInterval(30*60);
            Cookie userName = new Cookie("user", user);
            userName.setMaxAge(30*60);
            response.addCookie(userName);
            response.sendRedirect("LoginSuccess.jsp");
        }else{
            RequestDispatcher rd =
getServletContext().getRequestDispatcher("/login.html");
            PrintWriter out= response.getWriter();
            out.println("<font color=red>Either user name or password
is wrong.</font>");
            rd.include(request, response);
        }

    }

}
```

When client is authenticated, it's forwarded to LoginSuccess.jsp

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Login Success Page</title>
</head>
<body>
<%
//allow access only if session exists
String user = (String) session.getAttribute("user");
String userName = null;
String sessionID = null;
Cookie[] cookies = request.getCookies();
if(cookies !=null){
for(Cookie cookie : cookies){
    if(cookie.getName().equals("user")) userName = cookie.getValue();
    if(cookie.getName().equals("JSESSIONID")) sessionID =
cookie.getValue();
}
}
```

```
%>
<h3>Hi <%=userName %>, Login successful. Your Session ID=<%=sessionID
%></h3>
<br>
User=<%=user %>
<br>
<a href="CheckoutPage.jsp">Checkout Page</a>
<form action="LogoutServlet" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>
```

LogoutServlet is invoked when client clicks on Logout button in any of the JSP pages.

```
package com.journaldev.servlet.session;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 * Servlet implementation class LogoutServlet
 */
@WebServlet("/LogoutServlet")
public class LogoutServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        Cookie[] cookies = request.getCookies();
        if(cookies != null){
        for(Cookie cookie : cookies){
            if(cookie.getName().equals("JSESSIONID")){
                System.out.println("JSESSIONID="+cookie.getValue());
                break;
            }
        }
        }
        //invalidate the session if exists
        HttpSession session = request.getSession(false);
        System.out.println("User="+session.getAttribute("user"));
        if(session != null){
            session.invalidate();
```

```
        }
        response.sendRedirect("login.html");
    }

}
```

Now we will create logging and authentication filter classes.

```
package com.journaldev.servlet.filters;

import java.io.IOException;
import java.util.Enumeration;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;

/**
 * Servlet Filter implementation class RequestLoggingFilter
 */
@WebFilter("/RequestLoggingFilter")
public class RequestLoggingFilter implements Filter {

    private ServletContext context;

    public void init(FilterConfig fConfig) throws ServletException {
        this.context = fConfig.getServletContext();
        this.context.log("RequestLoggingFilter initialized");
    }

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        Enumeration<String> params = req.getParameterNames();
        while(params.hasMoreElements()){
            String name = params.nextElement();
            String value = request.getParameter(name);
            this.context.log(req.getRemoteAddr() + "::Request
Params::{"+name+"="+value+"}");
        }

        Cookie[] cookies = req.getCookies();
        if(cookies != null){
            for(Cookie cookie : cookies){
```

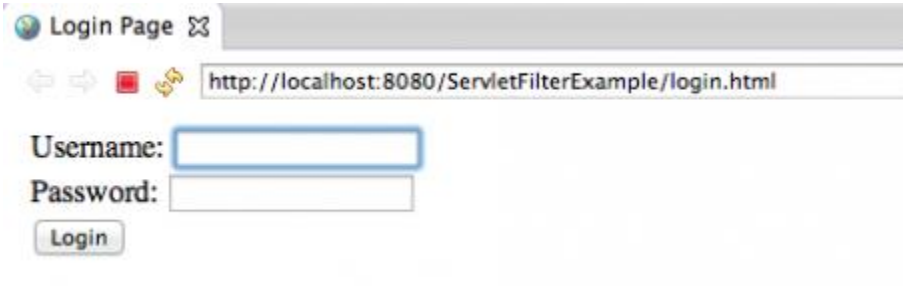
```
                this.context.log(req.getRemoteAddr() +
"::Cookie::{"+cookie.getName()+","+cookie.getValue()+"}");
            }
        }
        // pass the request along the filter chain
        chain.doFilter(request, response);
    }

    public void destroy() {
        //we can close resources here
    }

}
package com.journaldev.servlet.filters;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebFilter("/AuthenticationFilter")
public class AuthenticationFilter implements Filter {

    private ServletContext context;

    public void init(FilterConfig fConfig) throws ServletException {
        this.context = fConfig.getServletContext();
        this.context.log("AuthenticationFilter initialized");
    }

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;

        String uri = req.getRequestURI();
        this.context.log("Requested Resource::"+uri);

        HttpSession session = req.getSession(false);

        if(session == null && !(uri.endsWith("html") ||
uri.endsWith("LoginServlet"))){
            this.context.log("Unauthorized access request");
```

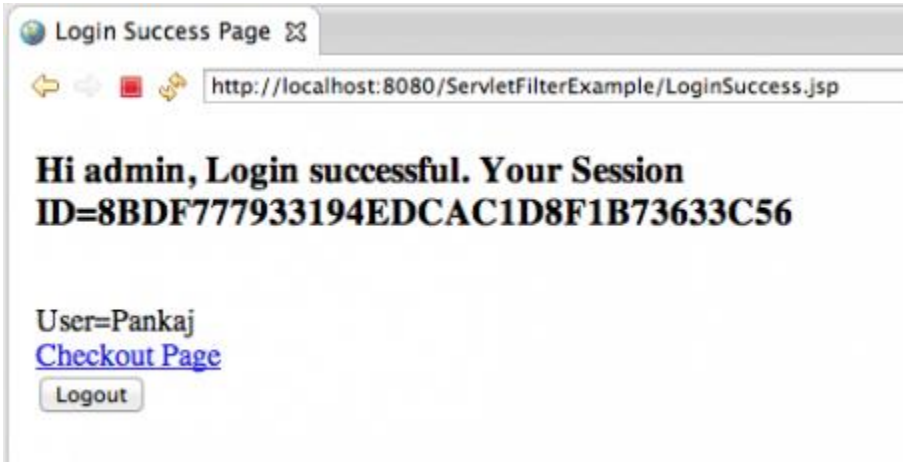
```
            res.sendRedirect("login.html");
        }else{
            // pass the request along the filter chain
            chain.doFilter(request, response);
        }


    }

     public void destroy() {
        //close any resources here
    }

}
```

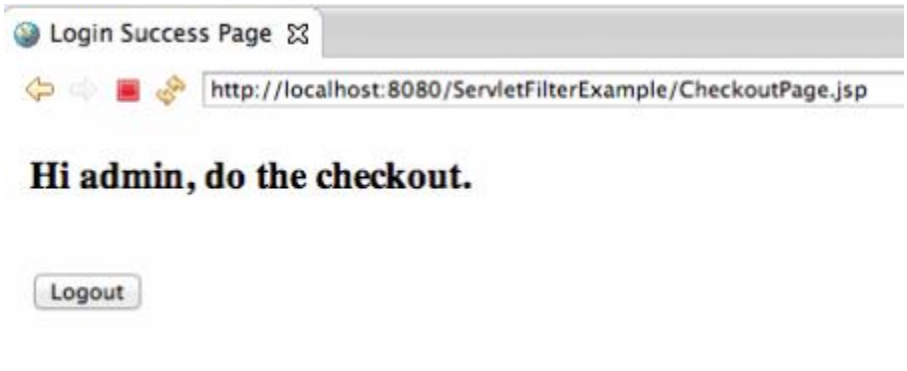
Notice that we are not authenticating any HTML page or LoginServlet. Now we will configure these filters mapping in the web.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
  <display-name>ServletFilterExample</display-name>
  <welcome-file-list>
    <welcome-file>login.html</welcome-file>
  </welcome-file-list>

  <filter>
    <filter-name>RequestLoggingFilter</filter-name>
    <filter-
class>com.journaldev.servlet.filters.RequestLoggingFilter</filter-
class>
  </filter>
  <filter>
    <filter-name>AuthenticationFilter</filter-name>
    <filter-
class>com.journaldev.servlet.filters.AuthenticationFilter</filter-
class>
  </filter>

  <filter-mapping>
    <filter-name>RequestLoggingFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
  </filter-mapping>
  <filter-mapping>
    <filter-name>AuthenticationFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

Now when we will run our application, we will get response pages like below images.




If you are not logged in and try to access any JSP page, you will be forwarded to the login page.

In server log file, you can see the logs written by filters as well as servlets.

```
Aug 13, 2013 1:06:07 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,B7275762B8D23121152B1270D6EB240A
}
Aug 13, 2013 1:06:07 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/
```

```
Aug 13, 2013 1:06:07 AM org.apache.catalina.core.ApplicationContext log
INFO: Unauthorized access request
Aug 13, 2013 1:06:07 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,B7275762B8D23121152B1270D6EB240A
}
Aug 13, 2013 1:06:07 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/login.html
Aug 13, 2013 1:06:43 AM org.apache.catalina.core.ApplicationContext log
INFO: 0:0:0:0:0:0:0:1%0::Request Params::{pwd=password}
Aug 13, 2013 1:06:43 AM org.apache.catalina.core.ApplicationContext log
INFO: 0:0:0:0:0:0:0:1%0::Request Params::{user=admin}
Aug 13, 2013 1:06:43 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,B7275762B8D23121152B1270D6EB240A
}
Aug 13, 2013 1:06:43 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/LoginServlet
Aug 13, 2013 1:06:43 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,8BDF777933194EDCAC1D8F1B73633C56
}
Aug 13, 2013 1:06:43 AM org.apache.catalina.core.ApplicationContext log
INFO: 0:0:0:0:0:0:0:1%0::Cookie::{user,admin}
Aug 13, 2013 1:06:43 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/LoginSuccess.jsp
Aug 13, 2013 1:06:52 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,8BDF777933194EDCAC1D8F1B73633C56
}
Aug 13, 2013 1:06:52 AM org.apache.catalina.core.ApplicationContext log
INFO: 0:0:0:0:0:0:0:1%0::Cookie::{user,admin}
Aug 13, 2013 1:06:52 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/CheckoutPage.jsp
Aug 13, 2013 1:07:00 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,8BDF777933194EDCAC1D8F1B73633C56
}
Aug 13, 2013 1:07:00 AM org.apache.catalina.core.ApplicationContext log
INFO: 0:0:0:0:0:0:0:1%0::Cookie::{user,admin}
Aug 13, 2013 1:07:00 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/LogoutServlet
JSESSIONID=8BDF777933194EDCAC1D8F1B73633C56
User=Pankaj
Aug 13, 2013 1:07:00 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,8BDF777933194EDCAC1D8F1B73633C56
}
Aug 13, 2013 1:07:00 AM org.apache.catalina.core.ApplicationContext log
INFO: 0:0:0:0:0:0:0:1%0::Cookie::{user,admin}
Aug 13, 2013 1:07:00 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/login.html
Aug 13, 2013 1:07:06 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,8BDF777933194EDCAC1D8F1B73633C56
}
Aug 13, 2013 1:07:07 AM org.apache.catalina.core.ApplicationContext log
```

```
INFO: 0:0:0:0:0:0:0:1%0::Cookie::{user,admin}
Aug 13, 2013 1:07:07 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/LoginSuccess.jsp
Aug 13, 2013 1:07:07 AM org.apache.catalina.core.ApplicationContext log
INFO: Unauthorized access request
Aug 13, 2013 1:07:07 AM org.apache.catalina.core.ApplicationContext log
INFO:
0:0:0:0:0:0:0:1%0::Cookie::{JSESSIONID,8BDF777933194EDCAC1D8F1B73633C56
}
Aug 13, 2013 1:07:07 AM org.apache.catalina.core.ApplicationContext log
INFO: 0:0:0:0:0:0:0:1%0::Cookie::{user,admin}
Aug 13, 2013 1:07:07 AM org.apache.catalina.core.ApplicationContext log
INFO: Requested Resource::/ServletFilterExample/login.html
```

That's all for Filter in servlet, it's one of the important feature of J2EE web application and we should use it for common tasks performed by various servlets. In future posts, we will look into servlet listeners and cookies.

**You can download the source code of "Servlet Filter Example Project" from here.**

Struts 2 uses Servlet Filter to intercept the client requests and forward them to appropriate action classes, these are called Struts 2 Interceptors.

# 4. Servlet Listeners

In this secttion, we will look into **servlet listener**, benefits of listeners, some common tasks that we can do with listeners, servlet API listener interfaces and Event objects. In the end we will create a simple web project to show example of commonly used Listener implementation for **ServletContext**, **Session** and **ServletRequest**.

1. Why do we have Servlet Listener?
2. Servlet Listener Interfaces and Event Objects
3. Servlet Listener Configuration
4. Servlet Listener Example
    1. ServletContextListener implementation
    2. ServletContextAttributeListener implementation
    3. HttpSessionListener implementation
    4. ServletRequestListener implementation

## A. Why do we have Servlet Listener?

We know that using *ServletContext*, we can create an attribute with application scope that all other servlets can access but we can initialize **ServletContext init parameters as String only** in deployment descriptor (web.xml). What if our application is database oriented and we want to set an attribute in ServletContext for Database Connection. If you application has a single entry point (user login), then you can do it in the first servlet request but if we have multiple entry points then doing it everywhere will result in a lot of code redundancy. Also if database is down or not configured properly, we won't know until first client request comes to server. To handle these scenario, servlet API provides Listener interfaces that we can implement and configure to listen to an event and do certain operations.

**Event** is occurrence of something, in web application world an event can be initialization of application, destroying an application, request from client, creating/destroying a session, attribute modification in session etc.

**Servlet API** provides different types of Listener interfaces that we can implement and configure in web.xml to process something when a particular

event occurs. For example, in above scenario we can create a Listener for the application startup event to read context init parameters and create a database connection and set it to context attribute for use by other resources.

# B. Servlet Listener Interfaces and Event Objects

Servlet API provides different kind of listeners for different types of Events. Listener interfaces declare methods to work with a group of similar events, for example we have ServletContext Listener to listen to startup and shutdown event of context. Every method in listener interface takes Event object as input. Event object works as a wrapper to provide specific object to the listeners.

Servlet API provides following event objects.

1. **javax.servlet.AsyncEvent** – Event that gets fired when the asynchronous operation initiated on a ServletRequest (via a call to ServletRequest#startAsync or ServletRequest#startAsync(ServletRequest, ServletResponse)) has completed, timed out, or produced an error.
2. **javax.servlet.http.HttpSessionBindingEvent** – Events of this type are either sent to an object that implements HttpSessionBindingListener when it is bound or unbound from a session, or to a HttpSessionAttributeListener that has been configured in the web.xml when any attribute is bound, unbound or replaced in a session.
   The session binds the object by a call to HttpSession.setAttribute and unbinds the object by a call to HttpSession.removeAttribute. We can use this event for cleanup activities when object is removed from session.
3. **javax.servlet.http.HttpSessionEvent** – This is the class representing event notifications for changes to sessions within a web application.

4. **javax.servlet.ServletContextAttributeEvent** – Event class for notifications about changes to the attributes of the ServletContext of a web application.
5. **javax.servlet.ServletContextEvent** – This is the event class for notifications about changes to the servlet context of a web application.
6. **javax.servlet.ServletRequestEvent** – Events of this kind indicate lifecycle events for a ServletRequest. The source of the event is the ServletContext of this web application.
7. **javax.servlet.ServletRequestAttributeEvent** – This is the event class for notifications of changes to the attributes of the servlet request in an application.

Servlet API provides following Listener interfaces.

1. **javax.servlet.AsyncListener** – Listener that will be notified in the event that an asynchronous operation initiated on a ServletRequest to which the listener had been added has completed, timed out, or resulted in an error.
2. **javax.servlet.ServletContextListener** – Interface for receiving notification events about ServletContext lifecycle changes.
3. **javax.servlet.ServletContextAttributeListener** – Interface for receiving notification events about ServletContext attribute changes.
4. **javax.servlet.ServletRequestListener** – Interface for receiving notification events about requests coming into and going out of scope of a web application.
5. **javax.servlet.ServletRequestAttributeListener** – Interface for receiving notification events about ServletRequest attribute changes.
6. **javax.servlet.http.HttpSessionListener** – Interface for receiving notification events about HttpSession lifecycle changes.
7. **javax.servlet.http.HttpSessionBindingListener** – Causes an object to be notified when it is bound to or unbound from a session.
8. **javax.servlet.http.HttpSessionAttributeListener** – Interface for receiving notification events about HttpSession attribute changes.
9. **javax.servlet.http.HttpSessionActivationListener** – Objects that are bound to a session may listen to container events notifying them that sessions will be passivated and that session will be activated. A container that migrates session between VMs or persists sessions is required to notify all attributes bound to sessions implementing HttpSessionActivationListener.

# C. Servlet Listener Configuration

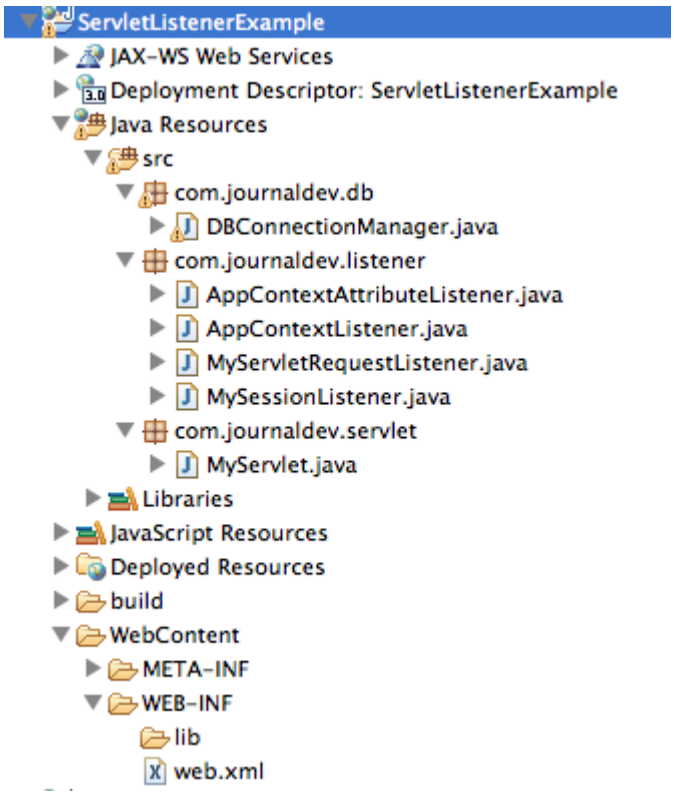
We can use **@WebListener** **annotation** to declare a class as Listener, however the class should implement one or more of the Listener interfaces.

We can define listener in web.xml as:

```
<listener>
    <listener-
class>com.journaldev.listener.AppContextListener</listener-class>
</listener>
```

# D. Servlet Listener Example

Let's create a simple web application to see listeners in action. We will create dynamic web project in Eclipse **ServletListenerExample** those project structure will look like below image.



**web.xml**: In deployment descriptor, I will define some context init params and listener configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID"
version="3.0">
  <display-name>ServletListenerExample</display-name>

  <context-param>
    <param-name>DBUSER</param-name>
    <param-value>pankaj</param-value>
  </context-param>
  <context-param>
    <param-name>DBPWD</param-name>
    <param-value>password</param-value>
  </context-param>
  <context-param>
    <param-name>DBURL</param-name>
    <param-value>jdbc:mysql://localhost/mysql_db</param-value>
  </context-param>

  <listener>
    <listener-
class>com.journaldev.listener.AppContextListener</listener-class>
  </listener>
  <listener>
    <listener-
class>com.journaldev.listener.AppContextAttributeListener</listener-
class>
  </listener>
  <listener>
    <listener-
class>com.journaldev.listener.MySessionListener</listener-class>
  </listener>
  <listener>
    <listener-
class>com.journaldev.listener.MyServletRequestListener</listener-class>
  </listener>
</web-app>
```

**DBConnectionManager**: This is the class for database connectivity, for simplicity I am not providing code for actual database connection. We will set this object as attribute to servlet context.

```
package com.journaldev.db;

import java.sql.Connection;
```

```
public class DBConnectionManager {

    private String dbURL;
    private String user;
    private String password;
    private Connection con;

    public DBConnectionManager(String url, String u, String p){
        this.dbURL=url;
        this.user=u;
        this.password=p;
        //create db connection now

    }

    public Connection getConnection(){
        return this.con;
    }

    public void closeConnection(){
        //close DB connection here
    }
}
```

**MyServlet**: A simple servlet class where I will work with session, attributes etc.

```
package com.journaldev.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebServlet("/MyServlet")
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            ServletContext ctx = request.getServletContext();
            ctx.setAttribute("User", "Pankaj");
            String user = (String) ctx.getAttribute("User");
            ctx.removeAttribute("User");
```

```
            HttpSession session = request.getSession();
            session.invalidate();

            PrintWriter out = response.getWriter();
            out.write("Hi "+user);
    }

}
```

Now we will implement listener classes, I am providing sample listener classes for commonly used listeners – ServletContextListener, ServletContextAttributeListener, ServletRequestListener and HttpSessionListener.

### 1. ServletContextListener implementation

We will read servlet context init parameters to create the DBConnectionManager object and set it as attribute to the ServletContext object.

```
package com.journaldev.listener;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

import com.journaldev.db.DBConnectionManager;

@WebListener
public class AppContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent
servletContextEvent) {
        ServletContext ctx = servletContextEvent.getServletContext();

        String url = ctx.getInitParameter("DBURL");
        String u = ctx.getInitParameter("DBUSER");
        String p = ctx.getInitParameter("DBPWD");

        //create database connection from init parameters and set it to
context
        DBConnectionManager dbManager = new DBConnectionManager(url, u,
p);
        ctx.setAttribute("DBManager", dbManager);
```

```
        System.out.println("Database connection initialized for
Application.");
    }

    public void contextDestroyed(ServletContextEvent
servletContextEvent) {
        ServletContext ctx = servletContextEvent.getServletContext();
        DBConnectionManager dbManager = (DBConnectionManager)
ctx.getAttribute("DBManager");
        dbManager.closeConnection();
        System.out.println("Database connection closed for
Application.");

    }

}
```

## 2. ServletContextAttributeListener implementation

A simple implementation to log the event when attribute is added, removed or replaced in servlet context.

```
package com.journaldev.listener;

import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class AppContextAttributeListener implements
ServletContextAttributeListener {

    public void attributeAdded(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("ServletContext attribute
added::{"+servletContextAttributeEvent.getName()+","+servletContextAttr
ibuteEvent.getValue()+"}");
    }

    public void attributeReplaced(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("ServletContext attribute
replaced::{"+servletContextAttributeEvent.getName()+","+servletContextA
ttributeEvent.getValue()+"}");
    }

    public void attributeRemoved(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("ServletContext attribute
removed::{"+servletContextAttributeEvent.getName()+","+servletContextAt
tributeEvent.getValue()+"}");
```

```
    }

}
```

### 3. HttpSessionListener implementation

A simple implementation to log the event when session is created or destroyed.

```
package com.journaldev.listener;

import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

@WebListener
public class MySessionListener implements HttpSessionListener {

    public void sessionCreated(HttpSessionEvent sessionEvent) {
        System.out.println("Session Created::
ID="+sessionEvent.getSession().getId());
    }

    public void sessionDestroyed(HttpSessionEvent sessionEvent) {
        System.out.println("Session Destroyed::
ID="+sessionEvent.getSession().getId());
    }

}
```

### 4. ServletRequestListener implementation

A simple implementation of ServletRequestListener interface to log the ServletRequest IP address when request is initialized and destroyed.

```
package com.journaldev.listener;

import javax.servlet.ServletRequest;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class MyServletRequestListener implements ServletRequestListener
{

    public void requestDestroyed(ServletRequestEvent
servletRequestEvent) {
```

```
        ServletRequest servletRequest =
servletRequestEvent.getServletRequest();
        System.out.println("ServletRequest destroyed. Remote
IP="+servletRequest.getRemoteAddr());
    }

    public void requestInitialized(ServletRequestEvent
servletRequestEvent) {
        ServletRequest servletRequest =
servletRequestEvent.getServletRequest();
        System.out.println("ServletRequest initialized. Remote
IP="+servletRequest.getRemoteAddr());
    }

}
```

Now when we will deploy our application and access MyServlet in browser with URL http://localhost:8080/ServletListenerExample/MyServlet, we will see following logs in the server log file.

```
ServletContext attribute
added::{DBManager,com.journaldev.db.DBConnectionManager@4def3d1b}
Database connection initialized for Application.
ServletContext attribute
added::{org.apache.jasper.compiler.TldLocationsCache,org.apache.jasper.
compiler.TldLocationsCache@1594df96}

ServletRequest initialized. Remote IP=0:0:0:0:0:0:0:1%0
ServletContext attribute added::{User,Pankaj}
ServletContext attribute removed::{User,Pankaj}
Session Created:: ID=8805E7AE4CCCF98AFD60142A6B300CD6
Session Destroyed:: ID=8805E7AE4CCCF98AFD60142A6B300CD6
ServletRequest destroyed. Remote IP=0:0:0:0:0:0:0:1%0


ServletRequest initialized. Remote IP=0:0:0:0:0:0:0:1%0
ServletContext attribute added::{User,Pankaj}
ServletContext attribute removed::{User,Pankaj}
Session Created:: ID=88A7A1388AB96F611840886012A4475F
Session Destroyed:: ID=88A7A1388AB96F611840886012A4475F
ServletRequest destroyed. Remote IP=0:0:0:0:0:0:0:1%0


Database connection closed for Application.
```

Notice the sequence of logs and it's in the order of execution. The last log will appear when you will shut down the application or shutdown the container.

**You can download the source code of "Servlet Listener Example Project" from here.**

# Copyright Notice

# References


1. http://www.journaldev.com/1854/java-web-application-tutorial-for-beginners
2. http://www.journaldev.com/1877/java-servlet-tutorial-with-examples-for-beginners
3. http://www.journaldev.com/1933/java-servlet-filter-example-tutorial
4. http://www.journaldev.com/1945/servlet-listener-example-servletcontextlistener-httpsessionlistener-and-servletrequestlistener