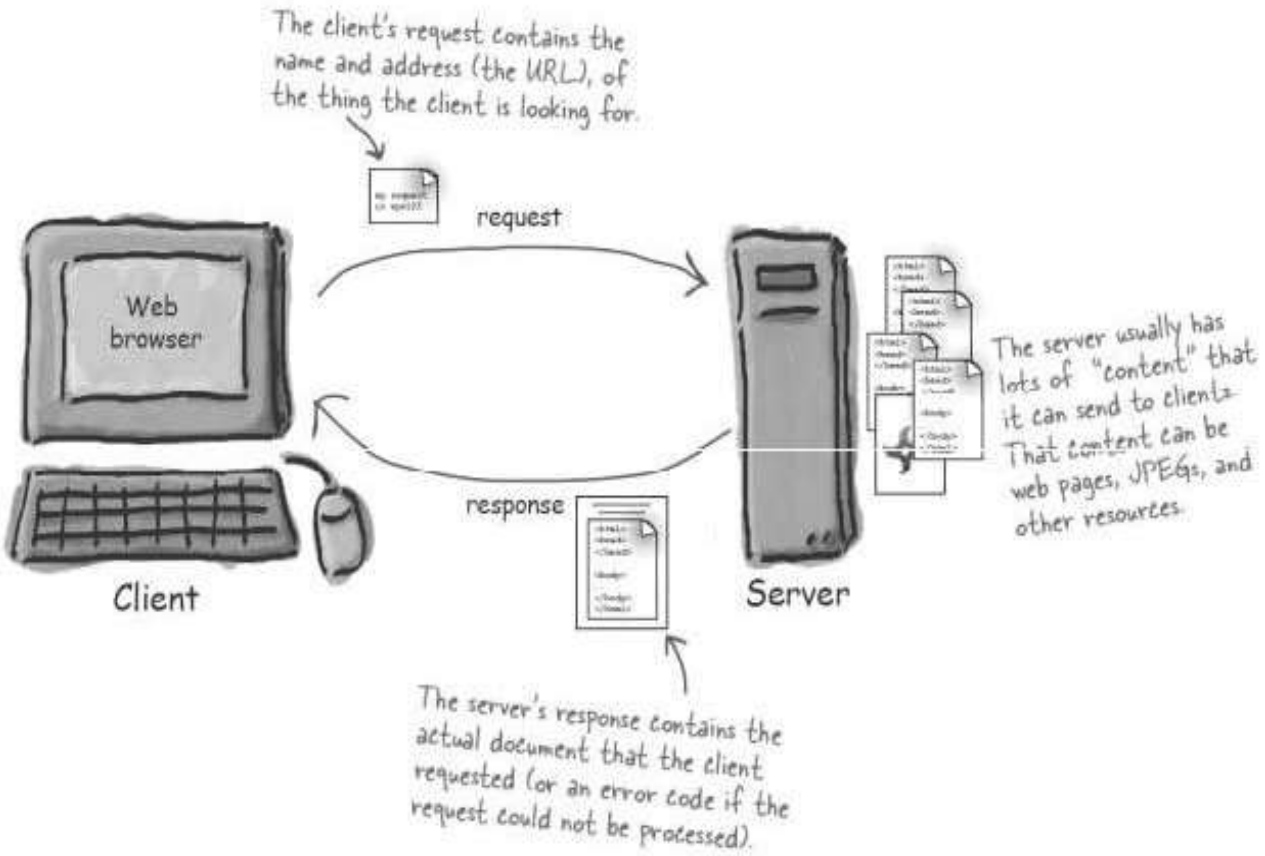
Web Application

A web application runs over the Internet. Examples of webapps are google, amazon, ebay, facebook and twitter.

Web consists of billions of clients and server connected through wires and wireless networks. The web clients make requests to web server. The web server receives the request, finds the resources and return the response to the client. When a server answers a request, it usually sends some type of content to the client. The client uses web browser to send request to the server. The server often sends response to the browser with a set of instructions written in HTML (Hypertext Markup Language). All browsers know how to display HTML page to the client



A Web Application is typically a 3-tier (or multi-tier) client-server database application run over the Internet as illustrated in the diagram below. It comprises five components:

1. **HTTP Server**

   E.g., Apache HTTP Server, Apache Tomcat Server, Microsoft Internet Information Server (IIS), JBOSS, WebLogic, WebSphere, GlassFish, etc.

2. **HTTP Client** (or Web Browser)

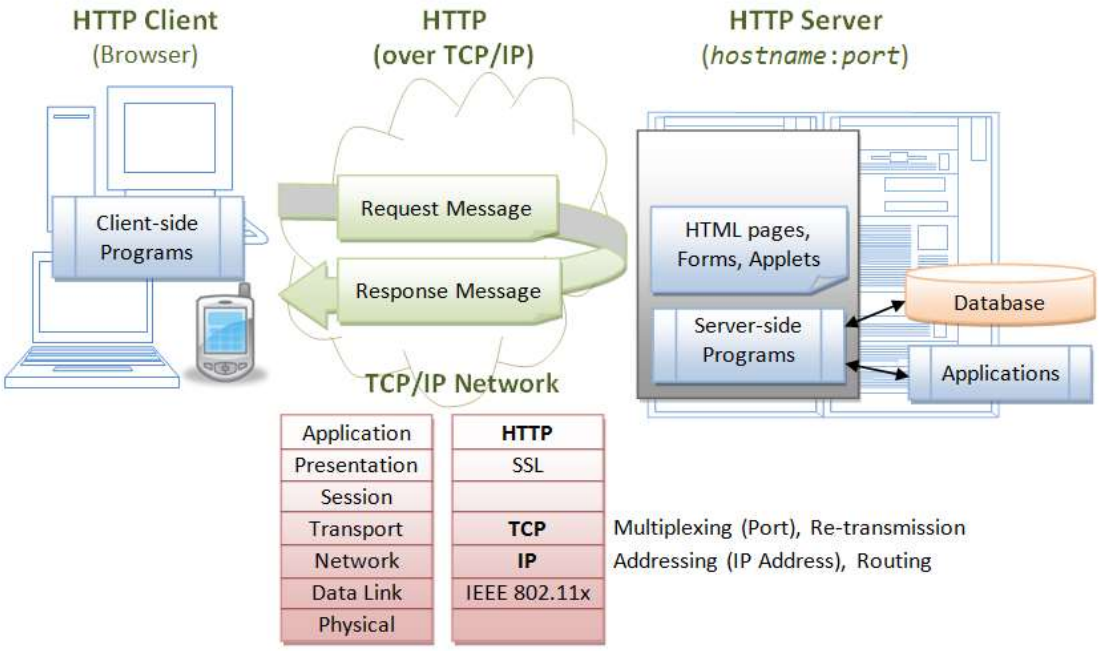
   E.g., Internet Explorer (MSIE), FireFox, Chrome, Safari, and others.

3. **Database**

E.g., Open-source: MySQL, Apache Derby, mSQL, SQLite, PostgreSQL and others.
Commercial: Oracle, IBM DB2, SAP SyBase, MS SQL Server, MS Access; and others.

4. **Client-Side Programs**
   Could be written in HTML, CSS, JavaScript, VBScript, Flash, and others

5. **Server-Side Programs**
   Could be written in Java Servlet/JSP, ASP, PHP, Perl, Python, CGI, NodeJS and others

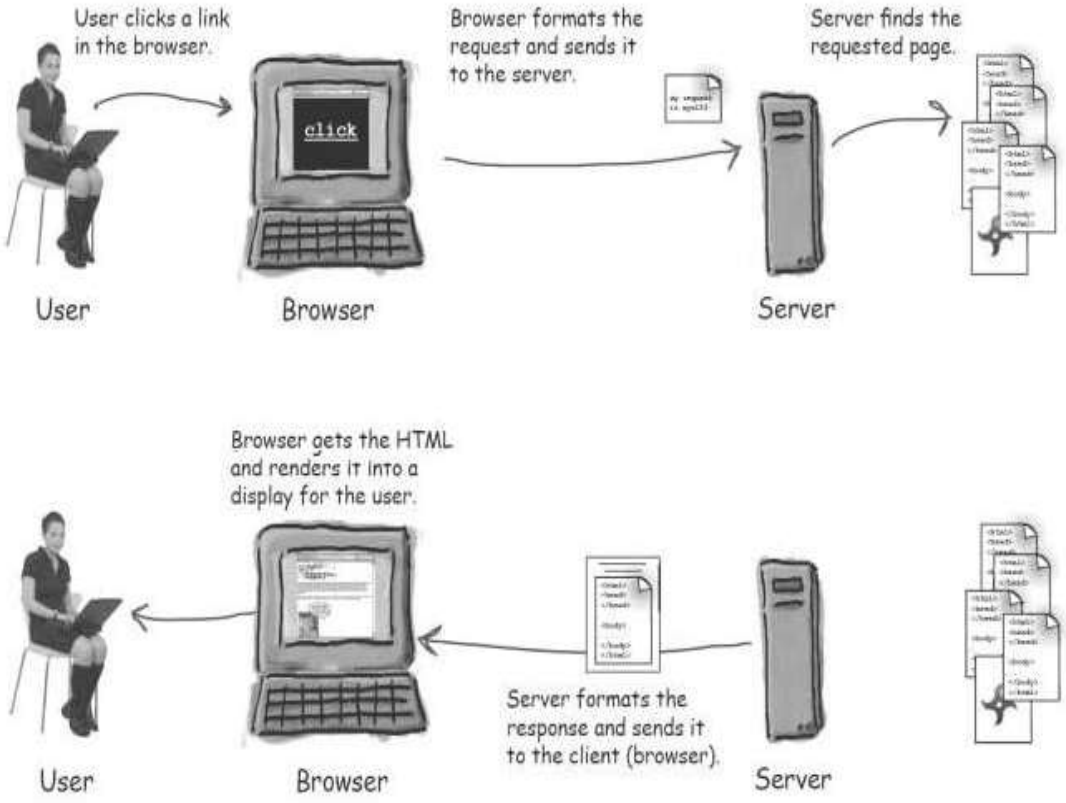


The typical use-case is:

1. A user, via a web browser (HTTP client), issues a URL request to an HTTP server to start a webapp.
2. The HTTP server returns an HTML form (client-side program), which is loaded into the client's browser.
3. The user fills up the query criteria inside the form and submits the form.
4. The client-side program sends the query parameters to a server-side program.
5. The server-side program receives the query parameters, queries the database based on these parameters, and returns the query result to the client-side program.
6. The client-side program displays the query result on the browser.
7. The process repeats for the next request.

What does a web client do?

- A web client lets the user request something on the server, and shows the user the result of the request
- When we talk about clients, we usually mean both (or either) the human user and the browser application
- The browser is a software (like internet Explorer, Mozilla, Chrome etc.) that knows how to communicate with the server
- Browsers interpret the HTML code and render the web page for the user



Server-Side Technologies

The server is a powerful computer that runs the back-end software, the database houses your site's data, and the software communicates between the two. For example, if a user is updating a profile on a networking site, the server-side scripts will gather the information the user enters, the application will process it on the server, then interact with the database to update that information there.

Server-side scripts are used by **back-end web developers** to build the back-end software of a website—the mechanics we don't see, but that make a site's usability and functionality possible. These languages create the communication channel between user, server, and database. Anything that isn't explicitly written into the text markup of a site is front-

end or back-end software. Any data that a user requests in the browser (e.g., the fields in drop-down menus, photos, or user profiles) is delivered via server-side scripts, which create a channel between server and end user that requests, edits, and deletes things in the database. In the browser, front-end scripts make that information available to the user.

Server-Side Script Basics

- **Runs on a server,** embedded in the site's code
- **Interact with back-end permanent storage**, like databases, and process information from the server to access the database—like a direct line from user to database
- **Facilitates the transfer of data** from server to browser, bringing pages to life in the browser, e.g., processing and then delivering a field that a user requests or submits in a form
- **Runs on-call.** When a webpage is "called up," or when parts of pages are "posted back" to the server with **AJAX**, server-side scripts process and return data
- **Powers functions in dynamic web applications,** such as user validation, saving and retrieving data, and navigating between other pages
- Plays a big role in how a database is built from the ground up and managed afterwards—an example of how roles often overlap in all aspects of development
- Build **application programming interfaces (APIs)**, which control what data and software a site shares with other apps

There are many server-side technologies available

- CGI Script
- ASP.NET
- PHP
- Python (Pyramid, Flask, Django)
- NodeJS
- **Java-based (Servlet, JSP, JSF, Struts, Spring, Hibernate)**
- and many others

Java **Servlet** is the foundation of the Java server-side technology, JSP (JavaServer Pages), JSF (JavaServer Faces), Struts, Spring, Hibernate, and others, are extensions of the servlet technology.

CGI

The common gateway interface (CGI) is a standard way for a Web server to pass a Web user's request to an application program and to receive data back to forward to the user. When the user requests a Web page (for example, by clicking on a highlighted word or entering a Web site address), the server sends back the requested page. However, when a user fills out a form on a Web page and sends it in, it usually needs to be processed by an application program. The Web server typically passes the form information to a small application program that processes the data and may send back a confirmation message. This method or convention for passing data back and forth between the server and the application is called the common gateway interface (CGI). It is part of the Web's Hypertext Transfer Protocol (HTTP).

PHP

The most popular server-side language on the web, PHP is designed to pull and edit information in the database. It's most commonly bundled with databases written in the SQL language. PHP was designed strictly for the web and remains one of the most widely used languages around. It's easy to install and deploy, is staying competitive with lots of modern frameworks, and is the foundation for many content-management systems. PHP-powered sites: WordPress, Wikipedia, Facebook

Frameworks: Laravel, Symfony, CodeIgniter, Phalcon, CakePHP, Zend Framework

Python

With fewer lines of code, the Python programming language is fast, making it ideal for getting things to market quickly. The emphasis is on readability and simplicity, which makes it great for beginners. It's the oldest of the scripting languages, is powerful, and works well in **object-oriented** designs. Python-powered sites: YouTube, Google, The Washington Post

Frameworks: Django, Flask, Pyramid, Tornado, Bottle, Diesel, Pecan, Falcon

Ruby

If you're expecting complicated logic on the database side of your site, the Ruby programming language is an excellent option. Unlike Python, Ruby is equal parts simplicity and complexity, pairing simple code with more flexibility and extra tools. Ruby bundles the back end with database functionality that PHP and SQL can offer as a pair—it's great for startups, easy maintenance, and high-traffic demands. It requires developers to use the Ruby on Rails framework, which has vast libraries of code to streamline back-end development. Ruby-powered sites: Hulu, Twitter (originally), Living Social, Basecamp

Frameworks: Ruby on Rails, Rack,

C#

The language of **Microsoft's .NET Framework**—the most popular framework on the web—C# combines productivity and versatility by blending the best aspects of the C and C++ languages. It's excellent for developing Windows applications, and can be used to build iOS, Android mobile apps with the help of a cross-platform technology like Xamarin.

Frameworks: ASP.NET

Java

A subset of the C language, Java comes with a huge ecosystem of add-on software components. At its core, Java is a variation of C++ with an easier learning curve, plus, it's platform independent thanks to the Java Virtual Machine. "Compile once, run anywhere" is its motto—and it's excellent for enterprise-level applications, high-traffic sites, and Android apps. Java sites: Twitter, Verizon, AT&T, Salesforce

Frameworks: Spring, Hibernate, EJB, Struts

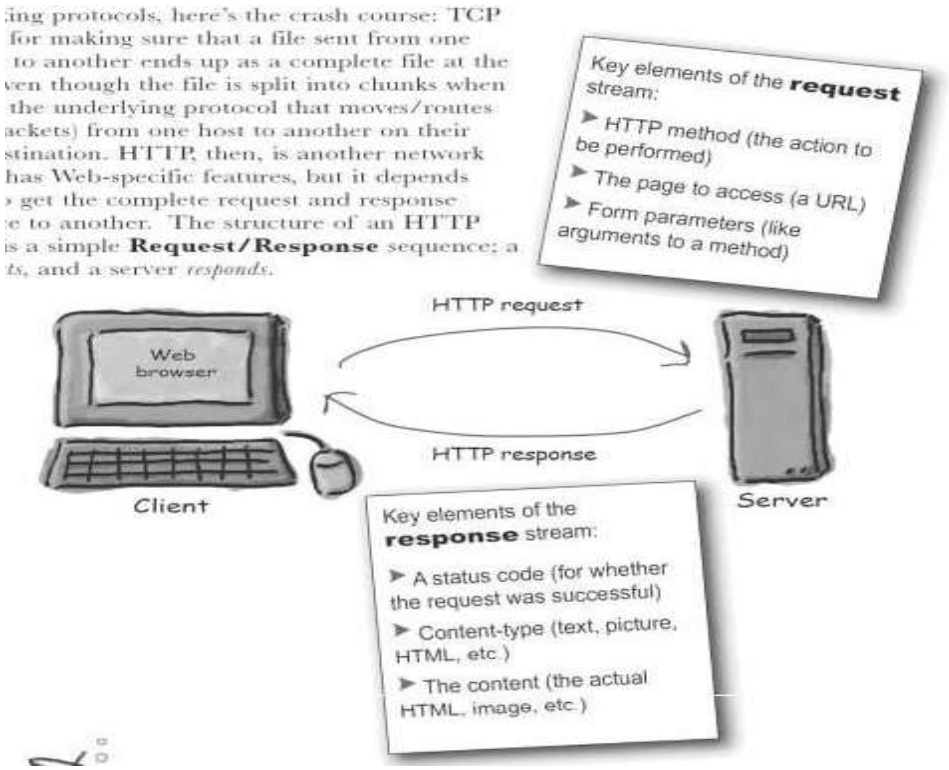
JavaScript

JavaScript is typically a front-end script, but with the Node.js framework, it can be used in server-side technology, from APIs to entire stacks. Its core selling point is how it handles client-server communication—it's fast, doesn't bottleneck, and is ideal for real-time apps like chat rooms, data-heavy applications, and any software that requires the streaming of fresh content, like a news feed. Node.js sites: Dow Jones, PayPal, LinkedIn

Frameworks: NodeJS, ExpressJS

Hypertext Transfer Protocol (HTTP)

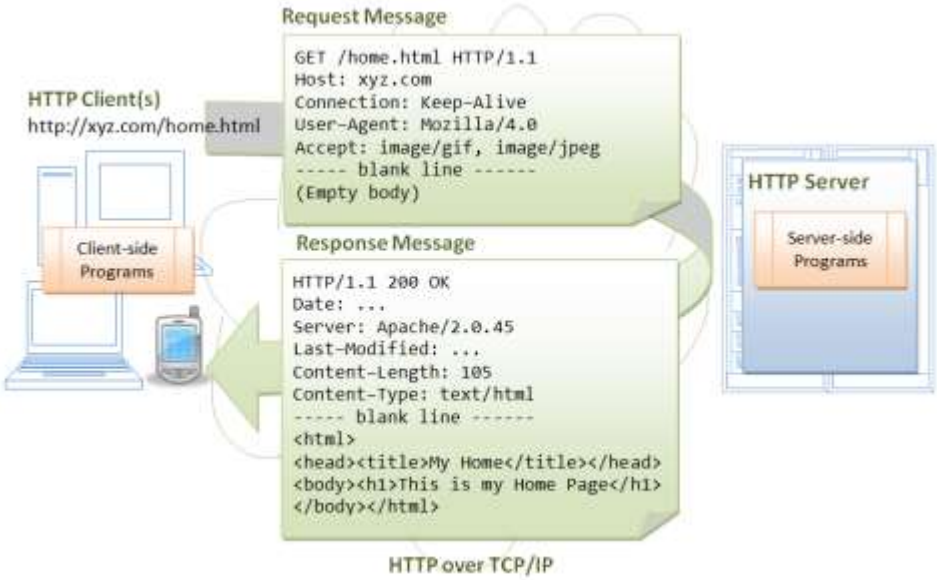
Most of the conversation held on the web between clients and servers are held using the HTTP protocol, which allows for simple request and response conversation. The client sends an HTTP request, and server answers with HTTP Response



HTTP is an application layer protocol runs over TCP/IP. The IP provides support for routing and addressing (via an unique IP address for machines on the Internet); while TCP supports multiplexing via 64K ports from port number 0 to 65535. The default port number assigned to HTTP is TCP port 80.

HTTP is an asynchronous request-response application-layer protocol. A client sends a request message to the server. The server then returns a response message to the client. In other words, HTTP is a pull protocol, a client pulls a page from the server (instead of server pushes pages to the clients).

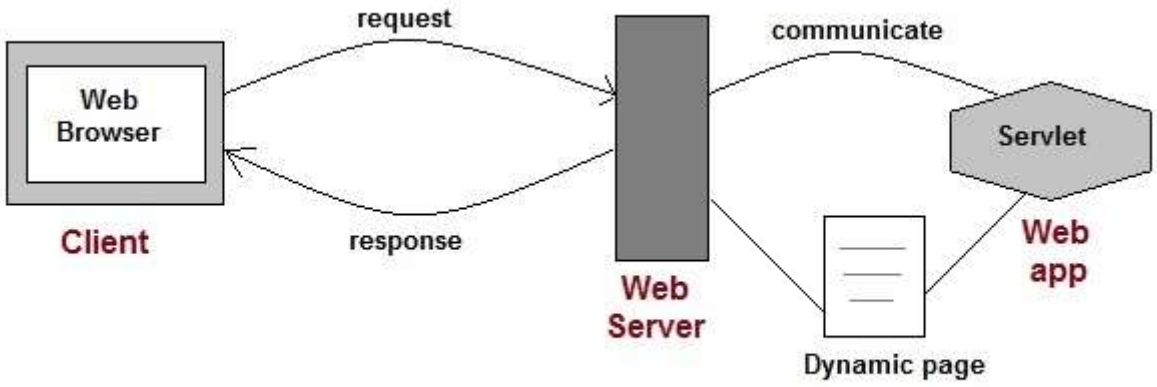
The syntax of the message is defined in the HTTP specification.



Servlet – Introduction

Servlet Technology is used to create web applications. **Servlet** technology uses Java language to create web applications.

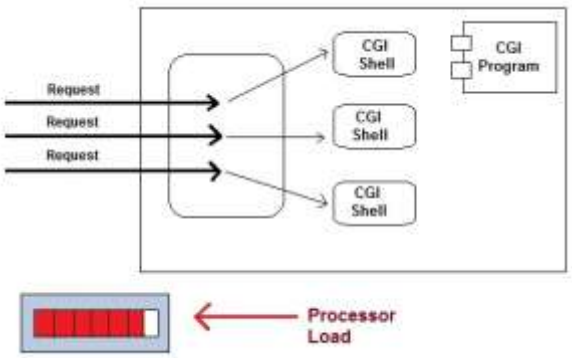
Web applications are helper applications that resides at web server and build dynamic web pages. A dynamic page could be anything like a page that randomly chooses picture to display or even a page that displays the current time.



CGI (Common Gateway Interface)

Before Servlets, CGI(Common Gateway Interface) programming was used to create web applications. Here's how a CGI program works :

- User clicks a link that has URL to a dynamic page instead of a static page.
- The URL decides which CGI program to execute.
- Web Servers run the CGI program in seperate OS shell. The shell includes OS enviroment and the process to execute code of the CGI program.
- The CGI response is sent back to the Web Server, which wraps the response in an HTTP response and send it back to the web browser.



Drawbacks of CGI programs

- High response time because CGI programs execute in their own OS shell.
- CGI is not scalable.
- CGI programs are not always secure or object-oriented.
- It is Platform dependent.

Because of these disadvantages, developers started looking for better CGI solutions. And then Sun Microsystems developed **Servlet** as a solution over traditional CGI technology.

CGI vs Servlet

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

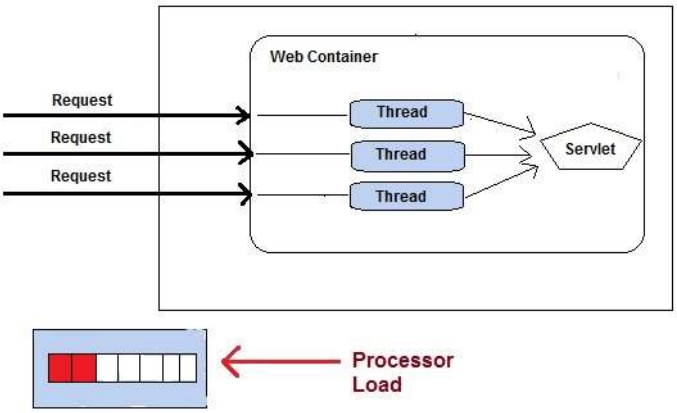
- Servlets provide better performance that CGI in terms of processing time, memory utilization because servlets uses benefits of multithreading and for each request a new thread is created, that is faster than loading creating new Object for each request with CGI.
- Servlets and platform and system independent,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

Servlet

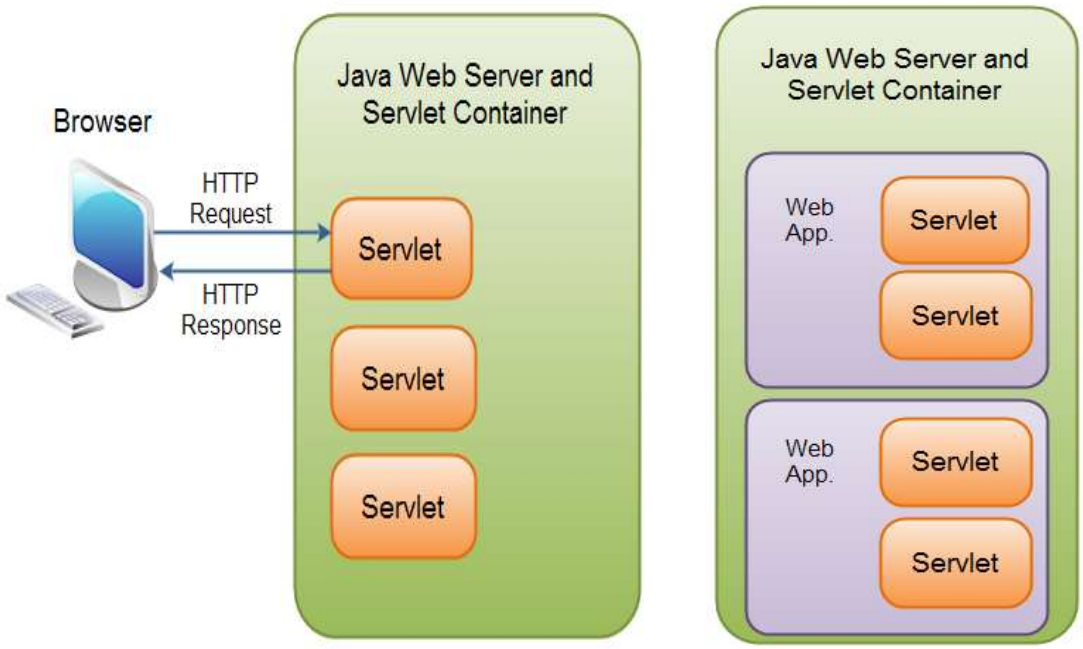
Java servlets are server-side programs (running inside a web server) that handle clients' requests and return a customized or dynamic response for each request. The dynamic response could be based on user's input (e.g., search, online shopping, online transaction) with data retrieved from databases or other applications, or time-sensitive data (such as news and stock prices).

Java servlets typically run on the HTTP protocol. HTTP is an asymmetrical request-response protocol. The client sends a request message to the server, and the server returns a response message.

Java Servlets are part of the Java Enterprise Edition (Java EE). You will need to run your Java Servlets inside a Servlet compatible "Servlet Container" (e.g. web server) for them to work.



A Java Servlet is a Java object that responds to HTTP requests. It runs inside a Servlet container. A Servlet container may run multiple web applications at the same time, each having multiple servlets running inside. A Java web application can contain other components than servlets. It can also contain Java Server Pages (JSP), Java Server Faces (JSF) and Web Services



Servlet Container / Web Container

Java servlet containers are usually running inside a Java web server. It is integrated set of objects that provide runtime environment for servlet. A web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access-rights.

A web container handles requests to servlets, JavaServer Pages (JSP) files, and other types of files that include server-side code. The Web container creates servlet instances, loads and unloads servlets, creates and manages request and response objects, and performs other servlet-management tasks.

The services provided by servlet containers are :

- Network Services
- Loading servlet class
- Decoding and encoding of MIME messages (Multiple Internet Mail Extension)
- Manage life cycle of a servlet
- Manage resources
- Security services
- Session management

The servlet container providers are Apache Software, IBM, Oracle, They provided Web Servers or Servlet engines such as Tomcat, Web Sphere, Resin, Web Logic, GlassFish etc.

Open source Web containers

- **Apache Tomcat** is an open source web container available under the Apache Software License.
- **GlassFish** from Oracle (an Application Server, but includes a web container).
- **JBoss Application Server** (now WildFly) is a full Java EE implementation by Red Hat Inc., division JBoss.

Commercial Web containers

- **WebLogic** Application Server, from Oracle Corporation (formerly developed by BEA Systems)
- **Resin** Pro, from Caucho Technology
- IBM **WebSphere** Application Server

Web Server Vs Application Server

Web Server

Web Server is a computer where the web content is stored. Basically web server is used to host the web sites but there exists other web servers also such as gaming, storage, FTP, email etc.

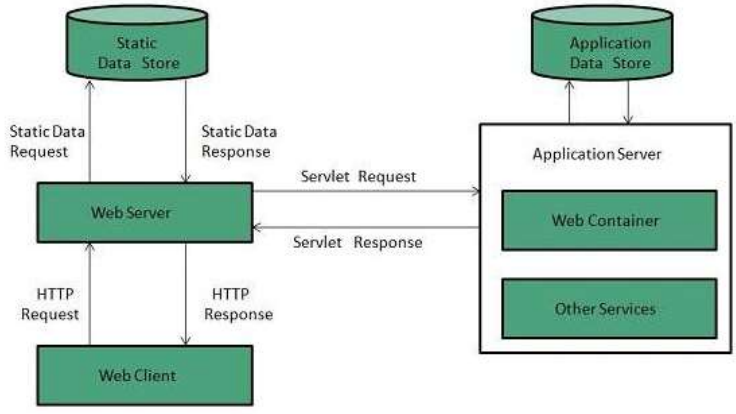
All computers that host Web sites must have Web server programs. Leading Web servers include **Apache** (the most widely-installed Web server), Microsoft's **Internet Information Server** (**IIS**) and **nginx** (pronounced engine X) from NGNIX. Other Web servers include Novell's **NetWare server**, **Google Web Server** (GWS) and IBM's family of **Domino** servers.

Web Server Working

It can respond to the client request in either of the following two possible ways:

- Generating response by using the script and communicating with database.
- Sending file to the client associated with the requested URL.

The block diagram representation of Web Server is shown below:



Important points

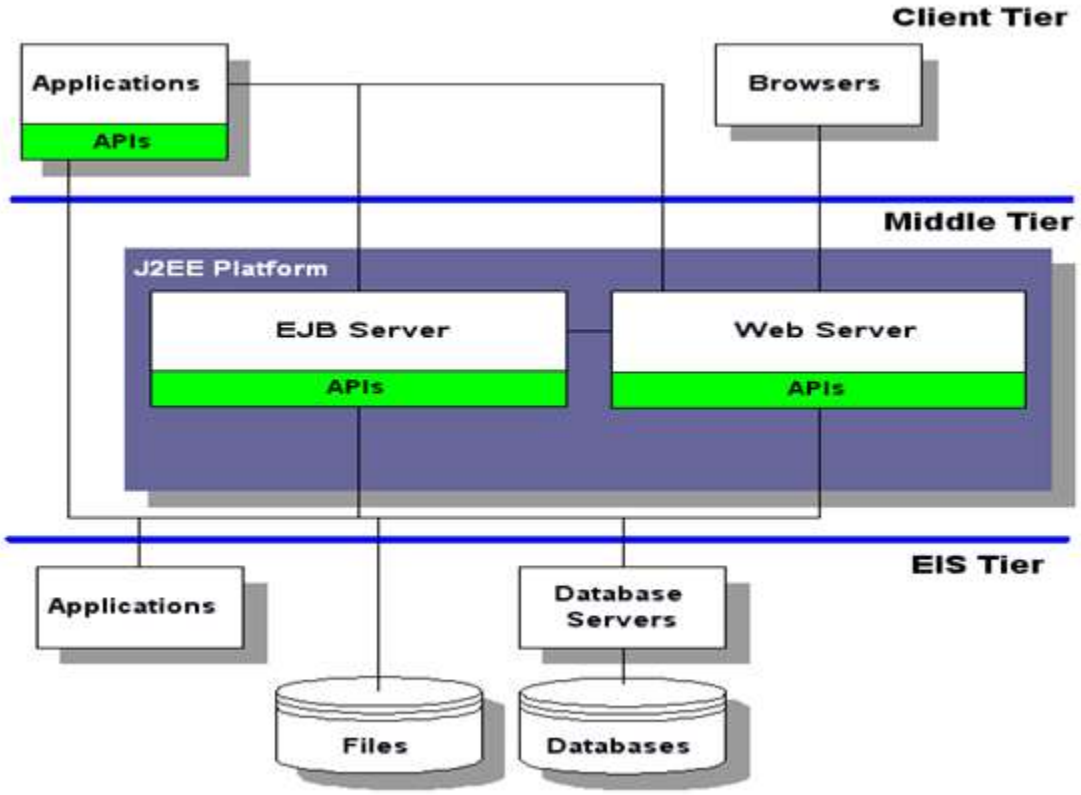
- If the requested web page at the client side is not found, then web server will send the HTTP response: Error 404 Not found.
- When the web server searching the requested page if requested page is found then it will send to the client with an HTTP response.
- If the client requests some other resources, then web server will contact to application server and data is store for constructing the HTTP response.

Application Server

Application server contains Web and EJB containers. It can be used for servlet, jsp, struts, jsf, ejb etc. It is a component-based product that lies in the middle-tier of a server centric architecture.

It provides the middleware services for state maintenance and security, along with persistence and data access. It is a type of server designed to install, operate and host associated services and applications for the IT services, end users and organizations.

The block diagram representation of Application Server is shown below:



The Example of Application Servers are:

- **JBoss:** Open-source server from JBoss community.
- **Glassfish:** Provided by Sun Microsystem. Now acquired by Oracle.
- **Weblogic:** Provided by Oracle. It more secured.
- **Websphere:** Provided by IBM.

Content Type

Content Type is also known as MIME (Multipurpose internet Mail Extension) Type. It is a **HTTP header** that provides the description about what are you sending to the browser.

MIME is an internet standard that is used for extending the limited capabilities of email by allowing the insertion of sounds, images and text in a message.

The features provided by MIME to the email services are as given below:

- It supports the non-ASCII characters
- It supports the multiple attachments in a single message
- It supports the attachment which contains executable audio, images and video files etc.
- It supports the unlimited message length.

List of Content Types

There are many content types. The commonly used content types are given below:

- text/html
- text/plain
- application/msword
- application/vnd.ms-excel
- application/jar
- application/pdf
- application/octet-stream
- application/x-zip
- images/jpeg
- images/png
- images/gif
- audio/mp3
- video/mp4
- video/quicktime etc.

HTTP Request and Response

The browser sends an HTTP request to the Java web server. The web server checks if the request is for a servlet. If it is, the servlet container is passed the request. The servlet container will then find out which servlet the request is for, and activate that servlet. The servlet is activated by calling the Servlet.service() method.

Once the servlet has been activated via the service() method, the servlet processes the request, and generates a response. The response is then sent back to the browser.

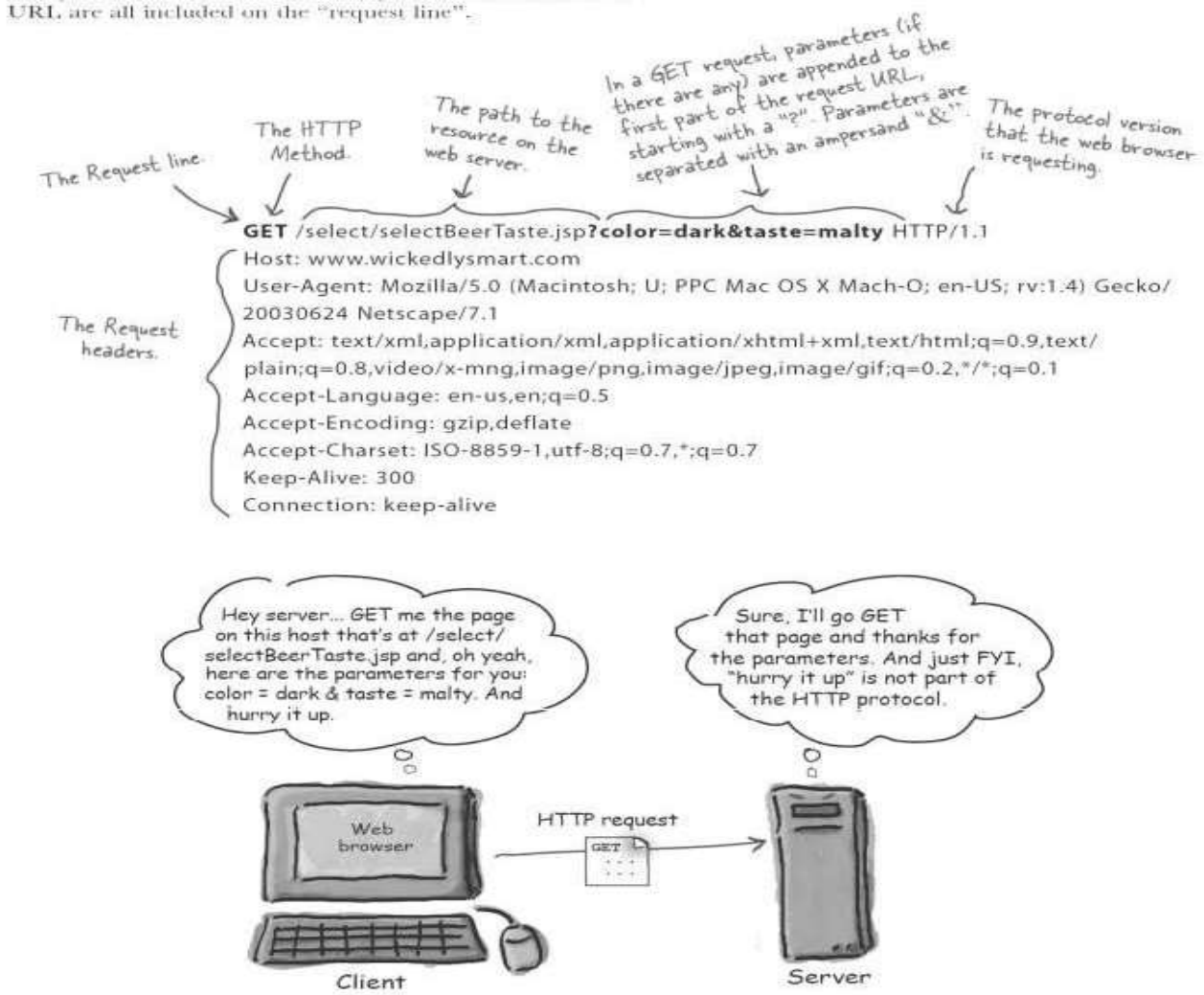
Http Request Methods

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs. Each of them implements a different semantic, but some common features are shared by a group of them: e.g. a request method can be safe, idempotent, or cacheable.

GET

The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data

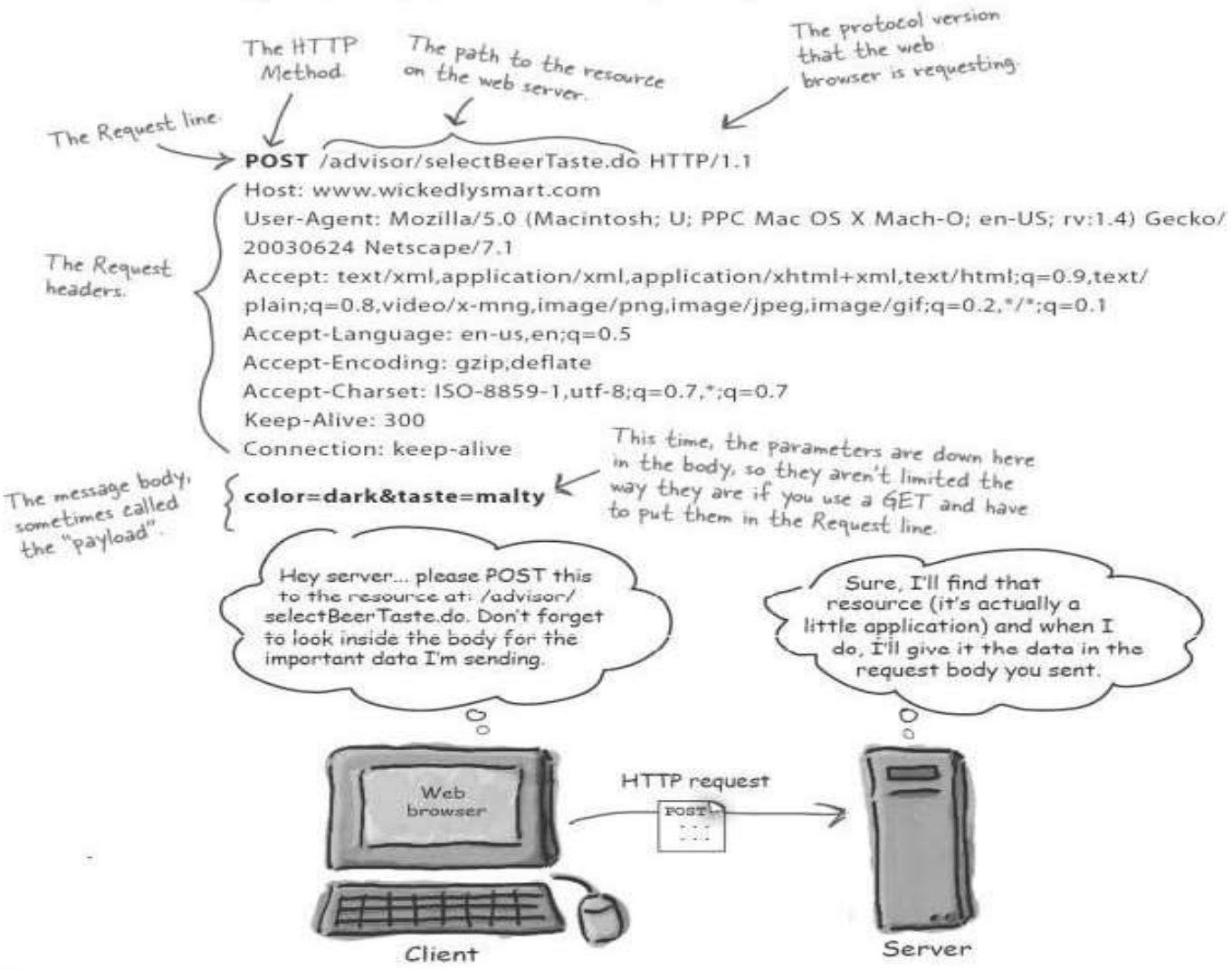
Anatomy of Get Request



POST

POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.

Anatomy of Post Request



HEAD

The HEAD method is functionally like GET, except that the server replies with a response line and headers, but no entity-body. The following example makes use of HEAD method to fetch header information about hello.htm

PUT

The PUT method replaces all current representations of the target resource with the request payload.

DELETE

The DELETE method deletes the specified resource.

CONNECT

The CONNECT method establishes a tunnel to the server identified by the target resource.

OPTIONS

The OPTIONS method is used to describe the communication options for the target resource.

TRACE

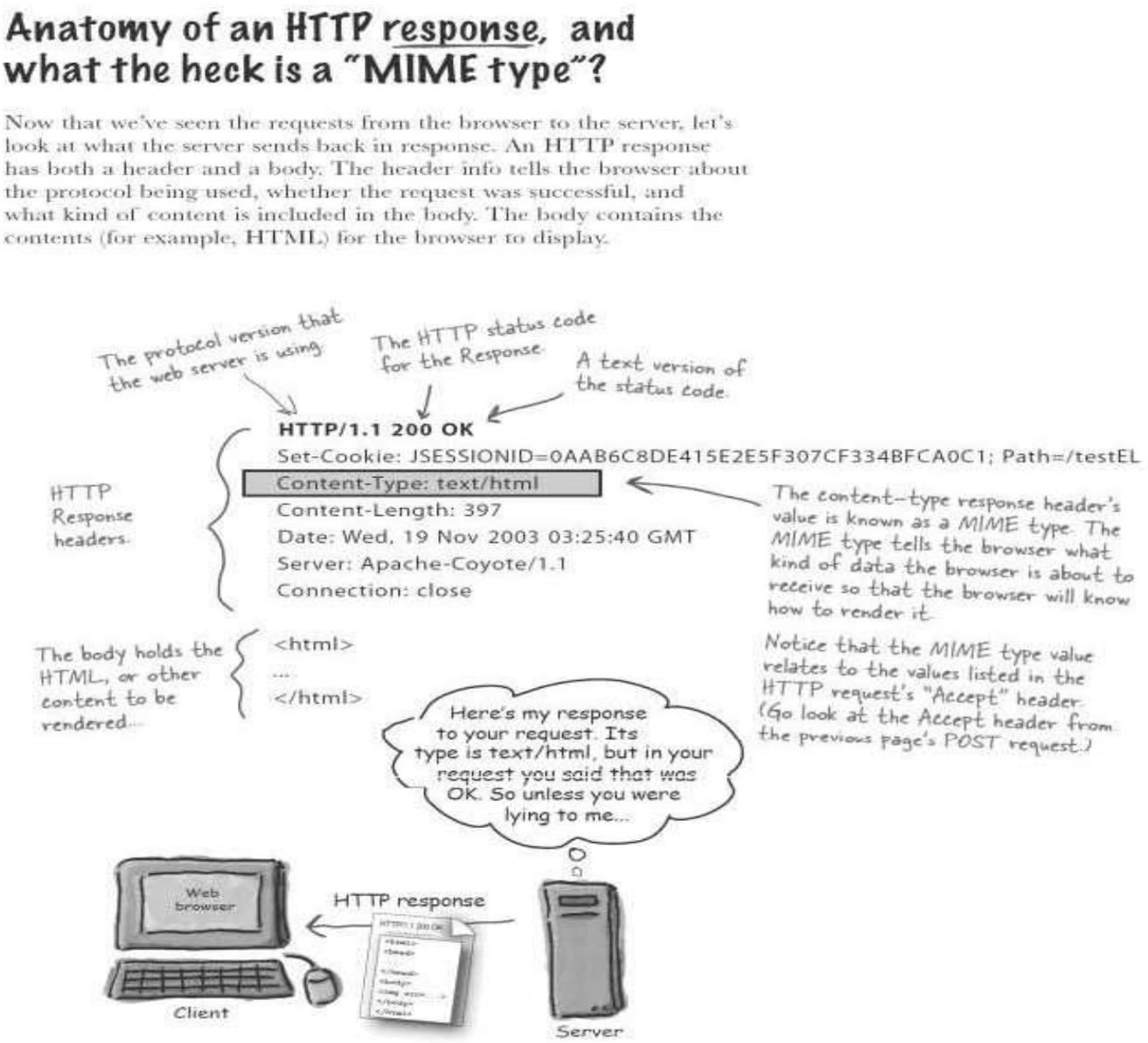
The TRACE method performs a message loop-back test along the path to the target resource.

PATCH

The PATCH method is used to apply partial modifications to a resource.

Http Response

After receiving and interpreting a request message, a server responds with an HTTP response message



Message Status-Line

A Status-Line consists of the protocol version followed by a numeric status code and its associated textual phrase. The elements are separated by space SP characters.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

HTTP Version

A server supporting HTTP version 1.1 will return the following version information:

```
HTTP-Version = HTTP/1.1
```

Status Code

The Status-Code element is a 3-digit integer where first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role. There are 5 values for the first digit:

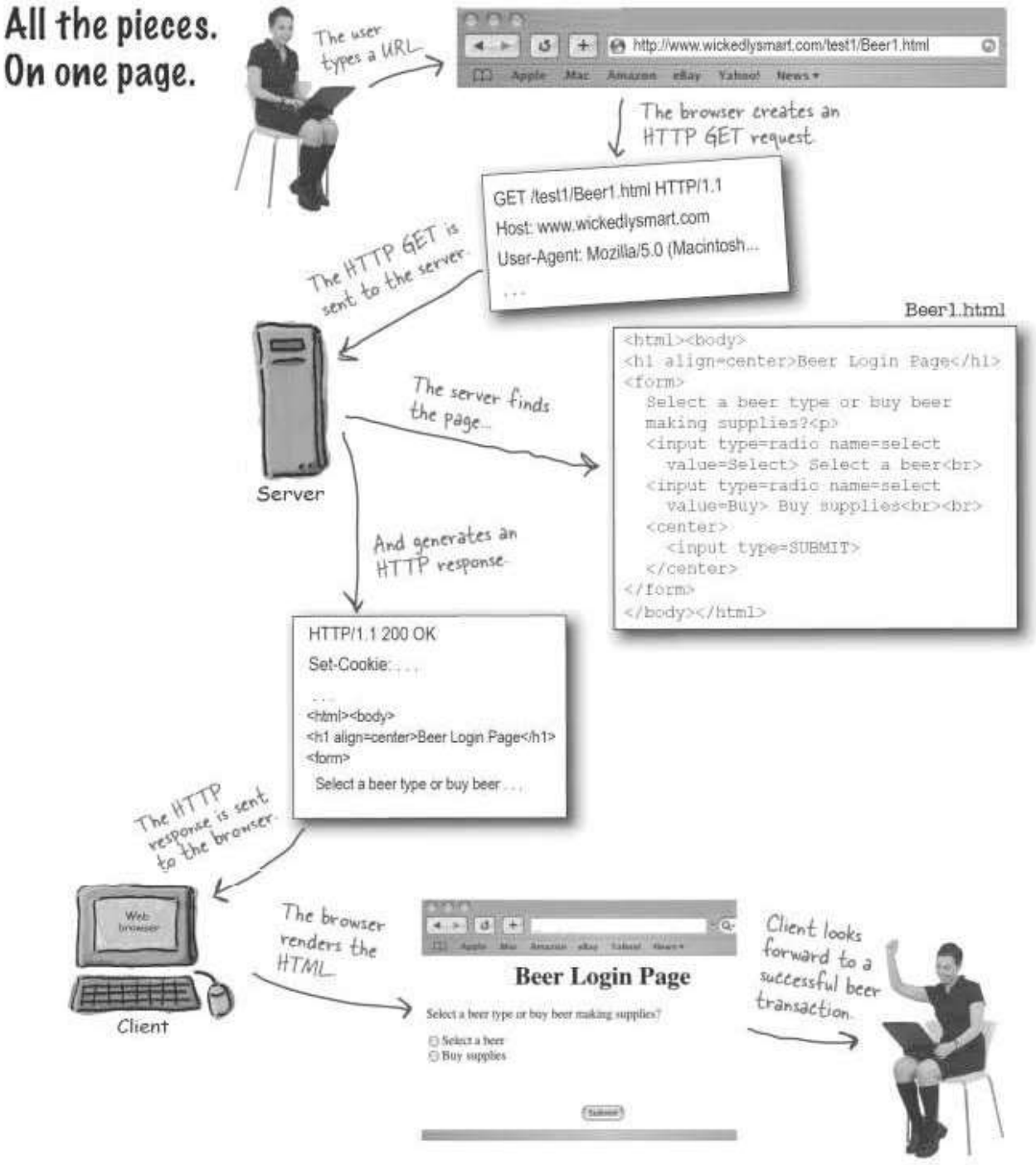
S.N.	Code and Description
1	1xx: Informational It means the request was received and the process is continuing.
2	2xx: Success It means the action was successfully received, understood, and accepted.
3	3xx: Redirection It means further action must be taken in order to complete the request.
4	4xx: Client Error It means the request contains incorrect syntax or cannot be fulfilled.
5	5xx: Server Error It means the server failed to fulfill an apparently valid request.

HTTP status codes are extensible and HTTP applications are not required to understand the meaning of all registered status codes. A list of all the status codes has been given in a separate chapter for your reference.

Response Header Fields

We will study General-header and Entity-header in a separate chapter when we will learn HTTP header fields. For now, let's check what Response header fields are.

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status- Line. These header fields give information about the server and about further access to the resource identified by the Request-URI



Apache Tomcat HTTP Server

Apache Tomcat is a Java-capable HTTP server, which could execute special Java programs known as "Java Servlet" and "Java Server Pages (JSP)". Tomcat is an open-source project, under the "Apache Software Foundation.

Tomcat was originally written by James Duncan Davison (then working in Sun), in 1998, based on an earlier Sun's server called Java Web Server (JWS). Sun subsequently made Tomcat open-source and gave it to Apache.

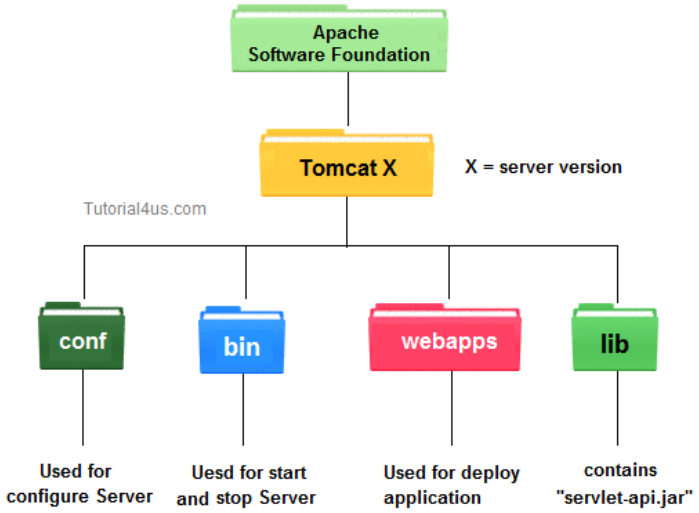
Tomcat is an HTTP application runs over TCP/IP. In other words, the Tomcat server runs on a specific TCP port from a specific IP address. The default TCP port number for HTTP protocol is 80, which is used for the production HTTP server. For test HTTP server, you can choose any unused port number between 1024 and 65535.

Install Tomcat 8

Step 1: Install Tomcat on Windows

1. Go to http://tomcat.apache.org ⇒ Under "Tomcat 8.5.{xx} Released" ⇒ Downloads ⇒ Under "8.5.{xx}" ⇒ Binary Distributions ⇒ Core ⇒ "ZIP" package
2. Create your project directory, say "d:\myProject" or "c:\myProject". UNZIP the downloaded file into your project directory. Tomcat will be unzipped into directory "d:\myProject\apache-tomcat-8.0.{xx}".

Tomcat's Directories



- **Bin**
  contains the binaries; and startup script (startup.bat for Windows and startup.sh for Unixes and Mac OS), shutdown script (shutdown.bat for Windows and shutdown.sh for Unix and Mac OS), and other binaries and scripts.

- **Conf**
  contains the system-wide configuration files, such as server.xml, web.xml, context.xml, and tomcat-users.xml.

- **Lib**
  contains the Tomcat's system-wide JAR files, accessible by all webapps. You could also place external JAR file (such as MySQL JDBC Driver) here.

- **Logs**
  contains Tomcat's log files. You may need to check for error messages here.

- **Webapps**
  contains the webapps to be deployed. You can also place the WAR (Webapp Archive) file for deployment here.

- **Work**
  Tomcat's working directory used by JSP, for JSP-to-Servlet conversion.

- **Temp**
  Temporary files

STEP 2: Create an Environment Variable JAVA_HOME

1. First, find your JDK installed directory. The default is "`c:\Program Files\Java\jdk1.8.0_{xx}`", where {*xx*} is the upgrade number. Take note of your JDK installed directory.
2. To set the environment variable `JAVA_HOME` in Windows 7/8/10: Start "Control Panel" ⇒ System and Security (Optional) ⇒ System ⇒ Advanced system settings ⇒ Switch to "Advanced" tab ⇒ Environment Variables ⇒ System Variables ⇒ "New" ⇒ In "Variable Name", enter "`JAVA_HOME`" ⇒ In "Variable Value", enter your JDK installed directory

STEP 3: Configure Tomcat Server

The Tomcat configuration files are located in the "conf" sub-directory of your Tomcat installed directory, e.g. "d:\myProject\tomcat\conf" (for Windows) or "/Applications/tomcat/conf" (for Mac OS). There are 4 configuration XML files:

- server.xml
- web.xml
- context.xml
- tomcat-users.xml

Set the TCP Port Number (`server.xml`)

The default TCP port number configured in Tomcat is 8080, you may choose any number between 1024 and 65535, which is not used by an existing application.

Locate the following lines that define the HTTP connector, and change port="8080" to port="8181".

```
<!-- A "Connector" represents an endpoint by which requests are received
      and responses are returned. Documentation at :
      Java HTTP Connector: /docs/config/http.html (blocking & non-blocking)
      Java AJP  Connector: /docs/config/ajp.html
      APR (HTTP/AJP) Connector: /docs/apr.html
      Define a non-SSL HTTP/1.1 Connector on port 8080
-->
<Connector port="8181" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

Enabling Directory Listing - `web.xml`

We shall enable directory listing by changing "listings" from "false" to "true" for the "default" servlet. This is handy for test system, but not for production system for security reasons.

Locate the following lines (around Line 103) that define the "default" servlet; and change the "listings" from "false" to "true".

```
<!-- The default servlet for all web applications, that serves static     -->
<!-- resources.  It processes all requests that are not mapped to other   -->
<!-- servlets with servlet mappings.                                      -->
```

```
<servlet>
  <servlet-name>default</servlet-name>
  <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <init-param>
    <param-name>listings</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Enabling Automatic Reload - `context.xml`

We shall add the attribute `reloadable="true"` to the `<Context>` element to enable automatic reload after code changes. Again, this is handy for test system but not for production, due to the overhead of detecting changes.

Locate the <Context> start element (around Line 19), and change it to <Context reloadable="true">.

```
<Context reloadable="true">
   ......
   ......
</Context>
```

Enable the Tomcat's manager - `tomcat-users.xml`

Enable the Tomcat's manager by adding the highlighted lines, inside the `<tomcat-users>` elements:

```
<tomcat-users>
  <role rolename="manager-gui"/>
  <user username="manager" password="xxxx" roles="manager-gui"/>
</tomcat-users>
```

STEP 4: Start Tomcat Server

The Tomcat's executable programs and scripts are kept in the "bin" sub-directory of the Tomcat installed directory, e.g., "d:\myProject\tomcat\bin" (for Windows)

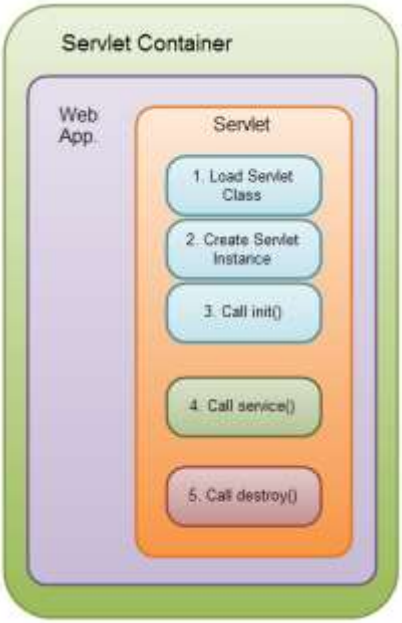
For Windows

Launch a CMD shell. Set the current directory to "`<TOMCAT_HOME>\bin`", and run "`startup.bat`" as follows:

```
// Change the current directory to Tomcat's "bin"
// Assume that Tomcat is installed in "d:\myProject\tomcat"
> d:                              // Change the current drive
> cd \myProject\tomcat\bin      // Change Directory to YOUR Tomcat's "bin" directory
// Start Tomcat Server
> startup
......
......
xxx xx, xxxx x:xx:xx xx org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-9999"]
xxx xx, xxxx x:xx:xx xx org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-bio-8009"]
xxx xx, xxxx x:xx:xx xx org.apache.catalina.startup.Catalina start
INFO: Server startup in 2477 ms
```

Servlet Life Cycle

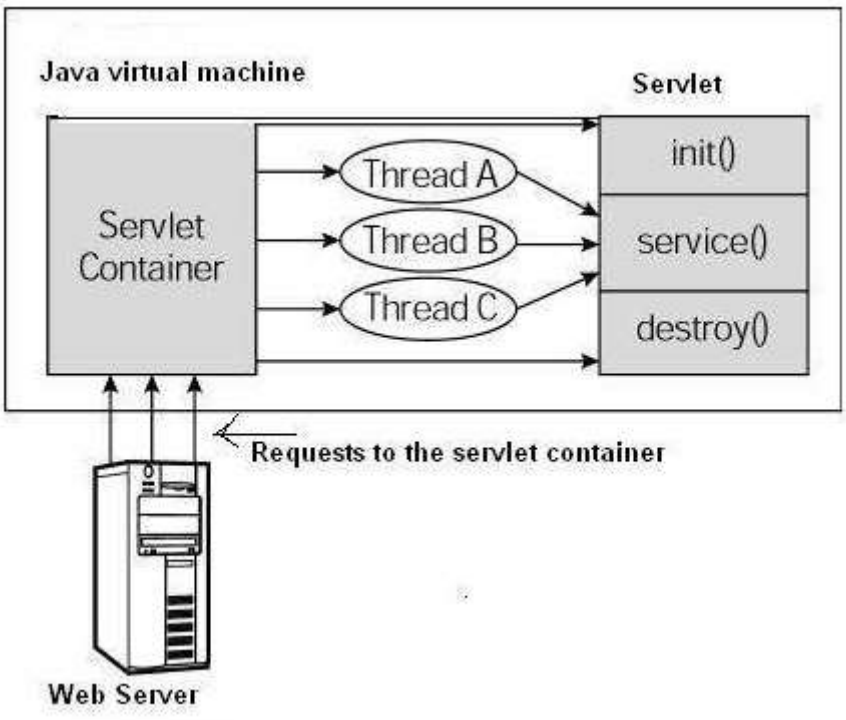
A servlet life cycle can be defined as the entire process from its creation till the destruction.



1. Loading servlet
2. Instantiation (Create Instance of Servlet)
3. Initialization (Call the Servlets init() Method)
4. Request handling (Call the Servlets service () Method)
5. Destroy servlet (Call the Servlets destroy () Method)

Servlet Life Cycle

- First the HTTP requests coming to the server are delegated to the servlet container.
- The servlet container loads the servlet before invoking the service() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.



1. Loading Servlet

Before a servlet can be invoked the servlet, container must first load its class definition. This is done just like any other class is loaded. The servlet container loads the Servlet class, using the normal java class loading options. The servlet class may be loaded from a local/remote file system or from any network service. If the container fails to load servlet class, the process is terminated.

Load Servlet Class -> Create Instance -> call init() -> call service() -> call destroy()

2. Instantiation (Create Instance of Servlet)

When the servlet class is loaded, the servlet container creates an instance of the servlet.

Typically, only a single instance of the servlet is created, and concurrent requests to the servlet are executed on the same servlet instance. This is up to the servlet container to decide, though. But typically, there is just one instance.

To create an instance of Servlet class, the container uses a non-argument(default) constructor

3. Initialization (Call the Servlets init() Method)

```
public void init() throws ServletException {
   // Initialization code...
}
```

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started. When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate.

When a servlet instance is created, its init () method is invoked. The init() method allows a servlet to initialize itself before the first request is processed. The container invokes init() method on servlet instance for initialization, after successful creation of servlet instance.

The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations, just as with the init method of applets.

The init() method accepts ServletConfig object as parameter, so servlet instance is initialized and some configurations are made. Once the initialization is completed successfully, the state of the servlet instance is changed to Active state and then it is made available and placed in service () method for handling client request.

If it fails to initialize, the container throws ServletException or UnavailableExcepton

4. Request handling (Call the Servlets service() Method)

For every request received to the servlet, the servlets service() method is called. For HttpServlet subclasses, one of the doGet(), doPost() etc. methods are typically called. As long as the servlet is active in the servlet container, the service() method can be called. Thus, this step in the life cycle can be executed multiple times.

```
public void service(ServletRequest request, ServletResponse response)
```

```
        throws ServletException, IOException {
    }
```

The service() method is the main method to perform the actual task. Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

To handle client request the container invokes service() method on servlet instance. It create two objects i.e. ServletRequest object and ServletResponse object and they are passed to service() method. In the service() method the request is processed and the response object is sent to Web Container/Web Server

5. Destroy the servlet

When a servlet is unloaded by the servlet container, its destroy() method is called. This step is only executed once, since a servlet is only unloaded once. A servlet is unloaded by the container if the container shuts down, or if the container reloads the whole web application at runtime.

```
public void destroy() {
    // Finalization code...
}
```

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

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

The container invokes destroy() method on Servlet Instance, after completing all the running thread's jobs

Servlet API

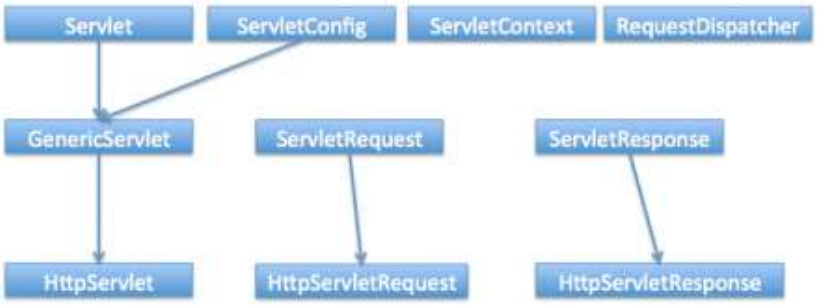
The `javax.servlet` and `javax.servlet.http` packages provide interfaces and classes for writing our own servlets.

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 thats why we mostly extend HttpServlet class.

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.



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.

Servlet interface provides common behavior to all the servlets. Servlet interface needs to be implemented for creating any servlet. It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods

1. public abstract void init(ServletConfig paramServletConfig)
2. public abstract ServletConfig getServletConfig()
3. public abstract void service(ServletRequest req, ServletResponse res)
4. public abstract String getServletInfo()
5. public abstract void destroy()

The methods declared in this interface are:

1. **`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.

2. **`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.

3. **`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**.

4. **`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.

5. **`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.

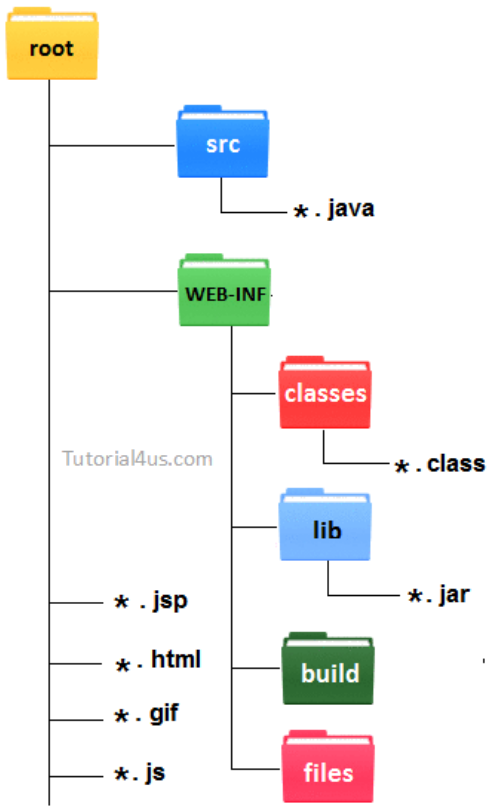
Steps to create Servlet Application

1. Create a directory structure
2. Create a Servlet
3. Compile the Servlet
4. Create a deployment descriptor (web.xml)
5. Start the server (tomcat) and deploy the project
6. Access the servlet

1. Create a directory structures

Servlet program is not like, writing java code and execute through command prompt. We need to follow the following steps to develop any servlets program. Even for a simple HelloWorld program also one must follow this standard directory structure which is prescribed by sun.

The **directory structure** defines that where to put the different types of files so that web container may get the information and respond to the client. The Sun Microsystem defines a unique standard to be followed by all the server vendors.



2. Create a Servlet

There are three ways to create the servlet.

1. By implementing the Servlet interface
2. By inheriting the GenericServlet class
3. By inheriting the HttpServlet class

The HttpServlet class is widely used to create the servlet because it provides methods to handle http requests such as doGet(), doPost, doHead() etc.

3. Compile the servlet

For compiling the Servlet, jar file is required to be loaded. For Tomcat, Set **servlet-api.jar** (available in your tomcat/lib folder) in your class path

Different Servers provide different jar files:

Jar file	Server
servlet-api.jar	Apache Tomcat
weblogic.jar	Weblogic
javaee.jar	Glassfish
javaee.jar	JBoss

4. Create the deployment descriptor (web.xml file)

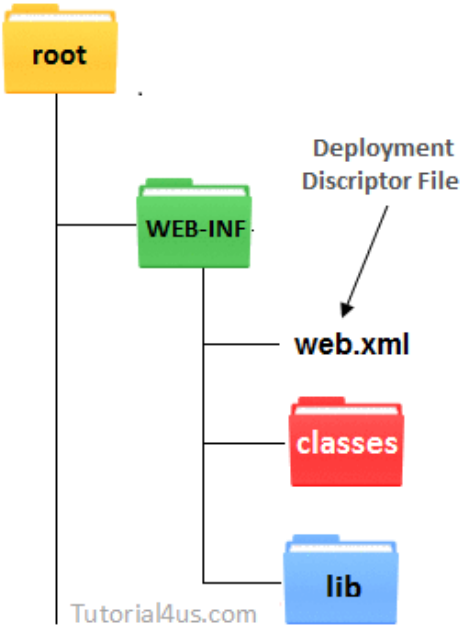
The **deployment descriptor** is an xml file, from which Web Container gets the information about the servlet to be invoked

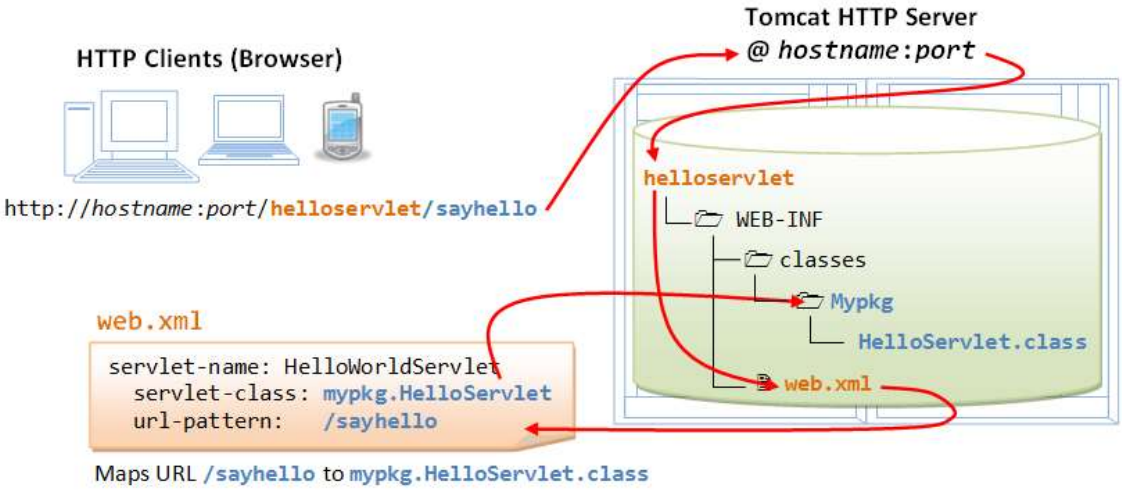
A web user invokes a servlet, which is kept in the web server, by issuing a specific URL from the browser. In this example, we shall configure the following request URL to trigger the "`HelloServlet`":

The "web.xml" is called web application deployment descriptor. It provides the configuration options for that web application, such as defining the mapping between URL and servlet class, contains detail description about web application like configuration of Servlet, Session management, Startup parameters, Welcome file..etc.

We can not change the directory or extension name of this **web.xml** because it is standard name to recognized by container at run-time.

web.xml is present inside the Web-INF folder.





- **<web-app>** represents the whole application.
- **<servlet>** is sub element of <web-app> and represents the servlet.
- **<servlet-name>** is sub element of <servlet> represents the name of the servlet.
- **<servlet-class>** is sub element of <servlet> represents the class of the servlet.
- **<servlet-mapping>** is sub element of <web-app>. It is used to map the servlet.
- **<url-pattern>** is sub element of <servlet-mapping>. This pattern is used at client side to invoke the servlet.

5. Start the Server and deploy the project

To start Apache Tomcat server, double click on the startup.bat file under apache-tomcat/bin directory

6. Access the servlet

Open browser and write http://hostname:portno/contextroot/urlpatternofservlet

First "Hello-world" Servlet

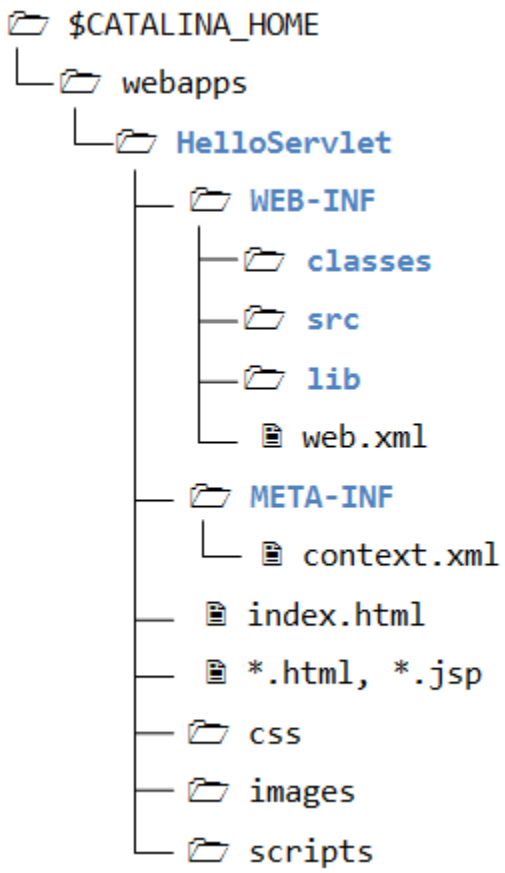
- by Implementing Servlet Interface

1. Create a new Webapp "helloservlet"

We shall begin by defining a new webapp (web application) called "helloservlet" in Tomcat. A webapp, known as a web context in Tomcat, comprises a set of resources, such as HTML files, CSS, JavaScripts, images, programs and libraries

Create a directory "`helloservlet`" under Tomcat's "`webapps`" directory (i.e., "`<CATALINA_HOME>\webapps\helloservlet`", where `<CATALINA_HOME>` denotes Tomcat's installed directory). Create sub-directories "`WEB-INF`" and "`META-INF`" under "`helloservlet`". Create sub-sub-directories "`classes`", "`lib`" and "`src`" under "`WEB-INF`". Take note that the directory names are case-sensitive.

- **webapps\helloservlet**
  This directory is known as *context root* for the web context "`helloservlet`". It contains the resources that are *accessible by the clients*, such as HTML, CSS, Scripts and images. These resources will be delivered to the clients *as it is*. You could create sub-directories such as `images`, `css` and `scripts`, to further categories the resources.



- **webapps\helloservlet\WEB-INF**
  This directory is NOT accessible by the clients directly. This is where you keep your application-specific configuration files (such as "`web.xml`"), and its sub-directories contain program classes, source files, and libraries.

- **webapps\helloservlet\WEB-INF\src**
  Keep the Java program source files. It is a good practice to separate the source files and classes to facilitate deployment.

- **webapps\helloservlet\WEB-INF\classes**
  Keep the Java classes (compiled from the source codes). Classes defined in packages must be kept according to the package directory structure.

- **webapps\helloservlet\WEB-INF\lib**
  keep the JAR files provided by external packages, available to this webapp only.

- **webapps\helloservlet\META-INF**
  This directory is also NOT accessible by the clients. It keeps resources and configurations (e.g., "`context.xml`") related to the particular server (e.g., Tomcat, Glassfish). In contrast, "`WEB-INF`" is for resources related to this webapp, independent of the server.

2. Write a Hello-world Java Servlet - "HelloServlet.java"

```
import javax.servlet.Servlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import java.io.*;

public class HelloServlet implements Servlet
{
	ServletConfig config;

	public void init(ServletConfig config){
		this.config = config;
	}

	public void service(ServletRequest req, ServletResponse res)throws IOException
	{
			PrintWriter writer;
			writer = res.getWriter();
			writer.println("Helloworld...Welcome to my 1st Servlet");
	}

	public void destroy(){

	}

	public String getServletInfo(){
		return "";
	}

	public ServletConfig getServletConfig(){
		return config;
	}
}
```

3. Compile the "HelloServlet.java"

4. Configure the Application Deployment Descriptor - "web.xml"

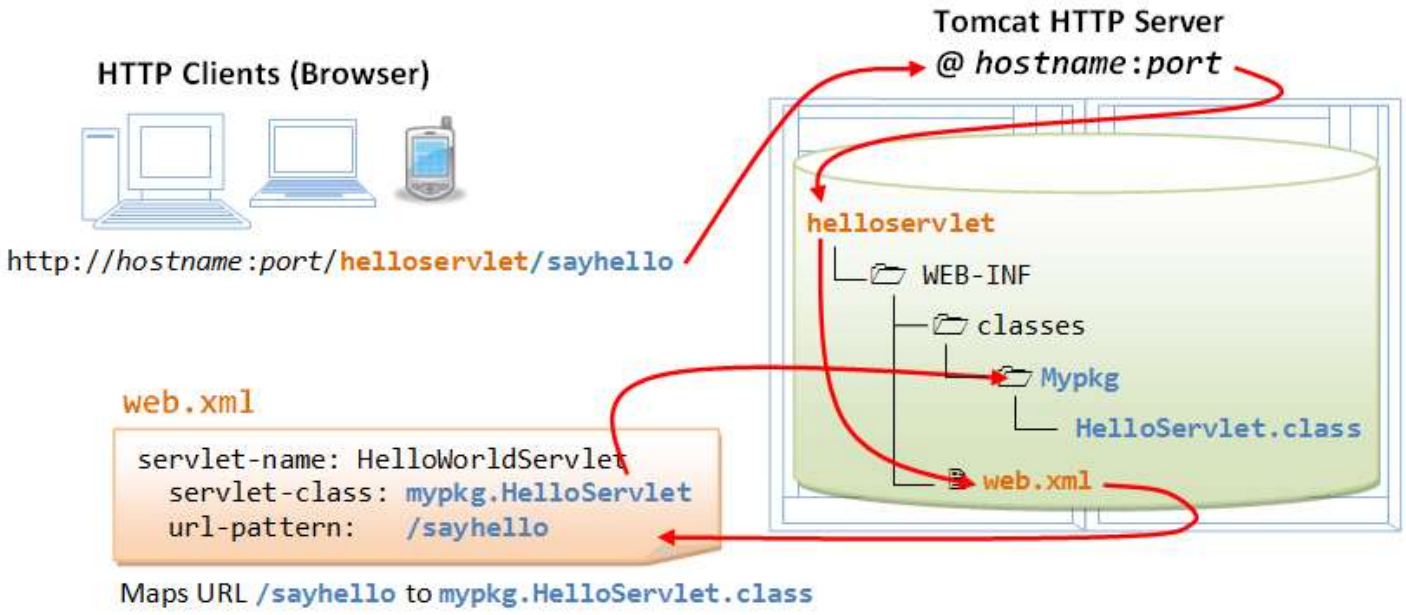
A web user invokes a servlet, which is kept in the web server, by issuing a specific URL from the browser. In this example, we shall configure the following request URL to trigger the "`HelloServlet`":

```
http://hostname:port/helloservlet/sayhello
```

Create a configuration file called "`web.xml`", and save it under "`webapps\helloservlet\WEB-INF`", as follows:

```
<web-app version="3.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
```

```
<!-- To save as <CATALINA_HOME>\webapps\helloservlet\WEB-INF\web.xml -->

            <servlet>
               <servlet-name>HelloWorldServlet</servlet-name>
               <servlet-class>HelloServlet</servlet-class>
            </servlet>

            <!-- Note: All <servlet> elements MUST be grouped together and
                  placed IN FRONT of the <servlet-mapping> elements -->

            <servlet-mapping>
               <servlet-name>HelloWorldServlet</servlet-name>
               <url-pattern>/sayhello</url-pattern>
            </servlet-mapping>
         </web-app>
```



Note: *Each servlet requires a pair of `<servlet>` and `<servlet-mapping>` elements to do the mapping, via an arbitrary but unique `<servlet-name>`. Furthermore, all the `<servlet>` elements must be grouped together and placed before the `<servlet-mapping>` elements (as specified in the XML schema).*

---

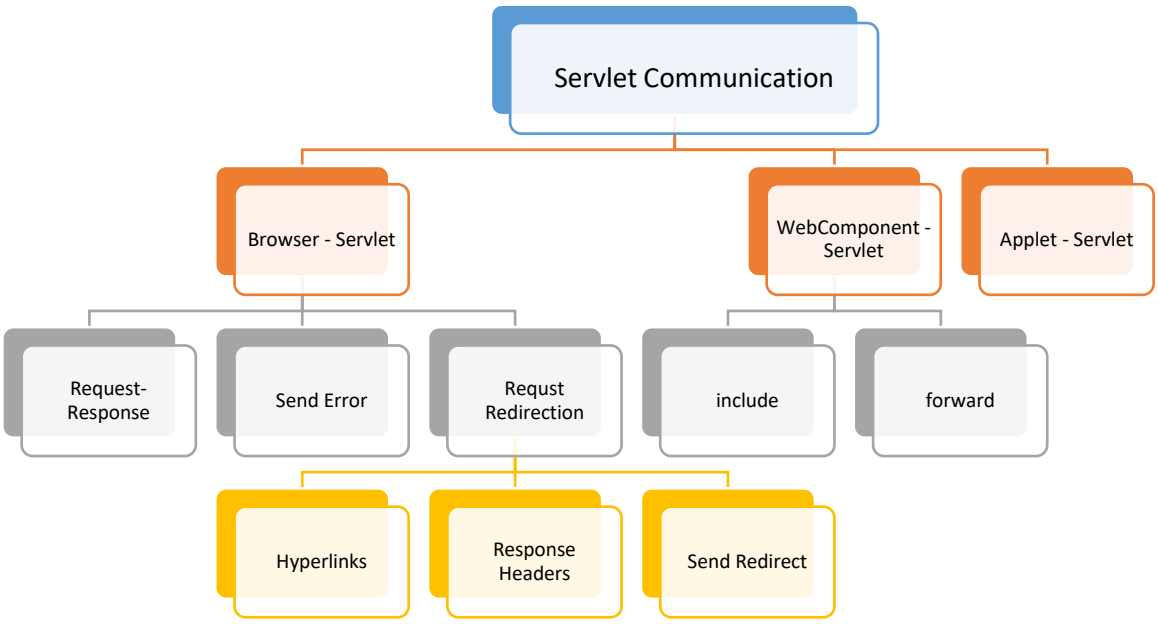
5. Start the Server and deploy the project

To start Apache Tomcat server, double click on the startup.bat file under apache-tomcat/bin directory

6. Access the servlet

http://hostname:port/**helloservlet/sayhello**

Servlet Communication



Session Management

Session in Java Servlet are managed through different ways, such as Cookies, HttpSession API, URL rewriting etc.

What is a Session?

HTTP protocol and Web Servers are stateless, what it means is that for web server every request is a new request to process and they can't identify if it's coming from client that has been sending request previously

But sometimes in web applications, we should know who the client is and process the request accordingly. For example, a shopping cart application should know who is sending the request to add an item and in which cart the item has to be added or who is sending checkout request so that it can charge the amount to correct client.

Session is a conversional state between client and server and it can consists of multiple request and response between client and server. Since HTTP and Web Server both are stateless, the only way to maintain a session is when some unique information about the session (session id) is passed between server and client in every request and response.

There are several ways through which we can provide unique identifier in request and response.

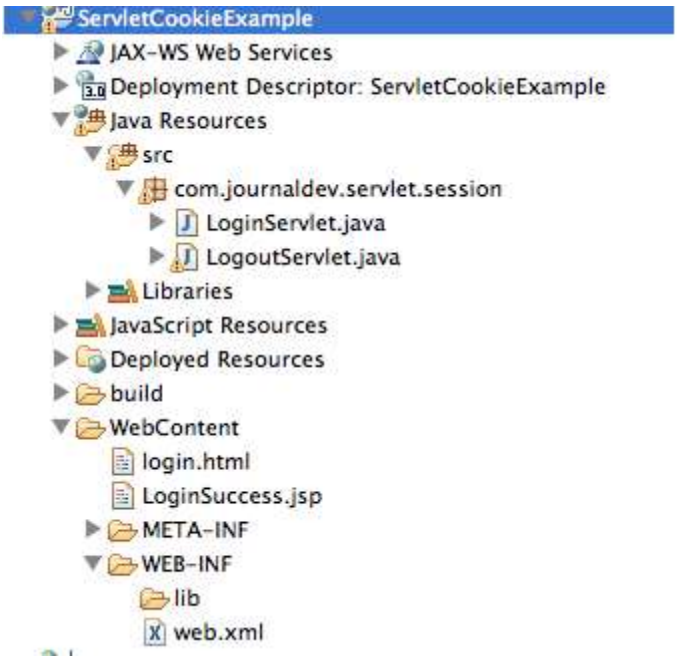
1. **User Authentication** – This is the very common way where we user can provide authentication credentials from the login page and then we can pass the authentication information between server and client to maintain the session. This is not very effective method because it wont work if the same user is logged in from different browsers.
2. **HTML Hidden Field** – We can create a unique hidden field in the HTML and when user starts navigating, we can set its value unique to the user and keep track of the session. This method can't be used with links because it needs the form to be submitted every time request is made from client to server with the hidden field. Also it's not secure because we can get the hidden field value from the HTML source and use it to hack the session.
3. **URL Rewriting** – We can append a session identifier parameter with every request and response to keep track of the session. This is very tedious because we need to keep track of this parameter in every response and make sure it's not clashing with other parameters.
4. **Cookies** – Cookies are small piece of information that is sent by web server in response header and gets stored in the browser cookies. When client make further request, it adds the cookie to the request header and we can utilize it to keep track of the session. We can maintain a session with cookies but if the client disables the cookies, then it won't work.
5. **Session Management API** – Session Management API is built on top of above methods for session tracking. Some of the major disadvantages of all the above methods are:
    - Most of the time we don't want to only track the session, we have to store some data into the session that we can use in future requests. This will require a lot of effort if we try to implement this.
    - All the above methods are not complete in themselves, all of them won't work in a particular scenario. So we need a solution that can utilize these methods of session tracking to provide session management in all cases.

That's why we need **Session Management API** and J2EE Servlet technology comes with session management API that we can use

Session Management in Java – Cookies

Cookies are used a lot in web applications to personalize response based on your choice or to keep track of session. Before moving forward to the Servlet Session Management API, I would like to show how can we keep track of session with cookies through a small web application.

We will create a dynamic web application **ServletCookieExample** with project structure like below image.



Deployment descriptor web.xml of the web application is:

```
<?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>ServletCookieExample</display-name>
  <welcome-file-list>
    <welcome-file>login.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Welcome page of our application is login.html where we will get authentication details from user.

```
<!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>
```

Here is the LoginServlet that takes care of the login request.

```
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;

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

  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)){
                  Cookie loginCookie = new Cookie("user",user);
                  //setting cookie to expiry in 30 mins
                  loginCookie.setMaxAge(30*60);
                  response.addCookie(loginCookie);
                  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);
          }

  }

}
```

Notice the cookie that we are setting to the response and then forwarding it to LoginSuccess.jsp, this cookie will be used there to track the session. Also notice that cookie timeout is set to 30 minutes. Ideally there should be a complex logic to set the cookie value for session tracking so that it won't collide with any other request.

```
<%@ 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>
<%
String userName = null;
Cookie[] cookies = request.getCookies();
if(cookies !=null){
for(Cookie cookie : cookies){
  if(cookie.getName().equals("user")) userName = cookie.getValue();
}
}
if(userName == null) response.sendRedirect("login.html");
%>
<h3>Hi <%=userName %>, Login successful.</h3>
<br>
<form action="LogoutServlet" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>
```

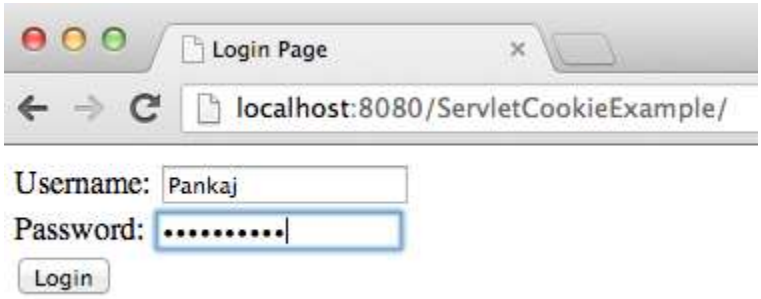
Notice that if we try to access the JSP directly, it will forward us to the login page. When we will click on Logout button, we should make sure that cookie is removed from client browser.

```
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 loginCookie = null;
        Cookie[] cookies = request.getCookies();
        if(cookies != null){
        for(Cookie cookie : cookies){
            if(cookie.getName().equals("user")){
                loginCookie = cookie;
                break;
            }
        }
        }
        if(loginCookie != null){
            loginCookie.setMaxAge(0);
        response.addCookie(loginCookie);
        }
        response.sendRedirect("login.html");
    }
}
```

There is no method to remove the cookie but we can set the maximum age to 0 so that it will be deleted from client browser immediately.

When we run above application, we get response like below images.



Session in Java Servlet – HttpSession

Servlet API provides Session management through `HttpSession` interface. We can get session from HttpServletRequest object using following methods. HttpSession allows us to set objects as attributes that can be retrieved in future requests.

1. **HttpSession getSession()** – This method always returns a HttpSession object. It returns the session object attached with the request, if the request has no session attached, then it creates a new session and return it.
2. **HttpSession getSession(boolean flag)** – This method returns HttpSession object if request has session else it returns null.

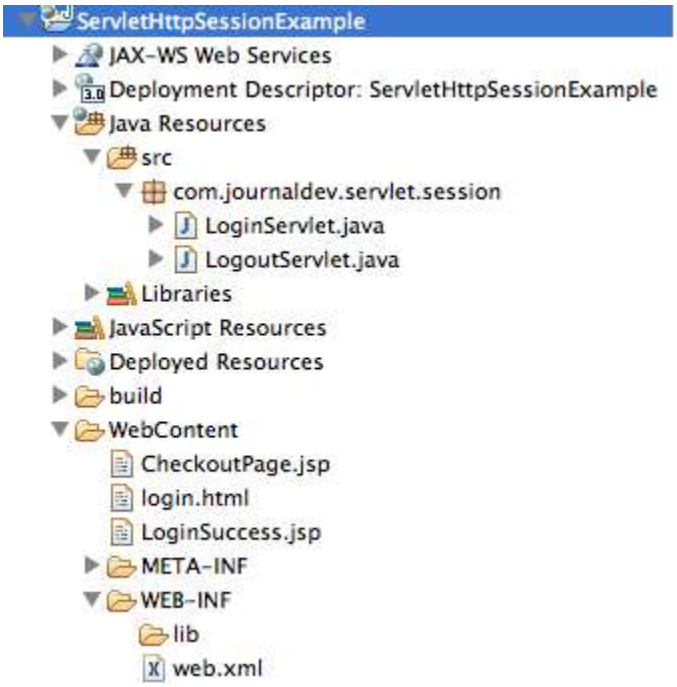
Some of the important methods of HttpSession are:

1. **String getId()** – Returns a string containing the unique identifier assigned to this session.
2. **Object getAttribute(String name)** – Returns the object bound with the specified name in this session, or null if no object is bound under the name. Some other methods to work with Session attributes are `getAttributeNames()`, `removeAttribute(String name)` and `setAttribute(String name, Object value)`.
3. **long getCreationTime()** – Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT. We can get last accessed time with `getLastAccessedTime()` method.
4. setMaxInactiveInterval(int interval) – Specifies the time, in seconds, between client requests before the servlet container will invalidate this session. We can get session timeout value from `getMaxInactiveInterval()` method.
5. **ServletContext getServletContext()** – Returns ServletContext object for the application.
6. **boolean isNew()** – Returns true if the client does not yet know about the session or if the client chooses not to join the session.
7. **void invalidate()** – Invalidates this session then unbinds any objects bound to it.

Understanding JSESSIONID Cookie

When we use HttpServletRequest getSession() method and it creates a new request, it creates the new HttpSession object and also add a Cookie to the response object with name JSESSIONID and value as session id. This cookie is used to identify the HttpSession object in further requests from client. If the cookies are disabled at client side and we are using URL rewriting then this method uses the jsessionid value from the request URL to find the corresponding session. JSESSIONID cookie is used for session tracking, so we should not use it for our application purposes to avoid any session related issues.

Let's see example of session management using HttpSession object. We will create a dynamic web project in Eclipse with servlet context as ServletHttpSessionExample. The project structure will look like below image.



login.html is same like earlier example and defined as welcome page for the application in web.xml

LoginServlet servlet will create the session and set attributes that we can use in other resources or in future requests.

```
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);
                }

          }

        }
```

Our LoginSuccess.jsp code is given 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>
        <%
        //allow access only if session exists
        String user = null;
        if(session.getAttribute("user") == null){
          response.sendRedirect("login.html");
        }else 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>
```

When a JSP resource is used, container automatically creates a session for it, so we can't check if session is null to make sure if user has come through login page, so we are using session attribute to validate request.

CheckoutPage.jsp is another page and it's code is given 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>
<%
//allow access only if session exists
if(session.getAttribute("user") == null){
  response.sendRedirect("login.html");
}
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();
}
}
%>
<h3>Hi <%=userName %>, do the checkout.</h3>
<br>
<form action="LogoutServlet" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>
```

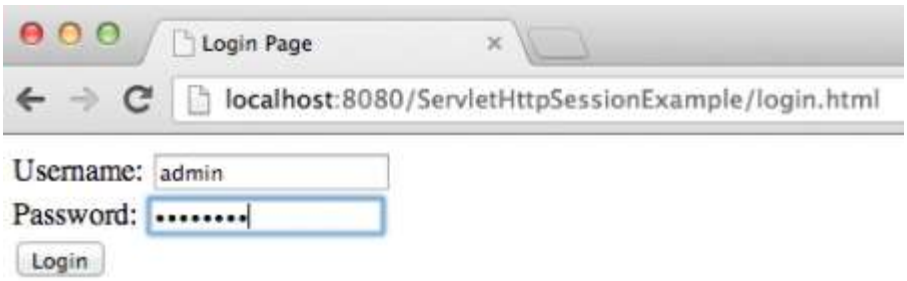
Our LogoutServlet code is given below.

```
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");
    }
}
```

Notice that I am printing JSESSIONID cookie value in logs, you can check server log where it will be printing the same value as Session Id in LoginSuccess.jsp

Below images shows the execution of our web application.



Session Management in Java Servlet – URL Rewriting

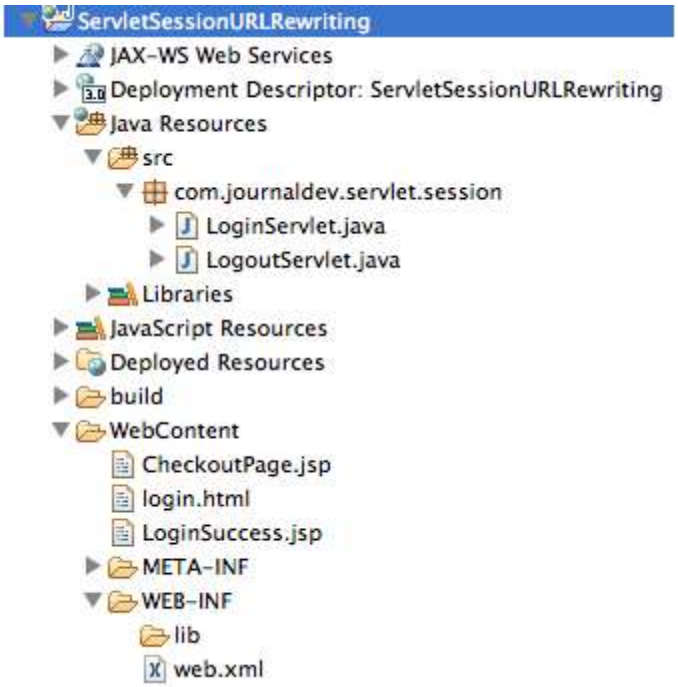
As we saw in last section that we can manage a session with HttpSession but if we disable the cookies in browser, it won't work because server will not receive the JSESSIONID cookie from client. Servlet API provides support for URL rewriting that we can use to manage session in this case.

The best part is that from coding point of view, it's very easy to use and involves one step – encoding the URL. Another good thing with Servlet URL Encoding is that it's a fallback approach and it kicks in only if browser cookies are disabled.

We can encode URL with HttpServletResponse `encodeURL()` method and if we have to redirect the request to another resource and we want to provide session information, we can use `encodeRedirectURL()` method.

We will create a similar project like above except that we will use URL rewriting methods to make sure session management works fine even if cookies are disabled in browser.

ServletSessionURLRewriting project structure in eclipse looks like below image.



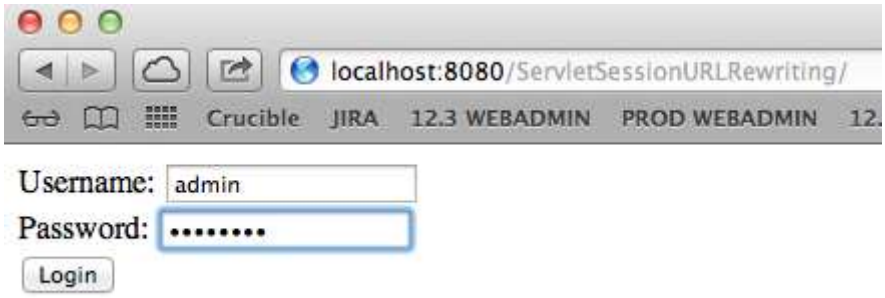

```
package com.journaldev.servlet.session;
import java.io.IOException;
import java.io.PrintWriter;

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;

@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);
                  response.addCookie(userName);
```

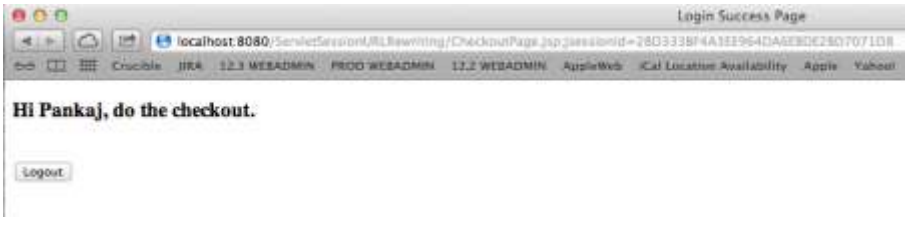
```
                    //Get the encoded URL string
                    String encodedURL = response.encodeRedirectURL("LoginSuccess.jsp");
                    response.sendRedirect(encodedURL);
            }else{
                    RequestDispatcher rd =
getServletContext().getRequestDispatcher("/login.html");
                    PrintWriter out= response.getWriter();
                    out.println("<font color=red>Either user name or password is
wrong.</font>");
                    rd.include(request, response);
            }

    }
}
<%@ 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 = null;
if(session.getAttribute("user") == null){
  response.sendRedirect("login.html");
}else 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();
}
}else{
  sessionID = session.getId();
}
%>
<h3>Hi <%=userName %>, Login successful. Your Session ID=<%=sessionID %></h3>
<br>
User=<%=user %>
<br>
<!-- need to encode all the URLs where we want session information to be passed -->
<a href="<%=response.encodeURL("CheckoutPage.jsp") %>">Checkout Page</a>
<form action="<%=response.encodeURL("LogoutServlet") %>" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>
```

```
<%@ 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>
<%
String userName = null;
//allow access only if session exists
if(session.getAttribute("user") == null){
  response.sendRedirect("login.html");
}else userName = (String) session.getAttribute("user");
String sessionID = null;
Cookie[] cookies = request.getCookies();
if(cookies !=null){
for(Cookie cookie : cookies){
  if(cookie.getName().equals("user")) userName = cookie.getValue();
}
}
%>
<h3>Hi <%=userName %>, do the checkout.</h3>
<br>
<form action="<%=response.encodeURL("LogoutServlet") %>" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>

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;

@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());
                    }
                    cookie.setMaxAge(0);
                    response.addCookie(cookie);
            }
            }
            //invalidate the session if exists
            HttpSession session = request.getSession(false);
            System.out.println("User="+session.getAttribute("user"));
            if(session != null){
                    session.invalidate();
            }
            //no encoding because we have invalidated the session
            response.sendRedirect("login.html");
        }

    }
```

When we run this project keeping cookies disabled in the browser, below images shows the response pages, notice the jsessionid in URL of browser address bar. Also notice that on LoginSuccess page, user name is null because browser is not sending the cookie send in the last response.





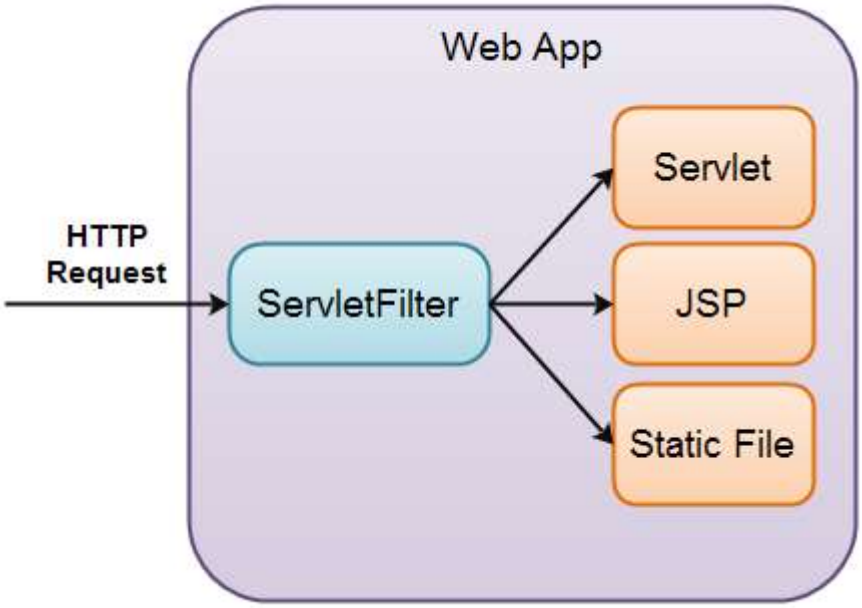
If cookies are not disabled, you won't see jsessionid in the URL because Servlet Session API will use cookies in that case

Servlet Filter

Java Servlet Filter is used to intercept request and do some pre-processing and can be used to intercept response and do post-processing before sending to client in web application.

A Servlet filter is an object that can intercept HTTP requests targeted at your web application.

A servlet filter can intercept requests both for servlets, JSP's, HTML files or other static content, as illustrated in the diagram below:



When the servlet container calls a method in a servlet on behalf of the client, the HTTP request that the client sent is, by default, passed directly to the servlet. The response that the servlet generates is, by default, passed directly back to the client, with its content unmodified by the container. In this scenario, the servlet must process the request and generate as much of the response as the application requires.

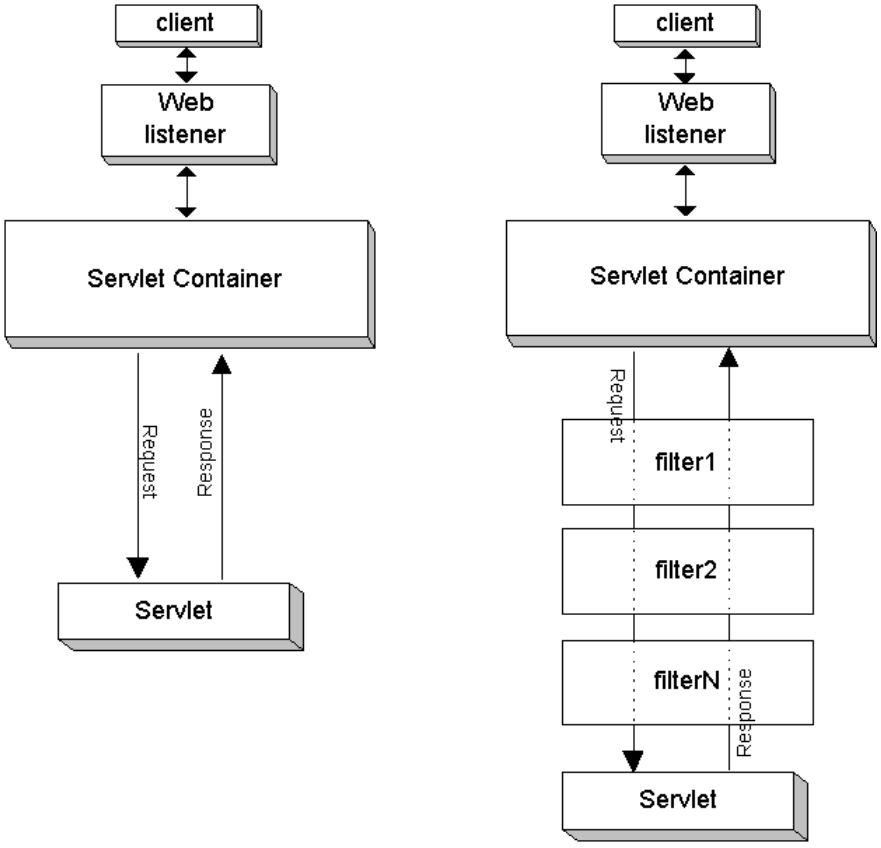
But there are many cases in which some preprocessing of the request for servlets would be useful. In addition, it is sometimes useful to modify the response from a class of servlets. One example is encryption. A servlet, or a group of servlets in an application, may generate response data that is sensitive and should not go out over the network in clear-text form, especially when the connection has been made using a nonsecure protocol such as HTTP. A filter can encrypt the responses. Of course, in this case the client must be able to decrypt the responses.

A common scenario for a filter is one in which you want to apply preprocessing or postprocessing to requests or responses for a group of servlets, not just a single servlet. If you need to modify the request or response for just one servlet, there is no need to create a filter—just do what is required directly in the servlet itself.

Note that filters are not servlets. They do not implement and override `HttpServlet` methods such as `doGet()` or `doPost()`. Rather, a filter implements the methods of the `javax.servlet.Filter` interface. The methods are:

1. init()
2. destroy()

3. doFilter()



Why Filter?

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 jsp pages, 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 servlet filters are:

- Logging request parameters to log files.
- Authentication and authorization 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 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 servlet 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.

There are various types of filters suggested by the specifications −

- Authentication Filters.
- Data compression Filters.
- Encryption Filters.
- Filters that trigger resource access events.
- Image Conversion Filters.
- Logging and Auditing Filters.
- MIME-TYPE Chain Filters.
- Tokenizing Filters .
- XSL/T Filters That Transform XML Content.

Filters are deployed in the deployment descriptor file **web.xml** and then map to either servlet names or URL patterns in your application's deployment descriptor.

When the web container starts up your web application, it creates an instance of each filter that you have declared in the deployment descriptor. The filters execute in the order that they are declared in the deployment descriptor.

Servlet Filter interface

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

Servlet 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.

Create Filter

In order to create a servlet filter you must implement the `javax.servlet.Filter` interface. Here is an example servlet filter implementation:

```
import javax.servlet.*;
import java.io.IOException;
```

```
/**

 */
public class SimpleServletFilter implements Filter {

    public void init(FilterConfig filterConfig) throws ServletException {
    }

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

    }

    public void destroy() {
    }
}
```

When the servlet filter is loaded the first time, its `init()` method is called, just like with servlets.

When a HTTP request arrives at your web application which the filter intercepts, the filter can inspect the request URI, the request parameters and the request headers, and based on that decide if it wants to block or forward the request to the target servlet, JSP etc.

It is the `doFilter()` method that does the interception. Here is a sample implementation:

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

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

    if(!"blockTheRequest".equals(myParam)){
        filterChain.doFilter(request, response);
    }
}
```

Notice how the `doFilter()` method checks a request parameter, `myParam`, to see if it equals the string "blockTheRequest". If not, the request is forwarded to the target of the request, by calling the `filterChain.doFilter()` method. If this method is not called, the request is not forwarded, but just blocked.

The servlet filter above just ignores the request if the request parameter `myParam` equals "blockTheRequest". You can also write a different response back to the browser. Just use the `ServletResponse` object to do so, just like you would inside a servlet.

You may have to cast the `ServletResponse` to a `HttpResponse` to obtain a `PrintWriter` from it. Otherwise you only have the `OutputStream` available via `getOutputStream()`.

Here is an example:

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

```
        String myParam = request.getParameter("myParam");

        if(!"blockTheRequest".equals(myParam)){
            filterChain.doFilter(request, response);
            return;
        }

        HttpResponse httpResponse = (HttpResponse) httpResponse;
        httpResponse.getWriter().write("a different response... e.g in HTML");
    }
```

Configuring the Servlet Filter in web.xml

Filters are defined and then mapped to a URL or Servlet, in much the same way as Servlet is defined and then mapped to a URL pattern. Create the following entry for filter tag in the deployment descriptor file **web.xml**

You need to configure the servlet filter in the web.xml file of your web application, before it works. Here is how you do that:

```
    <filter>
        <filter-name>myFilter</filter-name>
        <filter-class>servlets.SimpleServletFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>myFilter</filter-name>
        <url-pattern>*.simple</url-pattern>
    </filter-mapping>
```

With this configuration all requests with URL's ending in `.simple` will be intercepted by the servlet filter. All others will be left untouched.

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.

Using Multiple Filters

Your web application may define several different filters with a specific purpose. Consider, you define two filters AuthenFilter and LogFilter. Rest of the process would remain as explained above except you need to create a different mapping as mentioned below –

```
    <filter>
       <filter-name>LogFilter</filter-name>
       <filter-class>LogFilter</filter-class>
       <init-param>
          <param-name>test-param</param-name>
          <param-value>Initialization Paramter</param-value>
       </init-param>
```

```
</filter>

<filter>
   <filter-name>AuthenFilter</filter-name>
   <filter-class>AuthenFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Filters Application Order

The order of filter-mapping elements in web.xml determines the order in which the web container applies the filter to the servlet. To reverse the order of the filter, you just need to reverse the filter-mapping elements in the web.xml file.

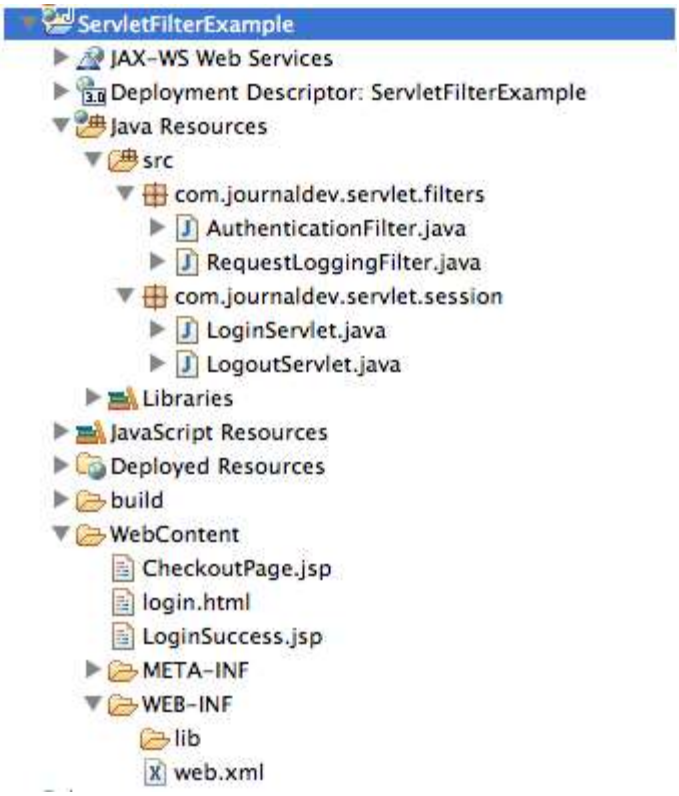
For example, above example would apply LogFilter first and then it would apply AuthenFilter to any servlet but the following example would reverse the order −

```
<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

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.

login.html code:

```
<!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;

@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

LoginSuccess.jsp code:

```
<%@ 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>
```

Notice that there is no session validation logic in the above JSP. It contains link to another JSP page, CheckoutPage.jsp.

CheckoutPage.jsp code:

```
<%@ 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>
<%
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();
}
}
%>
<h3>Hi <%=userName %>, do the checkout.</h3>
<br>
```

```
<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
  }

}
```

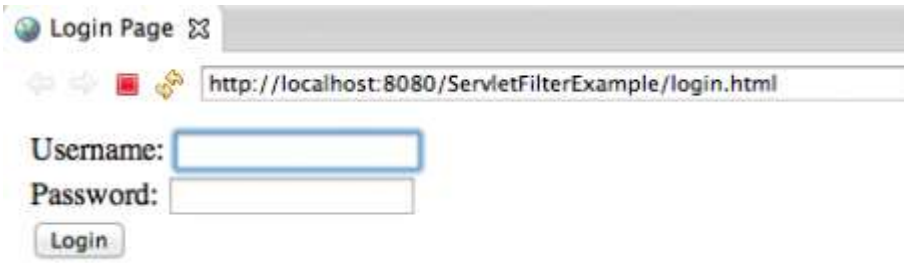
```
package com.journaldev.servlet.filters;

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

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 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.

Servlet Listener

Listeners are the classes which listens to a particular type of events and when that event occurs , triggers the functionality. Each type of listener is bind to a type of event. In this chapter we will discuss the types of listeners supported by servlet framework.

Events are basically occurrence of something. Changing the state of an object is known as an event.

We can perform some important tasks at the occurrence of these exceptions, such as counting total and current logged-in users, creating tables of the database at time of deploying the project, creating database connection object etc.

There are many Event classes and Listener interfaces in the javax.servlet and javax.servlet.http packages.

Why do we have Servlet Listener?

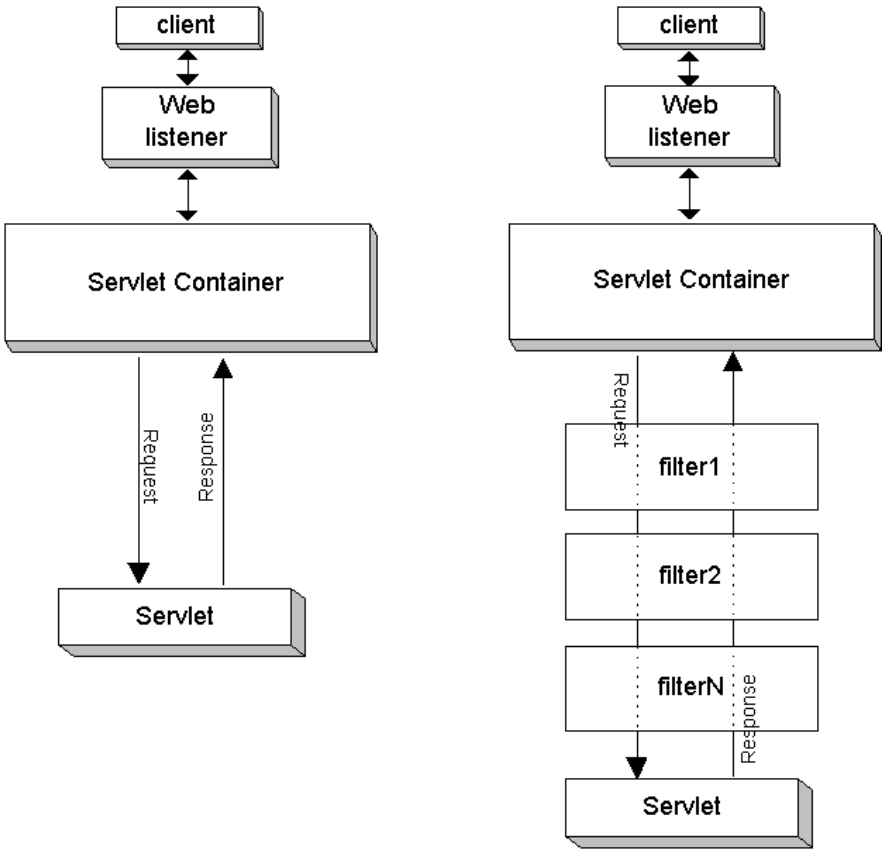
We know that using `ServletContext`, we can create an attribute with application scope that all other servlets can access but we can initialize **ServletContext init parameters as String only** in deployment descriptor (web.xml). What if our application is database oriented and we want to set an attribute 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.

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.



Event Categories and Listener Interfaces

There are two levels of servlet events:

Servlet context-level (application-level) event

This event involves resources or state held at the level of the application servlet context object.

Session-level event

> This event involves resources or state associated with the series of requests from a single user session; that is, associated with the HTTP session object.

Each of these two levels has two event categories:

- Lifecycle changes
- Attribute changes

You can create one or more event listener classes for each of the four event categories. A single listener class can monitor multiple event categories.

Create an event listener class by implementing the appropriate interface or interfaces of the `javax.servlet` package or `javax.servlet.http` package.

Event Category	Event Descriptions	Java Interface
Servlet context lifecycle changes	Servlet context creation, at which point the first request can be serviced Imminent shutdown of the servlet context	javax.servlet. ServletContextListener

Servlet context attribute changes	Addition of servlet context attributes Removal of servlet context attributes Replacement of servlet context attributes	javax.servlet. ServletContextAttributeListener
Session lifecycle changes	Session creation Session invalidation Session timeout	javax.servlet.http. HttpSessionListener
Session attribute changes	Addition of session attributes Removal of session attributes Replacement of session attributes	javax.servlet.http. HttpSessionAttributeListener

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

Event Listener Declaration and Invocation

Event listeners are declared in the application `web.xml` deployment descriptor through `<listener>` elements under the top-level `<web-app>` element. Each listener has its own `<listener>` element, with a `<listener-class>` subelement specifying the class name. Within each event category, event listeners should be specified in the order in which you would like them to be invoked when the application runs.

After the application starts up and before it services the first request, the servlet container creates and registers an instance of each listener class that you have declared. For each event category, listeners are registered in the order in which they are declared. Then, as the application runs, event listeners for each category are invoked in the order of their registration. All listeners remain active until after the last request is serviced for the application.

Upon application shutdown, session event listeners are notified first, in reverse order of their declarations, then application event listeners are notified, in reverse order of their declarations.

Here is an example of event listener declarations, from the Sun Microsystems Java Servlet Specification, Version 2.3:

```
<web-app>
   <display-name>MyListeningApplication</display-name>
   <listener>
      <listener-class>com.acme.MyConnectionManager</listenerclass>
   </listener>
   <listener>
      <listener-class>com.acme.MyLoggingModule</listener-class>
   </listener>
   <servlet>
      <display-name>RegistrationServlet</display-name>
```

```
        ...
    </servlet>
</web-app>
```

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

ServletContextEvent and ServletContextListener

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

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

Constructor of ServletContextEvent class

There is only one constructor defined in the ServletContextEvent class. The web container creates the instance of ServletContextEvent after the ServletContext instance.

1. ServletContextEvent(ServletContext e)

Method of ServletContextEvent class

There is only one method defined in the ServletContextEvent class:

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

Methods of ServletContextListener interface

There are two methods declared in the ServletContextListener interface which must be implemented by the servlet programmer to perform some action such as creating database connection etc.

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

index.html

```
<a href="servlet1">fetch records</a>
```

MyListener.java

```
import javax.servlet.*;
import java.sql.*;

public class MyListener implements ServletContextListener{
public void contextInitialized(ServletContextEvent event) {
```

```
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

//storing connection object as an attribute in ServletContext
ServletContext ctx=event.getServletContext();
ctx.setAttribute("mycon", con);

}catch(Exception e){e.printStackTrace();}
}

public void contextDestroyed(ServletContextEvent arg0) {}
}
```

FetchData.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

public class FetchData extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

try{
//Retrieving connection object from ServletContext object
ServletContext ctx=getServletContext();
Connection con=(Connection)ctx.getAttribute("mycon");

//retieving data  from emp32 table
PreparedStatement ps=con.prepareStatement("select * from emp32",
ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);

ResultSet rs=ps.executeQuery();
while(rs.next()){
out.print("<br>"+rs.getString(1)+" "+rs.getString(2));
}

con.close();
}catch(Exception e){e.printStackTrace();}

out.close();
}
}
```

Improving Servlet performance to fetch records from database

In this example, we are going to improve the performance of web application to fetch records from the database. To serve this, we are storing the data of the table in a collection, and reusing this collection in our servlet. So, we are not directly hitting the database again and again. By this, we are improving the performance.

To run this application, you need to create following table with some records.

```
1.  CREATE TABLE  "CSUSER"
2.    (    "USERID" NUMBER,
3.     "USERNAME" VARCHAR2(4000),
4.     "USERPASS" VARCHAR2(4000),
5.     "USEREMAIL" VARCHAR2(4000),
6.     "USERCOUNTRY" VARCHAR2(4000),
7.     "CONTACT" NUMBER,
8.      CONSTRAINT "CSUSER_PK" PRIMARY KEY ("USERID") ENABLE
9.    )
10. /
```

Example to Improve the performance of servlet to fetch records from database

In this example, we have created 6 pages.

1. **index.html**
2. **User.java**
3. **MyListener.java**
4. **MyServlet1.java**
5. **MyServlet2.java**
6. **web.xml**

1) index.html

This html file contains two links that sends request to the servlet.

```
<a href="servlet1">first servlet</a>|
<a href="servlet2">second servlet</a>
```

---

2) User.java

This is simple bean class containing 3 properties with its getters and setters. This class represents the table of the database.

```
public class User {
private int id;
private String name,password;

public int getId() {
```

```
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}


}
```

3) MyListener.java

It is the listener class. When the project will be deployed, contextInitialized method of ServletContextListener is invoked by default. Here, we are getting the records of the table and storing it in the User class object which is added in the ArrayList class object. At last, all the records of the table will be stored in the ArrayList class object (collection). Finally, we are storing the ArrayList object in the ServletConext object as an attribute so that we can get it in the servlet and use it.

```
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import java.sql.*;
import java.util.ArrayList;

public class MyListener implements ServletContextListener{

 public void contextInitialized(ServletContextEvent e) {

  ArrayList list=new ArrayList();
   try{
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection con=DriverManager.getConnection(
     "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

    PreparedStatement ps=con.prepareStatement("select * from csuser");
    ResultSet rs=ps.executeQuery();
    while(rs.next()){
     User u=new User();
     u.setId(rs.getInt(1));
     u.setName(rs.getString(2));
     u.setPassword(rs.getString(3));
     list.add(u);
    }
```

```
        con.close();

       }catch(Exception ex){System.out.print(ex);}

       //storing the ArrayList object in ServletContext
       ServletContext context=e.getServletContext();
       context.setAttribute("data",list);

     }
     public void contextDestroyed(ServletContextEvent arg0) {
        System.out.println("project undeployed...");
     }

    }
```

4) MyServlet1.java

This servlet gets the information from the servlet context object and prints it.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Iterator;
import java.util.List;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class MyServlet1 extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse
   response)throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    long before=System.currentTimeMillis();

    ServletContext context=getServletContext();
    List list=(List)context.getAttribute("data");

    Iterator itr=list.iterator();
    while(itr.hasNext()){
     User u=(User)itr.next();
     out.print("<br>"+u.getId()+" "+u.getName()+" "+u.getPassword());
    }

    long after=System.currentTimeMillis();
```

```
        out.print("<br>total time :"+(after-before));

        out.close();
        }

    }
```

5) MyServlet2.java

It is same as MyServlet1. This servlet gets the information from the servlet context object and prints it.

```
    import java.io.IOException;
    import java.io.PrintWriter;
    import java.sql.Connection;
    import java.sql.DriverManager;
    import java.sql.PreparedStatement;
    import java.sql.ResultSet;
    import java.util.Iterator;
    import java.util.List;

    import javax.servlet.ServletContext;
    import javax.servlet.ServletException;
    import javax.servlet.http.HttpServlet;
    import javax.servlet.http.HttpServletRequest;
    import javax.servlet.http.HttpServletResponse;


    public class MyServlet2 extends HttpServlet {
     public void doGet(HttpServletRequest request, HttpServletResponse
       response)throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        long before=System.currentTimeMillis();

        ServletContext context=getServletContext();
        List list=(List)context.getAttribute("data");

        Iterator itr=list.iterator();
        while(itr.hasNext()){
         User u=(User)itr.next();
         out.print("<br>"+u.getId()+" "+u.getName()+" "+u.getPassword());
        }

        long after=System.currentTimeMillis();
        out.print("<br>total time :"+(after-before));

        out.close();
        }

    }
```

6) web.xml

Here we are containing the information about servlets and listener.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

 <listener>
 <listener-class>MyListener</listener-class>
 </listener>

 <servlet>
    <servlet-name>MyServlet1</servlet-name>
    <servlet-class>MyServlet1</servlet-class>

  </servlet>
  <servlet>
    <servlet-name>MyServlet2</servlet-name>
    <servlet-class>MyServlet2</servlet-class>

  </servlet>

   <servlet-mapping>
    <servlet-name>MyServlet1</servlet-name>
    <url-pattern>/servlet1</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>MyServlet2</servlet-name>
    <url-pattern>/servlet2</url-pattern>
  </servlet-mapping>

</web-app>
```

HttpSessionEvent and HttpSessionListener

The HttpSessionEvent is notified when session object is changed. The corresponding Listener interface for this event is HttpSessionListener.

We can perform some operations at this event such as counting total and current logged-in users, maintaing a log of user details such as login time, logout time etc.

Methods of HttpSessionListener interface

There are two methods declared in the HttpSessionListener interface which must be implemented by the servlet programmer to perform some action.

1. **public void sessionCreated(HttpSessionEvent e)**: is invoked when session object is created.

2. **public void sessionDestroyed(ServletContextEvent e)**: is invoked when session is invalidated.

Example of HttpSessionEvent and HttpSessionListener to count total and current logged-in users

In this example, are counting the total and current logged-in users. For this purpose, we have created three files:

1. **index.html**: to get input from the user.
2. **MyListener.java**: A listener class that counts total and current logged-in users and stores this information in ServletContext object as an attribute.
3. **First.java**: A Servlet class that creates session and prints the total and current logged-in users.

Logout.java: A Servlet class that invalidates session.

index.html

```
<form action="servlet1">
Name:<input type="text" name="username"><br>
Password:<input type="password" name="userpass"><br>

<input type="submit" value="login"/>
</form>
```

MyListener.java

```
import javax.servlet.ServletContext;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class CountUserListener implements HttpSessionListener{
    ServletContext ctx=null;
    static int total=0,current=0;

    public void sessionCreated(HttpSessionEvent e) {
    total++;
    current++;

    ctx=e.getSession().getServletContext();
    ctx.setAttribute("totalusers", total);
    ctx.setAttribute("currentusers", current);

    }

    public void sessionDestroyed(HttpSessionEvent e) {
        current--;
        ctx.setAttribute("currentusers",current);
    }

}
```

First.java

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

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

public class First extends HttpServlet {
public void doGet(HttpServletRequest request,
 HttpServletResponse response)
    throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String n=request.getParameter("username");
        out.print("Welcome "+n);

        HttpSession session=request.getSession();
        session.setAttribute("uname",n);

        //retrieving data from ServletContext object
        ServletContext ctx=getServletContext();
        int t=(Integer)ctx.getAttribute("totalusers");
        int c=(Integer)ctx.getAttribute("currentusers");
        out.print("<br>total users= "+t);
        out.print("<br>current users= "+c);

        out.print("<br><a href='logout'>logout</a>");

        out.close();
    }

}
```

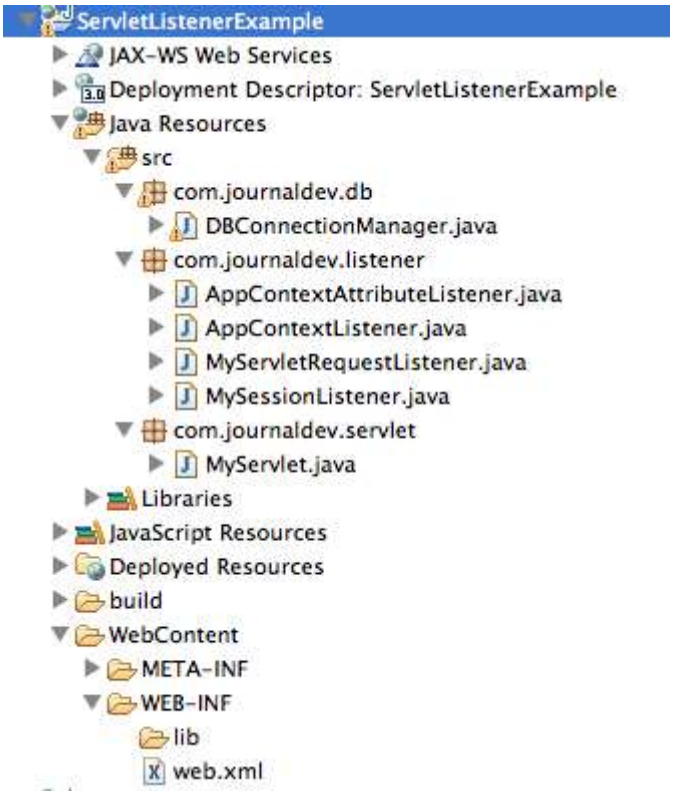
Logout.java

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

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;


public class LogoutServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
 HttpServletResponse response)
```

```
            throws ServletException, IOException {

            response.setContentType("text/html");
            PrintWriter out = response.getWriter();

            HttpSession session=request.getSession(false);
            session.invalidate();//invalidating session

            out.print("You are successfully logged out");


            out.close();
        }

    }
```

Servlet Listener Example – All Events

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



web.xml: In deployment descriptor, we can define 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.

ServletContextListener

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.");

    }

}
```

ServletContextAttributeListener

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()+","+servletContextAttributeEvent.getValue(
)+"}");
    }

    public void attributeReplaced(ServletContextAttributeEvent
servletContextAttributeEvent) {
          System.out.println("ServletContext attribute
replaced::{"+servletContextAttributeEvent.getName()+","+servletContextAttributeEvent.getVal
ue()+"}");
```

```
    }

    public void attributeRemoved(ServletContextAttributeEvent servletContextAttributeEvent)
{
          System.out.println("ServletContext attribute
removed::{"+servletContextAttributeEvent.getName()+","+servletContextAttributeEvent.getValu
e()+"}");
    }

}
```

HttpSessionListener

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());
    }

}
```

ServletRequestListener

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.TldLocation
sCache@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 shutdown the application or shutdown the container.

Event Listener Coding and Deployment Guidelines

Be aware of the following rules and guidelines for event listener classes:

- In a multithreaded application, attribute changes may occur simultaneously. There is no requirement for the servlet container to synchronize the resulting notifications; the listener classes themselves are responsible for maintaining data integrity in such a situation.
- Each listener class must have a public zero-argument constructor.
- Each listener class file must be packaged in the application WAR file, either under `/WEB-INF/classes` or in a JAR file in `/WEB-INF/lib`.

Note:

In a distributed environment, the scope of event listeners is one for each deployment descriptor declaration for each JVM. There is no requirement for distributed Web containers to propagate servlet context events or session events to additional JVMs. The servlet specification discusses this.

Use of JDBC in Servlets

A servlet can access a database using a JDBC driver. The power of servlets comes from their ability to retrieve data from a database. A servlet can generate dynamic HTML by getting information from a database and sending it back to the client. A servlet can also update a database, based on information passed to it in the HTTP request.

Servlet JDBC Database Connection Example

Develop a web application that should have following features.

1. User can register and then login to the application.
2. The user's information should be maintained in database.
3. Use standard logging framework log4j.
4. The application should support session management, no JSPs should be visible without session. Users can logout anytime from the application.
5. We should not show application and server details to user incase of any exception in application or other common errors like 404.

Design Decisions

1. Since login page is the entry point of application, we will have a simple login.html where user can enter their credentials; email and password. We can't rely on javascript validations, so we will do server side validation and incase of missing information we will redirect user to login page with error details.

   We will have a register.html from where user can register to our application, we will provide it's link in the login page for new user. User should provide email, password, name and country details for registration.

2. If any information is missing, user will remain on same page with error message. If registration is successful, user will be forwarded to the login page with registration success information and they can use email and password to login.

3. We will use MySql database for persisting user information. We will create a new database, user and Users table for our application. Since our application totally depends on Database Connection, we will create a servlet context listener to initialize the database connection and set it as context attribute for other servlets.

4. We will keep DB configuration details configurable through deployment descriptor. We will also add MySql Java Connector jar to the application libraries.

5. Since we want to use log4j and configure it properly before usage, we will utilize servlet context listener to configure log4j and keep the log4j configuration XML file location in web.xml init parameters. We will write our application logs in a separate log file dbexample.log for easier debugging.

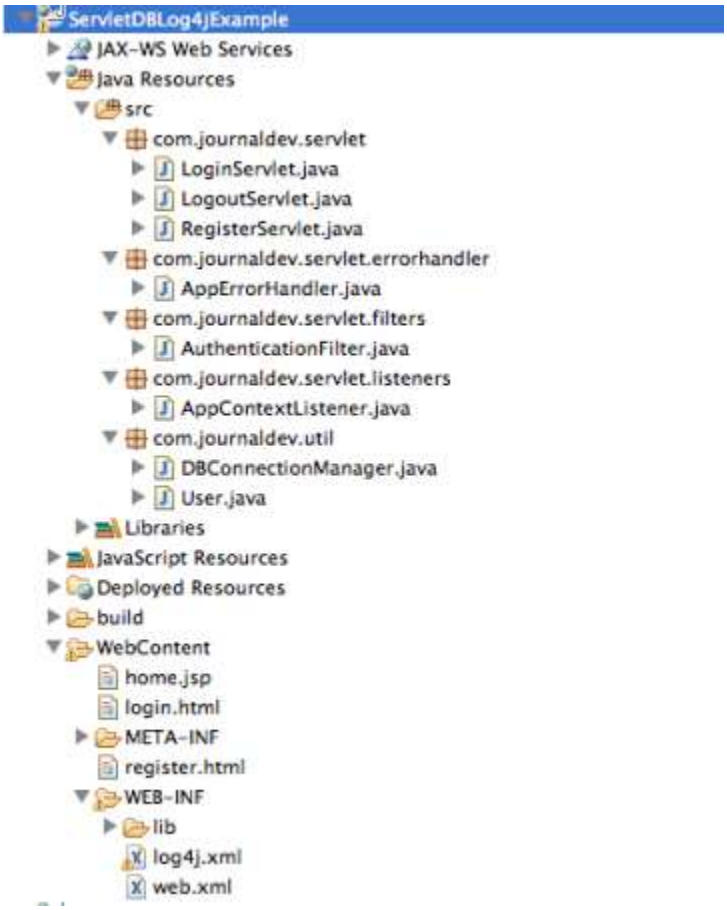
6. In case of any exceptions like “Database Connection Error” or 404 errors, we want to present a useful page to user. We will utilize servlet exception handling and write our own Exception Handler servlet and configure it in deployment descriptor.

7. Once the user logins successfully, we will create a session for the user and forward them to home.jsp where we will show basic information of the user. We will have a model class User that will store the user data into session. User home page also provide logout button that will invalidate the session and forward them to login page.

8. We need to make sure all the JSPs and other resources are accessible only when user has a valid session, rather than keeping session validation login in all the resources, we will create a Servlet Filter for session validation and configure it in deployment descriptor.

9. We will use Servlet 3.0 features for servlet configuration, listeners and filters rather than keeping all of these in deployment descriptor. We will use Eclipse for development and Tomcat 7 for deployment.

Based on above requirements and design decisions, we will create our dynamic web project whose project structure will look like below image.



Servlet JDBC Example – Database Setup

We will use below MySql script for new Database, User and Table setup to be used in our application.

```
-- login with root to create user, DB and table and provide grants
```

```
create user 'pankaj'@'localhost' identified by 'pankaj123';

grant all on *.* to 'pankaj'@'localhost' identified by 'pankaj123';

create database UserDB;

use UserDB;

CREATE TABLE `Users` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(20) NOT NULL DEFAULT '',
  `email` varchar(20) NOT NULL DEFAULT '',
  `country` varchar(20) DEFAULT 'USA',
  `password` varchar(20) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

Login and Registration HTML Pages

Login and Registration HTML pages are simple pages with form to submit the user information, their code looks like below.

login.html code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="US-ASCII">
<title>Login Page</title>
</head>
<body>
<h3>Login with email and password</h3>
<form action="Login" method="post">
<strong>User Email</strong>:<input type="text" name="email"><br>
<strong>Password</strong>:<input type="password" name="password"><br>
<input type="submit" value="Login">
</form>
<br>
If you are new user, please <a href="register.html">register</a>.
</body>
</html>
```

register.html code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="US-ASCII">
<title>Register Page</title>
</head>
<body>
<h3>Provide all the fields for registration.</h3>
<form action="Register" method="post">
<strong>Email ID</strong>:<input type="text" name="email"><br>
```

```
<strong>Password</strong>:<input type="password" name="password"><br>
<strong>Name</strong>:<input type="text" name="name"><br>
<strong>Country</strong>:<input type="text" name="country"><br>
<input type="submit" value="Register">
</form>
<br>
If you are registered user, please <a href="login.html">login</a>.
</body>
</html>
```

Deployment Descriptor and log4j configuration

We will have log4j configuration file inside WEB-INF folder and it will be packaged with the application WAR file.

log4j.xml code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"
  debug="false">
  <appender name="dbexample" class="org.apache.log4j.RollingFileAppender">
          <param name="File" value="${catalina.home}/logs/dbexample.log"/>
          <param name="Append"            value="true" />
          <param name="ImmediateFlush"    value="true" />
          <param name="MaxFileSize"       value="20MB" />
          <param name="MaxBackupIndex"    value="10" />
          <layout class="org.apache.log4j.PatternLayout">
                  <param name="ConversionPattern" value="%-4r [%t] %-5p %c %x - %m%n" />
          </layout>
  </appender>

  <logger name="com.journaldev" additivity="false">
          <level value="DEBUG" />
          <appender-ref ref="dbexample"/>
  </logger>

  <root>
          <level value="debug" />
          <appender-ref ref="dbexample" />
  </root>

</log4j:configuration>
```

Our deployment descriptor (web.xml) looks like below.

```
<?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>ServletDBLog4jExample</display-name>
  <welcome-file-list>
    <welcome-file>login.html</welcome-file>
  </welcome-file-list>
```

```
    <context-param>
      <param-name>dbUser</param-name>
      <param-value>pankaj</param-value>
    </context-param>
    <context-param>
      <param-name>dbPassword</param-name>
      <param-value>pankaj123</param-value>
    </context-param>
    <context-param>
      <param-name>dbURL</param-name>
      <param-value>jdbc:mysql://localhost:3306/UserDB</param-value>
    </context-param>
    <context-param>
      <param-name>log4j-config</param-name>
      <param-value>WEB-INF/log4j.xml</param-value>
    </context-param>

    <error-page>
      <error-code>404</error-code>
      <location>/AppErrorHandler</location>
    </error-page>
    <error-page>
      <exception-type>java.lang.Throwable</exception-type>
      <location>/AppErrorHandler</location>
    </error-page>

    <filter>
      <filter-name>AuthenticationFilter</filter-name>
      <filter-class>com.journaldev.servlet.filters.AuthenticationFilter</filter-class>
    </filter>
    <filter-mapping>
      <filter-name>AuthenticationFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

Notice following points in the web.xml configuration.

1. login.html is provided welcome file in the welcome files list.
2. Database connection parameters are made configurable and kept as servlet context init params.
3. log4j configuration file location is also configurable and relative location is provided as context init param.
4. Our custom exception handler servlet AppErrorHandler is configured to handle all the exceptions thrown by our application code and 404 errors.
5. AuthenticationFilter is configured to filter all the incoming requests to the application, this is the place where we will have session validation logic.

Model Classes and Database Connection Manager Class

User.java is a simple java bean that will hold the user information as session attribute.

```
package com.journaldev.util;
```

```
import java.io.Serializable;

public class User implements Serializable{

  private static final long serialVersionUID = 6297385302078200511L;

  private String name;
  private String email;
  private int id;
  private String country;

  public User(String nm, String em, String country, int i){
          this.name=nm;
          this.id=i;
          this.country=country;
          this.email=em;
  }

  public void setName(String name) {
          this.name = name;
  }


  public void setEmail(String email) {
          this.email = email;
  }


  public void setId(int id) {
          this.id = id;
  }


  public void setCountry(String country) {
          this.country = country;
  }


  public String getName() {
          return name;
  }

  public String getEmail() {
          return email;
  }

  public int getId() {
          return id;
  }

  public String getCountry() {
          return country;
  }
```

```
  @Override
  public String toString(){
          return "Name="+this.name+", Email="+this.email+", Country="+this.country;
  }
}
```

DBConnectionManager.java is the utility class for MySql database connection and it has a method that returns the connection object. We will use this class for database connection and then set the connection object to servlet context attribute that other servlets can use.

```
package com.journaldev.util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnectionManager {

  private Connection connection;

  public DBConnectionManager(String dbURL, String user, String pwd) throws
ClassNotFoundException, SQLException{
          Class.forName("com.mysql.jdbc.Driver");
          this.connection = DriverManager.getConnection(dbURL, user, pwd);
  }

  public Connection getConnection(){
          return this.connection;
  }
}
```

Servlet JDBC Example – Context Listener

AppContextListener.java is the servlet context listener implementation that will initialize the Database connection when application context is initialized and it also configures the log4j using it's configuration xml file. Notice the use of context init params for DB connection and log4j configuration.

When context will be destroyed, we are closing database connection in contextDestroyed() method.

Since we are using Servlet 3, we don't need to configure it in web.xml and we just need to annotate it with @WebListener annotation.

```
package com.journaldev.servlet.listeners;

import java.io.File;
import java.sql.Connection;
import java.sql.SQLException;

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

```
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.xml.DOMConfigurator;

import com.journaldev.util.DBConnectionManager;

@WebListener
public class AppContextListener implements ServletContextListener {

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

          //initialize DB Connection
          String dbURL = ctx.getInitParameter("dbURL");
          String user = ctx.getInitParameter("dbUser");
          String pwd = ctx.getInitParameter("dbPassword");

          try {
                  DBConnectionManager connectionManager = new DBConnectionManager(dbURL,
user, pwd);
                  ctx.setAttribute("DBConnection", connectionManager.getConnection());
                  System.out.println("DB Connection initialized successfully.");
          } catch (ClassNotFoundException e) {
                  e.printStackTrace();
          } catch (SQLException e) {
                  e.printStackTrace();
          }

          //initialize log4j
          String log4jConfig = ctx.getInitParameter("log4j-config");
          if(log4jConfig == null){
                  System.err.println("No log4j-config init param, initializing log4j with
BasicConfigurator");
                  BasicConfigurator.configure();
          }else {
                  String webAppPath = ctx.getRealPath("/");
                  String log4jProp = webAppPath + log4jConfig;
                  File log4jConfigFile = new File(log4jProp);
                  if (log4jConfigFile.exists()) {
                          System.out.println("Initializing log4j with: " + log4jProp);
                          DOMConfigurator.configure(log4jProp);
                  } else {
                          System.err.println(log4jProp + " file not found, initializing
log4j with BasicConfigurator");
                          BasicConfigurator.configure();
                  }
          }
          System.out.println("log4j configured properly");
    }

    public void contextDestroyed(ServletContextEvent servletContextEvent) {
          Connection con = (Connection)
servletContextEvent.getServletContext().getAttribute("DBConnection");
          try {
```

```
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

    }
```

Exception and Error Handler

AppErrorHandler.java is our application exception handler servlet configured in deployment descriptor, it provides useful information to the user incase of 404 errors or application level exceptions and provide them hyperlink to go to login page of application.

```
    package com.journaldev.servlet.errorhandler;

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

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

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

      protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
            processError(request, response);
      }

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

      private void processError(HttpServletRequest request,
                HttpServletResponse response) throws IOException {
            // Analyze the servlet exception
            Throwable throwable = (Throwable) request
                        .getAttribute("javax.servlet.error.exception");
            Integer statusCode = (Integer) request
                        .getAttribute("javax.servlet.error.status_code");
            String servletName = (String) request
                        .getAttribute("javax.servlet.error.servlet_name");
            if (servletName == null) {
                servletName = "Unknown";
            }
            String requestUri = (String) request
                        .getAttribute("javax.servlet.error.request_uri");
```

```
            if (requestUri == null) {
                    requestUri = "Unknown";
            }

            // Set response content type
          response.setContentType("text/html");

          PrintWriter out = response.getWriter();
          out.write("<html><head><title>Exception/Error Details</title></head><body>");
          if(statusCode != 500){
              out.write("<h3>Error Details</h3>");
              out.write("<strong>Status Code</strong>:"+statusCode+"<br>");
              out.write("<strong>Requested URI</strong>:"+requestUri);
          }else{
              out.write("<h3>Exception Details</h3>");
              out.write("<ul><li>Servlet Name:"+servletName+"</li>");
              out.write("<li>Exception Name:"+throwable.getClass().getName()+"</li>");
              out.write("<li>Requested URI:"+requestUri+"</li>");
              out.write("<li>Exception Message:"+throwable.getMessage()+"</li>");
              out.write("</ul>");
          }

          out.write("<br><br>");
          out.write("<a href=\"login.html\">Login Page</a>");
          out.write("</body></html>");
      }
    }
```

Servlet Filter

AuthenticationFilter.java is our Filter implementation and we are validating user session 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.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;

import org.apache.log4j.Logger;

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

  private Logger logger = Logger.getLogger(AuthenticationFilter.class);
```

```
  public void init(FilterConfig fConfig) throws ServletException {
          logger.info("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();
          logger.info("Requested Resource::"+uri);

          HttpSession session = req.getSession(false);

          if(session == null && !(uri.endsWith("html") || uri.endsWith("Login") ||
uri.endsWith("Register"))){
                  logger.error("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 the use of @WebFilter annotation, we can also provide URL Patterns for filter here but sometimes it's good to have in web.xml to easily disable the filters.

Servlet Classes

LoginServlet resource is used to validate the user input for login and forward them to home page or incase of missing data, provide useful information to user.

```
package com.journaldev.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.servlet.RequestDispatcher;
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;

import org.apache.log4j.Logger;

import com.journaldev.util.User;

@WebServlet(name = "Login", urlPatterns = { "/Login" })
public class LoginServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  static Logger logger = Logger.getLogger(LoginServlet.class);

  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
          String email = request.getParameter("email");
          String password = request.getParameter("password");
          String errorMsg = null;
          if(email == null || email.equals("")){
                  errorMsg ="User Email can't be null or empty";
          }
          if(password == null || password.equals("")){
                  errorMsg = "Password can't be null or empty";
          }

          if(errorMsg != null){
                  RequestDispatcher rd =
getServletContext().getRequestDispatcher("/login.html");
                  PrintWriter out= response.getWriter();
                  out.println("<font color=red>"+errorMsg+"</font>");
                  rd.include(request, response);
          }else{

          Connection con = (Connection) getServletContext().getAttribute("DBConnection");
          PreparedStatement ps = null;
          ResultSet rs = null;
          try {
                  ps = con.prepareStatement("select id, name, email,country from Users
where email=? and password=? limit 1");
                  ps.setString(1, email);
                  ps.setString(2, password);
                  rs = ps.executeQuery();

                  if(rs != null && rs.next()){

                          User user = new User(rs.getString("name"),
rs.getString("email"), rs.getString("country"), rs.getInt("id"));
                          logger.info("User found with details="+user);
                          HttpSession session = request.getSession();
                          session.setAttribute("User", user);
                          response.sendRedirect("home.jsp");;
```

```
                    }else{
                            RequestDispatcher rd =
        getServletContext().getRequestDispatcher("/login.html");
                            PrintWriter out= response.getWriter();
                            logger.error("User not found with email="+email);
                            out.println("<font color=red>No user found with given email id,
        please register first.</font>");
                            rd.include(request, response);
                    }
            } catch (SQLException e) {
                    e.printStackTrace();
                    logger.error("Database connection problem");
                    throw new ServletException("DB Connection problem.");
            }finally{
                    try {
                            rs.close();
                            ps.close();
                    } catch (SQLException e) {
                            logger.error("SQLException in closing PreparedStatement or
        ResultSet");;
                    }

            }
            }
      }

    }
```

LogoutServlet is simple and invalidates the user session and forward them to login page.

```
    package com.journaldev.servlet;

    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;

    import org.apache.log4j.Logger;

    @WebServlet(name = "Logout", urlPatterns = { "/Logout" })
    public class LogoutServlet extends HttpServlet {
      private static final long serialVersionUID = 1L;
      static Logger logger = Logger.getLogger(LogoutServlet.class);

        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")){
                            logger.info("JSESSIONID="+cookie.getValue());
                            break;
                    }
            }
            }
            //invalidate the session if exists
            HttpSession session = request.getSession(false);
            logger.info("User="+session.getAttribute("User"));
            if(session != null){
                    session.invalidate();
            }
            response.sendRedirect("login.html");
        }
    }
```

RegisterServlet is used by users to register to the application and then forward them to login page with registration success message.

```
    package com.journaldev.servlet;

    import java.io.IOException;
    import java.io.PrintWriter;
    import java.sql.Connection;
    import java.sql.PreparedStatement;
    import java.sql.SQLException;

    import javax.servlet.RequestDispatcher;
    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 org.apache.log4j.Logger;

    @WebServlet(name = "Register", urlPatterns = { "/Register" })
    public class RegisterServlet extends HttpServlet {
      private static final long serialVersionUID = 1L;

      static Logger logger = Logger.getLogger(RegisterServlet.class);

      protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
            String email = request.getParameter("email");
            String password = request.getParameter("password");
            String name = request.getParameter("name");
            String country = request.getParameter("country");
            String errorMsg = null;
            if(email == null || email.equals("")){
                    errorMsg = "Email ID can't be null or empty.";
            }
            if(password == null || password.equals("")){
```

```
                errorMsg = "Password can't be null or empty.";
        }
        if(name == null || name.equals("")){
                errorMsg = "Name can't be null or empty.";
        }
        if(country == null || country.equals("")){
                errorMsg = "Country can't be null or empty.";
        }

        if(errorMsg != null){
                RequestDispatcher rd =
getServletContext().getRequestDispatcher("/register.html");
                PrintWriter out= response.getWriter();
                out.println("<font color=red>"+errorMsg+"</font>");
                rd.include(request, response);
        }else{

        Connection con = (Connection) getServletContext().getAttribute("DBConnection");
        PreparedStatement ps = null;
        try {
                ps = con.prepareStatement("insert into Users(name,email,country,
password) values (?,?,?,?)");
                ps.setString(1, name);
                ps.setString(2, email);
                ps.setString(3, country);
                ps.setString(4, password);

                ps.execute();

                logger.info("User registered with email="+email);

                //forward to login page to login
                RequestDispatcher rd =
getServletContext().getRequestDispatcher("/login.html");
                PrintWriter out= response.getWriter();
                out.println("<font color=green>Registration successful, please login
below.</font>");
                rd.include(request, response);
        } catch (SQLException e) {
                e.printStackTrace();
                logger.error("Database connection problem");
                throw new ServletException("DB Connection problem.");
        }finally{
                try {
                        ps.close();
                } catch (SQLException e) {
                        logger.error("SQLException in closing PreparedStatement");
                }
        }
        }

  }
}
```

Home JSP Page

home.jsp is the home page for user after successful login, we are just showing some user information here and providing them option to logout.

home.jsp code:

```
<%@page import="com.journaldev.util.User"%>
<%@ 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>Home Page</title>
</head>
<body>
<%User user = (User) session.getAttribute("User"); %>
<h3>Hi <%=user.getName() %></h3>
<strong>Your Email</strong>: <%=user.getEmail() %><br>
<strong>Your Country</strong>: <%=user.getCountry() %><br>
<br>
<form action="Logout" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>
```

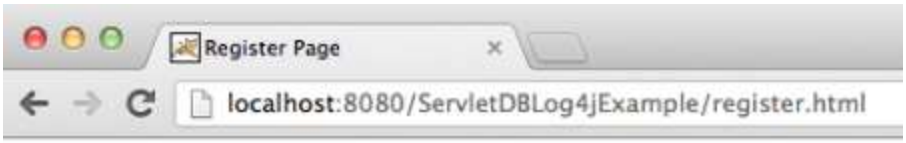
The JSP page still contains a lot of java code because we are not using JSP tags, we will look into this in JSP tutorials. As of now please bear with this.

Run the Servlet JDBC Example Application

Our application is ready for execution, I would suggest to export it as WAR file and then deploy to tomcat rather than deploying it directly to Tomcat server from Eclipse so that you can easily look into the log4j log file for debugging.

Some sample execution pages are shown in below images.

User Registration Page:



Provide all the fields for registration.

Email ID:
Password:
Name:
Country:
Register

If you are registered user, please login.

Registration Success Page:

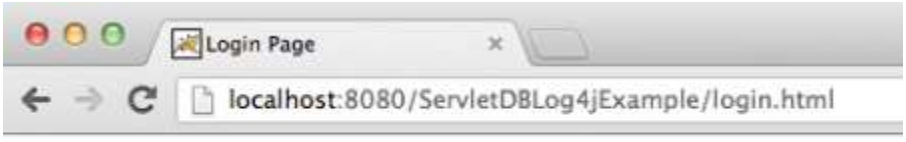
Registration successful, please login below.

Login with email and password

User Email:
Password:
Login

If you are new user, please register.

Login Page:

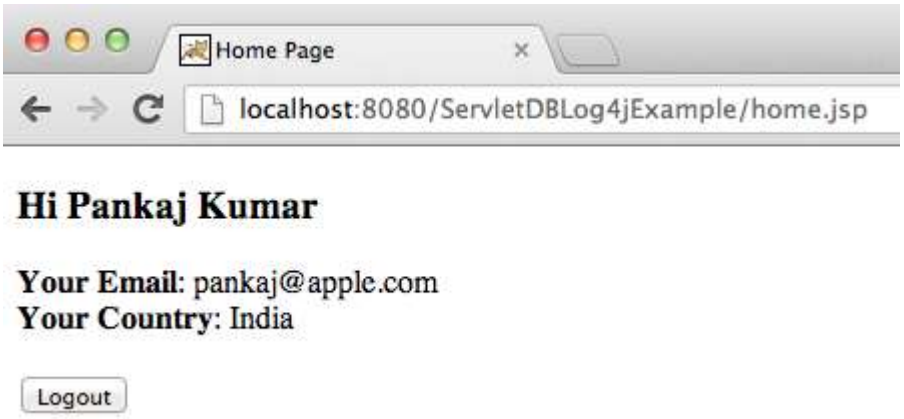


Login with email and password

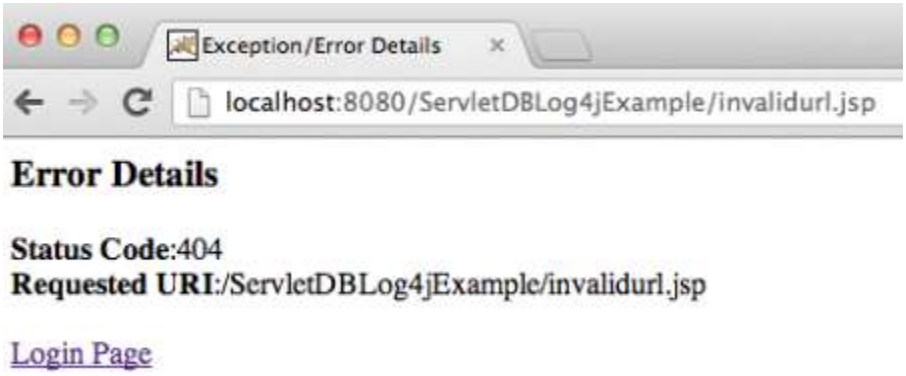
User Email:
Password:
Login

If you are new user, please register.

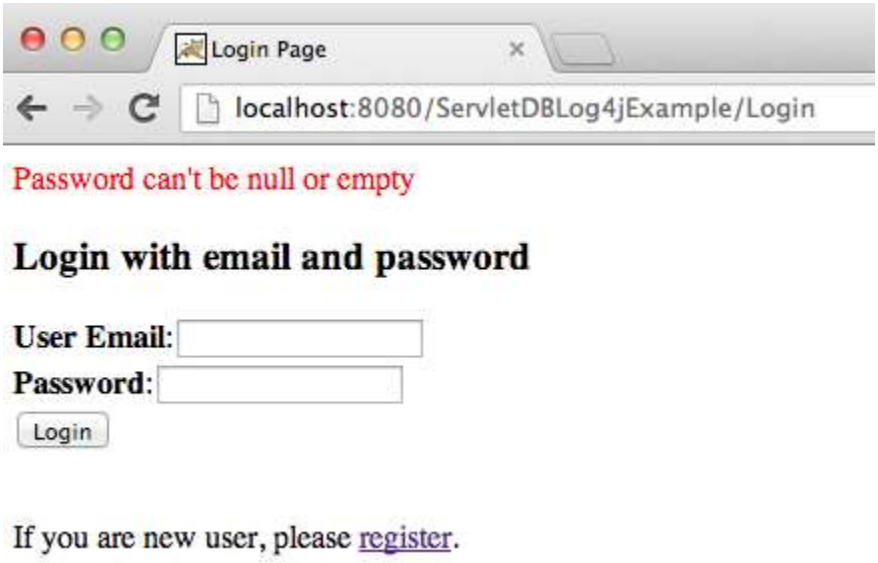
User Home Page:



404 Error Page:



Input Validation Error Page:



log4j Log File:

dbexample.log:

```
0    [localhost-startStop-1] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
AuthenticationFilter initialized
1    [localhost-startStop-1] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
AuthenticationFilter initialized
37689 [http-bio-8080-exec-3] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/
37689 [http-bio-8080-exec-3] ERROR com.journaldev.servlet.filters.AuthenticationFilter  -
Unauthorized access request
37693 [http-bio-8080-exec-4] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/login.html
51844 [http-bio-8080-exec-5] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/register.html
77818 [http-bio-8080-exec-7] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/Login
77835 [http-bio-8080-exec-7] INFO  com.journaldev.servlet.LoginServlet  - User found with
details=Name=Pankaj Kumar, Email=pankaj@apple.com, Country=India
77840 [http-bio-8080-exec-8] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/home.jsp
98251 [http-bio-8080-exec-9] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/Logout
98251 [http-bio-8080-exec-9] INFO  com.journaldev.servlet.LogoutServlet  -
JSESSIONID=367DE255789AC02F7C0E0298B825877C
98251 [http-bio-8080-exec-9] INFO  com.journaldev.servlet.LogoutServlet  - User=Name=Pankaj
Kumar, Email=pankaj@apple.com, Country=India
98254 [http-bio-8080-exec-10] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/login.html
109516 [http-bio-8080-exec-10] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/Register
109517 [http-bio-8080-exec-10] INFO  com.journaldev.servlet.RegisterServlet  - User
registered with email=abc@abc.com
127848 [http-bio-8080-exec-10] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/Login
223055 [http-bio-8080-exec-2] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/Login
223056 [http-bio-8080-exec-2] INFO  com.journaldev.servlet.LoginServlet  - User found with
details=Name=Pankaj Kumar, Email=pankaj@apple.com, Country=India
223059 [http-bio-8080-exec-2] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/home.jsp
231931 [http-bio-8080-exec-2] INFO  com.journaldev.servlet.filters.AuthenticationFilter  -
Requested Resource::/ServletDBLog4jExample/invalidurl.jsp
```

Download Resources

1. MySql Java Connector Jar
2. Log4j Jar

Servlet 3 File Upload

Since File Upload is a common task in web applications, Servlet Specs 3.0 provided additional support for uploading files to server and we don't have to depend on any third party APIs for this. In this tutorial we will see how we can use Servlet 3.0 API for uploading Files to server.

MultipartConfig

We need to annotate File Upload handler servlet with MultipartConfig annotation to handle multipart/form-data requests that is used for uploading file to server. MultipartConfig annotation has following attributes:

- **fileSizeThreshold**: We can specify the size threshold after which the file will be written to disk. The size value is in bytes, so 1024*1024*10 is 10 MB.
- **location**: Directory where files will be stored by default, it's default value is "".
- **maxFileSize**: Maximum size allowed to upload a file, it's value is provided in bytes. It's default value is -1L means unlimited.
- **maxRequestSize**: Maximum size allowed for multipart/form-data request. Default value is -1L that means unlimited.

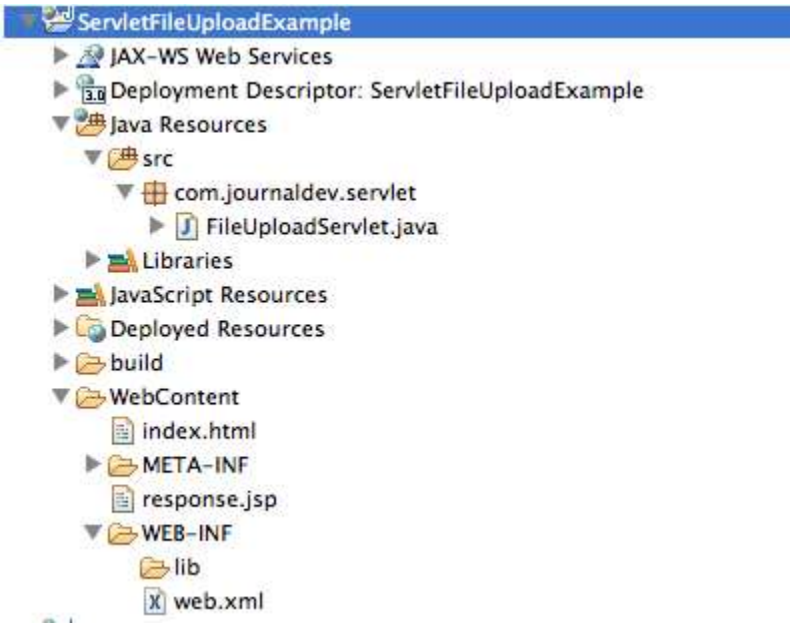
Part Interface

Part interface represents a part or form item that was received within a multipart/form-data POST request. Some important methods are `getInputStream()`, `write(String fileName)` that we can use to read and write file.

HttpServletRequest Changes

New methods got added in `HttpServletRequest` to get all the parts in multipart/form-data request through `getParts()` method. We can get a specific part using `getPart(String partName)` method.

Let's see a simple project where we will use above API methods to upload file using servlet. Our project structure will look like below image.



HTML Form

We have a simple html page where we can select the file to upload and submit request to server to get it uploaded.

`index.html`

```
<html>
<head></head>
<body>
<form action="FileUploadServlet" method="post" enctype="multipart/form-data">
Select File to Upload:<input type="file" name="fileName">
<br>
<input type="submit" value="Upload">
</form>
</body>
</html>
```

File Upload Servlet

Here is our File Upload Servlet implementation.

FileUploadServlet.java

```
package com.journaldev.servlet;

import java.io.File;
import java.io.IOException;

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

@WebServlet("/FileUploadServlet")
@MultipartConfig(fileSizeThreshold=1024*1024*10,    // 10 MB
                 maxFileSize=1024*1024*50,          // 50 MB
                 maxRequestSize=1024*1024*100)      // 100 MB
public class FileUploadServlet extends HttpServlet {

    private static final long serialVersionUID = 205242440643911308L;

    /**
     * Directory where uploaded files will be saved, its relative to
     * the web application directory.
     */
    private static final String UPLOAD_DIR = "uploads";

    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        // gets absolute path of the web application
        String applicationPath = request.getServletContext().getRealPath("");
        // constructs path of the directory to save uploaded file
        String uploadFilePath = applicationPath + File.separator + UPLOAD_DIR;

        // creates the save directory if it does not exists
        File fileSaveDir = new File(uploadFilePath);
        if (!fileSaveDir.exists()) {
```

```
            fileSaveDir.mkdirs();
        }
        System.out.println("Upload File Directory="+fileSaveDir.getAbsolutePath());

        String fileName = null;
        //Get all the parts from request and write it to the file on server
        for (Part part : request.getParts()) {
            fileName = getFileName(part);
            part.write(uploadFilePath + File.separator + fileName);
        }

        request.setAttribute("message", fileName + " File uploaded successfully!");
        getServletContext().getRequestDispatcher("/response.jsp").forward(
                request, response);
    }

    /**
     * Utility method to get file name from HTTP header content-disposition
     */
    private String getFileName(Part part) {
        String contentDisp = part.getHeader("content-disposition");
        System.out.println("content-disposition header= "+contentDisp);
        String[] tokens = contentDisp.split(";");
        for (String token : tokens) {
            if (token.trim().startsWith("filename")) {
                return token.substring(token.indexOf("=") + 2, token.length()-1);
            }
        }
        return "";
    }
}
```

Notice the use of @MultipartConfig annotation to specify different size parameters for upload file. We need to use request header “content-disposition” attribute to get the file name sent by client, we will save the file with same name.

The directory location is relative to web application where I am saving the file, you can configure it to some other location like in Apache Commons FileUpload example.

Response JSP

A simple JSP page that will be sent as response to client once the file is uploaded successfully to server.

`response.jsp`

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Upload File Response</title>
</head>
<body>
  <%-- Using JSP EL to get message attribute value from request scope --%>
```

```
        <h2>${requestScope.message}</h2>
    </body>
    </html>
```

Deployment Descriptor

There is nothing new in web.xml file for servlet file upload, it's only used to make the index.html as welcome file.

`web.xml`

```
<?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>ServletFileUploadExample</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Now when we run the application, we get following pages as response.

The logs will show the directory location where file is saved and content-disposition header information.

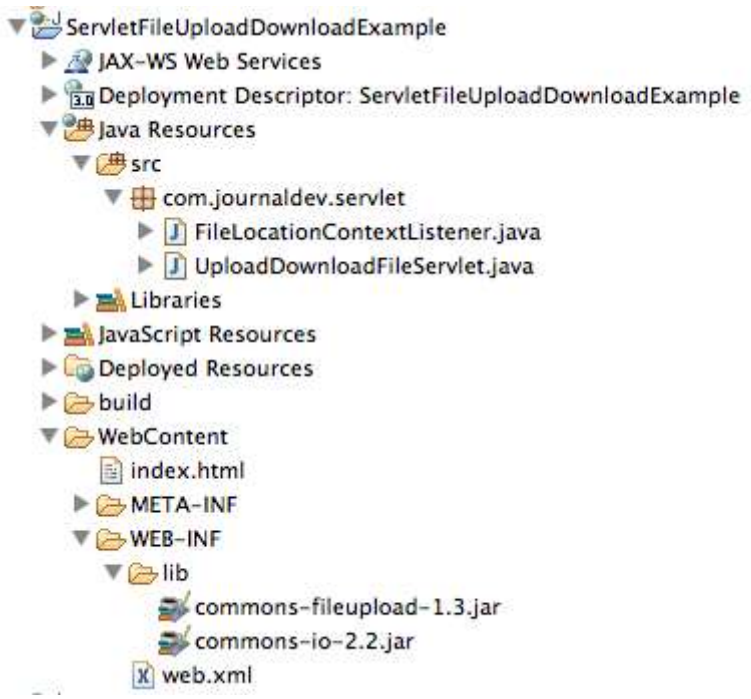
```
Upload File
Directory=/Users/pankaj/Documents/workspace/j2ee/.metadata/.plugins/org.eclipse.wst.server.
core/tmp0/wtpwebapps/ServletFileUploadExample/uploads
content-disposition header= form-data; name="fileName"; filename="IMG_2046.jpg"
```

Servlet Upload File and Download File Example

Servlet Upload File and Download File is a common task in java web application.

Servlet Upload File

Our use case is to provide a simple HTML page where client can select a local file to be uploaded to server. On submission of request to upload the file, our servlet program will upload the file into a directory in the server and then provide the URL through which user can download the file. For security reason, user will not be provided direct URL for downloading the file, rather they will be given a link to download the file and our servlet will process the request and send the file to user.



HTML Page for Java Uploading File to Server

We can upload a file to server by sending a post request to servlet and submitting the form. We can't use GET method for uploading file

Another point to note is that **enctype** of form should be **multipart/form-data**.

To select a file from user file system, we need to use **input** element with **type** as **file**.

So we can have a simple HTML page index.html for uploading file as:

```
<html>
<head></head>
<body>
<form action="UploadDownloadFileServlet" method="post" enctype="multipart/form-data">
Select File to Upload:<input type="file" name="fileName">
<br>
<input type="submit" value="Upload">
</form>
</body>
</html>
```

Server File Location for File Upload

We need to store file into some directory at server, we can have this directory hardcoded in program but for better flexibility, we will keep it configurable in deployment descriptor context params. Also we will add our upload file html page to the welcome file list.

Our web.xml file will look like below:

```
<?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>ServletFileUploadDownloadExample</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <context-param>
    <param-name>tempfile.dir</param-name>
    <param-value>tmpfiles</param-value>
  </context-param>
</web-app>
```

ServletContextListener for File Upload Location

Since we need to read context parameter for file location and create a File object from it, we can write a ServletContextListener to do it when context is initialized. We can set absolute directory location and File object as context attribute to be used by other servlets.

Our ServletContextListener implementation code is like below.

```
package com.journaldev.servlet;

import java.io.File;

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

@WebListener
public class FileLocationContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent servletContextEvent) {
          String rootPath = System.getProperty("catalina.home");
          ServletContext ctx = servletContextEvent.getServletContext();
          String relativePath = ctx.getInitParameter("tempfile.dir");
          File file = new File(rootPath + File.separator + relativePath);
          if(!file.exists()) file.mkdirs();
          System.out.println("File Directory created to be used for storing files");
          ctx.setAttribute("FILES_DIR_FILE", file);
          ctx.setAttribute("FILES_DIR", rootPath + File.separator + relativePath);
```

```
    }

  public void contextDestroyed(ServletContextEvent servletContextEvent) {
          //do cleanup if needed
  }

}
```

File Upload Download Servlet

Update: Servlet Specs 3 added support to upload files on server in the API, so we won't need to use any third party API.

For File upload, we will use **Apache Commons FileUpload** utility, for our project we are using version 1.3, FileUpload depends on **Apache Commons IO** jar, so we need to place both in the lib directory of the project, as you can see that in above image for project structure.

We will use **DiskFileItemFactory** factory that provides a method to parse the HttpServletRequest object and return list of **FileItem**. FileItem provides useful method to get the file name, field name in form, size and content type details of the file that needs to be uploaded. To write file to a directory, all we need to do it create a File object and pass it as argument to FileItem **write()** method.

Since the whole purpose of the servlet is to upload file, we will override init() method to initialise the `DiskFileItemFactory` object instance of the servlet. We will use this object in the doPost() method implementation to upload file to server directory.

Once the file gets uploaded successfully, we will send response to client with URL to download the file, since HTML links use GET method,we will append the parameter for file name in the URL and we can utilise the same servlet doGet() method to implement file download process.

For implementing download file servlet, first we will open the InputStream for the file and use **ServletContext.getMimeType()** method to get the MIME type of the file and set it as response content type.

We will also need to set the response content length as length of the file. To make sure that client understand that we are sending file in response, we need to set "**Content-Disposition**" header with value as **"attachment; filename="fileName"**.

Once we are done with setting response configuration, we can read file content from InputStream and write it to ServletOutputStream and the flush the output to client.

You can download Apache Commons IO jar and Apache Commons FileUpload jar from below URLs.

- http://commons.apache.org/proper/commons-fileupload/download_fileupload.cgi
- http://commons.apache.org/proper/commons-io/download_io.cgi

Our final implementation of UploadDownloadFileServlet servlet looks like below.

```
package com.journaldev.servlet;

import java.io.File;
import java.io.FileInputStream;
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.util.Iterator;
import java.util.List;

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

import org.apache.commons.fileupload.FileItem;
import org.apache.commons.fileupload.FileUploadException;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;

@WebServlet("/UploadDownloadFileServlet")
public class UploadDownloadFileServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;
    private ServletFileUpload uploader = null;
  @Override
  public void init() throws ServletException{
          DiskFileItemFactory fileFactory = new DiskFileItemFactory();
          File filesDir = (File) getServletContext().getAttribute("FILES_DIR_FILE");
          fileFactory.setRepository(filesDir);
          this.uploader = new ServletFileUpload(fileFactory);
  }
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
          String fileName = request.getParameter("fileName");
          if(fileName == null || fileName.equals("")){
                  throw new ServletException("File Name can't be null or empty");
          }
          File file = new
File(request.getServletContext().getAttribute("FILES_DIR")+File.separator+fileName);
          if(!file.exists()){
                  throw new ServletException("File doesn't exists on server.");
          }
          System.out.println("File location on server::"+file.getAbsolutePath());
          ServletContext ctx = getServletContext();
          InputStream fis = new FileInputStream(file);
          String mimeType = ctx.getMimeType(file.getAbsolutePath());
          response.setContentType(mimeType != null? mimeType:"application/octet-stream");
          response.setContentLength((int) file.length());
          response.setHeader("Content-Disposition", "attachment; filename=\"" + fileName +
"\"");

          ServletOutputStream os       = response.getOutputStream();
          byte[] bufferData = new byte[1024];
          int read=0;
```

```
          while((read = fis.read(bufferData))!= -1){
                  os.write(bufferData, 0, read);
          }
          os.flush();
          os.close();
          fis.close();
          System.out.println("File downloaded at client successfully");
  }

  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
          if(!ServletFileUpload.isMultipartContent(request)){
                  throw new ServletException("Content type is not multipart/form-data");
          }

          response.setContentType("text/html");
          PrintWriter out = response.getWriter();
          out.write("<html><head></head><body>");
          try {
                  List<FileItem> fileItemsList = uploader.parseRequest(request);
                  Iterator<FileItem> fileItemsIterator = fileItemsList.iterator();
                  while(fileItemsIterator.hasNext()){
                          FileItem fileItem = fileItemsIterator.next();
                          System.out.println("FieldName="+fileItem.getFieldName());
                          System.out.println("FileName="+fileItem.getName());
                          System.out.println("ContentType="+fileItem.getContentType());
                          System.out.println("Size in bytes="+fileItem.getSize());

                          File file = new
File(request.getServletContext().getAttribute("FILES_DIR")+File.separator+fileItem.getName(
));
                          System.out.println("Absolute Path at
server="+file.getAbsolutePath());
                          fileItem.write(file);
                          out.write("File "+fileItem.getName()+ " uploaded
successfully.");
                          out.write("<br>");
                          out.write("<a
href=\"UploadDownloadFileServlet?fileName="+fileItem.getName()+"\">Download
"+fileItem.getName()+"</a>");
                  }
          } catch (FileUploadException e) {
                  out.write("Exception in uploading file.");
          } catch (Exception e) {
                  out.write("Exception in uploading file.");
          }
          out.write("</body></html>");
  }

}
```

Async Servlet

Async servlet was introduced in Servlet 3. It's a great way to deal with thread starvation problem with long running threads.

Let's say we have a Servlet that takes a lot of time to process, something like below.

```
package com.journaldev.servlet;

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

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

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

  protected void doGet(HttpServletRequest request,
                  HttpServletResponse response) throws ServletException, IOException {
          long startTime = System.currentTimeMillis();
          System.out.println("LongRunningServlet Start::Name="
                          + Thread.currentThread().getName() + "::ID="
                          + Thread.currentThread().getId());

          String time = request.getParameter("time");
          int secs = Integer.valueOf(time);
          // max 10 seconds
          if (secs > 10000)
                  secs = 10000;

          longProcessing(secs);

          PrintWriter out = response.getWriter();
          long endTime = System.currentTimeMillis();
          out.write("Processing done for " + secs + " milliseconds!!");
          System.out.println("LongRunningServlet Start::Name="
                          + Thread.currentThread().getName() + "::ID="
                          + Thread.currentThread().getId() + "::Time Taken="
                          + (endTime - startTime) + " ms.");
  }

  private void longProcessing(int secs) {
          // wait for given time before finishing
          try {
                  Thread.sleep(secs);
          } catch (InterruptedException e) {
                  e.printStackTrace();
          }
  }

}
```

If we hit above servlet through browser with URL as `http://localhost:8080/AsyncServletExample/LongRunningServlet?time=8000`, we get response as "Processing done for 8000 milliseconds!!" after 8 seconds. Now if you will look into server logs, you will get following log:

```
LongRunningServlet Start::Name=http-bio-8080-exec-34::ID=103
LongRunningServlet Start::Name=http-bio-8080-exec-34::ID=103::Time Taken=8002 ms.
```

So our Servlet Thread was running for ~8+ seconds, although most of the processing has nothing to do with the servlet request or response.

This can lead to **Thread Starvation** – since our servlet thread is blocked until all the processing is done. If server gets a lot of requests to process, it will hit the maximum servlet thread limit and further requests will get Connection Refused errors.

Prior to Servlet 3.0, there were container specific solution for these long running threads where we can spawn a separate worker thread to do the heavy task and then return the response to client. The servlet thread returns to the servlet pool after starting the worker thread. Tomcat's Comet, WebLogic's FutureResponseServlet and WebSphere's Asynchronous Request Dispatcher are some of the example of implementation of asynchronous processing.

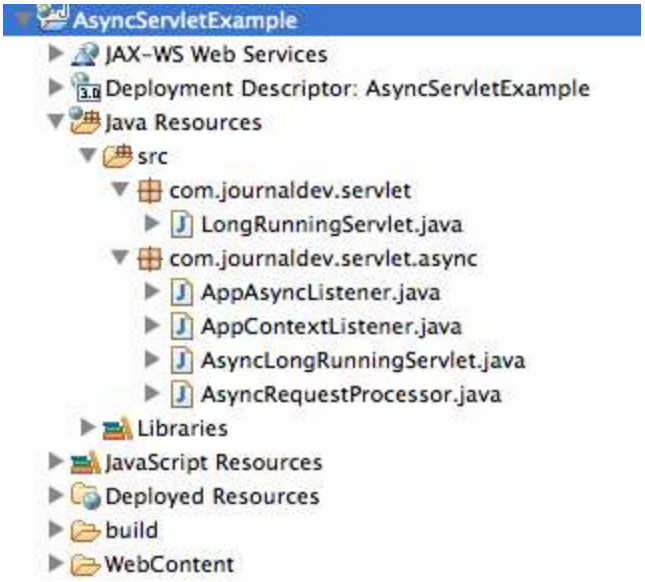
The problem with container specific solution is that we can't move to other servlet container without changing our application code, that's why Async Servlet support was added in Servlet 3.0 to provide standard way for asynchronous processing in servlets

Asynchronous Servlet Implementation

Let's see steps to implement async servlet and then we will provide async supported servlet for above example.

- First of all the servlet where we want to provide async support should have **@WebServlet** annotation with **asyncSupported** value as **true**.
- Since the actual work is to be delegated to another thread, we should have a thread pool implementation. We can create thread pool using Executors framework and use servlet context listener to initiate the thread pool.
- We need to get instance of **AsyncContext** through `ServletRequest.startAsync()` method. AsyncContext provides methods to get the ServletRequest and ServletResponse object references. It also provides method to forward the request to another resource using dispatch() method.
- We should have a Runnable implementation where we will do the heavy processing and then use AsyncContext object to either dispatch the request to another resource or write response using ServletResponse object. Once the processing is finished, we should call AsyncContext.complete() method to let container know that async processing is finished.
- We can add AsyncListener implementation to the AsyncContext object to implement callback methods – we can use this to provide error response to client incase of error or timeout while async thread processing. We can also do some cleanup activity here.

Once we will complete our project for Async servlet Example, it will look like below image.



Initializing Worker Thread Pool in Servlet Context Listener

```
package com.journaldev.servlet.async;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

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

@WebListener
public class AppContextListener implements ServletContextListener {

  public void contextInitialized(ServletContextEvent servletContextEvent) {

          // create the thread pool
          ThreadPoolExecutor executor = new ThreadPoolExecutor(100, 200, 50000L,
                          TimeUnit.MILLISECONDS, new ArrayBlockingQueue<Runnable>(100));
          servletContextEvent.getServletContext().setAttribute("executor",
                          executor);

  }

  public void contextDestroyed(ServletContextEvent servletContextEvent) {
          ThreadPoolExecutor executor = (ThreadPoolExecutor) servletContextEvent
                          .getServletContext().getAttribute("executor");
          executor.shutdown();
  }

}
```

Worker Thread Implementation

```
package com.journaldev.servlet.async;

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

import javax.servlet.AsyncContext;
```

```
public class AsyncRequestProcessor implements Runnable {

  private AsyncContext asyncContext;
  private int secs;

  public AsyncRequestProcessor() {
  }

  public AsyncRequestProcessor(AsyncContext asyncCtx, int secs) {
          this.asyncContext = asyncCtx;
          this.secs = secs;
  }

  @Override
  public void run() {
          System.out.println("Async Supported? "
                          + asyncContext.getRequest().isAsyncSupported());
          longProcessing(secs);
          try {
                  PrintWriter out = asyncContext.getResponse().getWriter();
                  out.write("Processing done for " + secs + " milliseconds!!");
          } catch (IOException e) {
                  e.printStackTrace();
          }
          //complete the processing
          asyncContext.complete();
  }

  private void longProcessing(int secs) {
          // wait for given time before finishing
          try {
                  Thread.sleep(secs);
          } catch (InterruptedException e) {
                  e.printStackTrace();
          }
  }
}
```

Notice the use of `AsyncContext` and it's usage in getting request and response objects and then completing the async processing with complete() method call.

AsyncListener Implementation

```
package com.journaldev.servlet.async;

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

import javax.servlet.AsyncEvent;
import javax.servlet.AsyncListener;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebListener;

@WebListener
public class AppAsyncListener implements AsyncListener {

  @Override
```

```
	public void onComplete(AsyncEvent asyncEvent) throws IOException {
		System.out.println("AppAsyncListener onComplete");
		// we can do resource cleanup activity here
	}

	@Override
	public void onError(AsyncEvent asyncEvent) throws IOException {
		System.out.println("AppAsyncListener onError");
		//we can return error response to client
	}

	@Override
	public void onStartAsync(AsyncEvent asyncEvent) throws IOException {
		System.out.println("AppAsyncListener onStartAsync");
		//we can log the event here
	}

	@Override
	public void onTimeout(AsyncEvent asyncEvent) throws IOException {
		System.out.println("AppAsyncListener onTimeout");
		//we can send appropriate response to client
		ServletResponse response = asyncEvent.getAsyncContext().getResponse();
		PrintWriter out = response.getWriter();
		out.write("TimeOut Error in Processing");
	}

}
```

Notice the implementation of `onTimeout()` method where we are sending timeout response to client.

Async Servlet Example implementation

Here is the implementation of our async servlet, notice the use of AsyncContext and ThreadPoolExecutor for processing.

```
package com.journaldev.servlet.async;

import java.io.IOException;
import java.util.concurrent.ThreadPoolExecutor;

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

@WebServlet(urlPatterns = "/AsyncLongRunningServlet", asyncSupported = true)
public class AsyncLongRunningServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;

	protected void doGet(HttpServletRequest request,
			HttpServletResponse response) throws ServletException, IOException {
		long startTime = System.currentTimeMillis();
		System.out.println("AsyncLongRunningServlet Start::Name="
				+ Thread.currentThread().getName() + "::ID="
				+ Thread.currentThread().getId());

		request.setAttribute("org.apache.catalina.ASYNC_SUPPORTED", true);

		String time = request.getParameter("time");
		int secs = Integer.valueOf(time);
```

```
            // max 10 seconds
            if (secs > 10000)
                    secs = 10000;

            AsyncContext asyncCtx = request.startAsync();
            asyncCtx.addListener(new AppAsyncListener());
            asyncCtx.setTimeout(9000);

            ThreadPoolExecutor executor = (ThreadPoolExecutor) request
                            .getServletContext().getAttribute("executor");

            executor.execute(new AsyncRequestProcessor(asyncCtx, secs));
            long endTime = System.currentTimeMillis();
            System.out.println("AsyncLongRunningServlet End::Name="
                            + Thread.currentThread().getName() + "::ID="
                            + Thread.currentThread().getId() + "::Time Taken="
                            + (endTime - startTime) + " ms.");
    }

}
```

Run Async Servlet web application

Now when we will run above servlet with URL as `http://localhost:8080/AsyncServletExample/AsyncLongRunningServlet?time=8000` we get the same response and logs as:

```
AsyncLongRunningServlet Start::Name=http-bio-8080-exec-50::ID=124
AsyncLongRunningServlet End::Name=http-bio-8080-exec-50::ID=124::Time Taken=1 ms.
Async Supported? true
AppAsyncListener onComplete
```

If we run with time as 9999, timeout occurs and we get response at client side as “TimeOut Error in Processing” and in logs:

```
AsyncLongRunningServlet Start::Name=http-bio-8080-exec-44::ID=117
AsyncLongRunningServlet End::Name=http-bio-8080-exec-44::ID=117::Time Taken=1 ms.
Async Supported? true
AppAsyncListener onTimeout
AppAsyncListener onError
AppAsyncListener onComplete
Exception in thread "pool-5-thread-6" java.lang.IllegalStateException: The request
associated with the AsyncContext has already completed processing.
  at org.apache.catalina.core.AsyncContextImpl.check(AsyncContextImpl.java:439)
  at org.apache.catalina.core.AsyncContextImpl.getResponse(AsyncContextImpl.java:197)
  at com.journaldev.servlet.async.AsyncRequestProcessor.run(AsyncRequestProcessor.java:27)
  at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
  at java.lang.Thread.run(Thread.java:680)
```

Notice that servlet thread finished execution quickly and all the major processing work is happening in other thread