Aprameyah
Technologies Pvt. Ltd.

Course Notes

JEE Web

- Maruthi R Janardhan
maruthi@aprameyah.com

Anatomy of HTTP

The Hypertext Transfer Protocol (HTTP) is an application protocol. Its a text based (non -binary) protocol meant for use over the web. Its the most common protocol supported on the web. It is stateless in nature. Stateless means that the client (which is usually the browser) does not maintain a consistent connection with the server. Every request is independent of the past request. A new connection is established, a request sent, response obtained and the connection is terminated. (There are some exceptions). So the server and the application has to device mechanisms outside of the protocol to establish a sense of conversation with the client.

Since http is a text based protocol, all data is transmitted back and forth in clear text. So security is quite weak. However with the addition of SSL, we get HTTP-Secure (https) that is now accepted as a reliable way to perform even high value banking transactions on the internet.

HTTP Can be used to transfer different kinds of content over it - text, html, images, audio, streaming video etc

HTTP Interaction

Http interaction consists of a request and a response. Requests and responses consist of headers + body. There are various types of request. The most common request: GET does not have a body. Only has headers. The most common response 200 has a body carrying data of the format specified in one of the response headers.

Some of the info that the request headers carry are as below:
Which client is sending the request (User-Agent)
What content types the client is willing to accept (Accept)
What host the request is meant for (Host)
Any cookies the client has stored for this host (Cookie)

Some of the information that the response headers carry are as below:
What type of data is being sent in the response body (Content-Type)
What encoding of data is used in the response content (Content-Encoding)
A response Code
Any cookies that the server wishes to store on client (Set-Cookie)

Firebug on firefox is a great way to inspect requests and responses.

HTTP Response Codes

When a client sends a HTTP request to the server, the server sends a response with a response code and a response message. This is the most important information. Some of the common response codes are:
200: Means everything went ok on the server and it returned a valid response. This is a protocol specific indication that things went ok. Cannot be used to indicate application specific error conditions such as incorrect data from user etc…Even error messages flow back in a 200 message

404: Not found - this means that the server did not find a file or a processing code for the specified path

302: Redirect - this is usually associated with a “location” header that tells the browser to send a repeat request to the url specified in the location header. This is used to send the user to another url. Example: if you visit google.com, it automatically sends the browser to google.co.in if you are from India.

500: Server error: Used to indicate various kinds of server errors. The associated message will usually have the information about the error.

401: Unauthorised: Used to indicate that the request is not authorised to access the resource identified in the url.

As a general rule, Any 2XX series response is successful, any 4XX and 5XX response codes are errors. For a full list of http response codes refer here: http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

GET Request

Get is the most common type of request. As the name suggests is used to retrieve resources from the server. It has to follow some rules. Get request cannot have a request body and hence cannot be used to send large amounts of data to the server. However some data can be sent to the server in the form of query string parameters. Any name value pairs placed after a ? symbol in an url is considered parameters

Another important aspect of GET requests is that the server should implement it to be idempotent. This basically means a certain GET request can be sent to the server any number of times without the that affecting the state of the server. Essentially you cannot update something in a database in response to a GET request. However the data thats returned for the request could change. The browser does not warn the user if user refreshes a page thats obtained using a GET request.

The parameters sent on GET request are displayed on the browser address bar and hence should not be used to send sensitive data such as passwords.

POST Request

POST requests are used to carry data to the server. These requests have a body. Please note, query strings sent in the URL are treated on par with name value pairs sent in the body of the POST request on a Java EE web server. The data carried in the body of the POST request is not visible to the user on the browser address bar. Please remember **that this does not make POST a secure request format**. Data is still sent on the wire in clear text and can be seen in browser debuggers, and wire monitors.

Post requests can be used to upload the contents of a file if needed. And hence should always specify a content type and content length to enable the server to process the request in a suitable manner.

Post requests are not navigable using the back button. Suppose your page is obtained by doing a POST request and you move forward from that page by clicking on some link. If you hit the back button the browser the browser pops up a warning saying that the document has expired and you have to send the request again. Sending the post request again is sometimes bad because it can re-do whatever operation it did on server (POST is not idempotent). This means if u added a record to database, it can be added again. If you submitted payment information on a credit card, it will get resubmitted (And probably recharged - if the server has not done enough to prevent it)

HTML Forms

Html forms allow you the user to input data. Example is below:

```
<form action="FormProcessing" method="post">
Name: <input type="text" name="name"/><br>
Address: <textarea rows="4" cols="100" name="address"></textarea><br>
<input type="submit" value="send"/>
</form>
```

Forms have an action attribute - url where the form data is submitted to
Forms have a method - either GET or POST to indicate how the data is sent to server
Forms have a submit button.
Default encoding of the form is application/x-www-form-urlencoded

Submit the above form in any server and watch the request headers and you will find something like this in the post body:

Content-Length 41
Content-Type application/x-www-form-urlencoded
name=Maruthi-Aprameyah&address=5th+avenue

Try putting in various types of characters such as @ &

You will see request being carried like this when I type Maruthi@$&:
name=Maruthi%40%24%26&address=5th+avenue

That encoding is called url encoding

Multipart Data

Multipart encoded forms are specified on the form tag like below:

```
<form action="MultipartHandler" method="post" enctype="multipart/form-data">
```

These forms can only be of type post. They do not encode the form data. In url encoded form data the presence of & indicates that it is separating parameters but not so in multipart data. The data could contain any character and no encoding is necessary and hence suitable for sending binary content like in case of a file upload.

The parameters are separated from one another using a boundary string. This is a random generated number and hypen characters. Shown below is an example of such random generated binary delimited upload data

----------------------------508265018186769340755462325O Content-Disposition: form-data; name="name" Maruthi
----------------------------508265018186769340755462325O Content-Disposition: form-data; name="address" 5th Avenue
----------------------------508265018186769340755462325O Content-Disposition: form-data; name="photo"; filename="SmallImage.jpg" Content-Type: image/jpeg ÿØÿàJFIFHHÿâXICC_PROFILE
————————————————508265018186769340755462325O--

it is called multipart data because as you can see the request has multiple parts and each part can have an independent content type!

JEE Spec

Java Enterprise Edition: Is a set of specifications, libraries and apis that is designed by an industry consortium led by Oracle. These specifications help in providing infrastructure layer services for enterprise applications. JEE Spec includes the following components:

EJB - Enterprise Java Beans spec provides services such as transaction management, pool management, security and dependency setup for business components
Servlet - The servlet spec allows us to write application code that handles http request and responses
JSP - Java server pages enables us to write html user interfaces that are dynamic in nature on the server side
JSF - Java server faces - is a specification thats built on top of Servlets and JSPs to provide a component based approach similar to Java Swings to building web based user interfaces
JTA - Java Transaction Api allows us to manage distributed and abstracted transactions over any underlying transactional resources such as databases.
JPA - Java Persistence Architecture - Allows us to write easy to handle persistence code based on object relational mapping
CDI - Contexts and Dependency Injection - Provides ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle context. The ability to inject components into an application in a typesafe way, including the ability to choose at deployment time which implementation of a particular interface to inject
JAX-WS/RS - Java API for XML WebServices/Restful Webservices - provides all necessary tools and APIs to build web services
JASPIC - Java Authentication Service Provider Interface for Containers - it is an SPI for authentication mechanism providers to integrate with JEE servers

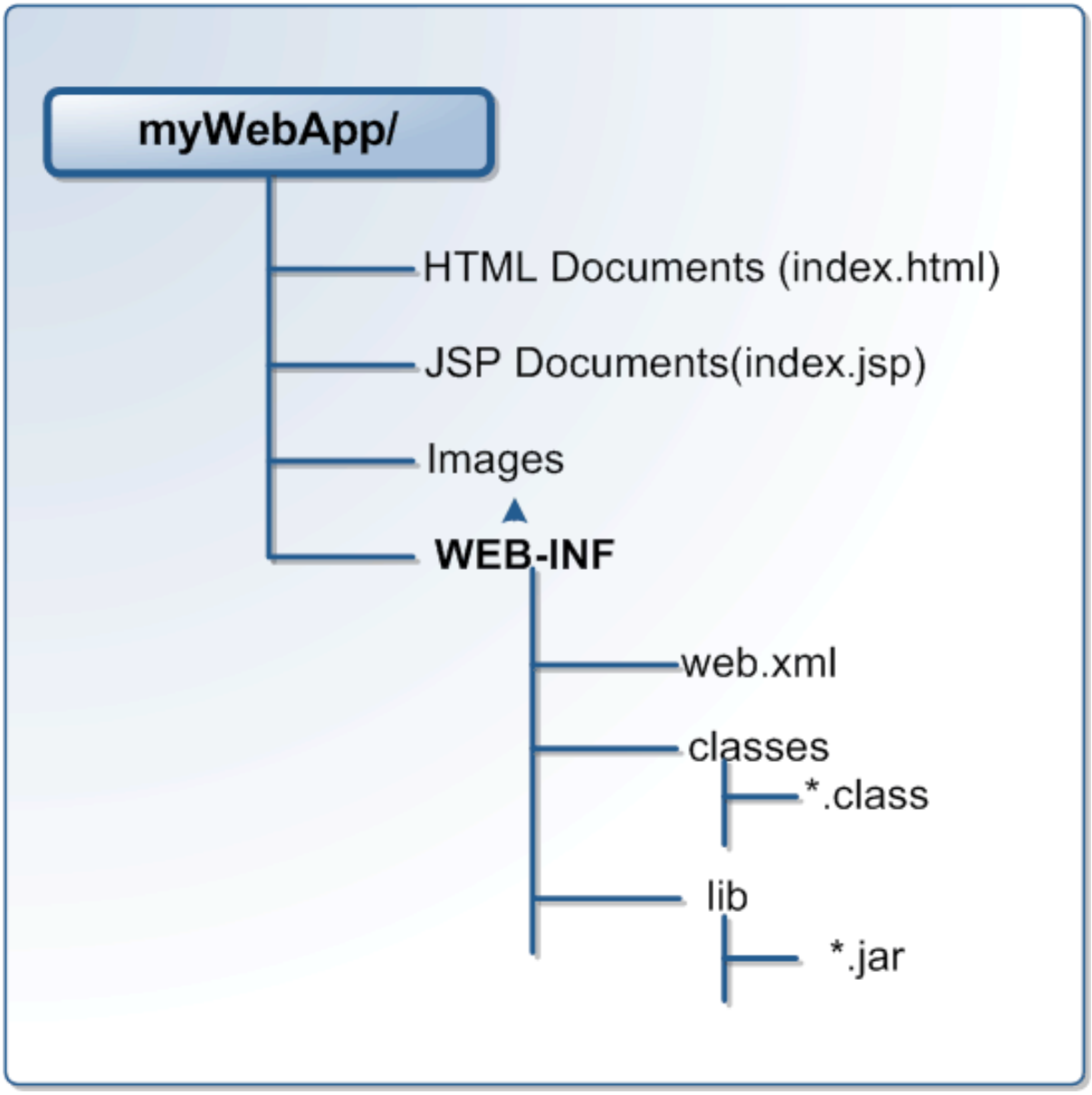
JAXB - Java Architecture for XML Binding - transform between Java objects and XML
JSR - 77 - Java spec request relating to provides a platform api for managing JEE app servers
JAXR - Java API for XML registries - API to access XML registries such as UDDI, ebXML Registry
JMS - Java messaging service
JCA - Java connector architecture - allows any resource to be accessible similar to databases - a transactional resource.
SAAJ - Soap With Attachments API for java
JAAS - Java Authentication and Authorisation Service - An api spec for common security algorithms and authentication mechanisms
JNDI - Java Naming and Directory Interface - a registry of objects and resources within the application server
JSR 88 - Provides an api for creating platform independent deployment tools

Servlet

```
@WebServlet("/FirstServlet")
public class FirstServlet extends HttpServlet {
	protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
		String name = request.getParameter("name");
		response.getOutputStream().print("Hello "+name);
	}
}
```

Our first hello servlet is above. Its passed two objects as parameters. One represents the request information and another contains methods to control the response from the server. The request object can be used to read parameters, headers and other information from the incoming request. Response object can provide an output stream to write to.
The @WebServlet annotation identifies this class as a servlet and also specifies the url pattern matching. More on that later.

WAR Deployment Package



For the application server to use our servlet, it needs to be packaged in a certain folder structure and zipped up. The zip file should have an extension of .war similar to a .jar
The folder structure is as below. Note the location of various files. All the static files that are needed by the web application such as HTML, CSS, Scripts etc goes directly under the root of the archive. They could contain folders such as “scripts”, “images” etc and the respective resources will be accessible on the url using these folder paths. Same applies to JSP files as we learn them later on.

As of JEE 6, the deployment descriptor web.xml is optional and its contents are now represented via annotations. Lib folder is used to keep any dependent jar files needed for the web application.
The package folder structure of the java classes has to go under the WEB-INF/classes folder.

When this archive is deployed into an app server, most app servers typically expand this archive into an exploded folder structure and read the files.

Files inside the WEB-INF are not accessible on the url

The name of the web archive will be the context root. For example if the web archive is called myWebApp.war then the url of index.html will be (Assuming the application is deployed locally on port 8080):
http://localhost:8080/myWebApp/index.html

There are eclipse built in features that allow for automation of creating this archive for us.

Handling Form Data

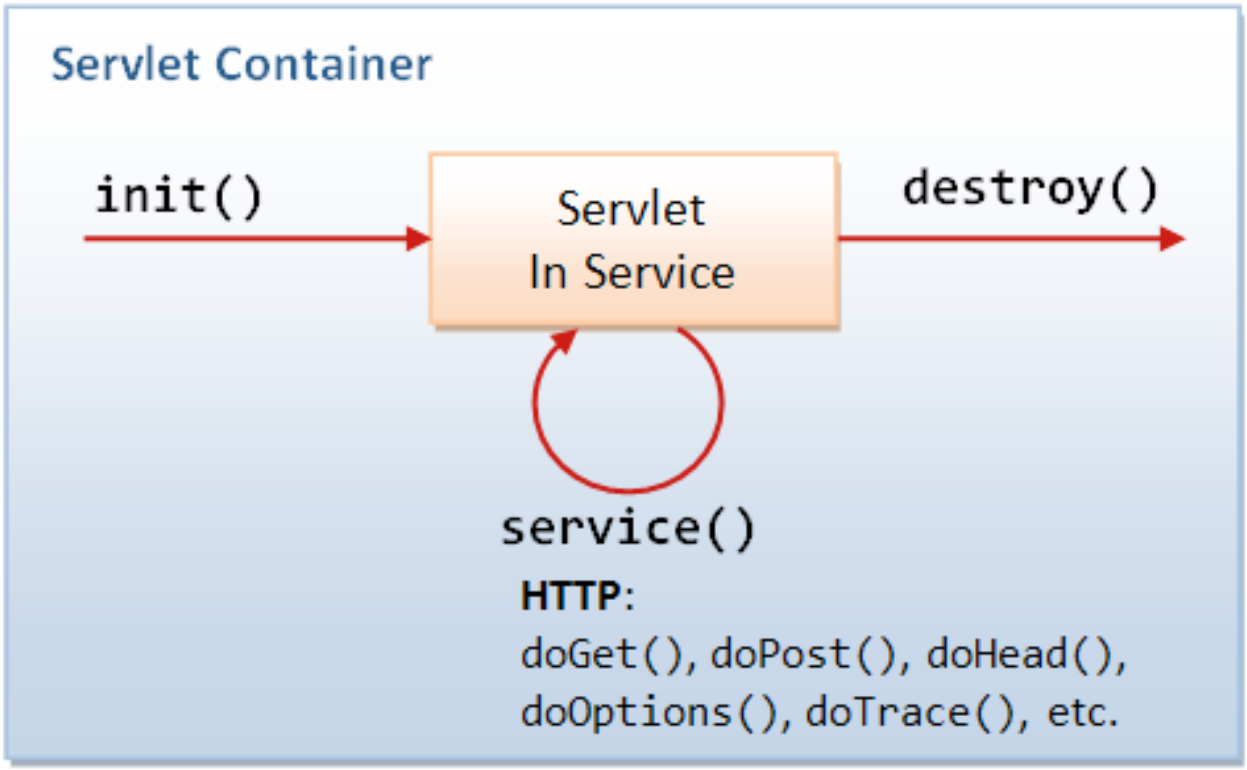
The the form that we created earlier could be processed with a servlet. All the api thats needed to access the form fields are present on the request object. Note the use of forwarding of the request to be handled by a html page. This is a way of delegating further processing to another html page. This way we can write complex view layouts in a html file rather than in code.

```
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
            Enumeration<String> paramNames = request.getParameterNames();
            while(paramNames.hasMoreElements()){
                  String paramName = paramNames.nextElement();
                  System.out.println("Parameter: "+paramName+" =
"+request.getParameter(paramName));
            }
            request.getRequestDispatcher("thankyou.html").forward(request,
response);
      }
```

Life Of A Servlet

Servlet class instance is not created by us but by the container. The servlet is instantiated and init method called soon after. Servlets are singletons by default in a container. There is just one instance of the servlet that serves multiple clients at the same time using different threads. The

servlet's base class has a "Service" method that in turn calls the doGet or doPost depending on the type of the request.
When the servlet is removed from the memory, destroy() method gets called. This typically happens when the server is shutting down or the application is being undeployed.



Init Params & DD

Servlets can define initialisation parameters for the servlet. This can be specified using annotations as shown below. Alternatively we can define what is known as a deployment descriptor (DD). The DD is a file called web.xml placed in the WEB-INF directory. Shown below is a sample way of declaring the same servlet without annotations but with a DD. Init parameters would probably make sense being put in an xml config file like the DD rather than in code as annotations. Covering this as this is part of the spec but I feel the best way to specify configuration options would be to use property bundles. That way the configuration is not scattered in a dozen different classes or xml.

```
@WebServlet(
            urlPatterns = { "/LifeCycleServlet" },
            initParams = {
```

```
                        @WebInitParam(name = "plancks_constant", value =
"6.62606957 × 10-34", description = "Plancks constant in string format")
            })
public class LifeCycleServlet extends HttpServlet {

      public void init(ServletConfig config) throws ServletException {
            System.out.println("Planck's constant is:
"+getInitParameter("plancks_constant"));
      }

      public void destroy() {
            System.out.println("Done with the short life.. time to go!!");
      }

      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            System.out.println("Hello there! Good you thought of me!");
      }

}
```

```
<web-app 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"
      version="3.0">

      <display-name>Servlet 3.0 Web Application</display-name>
      <servlet>
            <servlet-name>someservlet</servlet-name>
            <servlet-class>com.mydomai.servlets.LifeCycleServlet</servlet-
class>
            <init-param>
                  <param-name>plancks_constant</param-name>
                  <param-value>6.62606957 × 10-34</param-value>
            </init-param>
      </servlet>
      <servlet-mapping>
            <servlet-name>someservlet</servlet-name>
            <url-pattern>/LifeCycleServlet</url-pattern>
      </servlet-mapping>
</web-app>
```

Threading In Servlets

Try the below servlet and access it in multi user mode and see what happens:

```
@WebServlet("/ThreadingServlet")
public class ThreadingServlet extends HttpServlet {
      private String name ="";

      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            name = request.getParameter("name");
            for (int i = 0; i< 1000; i++){
                  //Just a dummy loop to represent some business logic
                  System.out.println("Thread:
"+Thread.currentThread().getName());
            }
            response.getWriter().print("Hello "+name);
      }

}
```

Lets Build A Wizard

A wizard is a series of pages that collects information from the user. The server has to create kind of a conversational state over a stateless http protocol for this to work. Here is one example of how this can be done:

```
@WebServlet("/WizardServlet1")
public class WizardServlet1 extends HttpServlet {
      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            String name = request.getParameter("name");
            String address = request.getParameter("address");
```

```
            String respStr = "<form action='WizardServlet2'><input
type='hidden' name='name' value='"+name+"'/><input type='hidden'
name='address' value='"+address+"'/>Phone: <input type='text'
name='phone'/><br>Email: <input type='text' name='email'/><br><input
type='submit' value='finish'/></form>";
            response.getWriter().print(respStr);
      }
}
```

```
@WebServlet("/WizardServlet2")
public class WizardServlet2 extends HttpServlet {
      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            String name = request.getParameter("name");
            String address = request.getParameter("address");
            String phone = request.getParameter("phone");
            String email = request.getParameter("email");
            System.out.println("Name: "+name+" Email: "+email);
            request.getRequestDispatcher("thankyou.html").forward(request,
response);
      }
}
```

here we are using the hidden fields in the html form to carry information thru the wizard for us. So essentially the state is maintained on the browser.

Redo the Wizards

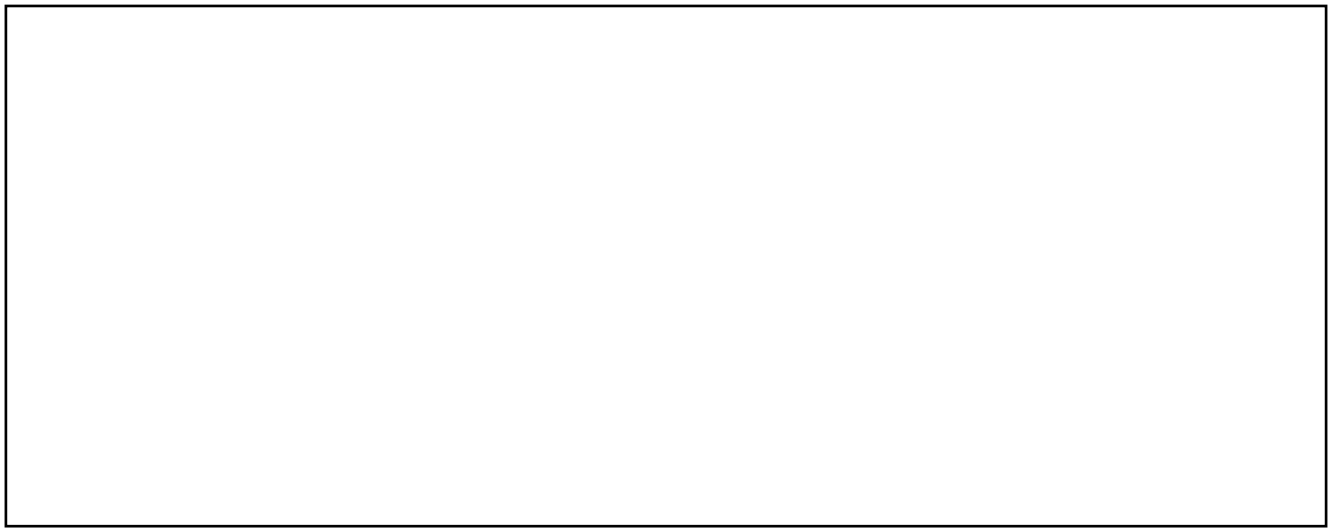
Lets use session for this now. Session is an object of type HttpSession. This class is part of the servlet api and represents operations on a session. Session is like a multilevel map. Each submap is associated with a session id. A session id is persisted to the browser client using a cookie. Once a cookie is set on a browser, it keeps returning that cookie every time there is a request back to the same server.
Code for the revised session based wizard is as follows:

```
@WebServlet("/WizardSessionServlet1")
public class WizardSessionServlet1 extends HttpServlet {
      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
```

```
            String name = request.getParameter("name");
            String address = request.getParameter("address");
            HttpSession session = request.getSession();
            session.setAttribute("name",name);
            session.setAttribute("address",address);
            String respStr = "<form action='WizardSessionServlet2'>Phone:
<input type='text' name='phone'/><br>Email: <input type='text'
name='email'/><br><input type='submit' value='finish'/></form>";
            response.getWriter().print(respStr);
      }
}
```

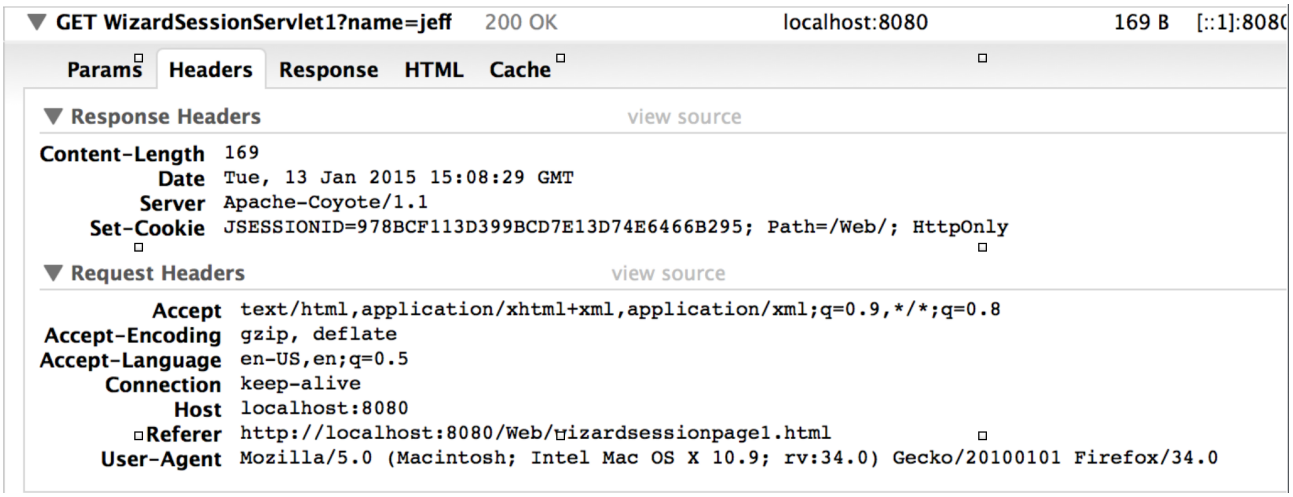
```
@WebServlet("/WizardServlet2")
public class WizardSessionServlet2 extends HttpServlet {
      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            HttpSession session = request.getSession();
            String name = (String)session.getAttribute("name");
            String address = (String)session.getAttribute("address");
            String phone = request.getParameter("phone");
            String email = request.getParameter("email");
            System.out.println("Name: "+name+" Email: "+email);
            request.getRequestDispatcher("thankyou.html").forward(request,
response);
      }
}
```

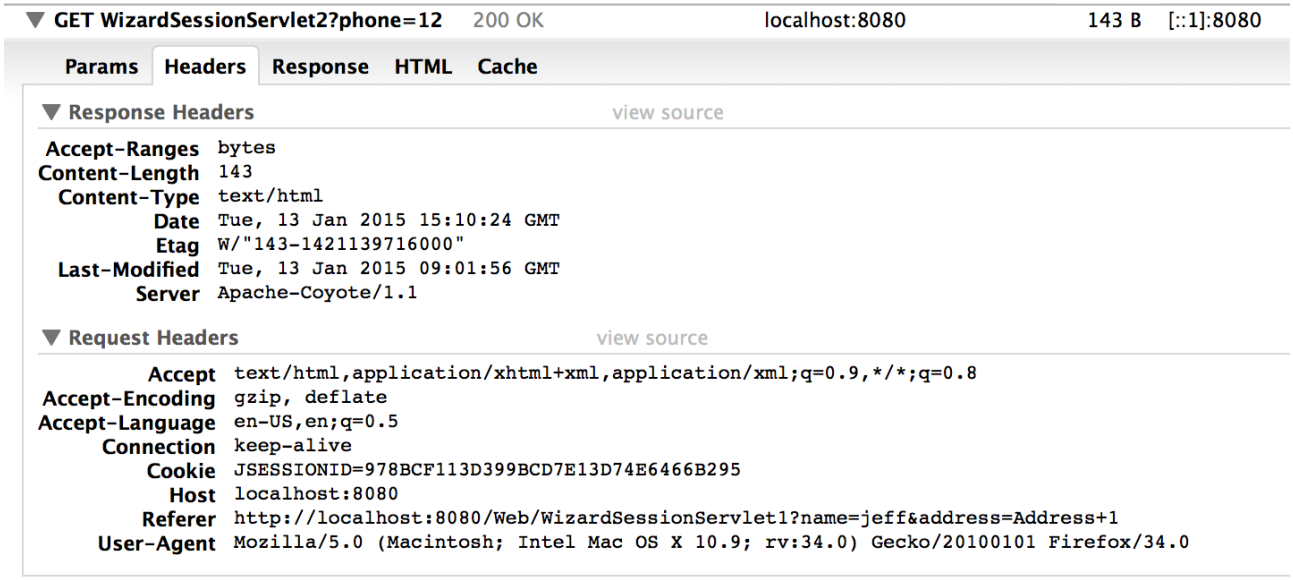
Notice that the hidden fields are no longer used. The second server retrieve the data from the session. This is a good way of maintaining state over the stateless http. Keep in mind that there is one session object per logged in user. Rather more technically - one session per open browser instance. So putting too much data in the session is considered bad programming practice since it can bloat the server memory very soon. All open tabs on a tabbed browser share the same session cookie.



Yummy Cookies

When we first access session using request.getSession() call, there is a set-cookie header present in the corresponding response. And every subsequent request will carry the cookie back to the server. The id in the cookie is used to locate the correct session data





Send Down Files

Response output stream can be used to pump out any kind of binary stream back to the browser as shown in this snippet of code. However it is recommended that this approach be used only for data files such as XML or JSON that is generated in the servlet. If the data that is to be sent back is in a physical file on the server's filesystem, it is best to redirect the user to the said file location with some auth checks. Else we are loading the server JVM with heavy stream operations.

```
File f = new File("filename");
        FileInputStream fis = new FileInputStream(f);
        OutputStream os = response.getOutputStream();
        byte[] data = new byte[(int)f.length()];
        fis.read(data);
        response.setContentType("application/pdf");
        os.write(data);
        fis.close();
```

Code Restrictions

There are many code restrictions that are to be followed while coding servlets. The compiler wont complain and the runtime system wont complain either, but are coding practices that lead to problems in server side environments.

1. Do not start threads from servlets because starting threads in an app server is to be avoided unless you know what you are doing. Typically app server uses a thread pool to invoke the servlet from. Any threads that we start from a servlet can outlive the servlet request and hence can interfere with server's life cycle. This thread is not under the app server's control
2. Do not create server sockets. Creating a server socket in a servlet locks up the request processing thread and hence can mess with thread pool management of app server. Any services that the server wants to start should be done in JEE recommended ways.
3. Servlets are often deployed in clustered environments where you cannot make an assumption that same physical machine is available at all times to write and read files from. So anything u persist to a file may not be available later
4. Dont use absolute path names to access filesystem resources on the server. This is because the actual deployment location could change depending on the server implementation and also can be moved around due to production system configuration changes.
5. Dont use JNI or any other native api to program in a servlet. This is because we cant make any assumption about the platform on which the server app server is deployed. Could be Linux, Windows, Mac.

Special Situation 1

There are times when we wish to run some initialisation code as soon as the app starts up. Such as clearing up some table or reading some startup settings etc. This can be done in web apps in two ways. One is using a dedicated servlet for this with a loadonstartup annotation. This servlet is loaded even without a request sent to it and it calls the init method.

@WebServlet(urlPatterns = "/LoadOnStartupServlet", loadOnStartup = 1)

The other option (and a recommended approach) is to write a servlet context listener. There are many lifecycle events in the webcontainer. And different listeners to handle them. ServletContextListener notifies the listener when we start and stop the server.

@WebListener
public class EventProcessor implements ServletContextListener

Other Listeners

There are various other listeners in the web container. The interface names and method names are self explanatory. Among these, the session activation/passivation is a concept that means that these events will occur when sessions are transferred in a clustered environment between different VMs and also in case of persistent sessions where session is stored to disk when the app server is being rebooted. This is a reason why session data should always be serializable.

```
HttpSessionListener
      sessionCreated
      sessionDestroyed
HttpSessionActivationListener
      sessionDidActivate
      sessionWillPassivate
```

```
HttpSessionAttributeListener
      attributeAdded
      attributeRemoved
      attributeReplaced
ServletRequestListener
      requestInitialized
      requestDestroyed

ServletRequestAttributeListener
      attributeAdded
      attributeRemoved
      attributeReplaced
```

How Clusters Work

One of the things that has to always be kept in mind is that our server has to run in a clustered environment. This is a thought that has to run thru in mind before any new code or design changes to an app. Clustering means more than one instance of the application server will run the same application war file which has the same servlets on two different physical boxes usually.
A load balancing web server such as apache is used as the front end for a cluster. This web server gets all requests and decides to send the request further on to one of the many app servers it is hooked up to.

Sessions are treated in two different ways. Sticky session is a configuration where an user once served by one instance of the app server is always sent to the same instance. This will ensure that there is no need to transfer session objects between app servers for replication. This is the most commonly used configuration. However most app servers support session replication and one can choose not to have sticky sessions.

The rules we saw about not creating a socket, not creating a thread, not accessing a file system resource etc continues to hold here. Also note that there is a single servlet context object per instance of the app server. Servlet contexts are not replicated over. So whatever is stored in a servlet context or in a static variable is only available to that particular server instance.

Java Server Pages

As you figured by now, servlets are a tedious place to put serious html. Also mixes up logic and presentation of data into a single class. To tide over this, we have the concept of JSPs. Servlets are about code and logic, JSPs are about presentation. The servlet technology is extended to form the JSP specification.
Another reason for JSPs to evolve is that JSPs are designer friendly. Web designers are not programmers, they will not be comfortable with dealing with java code and they need a page which is more of html to deal with.

JSP Elements

There are 4 main kind of elements you can use in a JSP apart from the custom tags.
Scriptlets are the elements that are enclosed in <% %>. It can carry any valid java code that can compile inside of the method body of a doGet/doPost method. Usually scriptlets are to be avoided in JSP code as that mixes up presentation html code with java.

Declarations are the second kind of element enclosed in <%! %> tags. The content of these tags are copied into the class body of a generated servlet. So this can be used to declare methods or class level variables in the JSP class.

Directives are wrapped in <%@ %> tags and are meant to provide instructions to the JSP compiler about how the servlet should be generated. For example something like this says we have to put an import statement in the generated code: <%@ page import="java.util.*" %>. <%@ include file="hello.jsp" %> says include the content of Hello.jsp in this place in generated code.

The fourth kind is the expressions. They are enclosed in <%= %> tags and its contents are placed in out.print statements. For example this: The time is now <%= new java.util.Date() %> will cause the date value to be placed in place of the expression

JSP Implicit Objects

Since in JSPs we do not write the class ourselves and we rely on the JSP translator to put our code in a generated servlet class, we do not have the freedom to name the variables and objects of the class. They are consistently named and are called implicit objects.

There are a few implicit objects for example the request and response objects that are passed to us in a doGet method call are called “request” and “response” respectively. Thats the name the generated code will have for those variables basically.

request - This is the HttpServletRequest object associated with the request.
response - This is the HttpServletResponse object associated with the response to the client.
out - This is the PrintWriter object used to send output to the client.
session - This is the HttpSession object associated with the request.
application - This is the ServletContext object associated with application context.
config - This is the ServletConfig object associated with the page.
pageContext - This encapsulates use of server-specific features like higher performance JspWriters.
page - This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class.
Exception - The Exception object allows the exception data to be accessed by designated JSP.

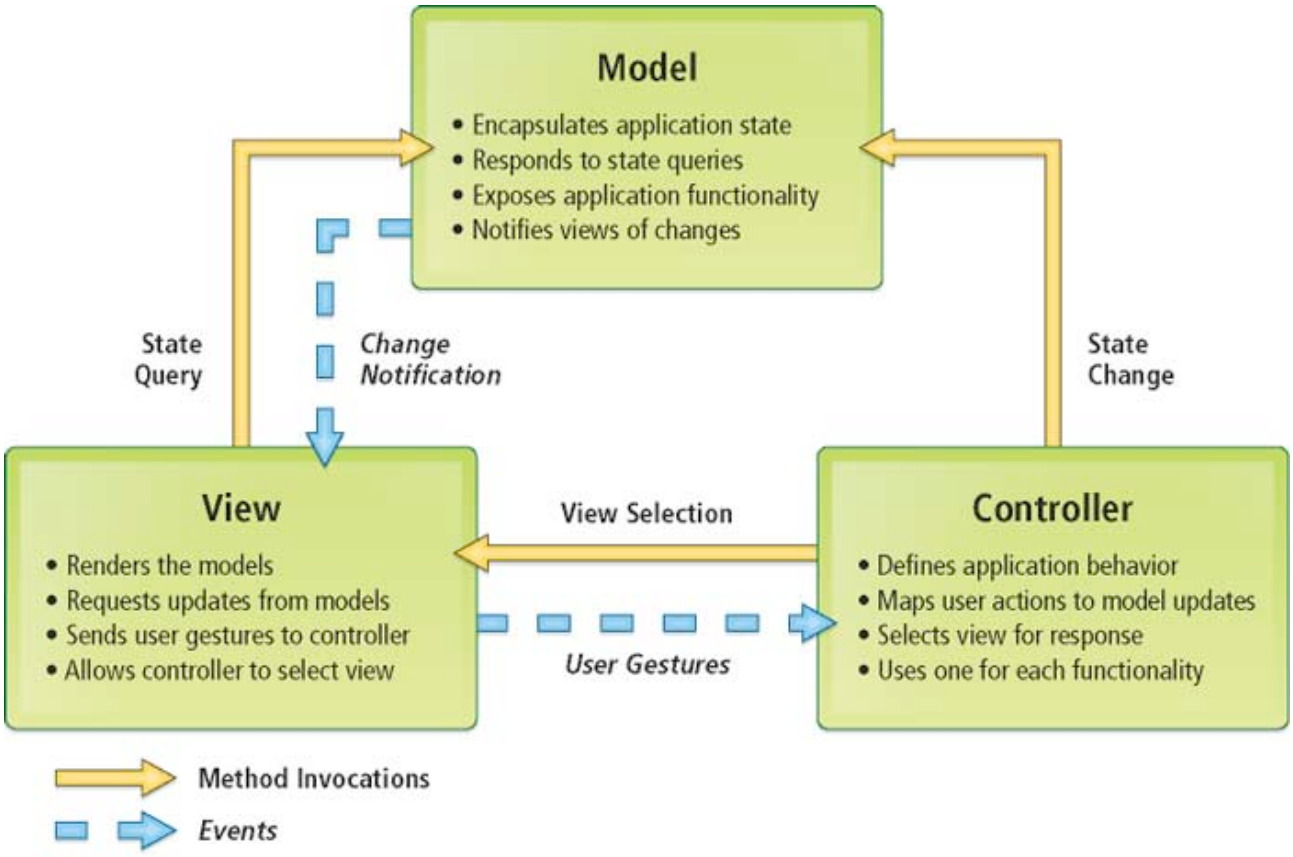
The Page Directive

Page directive is used like this as an example: <%@ page import=“java.uti.Date” %>. As you can see import is an attribute of the page directive. There are many attributes that can be used on a page directive and they are detailed below.

Attribute	Purpose
buffer	Specifies a buffering model for the output stream.
autoFlush	Controls the behavior of the servlet output buffer.
contentType	Defines the character encoding scheme.
errorPage	Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
isErrorPage	Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.

Attribute	Purpose
extends	Specifies a superclass that the generated servlet must extend
import	Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
info	Defines a string that can be accessed with the servlet's getServletInfo() method.
isThreadSafe	Defines the threading model for the generated servlet.
language	Defines the programming language used in the JSP page.
session	Specifies whether or not the JSP page participates in HTTP sessions
isELIgnored	Specifies whether or not EL expression within the JSP page will be ignored.
isScriptingEnabled	Determines if scripting elements are allowed for use.

Model View Controller



Model-View-Controller is a design pattern where the responsibilities are divided in a user interface based application. In servlet-JSP backend, servlets act as controllers, JSPs as views and Java beans as models.

Passing Data Controller->View

As per the MVC design, the servlet is meant to perform controller role and it could load data thats meant to be displayed on the JSP. To accomplish this we will put the data in the request object as an attribute and forward the request to a JSP. JSP accesses the attribute in the request object and renders the data (Note that the JSP is free to access session attributes too in the same way).

```
request.setAttribute("name", name);
request.getRequestDispatcher("thankyou.jsp").forward(request, response);

AND IN JSP

Thank you for submitting your data <%=request.getAttribute("name") %>
```

Build An MVC App

Now we have enough knowledge of the servlet asp platform to put it to some kind of descent usage. Lets build a little CRUD (Create Read Update Delete) App.
Here is what we will need to build.
1. We will need a table. Please create one using this statement on your database server

```
CREATE TABLE `user` (
  `id` int(10) unsigned NOT NULL,
  `name` varchar(100) NOT NULL,
  `email_id` varchar(100) NOT NULL,
  `password` varchar(100) NOT NULL,
 `join_date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
 `age` int(10) unsigned NOT NULL,
 `state` varchar(2) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2. We will need a User class to hold the fields of this table. Create it as a java bean
3. We will need a class “UserManager” with the following methods
   List<User> getAllUsers()
   void addUser(User u)
   void updateUser(User u)
   void deleteUser(Integer id)
4. We need the following servlets:
   UserListServlet, AddEditUserServlet, DeleteUserServlet
5. We will need two JSP pages
   UserList.jsp and UserForm.jsp

Expression Language

The EL allows page authors to use simple expressions to dynamically access data from JavaBeans components.

<%= request.getAttribute(“name”) %> will be replaced with ${name}

<%= ((User)request.getAttribute(“user”)).getName() %> will be replaced with ${user.name}

Notice that the type casting, method names etc are not needed. As long as the class uses java bean convention naming patterns for methods, it works fine.

Types of EL Expressions

Immediate evaluation expressions or deferred evaluation expressions. An immediate evaluation expression is evaluated at once by the underlying technology. A deferred evaluation expression can be evaluated later by the underlying technology using the EL.

Value expression or method expression. A value expression references data, whereas a method expression invokes a method.

Rvalue expression or lvalue expression. An rvalue expression can only read a value, whereas an lvalue expression can both read and write that value to an external object.

Expression Examples

Expressions can be used to access objects in any scope. Default is to search for the objects in the following scope order
page
request
session
application

Objects being accessed can be collections, maps or java beans. Accessing objects that are not in java bean format are not allowed in the immediate evaluation expressions of type ${..}

Deferred evaluation expressions are of the syntax #{..}

Some examples are as below:

${customer} - immediate evaluation, #{customer} - deferred evaluation
${user.name} is same as ${user[“name”]} is same as pageContext.findAttribute(“user”).getName()
${user.address[“city”]} - user.getAddress().getCity()
user.phones[1] - Accesses the first item in collection
user.phones[“home”] - Access the “home” value if phones is a map
${user.age + 15} - Does the arithmetic during rendering time

EL Implicit Objects

When we say something like ${user} EL looks for “user” attribute in page, request, session and application scopes in that order. However if we need it to look in a specific scope we can also write it like ${requestScope.user}. In this case it will only look for user attribute in the request object and not any other scope. Returns null if its not found. The word “requestScope” is a EL implicit object representing the current httpservletrequest object. There are other implicit objects as outlined below:

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

JSP Standard Tag Library (JSTL)

JSTL is a set of additional tags that can be used in asp. These tags are processed on the server side and the client browser will not see them at all. An if condition using the JSTL will look like this:

```
<c:if test="${user.age > 200}">
    <p>You are really really old!<p>
```

```
</c:if>
```

This is in line with the html syntax and just looks like some new set of tags. The definition and implementation of these tags is done in a tag library. We will get to define our own tags later on but for now lets learn to use tags that someone else has defined.
The letter “c” in the c:if is called the tag prefix. A prefix is defined by a tag lib directive like below:

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

Every tag has a unique uri (not url) in its tag library definition file.
The simplest of tags is the if tag from the JSTL library. C is the most common prefix used on these tags to indicate core jstl tags

Attribute	Description	Required	Default
test	Condition to evaluate	Yes	None
var	Name of the variable to store the condition's result	No	None
scope	Scope of the variable to store the condition's result	No	page

c:foreach

Provide a drop-down for age input. Options for that should come from the server and should be between 18 and 70

```
<c:forEach var="i" begin="1" end="5">
   Item ${i}
</c:forEach>
```

Attribute	Description	Required	Default
items	Information to loop over	No	None
begin	Element to start with (0 = first item, 1 = second item, ...)	No	0
end	Element to end with (0 = first item, 1 = second item, ...)	No	Last element
step	Process every step items	No	1
var	Name of the variable to expose the current item	No	None
varStatus	Name of the variable to expose the loop status	No	None

Choose, When... Otherwise...

There is no else clause in c:if tag. These set of 3 tags act like an if-else condition

```
<c:choose>
    <c:when test="${user.age <= 18}">
       You are not even legal here! get out.
    </c:when>
    <c:when test="${user.age>18 && user.age<70}">
        Welcome.. lets have fun.
    </c:when>
    <c:otherwise>
        Time to stay home and relax
    </c:otherwise>
</c:choose>
```

Format Tags

There are a number of tags meant for formatting data for rendering under the format tags. Here are some examples

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

<c:set var="now" value="<%=new java.util.Date()%>" />
<fmt:formatDate pattern="yyyy-MM-dd" value="${now}" />
```

Other Tags

There are many other format tags for numbers, timezones, strings etc
There are a set of tags called SQL Tags - It is not recommended that we use these tags as these violate the basic design premise of MVC model. Accessing db directly from the JSP should be avoided
There are many more core tags and
XML tags - for dealing with XML data

JSTL Functions

JSTL includes a number of standard functions, most of which are common string manipulation functions.

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

<c:if test="${fn:contains(user.name, 'admin')}">
   <p>Admin User!<p>
</c:if>

<c:set var="string1" value="This is first String."/>
<c:set var="string2" value="${fn:split(string1, ' ')}" />
<c:set var="string3" value="${fn:join(string2, '-')}" />
```

Tag Libraries

Beyond the tags that are provided by JSTL, there are many other third party companies and communities that provide tags for other uses. Here are some examples.
DisplayTag - a tag library for displaying a list of elements in a table structure
Struts - a set of tags that comes along with the struts libraries meant for easy use of the framework backend
Tiles - a struts based tag library that allows us to organize a page into many component tiles
Spring MVC tags - spring also comes with its own set of tags for convenient use of the framework

Now lets learn to use the DisplayTag library.

Figure out how to use this tag library, and adapt it into the code: http://www.displaytag.org/1.2/

Error Handling

Error handling has to be done in a user friendly way. Its never a good idea to let users see the crude error pages from the application server that look like this:

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error that prevented it from fulfilling this request.

exception

```
java.lang.NullPointerException
        com.mydomain.servlets.ErrorServlet.doGet(ErrorServlet.java:15)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:620)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:727)
        org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
```

note The full stack trace of the root cause is available in the Apache Tomcat/7.0.52 logs.

Apache Tomcat/7.0.52

Error pages are going to be more friendly. To setup such an error page we have to tell the container to use a specific page as an error page by putting in settings like this in web.xml

```
<error-page>
  <error-code>500</error-code>
  <location>/error.jsp</location>
</error-page>
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/error.jsp</location>
</error-page>
```

Then in the error page we have access to a bunch of variables that help us say something about the error to the end user

```
<h1>Opps...</h1>
<table width="100%" border="1">
<tr valign="top">
<td width="40%"><b>Error:</b></td>
<td>${pageContext.exception}</td>
</tr>
<tr valign="top">
<td><b>URI:</b></td>
<td>${pageContext.errorData.requestURI}</td>
</tr>
<tr valign="top">
<td><b>Status code:</b></td>
<td>${pageContext.errorData.statusCode}</td>
</tr>
<tr valign="top">
<td><b>Stack trace:</b></td>
<td>
<c:forEach var="trace"
         items="${pageContext.exception.stackTrace}">
<p>${trace}</p>
</c:forEach>
```

Creating Our Own Tags

Just like the standard and library tags, we could build our own tag library. Some of the main steps at creating our own tags are:

1. First need to describe the tag in a manner that the asp validator can understand. What tag name, what attributes it has, what are the types of attributes, what kind of body does the tag have, etc.. This is done in the TLD file
2. Second we have to define the function of the tag in a tag handler class. All tag handlers have a lifecycle just like servlets. They are created when a tag is used and certain methods on the tag handler is called. We implement these methods to provide the functionality of the tag
3. Define a uri for the tags in the TLD and use them in a JSP page.

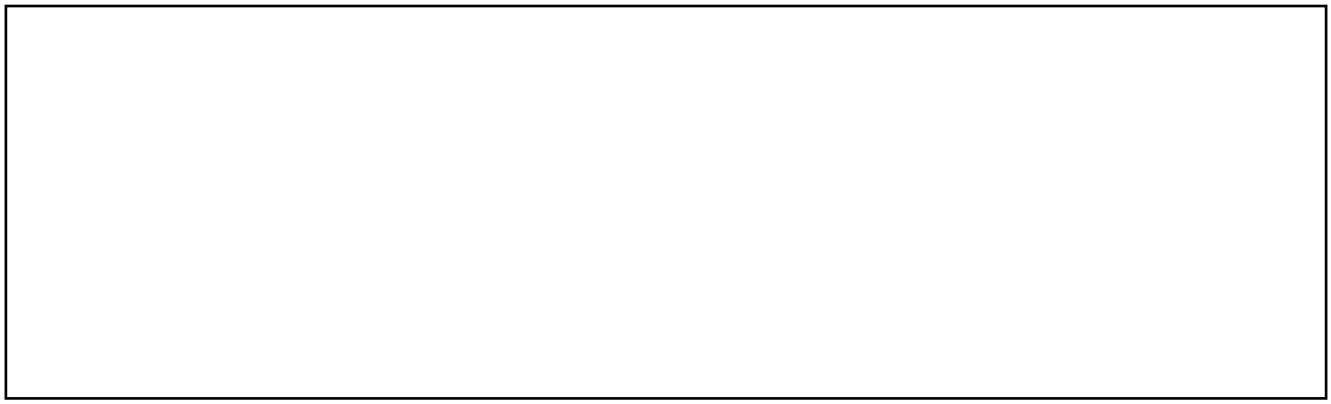
Shown below are fragments of code to create a simple tag that shows the US states

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>US States</short-name>
  <uri>http://com.mydomain/mytags</uri>
  <tag>
    <name>UsStates</name>
    <tag-class>com.mytag.UsStatesTag</tag-class>
    <body-content>empty</body-content>

  </tag>
</taglib>
```

```
public class UsStatesTag extends SimpleTagSupport {

      public void doTag() throws JspException, IOException {

            String selectTag = "<select name='states'><option
value='GA'>Georgia</option><option value='CA'>California</option></
select>";
            getJspContext().getOut().println(selectTag);
      }
}
```

```
<%@ taglib prefix="mt" uri="http://com.mydomain/mytags" %>

<mt:UsStates name="user.state"/>
```

This was a simple tag with no body or attributes. We could have tags that have regular html in body, another sub tag (like choose..when), have attributes etc



Tag Lifecycle

Tag life cycle is depicted on the image shown next.

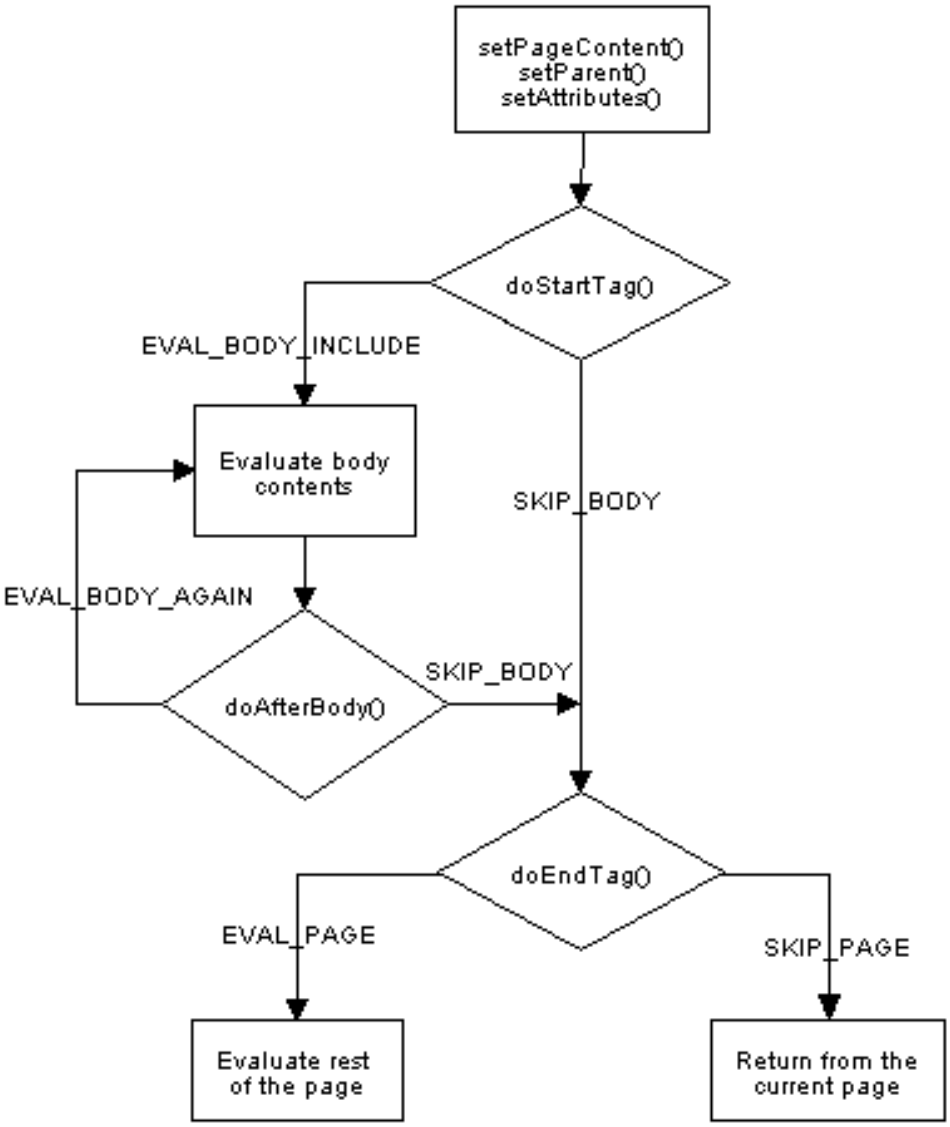
Tag is executed in conduction with the JSP engine. Execution starts by passing the pageContext, parentTag and attributes of the tag to the tag handler. Once the data is setup, doStartTag is called. The doStartTag method has access to the JSP page context and can write anything it wants to the page being rendered. After which it makes a decision weather to proceed with execution of the body or not (like in the c:if tag)

doStartTag returns one of the few constants such as EVAL_BODY, SKIP_BODY. It is back to the JSP engine to evaluate the body as a regular JSP or skip it.

If the body was executed, doAfterBody() of this tag is called. This can be used to check any state changes made by body and render additional content for this tag.

Finally irrespective of weather the body is executed or not, the doEndTag is called. This method gets to make a deception wether to proceed with the execution of the remaining JSP page or just skip and get out.

Life Cycle of a Tag extending TagSupport



Redirect vs Forward

After a servlet finishes processing a request, it has these options:
1. Create a response and send back to the client using the output stream or print writer in the response object
2. Forward the request to another servlet or JSP to do further processing
3. Send a redirect to the client browser

In an MVC pattern we do not typically use the servlet to create the response for us. So the options are really 2 and 3.

Forward is something that happens on the server and is transparent to the client browser. The url on the browser does not change. One servlet's request and response objects are used to call the doGet/doPost method of another servlet or a JSP. So the processing is done in a single thread. The request attributes that are set by the first servlet is available to the second one.

Where as a redirect involves a client browser. The first servlet finishes up and sends back a 302 response to the client asking it to send another request to another url. The new request to the next url is processed in a new thread and a new request object is created. None of the request attributes of the previous servlet are available. Infact the redirect can be done to any other web site in the world… not necessarily another servlet on the same app.

Typically its a bad practice to chain a post call with a get call on the server. Remember that the GET call is supposed to be idempotent and the POST call is not? So if the user does a refresh, it does a post again. User might be under the impression that he is just refreshing the idempotent data on screen.

Filters

Servlet Filters are Java classes that can be used in Servlet Programming for the following purposes:

To intercept requests from a client before they access a resource at back end.

To manipulate responses from server before they are sent back to the client.

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.

A filter is simply a Java class that implements the javax.servlet.Filter interface. The javax.servlet.Filter interface defines three methods:

1 public void doFilter (ServletRequest, ServletResponse, FilterChain)
This method is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain.

2 public void init(FilterConfig filterConfig)
This method is called by the web container to indicate to a filter that it is being placed into service.

3 public void destroy()
This method is called by the web container to indicate to a filter that it is being taken out of service.

```
<filter>
  <filter-name>usageStats</filter-name>
  <filter-class>com.lo.app.servlets.UsageStatsFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>usageStats</filter-name>
  <url-pattern>*.action</url-pattern>
</filter-mapping>
```

Filter Impl

To implement user authentication using filters, here are the steps:

1. Create a JSP with a form that has user/password fields for use as a login page.
2. Create a servlet that processes this form and if the user/password matches whats in database, places the user object in session and does a redirect to user listing screen
3. Now create a filter that checks if the user object exists in session or not. If it exists, it continues and calls the chain.doFilter method to forward the request further. If it does not exist, it wont call the chain.doFilter, instead sends a redirect to the login page JSP

Async Servlets

Traditional servlet implementations have the following problems that is solved by the Async servlet

First, was the issue of thread starvation. Each thread consumes server side resources – both memory and CPU resources. As a result, the number of threads within the pool must be constrained to some finite number. Furthermore, if the processing that the thread must perform for a client requires some blocking operation (say waiting on some slow external resource such as a database or web service), that thread is effectively unavailable to the server. In other words, it is

possible that all threads in the pool are soon blocked and incoming requests can no longer be effectively handled. This is not just a theoretical problem. With the prevalence of fine grained Ajax-requests, and the adoption of SOA, the load on web servers is on an upward trend.

Second was the issue of staleness. At any given point, a client gets a view of the web application's state at the instant that the request was processed. Any changes to state between two consecutive requests from a client, are effectively invisible to that client. In other words, each client only gets a static snapshot view into what is effectively a dynamic system. And the currency of that view decays the longer the time period between requests. Ajax's fine grained interactions, do mitigate this problem, but it does not completely solve the currency issue.

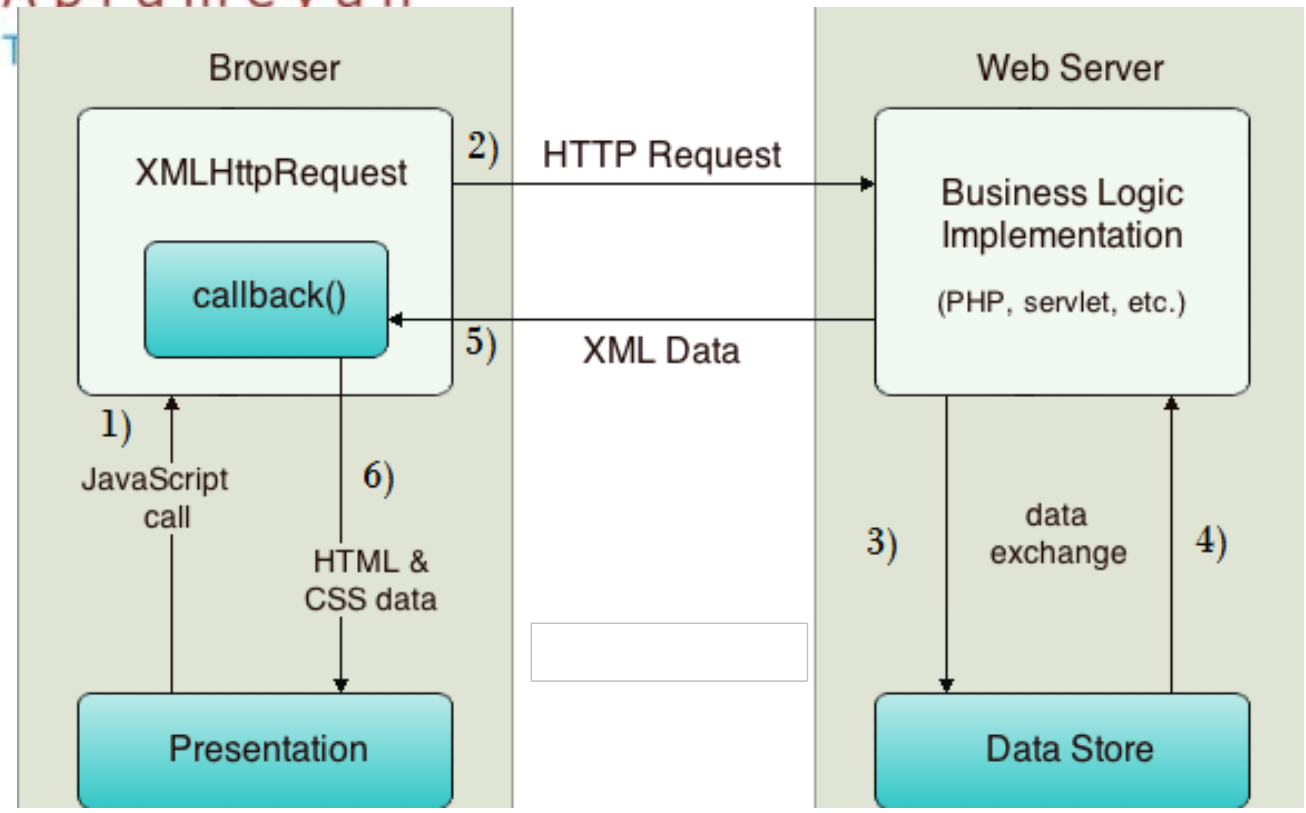
Third was the issue of non standard implementations. The developers of every servlet container decided to solve this problem of the synchronous web by implementing their own custom solutions. Tomcat's Comet, WebLogic's FutureResponseServlet, and WebSphere's Asynchronous Request Dispatcher are examples of developers not wanting to wait for the standards body to catch up.

Async servlets allow us to unblock the request processing thread pool while the servlet performs its long running task in another thread
Its like putting the request in cold storage and returning the thread till the request processing is done
AsyncContext class maintains the state of the request thats dormant now
Client browser sees it as just another long running request

Example AsyncServlet

We start async session by calling the request.startAsync method. There after we hand off the long processing over to another thread. That thread should call complete() or dispatch() on the async context before timeout for the user to get a response

```
ExecutorService serv = Executors.newFixedThreadPool(5);

	protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
		request.setAttribute("org.apache.catalina.ASYNC_SUPPORTED", true);
		final AsyncContext context = request.startAsync();
		System.out.println("Async started");
		Callable<String> c = new Callable<String>() {
			@Override
			public String call() throws Exception {
				context.getRequest().setAttribute("asyncdata","Hello World");
				context.dispatch("/asyncdone.jsp");
				return "Hello World";
			}
```




```
            };
            serv.submit(c);
            System.out.println("Servlet done");
    }
```

AJAX

Asynchronous JavaScript and XML is a group of interrelated Web development techniques used on the client-side to create asynchronous Web applications. With Ajax, Web applications can send data to and retrieve from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. Data can be retrieved using the XMLHttpRequest object. Despite the name, the use of XML is not required (JSON is often used in the AJAJ variant), and the requests do not need to be asynchronous.

Ajax is not a single technology, but a group of technologies. HTML and CSS can be used in combination to mark up and style information. The DOM is accessed with JavaScript to dynamically display – and allow the user to interact with – the information presented. JavaScript and the XMLHttpRequest object provide a method for exchanging data asynchronously between browser and server to avoid full page reloads.

XMLHttpRequest

An object of XMLHttpRequest is used for asynchronous communication between client and server.

It performs following operations:

Sends data from the client in the background
Receives the data from the server
Updates the webpage without reloading it.

The common properties of XMLHttpRequest object are as follows:

Property	Description
onReadyStateChange	It is called whenever readystate attribute changes. It must not be used with synchronous requests.
readyState	represents the state of the request. It ranges from 0 to 4 **0** UNOPENED open() is not called. **1** OPENED open is called but send() is not called. **2** HEADERS_RECEIVED send() is called, and headers and status are available. **3** LOADING Downloading data; responseText holds the data. **4** DONE The operation is completed fully.
reponseText	returns response as text.
responseXML	returns response as XML

The important methods of XMLHttpRequest object are as follows:

Method	Description
void open(method, URL)	opens the request specifying get or post method and url.

Method	Description
void open(method, URL, async)	same as above but specifies asynchronous or not.
void open(method, URL, async, username, password)	same as above but specifies username and password.
void send()	sends get request.
void send(string)	send post request.
setRequestHeader(header,value)	it adds request headers.

XMLHttpRequest

As you can see in the above example, XMLHttpRequest object plays a important role.

1. User sends a request from the UI and a javascript call goes to XMLHttpRequest object.
2. HTTP Request is sent to the server by XMLHttpRequest object.
3. Server interacts with the database using JSP, PHP, Servlet, ASP.net etc.
4. Data is retrieved.
5. Server sends XML data or JSON data to the XMLHttpRequest callback function.
6. HTML and CSS data is displayed on the browser.

Code in Conventional Javascript:

```
<script>
function loadXMLDoc()
{
	var xmlhttp;
	xmlhttp=new XMLHttpRequest();
	xmlhttp.onreadystatechange=function(){
		if (xmlhttp.readyState==4 && xmlhttp.status==200){

document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
	}
	}
	xmlhttp.open("GET","answer.txt",true);
	xmlhttp.send();
```

```
}
</script>

<div id="myDiv"><h2>How many cars does Ferrari produce in an year</h2></
div>
<button type="button" onclick="loadXMLDoc()">Find out!</button>
```

Code using JQuery library

```
<script>

function loadAjaxJquery(){
      $.get("answer.txt", function(data){
            $("#myDiv2").html(data);
      })
}

</script>

<div id="myDiv2"><h2>How many cars does Ferrari produce in an year</h2></
div>
<button type="button" onclick="loadAjaxJquery()">Find out!</button>
```

File Uploads

As we understood earlier files are sent to the server over a multipart request and parsing a multipart request is typically more involved and is never implemented from scratch by the developers. We rely on third party libraries such as Apache commons file upload.

Here is a sample code that moves uploaded files to a predefined file. Also notice the use of property files to identify the location where we store the files.

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

```
		if(ServletFileUpload.isMultipartContent(request)){
			// Create a factory for disk-based file items
			DiskFileItemFactory factory = new DiskFileItemFactory();

			// Configure a repository (to ensure a secure temp location is used)
			ServletContext servletContext = this.getServletConfig().getServletContext();
			File repository = (File)
servletContext.getAttribute("javax.servlet.context.tempdir");
			System.out.println("Temp Dir: "+repository);
			factory.setRepository(repository);

			// Create a new file upload handler
			ServletFileUpload upload = new ServletFileUpload(factory);

			// Parse the request
			try {
				List<FileItem> items = upload.parseRequest(request);
				for (FileItem fileItem : items) {
					String path = AppProperties.getString("file_save_location");
					File f = new File(path+fileItem.getName());
					byte[] data = fileItem.get();
					FileOutputStream fos = new FileOutputStream(f);
					fos.write(data);
					fos.close();
				}
			} catch (FileUploadException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
				throw new ServletException(e);
			}
		}
	}
```

There are other things that the library allows us to do. Such as set the limit of the max file size, also the approach used here reads the contents of the file into a byte array and thats usually not a good idea in terms of memory consumption. Better would be to use streams to handle the content of the file without getting the whole thing into memory. Explore the file upload library for all options: http://commons.apache.org/proper/commons-fileupload/using.html