**[Java Web application](https://www.vogella.com/tutorials/JavaWebTerminology/article.html" \l "webapp)**

A Java web application is a collection of dynamic resources (such as Servlets, JavaServer Pages, Java classes and jars) and static resources (HTML pages and pictures). A Java web application can be deployed as a WAR (Web ARchive) file.

A WAR file is a zip file which contains the complete content of the corresponding web application.

## ****Java Servlets: Introduction to Servlets****

A servlet is a [**Java Programming**](https://www.edureka.co/blog/java-tutorial/)language class that is used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. It is also a web component that is deployed on the server to create a dynamic we**Java Servlets: Servlet Architecture**

The **architecture**, here, discusses the communication interface, protocol used, requirements of client and server, the programming with the languages and software involved. Basically, it performs the below-mentioned tasks.

* First, it reads the explicit data sent by the clients (browsers).  This data can include an HTML form on a Web page, an applet or a custom HTTP client program. It also reads implicit HTTP request data sent by the clients (browsers). This can include cookies, media types and compression schemes the browser understands, and so forth.
* After that, the servlet processes the data and generate the results. This process may require communicating to a database, executing an RMI, invoking a Web service, or computing the response directly.
* After processing, it sends the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), or Excel formats.
* Finally, it also sends the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned.

Now, let’s understand the various methods in the life cycle of a servlet.

### **Servlet Life Cycle**

The Servlet life cycle mainly includes the following four stages,

* Loading a Servlet
* Initializing the Servlet
* Request handling
* Destroying the Servlet
* When the web server (e.g. Apache Tomcat) starts up, the servlet container deploy and loads all the servlets.
* The servlet is initialized by calling the init*()* method. The *Servlet.init()*method is called by the Servlet container to indicate that this Servlet instance is instantiated successfully and is about to put into service.
* The servlet then calls *service()*method to process a client’s request. This method is invoked to inform the Servlet about the client requests.
* The servlet is terminated by calling the *destroy().*
* The*destroy()* method runs only once during the lifetime of a Servlet and signals the end of the Servlet instance.

init() and destroy() methods are called only once. Finally, a servlet is garbage collected by the garbage collector of the JVM. So this concludes the life cycle of a servlet. Now, let me guide you through the steps of creating java servlets.

**Java Servlets: Steps to Create Servlet**

1. Create a directory structure
2. Create a Servlet
3. Compile the Servlet
4. Add mappings to the web.xml file
5. Start the server and deploy the project
6. Access the servlet

## ****Java Server Pages****

**JSP** or Java Server Pages is a technology that is used to create web application just like Servlet technology. It is an extension to Servlet – as it provides more functionality than a servlet such as expression language, JSTL, etc. A JSP page consists of HTML tags and JSP tags. The JSP pages are easier to maintain than Servlet because we can separate designing and development.

Now, as we know what JSP is, let’s compare JSP with Servlets and understand which is best suitable for the web.

## ****Servlet and JSP Tutorial: Advantages of JSP Over Servlets****

|  |  |
| --- | --- |
| JSP | Servlets |
| Extension to Servlet | Not an extension to servlet |
| Easy to Maintain | Bit complicated |
| No need to recompile or redeploy | The code needs to be recompiled |
| Less code than a servlet | More code compared to JSP |

**JSP Scripting Elements**

The scripting elements provide the ability to insert java code inside the JSP. There are three types of scripting elements:

* **scriptlet tag**–  A scriptlet tag is used to execute Java source code in JSP.

Syntax : <%  java source code %>

* **expression tag**– The code placed within *JSP expression tag*is *written to the output stream of the response*. So you need not write *out.print()* to write data. It is mainly used to print the values of [variable](https://www.edureka.co/blog/java-tutorial/#variables) or method.

Syntax : <%=  statement %>

* **declaration tag**– The JSP declaration tag is used *to declare fields and methods*. The code written inside the JSP declaration tag is placed outside the service() method of an auto-generated servlet. So it doesn’t get memory at each request.

Syntax: <%!  field or method declaration %>

## ****JSP Request and Response Objects****

**JSP request** is an implicit object of type HttpServletRequest that is created for each JSP request by the web container. It can be used to get request information such as a parameter, header information, remote address, server name, server port, content type, character encoding etc. It can also be used to set, get and remove attributes from the JSP request scope.

Let’s see the simple example of request implicit object where we are printing the name of the user with a welcome message. Let’s see how.

**Example of JSP request implicit object**

File: index.html

<**form**&nbsp;action="welcome.jsp">

<**input**&nbsp;type="text"&nbsp;name="uname">

<**input**&nbsp;type="submit"&nbsp;value="go">

</**form**>

File: welcome.jsp

<% String&nbsp;name=request.getParameter("uname"); print("welcome&nbsp;"+name);

### **JSP response implicit object**

In JSP, the response is an implicit object of type HttpServletResponse. The instance of HttpServletResponse is created by the web container for each JSP request. It can be used to add or manipulate responses such as redirect response to another resource, send error etc.

**Example of response implicit object**

*File: index.html*

<**form**&nbsp;action="welcome.jsp">

<**input**&nbsp;type="text"&nbsp;name="uname">

<**input** type="submit"&nbsp;value="go">

</**form**>

File: welcome.jsp

<% sendRedirect("[http://www.google.com](http://www.google.com/)"); %>

So, this is how request and response objects work. This brings to the end of Servlet and JSP tutorial article. I hope this blog was informative and added value to your knowledge.

**MVC Architecture in Java**

Model designs based on MVC architecture follow the MVC [design pattern](https://www.edureka.co/blog/java-design-patterns/) and they separate the application logic from the user interface when designing software. As the name implies MVC pattern has three layers, which are:

* ***Model*** – Represents the business layer of the application
* ***View*** – Defines the presentation of the application
* ***Controller*** – Manages the flow of the application

In Java Programming context, the Model consists of simple [Java classes](https://www.edureka.co/blog/java-objects-and-classes/#javaclass), the View displays the data and the Controller consists of [servlets](https://www.edureka.co/blog/java-servlets). This separation results in user requests being processed as follows:

1. The browser on the client sends a request for a page to the controller present on the server
2. The controller performs the action of invoking the model, thereby, retrieving the data it needs in response to the request
3. The controller then gives the retrieved data to the view
4. The view is rendered and sent back to the client for the browser to display

**Advantages of MVC Architecture in Java**

MVC architecture offers a lot of advantages for a programmer when developing applications, which include:

* Multiple developers can work with the three layers (Model, View, and Controller) simultaneously
* Offers improved *scalability*, that supplements the ability of the application to grow
* As components have a low dependency on each other, they are easy to maintain
* A model can be reused by multiple views which provides reusability of code
* Adoption of MVC makes an application more expressive and easy to understand
* Extending and testing of the application becomes easy

**Implementation of MVC using Java**

To implement a web application based on MVC design pattern, we will create

* ***Course Class***, which  acts as the *model layer*
* ***CourseView Class***, which defines the presentation layer (*view layer*)
* ***CourseContoller Class***, which acts as a *controller*

### **The Model Layer**

In the MVC design pattern, the model is the data layer which defines the business logic of the system and also represents the state of the application. The model [objects](https://www.edureka.co/blog/java-object/) retrieve and store the state of the model in a database. Through this layer, we apply rules to data, which eventually represents the concepts our application manages.

### **The View Layer**

This layer of the MVC design pattern represents the output of the application or the user interface. It displays the data fetched from the model layer by the controller and presents the data to the user whenever asked for. It receives all the information it needs from the controller and it doesn’t need to interact with the business layer directly.

### **The View Layer**

This layer of the MVC design pattern represents the output of the application or the user interface. It displays the data fetched from the model layer by the controller and presents the data to the user whenever asked for. It receives all the information it needs from the controller and it doesn’t need to interact with the business layer directly.

## Web Application Security

web application security can be configured when the application is installed, or deployed, to the web container. Annotations and/or deployment descriptors are used to relay information to the deployer about security and other aspects of the application. Specifying this information in annotations or in the deployment descriptor helps the deployer set up the appropriate security policy for the web application. Any values explicitly specified in the deployment descriptor override any values specified in annotations.

Security for Java EE web applications can be implemented in the following ways.

* **Declarative security**: Can be implemented using either metadata annotations or an application’s deployment descriptor. See [Overview of Java EE Security](https://docs.oracle.com/javaee/6/tutorial/doc/bnbwk.html) for more information.

Declarative security for web applications is described in [Securing Web Applications](https://docs.oracle.com/javaee/6/tutorial/doc/gkbaa.html).

* **Programmatic security**: Is embedded in an application and can be used to make security decisions when declarative security alone is not sufficient to express the security model of an application. Declarative security alone may not be sufficient when conditional login in a particular work flow, instead of for all cases, is required in the middle of an application. See [Overview of Java EE Security](https://docs.oracle.com/javaee/6/tutorial/doc/bnbwk.html) for more information.

Servlet 3.0 provides the authenticate, login, and logout methods of the HttpServletRequest interface. With the addition of the authenticate, login, and logout methods to the Servlet specification, an application deployment descriptor is no longer required for web applications but may still be used to further specify security requirements beyond the basic default values.

Programmatic security is discussed in [Using Programmatic Security with Web Applications](https://docs.oracle.com/javaee/6/tutorial/doc/gjiie.html)

* **Message Security**: Works with web services and incorporates security features, such as digital signatures and encryption, into the header of a SOAP message, working in the application layer, ensuring end-to-end security. Message security is not a component of Java EE 6 and is mentioned here for informational purposes only.

### **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.

1. 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

**Session in Java Servlet – HttpSession**

### Servlet API provides Session management through HttpSess**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.

ion 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.

**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.