Java Servlet

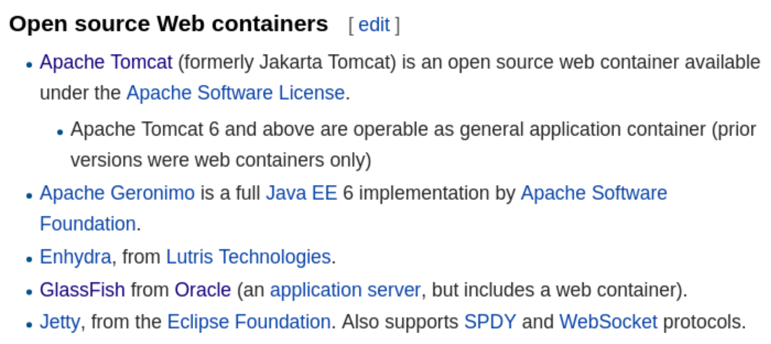
Web container vs Java servlet container (same thing)

A **web container** (also known as a **servlet container**) is the component of a web server that interacts with Java servlets. 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.

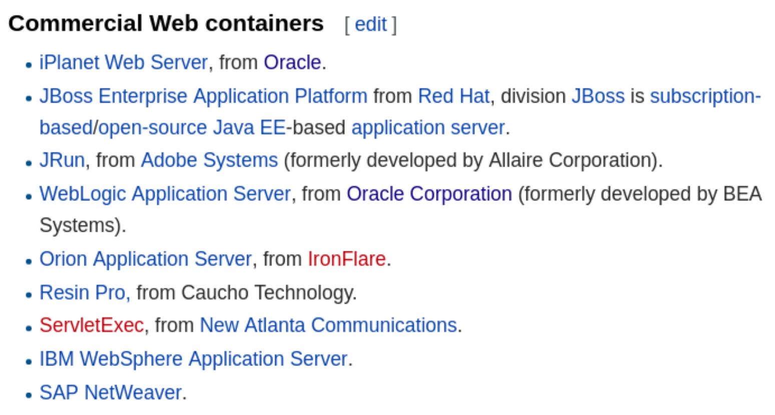
A web container implements the web component contract of the Java EE architecture, specifying a runtime environment for web components that includes security, concurrency, lifecycle management, transaction, deployment, and other services.

The following is a list of applications which implement the Java Servlet specification from Sun Microsystems, divided depending on whether they are directly sold or not.



Open source Web containers [edit]

- Apache Tomcat (formerly Jakarta Tomcat) is an open source web container available under the Apache Software License.
  - Apache Tomcat 6 and above are operable as general application container (prior versions were web containers only)
- Apache Geronimo is a full Java EE 6 implementation by Apache Software Foundation.
- Enhydra, from Lutris Technologies.
- GlassFish from Oracle (an application server, but includes a web container).
- Jetty, from the Eclipse Foundation. Also supports SPDY and WebSocket protocols.



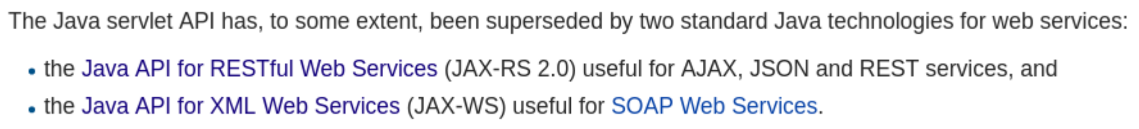
Commercial Web containers [edit]

- iPlanet Web Server, from Oracle.
- JBoss Enterprise Application Platform from Red Hat, division JBoss is subscription-based/open-source Java EE-based application server.
- JRun, from Adobe Systems (formerly developed by Allaire Corporation).
- WebLogic Application Server, from Oracle Corporation (formerly developed by BEA Systems).
- Orion Application Server, from IronFlare.
- Resin Pro, from Caucho Technology.
- ServletExec, from New Atlanta Communications.
- IBM WebSphere Application Server.
- SAP NetWeaver.

Java Servlet

A Java servlet is a Java program that extends the capabilities of a server. Although servlets can respond to any types of requests, they most commonly implement applications hosted on Web servers. Such Web servlets are the Java counterpart to other dynamic Web content technologies such as PHP and ASP.NET.

A Java servlet processes or stores a Java class in Java EE that conforms to the Java Servlet API,[2] a standard for implementing Java classes that respond to requests. Servlets could in principle communicate over any client–server protocol, but they are most often used with the HTTP. Thus "servlet" is often used as shorthand for "HTTP servlet".[3] Thus, a software developer may use a servlet to add dynamic content to a web server using the Java platform. The generated content is commonly HTML, but may be other data such as XML. Servlets can maintain state in session variables across many server transactions by using HTTP cookies, or URL rewriting.



The Java servlet API has, to some extent, been superseded by two standard Java technologies for web services:

- the Java API for RESTful Web Services (JAX-RS 2.0) useful for AJAX, JSON and REST services, and
- the Java API for XML Web Services (JAX-WS) useful for SOAP Web Services.

To deploy and run a **servlet**, a **web container** must be used. A web container (also known as a servlet container) is essentially the component of a web server that interacts with the servlets. The 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.

The Servlet API, contained in the Java package hierarchy `javax.servlet`, defines the expected interactions of the web container and a servlet.

A Servlet is an object that receives a request and generates a response based on that request. The basic Servlet package defines Java objects to represent servlet requests and responses, as well as objects to reflect the servlet's configuration parameters and execution environment. The package javax.servlet.http defines HTTP-specific subclasses of the generic servlet elements, including session management objects that track multiple requests and responses between the web server and a client. Servlets may be packaged in a WAR file as a web application.

Servlets can be generated automatically from JavaServer Pages (JSP) by the JavaServer Pages compiler. The difference between servlets and JSP is that servlets typically embed HTML inside Java code, while JSPs embed Java code in HTML. While the direct usage of servlets to generate HTML (as shown in the example below) has become rare, the higher level MVC web framework in Java EE (JSF) still explicitly uses the servlet technology for the low level request/response handling via the FacesServlet. A somewhat older usage is to use servlets in conjunction with JSPs in a pattern called "Model 2", which is a flavor of the model–view–controller.

The current version of Servlet is 4.0.

Are servlets / jsp still used?

Servlets and Jsps are considered as an outdated technologies and no longer choosen for the new projects. These were found in use significantly for the legacy projects. Servlets jsps were used enormously in around 2000. With the popularity of emerging MVC frameworks like Struts, Webwork, Sprint etc industry realized the potential which comes out of the box from these frameworks and hence people started using them over core Servlets jsp implementation. So vanila implementation of Servlets jsp is something which industry is trying to get rid of. However one must understand the core concepts of these two technology to build a strong foundation of any web application.

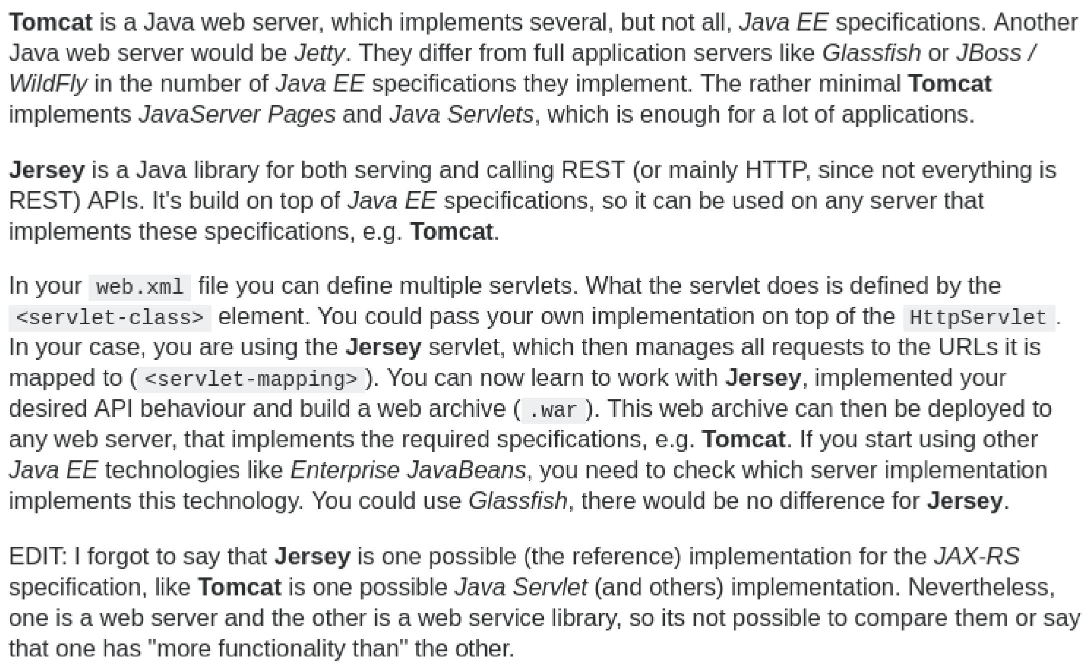
Nowadays use of framework is increasing. But without having understanding of servlet you can not understand the beauty of framework specially Spring.

Web service vs Servlet vs Web Application

What you're describing is a web application, where a human uses a browser to interact with a software system. Application is the software that is using this API provided by the web service.

A web service is a way for software systems to communicate with each other using HTTP and XML or JSON, without any humans involved.

A servlet is a Java-specific way of writing software that responds to HTTP requests. Spring MVC abstracts away a lot of the implementation detail to make writing web applications easier, but uses servlets under the covers.



Tomcat is a Java web server, which implements several, but not all, *Java EE* specifications. Another Java web server would be *Jetty*. They differ from full application servers like *Glassfish* or *JBoss / WildFly* in the number of *Java EE* specifications they implement. The rather minimal **Tomcat** implements *JavaServer Pages* and *Java Servlets*, which is enough for a lot of applications.

Jersey is a Java library for both serving and calling REST (or mainly HTTP, since not everything is REST) APIs. It's build on top of *Java EE* specifications, so it can be used on any server that implements these specifications, e.g. **Tomcat**.

In your `web.xml` file you can define multiple servlets. What the servlet does is defined by the `<servlet-class>` element. You could pass your own implementation on top of the `HttpServlet`. In your case, you are using the **Jersey** servlet, which then manages all requests to the URLs it is mapped to (`<servlet-mapping>`). You can now learn to work with **Jersey**, implemented your desired API behaviour and build a web archive (`.war`). This web archive can then be deployed to any web server, that implements the required specifications, e.g. **Tomcat**. If you start using other *Java EE* technologies like *Enterprise JavaBeans*, you need to check which server implementation implements this technology. You could use *Glassfish*, there would be no difference for **Jersey**.

EDIT: I forgot to say that **Jersey** is one possible (the reference) implementation for the *JAX-RS* specification, like **Tomcat** is one possible *Java Servlet* (and others) implementation. Nevertheless, one is a web server and the other is a web service library, so its not possible to compare them or say that one has "more functionality than" the other.

There are many (competing) server-side technologies available: Java-based (servlet, JSP, JSF, Struts, Spring, Hibernate), ASP, PHP, CGI Script, 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. You need to understand Servlet thoroughly before you could proceed to other Java server-side technologies such as JavaServer Pages (JSP) and JavaServer Faces (JSF).

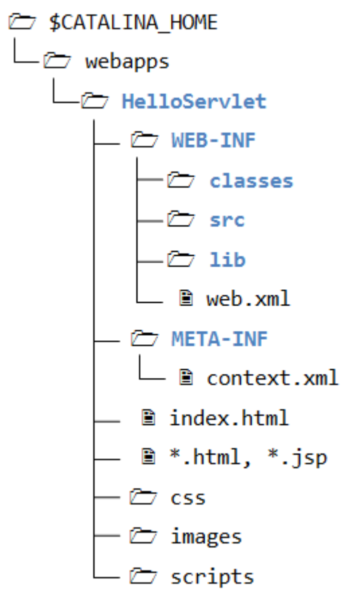
Apache Tomcat, often referred to as Tomcat Server, is an open-source Java Servlet Container developed by the Apache Software Foundation (ASF). Tomcat implements several Java EE specifications including Java Servlet, JavaServer Pages (JSP), Java EL, and WebSocket, and provides a "pure Java" HTTP web server environment in which Java code can run.

Tomcat Server is the official Reference Implementation for Java servlet and JSP.

For compiling the servlet, different jar files are required for different servers:

1.	servlet-api.jar	Apache Tomcat
2.	weblogic.jar	Weblogic
3.	javaee.jar	Glassfish
4.	javaee.jar	Jboss

A Java webapp has a standardized directory structure for storing various types of resources.



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

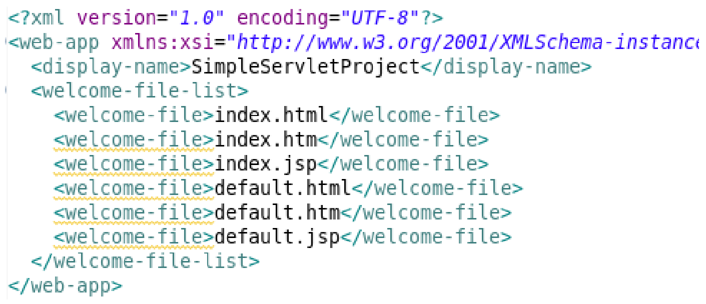
Creating a simple servlet project

Created a new Dynamic Web Project in eclipse. Using Tomcat 8.0. Eclipse configured all the necessary jars already In our Project (see Apache Tomcat library and JRE System Library). **servlet-api.jar** is under the Tomcat library.

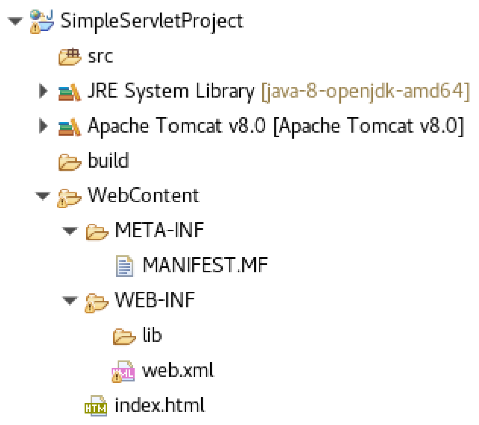
We need the Servlet API library to compile this program. **Servlet API** is not part of JDK or Java SE (but belongs to Java EE). Tomcat provides a copy of servlet API called "servlet-api.jar" in "<root>\lib". You could copy "servlet-api.jar" from "<root>\lib" to "<JAVA_HOME>\jre\lib\ext" (the JDK Extension Directory), or include the Servlet JAR file in your CLASSPATH.

WebContent: Contains all the web content (html,jsp…). Everything will be inside this folder.

web.xml:



```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
  <display-name>SimpleServletProject</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```



```
SimpleServletProject
    src
  JRE System Library [java-8-openjdk-amd64]
  Apache Tomcat v8.0 [Apache Tomcat v8.0]
    build
  WebContent
    META-INF
        MANIFEST.MF
    WEB-INF
        lib
        web.xml
    index.html
```

web.xml maps URL's to pages/servlets?

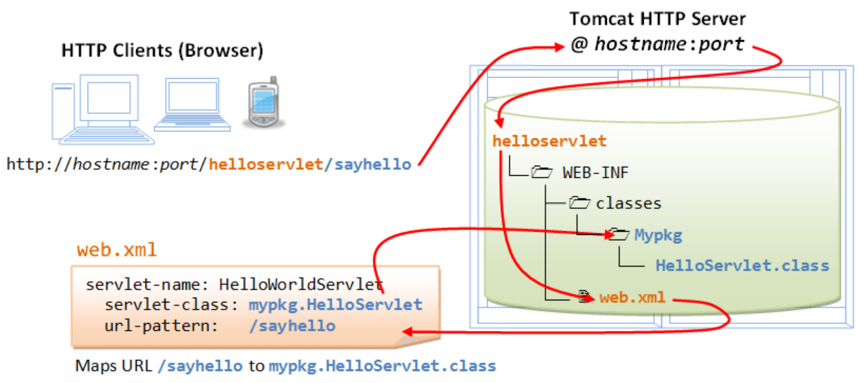
Welcome file is a file that will be called when no URL extension is entered, just localhost/OurAppName. We did not have any html files in our WebContent folder, so we created a simple index.html file.

Now going to `http://localhost:8080/SimpleServletProject/` gives us the index.html file we created.

- Our tomcat instance could have many applications running in it. We have to enter the name of the application after the root URL to be able to specify which application (also which servlet path) we want to call.

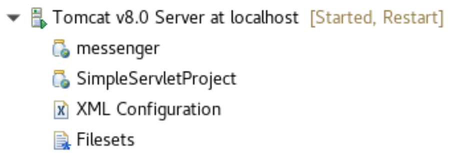
When an HTTP request comes to Tomcat, it creates 2 objects: `Request` and `Response`. It writes the data from the HTTP request to the Request object. It also looks to the URL to see which servlet should be called (there can be many servlets in an application).

It can be configured via web.xml file or different annotations. Then it passes the 2 objects to the corresponding servlet. After the servlet is done, the Response object is sent back as HTML by tomcat.

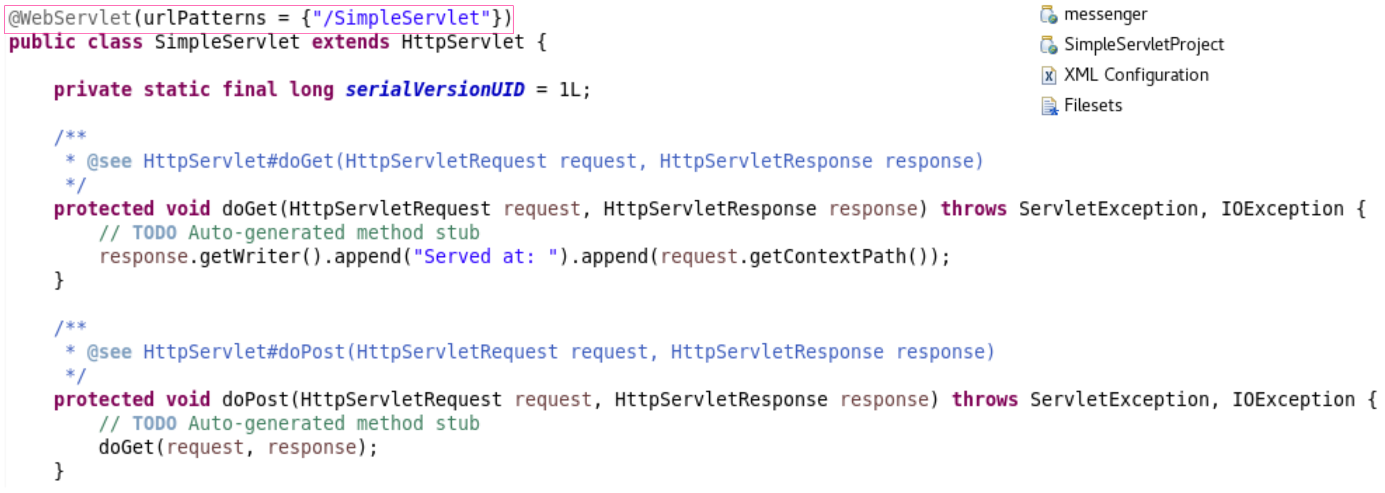

Tomcat HTTP Server
@ hostname:port
HTTP Clients (Browser)
http://hostname:port/helloservlet/sayhello
helloservlet
WEB-INF
classes
Mypkg
HelloServlet.class
web.xml
web.xml
servlet-name: HelloWorldServlet
servlet-class: mypkg.HelloServlet
url-pattern: /sayhello
Maps URL /sayhello to mypkg.HelloServlet.class

Creating a Servlet: A servlet is a java class that is saved on the tomcat instance. Tomcat acts like a container, the tomcat instance is going to run the java class that we've created(?). This class is going to run on the JMV which is on the server(?).

The servlet will not have a main method. There are some default methods executed when we access a servlet.



```
Tomcat v8.0 Server at localhost  [Started, Restart]
    messenger
    SimpleServletProject
    XML Configuration
    Filesets
```

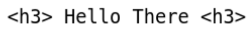


```java
@WebServlet(urlPatterns = {"/SimpleServlet"})
public class SimpleServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
        response.getWriter().append("Served at: ").append(request.getContextPath());
    }

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

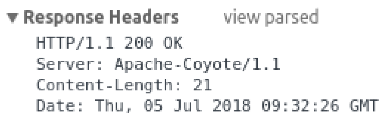
- "`UrlPatterns`" is important. Whenever a request comes to our application/project, it will look up these annotations and when it finds the matching url extension on a class, it will execute that class/servlet

When we send a **GET** request to `localhost:8080/SimpleServletProject/SimpleServlet` it will execute the `doGet()` method. If we send a **POST** request, it will execute the `doPost()` method. Our request will be transformed into a `HttpServletRequest` object and I guess in our methods we can modify the `HttpServletResponse` object "`response`" and that object will be sent as a response (since these methods return nothing)

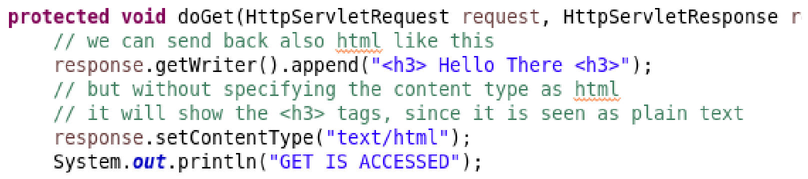
- Since our project is already deployed, when we change something, tomcat automatically restarts the server. So we don't have to deploy it again.



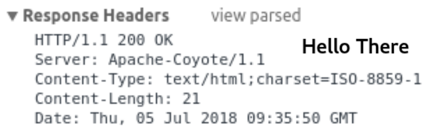
```
<h3> Hello There <h3>
```



```
▼ Response Headers      view parsed
   HTTP/1.1 200 OK
   Server: Apache-Coyote/1.1
   Content-Length: 21
   Date: Thu, 05 Jul 2018 09:32:26 GMT
```



```java
protected void doGet(HttpServletRequest request, HttpServletResponse r
    // we can send back also html like this
    response.getWriter().append("<h3> Hello There <h3>");
    // but without specifying the content type as html
    // it will show the <h3> tags, since it is seen as plain text
    response.setContentType("text/html");
    System.out.println("GET IS ACCESSED");
```



```
▼ Response Headers      view parsed
   HTTP/1.1 200 OK
   Server: Apache-Coyote/1.1
   Content-Type: text/html;charset=ISO-8859-1
   Content-Length: 21
   Date: Thu, 05 Jul 2018 09:35:50 GMT
```

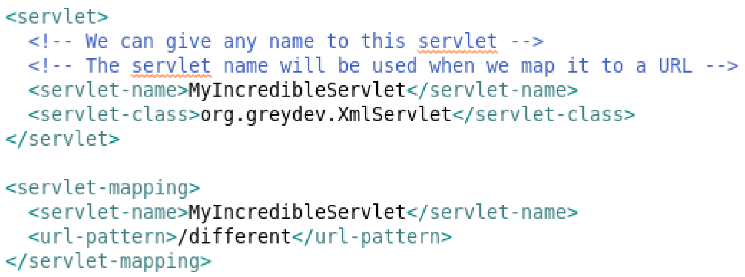
Hello There

We saw in this example how to configure servlets with annotations. Annotation support came with java 5.

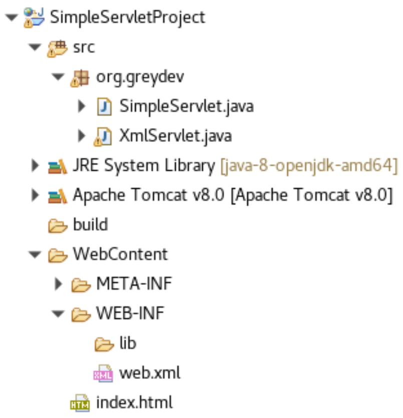
The second way is to configure our servlets via XML

The XML file called "**web.xml**" under the WEB-INF folder. It is also called deployment descriptor. Notice that we didn't use any annotations. Just added these tags to the web.xml file.

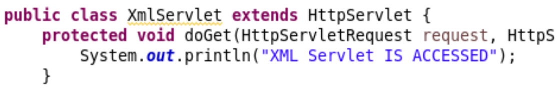
```
http://localhost:8080/SimpleServletProject/different
```



```xml
<servlet>
  <!-- We can give any name to this servlet -->
  <!-- The servlet name will be used when we map it to a URL -->
  <servlet-name>MyIncredibleServlet</servlet-name>
  <servlet-class>org.greydev.XmlServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>MyIncredibleServlet</servlet-name>
  <url-pattern>/different</url-pattern>
</servlet-mapping>
```



```
▼ SimpleServletProject
    ▼ src
        ▼ org.greydev
            ▶ SimpleServlet.java
            ▶ XmlServlet.java
    ▶ JRE System Library [java-8-openjdk-amd64]
    ▶ Apache Tomcat v8.0 [Apache Tomcat v8.0]
      build
    ▼ WebContent
        ▶ META-INF
        ▼ WEB-INF
              lib
              web.xml
          index.html
```

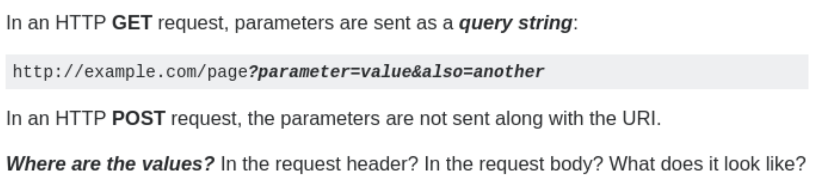


```java
public class XmlServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpS
        System.out.println("XML Servlet IS ACCESSED");
    }
```

2 ways to pass parameters:

- When we send a GET requets → with URL parameters **?name=can&city=potsdam…**
- Also POST parameters

We can access those parameters with different methods of request object. `Request.getParameter("name")`


In an HTTP **GET** request, parameters are sent as a ***query string***:

```
http://example.com/page?parameter=value&also=another
```

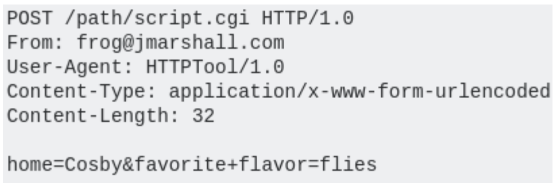
In an HTTP **POST** request, the parameters are not sent along with the URI.

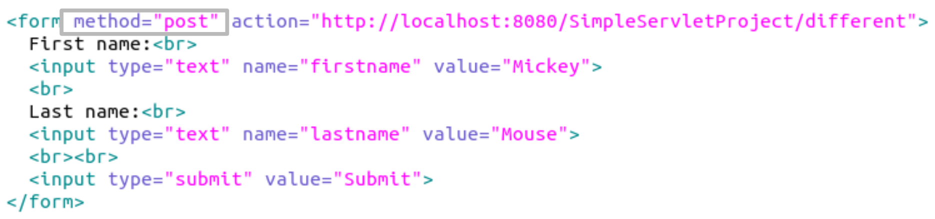
Where are the values? In the request header? In the request body? What does it look like?

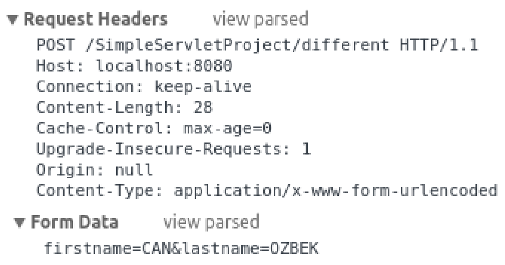
The values are sent in the request body, in the format that the content type specifies.

Usually the content type is application/x-www-form-urlencoded, so the request body uses the same format as the query string: parameter=value&also=another

The content is put after the HTTP headers. The format of an HTTP POST is to have the HTTP headers, followed by a blank line, followed by the request body. The POST variables are stored as key-value pairs in the body. You can see this in the raw content of an HTTP Post, shown here:


```
POST /path/script.cgi HTTP/1.0
From: frog@jmarshall.com
User-Agent: HTTPTool/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

home=Cosby&favorite+flavor=flies
```


```
<form method="post" action="http://localhost:8080/SimpleServletProject/different">
  First name:<br>
  <input type="text" name="firstname" value="Mickey">
  <br>
  Last name:<br>
  <input type="text" name="lastname" value="Mouse">
  <br><br>
  <input type="submit" value="Submit">
</form>
```



```
POST /SimpleServletProject/different HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Content-Length: 28
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: null
Content-Type: application/x-www-form-urlencoded
```

```
firstname=CAN&lastname=OZBEK
```

➢ a **html form,** if the method="get" it will send the parameters as query sting, appending them to the url. But if the method="post" then it will send it in the post body, not in the url

Object creation

- **Request** and **response** objects – Per access, created with every user per request.
- Servlet object –Same object is Reused, with different threads.
- Different requests have different servlet threads, not instances

Remembering User data, since HTTP protocol is stateless: (Login screen, Shopping carts …)

We can use a **session object**. Tomcat provides it us and we can use that to save values and retrieve the same data later. So if the same user send a different request, the data from the previous request will already be stored for that user. (but if he sends a new request with a different browser, data wont be visible)

In order to access/retrieve the session object: Lets save user name in a session so he/she doesnt have to enter it every time. Session objects are visible within different servlets(?) for the same user.

```
HTTPSession session = request.getSession();
session.setAttribute("key", request.getParameter("name"));
```

Now we can put any values into this session and it will remain there during the execution of different methods (for the same client I guess). Session objects: One per user/browser

```
Session.getAttribute("key");
```

To retrieve the data.

What if we want an object that will be persisted so different users will see or different browsers.

Across the entire application-specific. Shared across servlets and users

An example: One database connection will be opened for all applications, Not one for each.

- Use the **context object**. Tomcat implements it.
- Context objects are visible within different servlets(?) for the whole app and for all users.

Different ways to work with it, this is the second way:

```
ServletContext context = request.getServletContext();
context.setAttribute("key", value);
```

We know that with every request a new thread is created and that thread will access the one servlet objects doGet or doPost method.

There is actually some things happening before that.

There are 2 important phases for servlets. Every servlet is created once by tomcat (?) and with every request a new thread is created. So the 2 phases:

- **Servlet object creation** (just once will be instantiated)
- **New Thread creation** for that will execute that servlets method (happens many times)

There are some methods associated with these paheses. Methods which run before the doGet or doPost.

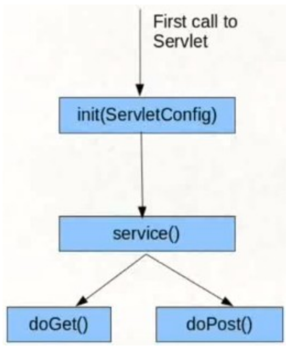
- **`init()`** : Corresponds to the first phase
- **`service()`** : Corresponds to the second phase

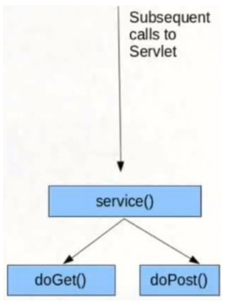
These are inherited methods from HTTPServlet class.

If the servlet isn't initialized yet, the init() method will run with ServletConfig argument. After that the service() method is called.

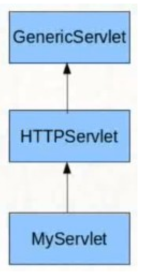
If the servlet object is already created, then the service method is called directly (?)

service methods decides which http method will run.





Servlet inheritance: HTTPServlet is one of the many types of servlet which are available. A servlet which uses the http protocol, there can be other servlet which use other protocols.



GenericServlet has **`init(ServletConfig)`**, **`init()`** and **`service()`** methods defined.

HTTPServlet overrides **`service()`** method (because of http purposes).

Our Servlet has **`doGet()`** and **`doPost()`** etc.

We can change some configurations about the init and service methods.

Raw servlets/JSP vs a framework like Spring MVC

(One or multiple servlets per webapp?)

Usually you will create a servlet per use case. Servlets acts like controllers for your application. When you identify an interaction from a user then implement a servlet to control that interaction.

That is, if you are using plain servlet/JSP to build the site. If you are using a framework like struts you will find that they implement the front controller pattern and use a single servlet that recieves all the requests and forwards these requests to action classes that implement the actual logic of the user request. this is much harder to do yourself but its a good practice...its the reason why so many people use these frameworks.

Most web frameworks use a dispatcher servlet (ex: Spring MVC) that takes care of routing requests to appropriate classes/controllers.

When you start having lots of pages, this approach works best because you have a more user friendly way (in regard to web.xml) of declaring/managing a class that handles http requests and its url. Example (spring mvc again). Also when you have a team of people working on a large web application, or you need to use Hibernate, then a framework like Spring really shines

Besides, having a dispatcher servlet keeps your code flow centralized.

I find that, with the addition of Spring version 3+, it becomes much more easier to bootstrap a Spring Web application with all the basics. The advantage of Spring MVC is that once you've bootstrapped the application context and the database connection, it becomes incredibly easy to create new controllers and it follows a much more logic architecture that newer developers may actually find easier as they get more familiar with it.

In fact, at my previous place of work, we were in the process of building a Java Servlet Web application, but we found that we had to create our own architecture or spine of the application and that is actually more work. Spring can take care of that which means that developers can get on with the actually application logic instead of worrying about the architecture too much.

Tomcat / JVM

Apache Tomcat is a Java servlet container, and is run on a Java Virtual Machine, or JVM. Tomcat utilizes the Java servlet specification to execute servlets generated by requests, often with the help of JSP pages, allowing dynamic content to be generated much more efficiently than with a CGI script.

If you want to run a high-performing installation of Tomcat, taking some time to learn about your JVM is essential. In this article, we'll learn how Tomcat and the JVM interact, look at a few of the different JVMs available, explain how to tune the JVM for better performance, and provide information about some of the tools available for monitoring your JVM's performance.

How Tomcat Interacts With The JVM

Utilizing servlets allows the JVM to handle each request within a separate Java thread, as each servlet is in fact a standard Java class, with special elements that allow it to respond to HTTP requests.

Tomcat's main function is to pass HTTP requests to the correct components to serve them, and return the dynamically generated results to the correct location after the JVM has processed them. If the JVM can't efficiently serve the requests Tomcat passes to it, Tomcat's performance will be negatively affected.

JSP (Java Server Pages)

One problem with using servlets: the html code needed for a simple web page is huge. So we cannot really write it in our servlet with a printwriter or something. Not maintainable.

JSP: Instead of having the html code inside our java servlets,it works the other way around: you have html code and inside of it you have java code. Just the dynamic portion of a web page will be generated by java code, the static parts will be constructed with html. This it the simple idea of jsp.

We need to mark our java code inside our html file so it wont be rendered as just text:

- **Scriplet** (Script) **tag:** <% my java code here %>
- **Expression tag:** <%= statement %>
- **Declaration tag:** <%! method or field declaration %>
- **Page directive**: <%@ directive attribute="value" %>

```
<% int k = 1 + 5;
out.println(“Value of k is: “ + k);  %>
```

<% java source code %>

This will print the output to our html file. If you write this in your html file ofc.

Prints the value of k in our html: Html code + Value of k is: <%=k %>

<%= statement %>

That can be also an expression like **<%= 1+2+3 %>** writes 6 to our html file.

Declaration tag: <%! method %>

→ This method will be available to all these script tags (<% %>). We can have n number of script tags where we execute code.

<%! field or method declaration %>

```
<%!
public int add(int arg1, int arg2) {
      return arg1+ arg2;
}
%>
```

Now the other tags can use this method:

```
<% k= add(123,423); %>
<br>
Value of k: <%=k %>
```

Where are the classes??? (Scopes and PageContext didn’t watch)

- You cannot have methods inside a script tag.
- You can have text between 2 script tags and those tags will be treated like they were one/togehter.

The whole jsp file is converted to a java class/servlet by tomcat. Every jsp is actually a different way to write a servlet. It’s actually this servlet that runs.

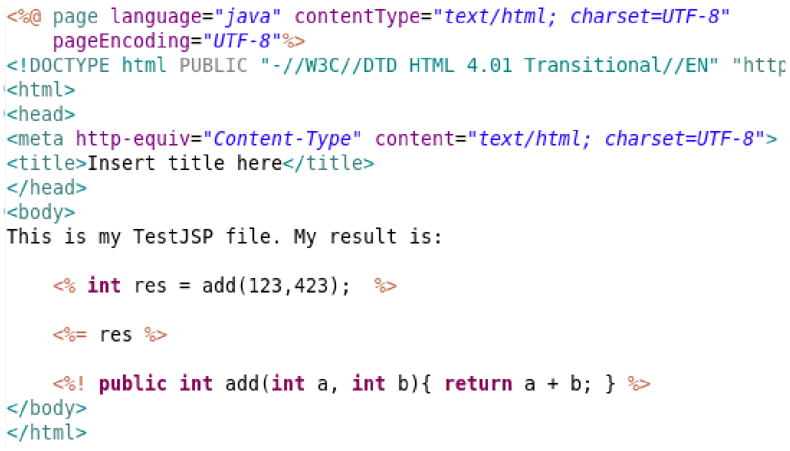
Every code inside a scriplet is converted to code inside **doGet()** method.

Every static text inside the html file is also printed (as we did before) **PrintWriter.println()**. **out.write()**

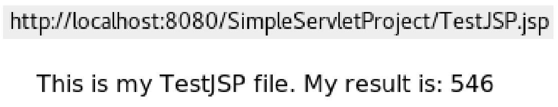
When we tried to define a method inside a script tag, it gave us an error. Now we know exactly why: everything inside the script tags goes to the method body of doGet/doPost… and since we cannot have a method inside another method, it shows us an error.

Every declaration tag forms a separate method/field, outside other methods.

We can actually look up what kind of a class tomcat generates from our jsp file. It's under the folder called "**work**". Can found it here: `projectworkspace/.metadata/.plugins/org.eclipse.wst.server.core/tmp0`



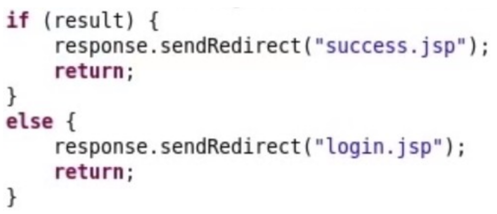
```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
This is my TestJSP file. My result is:

    <% int res = add(123,423);  %>

    <%= res %>

    <%! public int add(int a, int b){ return a + b; } %>
</body>
</html>
```



http://localhost:8080/SimpleServletProject/TestJSP.jsp

This is my TestJSP file. My result is: 546

Bu şekilde Servletımızın (controller) içinden hangi jsp file'ı göstermek istiyorsak yazabiliyoruz. (jsp file'ları da view olmuş oluyor)



```
if (result) {
    response.sendRedirect("success.jsp");
    return;
}
else {
    response.sendRedirect("login.jsp");
    return;
}
```

response.sendRedirect takes the path of the servlet!

- Accessing data that is shared between the servlet and the jsp can be done for example with the **session object** as we talked before. Servlet can write user data to session. Jsp can retrieve session and do **`session.getAttribute("key");`** and then present it as a html

Mvc Ingredients

- **Controller** : LoginServlet
- **Business Service**: AuthenticationService
- **Model**: AuthenticationResult
- **View**: Greeting Page, Login Page

The model is the data that flows between these components.

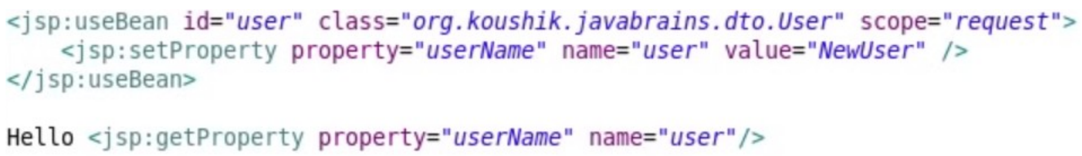
Customer → Waiter → Cook

Cook → Waiter → Presenter → Customer.

JSTL: **JSP** supports taglibs. A well known **taglib** is **JSTL**.

Problems with normal jsp tags: As your code increases, it becomes really hard to maintain. Also the code in these tags are not proper xml.

JSTL can be used in this case. JSTL helps us to write all this java code as xml. The whole document will be proper xml and it will be easier to maintain. <u>JSTL is comman and it is bad practice to use script tags</u>.



```
<jsp:useBean id="user" class="org.koushik.javabrains.dto.User" scope="request">
    <jsp:setProperty property="userName" name="user" value="NewUser" />
</jsp:useBean>

Hello <jsp:getProperty property="userName" name="user"/>
```