Servlets

Table of Contents

[Common Gateway Interfaces 2](#_Toc100518773)

[GET and POST Requests 4](#_Toc100518774)

[Terminology 5](#_Toc100518775)

[Lifecycle 7](#_Toc100518776)

[HTML Forms Overview 10](#_Toc100518777)

[Text and Passwords 11](#_Toc100518778)

[Checkboxes 12](#_Toc100518779)

[Radio Buttons 12](#_Toc100518780)

[Text Area 13](#_Toc100518781)

[Dropdown List 14](#_Toc100518782)

[Setting Up Servlets 15](#_Toc100518783)

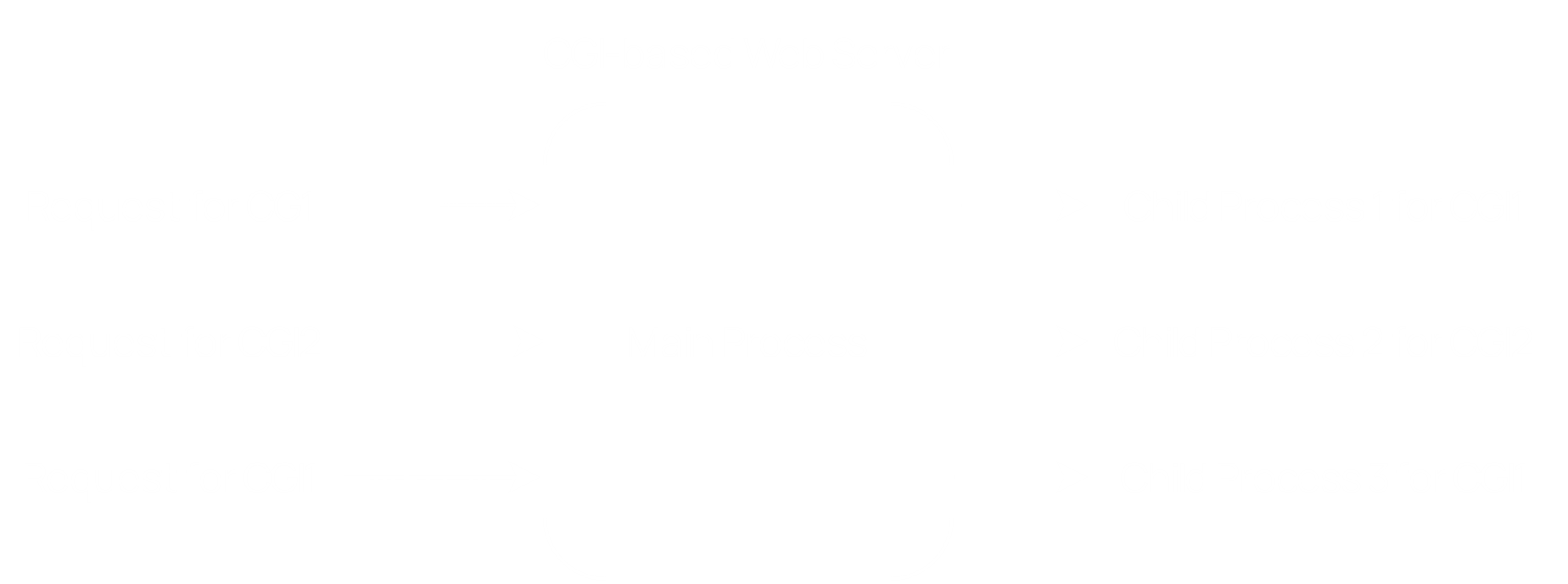
A **servlet** is a Java class that can extend the capabilities of servers that host applications which are accessed via a request-response model, although servlets can respond to any type of request. In simpler terms, it is a Java class that can take requests, process information, create an output and send back a response in the form of a web page.

Thus, the difference between a normal response and a servlet response is that a servlet response is **dynamic**, based on the information being processed, while a normal response is not. For example, we can send input to the servlet and get a response that provides us the output after processing was done on the input.

There are many types of servlets, but we will be looking into **HTTP Servlets** specifically.

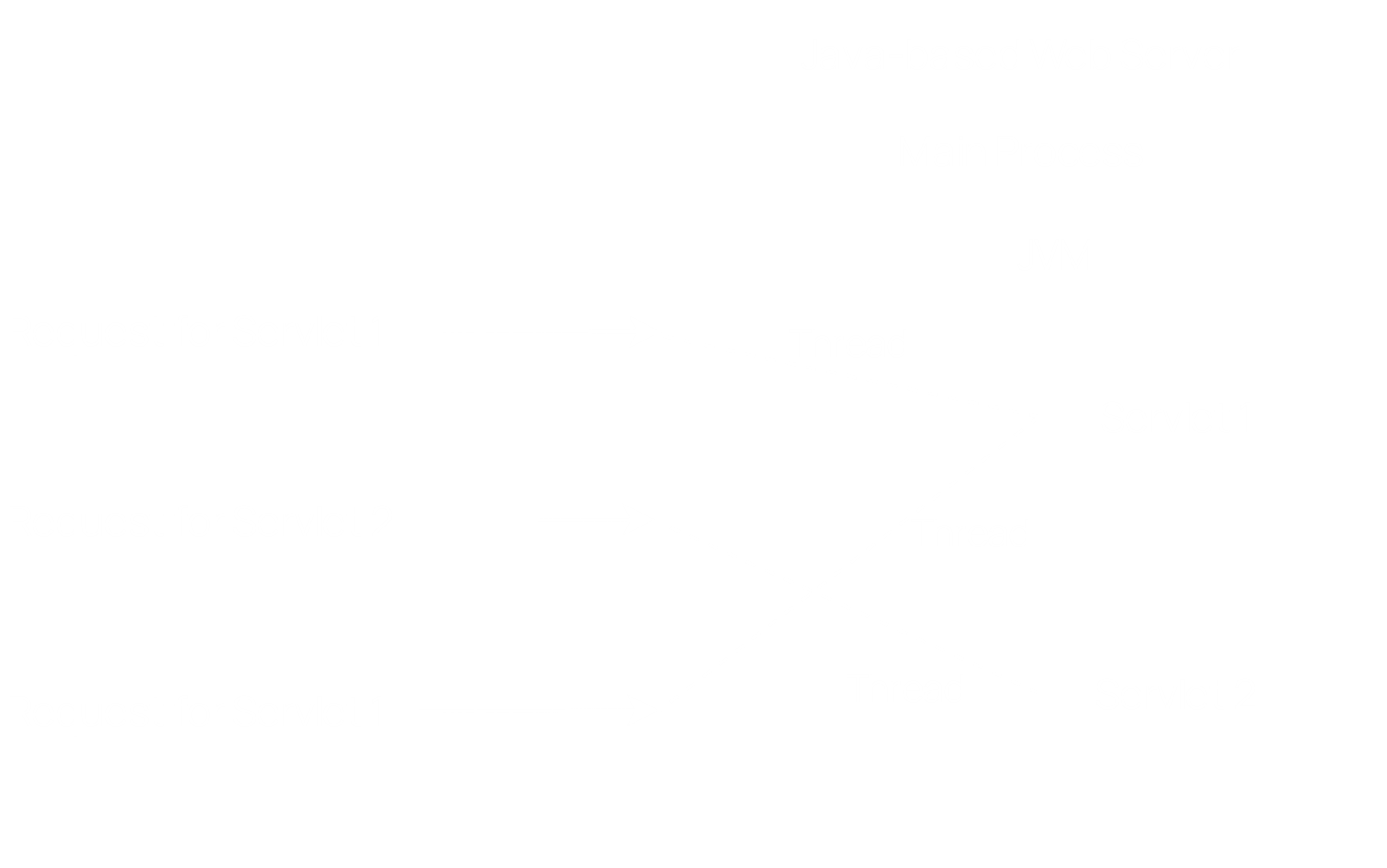
## Common Gateway Interfaces

When the Internet was first created, the content on it was all **static**. Soon after, people realized the need for **dynamic** content and created **Common Gateway Interfaces** (CGIs). With CGI, a webserver passes certain requests to an external program, gets the output from the program and sends back the output to the client instead of a static file.



As can be seen in the diagram above, for each client request for a particular CGI, a **new process** is created for that CGI to deal with the request. This is a problem. For every new process, the server must pass a whole lot of information, such as environment variables and other input. This takes time and uses up a lot of resources, which limits how many requests we can serve at a time. Additionally, the CGIs are separate from the server itself, meaning information common to each has to be passed back and forth. For example, a CGI cannot write to the server’s log file.

This is where **Java Servlets** come in. A single servlet creates just **one process**. When a client request arrives, a **new thread** for the required process is created, which serves the request. As opposed to separate processes, separate threads of the same process share a memory space, so we do not need to pass a lot of information to each new thread. This saves time and resources. Additionally, threads can run parallelly, which also contributes to improving scalability. Java Servlets run inside a JVM, which makes them safe and portable. All major web servers support servlets.



## GET and POST Requests

When a client connects to a server and makes an HTTP request, it can be of several types, called **methods**. The most frequently used methods are GET and POST methods.

The **GET method** is used to get information, such as a document or results for a database query. It can include some extra information to better describe what is required, passed as a sequence of characters appended to the URL, called a **query string**. Usually however, the length of the URL is limited to **240 characters** by servers, which means the extra information we can send is limited. Additionally, the URL is visible to all, which makes it inappropriate to pass **sensitive information** like login credentials.

The **POST method** is designed to post information to be stored in the database, such as new data. The data is passed as part of the HTTP **request body**. This allows it to send data of **unlimited length** and also pass **sensitive information**.

## Terminology

* **Web Server** – This is a server capable of handling HTTP requests sent by a client and responding back with an HTTP response.
* **Web Container** – Also called a Servlet Container or a Servlet Engine, this is a component inside the web server which is responsible for handling the life cycle of servlets. It uses a deployment descriptor to map URLs to the correct servlets and ensures that the client requesting the URL has the proper access rights. Some famous web containers are Apache Tomcat, Apache Geronimo, Glassfish from Oracle, etc.
* **Deployment Descriptor** – This is either an XML file or annotations.

For an XML file, web.xml, the servlet to be called for different requests are specified.

<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"  
 version="4.0">  
 <servlet>  
 <servlet-name>HelloWorld</servlet-name>  
 <servlet-class>HelloWorld</servlet-class>  
 </servlet>  
 <servlet-mapping>  
 <servlet-name>HelloWorld</servlet-name>  
 <url-pattern>/HelloWorld</url-pattern>  
 </servlet-mapping>  
</web-app>

XML

Annotations on the other hand, do not require an extra XML file. They are placed directly with the code of the servlet itself. Note however, that annotations are only available from Java Servlet version 3.0 onwards.

@WebServlet(name="HelloWorld")  
public class HelloWorld extends HttpServlet {  
 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
   
 }  
}

JAVA

* **WAR and WAR Exploded** – Web Application Resource or Web Application Archive (WAR) is a packaged file that stores all the components of the web application and can be deployed to web servers. However, repeatedly packaging the components is troublesome during the development stage, so the components are directly used without packaging them. This is called WAR Exploded. It allows us to make immediate and quick changes to our code, called hot deployment.

## Lifecycle

Consider the sample code for a servlet provided below:

import java.io.\*;  
import javax.servlet.http.\*;  
import javax.servlet.annotation.\*;  
  
@WebServlet(name = "helloServlet", value = "/hello-servlet")  
public class HelloServlet extends HttpServlet {  
 private String message;  
  
 public void init() {  
 message = "Hello World!";  
 }  
  
 public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {  
 response.setContentType("text/html");  
  
 // Hello  
 PrintWriter out = response.getWriter();  
 out.println("<html><body>");  
 out.println("<h1>" + message + "</h1>");  
 out.println("</body></html>");  
 }  
  
 public void destroy() {  
 }  
}

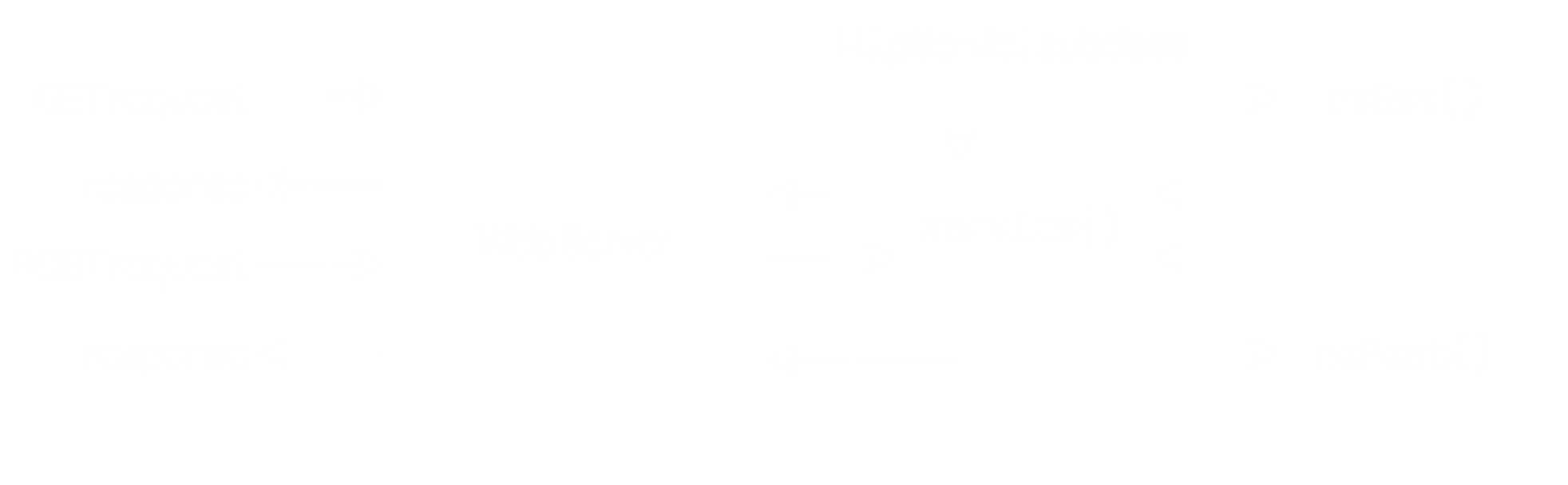
JAVA

Over the lifecycle of a servlet, there are several methods which get called in a specific order.

The first method that gets called is init(). This is called **only once**, when the servlet is first created and not when users make requests.

All Java servlets inherit from a **base servlet class**. This class is not for any specific type of servlet, but is generic. The HelloServlet class also inherits from this.

For every client request, the service() method gets called. This method comes from the base servlet class and is the one that actually performs the task. Depending on the type of task, i.e. whether it is a GET request, or a POST request, etc., a **new thread** gets created and handed over to a different method, such as doGet() or doPost(). The HelloServlet class does not override the service() method. It contains the doGet() method, the doPost() method and so on. When we create a new servlet, we override these methods.



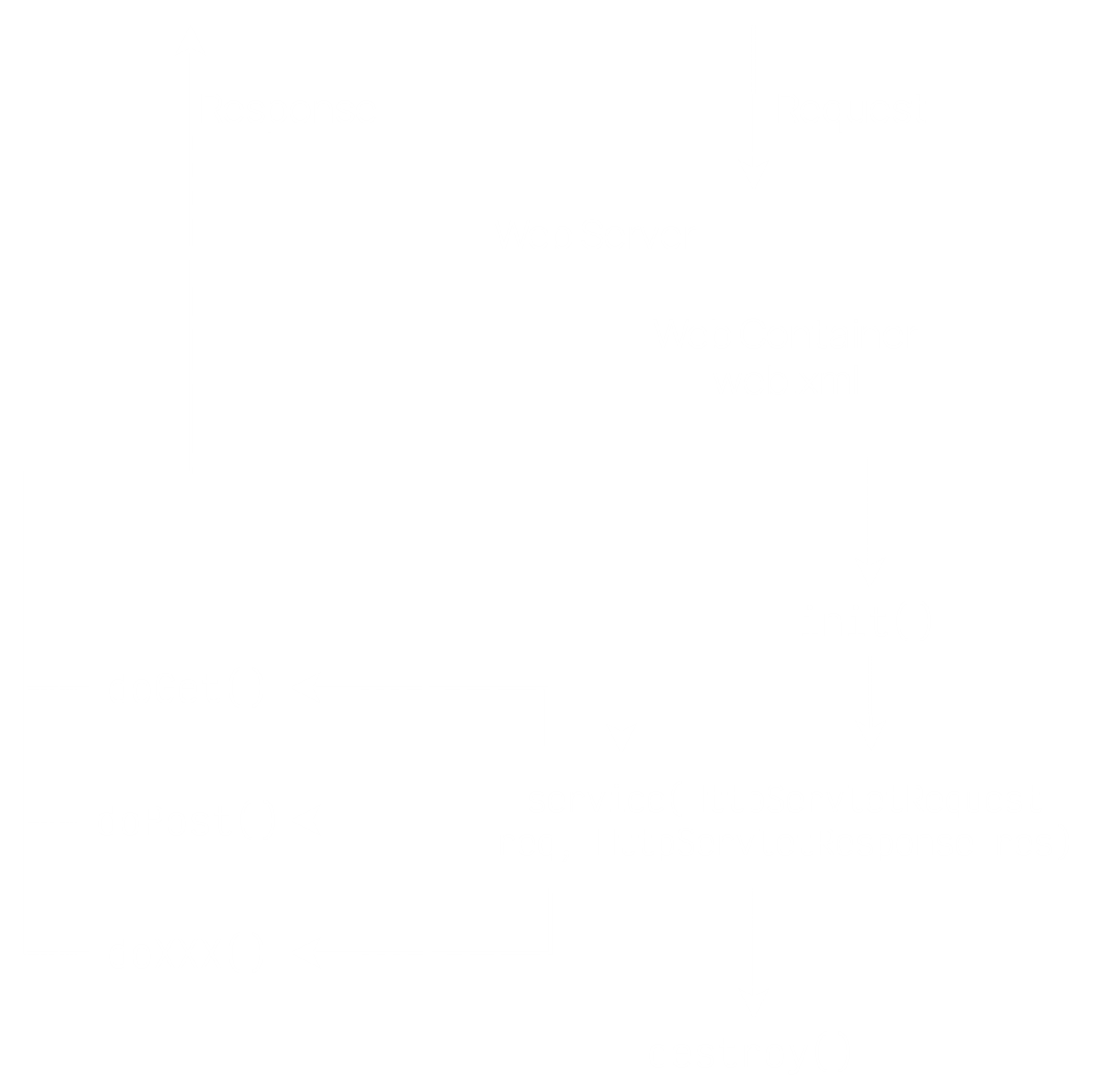
For example, in the sample code above, we use a PrintWriter object to write HTML text to the response.

All methods that derive from the service method have two parameters, HttpServletRequest and HttpServletResponse, which correspond to the request from the client and the response to be sent to the client respectively.

Finally, we have the destroy() method, which is called at the end of the servlet’s life cycle. Here, we can do things like close database connections, halt background threads, write cookie lists or hit counts to disk, etc. This method marks the servlet for garbage collection.

Thus, the steps are:

1. The client sends a request.
2. The web server accepts the request and forwards it to the web container.
3. The web container searches the web.xml file for the requested URL pattern and retrieves the address of the servlet.
4. If the servlet has not yet been initialized, the init() method is called.
5. The web container calls the public service() method, passing the ServletRequest and ServletResponse objects.
6. The public service() method typecasts the ServletRequest and ServletResponse objects to HttpServletRequest and HttpServletResponse objects respectively.
7. The public service() method calls the protected service() method, which checks the client request and calls the corresponding do\_\_() method as required.
8. The request is handled by sending the result generated by the do\_\_() to the client.

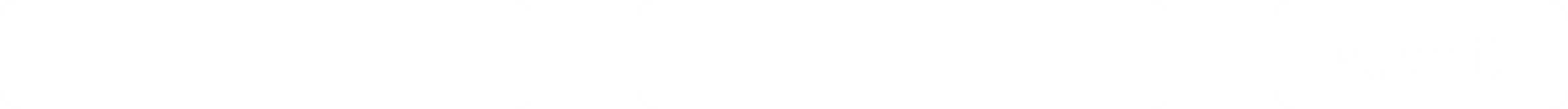


## HTML Forms Overview

Handling forms is a very common task in HTML since it allows us to get information from the user. An HTML form is encapsulated using <form> tags and contains many input types.

<html>  
<head>  
 <title>Hello, I am a Java web app!</title>  
</head>  
<body>  
<form method="get" action="HelloWorld">  
 <input type="text" name="sampleText"/>  
 <input type="password" name="samplePassword"/>  
 <input type="submit" value="submit"/>  
</form>  
</body>  
</html>

HTML



The <form> tag itself has two attributes, method and action. The method tag defines the type of the request, get, post, etc. The action tag defines the name of the servlet which will handle the request.

The submit input is special. It creates a button to allow the user to submit the form.

We will now be looking into the different input types.

### Text and Passwords

In the webpage:

Username: <input type="text" name="username"/>  
Password: <input type="password" name="pw"/>

HTML



In the servlet:

String username = request.getParameter("username");  
String password = request.getParameter("pw");

JAVA

Note that the parameter value in the request is always a **string**.

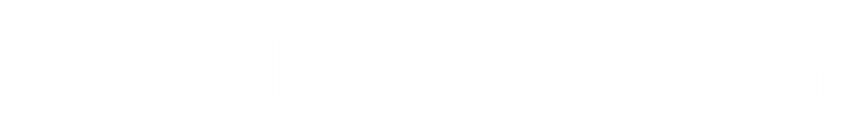
### Checkboxes

Checkbox inputs that share the same name will be considered to be part of the same group.

In the webpage:

<input type="checkbox" name="language" value="english"/>English  
<input type="checkbox" name="language" value="french"/>French

HTML



On the server end, we receive an array of the values of the checked box.

In the servlet:

String languages[] = request.getParameterValues("language");

JAVA

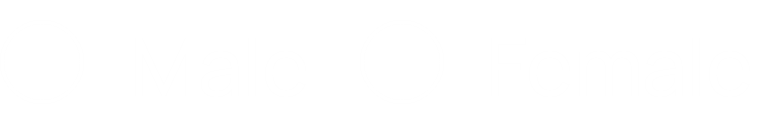
### Radio Buttons

Like checkboxes, radio buttons that have the same name will be consider part of the same group.

In the webpage:

<input type="radio" name="gender" value="male"/>Male  
<input type="radio" name="gender" value="female"/>Female

HTML



In the servlet:

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

JAVA

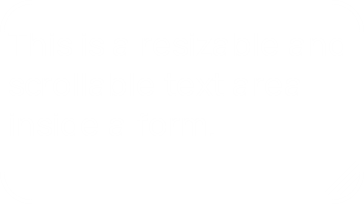
### Text Area

A **Text Area** allows the user to enter a larger amount of text.

In the webpage:

<textarea rows="5" cols="30" name="feedback"></textarea>

HTML



In the servlet:

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

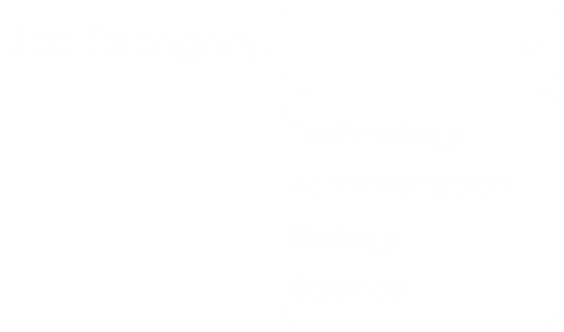
JAVA

### Dropdown List

In the webpage:

Job Category:  
<select name="jobCat">  
 <option value="tech">Technology</option>  
 <option value="admin">Administrator</option>  
 <option value="biology">Biology</option>  
 <option value="science">Science</option>  
</select>

HTML



In the servlet:

String jobCategory = request.getParameter("jobCat");

JAVA

## Setting Up Servlets

To start with servlets, we will require **IntelliJ IDEA Ultimate Edition** as well as **Apache Tomcat**.

1. Start a new project in IntelliJ IDEA.
2. Choose **Java Enterprise**, **Web Application** for the template, and **Tomcat** for the application server. You will need to locate the Tomcat folder if this is the first time you are using it. Also choose **Java** as the language, and **Maven** as the build system.
3. On the second page of the project setup, check **Web Profile** under **Specifications**.
4. In the project folder, under src → main → java, we have .java files, under src → main → webapp, we have .jsp files, which are a combination of Java and HTML and under src → main → webapp → WEB-INF we have the web.xml file. There should only be one .jsp file, index.jsp. This is the starting point of our application, the first webpage.
5. There should be a default HelloServlet.java file created and opened. Run the project using the **play button** on the top-right to see what this looks like.
6. If it does not say **Tomcat** next to the play button, you may need to **add a configuration**. Use a **Tomcat Local Server**. Keep the default settings.
7. While setting up the configuration, there may be a warning that says **No artifacts configured**. Clicking the **Fix** button should take you to a window where artifacts can be added. Add a **Web Application: Exploded** artifact.
8. Updates made to the servlets should automatically be reflected in the webpages. However, due to caching, the updates may not appear. In this case, **restart** the server.