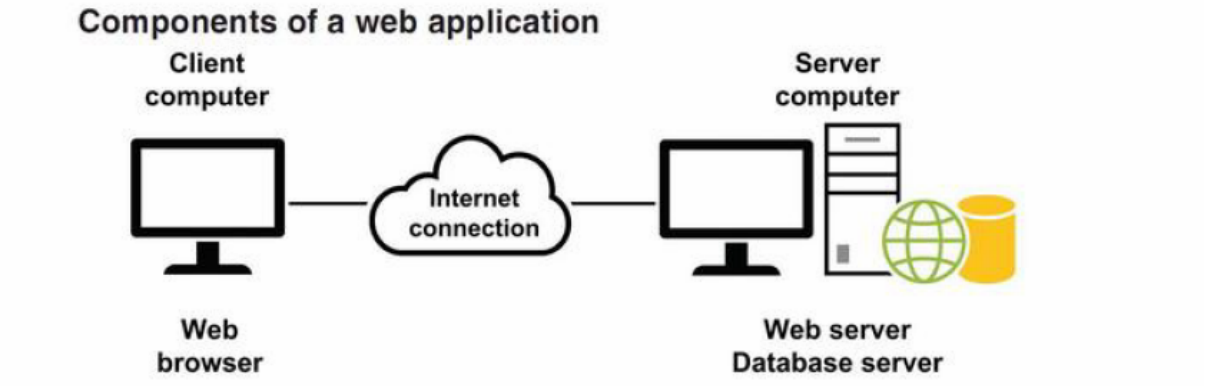
Module-3
SERVLET & JSP

Web application :
A **web application** is a software application that runs in a web browser. Unlike traditional desktop applications, web applications do not require installation on a user's device. They are accessed via the internet using a web browser like Google Chrome, Firefox, or Edge.
Key Features of Web Applications:

- **Accessible via a browser** (e.g., Gmail, Facebook, Google Docs).
- **No installation required** on the user's device.
- **Platform-independent**, working on Windows, macOS, Linux, and mobile devices.
- **Requires an internet connection**, though some web apps offer offline functionality.

Components of a Web Application:



Description

- Web applications are a type of *client/server application*. In a client/server application, a user at a *client* computer accesses an application at a *server* computer. For a web application, the client and server computers are connected via the Internet or an intranet.
- In a web application, the user works with a *web browser* at the client computer. The web browser provides the user interface for the application. One widely used web browser is Google Chrome, but other web browsers such as Mozilla Firefox and Internet Explorer are also widely used.
- A web application runs on the server computer under the control of *web server* software. The Apache server is one of the most widely used web servers.
- For most web applications, the server computer also runs a *database management system* (*DBMS*), which is also known as a *database server*. For servlet and JSP applications, Oracle and MySQL are two of the most popular database management systems.

How static web pages work

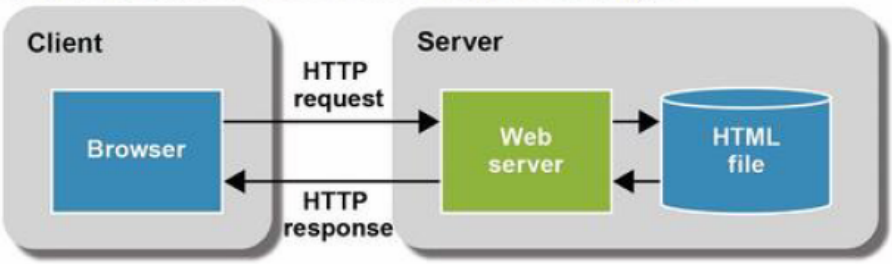
HTML (*Hypertext Markup Language*) is the language that the browser renders to the web pages that make up a web application's user interface. Some web pages are *static web pages*, which are the same each time they are viewed. In other words, they don't change in response to user input.

Figure 1-3 shows how a web server handles static web pages. The process begins when a user at a web browser requests a web page. This can occur when the user enters a web address into the browser's Address box or when the user clicks a link that leads to another page. In either case, the web browser uses a standard Internet protocol known as *Hypertext Transfer Protocol* (*HTTP*) to send a request known as an *HTTP request* to the website's server.

When the web server receives an HTTP request from a browser, the server gets the requested HTML file from disk and sends the file back to the browser in the form of an *HTTP response*. The HTTP response includes the HTML document that the user requested along with any other resources specified by the HTML code such as graphics files.

When the browser receives the HTTP response, it renders the HTML document into a web page that the user can view. Then, when the user requests another page, either by clicking a link or typing another web address in the browser's Address box, the process begins again.

How a web server processes static web pages



Description

- *Hypertext Markup Language* (*HTML*) is the language that the web browser converts into the web pages of a web application.
- A *static web page* is an HTML document that's stored in a file and does not change in response to user input. Static web pages have a filename with an extension of .htm or .html.
- *Hypertext Transfer Protocol* (*HTTP*) is the protocol that web browsers and web servers use to communicate.
- A web browser requests a page from a web server by sending the server a message known as an *HTTP request*. For a static web page, the HTTP request includes the name of the HTML file that's requested.
- A web server replies to an HTTP request by sending a message known as an *HTTP response* back to the browser. For a static web page, the HTTP response includes the HTML document that's stored in the HTML file.

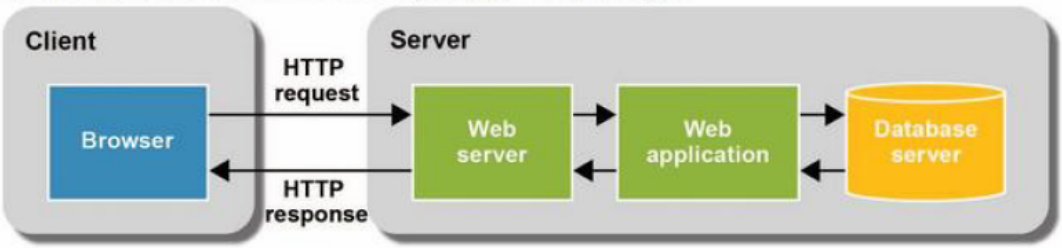
How dynamic web pages work

In contrast to a static web page, a *dynamic web page* changes based on the parameters that are sent to the web application from another page. For instance, when the Add To Cart button in the first page in figure 1-1 is clicked, the static web page calls the web application and sends one parameter to it. Then, the web application generates the dynamic web page and sends the HTML for it back to the browser.

Figure 1-4 shows how this works. When a user enters data into a web page and clicks the appropriate button, the browser sends an HTTP request to the server. This request contains the address of the next web page along with any data entered by the user. Then, when the web server receives this request and determines that it is a request for a dynamic web page, it passes the request back to the web application.

When the web application receives the request, it processes the data that the user entered and generates an HTML document. Next, it sends that document to the web server, which sends the document back to the browser in the form of an HTTP response. Then, the browser displays the HTML document that's included in the response so the process can start over again.

How a web server processes dynamic web pages



Description

- A *dynamic web page* is an HTML document that's generated by a web application. Often, the web page changes according to parameters that are sent to the web application by the web browser.
- When a web server receives a request for a dynamic web page, the server passes the request to the web application. Then, the application generates a response, such as an HTML document, and returns it to the web server. The web server, in turn, wraps the generated HTML document in an HTTP response and sends it back to the browser.
- Most modern web applications store and retrieve data from a database that runs on a database server.
- The browser doesn't know or care whether the HTML was retrieved from a static HTML file or was dynamically generated by the web application. Either way, the browser displays the HTML document that is returned.

Three approaches for Java web applications

There are many ways to develop Java web applications. Figure 1-5 describes three approaches that are commonly used today. When developing Java web applications, you typically use parts of the *Java Enterprise Edition* (*Java EE*) specification. This specification describes how web servers can interact with all Java web technologies including servlets, JavaServer Pages (JSP), JavaServer Faces (JSF), Java Persistence API (JPA), Enterprise JavaBeans (EJB), and more.

Servlet/JSP

In a well-structured servlet/JSP application, *servlets* store the Java code that does the server-side processing, and *JavaServer Pages* (*JSPs*) store the HTML that defines the user interface. This HTML typically contains links to CSS and JavaScript files. To run a web application that uses servlets and JSPs, you only need to work with the servlet/JSP part of the Java EE specification.

Since the servlet/JSP API is a relatively low-level API, it doesn't do as much work for the developer as the other two APIs. However, the servlet/JSP API gives the developer a high degree of control over the HTML, CSS, and JavaScript that's returned to the browser. In addition, the servlet/JSP API is the foundation for the other two approaches. As a result, it's a good place to get started with Java web programming. As you progress through this book, you'll learn how to develop servlet/JSP applications.

JSF

JavaServer Faces (*JSF*) is a newer technology that's designed to replace both servlets and JSPs. It provides a higher-level API that does more work for the programmer. When you use JSF, you typically use more Java EE features than you do with the servlet/JSP approach.

When you use JSF, you can also use *Enterprise JavaBeans* (*EJBs*) to define server-side components. Although there are some benefits to using EJBs, they're overkill for most websites. As a result, this book doesn't show how to use them.

Spring Framework

Like JSF, the Spring Framework is a higher-level API that does more work for the programmer than the servlet/JSP API. However, due to the way it's structured, the Spring Framework still gives the developer a high degree of control over the HTML/CSS/JavaScript that's returned to the browser. As a result, if control over HTML/CSS/JavaScript is a priority for your website, the Spring Framework might be the right approach for you.

Servlet /Jsp based web application

The software components

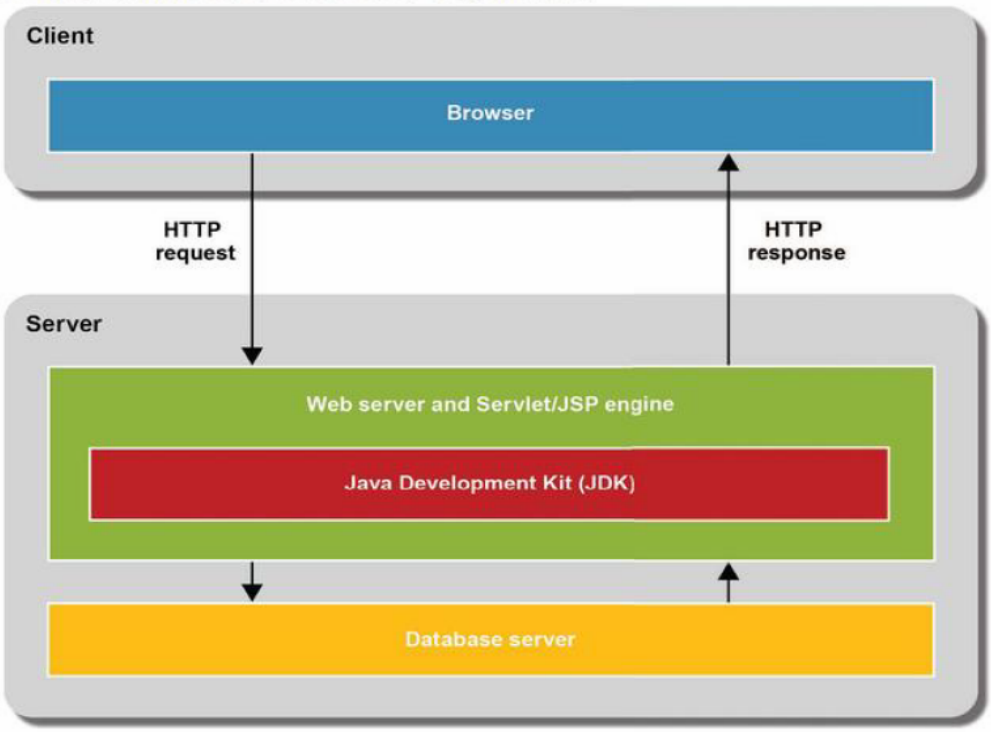
Figure 1-6 shows the primary software components for a servlet/JSP web application. By now, you should understand why the server must run web server software. In addition, to work with servlets and JSPs, the server must also run a *servlet/JSP engine*, which is also known as a *servlet/JSP container*. In this book, you'll learn how to use the Tomcat server. This server is one of the most popular servers for Java web applications, and it includes both a web server and a servlet/JSP engine.

For a servlet/JSP engine to work properly, the engine must be able to access the *Java Development Kit (JDK)* that comes as part of the *Java Standard Edition (Java SE)*. The JDK contains the Java compiler and the core classes for working with Java. It also contains the *Java Runtime Environment (JRE)* that's necessary for running compiled Java classes. Since this book assumes that you already have some Java experience, you should already be familiar with the JDK and the JRE.

Since all servlet/JSP engines must implement the servlet/JSP part of the Java EE specification, all servlet/JSP engines should work similarly. In theory, this makes servlet/JSP code portable between servlet/JSP engines and application servers. In practice, though, there are minor differences between each servlet/JSP engine and web server. As a result, you may need to make some modifications to your code when switching servlet/JSP engines or web servers.

Since most servlet/JSP web applications store their data in a database, the server typically runs a database server too. In this book, you'll learn how to use MySQL as the database server. This software is open-source and commonly used with servlet/JSP applications.

The components of a servlet/JSP application



Description

- A servlet/JSP application must have a web server and a *servlet/JSP engine*, also known as a *servlet/JSP container*, to process the HTTP request and return an HTTP response, which is typically an HTML page. Most servlet/JSP applications use Tomcat as both the web server and the servlet/JSP engine.
- Most servlet/JSP applications use a database to store the data that's used by the application. Many servlet/JSP applications use MySQL as the database, though there are many other databases to use.
- For a servlet/JSP engine to work, it must have access to Java's *Java Development Kit* (*JDK*), which comes as part of the *Java Standard Edition* (*Java SE*). Among other things, the JDK contains the core Java class libraries, the Java compiler, and the *Java Runtime Environment* (*JRE*).

The architecture

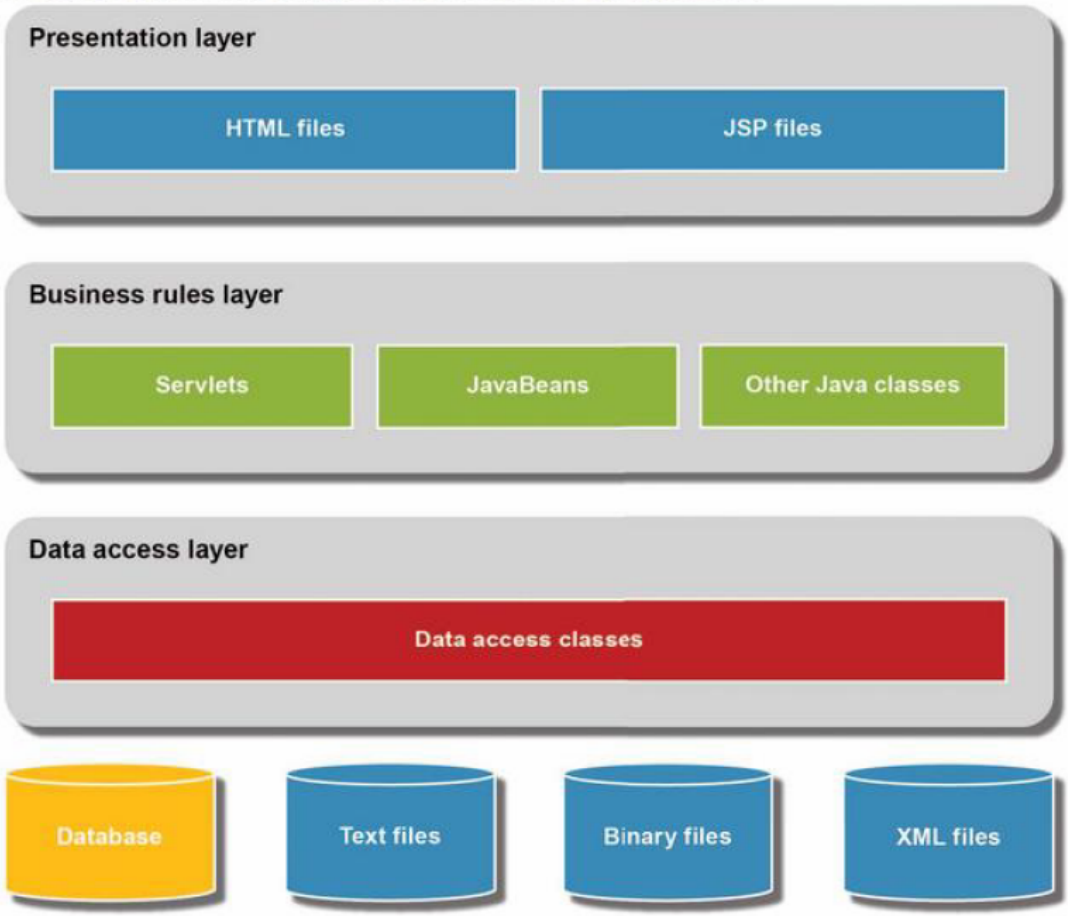
Figure 1-7 shows the architecture for a typical web application that uses servlets and JSPs. This architecture uses three layers: (1) the *presentation layer*, or *user interface layer*, (2) the *business rules layer*, and (3) the *data access layer*. In theory, the programmer tries to keep these layers as separate and independent as possible. In practice, though, these layers are often interrelated, and that's especially true for the business and data access layers.

The presentation layer consists of HTML pages and JSPs. Typically, a web designer works on the HTML stored in these pages to create the look and feel of the user interface. Later, a Java programmer may need to edit these pages so they work properly with the servlets of the application.

The business rules layer uses servlets to control the flow of the application. These servlets may call other Java classes to store or retrieve data from a database, and they may forward the results to a JSP or to another servlet. Within the business layer, Java programmers often use a special type of Java class known as a *JavaBean* to temporarily store and process data. A JavaBean is typically used to define a business object such as a User or Invoice object.

The data layer works with the data of the application on the server's disk. Typically, this data is stored in a relational database such as MySQL. However, this data can also be stored in text files, binary files, and XML files. Or, it can come from web services running on the other servers.

The architecture for a typical servlet/JSP application



Description

- The *presentation layer* for a typical servlet/JSP web application consists of HTML pages and JSPs.
- The *business rules layer* for a typical servlet/JSP web application consists of servlets. These servlets may call other Java classes including a special type of Java class known as a *JavaBean*. As you progress though this book, you'll learn how to use several special types of tags within a JSP to work with JavaBeans.
- The *data access layer* for a typical Java web application consists of classes that read and write data that's stored on the server's disk drive.
- For most web applications, the data is stored in a relational database such as MySQL. However, it may also be stored in binary files, text files, or XML files.

Software for developing a java web-application:

To develop a Java web application, you need several software tools, including an IDE, a web server, a database, and frameworks. Here are the essential tools:

1. Integrated Development Environments (IDEs)
IDEs help in writing, debugging, and running Java web applications efficiently.

- Eclipse – Free and widely used for Java EE development.
- IntelliJ IDEA – Powerful, with better code suggestions (Community & Ultimate versions).
- NetBeans – Supports Java EE.

2. Java Development Kit (JDK)

- Install the latest JDK (Java SE or Java EE) from Oracle or OpenJDK.

3. Web/Application Servers
Web servers run Java web applications using Servlets, JSP, or Spring Boot.

- Apache Tomcat – Lightweight, supports Servlets and JSP.
- WildFly (JBoss) – Full Java EE support.
- GlassFish – Open-source, Java EE-compliant.
- Spring Boot Embedded Servers – Comes with built-in Tomcat, Jetty, or Undertow.

4. Java Web Frameworks
Frameworks simplify web development by handling MVC (Model-View-Controller) architecture.

- **Spring Boot** – Modern, fast, and widely used for microservices.
- **JavaServer Faces (JSF)** – Oracle-backed UI framework.
- **Struts** – Good for MVC-based enterprise applications.
- **Play Framework** – Asynchronous and reactive web applications.

5. Build Tools
Manage dependencies and automate builds.

- **Maven** – XML-based dependency management.
- **Gradle** – More flexible than Maven, uses Groovy or Kotlin.

Introduction to MVC Architecture

1. What is MVC?

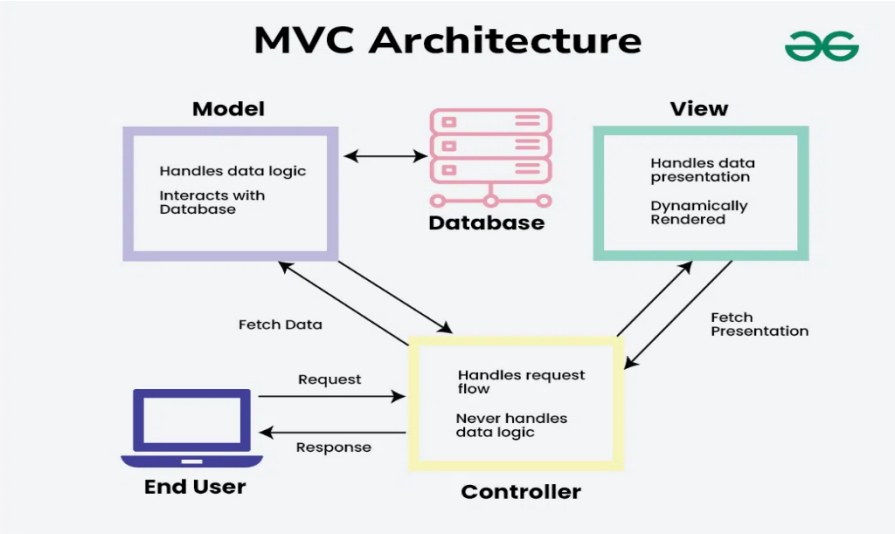
MVC stands for Model-View-Controller. It is a design pattern that helps organize code in a structured way.

- Model → Handles data and business logic
- View → Displays data (User Interface)
- Controller → Controls user interactions and updates Model & View

Example Analogy

Think of MVC as a restaurant setup:

- Model (Chef) → Prepares the food (data & logic)
- View (Waiter/Plate) → Presents the food to the customer (UI)
- Controller (Waiter) → Takes orders from customers and tells the chef what to prepare (Handles input)



2. Why Use MVC?

✓ Organized Code – Each part has a specific role

✓ Easier to Maintain – Changes in one part won't break everything

✓ Reusability – You can reuse components in different projects

✓ Better Debugging – Finding and fixing issues is easier

3. Components of MVC

1. Model (Data Layer)

- Represents the data and business logic.
- Communicates with the database or external sources.
- Notifies the View when data changes.

2. View (Presentation Layer)

- Displays data to the user.
- Does not contain business logic.
- Can be JSP, HTML, or JavaFX components in Java applications.

3. Controller (Logic & Flow)

- Handles user requests and updates the Model & View.
- Acts as a bridge between the View and Model.

Example in Java:

1. Model (Data Layer)

```
public class Student {
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

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

    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

2. View (Presentation Layer)

Example:

```
public class StudentView {
    public void displayStudentDetails(String name, int age) {
        System.out.println("Student: " + name + ", Age: " + age);
    }
}
```

3. Controller (Logic & Flow)

Example:

```
public class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view) {
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name) {
        model.setName(name);
    }

    public void updateView() {
        view.displayStudentDetails(model.getName(), model.getAge());
    }
}
```

4. How MVC Works Together

1. **User interacts with the View.**
2. **Controller handles user input.**
3. **Controller updates the Model.**
4. **Model notifies the View of changes.**
5. **View updates and presents data.**

5. Example Workflow

```
public class MVCDemo {
    public static void main(String[] args) {
        // Create Model
        Student student = new Student("Alice", 22);

        // Create View
        StudentView view = new StudentView();

        // Create Controller
        StudentController controller = new StudentController(student, view);

        // Update and display data
        controller.updateView();
        controller.setStudentName("Bob");
        controller.updateView();
    }
}
```

6. MVC in Real-World Java Applications

- **Web Applications**: Spring MVC (Spring Framework)
- **Desktop Applications**: Java Swing, JavaFX
- **Android Apps**: Uses an MVVM (variant of MVC)

7. Conclusion

- **MVC improves code organization and maintainability.**
- **Commonly used in Java-based applications, including Spring MVC and JavaFX.**
- **Understanding MVC is essential for advanced Java development.**

What is a Servlet

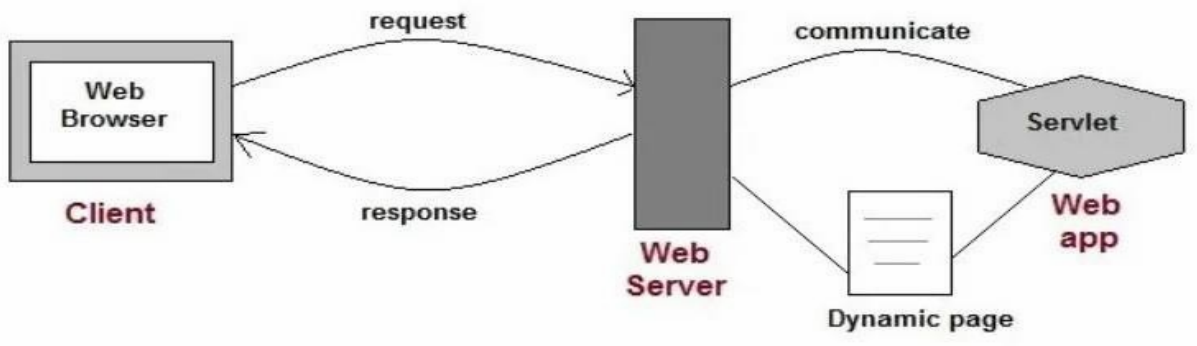
A **Servlet** is a Java program that runs on a web server and handles **requests and responses** from clients (like web browsers). It is used for creating **dynamic web applications** by processing user inputs, interacting with databases, and generating web content dynamically.

Describing Servlets

- Servlet is a Java Technology for server-side programming.
- It is a Java module that runs inside a Java-enabled web server and services the requests obtained from the web server.
- Servlets are the Java programs that run on the Java-enabled web server or application server.
- They are used to handle the request obtained from the webserver, process the request, produce the response, then send a response back to the webserver.
- Working with Servlets is an important step in the process of Application development and delivery through the Internet.
- A Web application is sometimes called a Web app, Thus, it is an application that is accessed using a Web browser over the network such as the Internet or an Intranet.

Working With Servlets

Working with Servlets is an important step in the process of Application development and delivery through the Internet.



- Servlets are not tied to a specific client-server protocol, but are most commonly used with HTTP.
- Execution of servlet consists of 4 steps:
  - The client sends request to server
  - The server alerts appropriate servlet
  - The servlet processes the request and generates the output (if any), and sends back to the Webserver
  - The Webserver sends back response to the client

Applications of Servlet :

- Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
- Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
- Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
- Send 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), Excel, etc.
- Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Jakarta Servlet API
Servlets are part of the jakarta.servlet package.
Main Classes and Interfaces:
for more details: https://en.wikipedia.org/wiki/Jakarta_Servlet

Class / Interface	Description
`jakarta.servlet.Servlet`	Basic interface for all servlets
`jakarta.servlet.GenericServlet`	Abstract class implementing `Servlet`
`jakarta.servlet.http.HttpServlet`	Specialization for handling HTTP requests
`jakarta.servlet.http.HttpServletRequest`	Represents client request
`jakarta.servlet.http.HttpServletResponse`	Represents server response

Servlet Interface

Servlet interface provides common behaviour to all the servlets. Servlet interface defines methods that all servlets must implement.

Servlet interface needs to be implemented for creating any servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods.

Methods of Servlet interface

There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

Method	Description
public void init(ServletConfig config)	initializes the servlet. It is the life cycle method of servlet and invoked by the web container only once.
public void service(ServletRequest request, ServletResponse response)	provides response for the incoming request. It is invoked at each request by the web container.
public void destroy()	is invoked only once and indicates that servlet is being destroyed.
public ServletConfig getServletConfig()	returns the object of ServletConfig.
public String getServletInfo()	returns information about servlet such as writer, copyright, version etc.

GenericServlet class:

GenericServlet class implements Servlet, ServletConfig and Serializable interfaces. It provides the implementation of all the methods of these interfaces except the service method.

GenericServlet class can handle any type of request so it is protocol-independent.
You may create a generic servlet by inheriting the GenericServlet class and providing the implementation of the service method.

Methods of GenericServlet class

There are many methods in GenericServlet class. They are as follows:

1. **public void init(ServletConfig config)** is used to initialize the servlet.
2. **public abstract void service(ServletRequest request, ServletResponse response)** provides service for the incoming request. It is invoked at each time when user requests for a servlet.
3. **public void destroy()** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.
4. **public ServletConfig getServletConfig()** returns the object of ServletConfig.
5. **public String getServletInfo()** returns information about servlet such as writer, copyright, version etc.
6. **public void init()** it is a convenient method for the servlet programmers, now there is no need to call super.init(config)
7. **public ServletContext getServletContext()** returns the object of ServletContext.

8. **public String getInitParameter(String name)** returns the parameter value for the given parameter name.
9. **public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.
10. **public String getServletName()** returns the name of the servlet object.
11. **public void log(String msg)** writes the given message in the servlet log file.
12. **public void log(String msg,Throwable t)** writes the explanatory message in the servlet log file and a stack trace.

HttpServlet class

The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

Methods of HttpServlet class

There are many methods in HttpServlet class. They are as follows:

1. **public void service(ServletRequest req,ServletResponse res)** dispatches the request to the protected service method by converting the request and response object into http type.
2. **protected void service(HttpServletRequest req, HttpServletResponse res)** receives the request from the service method, and dispatches the request to the doXXX() method depending on the incoming http request type.
3. **protected void doGet(HttpServletRequest req, HttpServletResponse res)** handles the GET request. It is invoked by the web container.
4. **protected void doPost(HttpServletRequest req, HttpServletResponse res)** handles the POST request. It is invoked by the web container.
5. **protected void doHead(HttpServletRequest req, HttpServletResponse res)** handles the HEAD request. It is invoked by the web container.
6. **protected void doOptions(HttpServletRequest req, HttpServletResponse res)** handles the OPTIONS request. It is invoked by the web container.
7. **protected void doPut(HttpServletRequest req, HttpServletResponse res)** handles the PUT request. It is invoked by the web container.
8. **protected void doTrace(HttpServletRequest req, HttpServletResponse res)** handles the TRACE request. It is invoked by the web container.
9. **protected void doDelete(HttpServletRequest req, HttpServletResponse res)** handles the DELETE request. It is invoked by the web container.
10. **protected long getLastModified(HttpServletRequest req)** returns the time when HttpServletRequest was last modified since midnight January 1, 1970 GMT.

Steps to create a servlet example

The servlet can be created by three ways:

1. By implementing Servlet interface,
2. By inheriting GenericServlet class, (or)
3. By inheriting HttpServlet class

The mostly used approach is by extending HttpServlet because it provides http request specific method such as doGet(), doPost(), doHead() etc.

1. Under (**Src/main/java** available under **Java Resources**) create a class File like below.

*Right-click -> New -> Class

package in.sp;

```
import java.io.IOException;
import java.io.PrintWriter;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class ServDemo1 extends HttpServlet
{
protected void doGet(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

{

//Printing in web Browser
                res.setContentType("text/html");
                PrintWriter pw=res.getWriter();
                pw.print("<h1>I am in DoGet for browser<h1>");

//Printing in Console
                System.out.println("Too Good.....");
        }

}
```

Create the deployment descriptor (web.xml file)

The deployment descriptor is an xml file, from which Web Container gets the information about the servlet to be invoked.
The web container uses the Parser to get the information from the web.xml file. There are many xml parsers such as SAX, DOM and Pull.
There are many elements in the web.xml file. Here is given some necessary elements to run the simple servlet program.

Web.xml file

It is known as deployment descriptor. This file is used to Maps the servlets to specific URLs.

```
<servlet>

 <servlet-name>myservlet</servlet-name>
 <servlet-class>in.sp.ServDemo1</servlet-class>

</servlet>

<servlet-mapping>

 <servlet-name>myservlet</servlet-name>
 <url-pattern>/test</url-pattern>

</servlet-mapping>
```

Description of the elements of web.xml file

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

How Servlet works?

It is important to learn how servlet works for understanding the servlet well. Here, we are going to get the internal detail about the first servlet program.

The server checks if the servlet is requested for the first time.

If yes, web container does the following tasks:

- loads the servlet class.
- instantiates the servlet class.
- calls the init method passing the ServletConfig object

else

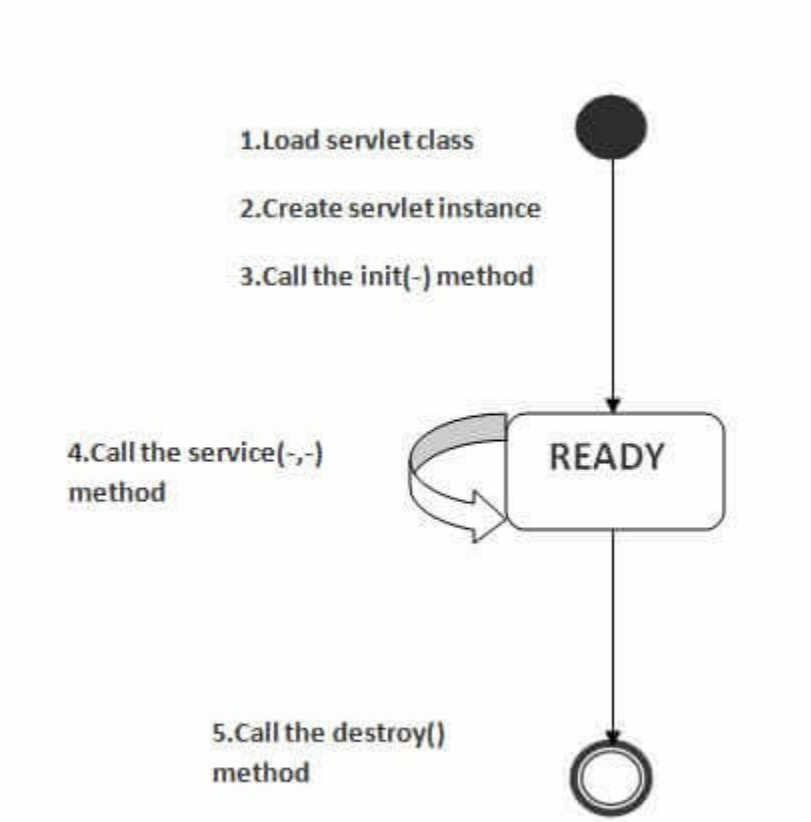
- calls the service method passing request and response objects

The web container calls the destroy method when it needs to remove the servlet such as at time of stopping server or undeploying the project.

Life Cycle of a Servlet (Servlet Life Cycle)

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded
The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

2) Servlet instance is created
The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

public void init(ServletConfig config) throws ServletException

4) service method is invoked
The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException

5) destroy method is invoked
The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

public void destroy()

When is Each Method Called?

Method	When is it Called?	How Many Times?
`init()`	When the servlet is first loaded	Once
`service()`	Every time a request is received	Multiple times
`doGet()` / `doPost()`	Based on the request type	Multiple times
`destroy()`	Before the servlet is removed from memory	Once

2. Complete Example: Servlet Life Cycle

```
public class LifeCycleServlet extends HttpServlet {

   // Called only once when the servlet is created
   public void init() throws ServletException
{
      System.out.println("Servlet Initialized...");
   }

   // Called every time a request is received
```

```
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException
{
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<h1>Servlet Life Cycle Example</h1>");
      System.out.println("Processing GET request...");
   }

   // Called only once when the servlet is destroyed
   public void destroy()
 {
      System.out.println("Servlet Destroyed...");
   }
}
```

HTTP Requests
The request sent by the computer to a web server, contains all sorts of potentially interesting information; it is known as HTTP requests.

The HTTP request method indicates the method to be performed on the resource identified by the Requested URI (Uniform Resource Identifier). This method is case-sensitive and should be used in uppercase.
The HTTP request methods are:

HTTP Request	Description
GET	Asks to get the resource at the requested URL.
POST	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
HEAD	Asks for only the header part of whatever a GET would return. Just like GET but with no body.
TRACE	Asks for the loopback of the request message, for testing or troubleshooting.
PUT	Says to put the enclosed info (the body) at the requested URL.
DELETE	Says to delete the resource at the requested URL.
OPTIONS	Asks for a list of the HTTP methods to which the thing at the request URL can respond

Get vs. Post
There are many differences between the Get and Post request. Let's see these differences:

GET	POST
1) In case of Get request, only limited amount of data can be sent because data is sent in header.	In case of post request, large amount of data can be sent because data is sent in body.
2) Get request is not secured because data is exposed in URL bar.	Post request is secured because data is not exposed in URL bar.
3) Get request can be bookmarked.	Post request cannot be bookmarked.
4) Get request is idempotent . It means second request will be ignored until response of first request is delivered	Post request is non-idempotent.
5) Get request is more efficient and used more than Post.	Post request is less efficient and used less than get.

GET and POST
Two common methods for the request-response between a server and client are:
- GET- It requests the data from a specified resource
- POST- It submits the processed data to a specified resource

Some other features of GET requests are:
- It remains in the browser history
- It can be bookmarked
- It can be cached
- It have length restrictions

- It should never be used when dealing with sensitive data
- It should only be used for retrieving the data

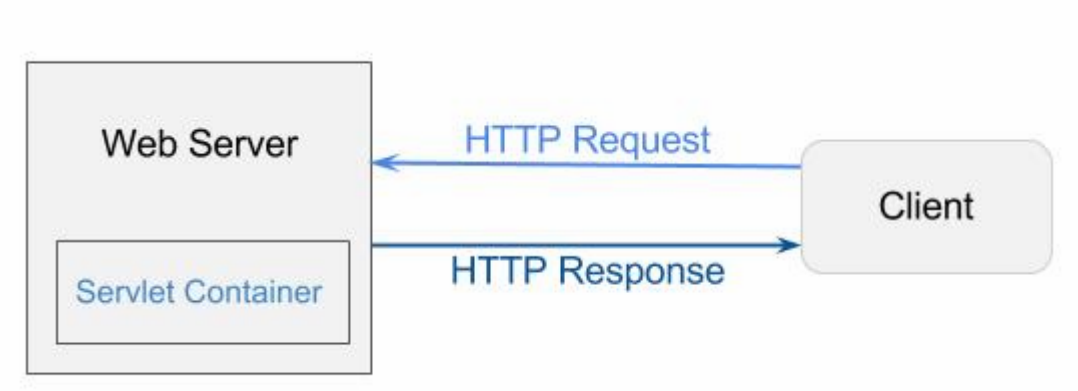
Some other features of POST requests are:

- This requests cannot be bookmarked
- This requests have no restrictions on length of data
- This requests are never cached
- This requests do not retain in the browser history

Servlet Container

It provides the runtime environment for JavaEE (j2ee) applications. The client/user can request only a static WebPages from the server. If the user wants to read the web pages as per input then the servlet container is used in java.

The servlet container is the part of web server which can be run in a separate process. We can classify the servlet container states in three types:



Servlet Container States

The servlet container is the part of web server which can be run in a separate process. We can classify the servlet container states in three types:

Standalone: It is typical Java-based servers in which the servlet container and the web servers are the integral part of a single program. For example:- Tomcat running by itself

In-process: It is separated from the web server, because a different program runs within the address space of the main server as a plug-in. For example:- Tomcat running inside the JBoss.

Out-of-process: The web server and servlet container are different programs which are run in a different process. For performing the communications between them, web server uses the plug-in provided by the servlet container.

The Servlet Container performs many operations that are given below:

- Life Cycle Management
- Multithreaded support
- Object Pooling
- Security etc.

Server: Web vs. Application

Server is a device or a computer program that accepts and responds to the request made by other program, known as client. It is used to manage the network resources and for running the program or software that provides services.
There are two types of servers:

1. Web Server
2. Application Server

Web Server

Web server contains only web or servlet container. It can be used for servlet, jsp, struts, jsf etc. It can't be used for EJB.
It is a computer where the web content can be stored. In general web server can be used to host the web sites but there also used some other web servers also such as FTP, email, storage, gaming etc.
Examples of Web Servers are: **Apache Tomcat** and **Resin**.

Application Server

Application server contains Web and EJB containers. It can be used for servlet, jsp, struts, jsf, ejb etc. It is a component-based product that lies in the middle-tier of a server centric architecture.
It provides the middleware services for state maintenance and security, along with persistence and data access. It is a type of server designed to install, operate and host associated services and applications for the IT services, end users and organizations.

The Example of Application Servers are:

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

Working with Initialization Parameters in Servlets

Introduction

Servlets often need configuration parameters to function dynamically. Instead of hardcoding values in the Java code, we use **initialization parameters** that can be defined in the deployment descriptor (web.xml) or through annotations.

Types of Initialization Parameters

a) Servlet Initialization Parameters

- Specific to a **single servlet**.
- Defined inside <servlet> in web.xml or using @WebInitParam annotation.
- Useful for passing servlet-specific configurations.

b) Context Initialization Parameters

- Shared across **all servlets** in an application.
- Defined inside <context-param> in web.xml.
- Used for application-wide settings, like database connection details.

Servlet Initialization Parameters: Implementation

Using web.xml

1. **Define parameters in web.xml**:

```
<servlet>
  <servlet-name>MyServlet</servlet-name>
  <servlet-class>com.example.MyServlet</servlet-class>
  <init-param>
    <param-name>username</param-name>
    <param-value>admin</param-value>
  </init-param>
  <init-param>
    <param-name>password</param-name>
    <param-value>12345</param-value>
  </init-param>
</servlet>
```

Retrieve parameters in the servlet:

```
import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyServlet extends HttpServlet {
  private String username;
  private String password;

  @Override
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    username = config.getInitParameter("username");
    password = config.getInitParameter("password");
  }

  @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.getWriter().println("Username: " + username + ", Password: " + password);
  }
}
```

Using Annotations (@WebInitParam)

Define parameters in the servlet class:

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.http.HttpServlet;

@WebServlet(
  name = "MyServlet",
  urlPatterns = {"/myservlet"},
  initParams = {
    @WebInitParam(name = "username", value = "admin"),
    @WebInitParam(name = "password", value = "12345")
  }
```

```
)
public class MyServlet extends HttpServlet {
    // Fetch parameters inside init() method
}
```

Context Initialization Parameters

Using web.xml

Define context parameters:

```
<context-param>
    <param-name>dbURL</param-name>
    <param-value>jdbc:mysql://localhost:3306/mydb</param-value>
</context-param>
```

Retrieve in any servlet:

```
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

public class MyServlet extends HttpServlet {
    private String dbURL;

    @Override
    public void init() throws ServletException {
        ServletContext context = getServletContext();
        dbURL = context.getInitParameter("dbURL");
    }
}
```

Key Differences: Servlet vs. Context Parameters

Feature	Servlet Initialization Parameter	Context Initialization Parameter
Scope	Specific to a servlet	Available to all servlets
Defined in	<init-param> inside <servlet>	<context-param> outside <servlet>
Retrieved using	ServletConfig.getInitParameter()	ServletContext.getInitParameter()

Use Cases

- **Servlet Initialization Parameters**: Used for servlet-specific settings like authentication details or file paths.
- **Context Initialization Parameters**: Used for application-wide settings like database configurations, logging paths, etc.

Conclusion

- Initialization parameters enhance flexibility and maintainability.
- web.xml approach allows separation of configuration from code.
- @WebInitParam annotation is a modern alternative.

Servlet Filters in Java

A **Servlet Filter** is an object that intercepts requests and responses in a Java web application before they reach the servlet (or after the servlet processes them). Filters allow you to modify request and response objects, perform logging, authentication, compression, or other pre/post-processing tasks.

Key Characteristics of Servlet Filters

- Filters do **not** generate responses directly; they modify requests before reaching the servlet or responses before being sent to the client.
- They can be **chained**, meaning multiple filters can be applied sequentially.
- Implemented using the javax.servlet.Filter interface.

How Servlet Filters Work

1. A client sends a request.
2. The request first passes through the filter(s).
3. The filter(s) process the request (e.g., validation, logging).
4. The request is then forwarded to the servlet.
5. The servlet processes the request and generates a response.
6. The response passes through the filter(s) again (if applicable).
7. The response is sent back to the client.

Uses of Servlet Filters

1. **Logging and Monitoring**
   - Track request and response times.
   - Log user activities for debugging and analytics.
2. **Authentication and Authorization**
   - Check if a user is logged in before allowing access to a resource.
   - Validate JWT tokens or session authentication.
3. **Data Compression**
   - Compress responses (e.g., GZIP) before sending them to the client.
4. **Request Modification**
   - Add headers or parameters to incoming requests.
   - Convert request payloads (e.g., JSON to XML).
5. **Response Modification**
   - Modify the response before sending it to the client (e.g., adding security headers).
6. **CORS (Cross-Origin Resource Sharing) Handling**
   - Add necessary headers to allow cross-origin requests.

Example of a Servlet Filter

Here's a simple logging filter that logs the request details:

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

@WebFilter("/*") // Applies to all requests
public class LoggingFilter implements Filter {

    public void init(FilterConfig fConfig) throws ServletException {
        System.out.println("Logging Filter Initialized");
    }
```

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

        HttpServletRequest req = (HttpServletRequest) request;
                    System.out.println("Incoming request: " + req.getMethod() + " " +
req.getRequestURI());

        // Continue request processing
        chain.doFilter(request, response);

        System.out.println("Response sent for: " + req.getRequestURI());
    }

    public void destroy() {
        System.out.println("Logging Filter Destroyed");
    }
}
```

Types of Filters

1. **Request Filters** – Modify request before reaching the servlet.
2. **Response Filters** – Modify response before sending to the client.
3. **Chained Filters** – Multiple filters applied in sequence.

Key Advantages of Using Filters

✓ **Reusability** – Filters are independent of servlets and can be applied to multiple servlets.

✓ **Modularity** – Centralized processing logic (e.g., authentication) instead of writing it in every servlet.

✓ **Performance Optimization** – Filters can compress responses or cache content.

Examples to demonstrate Servlet Filters Usage

Servlet filters are commonly used for **logging, authentication, compression, and modifying requests or responses**. Below are different examples of filters applied in real-world servlet applications.

1. Logging Filter

Use Case: Logs request details such as the URL, method, and execution time.

Code Example

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

@WebFilter("/*") // Apply to all requests
public class LoggingFilter implements Filter {

    public void init(FilterConfig fConfig) throws ServletException {
        System.out.println("Logging Filter Initialized");
```

```
    }

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

        HttpServletRequest req = (HttpServletRequest) request;
        long startTime = System.currentTimeMillis();

        System.out.println("Request: " + req.getMethod() + " " + req.getRequestURI());

        chain.doFilter(request, response); // Continue request processing

        long duration = System.currentTimeMillis() - startTime;
         System.out.println("Response sent for " + req.getRequestURI() + " in " + duration + "
ms");
    }

    public void destroy() {
        System.out.println("Logging Filter Destroyed");
    }
}
```

2. Authentication Filter

Use Case: Restricts access to certain resources based on user authentication.

Code Example

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebFilter("/secure/*") // Apply to all URLs under /secure/
public class AuthenticationFilter implements Filter {

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

        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;
        HttpSession session = req.getSession(false);

        if (session == null || session.getAttribute("user") == null) {
            res.sendRedirect(req.getContextPath() + "/login.jsp"); // Redirect to login page
        } else {
            chain.doFilter(request, response); // User authenticated, continue
        }
    }
}
```

3. Response Compression Filter (GZIP)
Use Case: Compresses large responses to improve performance.
Code Example

```
import java.io.IOException;
import java.util.zip.GZIPOutputStream;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServletResponseWrapper;

@WebFilter("/*") // Apply to all requests
public class GzipCompressionFilter implements Filter {

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

        HttpServletResponse httpResponse = (HttpServletResponse) response;
        if (request.getContentType() != null && request.getContentType().contains("text")) {
                                            GzipResponseWrapper wrappedResponse = new
GzipResponseWrapper(httpResponse);
            chain.doFilter(request, wrappedResponse);
            wrappedResponse.finishResponse();
        } else {
            chain.doFilter(request, response);
        }
    }
}

// Wrapper class for GZIP response
class GzipResponseWrapper extends HttpServletResponseWrapper {
    private GZIPOutputStream gzipOutputStream;
    private ServletOutputStream outputStream;

    public GzipResponseWrapper(HttpServletResponse response) throws IOException {
        super(response);
        gzipOutputStream = new GZIPOutputStream(response.getOutputStream());
    }

    public ServletOutputStream getOutputStream() {
        return outputStream;
    }

    public void finishResponse() throws IOException {
        gzipOutputStream.close();
    }
}
```

4. Cross-Origin Resource Sharing (CORS) Filter
Use Case: Allows cross-origin requests for APIs.
Code Example

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletResponse;

@WebFilter("/*") // Apply to all requests
public class CORSFilter implements Filter {

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

        HttpServletResponse httpResponse = (HttpServletResponse) response;
        httpResponse.setHeader("Access-Control-Allow-Origin", "*");
                httpResponse.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT,
DELETE, OPTIONS");
                        httpResponse.setHeader("Access-Control-Allow-Headers", "Content-Type,
Authorization");

        chain.doFilter(request, response);
    }
}
```

5. Request Parameter Validation Filter
Use Case: Validates incoming parameters before they reach the servlet.
Code Example

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

@WebFilter("/submitForm") // Apply only to specific endpoint
public class InputValidationFilter implements Filter {

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

        HttpServletRequest req = (HttpServletRequest) request;
        String username = req.getParameter("username");

        if (username == null || username.trim().isEmpty()) {
            request.setAttribute("errorMessage", "Username cannot be empty");
            request.getRequestDispatcher("/error.jsp").forward(request, response);
        } else {
            chain.doFilter(request, response);
        }
    }
}
```

6. Security Header Filter
Use Case: Adds security-related headers to HTTP responses.

Code Example

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletResponse;

@WebFilter("/*") // Apply to all requests
public class SecurityHeaderFilter implements Filter {

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

        HttpServletResponse httpResponse = (HttpServletResponse) response;
        httpResponse.setHeader("X-Content-Type-Options", "nosniff");
        httpResponse.setHeader("X-XSS-Protection", "1; mode=block");
        httpResponse.setHeader("X-Frame-Options", "DENY");

        chain.doFilter(request, response);
    }
}
```

Key Takeaways

- **Filters are powerful middleware in Java web applications** used for **logging, authentication, compression, security, and request validation**.
- **They are reusable and modular**—separating concerns instead of writing repetitive code in servlets.
- **Multiple filters can be chained** together to apply sequential transformations.

JavaServer Pages (JSP)

Introduction to JSP

JavaServer Pages (JSP) is a server-side technology that enables the creation of dynamic web content using Java. It simplifies the process of building web applications by allowing developers to embed Java code directly into HTML pages.

Features of JSP:

- Platform-independent
- Supports reusable components (JavaBeans, Custom Tags)
- Built-in support for session management
- Integrates easily with databases

Understanding JSP

JSP files are text-based documents that contain HTML, Java code, and special JSP tags. When a JSP page is requested, the server translates it into a Java Servlet and executes it to generate dynamic content.

Benefits of JSP:

- Separation of business logic from presentation

- Automatic conversion into servlets
- Simplifies web development

Describing the JSP Life Cycle

The JSP life cycle consists of the following stages:

1. **Translation**: JSP is translated into a servlet.
2. **Compilation**: The servlet is compiled into a bytecode (.class file).
3. **Initialization**: The jspInit() method is called.
4. **Execution**: The _jspService() method processes client requests.
5. **Destruction**: The jspDestroy() method is called when the JSP is removed from memory.

Addition Reference Link → https://www.tutorialspoint.com/jsp/jsp_syntax.htm

Creating a Simple JSP Page

A simple JSP page looks like this:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<html>
<head>
  <title>My First JSP Page</title>
</head>
<body>
  <h1>Hello, JSP!</h1>
</body>
</html>
```

Working with JSP Basic Tags and Implicit Objects

JSP provides built-in objects, also known as **implicit objects**, that can be used within JSP pages:

- request: Represents the HTTP request.
- response: Represents the HTTP response.
- session: Manages user sessions.
- application: Stores data shared across the application.
- out: Prints output to the browser.
- pageContext: Provides access to all scopes.
- config: Provides servlet configuration details.
- page: Refers to the JSP page itself.

Example using implicit objects:

```
<%
  String name = request.getParameter("name");
  out.println("Hello, " + name);
%>
```

Working with JavaBeans and Action Tags in JSP

JavaBeans

JavaBeans are reusable components that can be used to store and retrieve data in JSP.

JavaBean Example:

```
public class UserBean {
  private String name;
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
}
```

Using JavaBean in JSP:

```
<jsp:useBean id="user" class="UserBean" scope="session" />
<jsp:setProperty name="user" property="name" value="John Doe" />
```

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

Simple JSP Program to Fetch Database Records

To connect JSP with a database, follow these steps:

1. Load the database driver.
2. Establish a connection.
3. Execute the SQL query.
4. Process and display the result.

Example JSP page to fetch records:

```
<%@ page import="java.sql.*" %>
<html>
<body>
<table border="1">
<tr><th>ID</th><th>Name</th></tr>
<%
   try {
      Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
"root", "password");
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT id, name FROM users");
      while (rs.next()) {
%>
<tr>
   <td><%= rs.getInt("id") %></td>
   <td><%= rs.getString("name") %></td>
</tr>
<%
      }
      con.close();
   } catch (Exception e) {
      out.println("Error: " + e.getMessage());
   }
%>
</table>
</body>
</html>
```

Exercises

1. Create a simple JSP page that takes user input using a form and displays it on the next page.
2. Implement a JavaBean to store user details and retrieve them in a JSP page.
3. Modify the database JSP example to include additional fields like email and phone number.
4. Experiment with different implicit objects like session, application, and config.

These notes cover JSP basics and provide practical exercises to reinforce learning.

Solutions for Exercises

1. **Create a simple JSP page that takes user input using a form and displays it on the next page.**

index.jsp:

```
<html>
<body>
   <form action="display.jsp" method="post">
      Name: <input type="text" name="name">
      <input type="submit" value="Submit">
   </form>
</body>
</html>
```

display.jsp:

```
<%
   String name = request.getParameter("name");
%>
<html>
<body>
   <h1>Hello, <%= name %>!</h1>
</body>
</html>
```

2. **Implement a JavaBean to store user details and retrieve them in a JSP page.**

UserBean.java:

```
public class UserBean {
   private String name;
   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
}
```

JSP Page:

```
<jsp:useBean id="user" class="UserBean" scope="session" />
<jsp:setProperty name="user" property="name" value="John Doe" />
Hello, <jsp:getProperty name="user" property="name" />
```

3. **Modify the database JSP example to include additional fields like email and phone number.**

```
<%@ page import="java.sql.*" %>
<html>
<body>
<table border="1">
<tr><th>ID</th><th>Name</th><th>Email</th><th>Phone</th></tr>
<%
   try {
      Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
"root", "password");
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT id, name, email, phone FROM users");
      while (rs.next()) {
%>
<tr>
   <td><%= rs.getInt("id") %></td>
   <td><%= rs.getString("name") %></td>
   <td><%= rs.getString("email") %></td>
   <td><%= rs.getString("phone") %></td>
</tr>
```

```
<%
        }
        con.close();
    } catch (Exception e) {
        out.println("Error: " + e.getMessage());
    }
%>
</table>
</body>
</html>
```

4. **Experiment with different implicit objects like session, application, and config.**

```
<%
    session.setAttribute("user", "John Doe");
    application.setAttribute("appName", "JSP Demo");
%>
Hello, <%= session.getAttribute("user") %>.
Welcome to <%= application.getAttribute("appName") %>!
```