

## Spring project using MVC,Servlets and JSP

The Java programming language is divided into **four main platforms**, each designed for different types of development environments:

There are four platforms of the Java programming language: **Java Platform, Standard Edition (Java SE)** **Java Platform, Enterprise Edition (Java EE)**

### 1. Java SE (Standard Edition)

- **Overview:** Java SE provides the core functionality for Java applications, including the Java Development Kit (JDK) and Java Runtime Environment (JRE). It includes libraries, APIs, and the JVM necessary to develop and run Java applications on desktops, servers, and embedded devices. Java SE (Standard Edition) is the foundational Java platform that provides essential libraries and APIs for general-purpose programming. It forms the basis for other Java platforms, including Java EE and Java ME. Java SE includes database access, networking, GUI development, and security libraries.
- **Key Components:**
  - **JDK** (Java Development Kit): Includes tools for developing Java applications, such as the Java compiler, debuggers, and the JRE.
  - **JRE** (Java Runtime Environment): Provides the libraries, Java Virtual Machine (JVM), and other components needed to run Java applications.
  - **APIs:** Libraries that offer functionality such as file handling, networking, and GUI development (Swing, JavaFX).
  - **Core Libraries:** Provides fundamental libraries such as java.lang, java.util, java.io, and java.nio.
  - **Swing and AWT:** Offers APIs for creating GUI-based desktop applications.
  - **Networking:** Includes APIs for building network-based applications.
  - **Concurrency:** Supports multithreading and concurrent programming through java.util.concurrent.
  - **JDBC:** Java Database Connectivity (JDBC) for database interactions.
  -
- **Use Cases:** Desktop applications, utilities, enterprise-level applications, and small to medium web applications. E-commerce websites ,Banking applications

### Example: Java SE Application

Here's a simple example of a console-based Java SE application that reads user input and prints a message:

```
import java.util.Scanner;

public class HelloWorld {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your name: ");

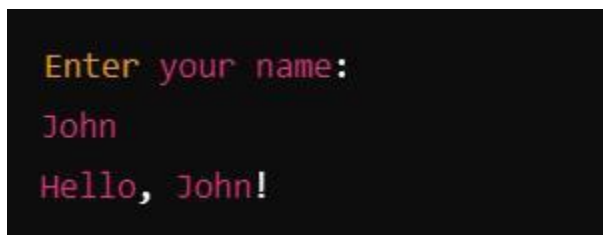
        String name = scanner.nextLine();

        System.out.println("Hello, " + name + "!");

    }

}
```

### Output:



```
Enter your name:
John
Hello, John!
```

### Explanation:

- **Import Statement:** `import java.util.Scanner;` imports the `Scanner` class for reading input.
- **Class Definition:** `public class HelloWorld` defines a public class named `HelloWorld`.
- **Main Method:** `public static void main(String[] args)` is the entry point of the application.
- **Scanner Object:** `Scanner scanner = new Scanner(System.in);` creates a `Scanner` object to read user input from the console.

- **Reading Input:** `String name = scanner.nextLine();` reads a line of text input by the user.
- **Output:** `System.out.println("Hello, " + name + "!");` prints a personalized greeting to the console.

## 2. Java EE (Enterprise Edition) - Now known as Jakarta EE

**J2EE** (1999–2006) → **JEE** (2006–2017) → **Jakarta EE** (2017–present).

- **Overview:** Java EE (now Jakarta EE after it was transferred to the Eclipse Foundation) extends Java SE with additional libraries and tools that make it suitable for developing large-scale enterprise applications. It provides support for web services, component-based architecture, distributed computing, and more.
- **Banking Systems:** Banks use Jakarta EE to manage large-scale transactional systems, ensuring the reliability and security of sensitive customer data. Jakarta EE's support for Enterprise JavaBeans (EJB) and JPA is ideal for handling complex business logic and data persistence in banking applications.
- **E-Commerce Platforms:** Large e-commerce sites like Amazon and eBay need scalable systems to handle thousands of transactions per second. Jakarta EE's JMS and RESTful web services (JAX-RS) are well-suited for managing order processing, inventory management, and payment processing.
- **Key Components:**
  - **Servlets and JSP:** For web-based applications.
  - **EJB (Enterprise JavaBeans):** For business logic.
  - **JPA (Java Persistence API):** For database access.
  - **JMS (Java Message Service):** For messaging.
  - **JTA (Java Transaction API):** For managing transactions.
- **Use Cases:** Enterprise applications, large-scale web applications, distributed systems

Users of Tomcat 10 onwards should be aware that, as a result of the move from Java EE to Jakarta EE as part of the transfer of Java EE to the Eclipse Foundation, the primary package for all implemented APIs has changed from `javax.*` to `jakarta.*`. This will almost certainly require code changes to enable applications to migrate from Tomcat 9 and earlier to Tomcat 10 and later. A [migration tool](#) has been developed to aid this process.

### Example: Java EE Application

Here's a simple example of a servlet in Java EE that handles HTTP requests and sends an HTML response:

```
import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/hello")

public class HelloServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

        response.setContentType("text/html");

        response.getWriter().write("<h1>Hello, Welcome to Java EE!</h1>");

    }

}
```

URL:

`http://localhost:8080/hello`

Output (in browser):

```
<h1>Hello, welcome to Java EE!</h1>
```

Explanation:

- **Import Statements:** `import java.io.IOException;` and `import javax.servlet.ServletException;` import required classes for handling exceptions.

import javax.servlet.annotation.WebServlet;, import javax.servlet.http.HttpServlet;, and related imports are for servlet functionality.

- **Servlet Annotation:** @WebServlet("/hello") maps the servlet to the URL pattern /hello.
- **Class Definition:** public class HelloServlet extends HttpServlet defines a servlet class extending HttpServlet.
- **doGet Method:** protected void doGet(HttpServletRequest request, HttpServletResponse response) processes GET requests. It sets the content type to text/html and writes a simple HTML response.

### 3 .Java ME (Micro Edition)

- **Overview:** Java ME is a subset of Java designed for resource-constrained devices such as embedded systems, mobile phones (prior to smartphones), IoT devices, and other small devices with limited processing power, memory, and storage.
- **Key Components:**
  - **CLDC (Connected Limited Device Configuration):** A set of libraries and virtual machine profiles for devices with limited resources.
  - **MIDP (Mobile Information Device Profile):** A set of APIs that allow developers to create mobile applications, such as games, utilities, and services for early mobile devices.
  - **CDC (Connected Device Configuration):** Used for more powerful embedded devices with larger resources.
- **Use Cases:** Mobile phones, IoT devices, smartwatches, and embedded systems.

### What is a web application?

A web application is an **application accessible from the web**. A web application is composed of **web components** like **Servlet, JSP, Filter, etc.** and other elements such as HTML, CSS, and JavaScript. The web components **typically execute in Web Server** and **respond to the HTTP request**.

### What is a Servlet in Java

Java Servlet technology is **used to create a web application**.

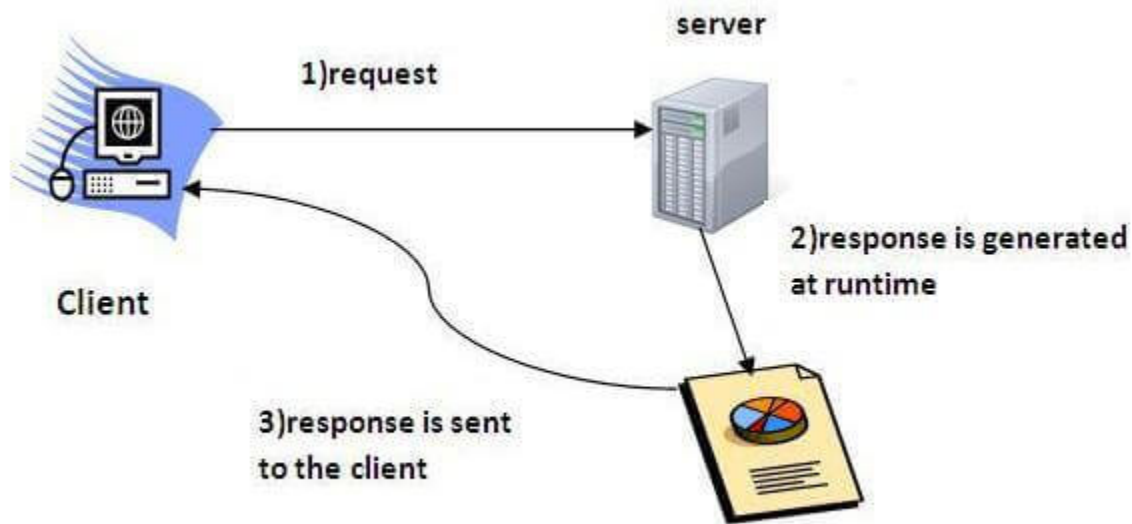
A **Servlet** is a **Java class** that **handles HTTP requests** and **generates HTTP responses**.

It runs inside a **Servlet container** (like web servers **Tomcat**, **Jetty**, etc.).

Servlet is a web component that is deployed on the server **to create a dynamic web page**.

or

**Java program that runs on a web server and builds web pages dynamically.**



### Why do we use Servlets?

- To create **dynamic web applications**.
- To **process user input** from HTML forms.
- To **connect with databases, perform operations, and return results** dynamically.
- To **generate content** like HTML, JSON, XML, etc.

### Servlet Life Cycle

Servlets go through these stages:

1. **Loading and Instantiation** (Container loads Servlet class)

## 2. Initialization (**init()**) (Only once) :

The `init()` method initializes a servlet and is called **once** during its lifecycle.

## 3. Request Handling (**service()**) (For each request) :

The `service()` method is called to **process client requests** and dispatches them to the **appropriate methods like `doGet()` or `doPost()`**.

## 4. Destroy (**destroy()**) (When shutting down):

The `destroy()` method is called **before a servlet is destroyed** to perform any **necessary cleanup operations**.

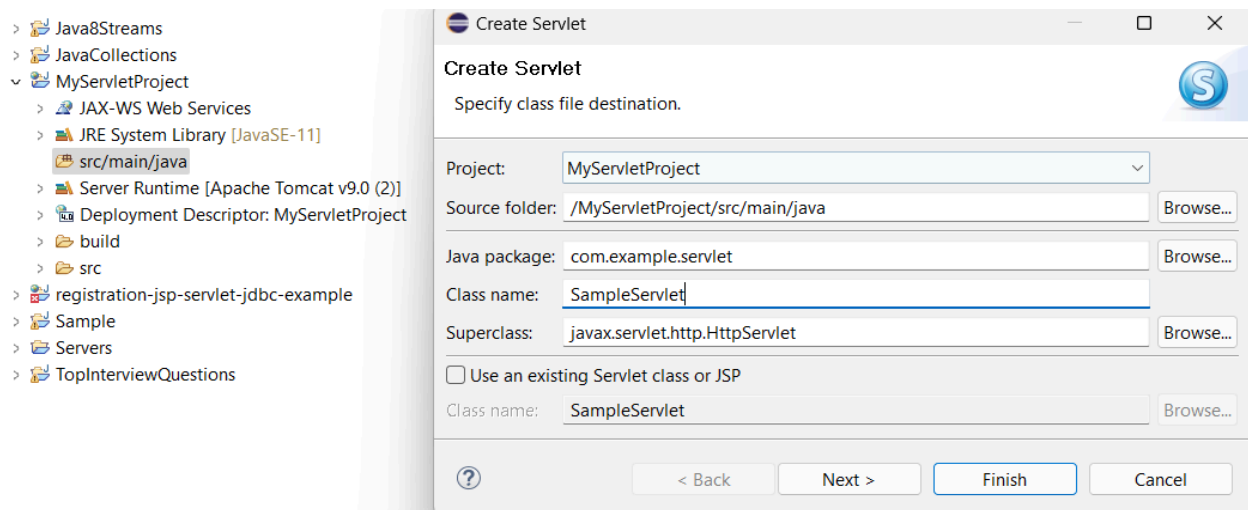
To create a servlet, a class must implement the Servlet interface.

Ex:

setContentType Line	Content Type	Meaning	Sample Output
<code>resp.setContentType("text/html");</code>	<code>text/html</code>	HTML content (web page)	<code>&lt;h1&gt;Welcome! &lt;/h1&gt;</code> (renders heading)
<code>resp.setContentType("text/plain");</code>	<code>text/plain</code>	Plain text (no formatting)	<code>Just some plain text.</code>
<code>resp.setContentType("application/json");</code>	<code>application/json</code>	JSON data for API responses	<code>{ "message": "Hello, JSON" }</code>
<code>resp.setContentType("application/xml");</code>	<code>application/xml</code>	XML data	<code>&lt;note&gt; &lt;to&gt;User&lt;/to&gt; &lt;/note&gt;</code>
<code>resp.setContentType("text/css");</code>	<code>text/css</code>	CSS stylesheet	<code>body { background-color: yellow; }</code>
<code>resp.setContentType("application/javascript");</code>	<code>application/javascript</code>	JavaScript file	<code>alert('Hello!');</code>
<code>resp.setContentType("image/png");</code>	<code>image/png</code>	PNG image	(Binary image data)
<code>resp.setContentType("image/jpeg");</code>	<code>image/jpeg</code>	JPEG image	(Binary image data)
<code>resp.setContentType("application/pdf");</code>	<code>application/pdf</code>	PDF document	(PDF file will open/download)

## Step 1: Create a Dynamic Web Project

1. Open **Eclipse**.
2. Go to **File → New → Dynamic Web Project**.
3. Enter the **Project Name** (e.g., **MyServletProject**).
4. Select **Target Runtime** (Apache Tomcat), and if not set, click **New Runtime** and configure it.
5. Choose **Dynamic Web Module Version** (3.1, 4.0, or 5.0) based on your Eclipse and Tomcat version.
  - **Target Runtime:** Apache Tomcat 9.0
  - **Dynamic Web Module Version:** 4.0
  - **Java Version / Compiler Compliance Level:** 17
6. Click **Finish**.



## Step 2: Create a Servlet Class

1. Right-click on the **src** folder → **New → Servlet**.  
Name your Servlet (e.g., **SampleServlet**).
2. Set package (e.g., **com.example.servlet**).
3. Eclipse will **auto-generate** a basic servlet for us and we can edit according to our need.

```
package com.example.servlet;  
import java.io.IOException;  
import java.io.PrintWriter;
```



```

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class SampleServlet
 */
//Servlet Annotation (mapping URL to Servlet)
@WebServlet("/SampleServlet")
public class SampleServlet extends HttpServlet {
    // 1. init() - called once when servlet is created
    @Override
    public void init() throws ServletException {
        System.out.println("Servlet is being initialized (init method called)");
    }
    // 2. service() - called for every request (GET, POST etc.)
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        System.out.println("Handling GET request (service/doGet method called)");
        // Set content type
        response.setContentType("text/html");
        // Get output stream
        PrintWriter out = response.getWriter();
        // Write response
        out.println("<html><body>");
        out.println("<h2>Hello from SampleServlet</h2>");
        out.println("</body></html>");
    }
    // 3. destroy() - called once when servlet is being destroyed
    @Override
    public void destroy() {
        System.out.println("Servlet is being destroyed (destroy method called)");
    }
}

```

Output

**Console Output (Eclipse console when you start the server and interact with servlet):**

When the **server starts** and the servlet is first loaded:

Servlet is being initialized (init method called)

When you **hit the URL** (like `http://localhost:8080/MyServletProject/SampleServlet`) in your browser:

Handling GET request (service/doGet method called)

When the **server stops** (or servlet gets unloaded):

Servlet is being destroyed (destroy method called)

### 🌟 Browser Output (what you see on the web page):

When you open `http://localhost:8080/MyServletProject/SampleServlet`, you will see a simple HTML page:

```
<html><body>
<h2>Hello from SampleServlet</h2>
</body></html>
```

Visually on the browser, it will show:

**Hello from SampleServlet**

(The text will appear in a slightly bigger size because of `<h2>` tag.)

### What will happen:

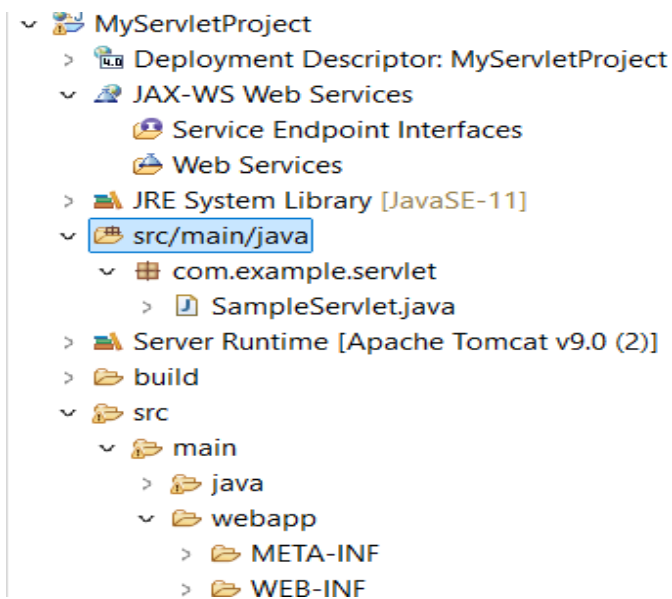
- When the servlet is first loaded → `init()` will print: **Servlet is being initialized.**
- When you hit the URL (like `/SampleServlet`) → `doGet()` will:
  - Print to console: **Handling GET request.**
  - Show HTML: `Hello from SampleServlet` in your browser.
- When you stop the server → `destroy()` will print: **Servlet is being destroyed.**

### Step 3: Project Structure

Your project will look like this:

MyServletProject/

```
|— src/
|   |— com/example/servlet/SampleServlet.java
|— WebContent/
|   |— WEB-INF/
|       |— web.xml (Optional if using @WebServlet)
|— build path libraries (Servlet API from Tomcat)
```



#### Step 4: Run the Project

1. Right-click on your project → **Run As** → **Run on Server**.
2. Choose **Apache Tomcat**.
3. Eclipse will deploy and launch the app.

✓ Open browser and go to:

<http://localhost:8080/MyServletProject/hell>

## Applications of Servlets

- Web apps like **online banking**, **e-commerce**, **ticket booking**, etc.
- Backend **APIs** serving mobile apps/websites.
- **Login and Authentication** systems.
- **Data fetching** from databases.
- **Admin panels**, **reporting tools**, etc.

## Disadvantages of Servlets

- ✗ Very **hard to maintain** (you write a lot of HTML inside Java code)
- ✗ Mixing **Java code** + **HTML** = messy and ugly
- ✗ Hard for designers and developers to work together
- ✗ Difficult for large projects — not modular

## What is JSP (JavaServer Pages)?

JSP technology is used to create web applications just like Servlet technology.

It is an **extension of Servlet technology** — internally JSP is **converted into a Servlet** by the container. It provides more functionality than servlet such as **expression language**, JSTL, etc.

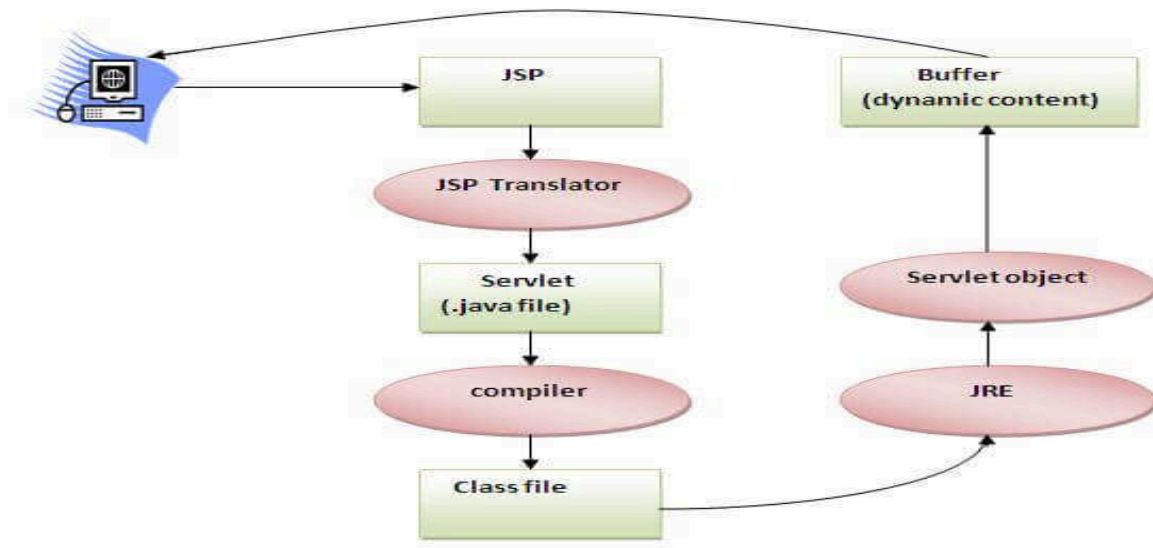
A JSP page consists of HTML tags and JSP tags. **The JSP pages are easier to maintain than Servlet because we can separate designing and development.** It provides some additional features such as Expression Language, Custom Tags, etc.

JSP = HTML page + Java code inside it to make it dynamic.

**JSP code is easier to manage than Servlets as it separates UI and business logic.**

## How JSP Works

1. **Client sends request** → JSP page.
2. **The JSP compiler** translates it into a **Servlet class**.
3. **Servlet executes**, generates HTML, and sends it back to the client.



## JSP Architecture

JSP follows a three-layer architecture:

- **Client Layer:** The browser sends a request to the server.
- **Web Server Layer:** The server processes the request using a JSP engine.
- **Database/Backend Layer:** Interacts with the database and returns the response to the client.

Layer	What It Really Means
Client Layer	Browser / Mobile App / Postman (sending HTTP request)
Web Server Layer	Controller + JSP (View) + Model classes (data) + DAO (all backend logic)
Database Layer	Actual Database (MySQL, Oracle) accessed through DAO

**What are we using nowadays instead of Servlets and JSP?**

## 1. Ease of Development (Code Simplicity)

**servlets:** You have to manually write a lot of HTML inside Java code using `out.println("<html>...</html>");`.

**JSP:** You write normal HTML and just insert Java code wherever needed using special tags (`<% ... %>`).

**Advantage:** JSP makes designing pages much **cleaner and readable** for frontend developers.

### Servlet HTML Example:

```
response.setContentType("text/html");

PrintWriter out = response.getWriter();

out.println("<html>");

out.println("<body>");

out.println("<h1>Welcome, User!</h1>");

out.println("</body>");

out.println("</html>");
```

### Same in JSP:

```
<html>

<body>

<h1>Welcome, User!</h1>

</body>
```

</html>

## 2. Separation of Concerns (MVC Architecture)

- **Servlets:** Mixes the *business logic* (Java code) and *presentation logic* (HTML) together.
- **JSP:** Encourages keeping **business logic in JavaBeans or Servlets** and using JSP **only for displaying data**.
  - Designers focus on JSP (HTML, CSS).
  - Developers focus on Servlets (backend logic).
- **Advantage:** Supports **MVC design pattern** easily — better for larger, maintainable applications.

## 3. Automatic Compilation

- **JSP:** When you deploy a JSP file, the server **automatically converts** it into a Servlet (internally) and compiles it.
- **Servlets:** You have to **manually write, compile, and deploy** your Java Servlet classes.

When you right-click on the `Servlet.java` file and select **Run As → Run on Server** in Eclipse, **the server will compile the servlet for you**

- **Advantage:** JSPs save development time because the **server handles the compilation**.

## 4. Built-in Features

- **JSP** supports powerful **tag libraries (JSTL)** and **Expression Language (EL)** to reduce Java code inside JSPs.
- Makes it much easier to display dynamic content without manually writing a lot of Java code.

**Example using EL:**

Hello, `${userName}`

## The evolution of web technologies

### 1. Servlets (Early Web Development)

#### What are Servlets?

- **Servlets** are Java programs that run on a web server and handle client requests (usually HTTP). They were introduced in **Java EE** (now Jakarta EE) to extend the capabilities of web servers and provide dynamic web content.

#### Advantages of Servlets:

- Servlets allow **dynamic web pages**, meaning they can respond to user actions and create content dynamically on the server.
- They are **platform-independent** since Java is platform-independent.
- Can handle **complex requests**, such as **database queries** or **file uploads**.

#### Drawbacks:

- The main issue with Servlets is that they **mix business logic** (Java code) with the **presentation logic** (HTML), making the code harder to maintain.
- They require a lot of boilerplate code to process HTTP requests and generate responses.
- **Server-side scripting** in Servlets isn't very clean or efficient, especially when it comes to separating logic and user interface (UI).

#### When and Why Replaced:

- Servlets were the foundation of dynamic content on the web, but their use became less favorable due to their lack of separation between logic and view (HTML).
  - Developers wanted a **cleaner separation of concerns**, which led to the adoption of **JSP**.
- 

### 2. JSP (JavaServer Pages)

#### What is JSP?



- **JSP** is a Java technology that enables developers to create dynamic web pages using HTML embedded with Java code. It was introduced to **separate business logic from the presentation layer**.

#### How JSP Works:

- HTML and Java code can coexist, but **Java logic is enclosed within special tags** (`<% %>`), so it's easier to manage the page's presentation while embedding logic for dynamic content.

#### Advantages of JSP over Servlets:

- **Separation of concerns:** Unlike Servlets, which mixed Java code and HTML, JSP allows Java logic to be embedded within HTML, making it easier to manage.
- Easier for **web designers** to work with, as they can focus on the HTML, while developers handle the Java logic.
- The **JSP engine** automatically compiles JSP pages into Servlets, so you don't have to manually deal with Servlets.

#### Drawbacks of JSP:

- Though JSP improved separation of logic and view, it still had issues with **too much Java code embedded within the HTML**.
- Large-scale applications became harder to maintain due to the amount of embedded logic.

#### When and Why Replaced:

- JSPs were often replaced or supplemented with modern frameworks like **Spring** because they didn't fully solve the problem of **maintainability** and **scalability** for large applications.

---

### 3. Spring Framework (2000s)

#### What is Spring?

- **Spring** is a comprehensive **Java framework** used to build enterprise-level applications. It provides features like **dependency injection (DI)**, **aspect-oriented programming (AOP)**, and a **modular approach** to application development.

## Key Features:

- **Inversion of Control (IoC):** Spring's DI mechanism allows objects to be managed by the framework, not by the application code. This leads to more maintainable and testable code.
- **Spring MVC:** A web module that implements the **Model-View-Controller (MVC)** design pattern, helping developers build clean, modular, and scalable web applications. This solved the issue of mixing logic and view in JSP.
- **Integration with other frameworks:** Spring integrates well with other technologies, such as **JPA (Java Persistence API)**, **Security**, **Messaging**, and more, allowing developers to build complex applications easily.

## Advantages of Spring:

- **Separation of concerns:** Spring's MVC model separates the **Controller (request handling)** from the **Model (business logic)** and the **View (presentation)**.
- **Modular:** Developers can use only the components they need, leading to less overhead.
- **Testability:** The use of DI makes Spring applications easier to test.
- **Flexibility:** Spring is agnostic of the underlying platform, allowing it to be used with various data sources, web servers, and other components.

## When and Why Replaced JSP?

- **Spring MVC** replaced JSP in many cases because it provided a much cleaner, more **modular approach** to handling HTTP requests and responses.
- The **MVC architecture** was easier to scale and maintain compared to JSP, which mixed too much logic and presentation.

---

## 4. Spring Boot (2013 - Present)

### What is Spring Boot?

- **Spring Boot** is an extension of the **Spring Framework** that simplifies **the setup and configuration of Spring applications**. It provides defaults for application setup and focuses on rapid development.

#### Key Features of Spring Boot:

- **Auto-configuration**: Spring Boot can automatically configure your application based on the libraries and dependencies you add to your project, removing the need for a lot of manual configuration.
- **Embedded Servers**: It comes with embedded web servers (e.g., Tomcat, Jetty) so that you don't need to install or configure a separate web server.
- **Minimal configuration**: Spring Boot provides **out-of-the-box solutions**, which make it easier for developers to quickly get started without requiring heavy XML configurations.
- **Microservices support**: Spring Boot simplifies the development of **microservices**, which has become increasingly popular for building scalable, maintainable applications.

#### Advantages of Spring Boot over Spring:

- **Reduced configuration**: Spring Boot eliminates most of the boilerplate configuration required in regular Spring applications.
- **Rapid development**: Developers can focus more on business logic and less on configuration.
- **Integrated tools**: It includes tools for testing, debugging, and monitoring, making development easier.
- **Microservices-ready**: Spring Boot makes it easy to build microservices using other Spring technologies like **Spring Cloud** and **Spring Data**.

#### When and Why Replaced Spring?

- While **Spring** is still a very powerful framework, **Spring Boot** has made it easier and faster to develop production-ready applications by automating the configuration and offering **standards for cloud-native applications**.

---

## 5. Microservices (2010s - Present)

### What are Microservices?

- **Microservices** are an architectural style where an application is structured as a collection of **small, independent services** that communicate over the network. Each service handles a specific business capability.

### Key Features of Microservices:

- **Independence:** Each microservice is **independent**, can be developed, deployed, and scaled separately.
- **Communication via APIs:** Microservices communicate with each other using standard protocols such as **REST** or **gRPC**.
- **Decentralized data management:** Each service typically has its own **database**, ensuring better isolation and scalability.
- **Fault tolerance:** Since microservices are independent, failure in one service doesn't bring down the whole system.

### Advantages of Microservices:

- **Scalability:** Individual services can be scaled independently, making the system more efficient.
- **Flexibility:** Microservices can be written in different programming languages and use different technologies, as long as they communicate through APIs.
- **Resilience:** If one service fails, others can continue to operate, improving the overall system's resilience.

### When and Why Replaced Monolithic Systems (like JSP/Servlets)?

- Traditional **monolithic architectures** (where everything is tightly coupled) became difficult to scale and maintain as applications grew.

- Microservices offer better modularity and scalability. They became popular in modern, cloud-based environments where **continuous integration/continuous deployment (CI/CD)** is key.
- Technologies like **Spring Boot** and **Spring Cloud** made it easier to implement microservices, offering tools for service discovery, configuration management, and inter-service communication.



Technology	Focus	Advantages	When and Why Replaced
<b>Servlets</b>	Server-side dynamic content generation	Allows dynamic web content	Replaced by JSP for better separation of concerns
<b>JSP</b>	Separating HTML and Java logic	Better separation of logic and presentation	Replaced by Spring (MVC) for better modularity and flexibility
<b>Spring</b>	Comprehensive Java framework (IoC, AOP)	Separation of concerns, easier testing	Replaced JSP for better modularity and testability
<b>Spring Boot</b>	Simplified Spring-based development	Auto-configuration, minimal setup	Replaced Spring for rapid development and microservices support
<b>Microservices</b>	Modular, independent services	Scalability, resilience, independent scaling	Replaced monolithic architectures (like JSP/Servlets) for cloud-native, scalable applications

### The difference between **Spring** and **Spring Boot**

Feature	Spring Framework	Spring Boot
Purpose	Comprehensive enterprise framework for developing applications using Spring	Provides simplified and rapid application development (RAD) features

Framework Complexity	High	Low to Medium
Configuration Options	XML/Annotation-based	XML/Annotation and application.properties/YAML based
Managing Project Dependency	Manual	Automatic with use of starter modules
Embedded Web Server	No	Yes ( <a href="#">Tomcat</a> , <a href="#">Jetty</a> , etc.). The default is Tomcat.
Project Setup Tools	Manual	<a href="#">Spring Initializer</a>
Convention over Configuration (CoC)	Limited	Strong
Auto-Configuration	No	Yes (Auto-configures beans based on libraries detected on the classpath)
External Configuration	Using <i>PropertyPlaceholderConfigurer</i>	<a href="#">Spring Boot properties</a> placed in <i>application.properties</i>
Logging	Explicit configuration via logback, log4j, etc.	Spring Boot Starter for <a href="#">Logging</a> (logback by default)

Database Connection Pooling	Explicit manual setup (e.g., Apache DBCP, C3P0)	Integrated ( <a href="#">HikariCP</a> by default)
Template Engines (e.g., Thymeleaf, FreeMarker)	Explicit manual configuration	Auto-configured based on included dependencies. <a href="#">Thymeleaf</a> and <a href="#">JSP</a> are automatically configured for web applications.
Microservices Support	Requires additional setup (e.g., <a href="#">Spring Cloud</a> )	Built-in support for <a href="#">microservices</a> architecture
Testing Support	Explicit manual configuration (e.g., <a href="#">JUnit 5</a> , TestNG)	Spring Boot Starter for Testing (e.g., <a href="#">@SpringBootTest</a> )
Production-Ready Features	Explicit manual setup for Actuator	<a href="#">Built-in Actuator</a> (metrics, health checks, etc.) by adding the starter module
Aspect-Oriented Programming (AOP)	Manual setup of AOP	Auto-configuration support for <a href="#">AOP</a>
Security Configuration	Requires manual setup (e.g., <a href="#">Spring Security</a> )	Simplified default security with overridable configurations
Custom Error Pages	Manual setup (e.g., <i>ErrorController</i> )	Simplified error handling with default error pages and custom configuration

Transaction Management	Requires manual setup (e.g., <i>@Transactional</i> )	Auto-configured using <i>@EnableTransactionManagement</i>
Database Access / DataSource Configuration	Explicit setup for JDBC, JPA, Hibernate, etc.	<a href="#">DataSource autoconfiguration</a> with Spring Data Starter with custom configurations
Internationalization (i18n)	Manual configuration (e.g., <i>ResourceBundleMessageSource</i> )	Auto-configuration for message sources in the classpath
Developer Experience	Steeper learning curve, more control	Faster development with conventions

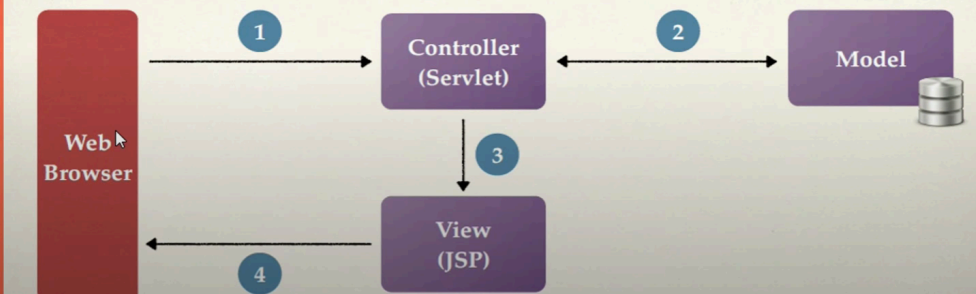
Feature	Spring Framework	Spring Boot
Configuration	Manual, XML or annotation-based configuration	Auto-configuration, minimal setup
Setup Time	Time-consuming setup and configuration	Rapid setup with sensible defaults
Web Server	External server setup required (Tomcat, Jetty)	Embedded servers (Tomcat, Jetty, Undertow)
Deployment	War files, needs an external server	Self-contained JAR with embedded server
Microservices	Requires additional tools like Spring Cloud	Built-in support for microservices with Spring Cloud
Dependencies	Manually handled by developers	Starter dependencies for automatic setup
Command Line Support	Not available	Spring Boot CLI for testing and running apps
Customization	Fully customizable	Sensible defaults, but customizable if needed



## Registration Form using JSP + Servlet + JDBC + Mysql Example

- In this video, we will build a simple Employee Registration module using JSP, Servlet, JDBC and MySQL database.

### Model-View-Controller (MVC)



## MVC Pattern

- Model-View-Controller (MVC) is a pattern used in software engineering to separate the application logic from the user interface. As the name implies, the MVC pattern has three layers.
- The Model defines the business layer of the application, the Controller manages the flow of the application, and the View defines the presentation layer of the application.

## Tools and technologies used

- JSP - 2.2 +
- IDE - STS/Eclipse Neon.3
- JDK - 1.8 or later
- Apache Tomcat - 8.5
- JSTL - 1.2.1
- Servlet API - 2.5
- MySQL - mysql-connector-java-8.0.13.jar

## Development Steps

---

- Create an Eclipse Dynamic Web Project
- Add Dependencies
- Project Structure
- MySQL Database Setup
- Create a JavaBean - Employee.java
- Create a EmployeeDao.java
- Create a EmployeeServlet.java
- Create a employeeeregister.jsp
- Create a employeeedetail.jsp
- Demo

1) **Created a** Table in MYsql work bench

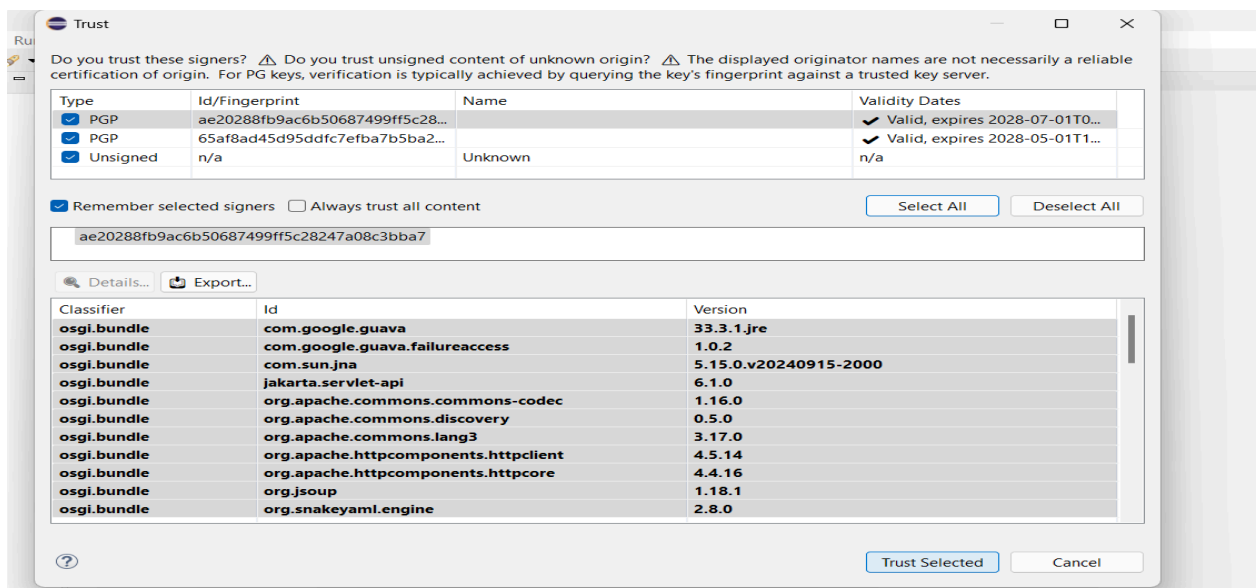
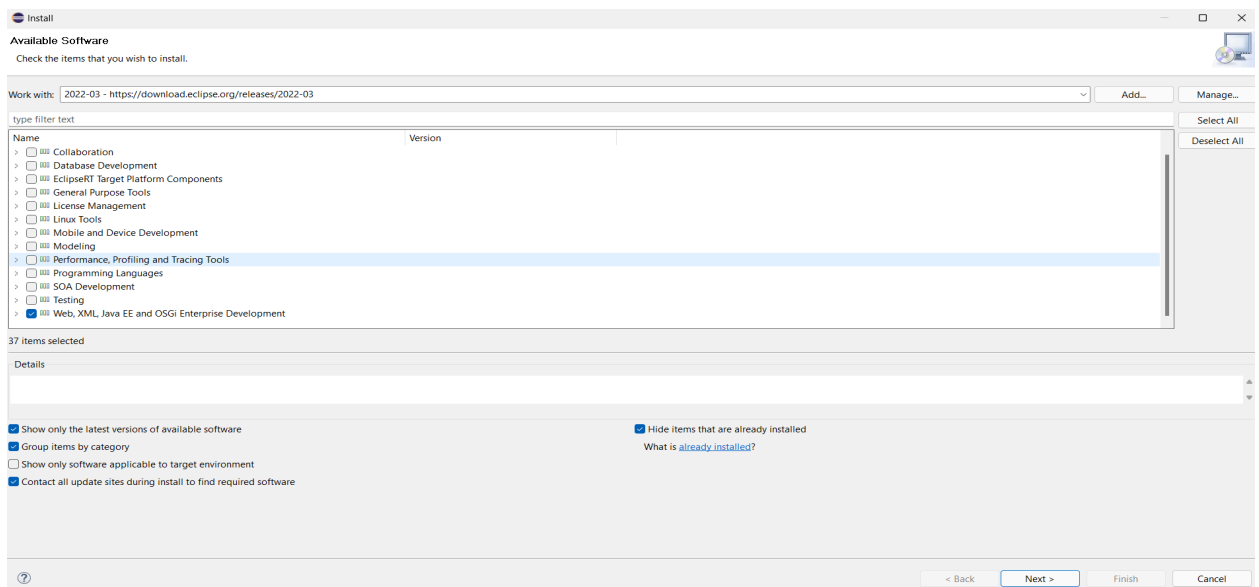
```
1 • create database springmvcexample;
2 • use springmvcexample;
3 • CREATE TABLE `employee` (
4     `id` int(3) NOT NULL PRIMARY KEY,
5     `first_name` varchar(20) DEFAULT NULL,
6     `last_name` varchar(20) DEFAULT NULL,
7     `username` varchar(250) DEFAULT NULL,
8     `password` varchar(20) DEFAULT NULL,
9     `address` varchar(45) DEFAULT NULL,
10    `contact` varchar(45) DEFAULT NULL
11 );
12 • select * from employee;
```

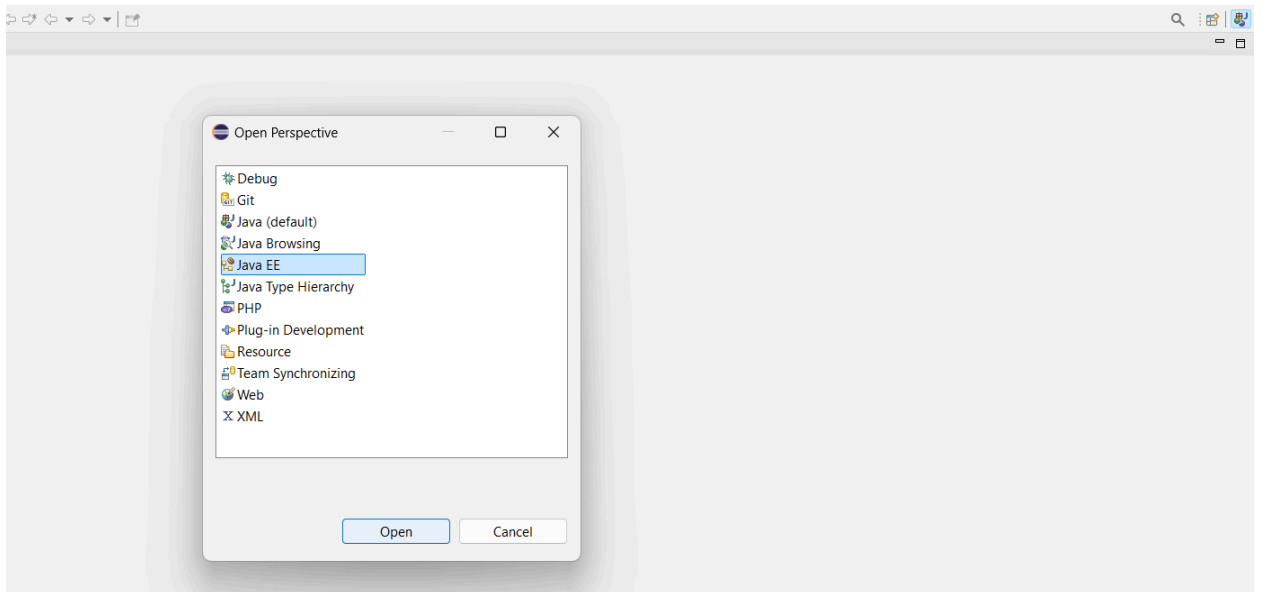
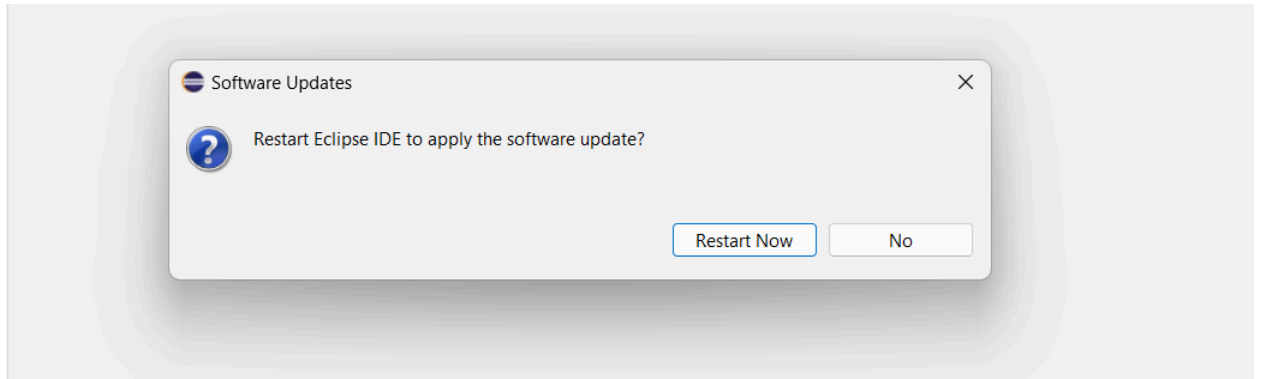
2) Create Dynamic Web Project in Eclipse , If it is not there follow below methods to get it.

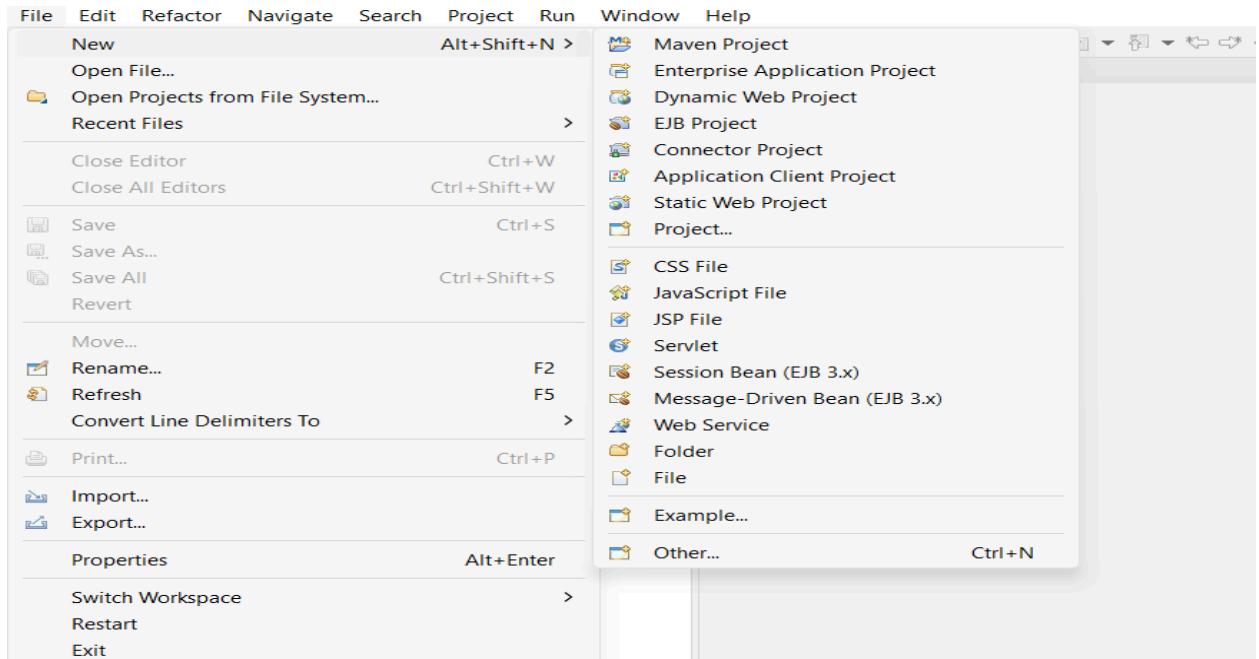
### Method 1

To create a **Dynamic Web Project** in Eclipse (Version: 2022-03), follow these steps:

- Go to **Help > Install New Software**. In Work worth click on drop down and select release notes



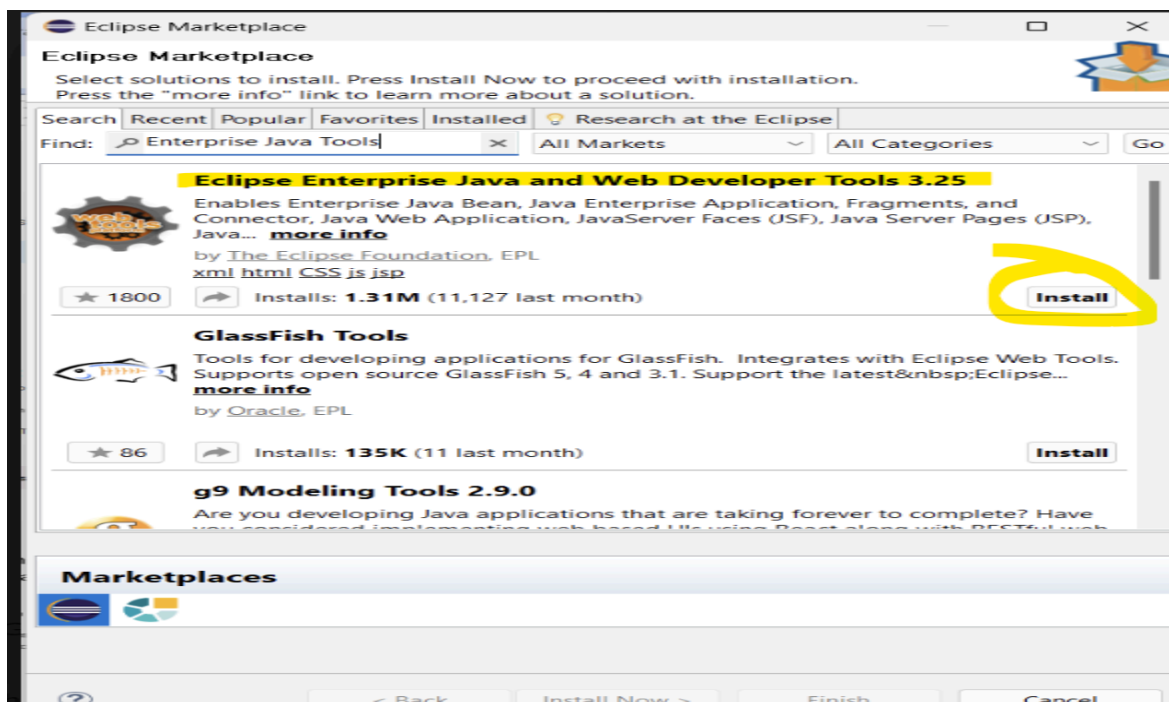




## Method 2

### ✓ Step 1: Install Java EE (Enterprise Edition) Plugins

1. Go to **Help** → **Eclipse Marketplace**.
2. Search for "Eclipse Java EE Developer Tools" or "**Enterprise Java Tools**".
3. Click **Install** and restart Eclipse when prompted.

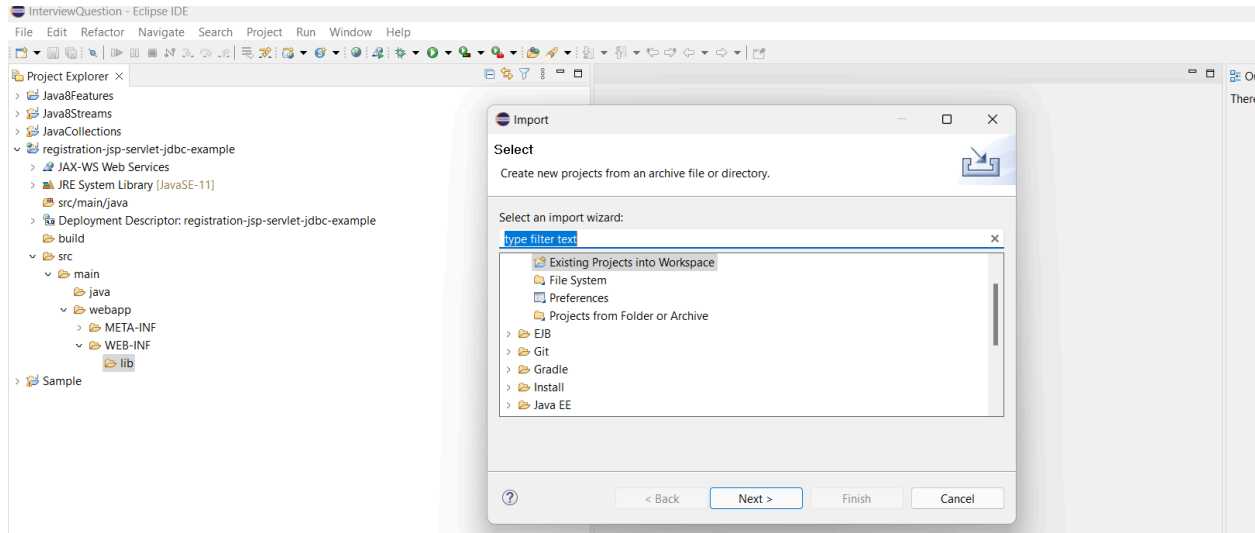


## Step2

Create a project by selecting as Dynamic web project

Add .jar dependency manually in the **lib** folder under **WEB-INF**.

Right-click on the **lib** folder → **Import** → **File System** → Browse and select your .jar files → **Finish**.



Create **views** folder in **WEB-INF** for JSPs.

## JDBC Flow

### JDBC Code Example

We'll create a simple Java program that:

1. Connects to a MySQL database.
2. Inserts a record into a table.
3. Retrieves the data from the table.

### Prerequisites:

- MySQL installed (you have both 5.7 and 8.0 installed).
- A database named **testdb** with a table **users**.

### Database Setup (MySQL):

**CREATE DATABASE** testdb;

**USE** testdb;

```
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(100)
);
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
public class JDBCDemo {
    public static void main(String[] args) {
        // JDBC URL, username, and password for MySQL
        String jdbcURL = "jdbc:mysql://localhost:3306/testdb";
        String username = "root"; // Replace with your MySQL username
        String password = "password"; // Replace with your MySQL password
        // SQL statements
        String insertSQL = "INSERT INTO users (name, email) VALUES (?, ?)";
        String selectSQL = "SELECT * FROM users";
        String updateSQL = "UPDATE users SET name = ?, email = ? WHERE id = ?";
        String deleteSQL = "DELETE FROM users WHERE id = ?";
        try {
            // Step 1: Load and register the JDBC driver (optional for newer versions)
            Class.forName("com.mysql.cj.jdbc.Driver");
            // Step 2: Establish a connection to the database
            Connection connection = DriverManager.getConnection(jdbcURL,
username, password);
            System.out.println("Connected to the database successfully!");
            // Step 3: Create a PreparedStatement for inserting data
            PreparedStatement insertStmt = connection.prepareStatement(insertSQL);
            insertStmt.setString(1, "John Doe"); // Setting the first ? (name)
            insertStmt.setString(2, "john.doe@example.com"); // Setting the second ?
(email)
            // Step 4: Execute the insert statement
            int rowsInserted = insertStmt.executeUpdate();
            if (rowsInserted > 0) {
                System.out.println("A new user was inserted successfully!");
            }
        }
    }
}
```

```

// Step 5: Create a PreparedStatement for updating data
PreparedStatement updateStmt =
connection.prepareStatement(updateSQL);
updateStmt.setString(1, "Jane Doe"); // New name
updateStmt.setString(2, "jane.doe@example.com"); // New email
updateStmt.setInt(3, 1); // Update user with ID 1
// Step 6: Execute the update statement
int rowsUpdated = updateStmt.executeUpdate();
if (rowsUpdated > 0) {
    System.out.println("User data updated successfully!");
}
// Step 7: Create a PreparedStatement for selecting data
PreparedStatement selectStmt = connection.prepareStatement(selectSQL);
// Step 8: Execute the select query
ResultSet resultSet = selectStmt.executeQuery();
// Step 9: Process the result set
while (resultSet.next()) {
    int id = resultSet.getInt("id");
    String name = resultSet.getString("name");
    String email = resultSet.getString("email");
    System.out.println("ID: " + id + ", Name: " + name + ", Email: " +
email);
}
// Step 10: Create a PreparedStatement for deleting data
PreparedStatement deleteStmt = connection.prepareStatement(deleteSQL);
deleteStmt.setInt(1, 1); // Delete user with ID 1
// Step 11: Execute the delete statement
int rowsDeleted = deleteStmt.executeUpdate();
if (rowsDeleted > 0) {
    System.out.println("User deleted successfully!");
}
// Step 12: Close resources
resultSet.close();
insertStmt.close();
updateStmt.close();
selectStmt.close();
deleteStmt.close();
connection.close();
} catch (ClassNotFoundException e) {
    System.out.println("MySQL JDBC Driver not found.");
}

```



```

        e.printStackTrace();
    } catch (SQLException e) {
        System.out.println("Database connection error.");
        e.printStackTrace();
    }
}
}
}

```

Step-by-Step Explanation:

Import JDBC Classes:

```
import java.sql.*;
```

1. These classes handle database connectivity (**Connection**, **DriverManager**), SQL execution (**PreparedStatement**), and data retrieval (**ResultSet**).

Define Database Connection Details:

```
String jdbcURL = "jdbc:mysql://localhost:3306/testdb";
```

```
String username = "root";
```

```
String password = "password";
```

2. Replace **username** and **password** with your MySQL credentials.

Load MySQL JDBC Driver:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

3. This registers the MySQL JDBC driver. For newer Java versions, this is optional.

Establish Connection:

```
Connection connection = DriverManager.getConnection(jdbcURL, username, password);
```

4. This opens a connection to the database using the provided credentials.

Insert Data Using **PreparedStatement**:

```
t
```

```
String insertSQL = "INSERT INTO users (name, email) VALUES (?, ?)";
```

```
PreparedStatement insertStmt = connection.prepareStatement(insertSQL);
```

```
insertStmt.setString(1, "John Doe");
```

```
insertStmt.setString(2, "john.doe@example.com");
```

```
insertStmt.executeUpdate();
```

5.
  - The **?** placeholders are filled using **setString()**.

- `executeUpdate()` runs the insert command.

Retrieve Data:

```
String selectSQL = "SELECT * FROM users";  
PreparedStatement selectStmt = connection.prepareStatement(selectSQL);  
ResultSet resultSet = selectStmt.executeQuery();
```

6.
  - `executeQuery()` runs the SELECT statement and returns the result set.

Process the Result Set:

```
while (resultSet.next()) {  
    int id = resultSet.getInt("id");  
    String name = resultSet.getString("name");  
    String email = resultSet.getString("email");  
    System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);  
}
```

7. This loop fetches each row from the `ResultSet` and prints the data.

Close Resources:

```
resultSet.close();  
insertStmt.close();  
selectStmt.close();  
connection.close();
```

8. Always close JDBC resources to avoid memory leaks.

UPDATE Operation:

```
String updateSQL = "UPDATE users SET name = ?, email = ? WHERE id = ?";  
PreparedStatement updateStmt = connection.prepareStatement(updateSQL);  
updateStmt.setString(1, "Jane Doe");  
updateStmt.setString(2, "jane.doe@example.com");  
updateStmt.setInt(3, 1); // Updates the user with ID 1  
int rowsUpdated = updateStmt.executeUpdate();
```

1. This updates the user's name and email for the record with `id = 1`.

DELETE Operation:

```
String deleteSQL = "DELETE FROM users WHERE id = ?";
```

```
PreparedStatement deleteStmt = connection.prepareStatement(deleteSQL);
deleteStmt.setInt(1, 1); // Deletes the user with ID 1
int rowsDeleted = deleteStmt.executeUpdate();
```

2. This deletes the user with `id = 1` from the `users` table.

Execution Flow:

- Insert a new user → Update the user's details → Display all users → Delete the user → Close resources.

## Adding Apache Tomcat to run web application

**Apache Maven** and **Apache Tomcat** are different tools, serving distinct purposes:

### 1. Apache Maven (Build Tool)

- **Purpose:** Manages project builds, dependencies, and documentation.
- **Usage:** Automates compiling code, running tests, packaging applications (like WAR or JAR files).

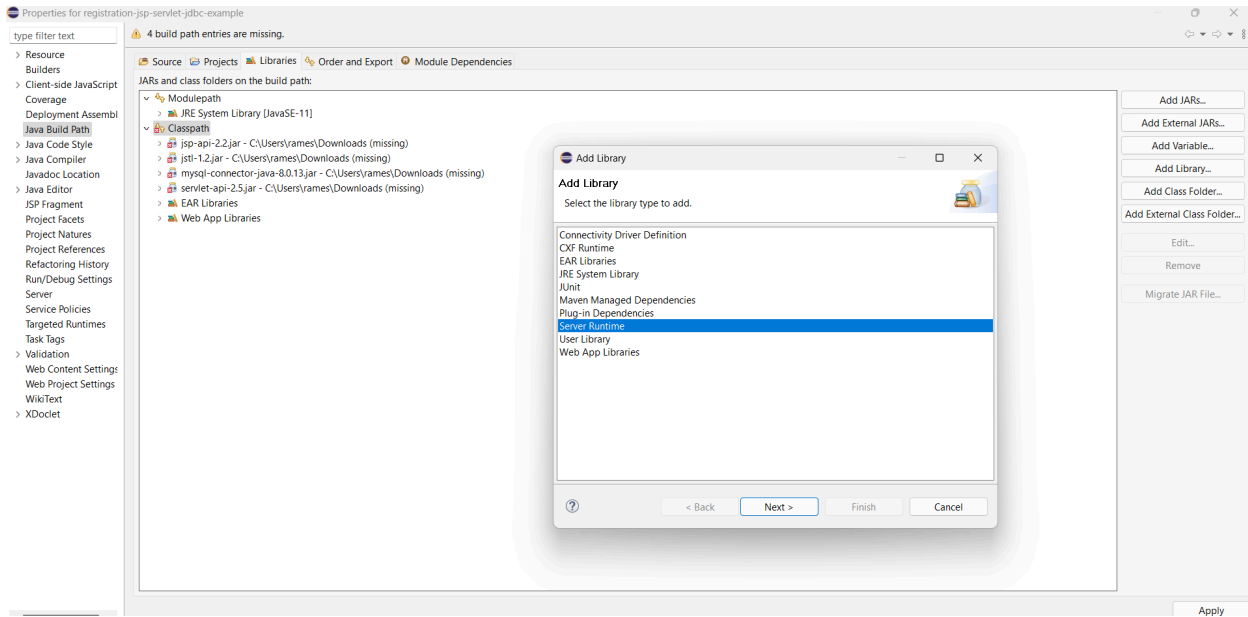
**Command Example:**

```
mvn clean install
```

- 
- **File:** Uses `pom.xml` to manage dependencies.

### 2. Apache Tomcat (Web Server)

- **Purpose:** Deploys and runs Java web applications (like Servlets, JSPs).
- **Usage:** Hosts web apps, handles HTTP requests.
- **File Example:** Deploys `.war` files generated by Maven.

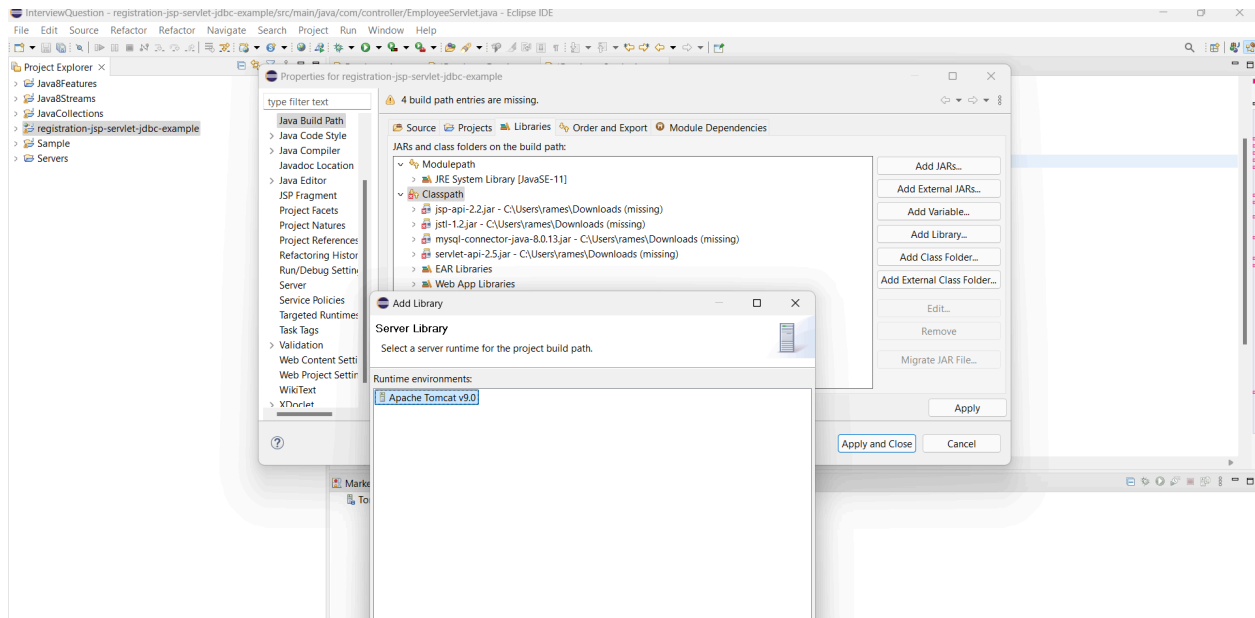


## Add Apache Tomcat as a Server in Eclipse:

- Open the **Servers** view in Eclipse. If it's not already open, go to **Window > Show View > Servers**.
- Right-click in the **Servers** tab and select **New > Server**.

## Configure Tomcat Server Runtime:

- In the **New Server** wizard, select **Apache > Tomcat v9.0 Server** (or the version you downloaded).
- Click **Next**.



The `EmployeeServlet` you've shared is a Java servlet that handles HTTP POST requests to register an employee in the system. Let's break it down:

## 1. Servlet Setup

- The `@WebServlet("/register")` annotation declares the servlet and maps it to the `/register` URL pattern. This means when the user submits a form to `/register`, the `doPost` method of this servlet will handle the request.

## 2. Member Variables

- `private EmployeeDao employeeDao;` A variable of type `EmployeeDao` which is presumably responsible for database interactions related to the `Employee` object (e.g., saving employee data to a database).

## 3. `init()` Method

- This method is called once when the servlet is initialized. It creates a new instance of `EmployeeDao` which will be used later to interact with the database.

```
employeeDao = new EmployeeDao();
```

## 4. `doPost()` Method

- This method handles HTTP POST requests. It extracts the employee's data from the request and performs the necessary operations.

Here's the flow:

#### Extracting Request Parameters:

```
String firstName = request.getParameter("firstName");
String lastName = request.getParameter("lastName");
String username = request.getParameter("username");
String password = request.getParameter("password");
String address = request.getParameter("address");
String contact = request.getParameter("contact");
```

1. These lines retrieve the parameters (e.g., first name, last name, username, password, etc.) that are sent by the form in the request body.

#### Creating an Employee Object:

```
Employee employee = new Employee();
employee.setFirstName(firstName);
employee.setLastName(lastName);
employee.setUsername(username);
employee.setPassword(password);
employee.setContact(contact);
employee.setAddress(address);
```

2. An `Employee` object is created, and the extracted parameters are set as its properties.

#### Registering the Employee:

```
employeeDao.registerEmployee(employee);
```

3. This calls the `registerEmployee` method from `EmployeeDao` to save the employee data to the database.

#### Exception Handling:

```
} catch (Exception e) {
    e.printStackTrace();
}
```

4. Any exceptions thrown during the registration process are caught and logged. This is a basic error handling mechanism, but it could be improved by providing more meaningful error messages to the user.

**Redirecting:**

```
response.sendRedirect("employeedetails.jsp");
```

5. After the employee is successfully registered, the user is redirected to [employeedetails.jsp](#). This could be a page showing the employee details or a confirmation message.