

## 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: ");

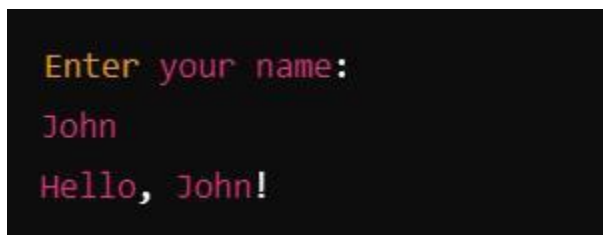
        String name = scanner.nextLine();

        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.

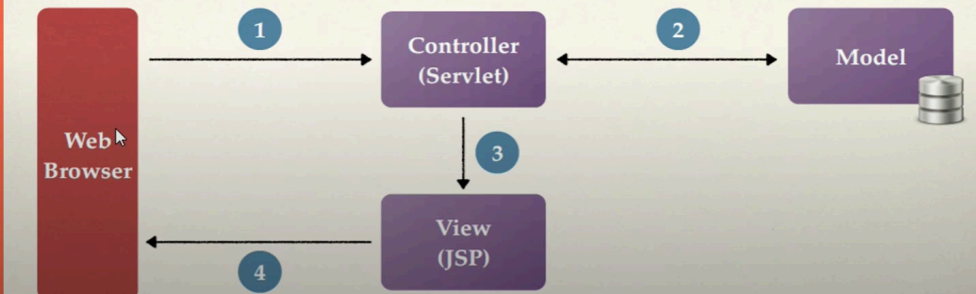
### 3 .Java ME (Micro Edition)

- **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.

## 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

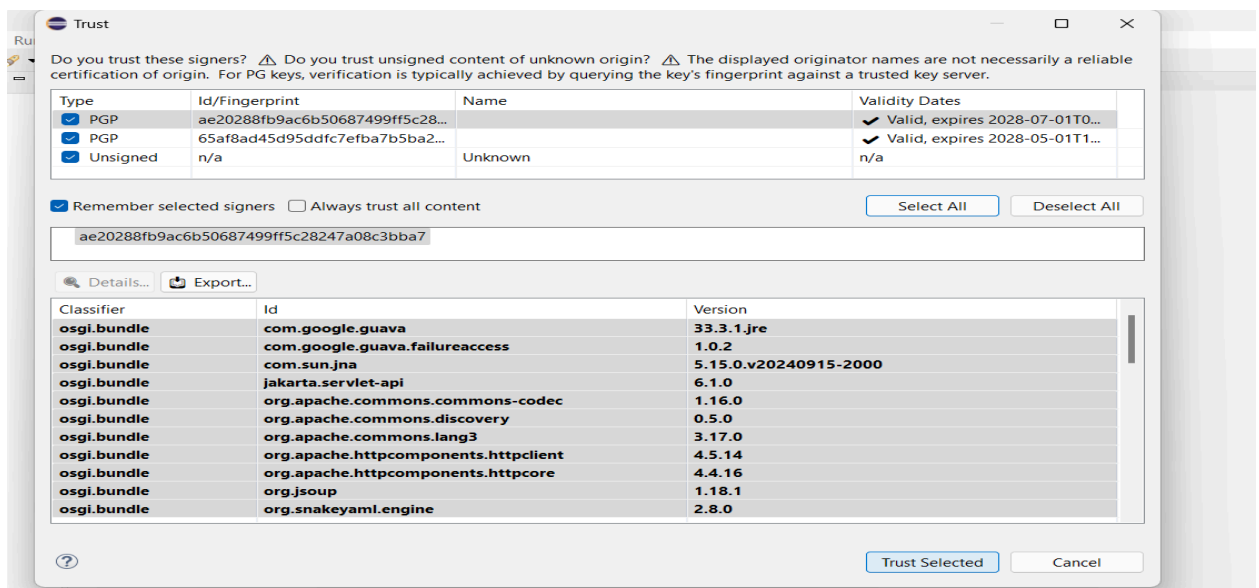
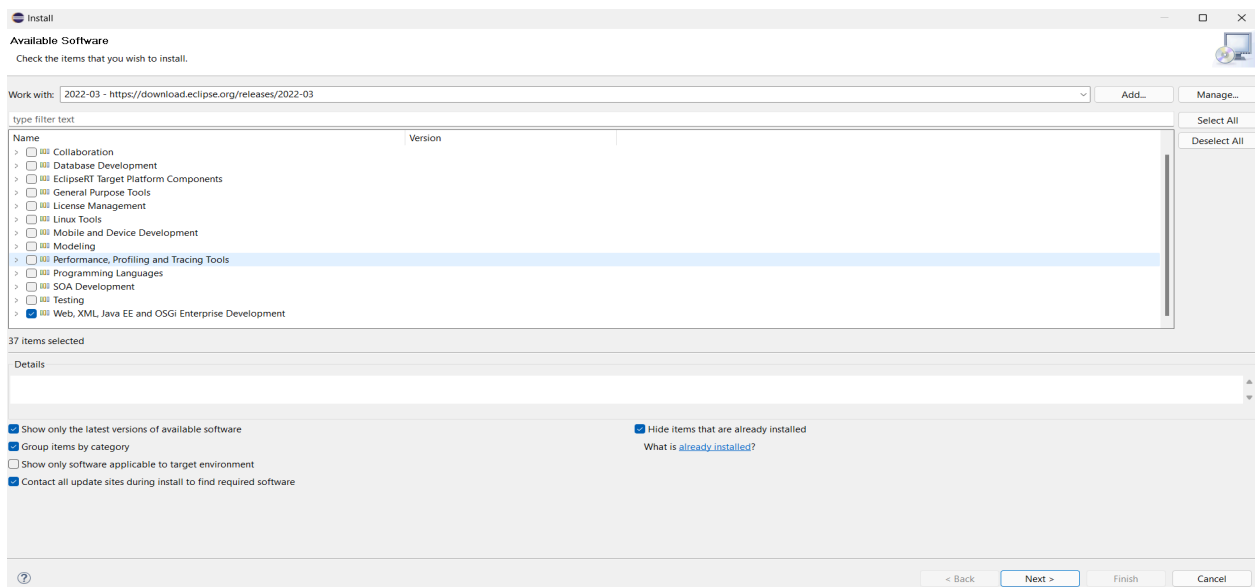
```
1 • create database springmvcexample;
2 • use springmvcexample;
3 • CREATE TABLE `employee` (
4     `id` int(3) NOT NULL PRIMARY KEY,
5     `first_name` varchar(20) DEFAULT NULL,
6     `last_name` varchar(20) DEFAULT NULL,
7     `username` varchar(250) DEFAULT NULL,
8     `password` varchar(20) DEFAULT 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.

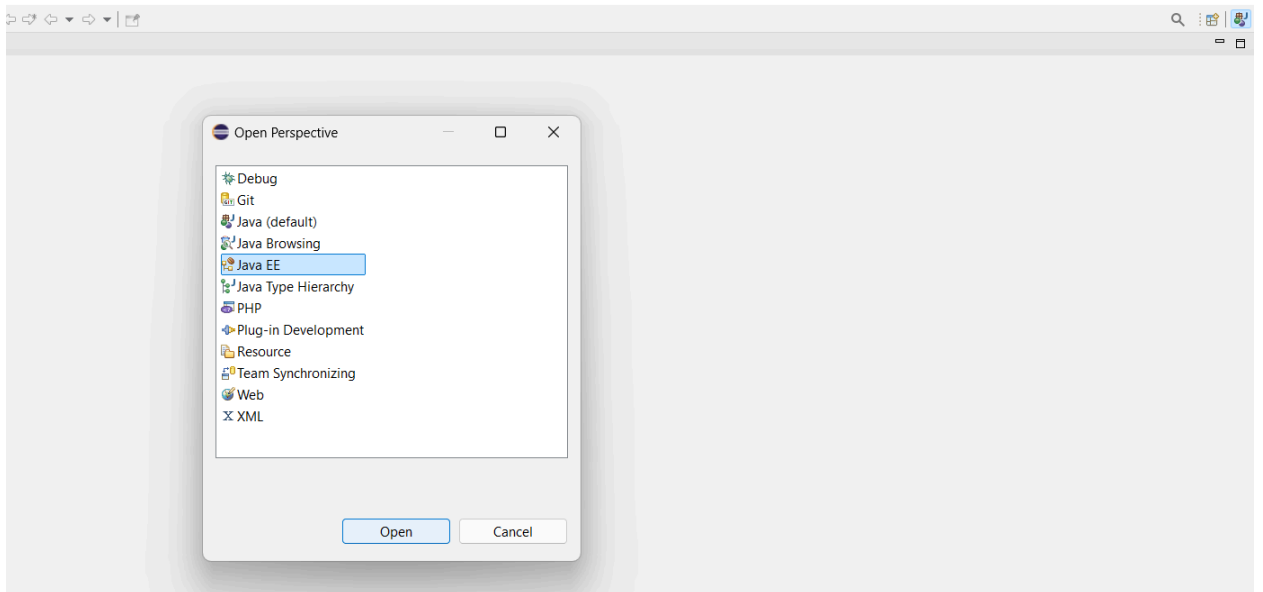
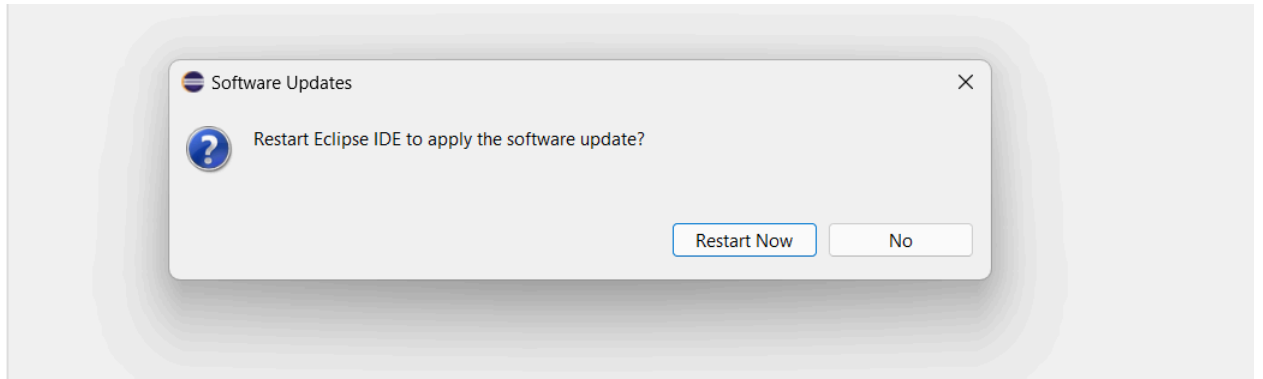
### Method 1

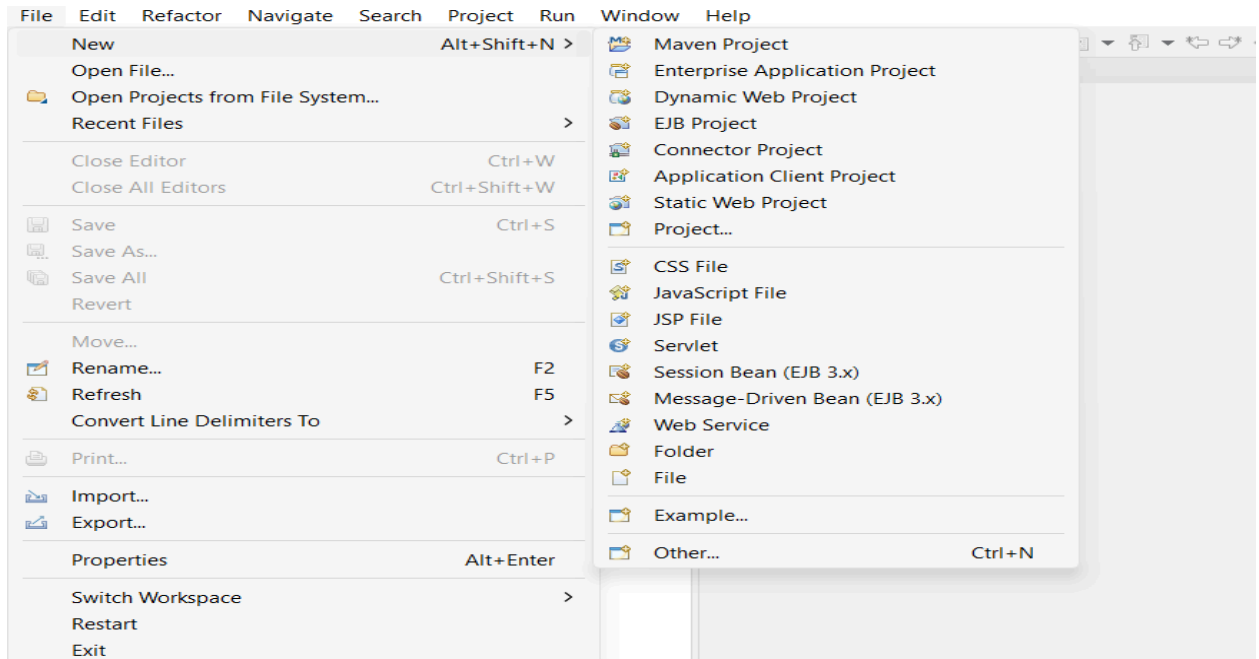
To create a **Dynamic Web Project** in Eclipse (Version: 2022-03), follow these steps:

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





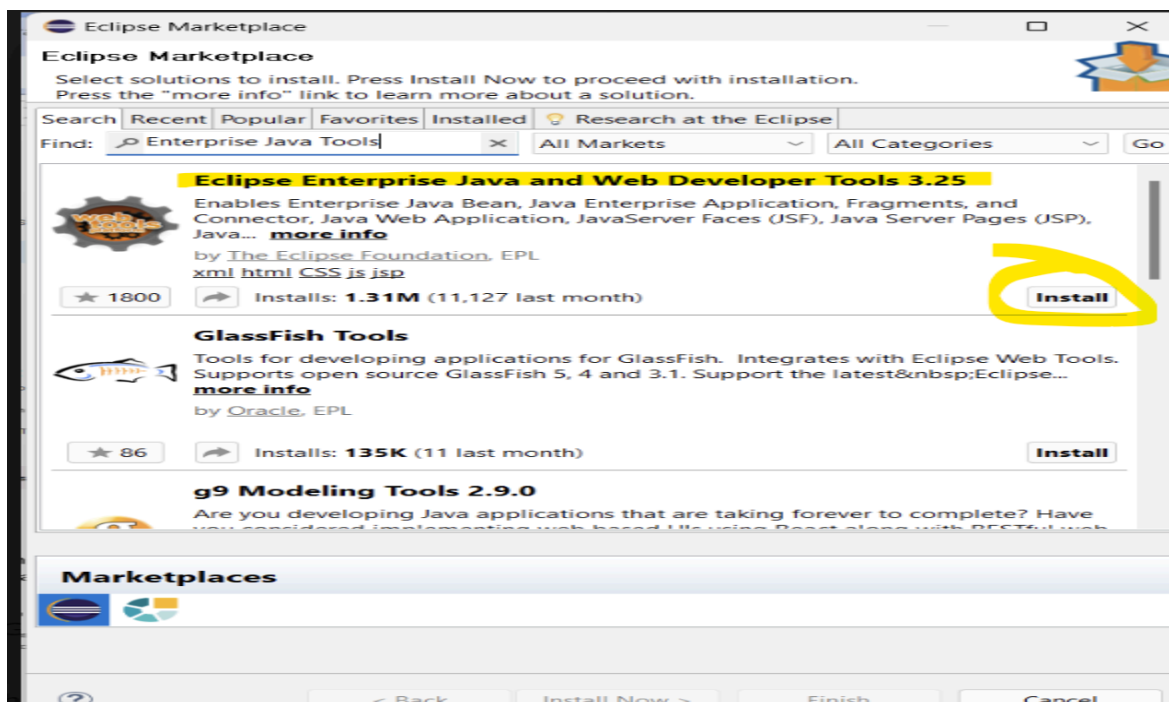




## Method 2

### ✓ 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.

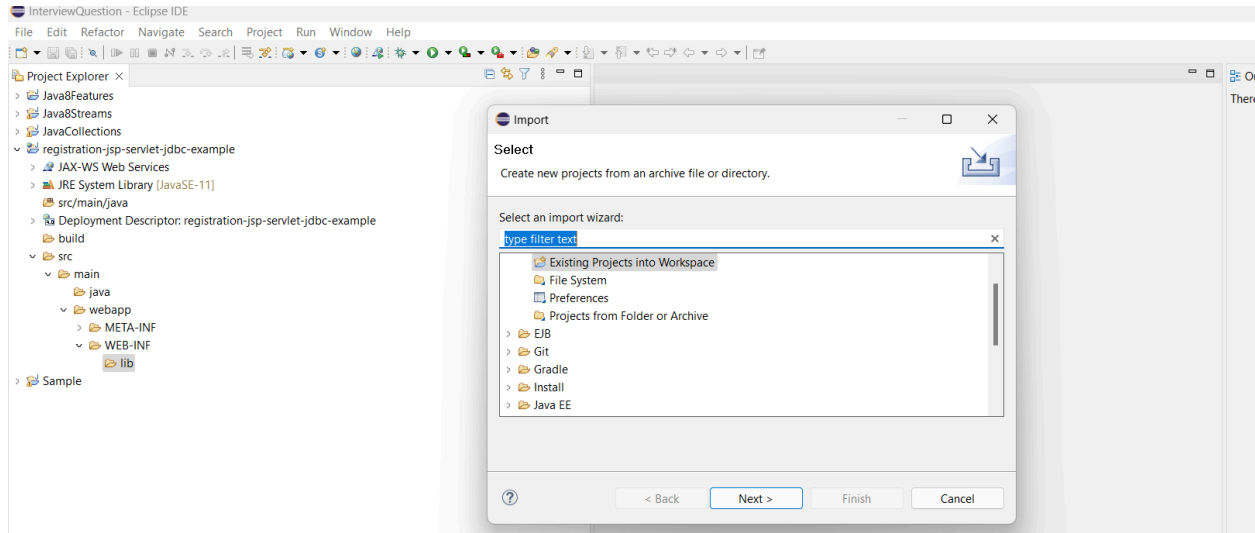


## Step2

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!");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
    // 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).

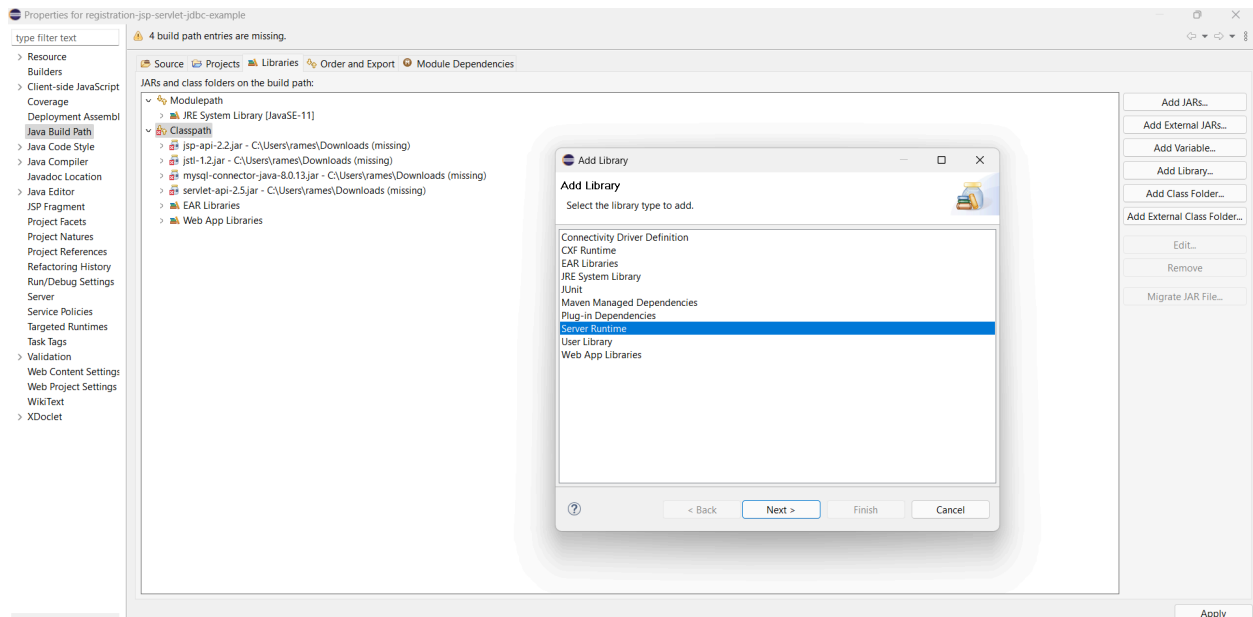
### Command Example:

`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.



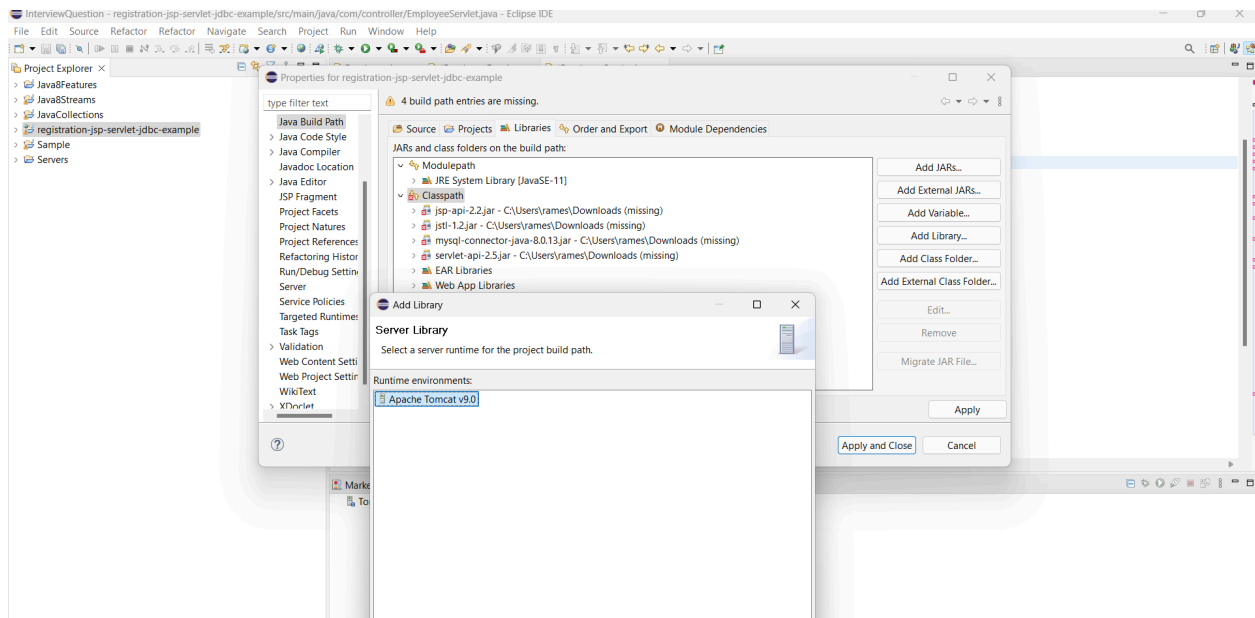
### Add Apache Tomcat as a Server in Eclipse:

- 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.