Institute of Technology

Scholl of Computer

Department of Software Engineering

Course Title: Advanced Programming

Chapter Five

Java Web Technologies

Lecture by: Demeke G.

AY.2017

Outline

- Introduction to Java web application
- Servlets
- JSP (JavaServer Pages)
- Hibernate
- Spring Boot Framework
- RESTful APIs

Introduction

- In 2017, **Java EE** transitioned to the Eclipse Foundation and was renamed **Jakarta EE**.
- **Java EE** (Java Platform, Enterprise Edition), now known as **Jakarta EE**, is a set of specifications and APIs for building large-scale, distributed, and robust enterprise applications in Java.
- **Jakarta EE** provides a framework for developing web applications, **microservices**, and **enterprise-level applications** with standardized solutions to common challenges like scalability, security, and persistence.
- A web application is an application accessible from the web.
- Servlet technology is used to create web application that resides at server side and generates dynamic web page.

Cont..

- A web application is composed of web components like **Servlet**, **JSP**, etc. and other components such as **HTML**.
- Servlet technology is **robust and scalable** because of java language.
- Java web applications can run on any platform with a JVM
- Java offers frameworks like **Spring**, **Hibernate**, and tools like **Tomcat** and **GlassFish** for efficient development.
- Java provides in-built security features like encryption, authentication, and secure socket layer (SSL) integration.
- Java applications can handle large amounts of traffic and data efficiently
- Servers like Apache Tomcat and GlassFish run Java web applications and manage request routing, application deployment, and scalability.

Cont..

Core Technologies in Java EE (Jakarta EE)

- **Servlets:** Core component for handling HTTP requests and responses and enables the creation of dynamic web applications.
- **JavaServer Pages (JSP)** : server-side technology that simplifies the creation of dynamic, platform-independent web pages by allowing developers to embed Java code directly into HTML.
- **Enterprise JavaBeans (EJB) :**is a server-side component architecture for modular, distributed, and transactional business applications in Java. Manages complex operations like distributed transactions and session management
- **Java Persistence API (JPA):** is a specification in Java for managing relational data in enterprise applications. JPA allows developers to interact with databases using Java objects without requiring extensive SQL code.

Cont..

- **Java Message Service (JMS)**
  - Provides messaging capabilities for asynchronous communication in distributed systems.
- **Java API for RESTful Web Services (JAX-RS)**
  - Simplifies the development of RESTful web services.
  - Annotated-based API for creating lightweight web services.
- **Java API for XML Web Services (JAX-WS)**
  - Simplifies the creation of Simple Object Access Protocol (SOAP-based) web services.
- **Contexts and Dependency Injection (CDI)**
  - Manages the lifecycle of objects and their dependencies in enterprise applications.
  - Promotes loose coupling and modularity.
- **JavaServer Faces (JSF)**
  - A framework for building component-based user interfaces for web applications
- **Java Transaction API (JTA)**
  - Manages transactions in enterprise applications, including distributed

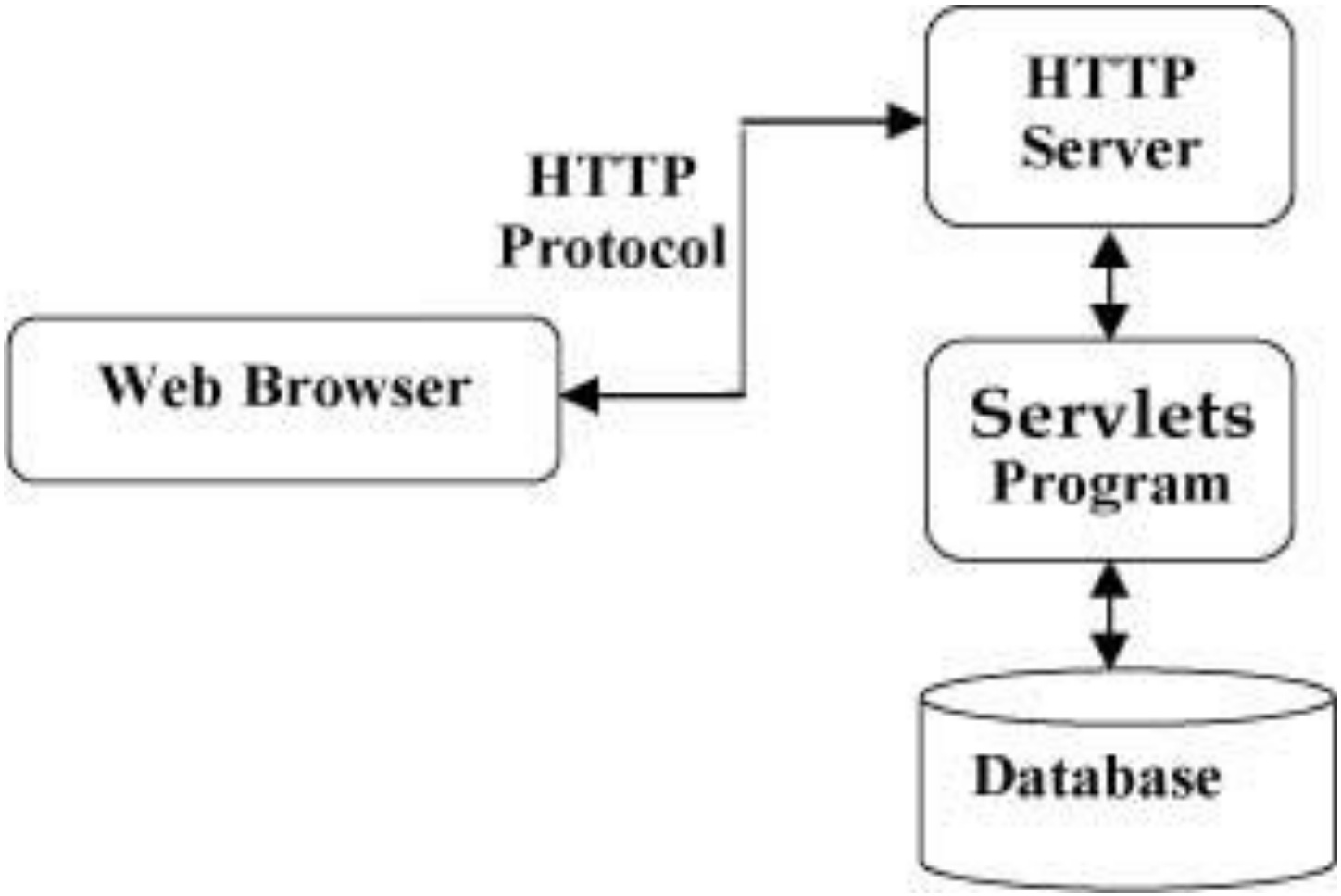
Servlet

- A technology used to create web application
- Servlets process client requests, execute business logic, and generate dynamic responses.
- An API that provides many interfaces and classes.
- Servlet is an interface that must be implemented for creating any servlet.
- Servlet is a class that extend the capabilities of the servers and respond to the incoming request.
- It can respond to any type of requests.
- Servlet is a web component that is deployed on the server to create dynamic web page.

Advantages of Servlets

- **Efficient**: Uses threads instead of creating new processes for each request.
- **Robust:** Servlets are managed by JVM so no need to worry about memory leak, garbage collection etc.
- **Scalability**: Can handle multiple client requests simultaneously.
- **Platform-Independent**: Runs on any server that supports the Java Servlet API (e.g., Tomcat, GlassFish).
- **Portability:** Can run on any servlet container that follows the Java Servlet specification.
- **Extensibility**: Can be integrated with other Java technologies like JSP, Hibernate, and Spring.
- **Secure:** Built on the Java platform, offering security features like encryption and authentication.

Servlets Architecture



Servlets Tasks

- **Read the explicit data sent by the clients.**

  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** . Like **cookies**, **media** types and compression schemes the browser understands

- **Process the data and generate the results**. Like talking to a database, invoking a Web service, or computing the response directly.

Cont..

- **Send the explicit data to the clients.** 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** 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.
- **Creating RESTful APIs.**
- **Managing sessions and user authentication.**

Servlet Terminology

1. HTTP
2. HTTP Request Types
3. Difference between Get and Post method
4. Web Server
5. Container
6. Content Type

HTTP (Hyper Text Transfer Protocol)

- Http is the protocol that allows web servers and browsers to exchange data over the web.
- It is a request response protocol.
- Http uses reliable TCP connections by default on TCP port 80.
- It is stateless means each request is considered as the new request. In other words, server doesn't recognize the user by default.

Http Request Methods

- Every request has a header that tells the status of the client.
- There are many request methods. **Get** and **Post** requests are mostly used.

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.

The difference between Get and Post

GET	POST
only limited amount of data can be sent because data is sent in header.	large amount of data can be sent because data is sent in body.
Get request is not secured because data is exposed in URL bar.	Post request is secured because data is not exposed in URL bar.
Get request can be bookmarked	Post request cannot be bookmarked
Get request is idempotent. It means second request will be ignored until response of first request is delivered.	Post request is non-idempotent
Get request is more efficient and used more than Post	Post request is less efficient and used less than get.
Only ASCII characters allowed	No restrictions. Binary data is also allowed

Content Type

- Content Type is also known as MIME (Multipurpose internet Mail Extension) Type.
- It is a **HTTP header** that provides the description about what are you sending to the browser.

There are many content types:

- text/html,text/plain,application/msword
- application/vnd.ms-excel
- application/jar,application/pdf
- application/octet-stream,application/x-zip
- images/jpeg,video/quicktime etc.

Container

- A part of a web server or application server that interacts with Java servlets.
- It provides runtime environment for JavaEE (j2ee) applications.
- It performs many operations that are given below:
  - ❖Managing the lifecycle of servlets
  - ❖Mapping URLs to servlets
  - ❖ handling requests and responses
  - ❖enforces security constraints defined in the deployment descriptor

Web Server

- Web server contains only web or servlet container.
- Receives and processes HTTP requests.
- Serves static content (HTML, CSS, JS, images, etc.) from the file system.
- Maps incoming URL requests to corresponding files on the server.
- Implements caching strategies to speed up content delivery.
- Uses load balancing to distribute traffic across multiple servers.
- Example of Web Servers are: **Apache HTTP Server with Apache Tomcat:, Nginx with Jetty**

Life Cycle of a Servlet

- A servlet life cycle can be defined as the entire process from its creation till the destruction.

1) The servlet is initialized by calling the init () method.
2) The servlet calls service() method to process a client's request.
3) The servlet is terminated by calling the destroy() method.
4) Finally, servlet is garbage collected by the garbage collector of the JVM.

The init() method

- The init method is called only once.
- It is called when the servlet is first created, and not called again for each user request.
- The servlet is normally created when a user first invokes a URL corresponding to the servlet
- When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to **doGet** or **doPost** as appropriate.
- The **init**() method simply creates or loads some data that will be used throughout the life of the servlet.

The service() method :

- The service() method is the main method to perform the actual task.
- The servlet container (i.e. web server) calls the service() method to handle requests coming from the client and to write the formatted response back to the client.
- Each time the server receives a request for a servlet, the server spawns a new thread and calls service.
- The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls **doGet, doPost, doPut, doDelete**, etc. methods as appropriate

The doGet() Method

- A request results from a normal request for a URL or from an HTML form that has no METHOD specified handled by **doGet**() method.

Example:

```
public void doGet(HttpServletRequest request,     HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet code
}
```

The doPost() Method

- A request results from an HTML form that specifically lists POST as the METHOD handled by **doPost**() method.

```
public void doPost(HttpServletRequest request,          HttpServletResponse
response)
  throws ServletException, IOException {
   // Servlet code
}
```

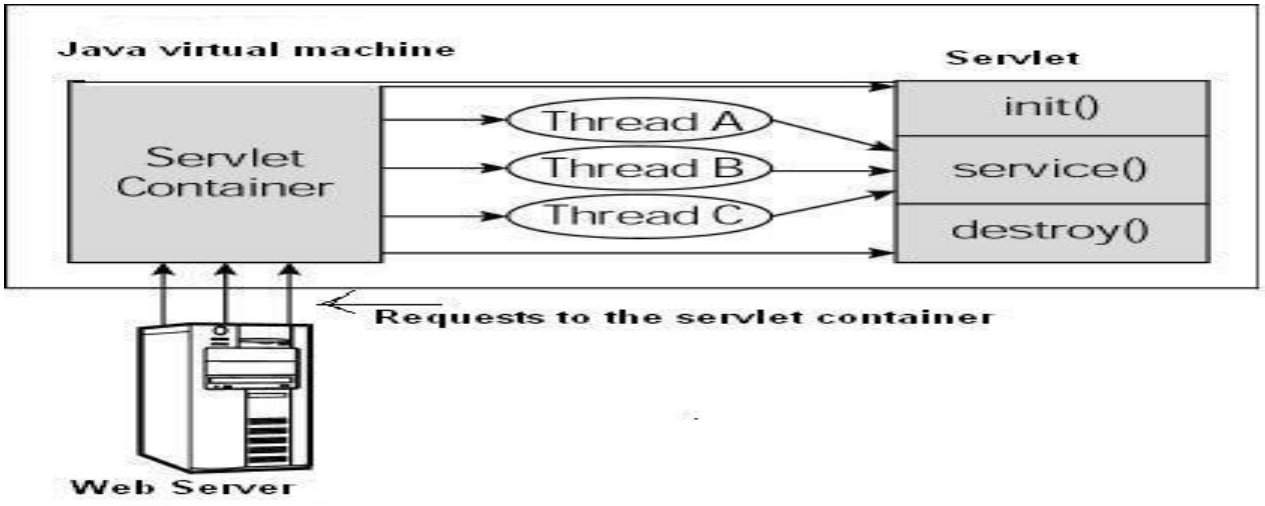
The destroy() method :

- The destroy() method is called only once at the end of the life cycle of a servlet.
- It gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.
- After the destroy() method is called, the servlet object is marked for garbage collection.
- method definition looks like this:

```
public void destroy() {
  // Finalization code...
  }
```

Architecture Diagram:

- First the HTTP requests coming to the server are delegated to the **servlet container.**
- The servlet container loads the servlet before invoking the **service**() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.



Servlet APIs

- Every servlet must implement **jakarta**.servlet.Servlet interface
- Most servlets implement the interface by extending one of these classes
  - ✓ **jakarta.servlet.GenericServlet**
  - ✓ **jakarta.servlet.http.HttpServlet**

Interface jakarta.servlet.Servlet

- The Servlet interface defines methods
  - to initialize a servlet
  - to receive and respond to client requests
  - to destroy a servlet and its resources
  - to get any startup information

Life Cycle Methods

- Developers need to directly implement this interface only if their servlets cannot inherit from **GenericServlet** or **HttpServlet**.

GenericServlet - Methods

- **void init**(ServletConfig config)
  - Initializes the servlet.
- void **service**(ServletRequest req, ServletResponse res)
  - Carries out a single request from the client.
- **void destroy**()
  - Cleans up whatever resources are being held (e.g., memory, file handles, threads) and makes sure that any persistent state is synchronized with the servlet's current in-memory state.
- ServletConfig **getServletConfig**()
  - Returns a servlet config object, which contains any initialization parameters and startup configuration for this servlet.
- String **getServletInfo**()
  - Returns a string containing information about the servlet, such as its author, version, and copyright.

HttpServlet - Methods

- void doGet (HttpServletRequest request,
  HttpServletResponse response)
  – handles GET requests
- void doPost (HttpServletRequest request,
  HttpServletResponse response)
  – handles POST requests
- void doPut (HttpServletRequest request,
  HttpServletResponse response)
  – handles PUT requests
- void doDelete (HttpServletRequest request,
  HttpServletResponse response)
  – handles DELETE requests

Servlet Request Objects

- Provides client request information to a servlet.
- The servlet container creates a servlet request object and passes it as an argument to the servlet's service method.
- The **ServletRequest** interface define methods to retrieve data sent as client request:
  - –parameter name and values
  - – attributes
  - – input stream
- **HTTPServletRequest** extends the **ServletRequest** interface to provide request information for HTTP servlets

HttpServletRequest - Methods

- **Enumeration getParameterNames() :** an Enumeration of String objects, each String containing the name of a request parameter; or an empty Enumeration if the request has no parameters
- **java.lang.String[] getParameterValues (java.lang.String name):** Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist.
- **java.lang.String getParameter (java.lang.String name):** Returns the value of a request parameter as a String, or null if the parameter does not exist.

HttpServletRequest - Methods

- Cookie[] **getCookies**(): Returns an array containing all of the Cookie objects the client sent with this request.
- java.lang.String **getMethod**() : Returns the name of the HTTP method with which the request was made, for example, GET, POST, or PUT.
- java.lang.String **getQueryString**() :Returns the query string that is contained in the request URL after the path.
- HttpSession **getSession**()

Returns the current session associated with this request, or if the request does not have a session, creates one.

Servlet Response Objects

- Defines an object to assist a servlet in sending a response to the client.
- The servlet container creates a **ServletResponse** object and passes it as an argument to the servlet's service method.
- Methods
  - **getWriter():** Returns a PrintWriter object that can send character text to the client
  - **setContentType (String type)** :Sets the content type of the response being sent to the client.

Steps to create a servlet

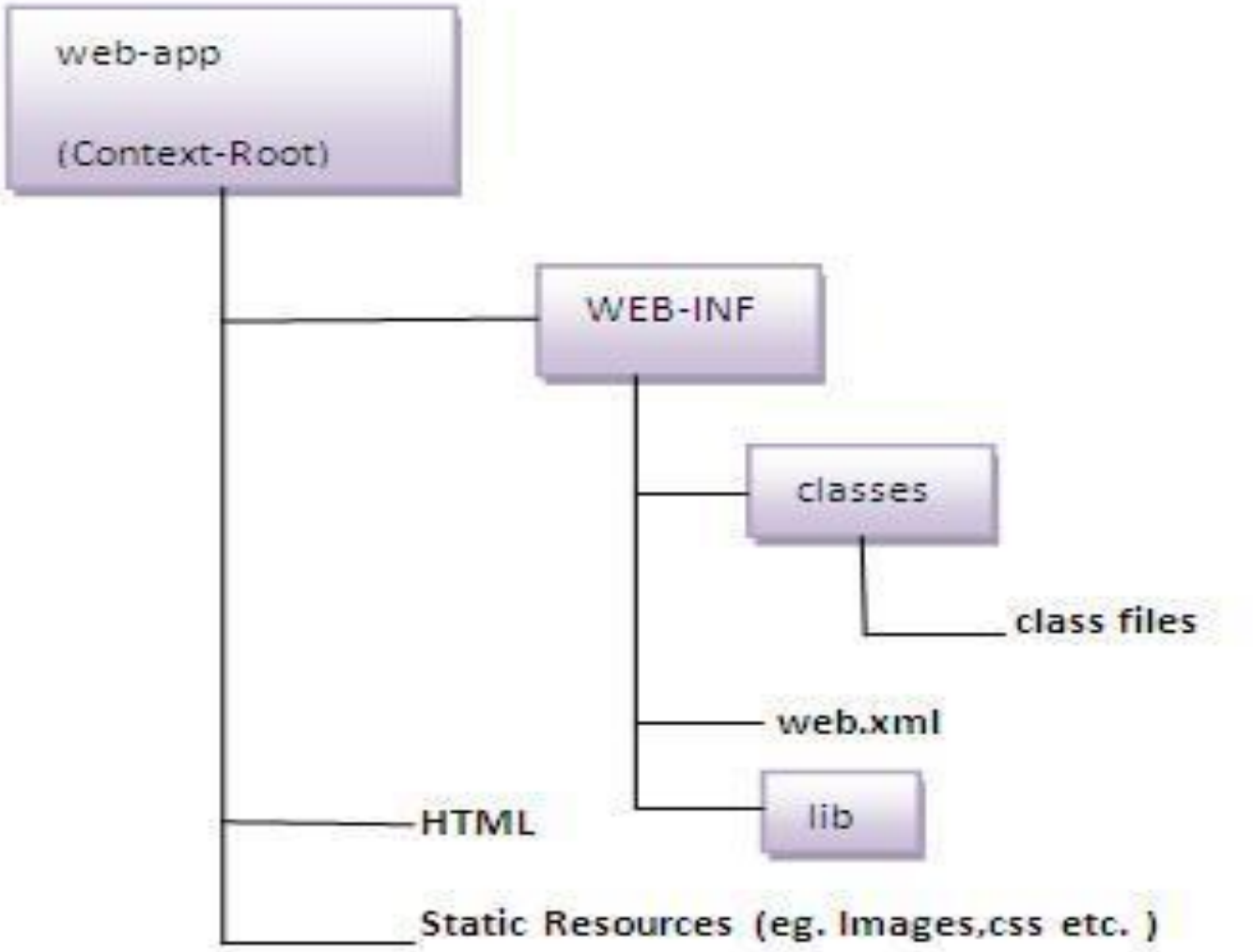
The servlet can be created by three ways

- ❖By implementing **Servlet** interface,
- ❖By inheriting **GenericServlet** class, (or)
- ❖By inheriting **HttpServlet** class

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

1) Create a directory structure
2) Create a Servlet
3) Compile the Servlet
4) Create a deployment descriptor
5) Start the server and deploy the project
6) Access the servlet

1. Create a directory structure



2.Create a Servlet

```
// Extend HttpServlet class
public class HelloWorld extends HttpServlet {
  private String message;
  public void init() throws ServletException
  {
     // Do required initialization
     message = "Hello World";
  }
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
     // Set response content type
     response.setContentType("text/html");
     // Actual logic goes here.
     PrintWriter out = response.getWriter();
     out.println("<h1>" + message + "</h1>");
  }
   public void destroy()  {       // do nothing.  } }
```

3. Compiling a Servlet

- Let us put above code **HelloWorld.java** file and put this file in C:\ServletDevel (Windows) or /usr/ServletDevel (Unix) then you would need to add these directories as well in **CLASSPATH**.
- Assuming your environment is setup properly, go in **ServletDevel** directory and compile **HelloWorld.java** as follows:

```
$> javac HelloWorld.java
```

- If the servlet depends on any other libraries, you have to include those JAR files on your **CLASSPATH** as well. But you should Include servlet-api.jar JAR file

4. Servlet Deployment

- By default, a servlet application is located at the path <Tomcat-installation-directory> / webapps /ROOT and the class file would reside in **<Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes.**
- If you have a fully qualified class name of com.myorg.MyServlet, then this servlet class must be located in **WEB-INF/classes/com/ myorg/ MyServlet.class.**
- Create **web.xml** file located in <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/ as follow

Servlet Deployment cont..

```
<web-app>
<servlet>
   <servlet-name>HelloWorld</servlet-name>
   <servlet-class>HelloWorld</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>HelloWorld</servlet-name>
   <url-pattern>/HelloWorld</url-pattern>
</servlet-mapping>
</web-app>
```

Accessing servlet

- finally type **http://localhost:8080/HelloWorld** in browser's address box.
- If everything goes fine, you would get following result:

Reading Form Data using Servlet

- Servlets handles form data, parsing automatically using the following methods depending on the situation:
  - **•getParameter():** You call **request.getParameter()** method to get the value of a form parameter.
  - **•getParameterValues():** Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
  - **•getParameterNames():** Call this method if you want a complete list of all parameters in the current request.
- POST Method Example Using Form

```
<form action="customer " method="POST">
First Name: <input type="text" name="first_name"><br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
```

Cont..

```
@WebServlet(name = "demoServlet ", value = "/customer ")
public class DemoServlet extends HttpServlet {
@Override
  public void doPost(HttpServletRequest req, HttpServletResponse res)
          throws ServletException, IOException {
       res.setContentType("text/html”)  ;
       PrintWriter pw = res.getWriter();//get the stream to write the data
        String fname=req.getParameter("name1");
        String lname=req.getParameter("name2");
        pw.println("<!DOCTYPE html > ");
       pw.println("<html><body> ");
        pw.println(fname);
        pw.println("<h1style='color:green'>Welcome "+ fname +lname+" my  servlet
            </h1>");
      pw.println("</body></html>");
       pw.close
   }
}
```

Reading All Form Parameters:

- Use **getParameterNames**() method of **HttpServletRequest** to read all the available form parameters.
- This method returns an **Enumeration** that contains the parameter names in an unspecified order.
- Once we have an **Enumeration**, we can loop down the Enumeration using **hasMoreElements**() method to determine when to stop and using **nextElement**() method to get each parameter name.
- Example: Reading All Form Parameters

```
<form action="customer" method="POST" target="_blank">
        <input type="checkbox" name="maths" checked="checked" /> Maths
        <input type="checkbox" name="physics"  /> Physics
        <input type="checkbox" name="chemistry" checked="checked" /> Chemistry
        <input type="submit" value="Select Subject" />
</form>
```

Example cont..

```
PrintWriter out = response.getWriter();
String title = "Reading All Form Parameters ";
 String docType ="<!doctype html ";
out.println(docType +"<html>\n" +
          "<head><title>" + title +
    "</title></head>\n" +
    "<body bgcolor="#f0f0f0">\n"
    +"<h1 align="center">" + title +
    "</h1>\n"
 "<table width="100%" border="1" align="center">\n" +    "<tr bgcolor="#949494">\n" +
"<th>Param Name</th><th>Param Value(s)</th>\n"+   "</tr>\n")}
```

```
Enumeration paramNames = request.getParameterNames();
while(paramNames.hasMoreElements()) {
  String paramName = (String)paramNames.nextElement();
out.print("<tr><td>" + paramName + "</td>\n<td>");
String[] paramValues = request.getParameterValues(paramName);
// Read single valued data
  if (paramValues.length == 1) {
   String paramValue = paramValues[0];
  if (paramValue.length() == 0)
 out.println("<i>No Value</i>");
  else
  out.println(paramValue);
   } else {
    // Read multiple valued data
out.println("<ul>");
  for(int i=0; i < paramValues.length; i++) {
out.println("<li>" + paramValues[i]);
      }   out.println("</ul>");
    }
```

Servlets - Session Tracking

- HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

The HttpSession Object:

- Servlet provides **HttpSession** Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.
- The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user.
- You would get **HttpSession** object by calling the public method **getSession**() of **HttpServletRequest**, as below:
  - **HttpSession session = request.getSession();**

Cont..

❖ Methods available through **HttpSession** object

1) **public Object getAttribute(String name) :**returns the object bound with the specified name in this session, or null if no object is bound.

2) **public Enumeration getAttributeNames() :**This method returns an Enumeration of String objects containing the names of all the objects bound to this session.

3) **public String getId() :** returns a string containing the unique identifier assigned to this session.

4) **public void invalidate():** invalidates the session and unbinds any objects bound to it.

5) **public boolean isNew() :** returns true if the client does not yet know about the session or if the client chooses not to join the session.

6) **public void removeAttribute(String name):** removes the object bound with the specified name from this session.

7) **public void setAttribute(String name, Object value):** binds an object to this session, using the name specified

example

```
String title = "Welcome Back to my website";
    Integer visitCount = new Integer(0);
    String visitCountKey = new String("visitCount");
    String userIDKey = new String("userID");
    String userID = new String("ABCD");
    // Check if this is new comer on your web page.
    if (session.isNew()){
      title = "Welcome to my website";
      session.setAttribute(userIDKey, userID);
    } else {
      visitCount = (Integer)session.getAttribute(visitCountKey);
      visitCount = visitCount + 1;
      userID = (String)session.getAttribute(userIDKey);
    }
    session.setAttribute(visitCountKey,  visitCount);
```

Servlets - Page Redirection

- The simplest way of redirecting a request to another page is using method **sendRedirect**()of response object.
- The signature of this method
- public void HttpServletResponse.sendRedirect(String location)throws IOException
- You can also use **setStatus**() and **setHeader**() methods together to achieve the same. i.e

```
String site = "http://www.newpage.com";
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site);
```

JSP(Java Server Page)

- A technology used to create dynamic web pages using Java.
- It allows developers to embed Java code directly into HTML pages using special JSP tags.
- JSP is part of the Java EE (Jakarta EE) standard and is often used with Servlets to create web applications.
- Java code can be inserted into HTML through JSP tags, enabling dynamic content generation.
- Supports custom tags and standard tag libraries (JSTL - JavaServer Pages Standard Tag Library) to simplify common tasks like iteration, conditional statements, and formatting.
- JSP provides a number of pre-defined objects like request, response, session, application, out, etc., for easier access to various parts of the request/response cycle.

JSP Syntax

- **Directives:** Provide global information about the JSP page, such as page encoding or imports.
  - <%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
- **Scriptlets:** Java code embedded directly in the JSP. The code between <% %> tags is executed on the server.

```
<%
    String message = "Hello, JSP!";
    out.println(message);
%>
```

- **Expressions:** Used to output Java expressions (variables, method calls) directly to the client.

```
<%= "Hello, World!" %>
```

JSP Syntax cont..

- **Declarations**: Used to declare variables or methods to be used in the JSP page.

```
<%! int count = 0; %>
```

- **HTML Tags**: JSP allows embedding HTML content alongside Java code.

```
<html>  <body>
    <h1>Welcome to JSP</h1>
</body></html>
```

- **JSP Tags (JSTL):** The JavaServer Pages Standard Tag Library (JSTL) includes tags for common tasks like loops, conditionals, and formatting.

```
<c:forEach var="item" items="${itemList}">
    <p>${item}</p>
</c:forEach>
```

JSP Implicit Objects

- JSP provides several implicit objects that help interact with the request, response, session, and more:
- **request**: Represents the HTTP request and provides methods to access request parameters, headers, and attributes.
- **response**: Represents the HTTP response and allows you to send data back to the client.out: Used to send output to the response stream.session: Provides access to the HTTP session, useful for maintaining user state.
- **application**: Represents the ServletContext, providing access to application-level resources and attributes.
- **config**: Represents the servlet configuration.
- **pageContext**: Used to access page-level attributes and methods.
- **exception**: Represents any exception thrown during the request processing.

JSP

```
@WebServlet("/showData")
public class DataServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
        List<String> data = dataService.getData();
        request.setAttribute("data", data);
            RequestDispatcher dispatcher =
            request.getRequestDispatcher("data.jsp");
        dispatcher.forward(request, response);
    }
}
```

JSP cont..

Data.jsp

```
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<html>
    <body>
        <h1>Data List</h1>
        <ul>
            <c:forEach var="item" items="${data}">
                <li>${item}</li>
            </c:forEach>
        </ul>
    </body>
</html>
```

Hibernate

- Hibernate ORM is a powerful, open-source Object-Relational Mapping (ORM) framework for Java.
- It simplifies the development of Java applications by providing an abstraction layer over the database,
- allowing developers to interact with databases using Java objects rather than SQL queries.
- Hibernate implements the Java Persistence API (JPA) specification, making it a widely-used choice for data persistence in Java-based applications
- Supports multiple database dialects and simplifies database migration.
- Integrates with JTA, JDBC, or container-managed transactions.

Hibernate Annotations

@Entity: Marks a class as a persistent entity.

@Table: Specifies the table name.

@Id: Indicates the primary key.

@GeneratedValue: Specifies the strategy for primary key generation.

@Column: Maps a field to a database column.

@OneToOne, @OneToMany, @ManyToOne, @ManyToMany: Defines relationships between entities.

@Transient: Ignores a field for persistence

Hibernate Example

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Column;
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "name")
    private String name;
    @Column(name = "department")
    private String department;
    // Getters and Setters
}
```

Hibernate configuration
hibernate.cfg.xml

```
<hibernate-configuration>
   <session-factory>
      <property
name="hibernate.connection.driver_class">org.h2.Driver</property>
      <property
name="hibernate.connection.url">jdbc:h2:mem:testdb</property>
      <property
name="hibernate.connection.username">sa</property>
      <property name="hibernate.connection.password"></property>
      <property
name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>
      <property name="hibernate.hbm2ddl.auto">update</property>
      <mapping class="com.example.Employee"/>
   </session-factory>
</hibernate-configuration>
```

Hibernate Example

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class MainApp {
   public static void main(String[] args) {
      SessionFactory factory = new  Configuration().configure ("hibernate.cfg.xml")
            .buildSessionFactory();
      Session session = factory.openSession();
      try {
         session.beginTransaction();
         Employee emp = new Employee();
         emp.setName("John Doe");
         emp.setDepartment("HR");
         session.save(emp); // Persist the entity
         session.getTransaction().commit();
         System.out.println("Employee saved!");
      } finally {
         session.close();
         factory.close();
      }
   }
}
```

RESTful Web API

- A RESTful Web API (Application Programming Interface) is an architectural style and approach to designing networked applications
- **REST** stands for **representational state transfer** and was created by computer scientist Roy Fielding.
- leverages the principles of REST to create web services that interact with clients over the HTTP protocol.
- RESTful APIs are used to facilitate communication between different software systems in a stateless, scalable, and flexible manner.
- REST API is a way of accessing web services in a simple and flexible way without having any processing.

Cont..

- REST technology is generally preferred to the more robust Simple Object Access Protocol (SOAP) technology because REST uses less bandwidth, simple and flexible making it more suitable for internet usage.
- It's used to fetch or give some information from a web service.
- All communication done via REST API uses only HTTP/Https request
- Stateless: Each request from a client to the server must contain all the information needed to understand and process the request
- The client and server operate independently, allowing them to evolve separately.

How REST works

- A request is sent from client to server in the form of a web URL as HTTP GET or POST or PUT or DELETE request.
- After that, a response comes back from the server in the form of a resource which can be anything like **HTML, XML, Image, or JSON**.
- But now **JSON** is the most popular format being used in Web Services.
- In HTTP there are five methods that are commonly used in a REST-based Architecture i.e., POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations respectively.

JSON Data Format

- JSON (JavaScript Object Notation) is a lightweight, text-based data format used for data interchange.
- It is easy for humans to read and write and easy for machines to parse and generate.
- JSON is widely used in web applications for data exchange between clients and servers.
- Key-value pairs for data representation.
- It is supported by almost all programming languages.
- Represents objects, arrays, numbers, strings, booleans, and null.
- Objects: Enclosed in curly braces {}, contain key-value pairs.
- Arrays: Enclosed in square brackets [], hold a list of values.
- Key-Value Pairs: Keys are strings, and values can be any valid JSON type.
- Keys must always be strings enclosed in double quotes (").

Cont..

- Strings must be in double quotes ("text"), not single quotes ('text’).
- Numbers can be integers or decimals but cannot have leading zeros.
- JSON does not support comments.
- Represents an empty or missing value.
- Example :

```
{
  "id": 101,
  "name": "Alice Johnson",
  "address": {
    "street": "123 Elm St",
    "city": "Springfield",
    "postalCode": "62704"
  },
  "isActive": true
}
```

Spring Boot Framework

- A powerful, open-source framework for building enterprise-level Java applications.
- It provides a comprehensive programming and configuration model, focusing on modern design patterns and simplifying Java development through dependency injection, aspect-oriented programming, and more.
- Spring is widely used in Java-based projects, offering solutions for various layers, including web, data access, messaging, and enterprise services.
- It creates stand-alone Spring applications that can be started using Java -jar.
- The main goal of Spring Boot is to reduce development, unit test, and integration test time.

Advantages of Using Spring Boot

- Reduces development time and increases productivity.
- Eliminates the need for manual configuration.
- Provides an opinionated default setup.
- Easy to test applications.
- Suitable for microservices architecture.
- Community support and extensive documentation.
- Simplifies dependency management (e.g., spring-boot-starter-web for web applications).

Spring Boot Architecture

- **Presentation Layer**: Handles user interfaces and requests **(e.g., controllers).**
- **Business Layer:** Handles computations, validations, and decision-making processes. i.e. **Services (@Service)**
- **Persistence Layer:** Manages interactions with the database and data repositories.
  - Repositories **(@Repository):**Provides CRUD operations using JPA, Hibernate, or other frameworks.
  - Maps database tables to Java objects.
- **Integration Layer:** Handles communication with external systems, APIs, or services
  - **RestTemplate**/WebClient: For consuming external **REST APIs.**
  - **Message Brokers**: For asynchronous communication (e.g., **RabbitMQ**, **Kafka**).

Spring Boot Starters

- Pre-defined dependency descriptors to simplify Maven/Gradle configurations.
- Examples:
  - **spring-boot-starter-web:** For building web applications (REST APIs).
  - **spring-boot-starter-data-jpa:** For database integration using JPA.
  - **spring-boot-starter-security:** For adding security features.
  - **spring-boot-starter-actuator:** spring-boot-starter-actuator
- Steps to setup the project.
  - Navigate to https://start.spring.io.
  - Choose either Gradle or Maven and the language you want to use
  - Click Dependencies and select **Spring Web**.
  - Click Generate.

Spring Boot Annotations

❑ Core Annotations

- **@SpringBootApplication**:Entry point for Spring Boot applications. Combines **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**.
- **@Configuration** Indicates that the class contains Spring configuration.
- **@Bean:**Marks a method as a bean producer, which Spring will manage in the application context.

Web Layer Annotations

- **@RestController:Combines @Controller and @ResponseBody.** Used to create RESTful web services.
- **@Controller**:Marks a class as a Spring MVC controller.
- **@GetMapping, @PostMapping , @PutMapping, @DeleteMapping** : :Maps HTTP GET, POST, PUT and DELETE requests to a specific handler method.
- **@RequestMapping:**General-purpose mapping annotation for requests.
- **@RequestParam:**Binds query parameters or form data to method parameters.
- **@PathVariable:**Binds URI template variables to method parameters.
- **@RequestBody**: Binds the body of a request to a method parameter (e.g., JSON to a Java object).
- **@ResponseBody:**Indicates the return value of a method should be serialized and written directly to the response.
- **@CrossOrigin**: Enables cross-origin resource sharing (CORS).

Data Access Layer Annotations

@Entity: Marks a class as a JPA entity (maps to a database table).

@Table: Specifies the database table name for an entity.

@Id: Identifies the primary key of an entity.

@GeneratedValue: Specifies how the primary key is generated (e.g., AUTO, IDENTITY).

@Column: Configures a column in a database table.

@Repository: Marks a class as a repository for data access.

@Query: Used to define custom JPQL or SQL queries in a repository.

Dependency Injection (DI) Annotations

- **@Autowired**: Automatically injects dependencies by type.
- **@Qualifier:** Used with @Autowired to specify a particular bean when multiple beans of the same type exist.
- **@Primary:**Specifies a default bean when multiple candidates are available.
- **@Componen:** Marks a class as a Spring-managed component.
- **@Service:** A specialization of @Component for service-layer classes.
- **@Repository:** A specialization of @Component for data access classes.

Validation Annotations

- **@Valid:** Triggers validation for the annotated object.
- **@NotNull:** Ensures the annotated field is not null.
- **@Size:**Specifies size constraints for a string, collection, map, or array.
- **@Min:**Specifies the minimum value for a numeric field.
- **@Max:**Specifies the maximum value for a numeric field.

Security Annotations

- @**EnableWebSecurity:**Enables Spring Security's web security support.
- **@PreAuthorize:**Allows method-level security using expressions.
- **@Secured**:Specifies roles required to execute a method.

Example : Spring Boot Entity

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
     public Long getId() {        return id;    }
    public void setId(Long id) {    this.id = id;    }
    public String getFirstName() {  return firstName;    }
    public void setFirstName(String firstName) {  this.firstName = firstName;    }
    public String getLastName() {  return lastName;    }
    public void setLastName(String lastName) {    this.lastName = lastName;    }
    public String getEmail() {  return email;    }
    public void setEmail(String email) {    this.email = email;    }
}
```

Example : Repository

```
Import org.springframework.data.jpa.repository.JpaRepository;
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

Service :Implement the CustomerService

```
@Service
public class CustomerService {
    private final CustomerRepository customerRepository;
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public List<Customer> getAllCustomers() {
        return customerRepository.findAll();
    }
    public Optional<Customer> getCustomerById(Long id) {
        return customerRepository.findById(id);
    }
}
```

Example : Service cont..

```
public Customer addCustomer(Customer customer) {
    return customerRepository.save(customer);
}
public Customer updateCustomer(Long id, Customer updatedCustomer) {
    return customerRepository.findById(id)
            .map(customer -> {
                customer.setFirstName(updatedCustomer.getFirstName());
                customer.setLastName(updatedCustomer.getLastName());
                customer.setEmail(updatedCustomer.getEmail());
                return customerRepository.save(customer);
            })
            .orElseThrow(() -> new RuntimeException("Customer not found"));
}

public void deleteCustomer(Long id) {
    customerRepository.deleteById(id);
}
```

Example : Controller

```
@RestController
@RequestMapping("/api/customers")
public class CustomerController {
    private final CustomerService customerService;
    public CustomerController(CustomerService customerService) {
        this.customerService = customerService;
    }
    @GetMapping
    public List<Customer> getAllCustomers() {
        return customerService.getAllCustomers();
    }
 @GetMapping("/{id}")
public ResponseEntity<Customer> getCustomerById(@PathVariable Long id) {
        return customerService.getCustomerById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }
```

Configure application.properties

- Set the MySQL database connection details in your **application.properties as bellow**

```
# Datasource Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/ur_database_name
spring.datasource.username=your_username
spring.datasource.password=your_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Example : Controller cont…

```
@PostMapping
   public Customer addCustomer(@RequestBody Customer customer) {
      return customerService.addCustomer(customer);
   }
   @PutMapping("/{id}")
   public ResponseEntity<Customer> updateCustomer(@PathVariable Long id,
@RequestBody Customer updatedCustomer) {
      try {
         return ResponseEntity.ok(customerService.updateCustomer(id, updatedCustomer));
      } catch (RuntimeException e) {      return ResponseEntity.notFound().build();
      }    }
   @DeleteMapping("/{id}")
   public ResponseEntity<Void> deleteCustomer(@PathVariable Long id) {
      customerService.deleteCustomer(id);
      return ResponseEntity.noContent().build();
   }}
```

Running the Application

- Run the Spring Boot application.
- Test the endpoints using **Postman, Swagger**, or any API testing tool:
- use the **Springdoc OpenAPI** library, which is a popular choice for integrating Swagger UI with Spring Boot
- Steps to configure swagger:
  - **Add Maven Dependency**

```
<dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-ui</artifactId>
         <version>1.7.0</version>
</dependency>
```

  - Access Swagger UI http://localhost:8080/swagger-ui.html
    - GET /api/customers - List all customers.
    - GET /api/customers/{id} - Get a specific customer by ID.
    - POST /api/customers - Add a new customer.
    - PUT /api/customers/{id} - Update an existing customer.
    - DELETE /api/customers/{id} - Delete a customer.

Thank you!!!

The End