Module-3 (Web Technologies In java)

Name:	**Prajapati Raj ManojKumar**
Institute:	**Tops Technologies**
Subject:	**Java**
Topic:	**Web Technologies In Java**

Module-3 (Web Technologies In java)

Indexes

Module-3 (Web Technologies In java)

HTML Tags: Anchor, Form, Table, Image, List Tags, Paragraph, Break, Label.

HTML (HyperText Markup Language) is the standard language used to create and design web pages. It comprises a set of elements, called tags, that define the structure and content of a web page.

Introduction to HTML and its Structure

HTML uses a tag-based structure to create and format the content on a web page. Tags are enclosed in angle brackets (< >) and usually come in pairs: an opening tag and a closing tag with a /. For example, <p> and </p>.

Structure:

Ex:
```
<!DOCTYPE html>
 <html>
<head>
        <title>Page Title</title>
</head>
<body>
        <h1>My First Heading</h1>
        <p>My first paragraph.</p>
</body>
</html>
```

Example Explained:

- The <!DOCTYPE html> declaration defines that this document is an HTML5 document
- The <html> element is the root element of an HTML page

- The <head> element contains meta information about the HTML page
- The <title> element specifies a title for the HTML page (which is shown in the browser's title bar or in the page's tab)
- The <body> element defines the document's body, and is a container for all the visible contents, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.
- The <h1> element defines a large heading
- The <p> element defines a paragraph

What is an HTML Element?

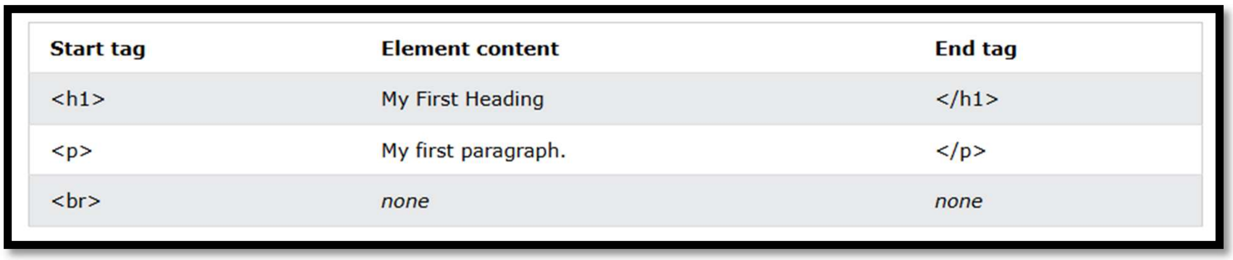
➔ An HTML element is defined by a start tag, some content, and an end tag:

<tagname> Content goes here... </tagname>

➔ The HTML **element** is everything from the start tag to the end tag:

<h1>My First Heading</h1>

<p>My first paragraph.</p>f



Start tag	Element content	End tag
<h1>	My First Heading	</h1>
<p>	My first paragraph.	</p>
<br>	*none*	*none*

Explanation of Key Tags:-

- <a>**: Anchor Tag for Hyperlinks** The anchor tag is used to create hyperlinks, which can navigate to another webpage, file, location within the same page, email addresses, or any URL.

Ex: <a href="https://www.example.com">Visit Example</a>

- <form>**: Form Tag for User Input** The form tag is used to create an HTML form for user input. It can contain various input elements like text fields, checkboxes, radio buttons, and submit buttons.

  Ex:
  ```
  <form action="/submit-form" method="post">
          <label for="name">Name:</label>
          <input type="text" id="name" name="name">
          <input type="submit" value="Submit">
      </form>
  ```

- <table>**: Table Tag for Data Representation** The table tag is used to create tables, which organize data into rows and columns.

  Ex:
  ```
  <table>
          <tr>
                  <th>Header 1</th>
                  <th>Header 2</th>
          </tr>
          <tr>
                  <td>Data 1</td>
                  <td>Data 2</td>
          </tr>
  </table>
  ```

- <img>: **Image Tag for Embedding Images** The image tag is used to embed images in a web page.

  Ex: <img src="image.jpg" alt="Description of image">

- **List Tags**
  - <ul>: **Unordered List** Creates a bulleted list.

    Ex:
    ```
    <ul>
            <li>Item 1</li>
            <li>Item 2</li>
            <li>Item 3</li>
    </ul>
    ```

  - <ol>: **Ordered List** Creates a numbered list.

    Ex:
    ```
    <ol>
            <li>First Item</li>
            <li>Second Item</li>
            <li>Third Item</li>
    </ol>
    ```

  - <li>: **List Item** Defines a list item inside <ul> or <ol>.

    Ex:
    ```
    <ul>
            <li>List item</li>
    </ul>
    ```

- <p>: **Paragraph Tag** The paragraph tag is used to define paragraphs of text.

  Ex: <p>This is a paragraph of text.</p>

- <br>**: Line Break** The break tag is used to insert a line break within text.

  Ex: This is some text<br>that will break to a new line.
- <label>**: Label for Form Inputs** The label tag is used to define labels for input elements in a form.

  Ex: <label for="email">Email:</label>
  <input type="email" id="email" name="email">

==

Module-3 (Web Technologies In java)

CSS: Inline CSS, Internal CSS, External CSS

CSS (Cascading Style Sheets):-

CSS (Cascading Style Sheets) is a style sheet language used to describe the presentation of a document written in HTML or XML. CSS controls the layout, colors, fonts, and overall visual appearance of web pages, allowing for a more attractive and consistent design across a website.

Overview of CSS and Its Importance in Web Design:-

CSS is essential in web design as it separates the content (HTML) from the presentation (CSS). This separation allows for:

- Easier maintenance and updating of styles across multiple pages.
- Improved accessibility and usability for users.
- Faster page loading times by reducing the amount of code in HTML files.
- Enhanced design flexibility and control over the look and feel of web pages.

Types of CSS:-

1. **Inline CSS: Directly in HTML Elements** Inline CSS is used to apply a unique style to a single HTML element by using the style attribute directly within the element.

   Ex:
   ```
   <p style="color: blue; font-size: 16px;">
           This is a paragraph with inline CSS.
       </p>
   ```

2. **Internal CSS: Inside a** <style> **Tag in the Head Section** Internal CSS is used to define styles for a single HTML document. The CSS rules are placed within a <style> tag inside the <head> section of the HTML file.

   Ex:
```
<head>
        <style>
                body {
                        background-color: lightgray;
                }
                h1 {
                        color: blue;
                        text-align: center;
                }
        </style>
</head>
<body>
        <h1>Welcome to My Website</h1>
</body>
```

3. **External CSS: Linked to an External File** External CSS is used to define styles for multiple HTML documents. The CSS rules are written in a separate .css file, which is then linked to the HTML files using the <link> tag.

Ex:
```
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <h1>Welcome to My Website</h1>
</body>
```

External CSS File (styles.css):

Ex:
```
body {
        background-color: lightgray;
}
h1 {
      color: blue;
      text-align:
}
```

CSS: Margin and Padding

CSS (Cascading Style Sheets) provides two key properties for controlling the layout and spacing of elements on a web page: margin and padding.

Definition and Difference Between Margin and Padding

- **Margin**: The margin property creates space outside of an element. It defines the outer space around the element, pushing other elements away from it. Margins are completely transparent and do not affect the element's background color or border.

  Ex:
  ```
  div
  {
      margin: 20px;
  }
  ```

- **Padding**: The padding property creates space inside of an element. It defines the inner space between the element's content and its border. Padding adds to the element's background color and is often used to create breathing room around the content.

  Ex:
  ```
  div
  {
      padding: 20px;
  }
  ```

How Margins Create Space Outside the Element and Padding Creates Space Inside

- **Margins**:
  - Margins control the space around an element, creating gaps between the element and its neighbours.
  - This property can be specified for each side of the element: top, right, bottom, and left (e.g., margin-top, margin-right, margin-bottom, margin-left).
  - Margins can collapse, meaning adjacent margins of block-level elements can overlap.

Ex: div

```
{
    margin: 10px 15px 20px 25px; /* top right bottom left */
}
```

- **Padding**:
  - Padding controls the space within an element, between its content and its border.
  - Like margins, padding can be specified for each side of the element: top, right, bottom, and left (e.g., padding-top, padding-right, padding-bottom, padding-left).

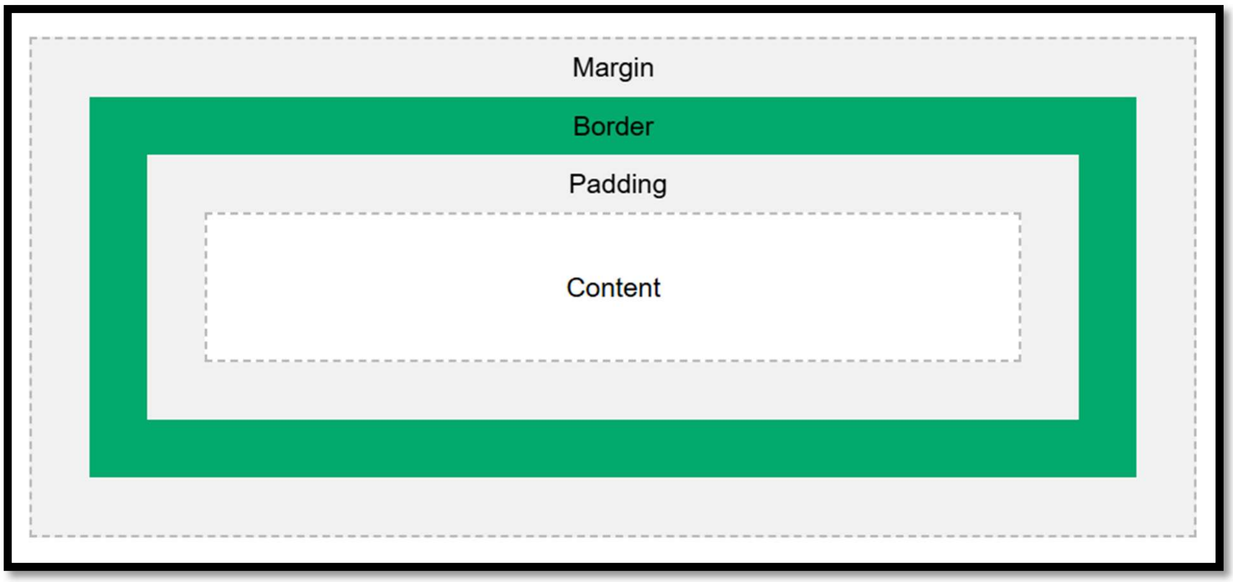
Ex: div

```
{
  padding: 10px 15px 20px 25px; /* top right bottom left */
}
```

Visual Representation:

Ex: [Margin] [Border] [Padding] Content [Padding] [Border] [Margin]

Image:



In summary, while both margin and padding are used to create space, the margin creates space outside the element, affecting its positioning relative to other elements, whereas padding creates space inside the element, affecting the space around its content.

CSS: Pseudo-Class

CSS (Cascading Style Sheets) pseudo-classes are special keyword additions to selectors that specify a special state of the selected elements. They are used to apply styles to elements based on their state or interaction with the user.

Introduction to CSS Pseudo-Classes

Pseudo-classes are used to define the styling of elements in different states. They are usually written with a colon (:) before the keyword. Here are some common pseudo-classes:

- :hover: Applied when the user designates an element (usually by placing the mouse pointer over it).

  Ex:
  ```
  a:hover {
          color: red;
  }
  ```

- :focus: Applied when an element has received focus (e.g., when a user clicks on a form input field).

  Ex:
  ```
  input:focus {
          border: 2px solid blue;
  }
  ```

- :active: Applied when an element is being activated (e.g., when a user is clicking on a link or button).

  Ex:
  ```
  button:active {
          background-color: green;
  }
  ```

Use of Pseudo-Classes to Style Elements Based on Their State

- :hover: This pseudo-class changes the style of an element when the user hovers over it with a mouse. It's commonly used for links and buttons to indicate they are interactive.

  Ex:
  ```
  a:hover {
          text-decoration: underline;
          color: blue;
  }
  ```

- :focus: This pseudo-class applies a style to an element when it gains focus, such as when a user clicks on or tabs into an input field. It helps improve accessibility by providing a visual cue.

  Ex:
  ```
  input:focus {
          outline: none;
          box-shadow: 0 0 5px rgba(81, 203, 238, 1);
          border: 1px solid rgba(81, 203, 238, 1);
          }
  ```

- :active: This pseudo-class changes the style of an element while it is being activated (e.g., during the click action for a button). It provides immediate feedback to the user that their action is being recognized.

  Ex:
  ```
  button:active {
          transform: translateY(2px);
          }
  ```

Additional Pseudo-Classes:

- :visited: Applied to links that have been visited by the user.

  Ex:
  ```
  a:visited {
          color: purple;
      }
  ```

- :first-child: Applied to the first child element of a parent.

  Ex:
  ```
  p:first-child {
          font-weight: bold;
          }
  ```

- :nth-child(n): Applied to the nth child element of a parent, useful for targeting specific elements in a sequence.

  Ex:
  ```
  tr:nth-child(even)
          {
          background-color: #f2f2f2;
          }
  ```

In summary, CSS pseudo-classes enhance user interaction by allowing you to apply styles based on the state of an element, making web pages more dynamic and engaging.

CSS: ID and Class Selectors

CSS (Cascading Style Sheets) provides mechanisms to select and style HTML elements using selectors. Two commonly used selectors are id and class.

Difference Between ID and Class in CSS

- **ID Selector (#):**
  - The id selector is used to specify a style for a single, unique element.
  - Each id value must be unique within an HTML document.
  - The id selector is defined with a (#) symbol followed by the id name.

Ex: Html: <div id="header">This is a header</div>

CSS: #header {

background-color: lightblue;

}

- **Class Selector (.):**
  - The class selector is used to specify a style for a group of elements.
  - Multiple elements can share the same class value, making it reusable.
  - The class selector is defined with a (.) symbol followed by the class name.

Ex: Html: <div class="content">This is a content block</div>

```
        <p class="content">This is another content block</p>
```

CSS:
```
.content {
        font-family: Arial, sans-serif;
        }
```

Usage Scenarios for ID (Unique) and Class (Reusable)

- **ID Selector (Unique)**:
  - The id selector is best used when you need to apply a specific style to a single, unique element.
  - Examples include styling a specific header, a footer, or any element that appears only once on a page.

Ex: Html: `<button id="submitBtn">Submit</button>`

CSS:
```
#submitBtn {
        background-color: green;
        color: white;
        }
```

- **Class Selector (Reusable)**:
  - The class selector is ideal for styling multiple elements with a common appearance.
  - Examples include styling all paragraphs in a section, buttons with the same design, or any group of elements that share the same style.

Ex: Html:
```
<div class="card">Card 1</div>
<div class="card">Card 2</div>
<div class="card">Card 3</div>
```

CSS:
```
.card {
    border: 1px solid #ccc;
    padding: 10px;
    margin: 5px;
    }
```

In summary, use the id selector for unique styling of individual elements and the class selector for reusable styling of multiple elements that share common characteristics.

Module-3 (Web Technologies In java)

Introduction to Client-Server Architecture

Client-Server Architecture is a network design model that divides computing tasks between clients and servers. This architecture underlies many web applications and services, ensuring efficient resource use and centralized control.

Overview of Client-Server Architecture

In a client-server model:

- **Clients**: Devices or programs that request services or resources from servers. Examples include web browsers, mobile apps, and desktop applications.
- **Servers**: Powerful computers or programs that provide services, manage resources, and respond to client requests. Examples include web servers, database servers, and application servers.

The communication between clients and servers occurs over a network, typically using the Internet or an internal network (intranet).

Difference Between Client-Side and Server-Side Processing

- **Client-Side Processing**:
    - Performed on the client's device (e.g., user's computer or smartphone).
    - Involves tasks like rendering web pages, validating user input, and handling user interactions.
    - Uses technologies like HTML, CSS, JavaScript.

Example: Form validation using JavaScript to check if all required fields are filled before submission.

- **Server-Side Processing**:
    - Performed on the server.
    - Involves tasks like database queries, business logic execution, and responding to client requests.
    - Uses technologies like Java, PHP, Python, Ruby.

Example: Authenticating user credentials against a database during login.

Roles of a Client, Server, and Communication Protocols

- **Client**:
    - Initiates requests for resources or services.
    - Displays data received from the server to the user.

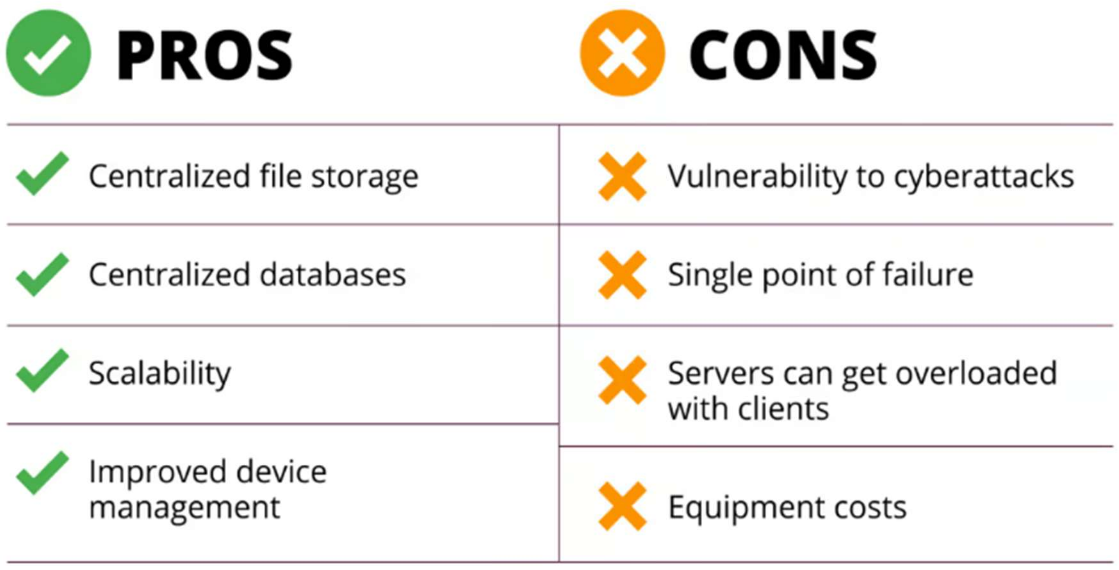
Example: A web browser requesting a webpage from a web server.

- **Server**:
    - Listens for and responds to client requests.
    - Provides requested resources or performs requested services.

Example: A web server delivering HTML files in response to client requests.

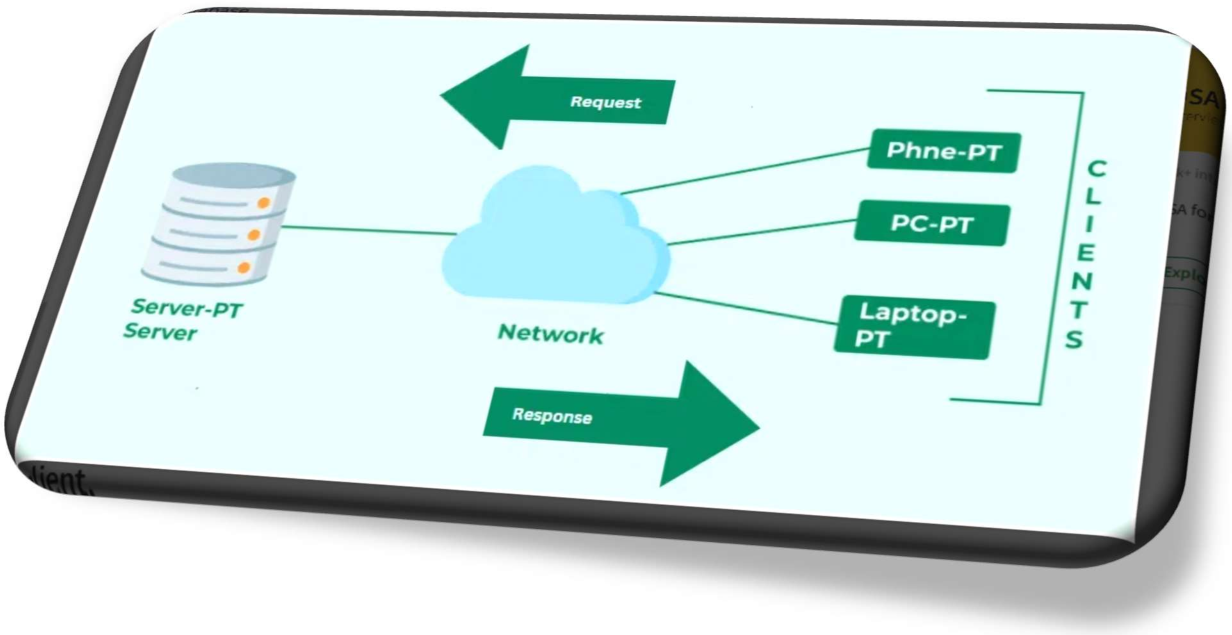
- **Communication Protocols**:
    - Standardize the exchange of data between clients and servers.
    - Common protocols include HTTP/HTTPS for web communication, FTP for file transfer, and SMTP for email.

Example: HTTP (HyperText Transfer Protocol) used by web browsers to communicate with web servers.



✓ PROS	✗ CONS
✓ Centralized file storage	✗ Vulnerability to cyberattacks
✓ Centralized databases	✗ Single point of failure
✓ Scalability	✗ Servers can get overloaded with clients
✓ Improved device management	✗ Equipment costs

Visual Representation of Client-Server Architecture:



Module-3 (Web Technologies In java)

HTTP Protocol Overview with Request and Response Headers

HTTP (HyperText Transfer Protocol) is the foundation of data communication on the World Wide Web. It defines the rules for exchanging information between a client (e.g., web browser) and a server.

Introduction to the HTTP Protocol and Its Role in Web Communication

- **HTTP**: A protocol used for transmitting hypertext documents, such as HTML. It facilitates communication between web browsers (clients) and web servers.
- **Role**: It enables the fetching of resources, such as HTML documents, images, and other content necessary for displaying web pages. HTTP operates as a request-response protocol, where the client sends a request, and the server responds with the requested resource.

Explanation of HTTP Request and Response Headers

- **HTTP Request Headers**: Sent by the client to provide information about the request or about the client itself. Common headers include:
    - **Host**: Specifies the domain name of the server and the TCP port number.

      Ex: - Host: www.example.com
    - **User-Agent**: Contains information about the client software (e.g., browser type, version).

  Ex: - User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)

- **Accept**: Indicates the content types that the client can understand.

  Ex: - Accept: text/html,application/xhtml+xml

- **Content-Type**: Indicates the media type of the resource being sent (for POST requests).

  Ex: - Content-Type: application/json

- **HTTP Response Headers**: Sent by the server to provide information about the response or about the server itself. Common headers include:
  - **Content-Type**: Indicates the media type of the resource being sent.

    Ex: - Content-Type: text/html; charset=UTF-8

  - **Content-Length**: Specifies the size of the response body in bytes.

    Ex: - Content-Length: 1024

  - **Server**: Contains information about the server software.

    Ex: - Server: Apache/2.4.41 (Ubuntu)

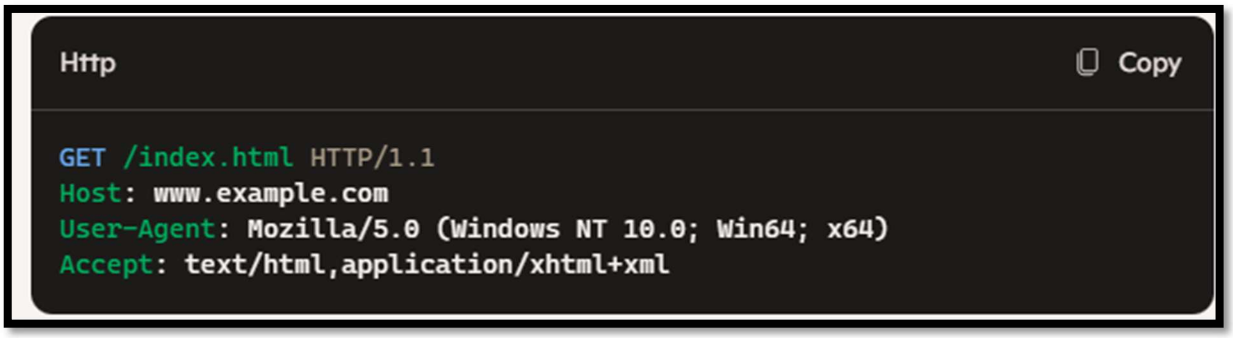
  - **Set-Cookie**: Used to send cookies from the server to the client.

    Ex: -

    Set-Cookie: sessionId=abc123; Path=/; HttpOnly

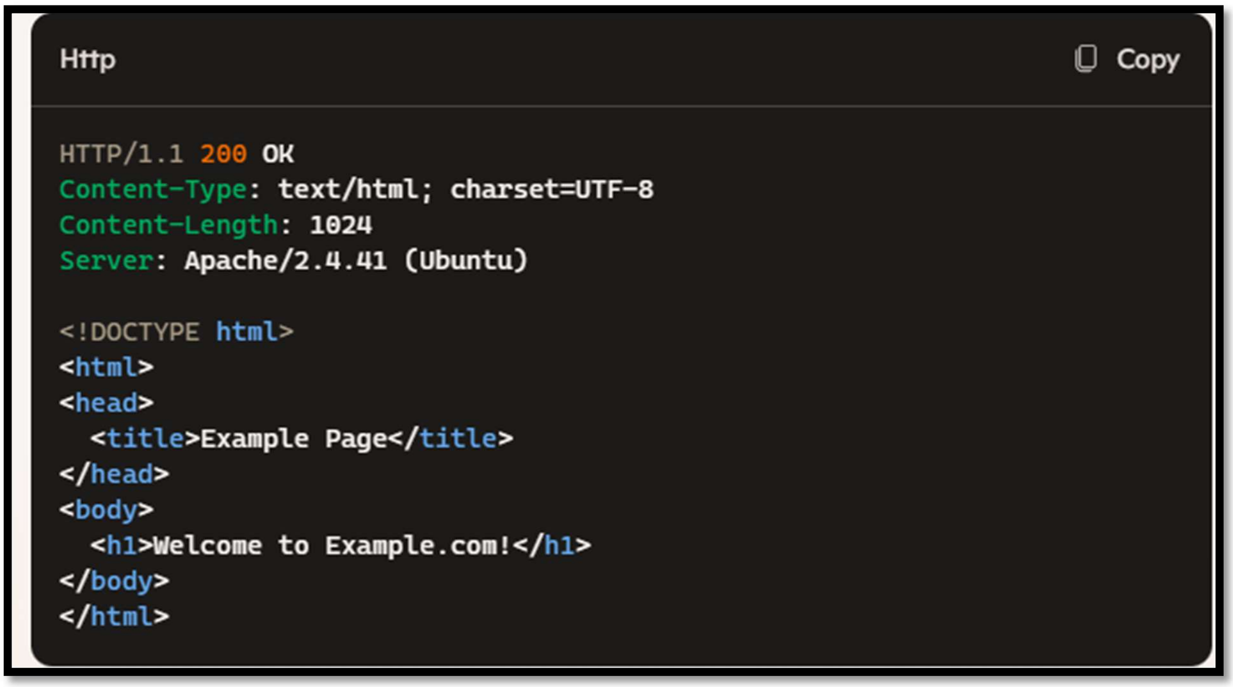
Example of an HTTP Request and Response:

- **HTTP Request:**



```http
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: text/html,application/xhtml+xml
```

- **HTTP Response:**



```http
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 1024
Server: Apache/2.4.41 (Ubuntu)

<!DOCTYPE html>
<html>
<head>
  <title>Example Page</title>
</head>
<body>
  <h1>Welcome to Example.com!</h1>
</body>
</html>
```

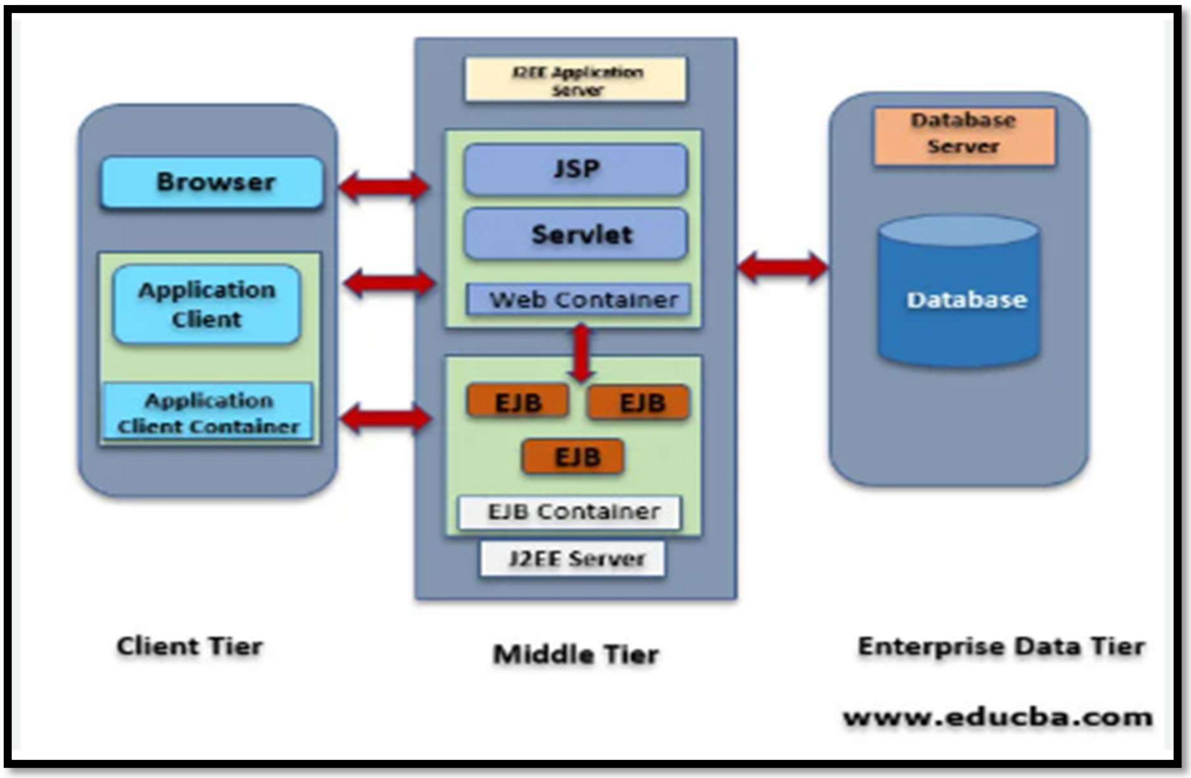
Module-3 (Web Technologies In java)

J2EE Architecture Overview

J2EE (Java 2 Platform, Enterprise Edition) is a platform for building distributed, multi-tier enterprise applications. It extends the standard Java platform (Java SE) with specifications for enterprise features such as web services, component-based development, and robust transaction management.

Introduction to J2EE and Its Multi-Tier Architecture

- **Visual Representation:**



J2EE architecture is designed to support a multi-tiered approach to building enterprise applications. The main tiers in J2EE architecture include:

1. **Client Tier**:
   - Represents the user's interface.

  - Clients can be web browsers, desktop applications, or mobile apps.

2. **Presentation Tier (Web Tier)**:
   - Handles the presentation logic and user interface components.
   - Typically involves web technologies like Servlets and JSP (JavaServer Pages).
3. **Business Logic Tier (Application Tier)**:
   - Contains the business logic and application-specific functionality.
   - Components like Enterprise JavaBeans (EJB) are used to implement business logic.
4. **Data Tier (Enterprise Information System Tier)**:
   - Manages data persistence and database interactions.
   - Includes database servers and data access technologies like JDBC (Java Database Connectivity).

The multi-tier architecture ensures separation of concerns, modularity, and scalability, allowing each tier to be developed, maintained, and scaled independently.

Role of Web Containers, Application Servers, and Database Servers

- **Web Containers**:
  - A web container, also known as a servlet container, is a component of a web server that interacts with Java servlets.

  - It manages the lifecycle of servlets, mapping URLs to servlets, and ensuring secure and efficient processing of client requests.
  - Example: Apache Tomcat.
- **Application Servers**:
  - An application server provides a runtime environment for executing business logic components, such as EJBs.
  - It offers services like transaction management, security, and resource pooling.
  - Example: IBM WebSphere, Oracle WebLogic.
- **Database Servers**:
  - A database server stores and manages data used by the application.
  - It handles data queries, transactions, and connections from application components.
  - Example: MySQL, Oracle Database.

Module-3 (Web Technologies In java)

Web Component Development in Java (CGI Programming)

CGI (Common Gateway Interface) is a standard protocol used to enable web servers to execute external programs, typically scripts, to generate dynamic web content.

Introduction to CGI (Common Gateway Interface)

- **CGI**: A protocol that allows web servers to interact with external programs, enabling the generation of dynamic content.
- **Use**: Commonly used for processing form data, generating web pages on the fly, and interacting with databases.

Process of CGI Programming

1. **Request**: A web client (browser) sends an HTTP request to the web server.
2. **Execution**: The web server executes the CGI script located on the server.
3. **Response**: The CGI script processes the request, generates the required output, and sends it back to the web server.
4. **Delivery**: The web server sends the generated content back to the client as an HTTP response.

Advantages of CGI Programming

- **Platform Independence**: CGI scripts can be written in various programming languages (e.g., Perl, Python, Java).
- **Simplicity**: Easy to implement for small-scale applications.
- **Flexibility**: Can be used to integrate different types of applications with web servers.

Disadvantages of CGI Programming

- **Performance**: Each request spawns a new process, leading to high overhead and slower performance.
- **Scalability**: Not suitable for high-traffic websites due to the performance overhead.
- **Security**: Requires careful handling of user input to prevent security vulnerabilities.

Module-3 (Web Technologies In java)

Servlet Programming: Introduction, Advantages, and Disadvantages

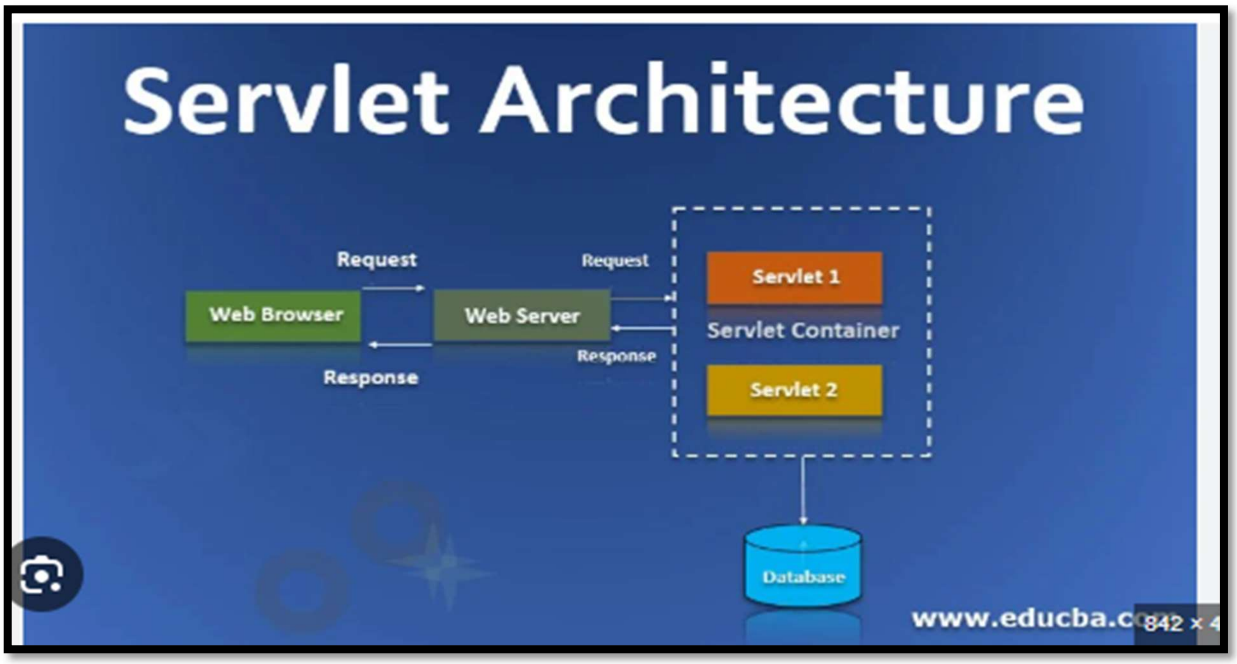
Servlets are Java programs that run on a web server and are used to handle client requests and generate dynamic web content. They are a key component of Java's server-side technology and are part of the Java EE (Enterprise Edition) platform.

Introduction to Servlets and How They Work

- **Servlets**: Java-based web components that extend the capabilities of servers hosting applications accessed via a request-response programming model.
- **How Servlets Work**:
    1. **Client Request**: A client (e.g., web browser) sends an HTTP request to the web server.
    2. **Servlet Invocation**: The web server receives the request and passes it to the appropriate servlet.
    3. **Request Processing**: The servlet processes the request, which may involve interacting with databases or other resources.
    4. **Response Generation**: The servlet generates a dynamic response (e.g., HTML, JSON) and sends it back to the web server.
    5. **Client Response**: The web server sends the response to the client, which displays the content.

Module-3 (Web Technologies In java)

Servlet Architecture:



Servlet Life Cycle:

1. **Initialization**: The servlet is instantiated and initialized by the server.
2. **Service**: The servlet handles client requests via the service() method.
3. **Destruction**: The servlet is removed from memory when no longer needed.

Advantages of Servlets

- **Performance**: Servlets are highly efficient as they are loaded into memory once and reused to handle multiple requests. This reduces the overhead of process creation and improves response time.
- **Platform Independence**: Being written in Java, servlets are platform-independent and can run on any server that supports the Java Servlet API.

- **Scalability**: Servlets can handle a large number of concurrent requests, making them suitable for high-traffic web applications.
- **Integration**: Servlets can easily interact with other Java-based technologies such as JSP (JavaServer Pages) and EJB (Enterprise JavaBeans).
- **Robustness**: Java's strong type-checking, exception handling, and memory management contribute to the robustness of servlets.

Disadvantages of Servlets Compared to Other Web Technologies

- **Complexity**: Servlets can be more complex to develop and manage compared to some other web technologies, such as PHP or ASP.NET.
- **Learning Curve**: Requires knowledge of Java programming, which may have a steeper learning curve for developers new to the language.
- **Server Dependency**: Servlets require a servlet container (e.g., Apache Tomcat) to run, which may add to the deployment and configuration complexity.

In summary, servlets are a powerful and efficient server-side technology that offers numerous advantages for building dynamic web applications. However, they also come with certain complexities and dependencies that developers need to consider.

==

Module-3 (Web Technologies In java)

Servlet Versions, Types of Servlets

History of Servlet Versions

Servlet technology has evolved significantly since its inception, with each version introducing new features and improvements. Here is an overview of the main servlet versions:

1. **Servlet 1.0** (June 1997)
   - The first official release, providing basic servlet functionality.
2. **Servlet 2.0** (December 1997)
   - Introduced major enhancements such as request dispatching.
3. **Servlet 2.1** (November 1998)
   - Added support for session tracking and request forwarding.
4. **Servlet 2.2** (August 1999)
   - Introduced the web application deployment descriptor (web.xml).
5. **Servlet 2.3** (August 2001)
   - Added filters and listeners.
6. **Servlet 2.4** (November 2003)
   - Enhanced deployment descriptor and support for web applications.
7. **Servlet 2.5** (September 2005)
   - Introduced annotations for servlets.

8. **Servlet 3.0** (December 2009)
   - Significant update with asynchronous support, pluggability, and annotations.
9. **Servlet 3.1** (April 2013)
   - Added non-blocking I/O support.
10. **Servlet 4.0** (September 2017)
    - Introduced HTTP/2 support and server push.

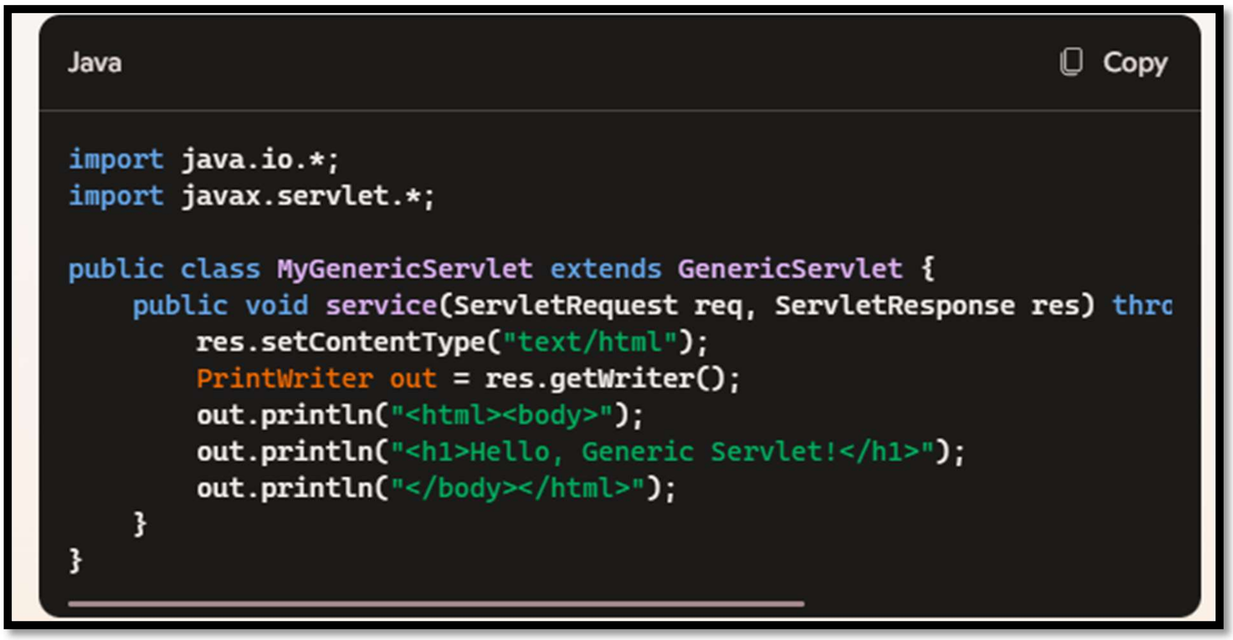
Types of Servlets

There are two main types of servlets: Generic Servlets and HTTP Servlets.

1. **Generic Servlets**:
   - **Definition**: A generic servlet is a protocol-independent, base class that implements the Servlet interface. It can be used to create servlets that are not dependent on a specific protocol.
   - **Usage**: Generic servlets are typically used for tasks that are not limited to a specific protocol, such as logging or communication over various protocols.
2. **HTTP Servlets**:
   - **Definition**: An HTTP servlet extends the HttpServlet class and is specifically designed to handle HTTP requests and responses. It provides methods to handle standard HTTP operations like GET, POST, PUT, DELETE, etc.

- **Usage**: HTTP servlets are widely used for building web applications and services that interact with web clients over the HTTP protocol.

Generic servlet Example:

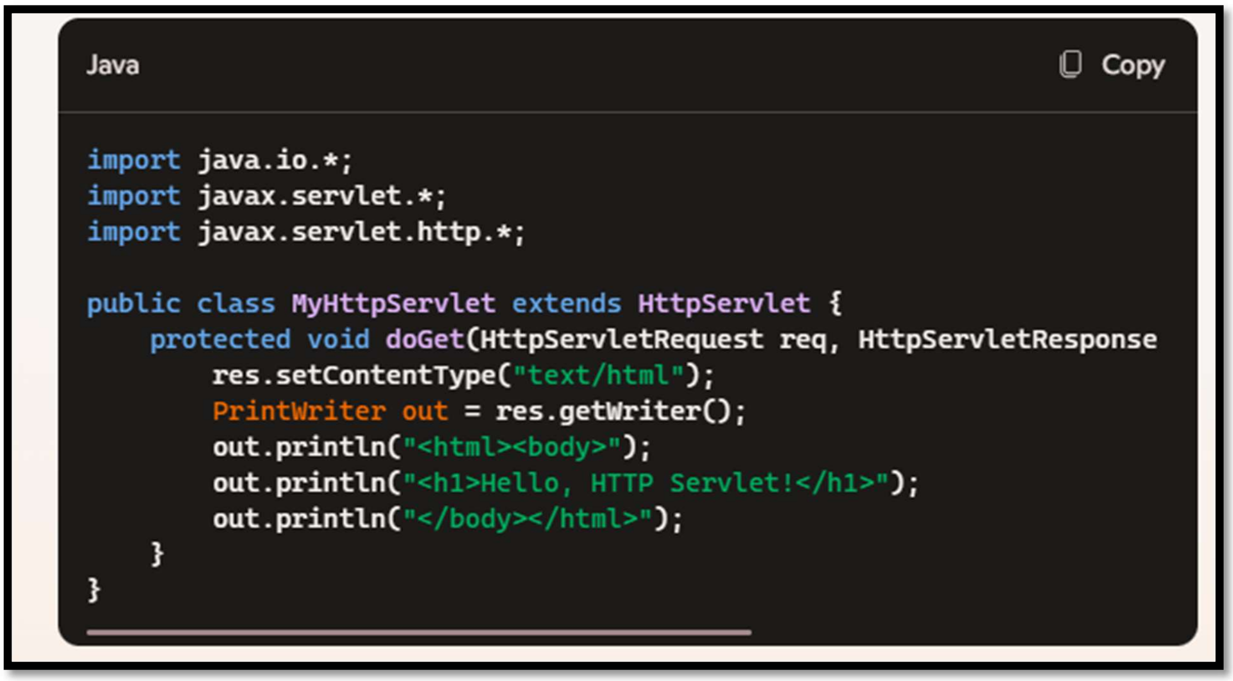


Java Copy

```java
import java.io.*;
import javax.servlet.*;

public class MyGenericServlet extends GenericServlet {
    public void service(ServletRequest req, ServletResponse res) thro
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><body>");
        out.println("<h1>Hello, Generic Servlet!</h1>");
        out.println("</body></html>");
    }
}
```

HTTP Servlet Example:



Java Copy

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyHttpServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><body>");
        out.println("<h1>Hello, HTTP Servlet!</h1>");
        out.println("</body></html>");
    }
}
```

In summary, servlet technology has evolved through various versions, adding significant enhancements and features. Servlets can be categorized into generic servlets, which are protocol-independent, and HTTP servlets, which are tailored for handling HTTP requests and responses.

Difference between HttpServlet and GenericServlet

HttpServlet and **GenericServlet** are two types of servlets in Java, each with distinct characteristics and use cases.

HttpServlet

- **Purpose**: Designed specifically to handle HTTP requests and responses.
- **Superclass**: Extends javax.servlet.http.HttpServlet.
- **Supported Methods**: Provides methods like doGet(), doPost(), doPut(), doDelete() to handle specific HTTP requests.
- **Use Case**: Ideal for web applications that interact using the HTTP protocol.
- **Request Handling**: Directly handles HTTP methods such as GET and POST.
- **Headers and Cookies**: Built-in support for managing HTTP headers and cookies.
- **Session Management**: Includes built-in support for managing HTTP sessions.
- **Ease of Use**: Easier to use for web-based applications due to its HTTP-specific methods.

- **Examples**: Commonly used for creating web pages, RESTful services, and APIs.

Code Example:

```
import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

public class MyHttpServlet extends HttpServlet {

   protected void doGet(HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException {

       res.setContentType("text/html");

       PrintWriter out = res.getWriter();

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

       out.println("<h1>Hello, HTTP Servlet!</h1>");

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

   }

}
```

GenericServlet

- **Purpose**: Designed to handle general, protocol-independent requests.
- **Superclass**: Extends javax.servlet.GenericServlet.
- **Supported Methods**: Provides the service() method to handle requests.
- **Use Case**: Suitable for applications that require generic request handling, not tied to any specific protocol.

- **Request Handling**: Requires overriding the service() method to handle requests.
- **Headers and Cookies**: Requires manual handling of headers and cookies.
- **Session Management**: Requires custom implementation for session management.
- **Ease of Use**: Requires more boilerplate code for handling protocol-specific tasks.
- **Examples**: Used for tasks such as logging or communication over various protocols.

Code Example:

```
import java.io.*;
import javax.servlet.*;
public class MyGenericServlet extends GenericServlet {
   public void service(ServletRequest req, ServletResponse res) 
throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><body>");
        out.println("<h1>Hello, Generic Servlet!</h1>");
        out.println("</body></html>");
   }
}
```

Module-3 (Web Technologies In java)

Servlet Life Cycle

Explanation of the servlet life cycle:

1. init() **Method:**
    - The init() method is called by the servlet container when the servlet is first created.
    - It is executed only once throughout the servlet's lifecycle.
    - Used for initialization tasks, such as establishing database connections, loading configuration settings, etc.
2. service() **Method:**
    - The service() method is invoked each time a request is made to the servlet.
    - It handles all client requests and processes them according to the request type (GET, POST, etc.).
    - It calls the corresponding doGet(), doPost(), doPut(), etc., methods based on the HTTP request type.
3. destroy() **Method:**
    - The destroy() method is called by the servlet container just before the servlet is destroyed.
    - It is executed only once throughout the servlet's lifecycle.
    - Used for cleanup activities, such as closing database connections, releasing resources, etc.

===

Creating Servlets and Servlet Entry in web.xml

How to create servlets and configure them using web.xml**:**

1. **Creating Servlets:**
   - To create a servlet, you need to extend the HttpServlet class provided by the Java Servlet API.
   - Override the necessary methods, such as doGet(), doPost(), etc., to handle HTTP requests.

Example:

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


public class MyServlet extends HttpServlet {
            protected void doGet(HttpServletRequest request,
HttpServletResponse response)
          throws ServletException, IOException {
    response.setContentType("text/html");
    response.getWriter().println("<h1>Hello, World!</h1>");
  }
}
```

2. **Configuring Servlets Using** web.xml**:**
   - The web.xml file, also known as the deployment descriptor, is used to configure servlets and map URLs to them.

Example web.xml entry:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5">
    <servlet>
        <servlet-name>MyServlet</servlet-name>
        <servlet-class>com.example.MyServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

In this example, MyServlet is mapped to the URL pattern /hello, meaning it will handle requests to http://yourdomain/hello.

==

Module-3 (Web Technologies In java)

Logical URL and ServletConfig Interface

Explanation of logical URLs and their use in servlets:

- **Logical URLs:**
  - Logical URLs are user-friendly URLs that map to servlet resources in a web application.
  - They help in creating meaningful and easily understandable paths for accessing different parts of a web application.
  - Logical URLs improve the readability and maintainability of web applications by abstracting the actual physical paths.
  - Example:
    - URL: http://example.com/products
    - Logical Path: /products maps to a servlet handling product-related requests.
- **Use in Servlets:**
  - In servlets, logical URLs are defined using the web.xml deployment descriptor or annotations.
  - They enable the servlet container to route incoming requests to the appropriate servlet based on the URL pattern.
  - Logical URLs make it easier to manage and modify the structure of web applications without changing the actual servlet code.

Module-3 (Web Technologies In java)

Overview of ServletConfig **and its methods:**

- ServletConfig **Interface:**
  - ServletConfig is an interface provided by the Java Servlet API.
  - It is used by the servlet container to pass configuration information to a servlet during initialization.
  - Each servlet has its own ServletConfig object.
- **Methods of** ServletConfig**:**
  - getServletName(): Returns the name of the servlet as defined in the deployment descriptor (web.xml).
  - getServletContext(): Returns a reference to the ServletContext object, which provides information about the web application.
  - getInitParameter(String name): Returns the value of a specified initialization parameter as defined in the web.xml file.
  - getInitParameterNames(): Returns an Enumeration of the names of the servlet's initialization parameters.

- **Example Usage:**

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

public class MyServlet extends HttpServlet {
    private String myParam;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        myParam = config.getInitParameter("myParam");
    }
}
```

RequestDispatcher Interface: Forward and Include Methods

Explanation of RequestDispatcher **and the** forward() **and** include() **methods:**

- RequestDispatcher **Interface:**
  - The RequestDispatcher interface is provided by the Java Servlet API.
  - It allows a servlet to forward a request to another resource (such as another servlet, JSP, or HTML file) or include the content of another resource in the response.
- forward() **Method:**
  - The forward() method is used to forward a request from one servlet to another resource on the server.
  - It transfers control to the specified resource and the original request and response objects are passed along.

Example usage: RequestDispatcher dispatcher =

```
request.getRequestDispatcher("targetResource");
dispatcher.forward(request, response);
```

  - Use cases:
    - When processing a request requires multiple resources.
    - To separate business logic and presentation by forwarding to JSPs.

- include() **Method:**
  - The include() method is used to include the content of another resource in the response.
  - It allows the original servlet to continue processing after including the content.
  - **Example usage:** RequestDispatcher dispatcher =

    ```
    request.getRequestDispatcher("header.jsp");
    dispatcher.include(request, response);
    ```
  - Use cases:
    - When a common resource needs to be included, such as headers, footers, or navigation bars.
    - To modularize the response by including multiple smaller resources.

Module-3 (Web Technologies In java)

ServletContext Interface and Web Application Listener

Introduction to ServletContext **and its scope:**

- ServletContext **Interface:**
  - The ServletContext interface is provided by the Java Servlet API.
  - It allows servlets to communicate with the servlet container and access resources shared across the entire web application.
  - ServletContext is available to all servlets and JSPs within a web application and serves as a shared memory space.
- **Scope of** ServletContext**:**
  - ServletContext has application-wide scope, meaning it is accessible from any servlet, filter, or listener in the web application.
  - It is used to store application-wide parameters, shared attributes, and configuration information.
  - ServletContext is created when the web application is deployed and remains available until the application is undeployed.
- **Common Methods of** ServletContext**:**
  - getInitParameter(String name): Returns the value of an initialization parameter.
  - getInitParameterNames(): Returns an Enumeration of the names of the initialization parameters.

  - setAttribute(String name, Object object): Sets an attribute in the context.
  - getAttribute(String name): Returns the attribute bound to the given name.
  - getResourceAsStream(String path): Returns an InputStream for reading a resource located at the specified path.

How to use web application listeners for lifecycle events:

- **Web Application Listeners:**
  - Web application listeners are used to respond to lifecycle events in a web application, such as context initialization, session creation, and request processing.
  - Listeners implement specific interfaces provided by the Java Servlet API, such as ServletContextListener, HttpSessionListener, ServletRequestListener, etc.
- **Common Listeners and Their Methods:**
  1. ServletContextListener**:**
      - Listens for lifecycle events related to the ServletContext.
      - Methods:
      - contextInitialized(ServletContextEvent sce): Called when the web application is started and the ServletContext is initialized.
      - contextDestroyed(ServletContextEvent sce): Called when the web application is stopped and the ServletContext is destroyed.

Example:

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class MyContextListener implements ServletContextListener
{
    public void contextInitialized(ServletContextEvent sce) {
        // Code to execute when the application starts
        System.out.println("Web application initialized.");
    }

    public void contextDestroyed(ServletContextEvent sce) {
        // Code to execute when the application stops
        System.out.println("Web application destroyed.");
    }
}
```

2. HttpSessionListener
    - Listens for lifecycle events related to HTTP sessions.
    - Methods:
        - sessionCreated(HttpSessionEvent se): Called when a new HTTP session is created.

- sessionDestroyed(HttpSessionEvent se): Called when an existing HTTP session is invalidated or timed out.

Example:

```
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class MySessionListener implements HttpSessionListener
{
  public void sessionCreated(HttpSessionEvent se)
  {
    // Code to execute when a session is created
    System.out.println("Session created.");
  }

  public void sessionDestroyed(HttpSessionEvent se)
  {
    // Code to execute when a session is destroyed
    System.out.println("Session destroyed.");
  }
}
```

3. ServletRequestListener:
   - Listens for lifecycle events related to HTTP requests.

- Methods:
  - requestInitialized(ServletRequestEvent sre): Called when a new request is received.
  - requestDestroyed(ServletRequestEvent sre): Called when a request is about to go out of scope.

Example:

```
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;


public class MyRequestListener implements ServletRequestListener
{
   public void requestInitialized(ServletRequestEvent sre)
   {
      // Code to execute when a request is initialized
      System.out.println("Request initialized.");
   }
   public void requestDestroyed(ServletRequestEvent sre)
   {
      // Code to execute when a request is destroyed
      System.out.println("Request destroyed.");
   }
}
```

Module-3 (Web Technologies In java)

Java Filters: Introduction and Filter Life Cycle

What are filters in Java and when are they needed?

- **Java Filters:**
  - Filters are Java components that can intercept and manipulate requests and responses in a web application.
  - They are used for tasks such as authentication, logging, input validation, and modifying request/response headers or content.
  - Filters operate on the request/response objects before they reach the servlet or after they leave the servlet.
- **When Filters are Needed:**
  - To perform preprocessing or postprocessing tasks on requests and responses.
  - To implement cross-cutting concerns such as security, logging, and data compression.
  - To apply transformations to request or response data.

Filter Life Cycle and How to Configure Them in web.xml**:**

- **Filter Life Cycle:**
  - init() **Method:**
    - Called by the servlet container when the filter is first instantiated.
    - Used for initialization tasks.
  - doFilter() **Method:**

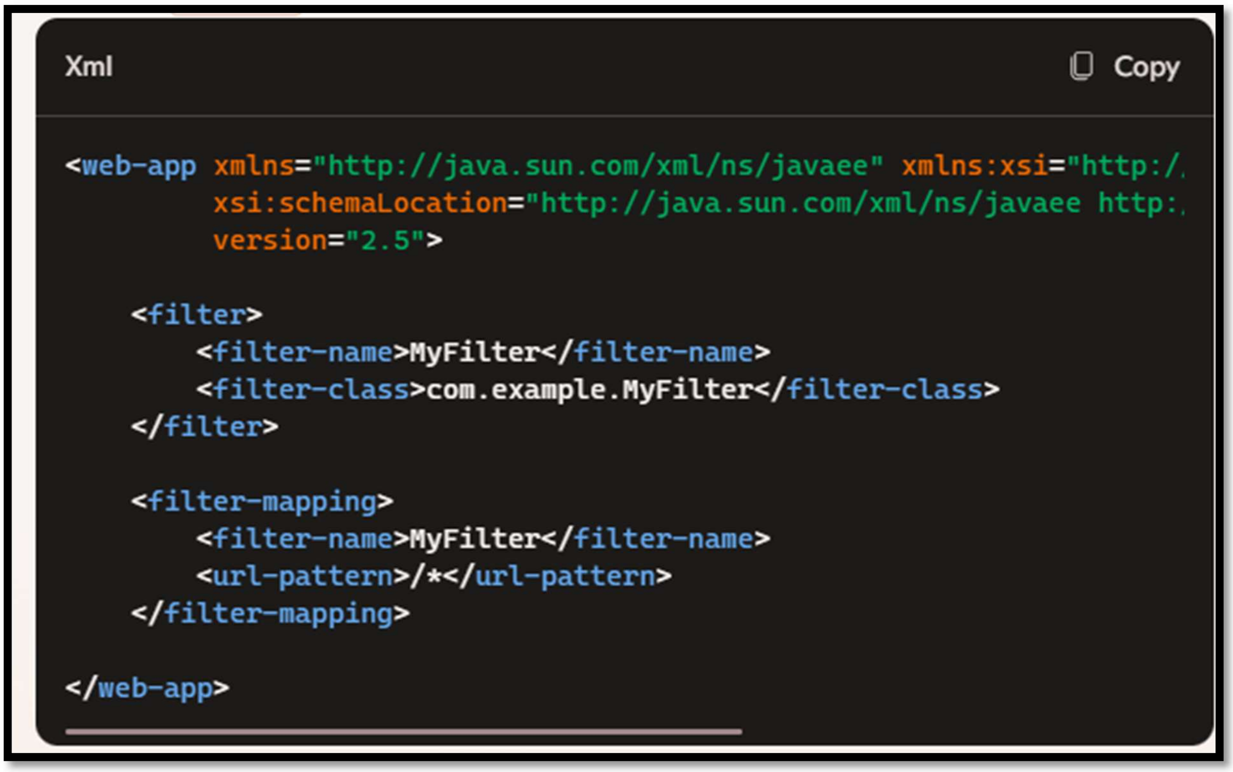
- Called each time a request/response pair is passed through the filter chain.
- Used to perform filtering tasks.

Example:

```
public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain)
        throws IOException, ServletException
{
    // Preprocessing
    // Pass the request along the filter chain
    chain.doFilter(request, response);
    // Postprocessing
}
```

- destroy() **Method:**
    - Called by the servlet container when the filter is taken out of service.
    - Used for cleanup tasks.

- **Configuring Filters in** web.xml**:**
  - **Example web.xml entry:**



```xml
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http:/
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http:
         version="2.5">

    <filter>
        <filter-name>MyFilter</filter-name>
        <filter-class>com.example.MyFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>MyFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

  - In this example, MyFilter is configured to apply to all URLs in the web application (/*).

Module-3 (Web Technologies In java)

JSP Basics: JSTL, Custom Tags, Scriptlets, and Implicit Objects

Introduction to JSP and its key components:

1. **JSP (JavaServer Pages):**
    - JSP is a server-side technology used to create dynamic web content.
    - It allows embedding Java code directly into HTML pages using special JSP tags.
2. **JSTL (JavaServer Pages Standard Tag Library):**
    - JSTL is a collection of standard tags that simplify the development of JSP pages.
    - It includes tags for common tasks such as iteration, conditionals, and formatting.

Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
            <c:forEach var="item" items="${items}">
                        ${item}
            </c:forEach>
```

3. **Custom Tags:**
    - Custom tags are user-defined tags that encapsulate reusable functionality in JSP pages.
    - They are created using the Java Servlet API and configured in the web.xml or using tag library descriptors.

Example:

```
<%@ taglib uri="customTagLib" prefix="custom" %>
        <custom:myTag attribute="value" />
```

4. **Scriptlets:**
    - Scriptlets are Java code embedded within JSP pages using <% %> tags.
    - They allow adding dynamic content and executing Java code directly within the HTML.

Example:

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

5. **Implicit Objects:**
    - Implicit objects are predefined objects in JSP that provide access to various aspects of the web application.
    - Common implicit objects:
        - **request:** Represents the client's request.
        - **response:** Represents the server's response.
        - **session:** Represents the HTTP session.
        - **application:** Represents the servlet context.
        - **out**: Used to send content to the client.

==

Module-3 (Web Technologies In java)

Session Management and Cookies

Overview of Session Management Techniques:

1. **Cookies:**
   - **Definition:** Cookies are small pieces of data stored on the client's browser.
   - **Purpose:** They maintain stateful information between the client and server across multiple requests.
   - **Usage:** Commonly used for tracking user sessions, preferences, and login states.
   - **Example:**

```
// Creating a cookie
Cookie userCookie = new Cookie("username", "JohnDoe");
// Setting cookie expiry to 24 hours
userCookie.setMaxAge(24 * 60 * 60);
// Adding the cookie to the response
response.addCookie(userCookie);
```

2. **Hidden Form Fields:**
   - **Definition:** Hidden form fields are used to store data within HTML forms that are not visible to users.
   - **Purpose:** They allow passing state information between pages when forms are submitted.
   - **Usage:** Useful in scenarios where cookies are not suitable or supported.
   - **Example:**

html

```html
<form action="nextPage.jsp" method="post">
    <input type="hidden" name="sessionID" value="12345">
    <input type="submit" value="Submit">
</form>
```

3. **URL Rewriting:**
   - **Definition:** URL rewriting appends session information to the URL query string.
   - **Purpose:** It is useful for tracking sessions in browsers that do not support cookies.
   - **Usage:** Commonly used as an alternative to cookies for maintaining session information.

Example:

```java
// Appending session ID to URL
String urlWithSessionID = response.encodeURL("NextPage’s");
response.sendRedirect(urlWithSessionID);
```

4. **Sessions:**
   - **Definition:** Sessions allow storing user-specific data on the server side.
   - **Purpose:** They provide a way to maintain state across multiple requests from the same user.
   - **Usage:** Typically used for storing user information, preferences, and login states.
   - **Example:**

-
  ```
  // Creating a session
   HttpSession session = request.getSession();
  // Storing data in the session
  session.setAttribute("username", "JohnDoe");
  ```

How to Track User Sessions in Web Applications:

1. **Using Cookies:**
   - **Method:** Store session identifiers (session IDs) as cookies on the client's browser.
   - **Process:** The server generates a session ID and sends it as a cookie to the client. The client includes the cookie in subsequent requests, allowing the server to identify the session.

Example:

```
 // Retrieving cookies from the request
Cookie[] cookies = request.getCookies();
for (Cookie cookie : cookies) {
if (cookie.getName().equals("JSESSIONID")) {
   String sessionID = cookie.getValue();
   // Use sessionID to retrieve session data
     }
 }
```

2. **Using URL Rewriting:**
   - **Method:** Append session identifiers to URLs.

- **Process:** The server appends the session ID to URLs. The client includes the session ID in the URL when making requests, allowing the server to identify the session.

Example:

```
// Appending session ID to URL
String urlWithSessionID = response.encodeURL("nextPage.jsp");
response.sendRedirect(urlWithSessionID);
```

3. **Using Sessions:**
   - **Method:** Create a session for each user on the server.
   - **Process:** The server generates a session ID and associates it with a session object. The session ID is sent to the client as a cookie or appended to URLs. The server retrieves the session data using the session ID for subsequent requests.

Example:

```
// Creating a session
HttpSession session = request.getSession();
// Storing data in the session
session.setAttribute("username", "JohnDoe");
// Retrieving data from the session
String username = (String) session.getAttribute("username");
```

The End