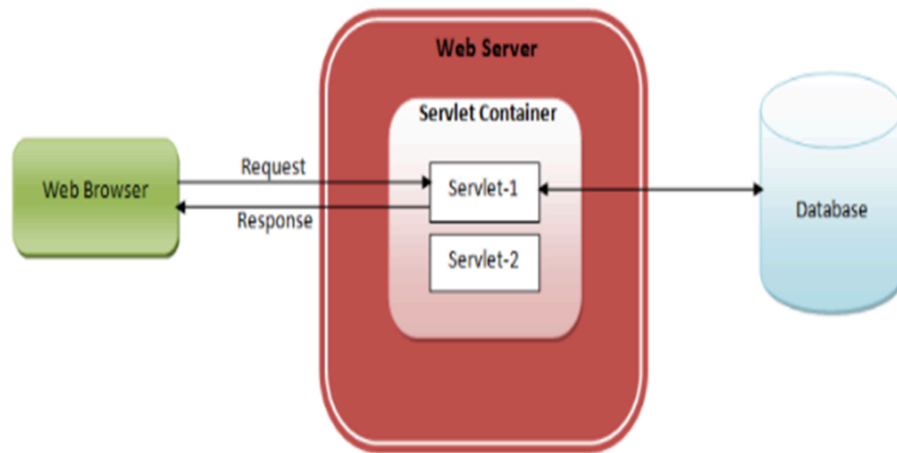


Q1. Explain Servlet architecture.

Servlet is a java program that runs inside JVM on the web server. They reside on the server side and their main purpose is to serve the client request.

Servlet class is instantiated by the server to produce a dynamic response and it acts as an interface between the HTTP server and the database.



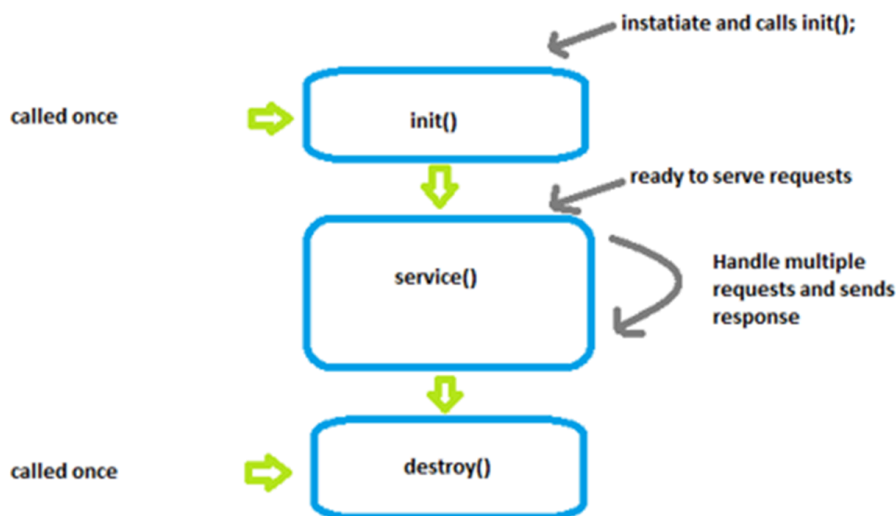
Execution of Servlets:

- 1) The client sends a request to the web server.
- 2) The web server receives the request and passes it to the servlet container.
- 3) The servlet container understands the request's URL and calls the corresponding servlet.
- 4) The servlet processes the request and generates an output response in the form of HTML, XML, JSON etc using the database.
- 5) The servlet sends the output response back to the web server.
- 6) The web server sends the response to the client

Q2. Explain the Servlet lifecycle.

The **Servlet lifecycle** describes the stages a servlet goes through during its existence in a web application, all managed by the **Servlet Container**.

The web container maintains the life cycle of the servlet because all the servlets are present in it.



### 1) Servlet class is loaded:

- The servlet class is responsible for loading the servlet. It is loaded when the first request for the servlet is received by the web container.
- It happens only once, unless the server is reloaded.

### 2) Servlet Instance is created:

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

### 3) Initialization (init method is invoked):

- The web container calls the init() method only once after creating the server instance. The init method is used to initialize the servlet and perform tasks like:
  1. Connecting to the database
  2. Loading configuration values
  3. Setting up the resources
- It is the life cycle method of the javax.servlet.Servlet interface.
- Syntax:

public void init (ServletConfig config) throws ServletException

### 4) Request Handling (Service method is invoked):

- The web container calls the service() method each time when the request for the servlet is received.
- The container provides two objects:
  1. `HttpServletRequest` → contains client data (parameters, headers, cookies).
  2. `HttpServletResponse` → used to send back a response (HTML, JSON, etc.).
- Syntax:

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

- In `HttpServlet`, service() dispatches requests to:
  1. `doGet()` → for HTTP GET requests.
  2. `doPost()` → for HTTP POST requests.
  3. `doPut()`, `doDelete()`, etc.
- This step happens **many times**, once for each request.

### 5) Destruction (destroy() method):

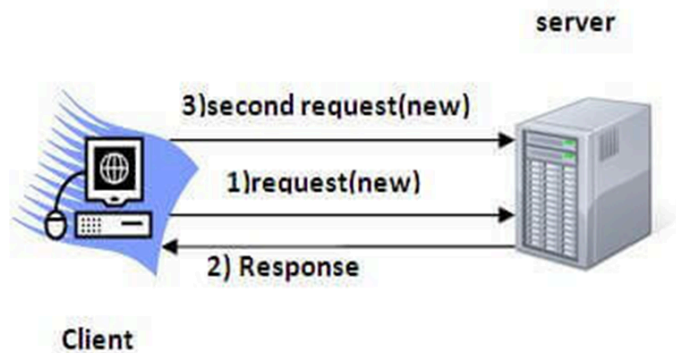
- The web container calls the destroy() method when the servlet is being removed (server shutdown, undeployed etc). Doing so, its is able to:
  1. Close the database connections.
  2. Stop the background threads

3. Clean up the memory so that variables can be reused in other parts of the program

- Syntax:  
`public void destroy()`

Q3. Explain session handling in Servlet

Session handling is one of the most important concepts in **Servlets** because HTTP is **stateless**, that means each request is considered a new request and all requests and responses are independent.



Hence we need to maintain states using session handling.

A **session** simply means a particular interval of time

For example:

- When a user logs in, the server remembers who they are.
- Their shopping cart items remain as they browse different pages.

Session handling is a way to maintain the state(data) of a user. It is also known as Session management in Servlet.

Techniques for session handling:

**1) URL Rewriting:**

- Session data is stored as **extra information in the URL**.
- The server appends a unique session ID to the URL.
- Example:

<http://example.com/welcome?sessionId=12345>

- Drawbacks: Session ID is visible in the URL, less secure.

## 2) Hidden Form Fields:

- A hidden field in an HTML form stores session information.
- Example:

```
<form action="nextPage" method="post">  
  <input type="hidden" name="sessionId" value="12345">  
  <input type="submit" value="Go">  
</form>
```

- Works only if the user navigates via forms (not links).

## 3) Cookies:

- Small pieces of data stored on the **client's browser**.
- The server creates a cookie with a **session ID**, and the browser sends it back with every request.
- Example code in servlet:

```
Cookie user = new Cookie("username", "John");  
response.addCookie(user);
```

- Commonly used, but may be disabled by users.

## 4) HttpSession (Most Common in Servlets):

- Servlet API provides `HttpSession` object for session tracking.
- Server automatically manages a **session ID** (usually with cookies).

- Example usage:

```
HttpSession session = request.getSession();
session.setAttribute("user", "John");

String user = (String) session.getAttribute("user");
```

- Session persists until:
  1. Timeout occurs.
  2. User logs out.
  3. Session is invalidated manually (`session.invalidate()`).

Q4. Explain steps to connect java application to database using JDBC using one example.

JDBC (Java Database Connectivity) is a standard Java API for connectivity between the Java programming language and a wide range of databases like MySQL, Oracle, PostgreSQL etc.

JDBC API takes care of converting java Commands to generic SQL statements.

JDBC API uses drivers provided by Database vendors to communicate with databases.

There are 5 steps to connect any java application with the database using JDBC:

### 1. Register the Driver class:

- This step tells Java **which database we are connecting to**.
- **forName** method of **Class** class is used to register the driver class
- This method is used to dynamically load the driver class.
- **Syntax:**

public static void forName (String className) throws ClassNotFoundException

- **Example:**

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

## 2. Create connection:

- After loading the driver, we need to connect Java to the database.
- getConnection method of DriverManager class is used to establish connection with the database.
- **Syntax:** (two forms)
  1. public static Connection getConnection(String url)throws SQLException
  2. public static Connection getConnection(String url,String name,String password)throws SQLException
- **Example:**

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/testdb", "root", "password");
```

## 3. Create statement:

- The statement object is responsible to execute queries with the database.
- **createStatement()** method of Connection interface is used to create statement.
- **Syntax:**

public Statement createStatement() throws SQLException

- **Example:**  
Statement stmt = con.createStatement();

## 4. Execute queries:

- Now, we use the statement object to execute SQL commands.
- executeQuery() method of Statement interface is used to execute queries to the database.
- This method returns the object of ResultSet that can be used to get all the records of a table
- **Syntax:**  
public ResultSet executeQuery (String sql )throws SQLException
- **Example:**  
ResultSet rs = stmt.executeQuery("SELECT \* FROM users");

## 5. Close connection:

- By closing connection object, Statement and ResultSet will be closed automatically

- **Syntax:**

public void close() throws SQLException

- **Example:**

rs.close();

stmt.close();

con.close();

## Example Program:



```

import java.sql.*;

class JdbcExample {
    public static void main(String args[]) {
        try {
            // Step 1: Register the Driver class
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Create connection
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/testdb", "root", "password");

            // Step 3: Create statement
            Statement stmt = con.createStatement();

            // Step 4: Execute query
            ResultSet rs = stmt.executeQuery("SELECT * FROM users");
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " " + rs.getString(2));
            }

            // Step 5: Close connection
            rs.close();
            stmt.close();
            con.close();

        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```

Q5. Explain JDBC Architecture with a diagram.

JDBC (**Java Database Connectivity**) is a standard **Java API** that allows Java applications to interact with relational databases.

Its architecture defines how a Java program communicates with different databases in a consistent way.

# JDBC Components

## 1. JDBC API (Java side)

- Provides classes & interfaces like **DriverManager**, **Connection**, **Statement**, **ResultSet**.
- Used by developers to send SQL statements from Java app to DB.

## 2. JDBC Driver Manager

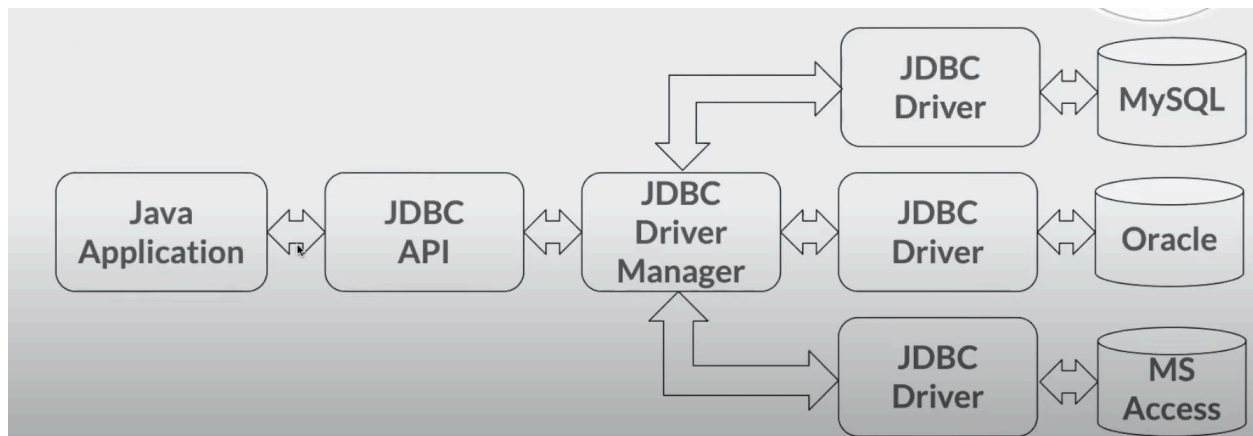
- Manages different types of database drivers.
- Loads drivers and establishes a connection.

## 3. JDBC Drivers

- Vendor-specific implementations (MySQL, Oracle, PostgreSQL).
- Translate JDBC calls into database-specific statements.

## 4. Database

- The actual database (e.g., MySQL, Oracle, etc.) that stores data.



# Flow of JDBC

1. Java application uses **JDBC API** to make a request.
2. The request is passed to the **DriverManager**.
3. **DriverManager** selects the correct JDBC **Driver** (e.g., MySQL driver).
4. The **Driver** translates the JDBC call into database-specific protocol.
5. The **Database** executes the SQL and sends results back.

6. The driver converts database results into `ResultSet` objects.
7. Java application processes results.

Q6. Explain characteristics of Rich Internet Application(RIA).

Applications developed to provide exciting rich interactivity features in software programs are known as Rich Internet Applications (RIA).

RIAs are developed as web applications behaving like desktop applications in their look & feel.

Eg. Google Maps, Google Docs

RIA can perform computation on both the client side and server side.

It is developed using various technologies that includes Java, JavaFX, Adobe Flash & Flex, AJAX

Characteristics of RIA:

### **1. Enhanced Accessibility**

- RIAs run inside standard web browsers, so they don't require complex installation.
- They are accessible from anywhere, anytime, as long as the user has an internet connection.
- Some RIAs even support mobile devices.

### **2. Direct Interaction**

- RIAs provide rich UI components (drag-and-drop, sliders, context menus, auto-suggestions).
- Users interact directly with the application, similar to desktop software.
- Example: In Google Maps, you can drag the map instead of clicking refresh or reloading the page.

### 3. Partial Update of Web Pages

- Unlike traditional web apps that reload the entire page, RIAs update only **specific parts** of a page using AJAX/JavaScript.
- This reduces page flicker and improves speed.
- Example: In Gmail, when a new mail arrives, only the inbox section updates, not the whole page.

### 4. Collaborative Web Application Access

- RIAs allow multiple users to access and work on the same application simultaneously.
- Example: Multiple people editing a Google Docs file at the same time.

### 5. Offline Use of the Application

- Some RIAs provide **offline capabilities** where data is temporarily stored locally.
- Once the internet connection is restored, changes sync back to the server.
- Example: Google Docs can be used offline and syncs later.

### 6. Impact on Performance

- Since RIAs process much of the data on the **client-side**, they reduce server load.
- Partial page updates and asynchronous calls make them faster and more responsive.
- However, heavy RIAs may consume more memory/CPU on weaker client devices.

Q7. Explain Traditional Approach and AJAX.

#### 1) Traditional Approach:

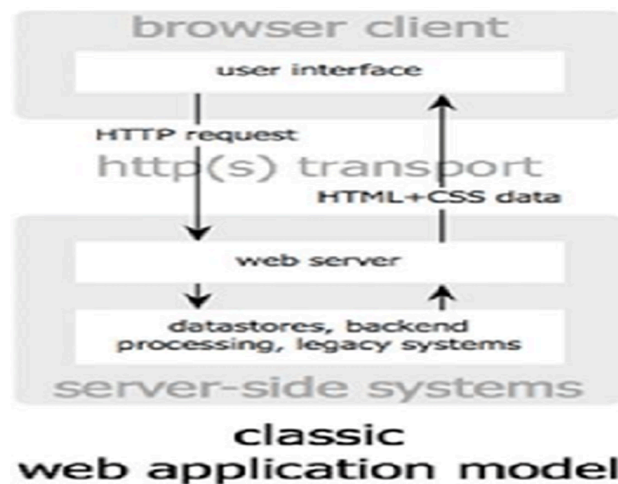
In the **traditional web model** (before AJAX), communication between **client** (browser) and **server** worked like this:

- Whenever a user interacts (e.g., clicks a link, submits a form), the browser sends a **full HTTP request** to the server.

- The server processes it by calling upon server-side scripts and databases required and sends back a **complete HTML page** as response.
- The **entire page reloads** in the browser, even if only a small portion needed to change.

### Drawbacks

- Slower performance (full-page reload every time).
- Flickering of pages (bad user experience).
- More data transfer (wastes bandwidth).
- Less interactive and responsive.



## 2) AJAX (Asynchronous JavaScript and XML):

AJAX is a **modern approach** that allows **asynchronous communication** between browser and server.

It is a technique, which describes how other technologies, JavaScript , DOM and XML can be used to create interactive web applications.

- Instead of reloading the entire page, AJAX lets the browser send/receive data **in the background** using JavaScript.
- Only the **required part of the web page is updated** (partial page refresh).
- Data can be sent/received in **XML, JSON, or plain text**.

## Advantages

- Faster and more responsive.
- Smooth user experience (no flickering/reload).
- Reduces bandwidth usage (only required data sent).
- Allows real-time applications (chat apps, live search, stock updates).



Q8. Write a short note on JSTL (for each JSTL mention one example)

The JSTL (JavaServer Pages Standard Tag Library) is a collection of predefined ready-made tags to simplify JSP development.

It simplifies JSP development by reducing the need for java code inside JSP

## Types JSTL Tags:

### 1) Core Tags (c tag library):

- Used for iteration, URL checking, and condition checking.
- **Example (Loop using <c:forEach>):**  

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

## 2) Formatting Tags (fmt tag library):

- Used for formatting numbers, dates, currencies, and for internationalization.
- **Example (Formatting a number as currency):**  
`<fmt:formatNumber value="12345.67" type="currency"/>`

## 3) SQL Tags (sql tag library):

- Used for executing SQL queries and updates directly from JSP
- **Example (Select query):**  
`<sql:query var="result" dataSource="${myDS}">  
 SELECT * FROM users;  
</sql:query>`

## 4) XML Tags (x tag library):

- Used for parsing XML data
- **Example (Parsing XML):**  
`<x:parse xml="${xmlData}" var="doc"/>`

## 5) Functions Tags (fn tag library):

- Provide standard string manipulation functions.
- **Example (Check if string contains substring):**  
`<c:if test="${fn:contains('HelloWorld', 'World')}">  
 Found substring!  
</c:if>`

## Advantages:

- 1) **Fast Development:** JSTL provides many predefined tags that simplify JSP
- 2) **Code Reusability:** We can use JSTL tags on various pages.
- 3) **No need to use the scriptlet tag:** It avoids the use of scriptlet tag which in turn avoids a whole lot of syntax.

Q9. Explain Schema rules for DTD in XML using proper examples

## DTD (Document Type Definition)

- A **DTD** defines the **structure and rules** for an XML document.
- It ensures that the XML data is **valid** (follows the defined structure).
- DTD can be **internal** (inside XML file) or **external** (separate file).

# Schema Rules for DTD in XML

## 1. Defining Elements

- Every XML element used must be declared in the DTD.
- **Syntax:**  
`<!ELEMENT element-name content-type>`
- **Content Types:**
  1. `#PCDATA` → parsed character data (text).
  2. `EMPTY` → element cannot contain content.
  3. `ANY` → element can contain anything.
  4. `(child1, child2, ...)` → defines a structure (they are the children tags).
- **Example:**

```
<!DOCTYPE student [  
    <!ELEMENT student (name, age, course)>  
    <!ELEMENT name (#PCDATA)>  
    <!ELEMENT age (#PCDATA)>  
    <!ELEMENT course (#PCDATA)>  
>  
<student>  
    <name>Amit Sharma</name>  
    <age>21</age>  
    <course>Computer Science</course>  
</student>
```

## 2. Defining Attributes:

- Attributes are declared using `<!ATTLIST>`.
- **Syntax:**  
`<!ATTLIST element-name attribute-name attribute-type  
default-value>`



- **Attribute Types:**

1. **CDATA** → character data.
2. **ID** → unique identifier.
3. **IDREF / IDREFS** → references to IDs.
4. **ENUMERATION** → one value from a list.

- **Example:**

```
<!DOCTYPE book [  
  <!ELEMENT book (title, author)>  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT author (#PCDATA)>  
  <!ATTLIST book id ID #REQUIRED>  
>  
<book id="b101">  
  <title>Java Basics</title>  
  <author>James Gosling</author>  
</book>
```

### 3. Cardinality (Occurrence Indicators):

- **To specify how many times an element can appear:**

1. **?** → Allows element to appear zero or just once within parent tag
2. **\*** → Allows elements to appear zero or more times in the parent tag..
3. **+** → Allow elements to appear one or more times in the parent tag.

- **Example:**

```
<!DOCTYPE student [  
  <!ELEMENT student (name, age?)>  
  <!ELEMENT name (#PCDATA)>  
  <!ELEMENT age (#PCDATA)>  
<student>  
  <name>Amit</name>  
  <!-- age is optional -->  
</student>
```

#### 4) Order of elements:

- DTD enforces the order of elements.
- If order differs in XML, it is invalid.
- **Example:**  
<!ELEMENT book (title, author, price)>


```
<book>  
  <title>Java Basics</title>  
  <author>James</author>  
  <price>400</price>  
</book>
```

#### 5) Nesting of Elements:

- Elements can be nested, and DTD defines valid nesting.

##### Rule in DTD

dtd

 Copy code

```
<!ELEMENT library (book+)>  
<!ELEMENT book (title, author)>  
<!ELEMENT title (#PCDATA)>  
<!ELEMENT author (#PCDATA)>
```

- `library` contains one or more `book` elements ( `book+` ).
- `book` must contain `title` and `author` elements, in that order.
- `title` and `author` contain text ( `#PCDATA` ).

- **Example:**

```
<!DOCTYPE library [  
  <!ELEMENT library (book+)>  
  <!ELEMENT book (title, author)>  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT author (#PCDATA)>  
>  
<library>  
  <book>  
    <title>Operating System</title>  
    <author>Galvin</author>  
  </book>  
  <book>  
    <title>Database System</title>  
    <author>Korth</author>  
  </book>  
</library>
```

Q10. Explain XML tag naming conventions.

XML has some rules for naming the tags in the XML files. Following are the rules:

1. **Case Sensitivity:**

XML tags are case-sensitive, meaning the opening and closing tags must match exactly in capitalization. <Book>, <BOOK>, and <book> are treated as three different elements, and mismatched cases will result in validation errors.

Example: <Title> must be closed with </Title>, not </title>

2. **Begin with a Letter or an Underscore**

XML tag names must start with a letter (a-z, A-Z) or an underscore character. They cannot begin with numbers, punctuation marks, or any

other special characters, ensuring proper parsing and compatibility across systems.

Example: <\_privateData> - Valid, <123Data> - Invalid

### **3. Valid Character Mix**

Tag names can contain any combination of letters, numbers, underscores, hyphens, and periods after the first character. This provides flexibility while maintaining clean, readable naming conventions without confusing the parser.

Example: <product.id-123> or <user\_name.v2>

### **4. Reserved "xml" Prefix**

The string "xml" in any capitalization combination is reserved and cannot be used as the beginning of tag names. This prevents conflicts with XML processing instructions and built-in system functionality.

Example: Use <data> instead of <xmlData>

### **5. Prohibited Characters < and &**

Tag names cannot contain the characters < or & as these have special meanings in XML syntax. The < indicates the start of a tag, while & denotes entity references, making them invalid within tag names.

Example: Use <price\_in\_dollars> instead of <price&dollars>

Q11. Explain the difference between XML and HTML.

Feature	XML (Extensible Markup Language)	HTML (HyperText Markup Language)
Primary Purpose	Transports and Stores Data. It is a framework for defining custom markup languages.	Displays Data and Structures Web Pages. It is the standard language for creating web pages.
Tag Nature	User-Defined / Extensible. The author creates their own tags tailored to their data. e.g., <book>, <author>, <price>	Pre-Defined / Fixed. Comes with a fixed set of tags for web structure. e.g., <h1>, <p>, <table>, <div>
Focus	On the "What" data is. It focuses on the nature and structure of the information itself.	On the "How" data looks. It focuses on the presentation and layout of information.
Case Sensitivity	Case Sensitive. <Message> and <message> are considered different tags.	Not Case Sensitive. <P> and <p> are treated the same, though lowercase is the standard.
Validation	Can be Validated. Uses DTD (Document Type Definition) or XML Schema (XSD) to enforce strict rules on structure and data types.	HTML can have errors but browsers will still display it, unlike XML which fails completely with even a single mistake.
Error Handling	Strict. An XML parser will stop processing if it encounters an error, like a missing end tag.	Lenient. Web browsers will try to render a page even if the HTML contains errors, like missing or mismatched tags.
*Data Presentation	No Built-in Presentation. Requires other technologies like CSS (Cascading Style Sheets) or XSLT (Extensible Stylesheet Language Transformations) to style and display the data.	Built-in Presentation. Has inherent presentational tags and attributes (e.g., <b>, <font>, color), though modern best practice uses CSS.

*Quoting Attributes	Mandatory. All attribute values must be enclosed in quotes. e.g., <note id="501">	Optional (but recommended). Browsers will often accept unquoted attribute values, but it is a best practice to use them. e.g., <img src=image.png> may work, but  is correct.
---------------------	--	--

\* → Not so important

## Q12. Explain Life Cycle of JSP

JSP (Java Server Pages) is a server-side technology used to create dynamic web pages.

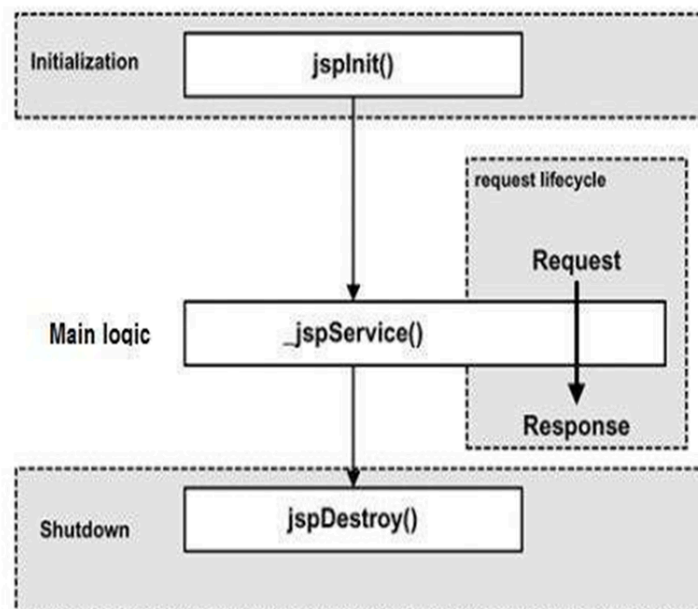
Earlier it was tough to use dynamic web pages using servlet, hence JSP was introduced

JSP is internally converted into a Servlet by the web container.

The JSP Life Cycle describes the complete process of how a JavaServer Page is handled by the web container from its initial request to final destruction.

### Life Cycle of JSP:

**Translation → Compilation → Initialization → Execution → Destroy.**



## 1. Translation Phase

- The JSP file (`.jsp`) is **translated into a Java Servlet class** by the web container (like Tomcat).
- Example: `index.jsp` → `index_jsp.java`.
- During this phase, all JSP elements like scriptlets, expressions, and directives are translated into equivalent Java code while static HTML content is converted into `out.write()` statements.
- This translation occurs when the JSP is accessed for the first time

## 2. Compilation Phase

- The generated Java Servlet (`index_jsp.java`) is compiled into a bytecode (`.class` file) using the Java compiler.
- Example: `index_jsp.class`.
- In this phase, the container checks for any syntax errors and validates all JSP elements and if compilation fails, it sends an error to the client, otherwise proceeds to compile the servlet.

## 3. Loading and Initialization

- The compiled Servlet class is **loaded into memory** by the class loader.
- The container calls the `jspInit()` method (similar to `init()` in Servlets).
- This method is called only **once** when JSP is first loaded.
- Use this for **resource initialization** (like DB connection setup, opening files).

## 4. Execution (Request Processing)

- For each client request, container calls `_jspService()` method.
- This method automatically processes all the dynamic parts of your JSP page and generates the final HTML that gets sent back to the browser

- It works with built-in objects like (`HttpServletRequest`) and (`HttpServletResponse`) to handle user input and create the webpage output.
- This method is called **every time a request comes**.

## 5. Destroy

- When the JSP is no longer needed or the server shuts down, the container calls the `jspDestroy()` method.
- Used for **cleanup tasks** like closing DB connections, releasing resources, etc.

Q13. Explain various JSP Scripting elements with examples.

JSP (JavaServer Pages) has **scripting elements** that allow us to embed Java code into HTML pages. They make JSP dynamic.

## 1. Declarations (`<%! . . . %>`)

- Used to declare variables and methods.
- Declarations go into the **Servlet class** (outside the `_jspService()` method).
- Syntax:  
`<%! declaration %>`
- Example:

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

## 2. Scriptlets (`<% . . . %>`)

- Used to write Java code inside JSP.
- Code written here goes into the **`_jspService()` method** (executed for every request).



- Syntax:  
`<% java code %>`
- Example:

```
<%  
    int a = 10, b = 20;  
    int sum = a + b;  
%>
```

### 3. Expressions (`<%= ... %>`)

- Used to output values directly to the response.
- Equivalent to `out.print(expression)`.
- Syntax:  
`<%= expression %>`
- Example:  
`<p>Current Time: <%= new java.util.Date() %></p>`

### 4. Directives (`<%@ ... %>`)

- Provide global information to the JSP container (like importing classes, including files, page settings).
- Syntax:  
`<%@ directive attribute="value" %>`
- Types:
  1. `<%@ page ... %>`:
    - Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
    - Example:  
`<%@ page language="java" import="java.util.*" %>`
  2. `<%@ include ... %>`:
    - Includes a file during the translation phase.

- Example:

```
<%@ include file="header.jsp" %>
```

### 3. `<%@ taglib ... %>`:

- Declares a tag library, containing custom actions, used in the page
- Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

## 5. Comments

- JSP supports two types of comments:
- **HTML comment** (visible in source code):

```
<!-- This is HTML comment -->
```

- **JSP comment** (not sent to client):

```
<%-- This is JSP comment --%>
```

## 6. Actions

- Actions are used to **dynamically include resources, forward requests, create/use JavaBeans, or interact with objects.**

- Syntax:

```
<jsp:actionName attribute="value" ... />
```

- Example:

```
<jsp:forward page="welcome.jsp" />
```

(Forwards the request to another resource)

Q14. How to create an XMLHttpRequest object in AJAX.

Allows requests to be sent from javascript to the web server.

### 1. Creating XMLHttpRequest Object:

```
var xhr = new XMLHttpRequest();
```

The XMLHttpRequest object is supported by all modern browsers.

But the older browser support can be checked using: **ActiveXObject**

Example:

```
if(ActiveXObject)
    request=new ActiveXObject("Microsoft.XMLHTTP");
else
    return false;
```

### 2. Open() Method:

- The **open()** method of the **XMLHttpRequest** object **initializes a request**.
- It tells the browser:
  - **What type of HTTP request** to make (GET, POST, etc.)
  - **Which URL** to send the request to
  - **Whether the request should be asynchronous or synchronous**
- **Syntax:**

```
xhr.open(method, url, async);
```

### 3. Send():

- Requests can then be sent to the server or the null keyword.
- **Syntax:**  

```
r.send()
```

### 4. readyState Property:

The readyState property of the XMLHttpRequest object indicates the status of the request. It can have 5 possible values:

Number	State
0	The request is uninitialized ,as open() has not been called
1	The request is specified ,as send() has not been called.
2	The request is being sent, as send() has now been called
3	The response is being received, but is not yet complete
4	The response is complete and returned data is available

## 5. Status property:

The `status` property gives the HTTP status code of the response:

- `200` → OK
- `404` → Not Found
- `500` → Server Error

### Example:

```

<!DOCTYPE html>
<html>
<body>

<div id="demo">Server response will appear here</div>

<script>
  // 1. Create XMLHttpRequest object
  var xhr = new XMLHttpRequest();

  // 2. Define callback function
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      // 4 = DONE, 200 = OK
      document.getElementById("demo").innerHTML = xhr.responseText;
    }
  };

  // 3. Initialize request
  xhr.open("GET", "data.txt", true);

  // 4. Send request
  xhr.send();
</script>

</body>
</html>

```

Q15. Explain AJAX methods provided by jQuery with examples.

jQuery provides several methods for AJAX functionality.

With the jQuery AJAX methods, you can request text, HTML, XML, or JSON from a remote server using both HTTP Get and HTTP Post.

1) load() method:

- The load() method loads data from a server and puts the returned data into the selected element.
- Syntax:

\$(selector).load(URL, data, callback);

- The required URL parameter specifies the URL you wish to load.

- The optional data parameter specifies a set of querystring key/value pairs to send along with the request.
- The optional callback parameter is the name of a function to be executed after the load() method is completed.
- Example:  

```
$("#demo").load("data.txt");
```

## 2) Get() Method:

- The get() method requests data from the server with an HTTP GET request.
- Syntax:

```
$.get(URL,callback);
```

- Example:  

```
$.get("data.txt", function(response) {  

    $("#demo").html(response);  

});
```

## 3) Post() Method:

- The post() method requests data from the server using an HTTP POST request.
- Syntax:

```
$.post(URL,data,callback);
```

- Example:  

```
$.post("submit.php", { name: "Amit", age: 21 }, function(response) {  

    $("#demo").html(response);  

});
```

Q16. State the various ways of receiving response from server using AJAX.  
Support your answer with an example.

## Ways of Receiving Response from Server Using AJAX

When a client sends a request to the server using AJAX, the server can respond in different formats:

1. Text Response
  - Server sends plain text.
  - Use `xhr.responseText` to read the response.
2. XML Response
  - Server sends **XML data**.
  - Use `xhr.responseXML` and DOM methods to read it.
3. JSON Response
  - Server sends **JSON data**.
  - Use `JSON.parse(xhr.responseText)` to convert to JavaScript object.

Example:

```
<script>
var xhr = new XMLHttpRequest();


xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {

        // ----- TEXT RESPONSE -----
        document.getElementById("textDiv").innerHTML = xhr.responseText;

        // ----- XML RESPONSE -----
        if (xhr.responseXML) {
            var msg = xhr.responseXML.getElementsByTagName("message")[0];
            if(msg) document.getElementById("xmlDiv").innerHTML = msg.childNodes[0].nodeValue;
        }

        // ----- JSON RESPONSE -----
        try {
            var data = JSON.parse(xhr.responseText);
            if(data.name) document.getElementById("jsonDiv").innerHTML = "Name: " + data.name;
        } catch(e) {
            // Not JSON, ignore
        }
    }
};

// Initialize request (you can change URL to data.txt, data.xml, or data.json)
xhr.open("GET", "data.txt", true);
xhr.send();
</script>
```

 Copy code

Q17. How jQuery uses GET and POST method to request data from server.

Refer Q15



Q18. Write a short note on XML DOM.

## XML DOM (Document Object Model)

- **Definition:**

The **XML DOM** is a **standard way to represent and interact with an XML document as a tree structure**.

It allows programs (JavaScript, Java, etc.) to **access, read, and manipulate XML elements, attributes, and content** dynamically.

---

### Key Features

1. **Tree Structure:**

- XML elements are nodes in a tree (root, parent, child, sibling nodes).

2. **Platform and Language Independent:**

- Works across browsers and programming languages.

3. **Access and Manipulation:**

- Read, edit, add, or delete elements and attributes.

4. **Node Types in XML DOM:**

- **Element Node** → Represents an XML element
  - **Attribute Node** → Represents an attribute of an element
  - **Text Node** → Represents the text inside an element
  - **Document Node** → Represents the entire XML document
-

## Example: Accessing XML using DOM (JavaScript)

Sample XML (`data.xml`):

```
<student>
  <name>Amit</name>
  <age>21</age>
</student>
```

JavaScript using XML DOM:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "data.xml", false);
xhr.send();

var xmlDoc = xhr.responseXML;
var name = xmlDoc.getElementsByTagName("name")[0].childNodes[0].nodeValue;
var age = xmlDoc.getElementsByTagName("age")[0].childNodes[0].nodeValue;

console.log("Name: " + name + ", Age: " + age);
```

**Output:**

yaml

```
Name: Amit, Age: 21
```