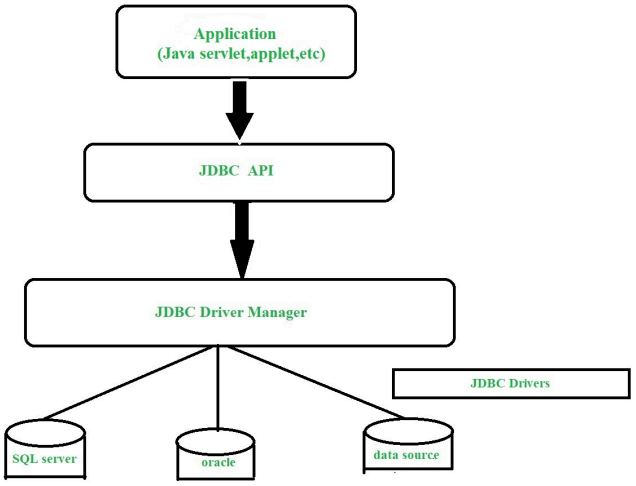
MODULE 3: Back End Development

JDBC

Q.1) write short note on JDBC.

⇒



JDBC (Java Database Connectivity)

JDBC, or Java Database Connectivity, is an API (Application Programming Interface) provided by Java for connecting and interacting with databases.

JDBC allows Java applications to execute SQL statements, retrieve data, and perform various database operations, making it an essential part of Java-based data-driven applications.

Key Features of JDBC

1. **Database Connectivity**: JDBC provides a uniform interface for connecting Java applications to various databases, like MySQL, Oracle, PostgreSQL, and others, enabling the use of SQL to query and update data.
2. **Platform Independence**: JDBC API is part of the Java Standard Library, making it portable across platforms. A single JDBC application can connect to different databases without changes in code.
3. **SQL Support**: JDBC supports SQL, the standard language for database interactions, allowing developers to use SQL queries, updates, and stored procedures directly within Java code.
4. **Error Handling**: JDBC provides classes for handling SQL exceptions and errors, making it easier to manage and troubleshoot database operations.
5. **Transaction Management**: JDBC allows for transaction management, so developers can commit or roll back transactions, ensuring data integrity during multiple database operations.

Components of JDBC

1. **JDBC Drivers**: JDBC uses drivers that establish communication between Java applications and the database. JDBC drivers come in different types:
   - **Type 1**: JDBC-ODBC bridge driver
   - **Type 2**: Native-API driver
   - **Type 3**: Network protocol driver
   - **Type 4**: Thin driver (pure Java)
2. **Connection Interface**: Establishes a connection to the database.
3. **Statement Interface**: Executes SQL queries and updates.
4. **ResultSet Interface**: Represents the result set of a query and allows reading of data returned by the database.
5. **SQLException Class**: Handles SQL-related errors that occur during database operations.

Basic Steps in JDBC

1. **Load the Driver**: Load the appropriate JDBC driver for the database.

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

2. **Establish a Connection**: Connect to the database using `DriverManager.getConnection()`.

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

3. **Create a Statement**: Prepare SQL statements using the `Statement` or `PreparedStatement` interfaces.

   ```
   Statement stmt = con.createStatement();
   ```

4. **Execute SQL Queries**: Execute queries or updates using the `executeQuery` or `executeUpdate` methods.

   ```
   ResultSet rs = stmt.executeQuery("SELECT * FROM users");
   ```

5. **Process Results**: Use `ResultSet` to retrieve data.

```
while (rs.next()) {
   System.out.println("Name: " + rs.getString("name"));
}
```

6. **Close the Connection**: Release database resources.

```
rs.close();
stmt.close();
con.close();
```

Example Code

```
import java.sql.*;

public class JDBCExample {
   public static void main(String[] args) {
      try {
         // Load the driver
         Class.forName("com.mysql.cj.jdbc.Driver");

         // Establish connection
         Connection con = DriverManager.getConnection("jdbc:mysql://local
host:3306/dbname", "username", "password");

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

         // Execute query
         ResultSet rs = stmt.executeQuery("SELECT * FROM users");

         // Process results
         while (rs.next()) {
            System.out.println("Name: " + rs.getString("name"));
         }

         // Close connection
```

```
            rs.close();
            stmt.close();
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Advantages of JDBC

- **Platform Independent**: Works across different operating systems.
- **Database Independence**: Supports various databases with minimal code changes.
- **Simple API**: Easy to use with a minimal learning curve for SQL users.
- **Efficient Data Retrieval**: Offers optimized data access for database interactions.

Summary

JDBC is a core Java API that simplifies database interactions, providing a robust way to connect, query, and update relational databases. Its straightforward design, error handling, and support for transactions make JDBC a powerful tool for building Java applications that require database connectivity.

Q.2) explain the steps to connect Java application to database using JDBC.

⇒

To connect a Java application to a database using JDBC (Java Database Connectivity), follow these essential steps:

1. Load the JDBC Driver

- The first step is to load the appropriate JDBC driver for your database, which acts as a bridge between Java and the database.

- For most modern JDBC drivers, you don't need to explicitly load the driver, as the `DriverManager` will automatically handle it. However, you may still see `Class.forName()` used in some codebases, which loads the driver class explicitly.

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

2. **Establish a Connection**

- Use `DriverManager.getConnection()` to establish a connection to the database.
- This method requires a connection URL, database username, and password.
- The URL typically follows the pattern: `jdbc:[dbms]://[host]:[port]/[database_name]`

Example for MySQL:

```
Connection connection = DriverManager.getConnection("jdbc:mysql://local
host:3306/yourDatabase", "username", "password");
```

3. **Create a Statement**

- A `Statement` object is required to execute SQL queries.
- You can use three types of statements:
  - **Statement**: Used for simple SQL statements without parameters.
  - **PreparedStatement**: Used for parameterized SQL queries, offering security against SQL injection.
  - **CallableStatement**: Used for calling stored procedures in the database.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.SQLException;

public class JDBCExample {
    public static void main(String[] args) {
        // Database credentials
```

```
        String url = "jdbc:mysql://localhost:3306/yourDatabase";
        String username = "username";
        String password = "password";

        // Establishing a connection
        try (Connection connection = DriverManager.getConnection(url, u
sername, password)) {
            System.out.println("Connected to the database successfully!");

            // Create a statement
            Statement statement = connection.createStatement();

            // Execute a query
            String sql = "SELECT * FROM students";
            ResultSet resultSet = statement.executeQuery(sql);

            // Process the result set
            while (resultSet.next()) {
                String name = resultSet.getString("name");
                int age = resultSet.getInt("age");
                System.out.println("Name: " + name + ", Age: " + age);
            }

            // Close resources
            resultSet.close();
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Example of creating a `Statement`:

```
Statement statement = connection.createStatement();
```

4. Execute SQL Queries

- Use the `Statement` object to execute queries using methods such as `executeQuery()` for SELECT statements and `executeUpdate()` for INSERT, UPDATE, or DELETE operations.
- `executeQuery()` returns a `ResultSet` object that contains the data produced by the query.

Example of executing a query:

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM student
s");
```

5. **Process the Results**

- Use the `ResultSet` object to process the data retrieved from the query.
- The `ResultSet` object allows you to navigate through rows and access each column in the result set.

Example:

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

6. **Close Resources**

- Always close the `ResultSet`, `Statement`, and `Connection` objects after using them to free up resources and avoid memory leaks.
- Closing the `Connection` automatically closes any associated `Statement` and `ResultSet` objects.

```
resultSet.close();
statement.close();
connection.close();
```

Full Example Code

Below is a complete example that follows these steps to connect to a MySQL database, execute a query, and display the results.

Summary of Steps

1. **Load JDBC Driver** – `Class.forName()`
2. **Establish Connection** – `DriverManager.getConnection()`
3. **Create Statement** – `connection.createStatement()`
4. **Execute SQL Query** – `statement.executeQuery() or executeUpdate()`
5. **Process Results** – Loop through `ResultSet`
6. **Close Resources** – Close `ResultSet`, `Statement`, and `Connection`

JSP

Q.3) write short note on JSP.

⇒ (same as below)

Q.4) what is JSP? explain life cycle of JSP.

⇒

JSP (JavaServer Pages) is a server-side technology that allows developers to create dynamic web content by embedding Java code within HTML pages.

JSP is built on top of the Java Servlet API and provides an easier way to develop web applications by enabling a combination of Java code and HTML in a single file.

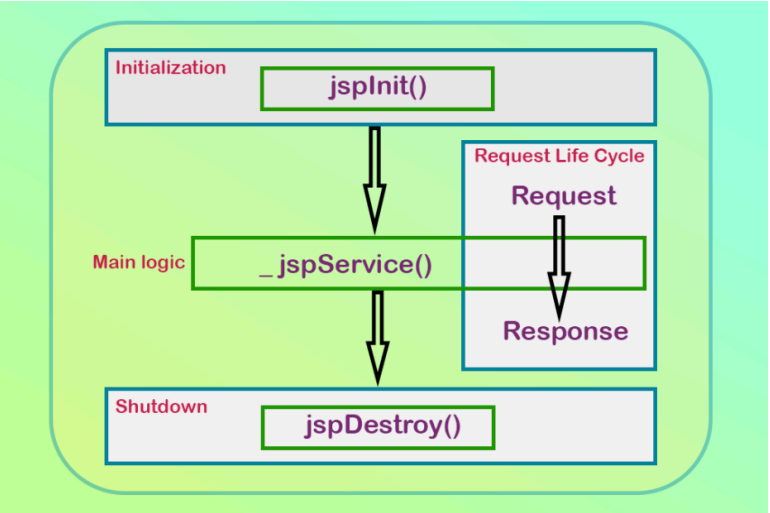
JSP files have the extension `.jsp` and are compiled into servlets by the server, which makes them faster and more efficient for dynamic web content generation.

Key Features of JSP:

- **Ease of Use**: Simplifies creating dynamic web pages by embedding Java code in HTML.

- **Reusable Components**: Supports JavaBeans, custom tags, and JSP tags, promoting reusable components.
- **Separation of Business Logic and Presentation**: Helps separate the presentation layer from business logic by allowing the embedding of backend code in HTML.
- **Platform Independence**: Runs on any web server that supports Java Servlets, making JSP platform-independent.

Life Cycle of a JSP



The lifecycle of a JSP page is similar to that of a servlet, as JSP pages are ultimately compiled into servlets by the server. The life cycle includes several stages:

1. **Translation Phase**: The JSP file is translated into a servlet.
2. **Compilation Phase**: The translated servlet code is compiled into bytecode.
3. **Initialization**: The `jspInit()` method is called for any initialization.
4. **Request Processing**: The `service()` method processes requests.
5. **Loading and Instantiation**: The servlet class is loaded into memory, and an instance of it is created.
6. **Destruction**: The `jspDestroy()` method is called when the JSP is removed from service.

Detailed Life Cycle Stages

1. **Translation Phase**
   - When a JSP page is requested for the first time, the JSP engine translates the `.jsp` file into a Java servlet source file.
   - This involves converting the HTML content and JSP tags into Java code within a servlet class.
2. **Compilation Phase**
   - After translation, the servlet source file is compiled into bytecode, creating a `.class` file.
   - This bytecode can then be executed by the Java Virtual Machine (JVM).
3. **Loading and Instantiation**
   - The compiled servlet is loaded by the server, and an instance of the servlet is created.
   - This instance will serve as the object handling requests for the JSP page.
4. **Initialization (`jspInit()` method)**
   - The `jspInit()` method is called once during the JSP lifecycle to perform any required initialization.
   - Similar to the `init()` method in servlets, this method can be overridden in the JSP to set up resources needed by the JSP (e.g., opening database connections).

```
public void jspInit() {
    // Initialization code (e.g., database connection setup)
}
```

5. **Request Processing (`_jspService()` method)**
   - For each client request, the `_jspService()` method is called to process the request and generate a response.
   - The `_jspService()` method handles HTTP requests (`GET`, `POST`, etc.) and contains the generated code that corresponds to the JSP content.

- It cannot be overridden in JSP, as it is generated by the container during the translation phase.

6. **Destruction (`jspDestroy()` method)**
   - The `jspDestroy()` method is called before the JSP is removed from memory, allowing the page to release any resources it holds.
   - This is similar to the `destroy()` method in servlets, where you can clean up resources (e.g., closing database connections).

```
public void jspDestroy() {
    // Cleanup code
}
```

Q.5) write a JSP program to perform four basic arithmetic operations: addition, subtraction division, and multiplication.

⇒

JSP Program for Basic Arithmetic Operations

To perform addition, subtraction, multiplication, and division, we can use form inputs in HTML to get values from the user and display the results after performing operations in JSP.

Code: `arithmetic.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pag
eEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Basic Arithmetic Operations</title>
</head>
<body>
```

```
    <h2>Enter Two Numbers for Arithmetic Operations</h2>
    <form method="post">
        <label>Number 1:</label>
        <input type="number" name="num1" required><br><br>
        <label>Number 2:</label>
        <input type="number" name="num2" required><br><br>
        <input type="submit" value="Calculate">
    </form>

    <%
        // Get parameters from form submission
        String num1Str = request.getParameter("num1");
        String num2Str = request.getParameter("num2");

        // Check if numbers are provided
        if(num1Str != null && num2Str != null) {
            try {
                // Parse input values
                double num1 = Double.parseDouble(num1Str);
                double num2 = Double.parseDouble(num2Str);

                // Perform arithmetic operations
                double addition = num1 + num2;
                double subtraction = num1 - num2;
                double multiplication = num1 * num2;
                double division = num2 != 0 ? num1 / num2 : 0;

                // Display results
    %>
                <h3>Results:</h3>
                <p>Addition (num1 + num2): <%= addition %></p>
                <p>Subtraction (num1 - num2): <%= subtraction %></p>
                <p>Multiplication (num1 * num2): <%= multiplication %></p>
                <p>Division (num1 / num2): <%= num2 != 0 ? division : "Cannot d
ivide by zero" %></p>
    <%
            } catch (NumberFormatException e) {
                out.println("<p>Please enter valid numbers.</p>");
```

```
        }
      }
    %>
</body>
</html>
```

Explanation

- **Arithmetic Operations Program**:
  - Takes two numbers as input and performs addition, subtraction, multiplication, and division.
  - It checks for a valid division operation to prevent division by zero.
  - Outputs the result of each operation.
- **System Date and Time Program**:
  - Retrieves the current system date and time using `java.util.Date`.
  - Displays the date and time in the format provided by `Date.toString()`.

Q.6) write a program in JSP to display system data and time.

JSP Program to Display System Date and Time

The following program retrieves and displays the current system date and time.

Code: `dateAndTime.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pag
eEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Display System Date and Time</title>
</head>
<body>
    <h2>Current System Date and Time</h2>
    <%
        // Get current date and time
        java.util.Date currentDate = new java.util.Date();
    %>
    <p>The current date and time is: <%= currentDate.toString() %></p>
</body>
</html>
```

Explanation

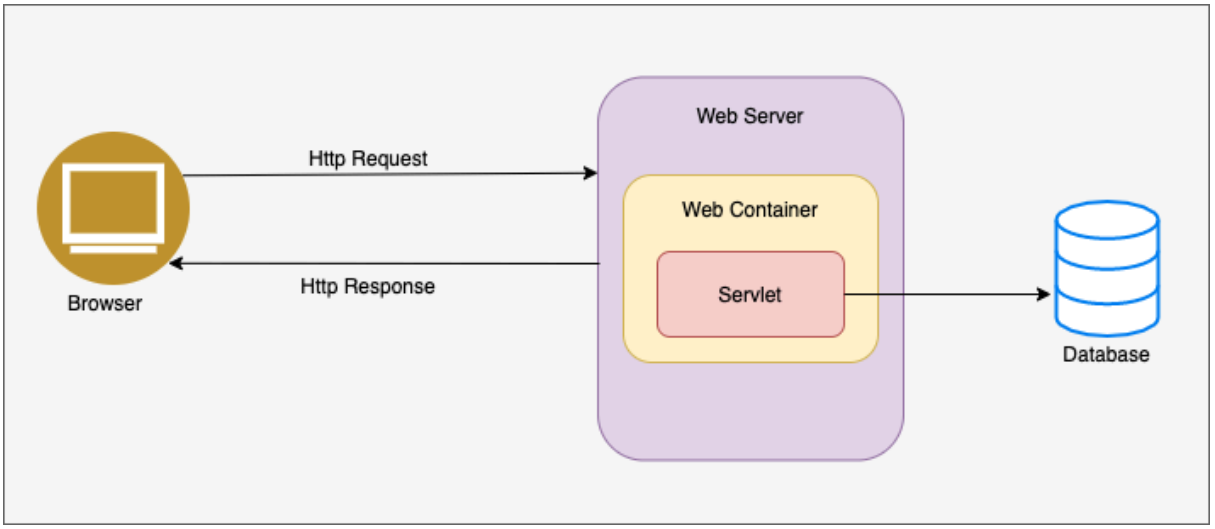
- **System Date and Time Program**:
  - Retrieves the current system date and time using `java.util.Date`.
  - Displays the date and time in the format provided by `Date.toString()`.

SERVLET & cookies

Q.7) draw an explain servlet architecture and its life cycle.

⇒

Servlet Architecture and Life Cycle



JAVA SERVLET ARCHITECHTURE

Java Servlet are server side programs that run on a web or application middleware server and act as a middle layer between a request coming from a web browser or other HTTP client and databases or applications like Java Beans on the HTTP server.

Using servlets input can be collected from users through webpages forms(client), present records from a database or another source, and create webpage dynamically.

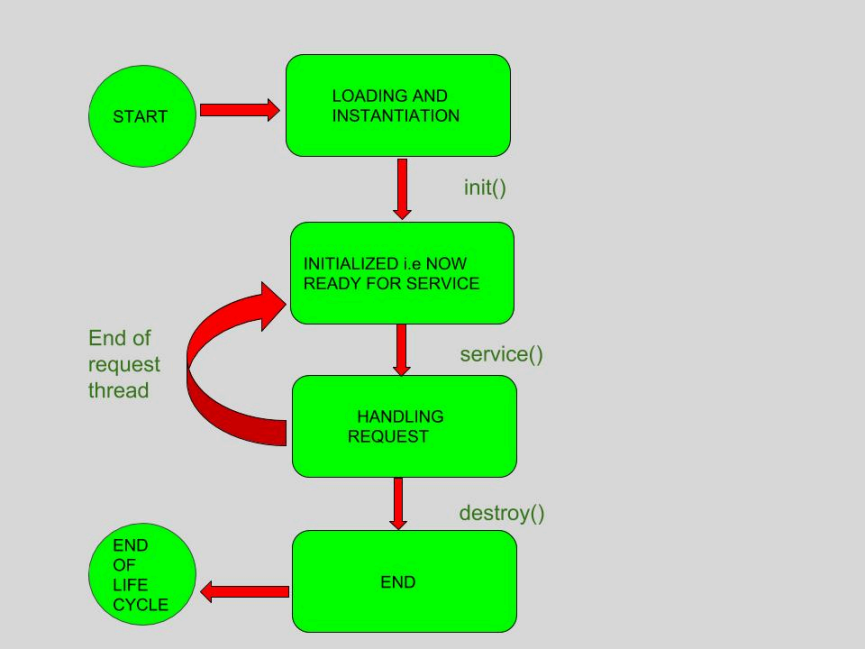
Java Servlet Architecture

The servlet architecture follows a request-response model between the client and server:

1. **Client Requests**: A client (usually a web browser) sends a request to the web server, often via HTTP.
2. **Web Server**: The server receives this request and forwards it to the servlet container if the request is meant for a servlet.
3. **Servlet Container**: This is a part of the server responsible for managing servlets. It performs various functions such as loading, initializing, invoking, and destroying servlets.
4. **Servlet**: The servlet processes the request (often involving interaction with databases or other backend resources) and generates a response.
5. **Response to Client**: The response, usually HTML, is sent back to the client through the web server.

6. **Backend Interaction**: Servlets often interact with other resources like databases, other servers, and files for processing.

Servlet Life Cycle



A servlet's life cycle defines the stages it goes through from creation to termination, managed by the servlet container. The life cycle has five main stages:

1. **Loading and Instantiation**
    - The servlet container loads the servlet class and creates an instance of the servlet.
    - This is done only once when the servlet is first requested or when the server starts up, depending on the server configuration.
2. **Initialization (`init` method)**
    - The container calls the `init()` method only once, after instantiation.
    - The `init()` method initializes resources needed by the servlet, such as database connections or configuration settings.
    - This method is called once in the servlet's lifetime.

```
public void init(ServletConfig config) throws ServletException {
    // Initialization code, like establishing database connections
```

```
}
```

3. **Request Handling (`service` method)**
   - For each client request, the `service()` method is called.
   - The `service()` method then delegates requests based on HTTP methods (`doGet()`, `doPost()`, etc.).
   - This method may be called multiple times throughout the servlet's life, handling multiple client requests concurrently.

```
public void service(HttpServletRequest req, HttpServletResponse res) t
hrows ServletException, IOException {
    // Processing requests, such as reading user input or querying datab
ases
}
```

4. **Destruction (`destroy` method)**
   - When the servlet is no longer needed, the container calls the `destroy()` method once.
   - This stage is for cleanup, such as releasing database connections or closing files.
   - The servlet is then garbage collected by the JVM.

```
public void destroy() {
    // Cleanup code, like closing database connections
}
```

1. **Servlet Instantiation**: Loads and creates a servlet instance.
2. **Initialization (init)**: Initializes resources.
3. **Request Handling (service)**: Processes each request with service, doGet, or doPost.
4. **Destruction (destroy)**: Cleans up resources.

Summary

- **Servlet Container** manages the life cycle.

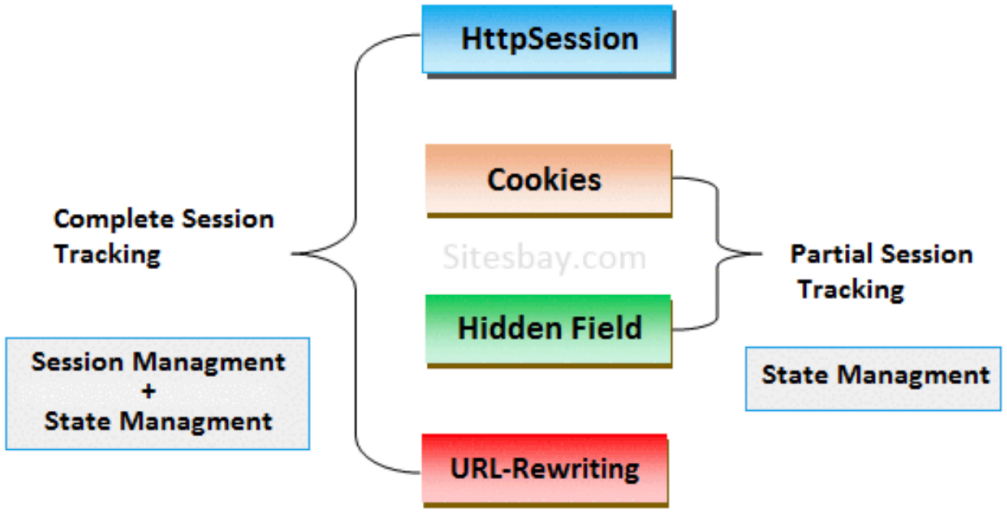
- **init()** and **destroy()** are called only once per servlet instance.
- **service()** is called for each request, making the servlet reusable and efficient for high-volume traffic.

This architecture enables Java-based dynamic content on web servers, handling multiple requests with minimal instantiation and efficient resource management.

Q.8) explain the servlet life cycle with neat diagram.

⇒ (same as above)

Q.9) What is Session Tracking, and How is Session Tracking Achieved Using Cookies?



session tracking in servlet

Session Tracking is the method of maintaining a "session" between the client and the server, allowing the server to remember specific data about the user, such as login details, preferences, or items in a shopping cart, across multiple requests.

HTTP, by default, is a stateless protocol, meaning each request is treated independently, without memory of prior interactions.

Session tracking makes it possible to build interactive and personalized web applications by bridging this gap.

Session Tracking Using Cookies:
Cookies are the most common way to manage sessions. A **cookie** is a small text file that the server sends to the client's browser to store data.

This cookie contains a unique session ID or information that identifies the client.

Each time the client sends a new request, it automatically includes the cookie, which allows the server to retrieve the session data associated with that session ID.

Steps for Session Tracking Using Cookies:

1. **Creating and Sending the Cookie**:
    - When a client (user) makes a request, the server generates a unique session ID and sends it to the client in the form of a cookie.
    - Example:

```
Cookie sessionCookie = new Cookie("sessionID", "XYZ12345");
sessionCookie.setMaxAge(3600); // Cookie expires after 1 hour
response.addCookie(sessionCookie);
```

2. **Storing the Cookie**:
    - The client's browser stores the cookie with the session ID. Cookies can be stored as session cookies (deleted when the browser closes) or persistent cookies (stored until a specific expiration time).
3. **Retrieving the Cookie on Subsequent Requests**:
    - When the user sends another request, the browser automatically includes the cookie in the request, allowing the server to recognize the user.
    - Example of retrieving cookies in a servlet:

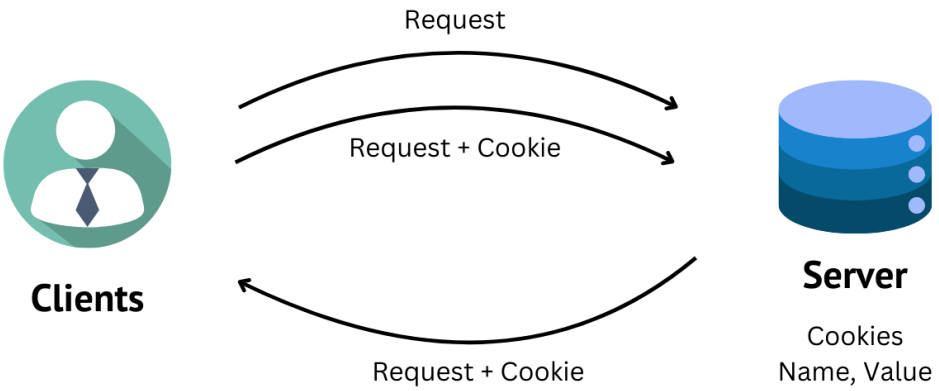
```
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    for (Cookie cookie : cookies) {
        if ("sessionID".equals(cookie.getName())) {
            String sessionId = cookie.getValue();
            // Use session ID for session tracking
        }
    }
}
```

4. **Session Management Using the Cookie Data**:
    - Once the server identifies the session ID from the cookie, it retrieves the associated session data, allowing it to maintain a continuous experience for the user.

Q.10) What Are Cookies, and How Do Cookies Work in Servlets?

Cookies are small data files stored on the client's device by the server. They allow the server to "remember" specific data about a client between HTTP requests.

Cookies are often used for session tracking, but they can also store preferences, login information, or any other data that may need to persist between user sessions.



How Cookies Work in Servlets:

1. **Creating a Cookie**:
   - A servlet can create a cookie using the `javax.servlet.http.Cookie` class. Once created, the servlet can attach the cookie to the HTTP response using `response.addCookie()`.
   - Example:

```
Cookie userCookie = new Cookie("userID", "12345");
userCookie.setMaxAge(60 * 60 * 24); // Set cookie to expire in 24 hours
response.addCookie(userCookie);
```

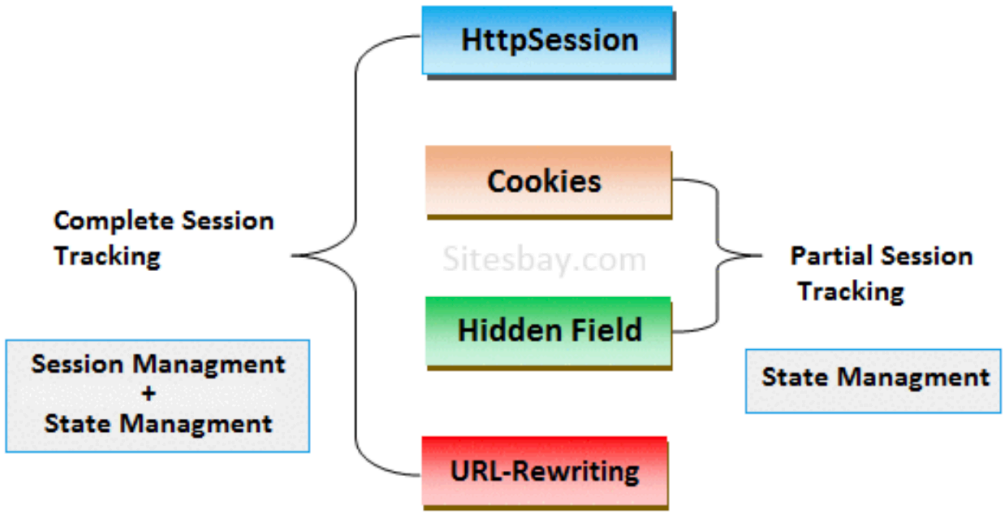
2. **Sending Cookies**:
   - The cookie is sent to the client as part of the HTTP response header, allowing the browser to store it.
3. **Storing Cookies**:
   - The client's browser stores the cookie until it expires or is manually deleted. If set as a persistent cookie, it remains even after the browser is closed.
4. **Retrieving Cookies**:
   - For each subsequent request from the client, the browser automatically sends all cookies related to the domain, which the server can retrieve

using `request.getCookies()`.

- Example:

```
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    for (Cookie cookie : cookies) {
        System.out.println("Cookie Name: " + cookie.getName());
        System.out.println("Cookie Value: " + cookie.getValue());
    }
}
```

Q.11) List and Explain Session Tracking Techniques



session tracking in servlet

There are four main techniques for session tracking in servlets:

1. **Cookies**:
    - **Description**: A small piece of data stored on the client's device and sent back to the server with each request.
    - **Advantages**: Supported by all browsers, allows persistent sessions.
    - **Disadvantages**: Storage is limited, can be disabled by users, not as secure for sensitive data.

- **Example**: Session tracking for login credentials or shopping cart items.

2. **URL Rewriting**:
   - **Description**: The server appends the session ID or other session information to the URL of each link within the application. This way, when the user clicks a link, the session ID is sent back to the server.
   - **Advantages**: Works even if cookies are disabled.
   - **Disadvantages**: Session ID is visible in the URL, potentially less secure, and requires each URL to be rewritten manually.
   - **Example**: `https://example.com/page?sessionId=ABC123`
   - **Usage**:

```
String url = response.encodeURL("<https://example.com/home>");
```

3. **Hidden Form Fields**:
   - **Description**: Stores session data in hidden fields within HTML forms. Each time the user submits the form, the session data is sent along with it.
   - **Advantages**: Effective for applications that use forms for data submission and do not rely on cookies.
   - **Disadvantages**: Limited to form-based navigation and does not work well with applications that use a variety of page requests.
   - **Example**:

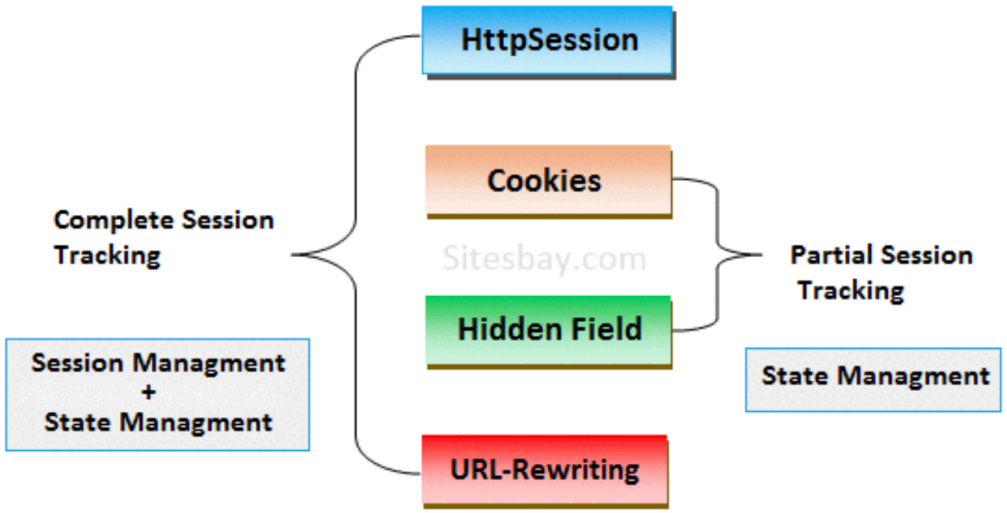
```
<form action="nextPage" method="post">
  <input type="hidden" name="sessionID" value="XYZ123">
  <input type="submit" value="Continue">
</form>
```

4. **HTTP Session API**:
   - **Description**: A built-in session management mechanism provided by the Java Servlet API. It is a more secure and convenient way to handle sessions.

- **Advantages**: Provides automatic management of session IDs, large data storage, and server-side storage, which is more secure.
- **Disadvantages**: Depends on server memory, requires server-side resources, and can be complex in distributed systems.
- **Example**:

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

Q.12) What is Session in Servlet, and List Different Ways to Handle It?



session tracking in servlet

A **session** in servlets is a unique interaction between a client and server that allows data to be maintained across multiple requests from the client. The session data typically includes user-specific information, like login status, preferences, or shopping cart items.

Different Ways to Handle Sessions in Servlets:

1. **Using `HttpSession` API**:
   - **Method**: Java provides the `HttpSession` API, which allows you to create and manage sessions by storing and retrieving data for each unique client session.

- **Code Example**:

```
HttpSession session = request.getSession();
session.setAttribute("username", "Alice");
String username = (String) session.getAttribute("username");
```

2. **Using Cookies**:
   - **Method**: Session IDs can be stored as cookies on the client-side, which can be retrieved to identify the user in subsequent requests.
   - **Code Example**:

```
Cookie sessionCookie = new Cookie("sessionID", "XYZ123");
response.addCookie(sessionCookie);
```

3. **Using URL Rewriting**:
   - **Method**: Session ID is appended to each URL, allowing the server to track the session without relying on cookies.
   - **Code Example**:

```
String encodedURL = response.encodeURL("<https://example.com/page>");
```

4. **Hidden Form Fields**:
   - **Method**: A hidden field in each form can contain the session ID or user-specific data.
   - **Code Example**:

```
<input type="hidden" name="sessionID" value="XYZ123">
```

These session tracking methods in servlets allow for versatile session management tailored to the needs and requirements of each web application. Each method has its unique advantages, allowing you to choose the most appropriate approach based on application requirements, security considerations, and user preferences.