**Basic Term to know before learning Servlet**

**1. HTTP (Hypertext Transfer Protocol)**

**HTTP** is a protocol used for transmitting hypertext requests and information on the internet. It's the foundation of any data exchange on the Web, and it is a protocol used by the World Wide Web.

* **Methods of HTTP:**
  + **GET**: Requests data from a specified resource. Example: Fetching a webpage.
  + **POST**: Submits data to be processed to a specified resource. Example: Submitting a form.
  + **PUT**: Updates a current resource with new data.
  + **DELETE**: Deletes the specified resource.
  + **HEAD**: Similar to GET, but it only requests the headers without the body.
  + **OPTIONS**: Returns the HTTP methods that the server supports for the specified URL.
  + **PATCH**: Applies partial modifications to a resource.

**2. Client and Server**

* **Client**: The client is the end user’s device (like a computer or smartphone) that sends requests to a server over the network. It could be a web browser, a mobile app, or any software capable of making network requests.
* **Server**: The server is a powerful computer that provides resources, data, services, or programs to the client over a network. When a client sends a request, the server processes it and sends back the appropriate response.

**Example**: When you type a URL in your browser (client), it sends an HTTP request to the server where the website is hosted. The server processes this request and sends back the HTML, CSS, and JavaScript files to your browser, which then renders the webpage.

**3. Static Web Page vs. Dynamic Web Page**

* **Static Web Page**: A static web page is a simple HTML page that doesn't change content or layout with every request. The content is fixed and the same for all users.

**Example**: An "About Us" page of a website with just text and images that doesn’t change until someone manually updates it.

* **Dynamic Web Page**: A dynamic web page is generated on-the-fly and can show different content and layout based on user interaction, database queries, or other factors. It usually involves server-side scripting.

**Example**: A user’s profile page on a social media site, where the content changes based on the logged-in user’s data.

**4. URL (Uniform Resource Locator)**

A **URL** is the address used to access resources on the internet. It specifies the location of a resource and the protocol to retrieve it.

**Example**:

https://www.example.com/index.html

* https: The protocol.
* www.example.com: The domain name or IP address of the server.
* /index.html: The specific resource on the server.

**5. URI (Uniform Resource Identifier)**

A **URI** is a string of characters that unambiguously identifies a particular resource. It can be further classified into:

* **URL**: Identifies a resource by its location.
* **URN**: Identifies a resource by its name within a given namespace.

**Example**:

URN: urn:isbn:0451450523

In this example, urn:isbn:0451450523 is a URI that refers to a book by its ISBN.

**6. Authentication and Authorization**

* **Authentication**: The process of verifying the identity of a user or entity. It's the act of confirming whether someone is who they claim to be.

**Example**: Entering a username and password to log into a website. The system checks if the credentials match an account in its database.

* **Authorization**: The process of determining what an authenticated user is allowed to do. It controls access to resources based on the user's permissions.

**Example**: After logging in, a user might be authorized to view their profile, but not to access the admin panel of a website.

**Example Scenario:**

When you log into an online banking application:

1. **Authentication** verifies that you are the account holder by checking your credentials (e.g., username and password).
2. **Authorization** determines what actions you can perform (e.g., viewing your balance, transferring money).

**Q. How many we is there to create servlet ?**

**1. Implementing Servlet Interface Directly**

* **Description:** This involves implementing the Servlet interface directly and overriding its core methods: init(), service(), destroy(), getServletConfig(), and getServletInfo().

**2. Extending GenericServlet Class**

* **Description:** The GenericServlet class is an abstract class that implements the Servlet interface and provides default implementations for most of the methods. You only need to override the service() method.

**3. Extending HttpServlet Class**

* **Description:** The HttpServlet class is a subclass of GenericServlet that provides methods specifically for handling HTTP requests (e.g., doGet(), doPost()). It simplifies handling HTTP-specific functionality.

### Q. Difference Between GenericServlet and HttpServlet?

#### 1. **Purpose:**

* **GenericServlet:**
  + It is a generic, protocol-independent servlet class. It provides a generic implementation of the Servlet interface, which can be extended to create servlets that handle any kind of protocol, not just HTTP.
  + It is useful when you need to create servlets that handle non-HTTP protocols like FTP, or custom protocols.
* **HttpServlet:**
  + It is a subclass of GenericServlet specifically designed to handle HTTP requests. It provides methods tailored for handling HTTP-specific tasks and simplifies the creation of HTTP servlets.
  + It is the most commonly used servlet class for web applications as it handles HTTP requests and responses.

#### 2. **Methods:**

* **GenericServlet Methods:**
  + init(): Initializes the servlet. This method is called once when the servlet is first loaded into memory.
  + service(ServletRequest req, ServletResponse res): Processes requests and generates responses. This method is called for every request the servlet receives.
  + destroy(): Cleans up resources before the servlet is destroyed.
* **HttpServlet Methods:**
  + doGet(HttpServletRequest req, HttpServletResponse res): Handles HTTP GET requests. This method is used to process form submissions and query string data.
  + doPost(HttpServletRequest req, HttpServletResponse res): Handles HTTP POST requests. This method is used to process data sent in the body of the request.
  + doPut(HttpServletRequest req, HttpServletResponse res): Handles HTTP PUT requests, typically used to update resources.
  + doDelete(HttpServletRequest req, HttpServletResponse res): Handles HTTP DELETE requests, typically used to delete resources.
  + doHead(HttpServletRequest req, HttpServletResponse res): Handles HTTP HEAD requests, used to retrieve metadata.
  + doOptions(HttpServletRequest req, HttpServletResponse res): Handles HTTP OPTIONS requests, used to retrieve allowed HTTP methods.
  + doTrace(HttpServletRequest req, HttpServletResponse res): Handles HTTP TRACE requests, used for diagnostic purposes.

### Summary of Differences:

1. **Protocol Handling:**
   * GenericServlet: Protocol-independent. Suitable for custom protocols.
   * HttpServlet: Designed specifically for handling HTTP requests and responses.
2. **Request Handling Methods:**
   * GenericServlet: Single method service(ServletRequest req, ServletResponse res) for handling requests.
   * HttpServlet: Multiple methods (doGet, doPost, doPut, doDelete, doHead, doOptions, doTrace) for handling different HTTP methods.

### Example Code:

#### Using GenericServlet:

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.GenericServlet;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.ServletException;

public class MyGenericServlet extends GenericServlet {

@Override

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 from GenericServlet</h1>");

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

}

}

#### Using HttpServlet:

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.ServletException;

public class MyHttpServlet extends HttpServlet {

@Override

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 from HttpServlet</h1>");

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

}

}

In summary, HttpServlet is an extension of GenericServlet tailored for HTTP requests, making it more suitable for web applications that handle HTTP. GenericServlet can be used for more general-purpose servlets that deal with various protocols.

**Q. When implementing the Servlet interface directly, you must define the following core methods:**

**1. init(ServletConfig config)**

* **Purpose:** Initializes the servlet. This method is called once when the servlet is first loaded into memory.
* **Parameters:** ServletConfig object, which contains initialization parameters and the servlet's context.
* **Signature:**

void init(ServletConfig config) throws ServletException;

**2. service(ServletRequest req, ServletResponse res)**

* **Purpose:** Handles requests and generates responses. This method is called for each request the servlet receives.
* **Parameters:**
  + ServletRequest req: The request object that contains the client request data.
  + ServletResponse res: The response object that the servlet uses to return data to the client.
* **Signature:**

void service(ServletRequest req, ServletResponse res) throws ServletException, IOException;

**3. destroy()**

* **Purpose:** Cleans up resources before the servlet is destroyed. This method is called once when the servlet is about to be removed from service.
* **Signature:**

void destroy();

**4. getServletConfig()**

* **Purpose:** Returns the ServletConfig object, which provides initialization parameters and servlet context information.
* **Signature:**

ServletConfig getServletConfig();

**5. getServletInfo()**

* **Purpose:** Returns information about the servlet, such as author, version, and copyright.
* **Signature:**

String getServletInfo();

**Example Implementation:**

Here is an example of implementing the Servlet interface directly:

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.Servlet;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

public class MyServlet implements Servlet {

private ServletConfig config;

@Override

public void init(ServletConfig config) throws ServletException {

this.config = config;

// Initialization code here

}

@Override

public ServletConfig getServletConfig() {

return config;

}

@Override

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 from MyServlet</h1>");

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

}

@Override

public String getServletInfo() {

return "MyServlet v1.0";

}

@Override

public void destroy() {

// Cleanup code here

}

}

**Summary:**

When you implement the Servlet interface directly, you need to define the core methods to handle servlet lifecycle and request processing. The GenericServlet class simplifies this by providing default implementations of init, service, and destroy methods, while the HttpServlet class adds additional functionality specifically for handling HTTP requests.

**Q. The lifecycle of a servlet in Java is managed by the servlet container (e.g., Apache Tomcat, Jetty). The servlet lifecycle consists of several key phases, which are managed through specific lifecycle methods provided by the Servlet interface and its subclasses like GenericServlet and HttpServlet. Here’s a detailed explanation of each phase:**

### 1. ****Loading and Instantiation****

* **When it Happens:** When the servlet container starts or when the servlet is first requested.
* **Details:** The servlet container loads the servlet class into memory and creates an instance of it.
* **Purpose:** To prepare the servlet for handling requests.
* **Method Involved:** No specific method is involved directly in this phase, but it is followed by the init() method.

### 2. ****Initialization****

* **When it Happens:** After the servlet instance is created and before it starts handling requests.
* **Details:** The servlet container calls the init() method of the servlet.
* **Purpose:** To perform any one-time initialization tasks (e.g., reading configuration parameters, establishing database connections).
* **Method Involved:** init(ServletConfig config) method.
* **Code Example:**

@Override

public void init(ServletConfig config) throws ServletException {

super.init(config);

// Initialization code here

}

### 3. ****Request Handling****

* **When it Happens:** For each request made by a client (e.g., browser) to the servlet.
* **Details:** The servlet container calls the service() method, passing ServletRequest and ServletResponse objects.
* **Purpose:** To process the request and generate a response.
* **Method Involved:** service(ServletRequest req, ServletResponse res) method.
* **Code Example:**

@Override

public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {

// Code to handle request and generate response

}

### 4. ****Destruction****

* **When it Happens:** When the servlet container is shutting down or when the servlet is being unloaded.
* **Details:** The servlet container calls the destroy() method.
* **Purpose:** To perform any cleanup tasks (e.g., releasing resources, closing database connections).
* **Method Involved:** destroy() method.
* **Code Example:**

@Override

public void destroy() {

// Cleanup code here

super.destroy();

}

### Summary of Lifecycle Methods:

1. **init(ServletConfig config)**
   * Called once when the servlet is first loaded.
   * Used for initialization tasks.
   * Parameters:
     + ServletConfig: Contains initialization parameters.
2. **service(ServletRequest req, ServletResponse res)**
   * Called for each request to the servlet.
   * Used to handle the request and generate a response.
   * Parameters:
     + ServletRequest: Provides request data.
     + ServletResponse: Used to send response data.
3. **destroy()**
   * Called once when the servlet is being removed from service.
   * Used for cleanup tasks.

### Servlet Lifecycle Diagram

1. **Loading and Instantiation** ➔
2. **Initialization (init())** ➔
3. **Request Handling (service())** ➔
4. **Destruction (destroy())**

### Example Lifecycle in Action

1. **Loading and Instantiation:**

// Servlet class is loaded and instantiated by the container.

1. **Initialization:**

@Override

public void init(ServletConfig config) throws ServletException {

// Initialization code

System.out.println("Servlet Initialized");

}

1. **Request Handling:**

@Override

public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {

PrintWriter out = res.getWriter();

out.println("Hello World");

}

1. **Destruction:**

@Override

public void destroy() {

// Cleanup code

System.out.println("Servlet Destroyed");

}

Understanding the servlet lifecycle helps in managing resources efficiently and ensuring that servlets perform optimally throughout their lifespan.

**Q. Who is here work as container?**

In the context of Java Servlets, the **servlet container** (also known as a servlet engine or servlet container) is a part of a web server or application server that manages the lifecycle of servlets and provides the environment necessary for servlets to operate.

Here’s a more detailed breakdown of the servlet container's responsibilities:

### Responsibilities of a Servlet Container

1. **Servlet Lifecycle Management**
   * **Loading and Instantiating:** Loads servlet classes into memory and creates instances.
   * **Initialization:** Calls the init() method to initialize the servlet.
   * **Request Handling:** Calls the service() method to handle client requests.
   * **Destruction:** Calls the destroy() method for cleanup before unloading the servlet.
2. **Request and Response Management**
   * **Request Parsing:** Parses incoming requests, including HTTP headers and parameters.
   * **Response Handling:** Formats and sends responses back to clients.
3. **Resource Management**
   * **Session Management:** Manages user sessions and session attributes.
   * **Concurrency Management:** Handles concurrent requests to the same servlet.
4. **Configuration and Deployment**
   * **Configuration:** Reads configuration details from web.xml or annotations (like @WebServlet).
   * **Deployment:** Manages the deployment of web applications.
5. **Integration with Web Server**
   * **Communication:** Communicates with the web server to handle HTTP requests and responses.

### Examples of Servlet Containers

1. **Apache Tomcat:**
   * **Description:** A widely used open-source servlet container and web server.
   * **Usage:** Supports servlets and JSPs (JavaServer Pages).
2. **Jetty:**
   * **Description:** A lightweight, high-performance servlet container and web server.
   * **Usage:** Often used for embedding in applications.
3. **IBM WebSphere:**
   * **Description:** A robust enterprise-level application server.
   * **Usage:** Provides extensive features for large-scale enterprise applications.
4. **Red Hat JBoss EAP:**
   * **Description:** An open-source Java EE application server.
   * **Usage:** Offers comprehensive support for Java EE specifications.
5. **GlassFish:**
   * **Description:** An open-source application server for Java EE.
   * **Usage:** Known for its full support of Java EE specifications.

### Servlet Container Workflow

1. **Startup:**
   * The container initializes, loads servlets, and reads configuration settings.
2. **Request Processing:**
   * The container receives requests from clients, delegates them to the appropriate servlets, and sends responses.
3. **Shutdown:**
   * The container performs cleanup tasks and shuts down, calling the destroy() method of servlets.

The servlet container acts as the bridge between web clients (like browsers) and the servlets, managing the overall lifecycle and providing necessary services to servlets.

### Q. What is Servlet Collaboration?

Servlet collaboration refers to how servlets communicate or work together within a web application to perform a common task. It enables multiple servlets to share data, forward requests, or include responses.

# **With the help of RequestDispatcher in Servlet Interface**

The RequestDispatcher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp. This interface can also be used to include the content of another resource also. It is one of the way of servlet collaboration.

**There are two methods defined in the RequestDispatcher interface.**

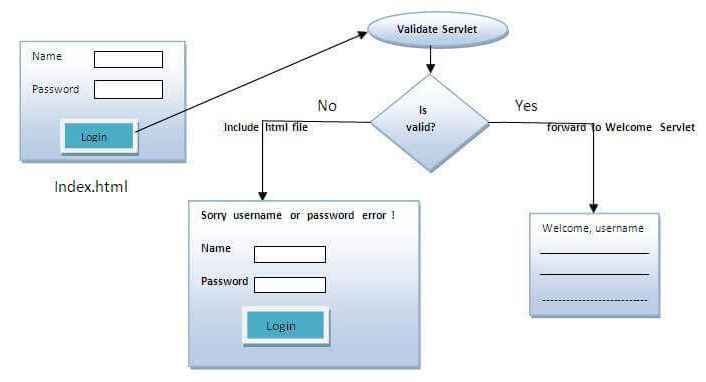
### 1. forward(ServletRequest request, ServletResponse response)

* **Purpose**: This method is used to forward a client's request from one servlet to another resource (such as another servlet, a JSP file, or an HTML file) **on the server**.
* **How it works**:
  + When the forward() method is called, the original servlet hands off the request to another resource.
  + The response is generated by the forwarded resource (e.g., servlet or JSP), and the client never knows that the forwarding happened.
  + **Important**: The browser URL remains the same as the original servlet; the user won't see the redirected path.
* **Key Points**:
  + The forwarded resource completely handles the request.
  + The original servlet does not output anything; only the forwarded resource's response is sent back to the client.

### 2. include(ServletRequest request, ServletResponse response)

* **Purpose**: This method is used to **include** the content of another resource (such as a servlet, JSP page, or HTML file) **within the response** of the current servlet.
* **How it works**:
  + When the include() method is called, the response of another resource (e.g., a servlet or JSP) is embedded inside the response of the current servlet.
  + **Important**: Unlike forward(), control returns back to the original servlet after the inclusion is complete.
* **Key Points**:
  + Both the original servlet and the included resource contribute to the final output sent to the client.
  + The original servlet sends its response, and the included servlet adds its content within that response.

**Below is the example of forward and include working**

**Q. What is SendRedirect in servlet?**

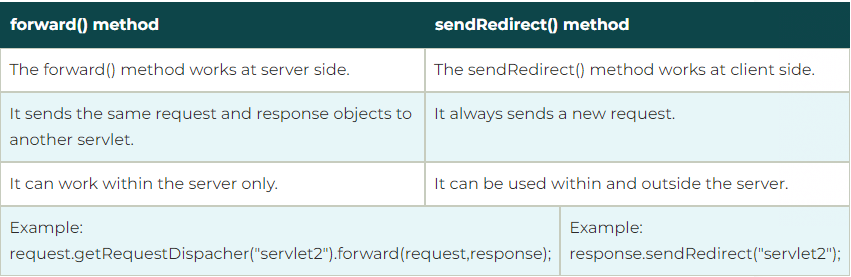
The **sendRedirect()** method of **HttpServletResponse** interface can be used to redirect response to another resource, it may be servlet, jsp or html file.

It accepts relative as well as absolute URL.

It works at client side because it uses the url bar of the browser to make another request. So, it can work inside and outside the server.

**Q**.**Difference between forward() and sendRedirect() method**

There are many differences between the forward() method of RequestDispatcher and sendRedirect() method of HttpServletResponse interface. They are given below:



### Q. ****Session Management Techniques in Servlets****

Session management is vital in web applications to maintain user state and data across multiple requests. Since HTTP is a stateless protocol, various techniques like **cookies**, **hidden form fields**, **URL rewriting**, and **HTTP sessions** are used to manage sessions in servlet-based web applications.

Below are detailed notes on each session management technique, including how to implement them using **constructors**, **classes**, **interfaces**, and **methods**. We'll also discuss their **advantages** and **disadvantages**.

### ****1. Cookies****

A **cookie** is a small piece of data stored on the client-side (browser). The server sends cookies to the browser, and the browser sends them back with every request, enabling the server to identify the user and manage sessions.

#### **Class and Constructor for Cookies**

public class CookieExample {

public CookieExample() {

// Constructor to initialize any necessary objects

}

public void createCookie(HttpServletResponse response) {

Cookie sessionCookie = new Cookie("username", "JohnDoe");

sessionCookie.setMaxAge(60 \* 60); // 1 hour

response.addCookie(sessionCookie);

}

public Cookie[] getCookies(HttpServletRequest request) {

return request.getCookies();

}

}

#### **Advantages of Cookies**

* **Simplest technique of maintaining the state**: Easy to implement and widely supported.
* **Cookies are maintained at the client-side**: Reduces the server's memory usage.
* **Persistent storage**: Cookies can last after a session ends if they are persistent.
* **Wide browser support**: Supported by almost all modern browsers.

#### **Disadvantages of Cookies**

* **Will not work if cookies are disabled from the browser**: Some users may block cookies.
* **Security risks**: Cookies can be intercepted (via XSS attacks) or manipulated unless secured.
* **Limited storage**: Maximum cookie size is 4KB.
* **Only textual information can be set in Cookie objects**: Only strings can be stored.

### ****2. Hidden Form Fields****

**Hidden form fields** include session-related data as hidden input elements in forms. The session data is transferred when the form is submitted to the server.

#### **Class and Method for Hidden Form Fields**

public class HiddenFormFieldExample {

public HiddenFormFieldExample() {

// Constructor

}

public String generateHiddenFieldForm() {

return "<form action='someServlet' method='POST'>" +

"<input type='hidden' name='sessionId' value='12345' />" +

"<input type='submit' value='Submit' />" +

"</form>";

}

}

#### **Advantages of Hidden Form Fields**

* **It will always work whether cookies are disabled or not**: Does not rely on client-side cookies.
* **Simple to implement**: Easy to add hidden fields in forms.
* **No browser storage**: The data is embedded in the form and passed along with the form submission.

#### **Disadvantages of Hidden Form Fields**

* **It is maintained at the server-side**: Relies on the server to manage session data.
* **Extra form submission is required on each page**: Only works with forms and requires submission.
* **Only textual information can be used**: Can only handle string data.
* **Less secure**: Hidden fields can be viewed and modified using browser tools.

### ****3. URL Rewriting****

**URL rewriting** involves appending session-related data to the URL. The session ID or other session data is passed through URLs between different requests.

#### **Class and Method for URL Rewriting**

public class URLRewritingExample {

public URLRewritingExample() {

// Constructor

}

public String rewriteURL(HttpServletRequest request, HttpServletResponse response, String url) {

String sessionId = request.getSession().getId();

return response.encodeURL(url + "?sessionId=" + sessionId);

}

}

#### **Advantages of URL Rewriting**

* **It will always work whether cookies are disabled or not**: Does not rely on cookies.
* **Extra form submission is not required on each page**: Works with hyperlinks instead of forms.
* **Browser independent**: The session information is passed in the URL.

#### **Disadvantages of URL Rewriting**

* **It will work only with links**: Cannot work with forms and AJAX.
* **It can send only textual information**: Limited to string data.
* **Security risk**: The session ID is visible in the URL, making it vulnerable.
* **Messy URLs**: Session data makes URLs longer and more complex.

### ****4. HTTP Sessions****

**HTTP sessions** store session data on the server. A unique session ID is assigned to each user, which is sent to the client either through cookies or URL rewriting.

#### **Class, Constructor, and Methods for HTTP Sessions**

public class SessionExample {

public SessionExample() {

// Constructor

}

public void createSession(HttpServletRequest request) {

HttpSession session = request.getSession();

session.setAttribute("username", "JohnDoe");

session.setMaxInactiveInterval(60); // Set session expiry to 1 minute

}

public String getSessionData(HttpServletRequest request) {

HttpSession session = request.getSession(false);

if (session != null) {

return (String) session.getAttribute("username");

} else {

return "Session expired or not found.";

}

}

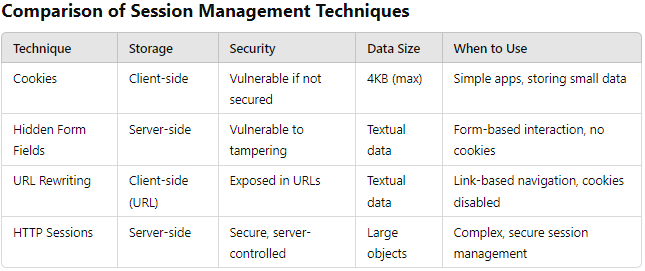
}

#### **Advantages of HTTP Sessions**

* **Server-side storage**: Sensitive information is stored securely on the server.
* **Works with all browsers**: Doesn't rely on cookies if URL rewriting is used.
* **Can store complex objects**: Unlike cookies, sessions can store entire objects.

#### **Disadvantages of HTTP Sessions**

* **Server load**: Storing session data on the server can increase load, especially with many active users.
* **Dependent on session ID**: If the session ID is lost (e.g., if cookies are cleared), the session will break.
* **Additional mechanisms for large-scale apps**: Distributed applications need session replication or sticky sessions.



By understanding the methods and constructors associated with each session management technique, you can choose the most suitable method based on your application's needs.

Top of Form

Bottom of Form