Web programming in Java

What is Java or Jakarta EE (Enterprise Edition) ?

Java EE Developed by **Sun Microsystems**, later managed by **Oracle**.

The latest Oracle-supported version was **Java EE 8**.

Packages – jakarta.*

Jakarta EE is the **successor of Java EE**, now managed by the **Eclipse Foundation**.

Packages – jakarta.*

Versions - Jakarta EE 9, 10, 11...

It is a set of **specifications** that extends the Java SE (Standard Edition) for additional enterprise features.

eg -Distributed computing , web services.

It's designed to provide a platform for building and deploying large-scale, multi-tiered, scalable, and secure network applications.

Which are these specifications (or contract or APIs) ?

Specifications of **primary essential services** required for any enterprise application.

What is an **enterprise application** ?

- An enterprise application (EA) is a large software system platform designed to operate in a corporate environment.
- Typically, it is a server side, remotely deployed, transactional, multi threaded ,complex and scalable application.
- Examples – Ecommerce, Content management ,Business intelligence, Human resource management etc.

Key APIs

- **Web:** Servlets, JSP
- **Persistence:** JPA
- **Messaging:** JMS
- **Transactions:** JTA
- **Web Services:** JAX-RS (REST), JAX-WS (SOAP)

Web Server vs. Application Server

A **Web Server** is designed mainly to **handle HTTP requests/responses**.

- Serves **static content** (HTML, CSS, JS, images).
- Handles **dynamic requests** using Servlets/JSP Manages
- **Manages HTTP request/response lifecycle**, session management, URL mapping,page navigation etc.
- **Example**
  - **Apache Tomcat** (provides a **Servlet container** to execute Servlets, JSPs.)

An **Application Server** provides everything a **Web Server does**, plus **enterprise features** to build large-scale applications.

In addition to web functionalities, it provides enterprise services.

- Example services – JPA , JMS , JTA , Web Services , full-fledged connection pooling , security , load balancing etc.
- **Examples of application server**
  - WildFly / JBoss from Red Hat
  - GlassFish as reference implementation from Oracle
  - WebSphere (IBM)
  - WebLogic (BEA Systems / Oracle)

Why Java EE?

- Multi-client support
- Server independence (specifications vs implementation)
- Standardized enterprise services
- Scalable, robust, secure infrastructure
- Interoperability with other systems

What is a **dynamic web application**?

It is a server side application , deployed on web server , meant for servicing web clients,using application layer protocol HTTP /HTTPS.

Typical **Layers involved in HTTP request-response flow**

- Refer to request-response-flow.png

Analyze the URL sent by the web browser

URL : http://www.mybank.com:8080/banking

http : application layer protocol(or also known as scheme)

www.mybank.com : DNS qualified host name (maps to IP address, to resolve the host)

8080 : TCP port no (to identify the portto reach the web server)

/banking : path or URI (uniform resource identifier)

In Java , when you are creating a web application

URI, typically starts with context path (path of the web application) , by default set as web project name.

Jakarta EE compliant web application folder structure created by IDE

- Refer to a diagram - web application folder structure.png"

Web Container (Servlet Container / WC)

- A **Web Container** (aka **Servlet Container**) is the engine within a web server that provides the **runtime environment for dynamic web components** such as **Servlets, JSPs, and Filters**.
- It sits inside a **web server** (like Tomcat or Jetty).
- It’s responsible for managing the **execution, lifecycle, and interaction** of web components with the client (via HTTP).
- Eg. **Apache Tomcat** or Eclipse Jetty
- It is **not just a web server**
- It **can serve static files** like a web server.
- But more importantly, it provides the **Servlet + JSP container** (i.e., the Web Container).
- Tomcat | Jetty implements **the Servlet Specification & JSP Specification**. You can deploy a .war file with servlets in Tomcat.You

cannot do that with Apache HTTP Server or Nginx alone.

What happens when you “Start Tomcat” (either standalone or via Eclipse)?

1. **JVM starts**
   - Eclipse (or standalone startup.bat) launches a **Java process**.
   - This process is just a **JVM instance**.
2. **Tomcat bootstrap loads**
   - Tomcat classes (org.apache.catalina.startup.Bootstrap) are loaded in the JVM.
   - Tomcat now initializes its core components.
3. **Servlet Container (Catalina) starts**
   - Catalina is the **Servlet Container implementation** inside Tomcat.
   - It loads web.xml, initializes contexts, servlets, listeners, filters, etc.
   - This is the heart of the **Web Container (WC)**.
4. **JSP Container (Jasper) starts**
   - Jasper is the **JSP engine** inside Tomcat.
   - It only activates when a .jsp is requested.
   - It translates .jsp → .java → .class (servlet) and hands it back to Catalina.
5. **Together (Catalina + Jasper) = Web Container (WC)**
   - Catalina handles **all servlets** (including JSPs once compiled).

- Jasper is a **part of Catalina**, dedicated to JSP compilation.

Version information

Tomcat 10.1.x Specifications

Component	Version	Notes
Servlet API	6.0 (jakarta.servlet.*)	Jakarta EE 10 specification
JSP API	3.0 (jakarta.servlet.jsp.*)	JSP 3.0, supports EL 4.0
EL (Expression Language)	4.0	Part of JSP 3.0
JSTL	3.0	Extra JARs jakarta.servlet.jsp.jstl jakarta.servlet.jsp.jstl-api

Main Jobs of any Web Container

1. **Request & Response Handling**

Creates HttpServletRequest and HttpServletResponse objects for every client request.

Maps requests to the appropriate servlet/filter/JSP.

2. **Lifecycle Management**

   Loads classes, instantiates servlets/JSPs/filters.

   Calls lifecycle methods:

   - init() → initialization
   - service() → request handling
   - destroy() → cleanup

3. **Container Services**

   Provides ready-made support for:

   - **Naming & directory services (JNDI)**
   - **Security (authentication, authorization)**
   - **Connection pooling & resource management**

4. **Concurrency Handling**

- Manages multiple client requests concurrently
  (multi-threaded request handling).

5. **Session Management & State Tracking**
   - Maintains client state across multiple requests using:
     - Cookies
     - URL rewriting
     - HTTP Session objects
6. **Page Navigation**
   - Provides mechanisms for request dispatching and navigation:
     - RequestDispatcher.forward()
     - RequestDispatcher.include()
7. **JSP & Filter Handling**
   - Translates JSP into servlets, compiles, and executes them.
   - Applies filter chains before/after requests.

What is **web.xml** ?

- It is a Deployment descriptor.
- Exists one per web application.
- Created by,developer (with help of IDE)
- It's location is under WEB-INF folder (meant for private contents)
- It's read by WC , when you run the server side application what does it consist of ?
- It consists of Deployment instructions
  - Eg. welcome page, servlet deployment tags, session configuration, security configuration etc.

Enter Servlets
Need
A plain **web server** can only serve **static content**
(HTML, CSS, images).
To add the dynamic nature to the web applications, use Servlets.

Jobs of a Servlet

1. **Request Processing**

   Accepts HTTP requests from clients (via HttpServletRequest).

2. **Business Logic (B.L.) Execution**

   Performs calculations, validations, interacts with services.

3. **Dynamic Response Generation**

   Generates **HTML, JSON, XML** dynamically (via HttpServletResponse).

4. **Manages Database Access (DAO) layer**

5. **Page Navigation**

   Forwards/redirects to JSPs, HTML, or other servlets.

   It is a **Java class** (without main() method).

- It represents a **dynamic web component** that runs inside a **Web Container (Servlet Container)**.
- The **Web Container** manages its **lifecycle**.

Servlet Lifecycle Methods

The WC calls these methods automatically:

1. **init(ServletConfig config)**
   - Called **once** when the servlet is first loaded.
   - Used for initialization tasks (loading configurations,

     setting up resources).
2. **service(HttpServletRequest req, HttpServletResponse res)**
   - Called **for each request**.
   - Handles both request processing & response generation.
   - Dispatches to doGet(), doPost(), doPut etc.
3. **destroy()**
   - Called **once before servlet is unloaded**.
   - Cleanup tasks (close DB connections, free resources).

Servlet API details

- Refer to a diagram servlet-api.png

1. Starting point is jakarta.servlet.**Servlet interface**.

It has defined 3 main life cycle methods

- void init(ServletConfig config) throws ServletException
- void service(ServletRequest request,ServletResponse resp) throws ServletException,IOException
- void destroy()

Other 2 methods are

- ServletConfig getServletConfig()
- String getServletInfo()

2. Servlet interface is implemented by jakarta.servlet.**GenericServlet** class
- It is an abstract class , represents protocol independent servlet.
- It has implemented init & destroy methods , but service method is still abstract.

4. For HTTP specific support , GenericServlet is further extended by jakarta.servlet.http.HttpServlet.

   It Provides **100% concrete implementations** of lifecycle and service methods (init(), destroy(), service()).
   It is used as the super class for all of our servlets.

Why HttpServlet is declared as an abstract class BUT with 100 % concrete functionality ?

- **Abstract** because the class is **not meant to be instantiated directly**(since it does not contain actual servicing logic)

- A servlet developer **must override at least one of the HTTP methods** (usually doGet() or doPost()) to make the servlet useful.

- Its concrete methods provide the **HTTP request dispatching logic**:

- The service() method **automatically routes** requests to one of:

    - doGet(), doPost(), doPut(), doDelete(), etc.

- If you don’t override these methods , they will always return **HTTP 405 Method Not Allowed**
- As a developer, you can **override only the methods your servlet needs**:
    - Example: A servlet that only handles GET requests: override doGet() and ignore doPost().
    - Lifecycle methods (init(), destroy()) can also be overridden if you need custom initialization or cleanup.

Deployment of a Servlet

Servlets can be deployed in **two ways** in a Java EE / Jakarta EE web application:

1 Deployment via Annotation (Modern Approach)

- Use **@WebServlet** at the **class level**.
- Eg. @WebServlet(value="/hello")

```
public class HelloWorldServlet extends HttpServlet {
    // override doGet()/doPost() as needed
}
```

How it works

- The **Web Container (WC)** reads the annotation **at deployment time**.
- It creates a **URL-to-servlet map**:
    - **Key:** URL pattern → /hello
    - **Value:** Fully qualified servlet class → pages.HelloWorldServlet
- Example URLs:
    - **URL:** http://host:port/day1.1/hello

  - **URI:** /day1.1/hello
  - **URL pattern:** /hello
- **Benefit:** No need to touch web.xml. Simple and less error-prone.

  OR

2 Deployment via Deployment Descriptor (web.xml) — Legacy Approach

```
<servlet>
    <servlet-name>abc</servlet-name>
    <servlet-class>pages.SecondServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>abc</servlet-name>
    <url-pattern>/test2</url-pattern>
</servlet-mapping>
```

How it works

- At **web app deployment**, the WC reads web.xml to:
- Instantiate the servlet class.
- Create a **URL-to-servlet map** using the <servlet-mapping> tags:
  1. **Key:** /test2
  2. **Value:** pages.SecondServlet
- **Notes:**
  1. Multiple URL patterns can map to the same servlet.
  2. You can also define **init parameters**, **load-on-startup**, etc. in web.xml.

3 Web Container Behavior

- During deployment (whether **annotation or web.xml**):

1. Scans all servlets.
2. Builds **URL map** → URL pattern to servlet class.
3. Uses **Reflection** to instantiate the servlet when the first request arrives (or at startup if load-on-startup is used).
4. Manages servlet **lifecycle** (init(), service(), destroy()).

Default Loading Policy of Servlets

Lazy Loading (Default)

- By default, the **Web Container (WC)** loads and initializes a servlet **only when the first request comes in**.
- This conserves resources at startup.

2 Eager Loading (Optional)

- You can configure the WC to load a servlet **at application deployment time**
- Useful for **heavy-weight initialization tasks** like:
    - Setting up a database connection pool.
    - Bootstrapping frameworks (Spring, Hibernate).
    - Preloading large caches.

---

3 How to Enable Eager Loading

You can configure it in two ways:

A. Annotation

```
@WebServlet(value="/hello", loadOnStartup=1)

public class HelloWorldServlet extends HttpServlet {...}
```

OR

B. Deployment Descriptor (web.xml)

```
<servlet>
```

```
    <servlet-name>hello</servlet-name>
    <servlet-class>pages.HelloWorldServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

- **Positive value (e.g., 1, 2, 3 ...)** → servlet is loaded eagerly, at startup.
- **0 or negative / absent** → servlet is loaded lazily (default).
- If multiple servlets are eager-loaded, **lower numbers load first**.

Servlet life cycle managed by WC (Servlet Container)

1. At the time of Web Server startup (eg. Tomcat) , it refers to configuration file to create a thread pool.

   Why Thread Pool?
   - Creating/destroying threads per request/response = expensive.
   - Pool improves performance by reusing threads.
   - Supports **concurrent request handling**.

2. For Tomcat , it's configuration is in server.xml.
3. Default configuration
   - **minSpareThreads = 10** /25 (default) → when Tomcat starts, it pre-creates **10 /25 threads** in the pool , **maxThreads = 200** (default) → at most 200 concurrent request-handling threads, **acceptCount = 100** (default) → if 200 threads are busy, Tomcat queues up to 100 more requests.

4. At the time of Web Application Deployment , WC creates one ServletContext instance per web application.
5. WC builds URL mapping (from @WebServlet or web.xml)
6. WC checks Servlet loading policy.

   In case of **eager initialization**, WC starts the servlet life cycle immediately.

- Loads servlet class
- Creates singleton instance
- Creates ServletConfig , as dependency
- Invokes init() method & passes ServletConfig to it.

In case of **lazy initialization** , same steps are execute , but after receiving the first request from client.

In case of any errors during init(ServletException or any RuntimeException) , WC marks the servlet as unavailable. In such case client will later get 500 (Internal Server Error) or 503 (Service unavailable) . WC aborts servlet life cycle.In case of success it continues with servicing phase.

7. Servicing or Request Processing Phase

After Client sends a request , WC takes a thread from pool

- WC calls service(req, res) on servlet

- HttpServlet.service() dispatches:

doGet() → for GET

doPost() → for POST

doPut() → for PUT

doDelete() → for DELETE

- After handling, pooled out thread , returns to pool , ensuring reusability.

(Since it is One servlet instance shared between multiple threads, servlet programmer must ensure thread-safety!)

8. **Destroy phase** - On server shutdown or undeploy or redeploy

WC calls destroy() , to Cleanup resources (DB, files, threads)

- After this Servlet instance is eligible for Garbage Collection , thus ending servlet life cycle.

Important Servlet API

- To read request parameters sent from the client

jakarta.servlet.ServletRequest i/f methods

1. public String getParameter(String paramName)

2. public String[] getParameterValues(String paramName)

Page Navigation Techniques in Servlets / JSP

Page Navigation = Taking user from one page (resource) to another page.

Two standard approaches exist: **Client Pull** and **Server Pull**.

1. Client Pull

The client's web browser makes a **new request** to the next page.

1.1 User Action (Explicit Navigation)

- Triggered when the user **clicks a link / button / submits a form**.
- Browser generates a **new URL request**.

1.2 Redirect Scenario (Automatic Navigation)

- Triggered by the **server** via HttpServletResponse.sendRedirect().
- Browser automatically issues a new request.

API of HttpServletResponse interface

public void sendRedirect(String redirectLocation) throws IOException

Example

response.sendRedirect("admin_page");

What Happens Internally?

1. Web container (WC) **discards response buffer**.
2. WC sends a **temporary redirect response**:
   - **Status Code:** 302 Found
   - **Header:** Location: admin_page

- **Body:** empty

3. Browser makes a **new GET request**:
4. URL: http://host:port/context_path/admin_page
5. Method: GET
6. New request → New response.

Important:

- If response is already committed (writer flushed/closed), IllegalStateException is thrown.
- Navigation can be **within the same app, another app, or an external web site**.

2. Server Pull

Navigation happens **within the same request** (no new request comes from the client).
Also known as **Resource Chaining** or **Request Dispatching**.

jakarta.servlet.RequestDispatcher represents a wrapper , around the next resource(HTML , Servlet or JSP) , created by Web container.

Steps

1. Create a RequestDispatcher for the target resource.
2. RequestDispatcher rd = request.getRequestDispatcher("nextPage");

2.1 Forward Scenario

API of RequestDispatcher

public void forward(ServletRequest req, ServletResponse res)

Behavior

- Current servlet does some initial processing.
- Another resource (Servlet/JSP) generates the final response.
- Response buffer is **cleared automatically** before forward.

- If response is already committed → IllegalStateException.

Limitation

- Only the **last resource** in the chain generates the final response.

2.2 Include Scenario

API

public void include(ServletRequest req, ServletResponse res)

Behavior

- Includes the output of another resource **at runtime** (server-side include).
- Useful for adding headers, footers, menus, etc.
- Output is **added** to the response; main servlet’s output remains.

Limitation

- Included resource **cannot change response headers or status code** (if tried WC will ignore it!).

Quick Comparison

Aspect	Client Pull (Redirect)	Server Pull (Forward / Include)
Request Count	Two (new request)	One (same request)
Who initiates?	Browser	Server (WC)
URL in browser	Changes to new URL	Stays same as original servlet
Use Case	Navigate to external app, PRG pattern	Internal chaining, reuse components, separation of concerns
Exception	IllegalStateException if	

Aspect	Client Pull (Redirect)	Server Pull (Forward/Include)
	response committed	

Server side state management techniques

Why State Management is Needed ?

1. To **identify a specific client** among multiple clients.
2. To **remember the client's state** across multiple requests.

   Examples

   - Shopping cart items in Ecommerce app
   - Banking account information , in Banking
   - Customer Portfolio in Wealth Management apps
   - User preferences in any web app

Monolithic web apps (traditional JSP/Servlet, Spring MVC, PHP, etc.) → server-side session management (HttpSession) is common. The server holds session state, and clients are tracked using cookies like JSESSIONID.

Modern full stack apps (React/Angular + backend APIs) →

- Server is designed to be **stateless** (REST principle).
- Client maintains state locally (localStorage, sessionStorage, Redux, etc.).
- Backend uses **JWT tokens (or similar)** for authentication/authorization, passed on every request.
- This avoids server-side session storage and supports scalability (multiple servers, microservices, load balancers).

What is a Session?

- A **session** represents the *conversational state* between a client and server.

- It consists of **multiple requests and responses** exchanged during a user's interaction with a web app.
- Since **HTTP** and **Web servers** are **stateless**, sessions are required to maintain continuity.

Without sessions: every request is independent. The server cannot tell if two requests are from the same client.

With sessions: server can identify a client and remember their state (e.g., shopping cart, login status).

Lifetime of a Session

- A session starts when a client logs in or makes the first request that triggers session creation.
- It ends when:
  - The session times out
  - Or it's invalidated manually.
- **Default timeout in Tomcat = 30 minutes** (configurable).

Session Tracking Techniques in Jakarta EE

1. Plain Cookie-based Tracking

- **Cookie** is a small piece of text data stored by the browser.
- It is created by the server, sent in **response headers**, returned by client in **subsequent requests**.

API Steps

1. Create a cookie

jakarta.servlet.http.Cookie class constructor.

Public Cookie (String name,String value)

Eg .Cookie cookie = new Cookie("user_name", "Madhura");

2. Add the cookie to response header

Method of HttpServletResponse

public void addCookie(Cookie cookie)

Eg. response.addCookie(cookie);

3. Retrieve the cookie / cookies from request header.

Method of HttpServletRequest

Public Cookie[] getCookies()

Returns an array containing all of the Cookies that client sent with this request. This method returns null if no cookies were sent.

Eg. Cookie[] cookies = request.getCookies();

Cookie class Methods

- public String getName()
- public String getValue()
- public void setMaxAge(int seconds)
  - -1 → temporary (in memory until browser closes)
  - 0 → delete immediately
  - >0 → persistent (stored on disk)

Disadvantages of Cookie based Session Tracking

- Developer must manually manage cookies.
- Only text data supported (hard for objects).
- More cookies means more network overhead.
- Entire state is stored on **client side** , meaning if cookies disabled, session tracking will fail.

2. HttpSession-based Tracking (Preferred)

- State is stored on the **server side** inside a HttpSession object.

- Only a **session ID** is stored on the client (typically in a cookie like JSESSIONID).
- Servlet container(WC) manages the cookies automatically.

Development Steps (API)

1. Get session from WC (Web Container)

Method of HttpServletRequest

public HttpSession getSession()

- If the client already has a session → returns the existing HttpSession.
  - If the client doesn't have one → creates a new session and returns it.
  - Eg. HttpSession session = request.getSession();

public **HttpSession getSession(boolean create)**

- create : true → Same as above (create a new session if none exists).
- create : false → Returns existing session, but if none exists, returns **null**.
- If new session is created , WC will invoke sessionCreated method of the HttpSessionListener

2. Store attribute under HttpSession

Method of HttpSession

public void **setAttribute**(String name,Object value)

Eg . session.setAttribute("cart", myCart);

3. Retrieve attribute from HttpSession

Eg. Cart cart = (Cart) session.getAttribute("my_cart");

4. **Remove attribute**

Method of HttpSession

public void **removeAttribute**(String name)

Eg. session.removeAttribute("my_cart");

5. To Check if session is new

public boolean **isNew()**

Returns **true** → if the session was **just created** during the current request and the client has not yet joined it (typically after successful sign-in) , otherwise returns false.

6. To get unique Session identifier , generated by WC , uniquely per client

public String **getId()**

7. Get all attribute names , bound to the current session.

public Enumeration<String> **getAttributeNames()**

8. To Change session expiration time (Default is 30 minutes for Tomcat)

Either use API -

public void **setMaxInactiveInterval**(int seconds);

OR use **declarative approach**, via configuration in web.xml

```
<session-config>
  <session-timeout>Timeout in minutes</session-timeout>
</session-config>
```

9. To get session creation time

public long **getCreationTime**()

10. Invalidate session

public void **invalidate**();

The session object becomes invalid, and any attributes stored in are discarded.

If the client had a session cookie (JSESSIONID) it may still exist but points to a session that no longer exists. Meaning it’s no longer valid for subsequent requests. WC will invoke sessionDestroyed method of the HttpSessionListener

□ Advantages:

- Server stores Java objects (no manual serialization).
- Less traffic, better security than pure cookies.

□□ Limitation:

- If cookies are disabled in browser, session still fails (unless combined with **URL rewriting**).

---

3. HttpSession + URL Rewriting

- If cookies disabled, container appends ;jsessionid=XYZ to the URL.
- Ensures session ID is still passed.
- Example:
- http://localhost:8080/shop/cart;jsessionid=123456789

□ Attributes in Servlet API

- **Attribute = key–value pair (name: String, value: Object)** stored in different scopes.

Scopes

1. **Request Scope** → available only for current request.
2. **Session Scope** → available across multiple requests from the same client.
3. **Application Scope** → shared across all clients for the entire web app.

□ Summary:

- **Cookie** = state stored on client.
- **HttpSession** = state stored on server (preferred).
- **URL rewriting** = fallback when cookies disabled.
- **Session** helps web apps remain *conversational* despite HTTP’s stateless nature.

Scopes of attributes(server side objects)

- **lifetime and visibility** of objects in a web application.

There are **four main scopes** in Java web apps (Servlets, JSP, Spring MVC, etc.):

1. Request Scope

- **Lifetime:** Exists **only for a single HTTP request**.
- **Stored as :** HttpServletRequest attributes.
- **Use case:** Passing data from a servlet to a JSP for a single page rendering, in request dispatching.
- **Code example:**

```
request.setAttribute("results", "Some Results…");

RequestDispatcher rd = request.getRequestDispatcher("result.jsp");

rd.forward(request, response);
```

- **In JSP:** ${requestScope.results} can access the attribute.

Key point: Data disappears after the response is sent.

2. Session Scope

- **Lifetime:** Exists **across multiple requests from the same user** until the session expires or is invalidated.
- **Stored as :** HttpSession attributes.
- **Use case:** Store user login info, shopping cart, or preferences.
- **Code example:**

```
HttpSession session = request.getSession();

session.setAttribute("user_details", user);
```

- Available in all pages for that user until logout or session timeout.

3. Application (Context) Scope

- **Lifetime:** Exists **for the entire lifetime of the web application** (until server shuts down or app is redeployed).
- **Stored as :** ServletContext attributes.
- **Use case:** Stores global configuration, application-wide counters, or shared resources.
- **Code example:**

```
ServletContext context = getServletContext();

context.setAttribute("sale_info", currentSale);
```

- Accessible in **all servlets/JSPs** in the application.

4. Page Scope (specific to JSP)

- **Lifetime:** Exists **only within the JSP page** where it is declared.
- **Storedas :** Implicit JSP objects (pageContext).
- **Use case:** objects private within a JSP.
- **Code example:**

```
<% pageContext.setAttribute(“message”, “Hello Scopes!”); %>
```

Summary Table

Scope	**Stored In**	**Lifetime**	**Accessible By**
Page	pageContext	One JSP page	Only that page
Request	HttpServletRequest	One HTTP request	Servlets/JSPs handling same request
Session	HttpSession	User session	All requests from same user
Application	ServletContext	Entire application	All users, all requests

ServletConfig Overview

- **Purpose:** Pass **servlet-specific configuration** information (init parameters) from the web container to a servlet.
- **Interface:** jakarta.servlet.ServletConfig
- **Created by: Web Container (WC)**
- **When:** Right after the servlet instance is created via default constructor, & **before init()** is called(dependency)

Key Points

1. **Servlet-specific** – The init parameters set here are **private to that servlet only**. Can be accessed from init() method onwards.
2. **To access these , use these steps**

- public ServletConfig getServletConfig() – method of GenericServlet
- ServletConfig's method
  - String getInitParameter(String name) → Get the value of a specific init-param.
  - Enumeration<String> getInitParameterNames() → Enumerate all init parameters

How to Configure Servlet Init Parameters

1. Using web.xml

```
<servlet>
    <servlet-name>test</servlet-name>
    <servlet-class>pages.TestInitParam</servlet-class>
    <init-param>
        <param-name>name</param-name>
        <param-value>value</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>test</servlet-name>
    <url-pattern>/test_init</url-pattern>
</servlet-mapping>
```

OR

2. Using @WebServlet Annotation

```
@WebServlet(
    value="/test",
    initParams={
        @WebInitParam(name="nm1", value="val1"),
        @WebInitParam(name="nm2", value="val2")
    }
)
public class MyServlet extends HttpServlet {...}
    @Override
    public void init() throws ServletException {
```

Summary Table

Interface	**ServletConfig**
Scope	Servlet-specific
Created by	Web container
When	After servlet instance creation & before init()

Interface	**ServletConfig**
Stores	init parameters for that servlet only
Access from servlet	getServletConfig().getInitParameter()

ServletContext Overview

- **Purpose:** Provides **application-wide (global) information** and services to servlets.
- **Interface:** jakarta.servlet.ServletContext
- **Created by: Web Container (WC)**
- **When:** At **web application deployment time**, before any servlet is initialized.

Note: Each web application gets **exactly one ServletContext instance**, shared by all servlets / JSPs in that app.

- **Relationship with ServletConfig:**
  The ServletContext object is **contained in the ServletConfig**, which the WC passes to each servlet during initialization.

Key Uses of ServletContext

1. Server-side logging

```
ServletContext context = getServletContext();

context.log("This is a log message");
```

- Logs messages to the server side log file (Eg. <TOMCAT_HOME>\logs\catalina.out)
- Useful in debugging .

2. Application-scoped attributes

- Attributes stored in ServletContext live **for the entire application** (shared by all servlets and all users).
- Thread safety: Since multiple threads may access attributes, use

synchronized blocks if modifying mutable objects.

- Lifetime: Until the application is undeployed or server shuts down.

3. Accessing global (context) parameters

- **Add in web.xml:**

```
<context-param>
    <param-name>user_name</param-name>
    <param-value>abc</param-value>
</context-param>
```

- **Access in servlet:**

```
ServletContext context = getServletContext();
String userName = context.getInitParameter("user_name"); // returns "abc"
```

These parameters are **global for the whole web app**, unlike ServletConfig parameters which are servlet-specific.

4. Creating RequestDispatcher

- To forward or include requests within the same web application:

```
ServletContext context = getServletContext();
RequestDispatcher rd = context.getRequestDispatcher("/next_page");
rd.forward(request, response);
```

5. Summary Table

Interface	ServletContext
Scope	Application-wide (global)
Created by	Web Container
Number of instances	1 per web application

Interface	**ServletContext**
Access	getServletContext() from Servlet
Stores	Context init parameters, application attributes
Lifetime	Entire application lifecycle
Logging	log(String msg)
Request dispatching	getRequestDispatcher(String path)

Note : You can access RequestDispatcher either from ServletRequest or from ServletContext.

Difference –

ServletContext.getRequestDispatcher(String path)

- **Path:** Must be **absolute**, starting with / and relative to the **root of the web application**.
- **Use case:** Forward or include requests to **any servlet or JSP** in the same web app, regardless of the current request URL.

Eg.

```
ServletContext context = getServletContext();

RequestDispatcher rd = context.getRequestDispatcher("/pages/hello.jsp");

rd.forward(request, response);
```

HttpServletRequest.getRequestDispatcher(String path)

- Can use **Relative path**(Resolved relative to the current servlet or JSP) or can use Absolute path (starting with /)similar to ServletContext

  Eg.

  ```
  RequestDispatcher rd =
  request.getRequestDispatcher("../pages/hello.jsp");

  rd.forward(request, response);
  ```

Servlet Listeners (Web app listeners) ?

A **Servlet Listener** is a **special Java class** that listens for **specific events in a web application** and executes code in response.

They represent **Inversion of Control (IoC)**—the web container **notifies your code of events** rather than your code actively polling for them.

- Typical events include:
  - **Requests** being created or destroyed
  - **Sessions** being created or destroyed
  - **Web application (context)** being initialized or destroyed
  - **Attributes** being added, removed, or modified in request, session, or context
- Listeners implement **interfaces defined by the Servlet API**, which extend the **java.util.EventListener** interface.

2. Types of Listeners

a) Request Events

- ServletRequestListener → reacts to request lifecycle events (requestInitialized, requestDestroyed)

b) Session Events

- HttpSessionListener → reacts to session lifecycle events (sessionCreated, sessionDestroyed)
- HttpSessionAttributeListener → reacts to session attribute changes

c) Context (Web App) Events

- ServletContextListener → reacts to web app startup/shutdown (contextInitialized, contextDestroyed)
- ServletContextAttributeListener → reacts to context attribute changes

d) Other Events

- ServletRequestAttributeListener → request attribute changes
- AsyncListener → for asynchronous processing

3. Event Handling Steps

1: Create Listener Class

Implement the required listener interface/interfaces.

Eg.

```
public class MyWebAppListener implements
ServletContextListener,HttpSessionListener,ServletRequestListener {

/* Implement callback / notification methods */

}
```

2: Register the Listener

2.1 Either Using Annotation

- Add @WebListener at the class level.
- The container automatically detects it.

2.2 Or Using tags in web.xml

```
<listener>

      <listener-class>com.example.MyContextListener</listener-class>

</listener>
```

4. Practical Uses of event listeners

- Logging request/session creation or destruction
- Initializing resources (DB connection pool) on app startup
- Cleaning up resources on shutdown
- Counting active sessions/users
- Monitoring attribute changes

Servlet Filters

. Why Filters?

Filters provide **reusability and separation of concerns**:

1. **Encapsulate recurring tasks (cross-cutting concerns)**
   - Authentication, logging, encryption, compression, session checks, etc.
   - Keeps **business logic** (servlets/JSPs) clean.
2. **Dynamic interception**
   - Can intercept **requests** to any resource (servlet, JSP, static content)
   - Can intercept **responses** from a resource before sending to the client.

2. What is a Filter?

- **A dynamic web component**, like a servlet or JSP.
- Resides within the web application.
- Life-cycle is **managed by the Web Container (WC)**.
- Performs **filtering tasks** on:
  - Incoming requests
  - Outgoing responses
  - Or both.

3. Common Uses of Filters

Type	**Use Case**
Authentication Filter	Check user login/session before accessing protected resources
Logging Filter	Log request/response details
Image Conversion	Resize or convert images on the fly
Data Compression	GZIP compress responses to save bandwidth

Type	**Use Case**
Encryption	Encrypt/decrypt request/response data
Session Check	Verify session validity

4. Creating a Filter

1. **Create a class that implements the jakarta.servlet.Filter interface**
2. **Implement three life-cycle methods:**

a) public void init(FilterConfig filterConfig)

Called **once** when filter is created(at the web app deployment time)

Use it to initialize resources or read init parameters.

b) doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

Called **per request/response**.

- Can do **pre-processing** before passing request to servlet/JSP.
- Can do **post-processing** on response.
- **Important:** Call chain.doFilter(request, response) to pass control to next filter or resource.

Example :

```
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
    // Pre-processing
    System.out.println("Request intercepted");
    chain.doFilter(request, response); // pass to next filter / servlet
    // Post-processing
    System.out.println("Response processed");
}
```

c) destroy()

- Called **once** when filter is removed/unloaded(at the end of web app)
- Use it to clean up resources (DB connections)

5. Deploying a Filter

5.1 Either Using Annotation

```
@WebFilter(
    urlPatterns = "/*",
    initParams = {
        @WebInitParam(name="param1", value="value1")
    }
)
public class AuthenticationFilter implements Filter { ... }
```

5.2 OR Using web.xml

```
<filter>
    <filter-name>authFilter</filter-name>
    <filter-class>filters.AuthenticationFilter</filter-class>
    <init-param>
        <param-name>param1</param-name>
        <param-value>value1</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>authFilter</filter-name>
    <url-pattern>/*</url-pattern>
```

```
</filter-mapping>
```

- **Filter mapping** determines **which requests the filter applies to**.
- Filters are applied **in the order they appear in web.xml** or **annotation ordering**.

6. Request-Response Flow with Filters

Client Request

↓

Filter1.doFilter()   (pre-processing)

↓

Filter2.doFilter()   (pre-processing)

↓

Servlet/JSP service()/_jspService

↓

Filter2.doFilter()   (post-processing)

↓

Filter1.doFilter()   (post-processing)

↓

Client Response

- Above describes a FilterChain, allowing multiple filters to work together.

Summary

- Filters are **reusable, decoupled, and dynamic** components.
- Ideal for **cross-cutting concerns**.
- Always **call chain.doFilter()**, otherwise the request won’t reach the servlet/JSP.

Flow

1. **Pre-processing:** Filters execute their code **before** passing control to the next filter or servlet.
2. **chain.doFilter(request, response)** passes control **down the chain** to the next filter or target servlet/JSP.
3. **Post-processing:** After the servlet/JSP has handled the request, control returns **back up the chain**, allowing filters to modify the response or perform cleanup.