# ****Java Servlets****

### ****1. Introduction to Java Servlets****

#### ****What is a Servlet?****

A **Servlet** is a Java class that runs in a servlet container (such as Tomcat or WildFly). It handles client requests, generates dynamic responses, and plays a key role in building web applications in Java.

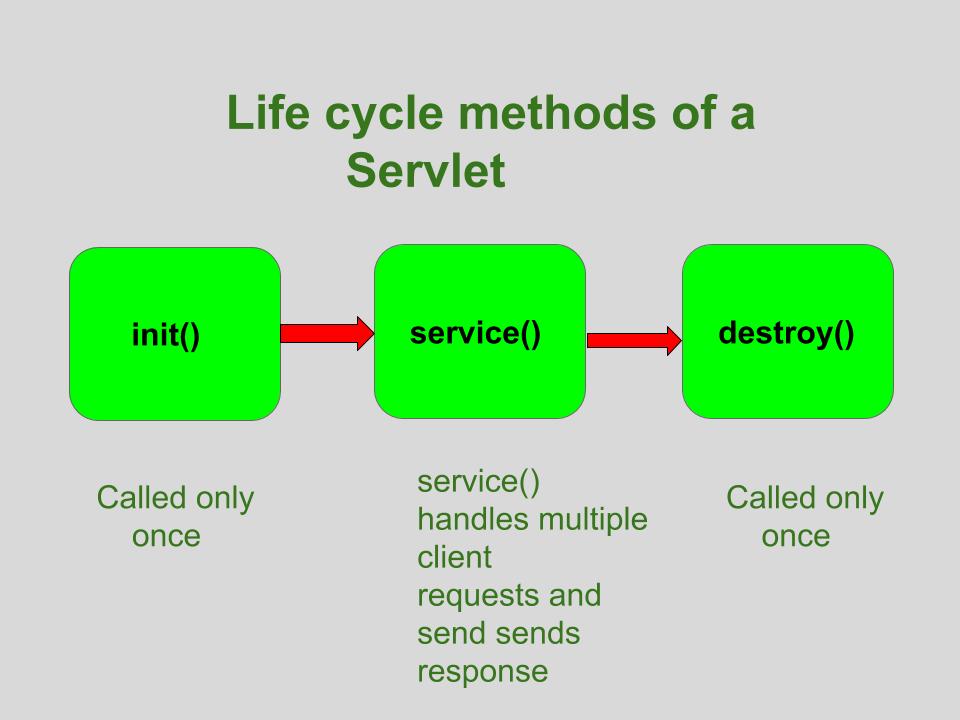
Servlets are server-side components that communicate with clients (typically web browsers), process their requests, and return appropriate responses (like HTML pages, JSON data, etc.).

**Why Use Servlets?**

* Servlets offer **dynamic content generation** (based on user input or data from databases).
* **Scalability and Efficiency**: Servlet-based applications can handle high traffic, as they work well with multi-threading.
* They allow for **separation of concerns** between the client and server-side logic.

#### ****Servlet Architecture and Lifecycle****

Servlets follow a well-defined lifecycle:



1. **Initialization**: The servlet is loaded into memory, and the init() method is called. This is done only once during the servlet's lifecycle.
2. **Request Handling**: Each client request triggers the service() method, where HTTP methods like doGet() and doPost() handle the request.
3. **Destruction**: The servlet is removed from the server when it’s no longer needed, and the destroy() method is called.

#### ****Servlet vs JSP: Key Differences****

* **Servlets**: Primarily used for handling requests, business logic, and server-side processing.
* **JSP**: Focuses on generating dynamic HTML. JSPs are compiled into Servlets by the server but provide a more readable format by separating Java logic from HTML.

#### ****Real-World Use Cases of Servlets****

* **E-commerce Platforms**: For handling user requests such as browsing products, adding items to the cart, and completing transactions.
* **Login Systems**: For authenticating users and managing sessions across multiple pages.

### ****2. Handling Client Requests in Servlets****

#### ****HTTP Request Methods****

HTTP methods define the type of action the client wants the server to perform:

* **GET**: Used to request data from the server (idempotent and safe).
* **POST**: Used to submit data (such as form data) to be processed by the server.
* **PUT/DELETE**: Used for updating or deleting data.

#### ****Retrieving Data from Requests****

Servlets can retrieve data from client requests:

* request.getParameter("param") for form data or query parameters.
* request.getAttribute("attr") for forwarded data.
* request.getInputStream() for reading raw request data (e.g., file uploads).

#### ****Handling Form Data (GET and POST)****

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class FormServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

response.setContentType("text/html");

PrintWriter out = response.getWriter();

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

out.println("<h2>Enter Your Name:</h2>");

out.println("<form method='POST' action='formServlet'>");

out.println("Name: <input type='text' name='username' /><br>");

out.println("<input type='submit' value='Submit' />");

out.println("</form>");

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

}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String username = request.getParameter("username");

response.setContentType("text/html");

PrintWriter out = response.getWriter();

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

out.println("<h1>Hello, " + username + "!</h1>");

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

}

}

#### ****Handling Cookies and Sessions****

* **Cookies**: A small piece of data stored in the client’s browser, sent back to the server with each request.

**Example Code for Cookie Management:**

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

public class CookieExampleServlet extends HttpServlet {

// Handling Cookie Creation and Setting Value

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// Create a new cookie to store the username

String username = "JohnDoe";

Cookie userCookie = new Cookie("username", username);

// Set the cookie's maximum age to 1 hour (3600 seconds)

userCookie.setMaxAge(3600); // 1 hour

// Add the cookie to the response

response.addCookie(userCookie);

// Output message

response.getWriter().println("Cookie has been set with username: " + username);

}

// Handling Cookie Retrieval

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// Get all cookies from the request

Cookie[] cookies = request.getCookies();

// Loop through the cookies to find the username cookie

String username = null;

if (cookies != null) {

for (Cookie cookie : cookies) {

if ("username".equals(cookie.getName())) {

username = cookie.getValue();

break;

}

}

}

// Output the value of the username cookie

if (username != null) {

response.getWriter().println("Retrieved Cookie Value: " + username);

} else {

response.getWriter().println("Username cookie not found.");

}

}

}

**Key Points:**

* **Setting Cookies:**
  + You create a Cookie object with a name ("username") and value ("JohnDoe").
  + Use setMaxAge() to specify how long the cookie should be valid (in seconds).
  + Add the cookie to the response using response.addCookie(cookie).
* **Reading Cookies:**
  + You retrieve cookies with request.getCookies().
  + Then, iterate through the cookies array to find the one with the name "username", and get its value.

This code demonstrates how to handle cookies by creating one on a GET request and reading it on a POST request. You can adapt it for different scenarios based on your requirements.

* **Sessions**: A mechanism to store user-specific data on the server, providing stateful interaction. HttpSession can be used to store session data like user IDs or preferences.

**Example Code for Session Management:**

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

public class SessionExampleServlet extends HttpServlet {

// Handling Session Creation and Setting Value

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// Get or create a session

HttpSession session = request.getSession();

// Set a session attribute

String username = "JohnDoe";

session.setAttribute("username", username);

// Output message

response.getWriter().println("Session has been set with username: " + username);

}

// Handling Session Retrieval

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// Get the current session

HttpSession session = request.getSession(false); // Use false to not create a new one if it doesn't exist

if (session != null) {

// Retrieve the session attribute

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

// Output the value of the username session attribute

if (username != null) {

response.getWriter().println("Retrieved Session Value: " + username);

} else {

response.getWriter().println("No username set in the session.");

}

} else {

response.getWriter().println("No session exists.");

}

}

}

**Key Points:**

* **Creating and Storing Session Data:**
  + The session is created or retrieved using request.getSession(). If no session exists, a new one will be created.
  + You can store objects in the session using session.setAttribute("key", value).
* **Retrieving Session Data:**
  + To retrieve session data, you can use session.getAttribute("key"), where "key" is the name of the attribute you want to fetch (in this case, "username").
* **Session Management Options:**
  + request.getSession(false) retrieves the session if it exists, but it won’t create a new session if one doesn't exist (unlike request.getSession() which always creates a session).
* **Session Persistence:**
  + Sessions are stored server-side, and their identifiers are typically sent to the client as a cookie (called JSESSIONID by default).

This example demonstrates how to use sessions to store and retrieve data on the server side across multiple HTTP requests from the same client.

### ****3. Advanced Servlet Features****

#### ****Request Dispatching and Forwarding****

There are two main methods provided by RequestDispatcher:

1. **forward(ServletRequest request, ServletResponse response)**:
   * Forwards the request and response to another resource (like a servlet, JSP page, or HTML file) on the server.
   * The original request and response are passed along, and the target resource handles them as though it was the original destination for the request.
2. **include(ServletRequest request, ServletResponse response)**:
   * Includes the content of another resource in the current response. This allows embedding the output of one resource inside another.
   * The response from the included resource is appended to the current response.

**Forwarding a Request:**

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

public class ForwardExampleServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// Business logic or processing

request.setAttribute("message", "This is a forwarded request");

// Get the RequestDispatcher for another servlet or JSP

RequestDispatcher dispatcher = request.getRequestDispatcher("/targetServlet");

// Forward the request to the target servlet

dispatcher.forward(request, response);

}

}

In this example, the request is forwarded to another servlet (/targetServlet) for further processing.

**Including a Resource:**

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

public class IncludeExampleServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// Business logic or processing

// Get the RequestDispatcher for the content to include

RequestDispatcher dispatcher = request.getRequestDispatcher("/header.jsp");

// Include the content of the header.jsp in the response

dispatcher.include(request, response);

// The response will now include the content of header.jsp

response.getWriter().println("This is the main page content.");

}

}

In this case, the header.jsp content is included in the current response, and after that, the main page content is written.

**Key Points:**

* **Forwarding**: Once the request is forwarded, no further processing in the original servlet or JSP happens; the target resource takes over the request handling.
* **Including**: The included resource's content is added to the response, but processing continues after that in the calling servlet/JSP.
* **Same Request and Response**: Both methods pass the original request and response, meaning they maintain the same session and attributes.

The RequestDispatcher is useful for controlling the flow of your web application, as it allows you to modularize your logic, forward control, or combine the output of multiple resources.

#### ****Using Servlet Filters****

A **Servlet Filter** is an object in Java web applications that performs filtering tasks on either the request to a servlet, the response from a servlet, or both. Filters can be used for a variety of purposes, including logging, authentication, input validation, modifying request and response objects, and more. Filters are part of the Java Servlet API and are typically configured in the web.xml or using annotations.

**Key Roles of Servlet Filters:**

1. **Request Filtering**:
   * Filters can inspect, modify, or reject incoming requests before they reach the target servlet or JSP.
   * For example, a filter can check if a user is authenticated before allowing access to the servlet or modify the request headers before processing.
2. **Response Filtering**:
   * Filters can inspect or modify the response before it is sent back to the client. This can include things like adding headers to the response or modifying the response body.
   * For instance, a filter can be used to compress the response content or add caching headers.
3. **Pre-Processing (Before Servlet Execution)**:
   * A filter can be used to perform tasks before the servlet or JSP processes the request, such as logging or auditing, security checks, or setting request attributes.
4. **Post-Processing (After Servlet Execution)**:
   * Filters can perform tasks after the servlet has processed the request but before the response is sent to the client. This could include actions like modifying the response body or performing cleanup tasks.
5. **Chain of Filters**:
   * Filters are often configured in a chain (filter chain), where each filter in the chain can process the request and/or response before passing it to the next filter or servlet. Filters can also pass control to the next filter in the chain using filterChain.doFilter().

**Example Use Cases for Servlet Filters:**

1. **Authentication and Authorization**:
   * A filter can check if a user is logged in or has the proper permissions to access a particular resource. If not, the filter can redirect the user to a login page or deny access.
2. **Logging and Monitoring**:
   * Filters can be used to log request details, such as IP address, request parameters, and response status codes, for auditing or monitoring purposes.
3. **Input Validation**:
   * Filters can validate incoming request data (e.g., form inputs) before passing it to a servlet for further processing.
4. **Compression**:
   * Filters can be used to compress the response (such as GZIP compression) before it is sent to the client.
5. **Caching**:
   * Filters can manage caching headers, control caching policies, or even cache response content itself to improve performance.
6. **Character Encoding**:
   * Filters can set or modify the character encoding for requests and responses to ensure proper encoding handling (e.g., UTF-8).

**How to Define a Servlet Filter:**

A servlet filter must implement the javax.servlet.Filter interface, which has three main methods:

1. **init(FilterConfig filterConfig)**: This method is called when the filter is initialized. It's a good place to read initialization parameters from the web deployment descriptor (i.e., web.xml).
2. **doFilter(ServletRequest request, ServletResponse response, FilterChain chain)**: This is the core method where filtering logic occurs. Here, you can inspect or modify the request and response, and then decide whether to pass the request/response along the filter chain.
3. **destroy()**: This method is called when the filter is being destroyed, usually at the end of the application's lifecycle. It can be used for cleanup tasks.

**Example of a Servlet Filter with Pre & Post-Processing:**

import javax.servlet.\*;

import javax.servlet.annotation.WebFilter;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

@WebFilter("/protected/\*") // Filter applied to URLs under "/protected/"

public class AuthenticationFilter implements Filter {

public void init(FilterConfig filterConfig) throws ServletException {

// Initialization logic (optional)

}

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

HttpServletRequest httpRequest = (HttpServletRequest) request;

HttpServletResponse httpResponse = (HttpServletResponse) response;

// Pre-processing: Check if the user is logged in

if (httpRequest.getSession().getAttribute("user") == null) {

// If not logged in, redirect to the login page

httpResponse.sendRedirect("/login.jsp");

} else {

// If logged in, pass the request along the filter chain

chain.doFilter(request, response); // Continue the request-response cycle

// Post-processing: Modify the response after the servlet has processed the request

// Example: Set a custom header or log the status code

httpResponse.setHeader("X-Processed-By", "AuthenticationFilter");

System.out.println("Response processed. Status Code: " + httpResponse.getStatus());

}

}

public void destroy() {

// Cleanup logic (optional)

}

}

**Flow of Execution:**

* **Request Pre-Processing**: The filter checks if the user is logged in. If not, it redirects the request to the login page and stops further processing.
* **Request Passing to Filter Chain**: If the user is logged in, the filter passes the request and response along the chain to the target servlet.
* **Servlet Execution**: The target servlet processes the request and generates the response.
* **Response Post-Processing**: After the servlet (or any other filter in the chain) has finished processing, the filter can modify the response as needed (in this case, by setting a custom header or logging information).

**Filter Configuration (via web.xml):**

Filters can be configured in the web.xml deployment descriptor (though using annotations, like @WebFilter, is more common in modern Java web applications).

Example web.xml configuration for the above filter:

<filter>

<filter-name>AuthenticationFilter</filter-name>

<filter-class>com.example.AuthenticationFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>AuthenticationFilter</filter-name>

<url-pattern>/protected/\*</url-pattern>

</filter-mapping>

**Key Points:**

* **Pre-Processing and Post-Processing**: Filters can work both before and after a request is processed by a servlet.
* **Filter Chain**: A filter can pass the request/response to the next filter in the chain using filterChain.doFilter().
* **Lightweight and Efficient**: Filters are typically lightweight components that don't have direct access to the business logic, but they can be very effective for tasks like security, logging, and monitoring.
* **Flexibility**: Filters provide a flexible and reusable way to handle cross-cutting concerns in a Java web application.

**Advantages of Using Servlet Filters:**

* **Centralized Functionality**: You can centralize common functionality (e.g., logging, authentication) in a filter rather than repeating code in each servlet or JSP.
* **Separation of Concerns**: Filters allow you to separate concerns like logging or input validation from your business logic in servlets.
* **Performance**: Filters can improve performance (e.g., by caching responses or compressing data) and reduce the load on back-end resources.

#### ****Servlet Listeners for Application Events****

Servlet listeners play an important role in the Java Servlet API by providing a mechanism for listening to and responding to various lifecycle events in a web application's environment. These listeners are designed to react to specific events, such as the creation or destruction of a session, the initialization or destruction of a servlet context, and changes to attributes in the session, context, or request.

There are three main types of servlet listeners:

1. **ServletContextListener**
2. **HttpSessionListener**
3. **ServletRequestListener**

**1. ServletContextListener**

* **Purpose**: This listener is invoked when the ServletContext is initialized or destroyed. The ServletContext is an object that represents the entire web application and is created when the web application starts and destroyed when the web application is stopped.
* **Common Use Cases**:
  + Initializing resources when the web application starts (e.g., setting up database connections).
  + Cleaning up resources when the web application is stopped (e.g., closing database connections).
* **Key Methods**:
  + contextInitialized(ServletContextEvent sce): Invoked when the ServletContext is initialized.
  + contextDestroyed(ServletContextEvent sce): Invoked when the ServletContext is destroyed.

import javax.servlet.\*;

import javax.servlet.annotation.WebListener;

import javax.servlet.ServletContextEvent;

@WebListener

public class MyServletContextListener implements ServletContextListener {

@Override

public void contextInitialized(ServletContextEvent sce) {

System.out.println("ServletContext initialized.");

// Initialize resources here, e.g., database connections

}

@Override

public void contextDestroyed(ServletContextEvent sce) {

System.out.println("ServletContext destroyed.");

// Clean up resources here, e.g., closing database connections

}

}

**2. HttpSessionListener**

* **Purpose**: This listener is invoked when an HTTP session is created or destroyed. An HTTP session is created when a user accesses the application and is typically used to track user-specific data.
* **Common Use Cases**:
  + Tracking user activity, such as logging when a session starts or ends.
  + Storing or cleaning up session-related data when a session is destroyed.
* **Key Methods**:
  + sessionCreated(HttpSessionEvent se): Invoked when a session is created.
  + sessionDestroyed(HttpSessionEvent se): Invoked when a session is destroyed.

import javax.servlet.\*;

import javax.servlet.annotation.WebListener;

import javax.servlet.http.HttpSessionEvent;

import javax.servlet.http.HttpSessionListener;

@WebListener

public class MyHttpSessionListener implements HttpSessionListener {

@Override

public void sessionCreated(HttpSessionEvent se) {

System.out.println("Session created: " + se.getSession().getId());

// Perform actions when the session is created

}

@Override

public void sessionDestroyed(HttpSessionEvent se) {

System.out.println("Session destroyed: " + se.getSession().getId());

// Perform actions when the session is destroyed

}

}

**3. ServletRequestListener**

* **Purpose**: This listener is invoked when an HTTP request is received and when it is about to be completed. It provides a mechanism to listen to the lifecycle of an HTTP request, enabling you to perform actions before and after request processing.
* **Common Use Cases**:
  + Logging the incoming request or tracking the request lifecycle.
  + Performing any request-level setup or cleanup (e.g., setting attributes on the request).
* **Key Methods**:
  + requestInitialized(ServletRequestEvent sre): Invoked when a request is initialized.
  + requestDestroyed(ServletRequestEvent sre): Invoked when a request is destroyed.

import javax.servlet.\*;

import javax.servlet.annotation.WebListener;

import javax.servlet.ServletRequestEvent;

import javax.servlet.ServletRequestListener;

@WebListener

public class MyServletRequestListener implements ServletRequestListener {

@Override

public void requestInitialized(ServletRequestEvent sre) {

System.out.println("Request initialized.");

// Perform actions when the request is initialized

}

@Override

public void requestDestroyed(ServletRequestEvent sre) {

System.out.println("Request destroyed.");

// Perform actions when the request is destroyed

}

}

**Key Differences Between the Listeners:**

1. **Scope**:
   * **ServletContextListener** deals with the application-wide lifecycle, responding to events related to the ServletContext, which represents the whole web application.
   * **HttpSessionListener** deals with events related to user sessions and tracks the lifecycle of a user's session.
   * **ServletRequestListener** handles events related to individual HTTP requests.
2. **Usage**:
   * **ServletContextListener** is often used for setting up application-wide resources such as database connections, file caches, or any shared resources that need to be initialized when the application starts.
   * **HttpSessionListener** is useful for tracking user activity (e.g., login/logout), cleaning up session-related resources, or managing user-specific information.
   * **ServletRequestListener** is useful for monitoring the flow of requests within the web application, such as logging requests or setting/requesting attributes from the request object.

**Example Use Cases for Servlet Listeners:**

1. **Logging User Activity**:
   * Using an HttpSessionListener, you can log when a user starts and ends a session, giving you insight into session activity.
2. **Managing Application Resources**:
   * A ServletContextListener can be used to initialize database connections or application-wide resources when the web application starts, and then clean them up when the application shuts down.
3. **Request Tracking**:
   * A ServletRequestListener can be useful for logging all incoming requests to the application. It helps track requests, monitor performance, or apply necessary request modifications.

### ****4. Servlet Configuration and Deployment****

#### ****Configuration in**** web.xml

The web.xml file configures Servlets, URL mappings, and other web application settings.

* Define a Servlet’s name and class.
* Map the Servlet to a specific URL pattern.

**Example Code for web.xml:**

<web-app>

<servlet>

<servlet-name>HelloServlet</servlet-name>

<servlet-class>com.example.HelloServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HelloServlet</servlet-name>

<url-pattern>/hello</url-pattern>

</servlet-mapping>

</web-app>

#### ****Servlet Annotations: @WebServlet****

Servlets can be configured using annotations instead of web.xml. For example, @WebServlet("/hello").

**Example Code:**

@WebServlet("/hello")

public class HelloServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

response.getWriter().write("Hello, World!");

}

}

**ServletConfig vs ServletContext**

In the Java Servlet API, both **ServletConfig** and **ServletContext** provide configuration and context-related information to servlets, but they serve different purposes and are used in different contexts. Here's a breakdown of both:

**1. ServletConfig**

* The ServletConfig object provides configuration information specific to a single servlet instance. It is used to pass initialization parameters for a servlet.

**Example:**

In this example, a servlet uses ServletConfig to access its initialization parameters:

**web.xml** (Servlet Configuration in Deployment Descriptor):

<web-app>

<servlet>

<servlet-name>MyServlet</servlet-name>

<servlet-class>com.example.MyServlet</servlet-class>

<init-param>

<param-name>username</param-name>

<param-value>admin</param-value>

</init-param>

<init-param>

<param-name>password</param-name>

<param-value>secret</param-value>

</init-param>

</servlet>

</web-app>

**Servlet Code (Using ServletConfig)**:

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

public class MyServlet extends HttpServlet {

@Override

public void init(ServletConfig config) throws ServletException {

super.init(config);

String username = config.getInitParameter("username");

String password = config.getInitParameter("password");

System.out.println("Username: " + username);

System.out.println("Password: " + password);

}

@Override

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

response.getWriter().println("Servlet Config Example");

}

}

In this case, ServletConfig is used to retrieve the initialization parameters (username and password) specific to this servlet.

**2. ServletContext**

* The ServletContext object provides information about the servlet container (web application environment) and allows servlets to share information across servlets within the same web application.

In the web.xml file, you can define **context parameters** that can be accessed by the entire web application. These parameters are usually defined within the <web-app> tag and can be accessed globally throughout the application.

**Example:**

<web-app

<!-- Context Parameter Configuration -->

<context-param>

<param-name>appName</param-name>

<param-value>MyWebApp</param-value>

</context-param>

<context-param>

<param-name>databaseUrl</param-name>

<param-value>jdbc:mysql://localhost:3306/mydb</param-value>

</context-param>

<!-- Servlet and other configurations -->

<servlet>

<servlet-name>MyServlet</servlet-name>

<servlet-class>com.example.MyServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>MyServlet</servlet-name>

<url-pattern>/myServlet</url-pattern>

</servlet-mapping>

</web-app>

* **<context-param>**: This tag defines a parameter that is available to all components in the web application.
  + **<param-name>**: Name of the parameter.
  + **<param-value>**: Value of the parameter.

In this example, two context parameters (appName and databaseUrl) are defined. These parameters can be accessed globally throughout the web application.

**2. Accessing Context Parameters in a Servlet**

To retrieve these context parameters in a servlet, you can use the ServletContext object. The ServletContext provides the getInitParameter() method to access the context parameters defined in web.xml.

**Example: Servlet Accessing Context Parameters**

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.io.\*;

public class MyServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// Access the ServletContext

ServletContext context = getServletContext();

// Retrieve context parameters defined in web.xml

String appName = context.getInitParameter("appName");

String databaseUrl = context.getInitParameter("databaseUrl");

// Use the parameters in the servlet

response.setContentType("text/html");

PrintWriter out = response.getWriter();

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

out.println("<h1>Application Name: " + appName + "</h1>");

out.println("<h1>Database URL: " + databaseUrl + "</h1>");

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

}

}

**Explanation:**

1. **getServletContext()**: This method returns the ServletContext object for the servlet, which is used to interact with the web application's context.
2. **getInitParameter("param-name")**: This method retrieves the context parameter by its name. In this example, it fetches the appName and databaseUrl parameters that were defined in the web.xml file.

The output of the servlet will be an HTML page displaying the application name and database URL.

### ****Real-Time Applications of Servlets****

A simple login system in a web application can be built using Servlets for authenticating users and managing sessions. Servlets handle form submissions, validate credentials, and maintain user sessions.

**Example Code for Login Servlet:**

// Handling form data and login logic

String username = request.getParameter("username");

String password = request.getParameter("password");

// Validate credentials (e.g., check against a database)

if (validUser(username, password)) {

HttpSession session = request.getSession();

session.setAttribute("user", username);

response.sendRedirect("welcome.jsp");

} else {

response.sendRedirect("login.jsp?error=true");

}

#### ****Building a Shopping Cart****

Servlets are used in e-commerce sites to handle the shopping cart operations (adding/removing products, calculating totals, etc.).

**Example Code for Cart Servlet:**

HttpSession session = request.getSession();

List<Product> cart = (List<Product>) session.getAttribute("cart");

if (cart == null) {

cart = new ArrayList<>();

}

cart.add(product); // Add selected product to cart

session.setAttribute("cart", cart);