|  |
| --- |
| **Servlet Basics** |

**Servlet Life Cycle Methods and Phases**

For a java program, there is one method which serves as the entry point to the application which is the main() method. For a servlet, we have three methods listed below.

**Life Cycle methods**

init()

service()

destroy()

These methods are called by the Web Container such as Apache Tomcat, WebLogic.

Init() method is called once. So, this is a good place to write code to establish connections to the database server.

The Service method is where all our business logic is kept. The service method is called n-times.

The destroy method should have code which does the opposite of what the init() method does. The destroy method is also called once.

Putting code in these method is our responsibility but calling the methods is the container’s responsibility.

There are 4 life cycle phases associated with the above three methods, which are listed below.

**Life Cycle Phases**

Instantiation – This is when the classes are loaded by the container.

Initialization – This is when the init method is used to open resources.

Servicing – this is where the service method is executed.

Destruction – this is when the destroy method is called.

**Web Application Folder Structure**

com/samsonmarikwa/OrderServlet.class

login.jsp

Mysql.jar spring.jar hibernate.jar

lib

classes

web.xml

home.html

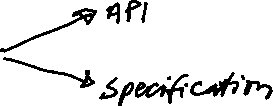
WEB-INF

WEBAPP

**Servlets**

A technology in the JEE that allows us to build dynamic web applications using java. The Servlet standard has an API and a Specification just like other standards from Oracle.

The API is for developers to develop dynamic web applications. There is a set of interfaces and classes. The Specification itself is a set of rules written in plain English which is for Application Server or Web Container developers like Apache Tomcat, WebSphere, WebLogic. The developers follow the rules described in the spec.



**Servlet**

is a program which runs on a web container. It receives input from a browser, processes it and depending on the request, can make calls to a database server. It can do a lot of things such as writing to files and reading from files. It receives responses from the database server and processes it before sending the response back to the client (browser).

OrderServlet.java

Servlet

Web Browser

DB

Web Container

**Servlet Annotations 3.0**

Servlet 3.0 version introduces annotations that can be used to configure various servlet components instead of using the web.xml file.

Examples include

@WebServlet

@WebInitParam

@WebFilter

@WebListener

*web.xml configurations override the annotations defined for the same component.*

**Registering a servlet**

Within the web.xml, create the following entries

|  |
| --- |
| <servlet>  <servlet-name>HelloServlet</servlet-name>  <servlet-class>com.samsonmarikwa.training.servlets.HelloWorldServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>HelloServlet</servlet-name>  <url-pattern>/hello</url-pattern>  </servlet-mapping> |

**Application Flow**

When a request comes from the browser to <http://localhost:8080/ServletBasics/hello> Apache Tomcat will handle the request as it is running on localhost port 8080. Tomcat determines the project name from the url that should handle the request. It then looks at the web.xml file for the /hello path configured in the servlet-mapping url-pattern to see which servlet should handle the request. It then instantiates the servlet HelloWorldServlet to handle the request. It will then execute the service method that is in the logic in our servlet.

**When using html**

To send input to the servlet via html form

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Addition</title>  </head>  <body>  <form action=*"additionServlet"*>  <h2>Enter the Numbers</h2><br />  Number1: <input type=*"text"* name=*"number1"* /><br/>  Number2: <input type=*"text"* name=*"number2"* /><br/>  <input type=*"submit"* name=*"submit"* />  </form>  </body>  </html> |

Add a servlet to retrieve the input, add and send the response

|  |
| --- |
| **public** **class** AdditionServlet **extends** GenericServlet {  @Override  **public** **void** service(ServletRequest request, ServletResponse response)  **throws** ServletException, IOException {  **if** (request.getParameter("number1") != **null** && request.getParameter("number2") != **null**) {  **int** num1 = Integer.*parseInt*(request.getParameter("number1"));  **int** num2 = Integer.*parseInt*(request.getParameter("number2"));  PrintWriter out = response.getWriter();  out.println("The result is " + (num1 + num2));  }  }  } |

Add a servlet to retrieve the input, add and send the response

|  |
| --- |
| <servlet>  <servlet-name>AdditionServlet</servlet-name>  <servlet-class>com.samsonmarikwa.training.servlets.AdditionServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>AdditionServlet</servlet-name>  <url-pattern>/additionServlet</url-pattern>  </servlet-mapping> |
| **JDBC** |

**JDBC Architecture**

|  |  |
| --- | --- |
| JDBC Client | The client is the code that we write. Connects to the DB. Perform CRUD operations. Processes the response. Close the connection. |
| JDBC API | Part of java.sql.\* package  We as developers, we learn and use the JDBC API  Comprises Connection, Statement, ResultSet and many more |
| JDBC Driver | Exists between the client software and the DB. Implements the JDBC API under the JDBC Specification. |
| JDBC Manager | Driver Manager establishes connection between the Client and Driver. After that the communication takes place between the client and the driver. |

Only Once to get a connection

Communicates ‘n’ times after a connection is acquired via the Driver Manager

DatabaseServer

JDBC API

Driver Manager

JDBC Driver

JDBC Client

**Steps to perform CRUD**

1. Establish the connection
2. Create the statement object
3. Submit the SQL query to DBMS
4. Close the statement
5. Close the connection

**Download MySQL Connector/J jar file.**

Copy the file to lib folder created in the project. Add the lib folder to the classpath under the Project Properties

Graphical user interface, text, application

Description automatically generated

**Steps to Use JDBC**

|  |  |
| --- | --- |
| Establish Connection interface | Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd"); // default port is 3306 |
| Create the Statement interface used to execute different different types of SQL statements | Statement statement = connection.createStatement(); |
| JDBC Driver | Exists between the client software and the DB. Implements the JDBC API under the JDBC Specification. |
| JDBC Manager | Driver Manager establishes connection between the Client and Driver. After that the communication takes place between the client and the driver. |

**executeUpdate()**

**try** {

Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");

Statement statement = connection.createStatement();

**int** result = statement.executeUpdate("insert into account values(1, 'Marikwa', 'Samson', 10000)");

System.***out***.println(result + " rows got inserted");

**int** result = statement.executeUpdate("update account set bal = 25000 where accno = 1");

System.***out***.println(result + " rows got updated");

**int** result = statement.executeUpdate("delete from account where accno = 1");

System.***out***.println(result + " rows got deleted");

} **catch** (SQLException e) {

e.printStackTrace();

}

**ResultSet**

ResultSet interface is used to handle the data that comes back when we execute a select query. It is an OO representation of the table records. It consists of three areas as shown below.

No Record Area

Cursor

Record Area

Zero Record Area

Moves the cursor

next() getXXX();

used to read individual fields in the record. XXX represents Int, Boolean, String etc. for each Primitive Type. The parameters for the method can be the column name or number.

Returns Boolean

**executeQuery()**

**try** (Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery("select \* from account");

){



**int** result = statement.executeUpdate("insert into account values(1, 'Marikwa', 'Samson', 10000)");

System.***out***.println(result + " rows got inserted");

**int** result = statement.executeUpdate("update account set bal = 25000 where accno = 1");

System.***out***.println(result + " rows got updated");

**int** result = statement.executeUpdate("delete from account where accno = 1");

System.***out***.println(result + " rows got deleted");

**while** (resultSet.next()) {

System.***out***.println(resultSet.getString(2));

System.***out***.println(resultSet.getString(3));

System.***out***.println(resultSet.getInt(4));

}

} **catch** (SQLException e) {

e.printStackTrace();

}

Closing of resources like connection, statement, resultSet was done in the finally block before it was introduced in Java 7. Now, by using try-with-resources, the resources are auto-closed. The resources that can be closed are those that implement or extend the AutoCloseable interface.

**The Service Provider Mechanism**

A Service Provider is configured and identified through a provider configuration file which we put in the resource directory *META-INF/services*. The file name is the fully-qualified name of the SPI (SP Interface) and its content is the fully-qualified name of the SPI implementation.

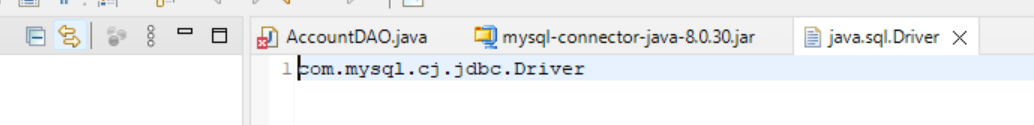
The Service Provider is installed in the form of extensions, a jar file which we place in the application classpath, the Java extension classpath or the user-defined classpath.

This is what is used by the DriverManager to find the appropriate driver to connect to the database.

The SPI feature was introduced in Java 6.

Graphical user interface, application, Word

Description automatically generated



|  |
| --- |
| **BUILDING DYNAMIC WEB APPLICATIONS** |

**HTTP Methods**

GET vs POST

|  |  |
| --- | --- |
| GET | POST |
| Default (if no HTTP method is mentioned, the web browser uses a GET by default) | The POST method has to be mentioned explicitly. <form method=”post” … |
| 3 cases |  |
| * User Specified URL – when a user types in a URL in the browser * User click on a hyperlink * User submits a form |  |
| Meant for getting data | Meant for posting data, creating, updating or deleting |
| No body | Body is present |
| Query String | Payload / Body |
| No sensitive data | Can be used for sensitive data |
| Restriction on amount of data | No restriction |
| Idempotent – we can execute GET any number of times and it will not affect the state of the application. It is considered a safe operation because no changes takes place to the data that is retrieved. | Not idempotent – if you submit data more than once, there is a chance that duplicate or multiple records will be created. Multiple updates can also take place. This can also occur if you refresh your browser. |

Creating a dynamic web project to use MySQL

1. Copy the MySQL Connector jar file to the lib folder shown below

Graphical user interface, text, application, email

Description automatically generated

The file will be added to the build path.

1. Create the html files and the CreateUserServlet.

Add the annotation for @WebServlet(“/addServlet”)

1. Implement the init() method in the CreateUserServlet. The init method is called once in the lifecycle of the servlet.

**public** **void** init() {

**try** {

// Before Java6 Service Provider Mechanism we had to load the driver class. This is required for Tomcat as

// this the auto-loading of the class is disabled due to memory leaks and we are required to load the class manually.

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

connection = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");

} **catch** (SQLException | ClassNotFoundException e) {

e.printStackTrace();

}

}

1. Implement the doPost() method which gets called from every request from the browser.

|  |
| --- |
| **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String firstName = request.getParameter("firstName");  String lastName = request.getParameter("lastName");  String email = request.getParameter("email");  String password = request.getParameter("password");  **try** {  Statement statement = connection.createStatement();  **int** result = statement.executeUpdate(  "insert into user values('"  + firstName + "','" + lastName + "','"  + email + "','" + password + "')");  PrintWriter out = response.getWriter();  **if** (result > 0) {  out.print("<H1>USER CREATED</H1>");  } **else** {  out.print("<H1>Error Creating the User</H1>");  }  } **catch** (SQLException e) {  e.printStackTrace();  }  } |

1. Implement the destroy() method.

|  |
| --- |
| **public** **void** destroy() {  logger.info("destroy() started");  **try** {  connection.close();  } **catch** (SQLException e) {  logger.error(e.getMessage());  }    } |

|  |
| --- |
| **Init Parameters** |

**Servlet Init Parameters**

Init parameters are name value pairs of textual information that are supplied to a servlet declaratively, that is, through a web.xml file during its initialization phase. During the initialization phase, the servlet container creates the ServletConfig object from the web.xml through which parameters are passes to the servlet through the init method. A ServletConfig is applicable to a single Servlet.

Init(ServletConfig config)

Servlet

Container

String dbUser = config.getInitParameter(“dbuser”);

**Configure using Annotations**

|  |
| --- |
| @WebServlet(urlPatterns = "/addServlet", initParams = {  @WebInitParam(name = "dbUrl", value = "jdbc:mysql://localhost/mydb"),  @WebInitParam(name = "dbUser", value = "root"),  @WebInitParam(name = "dbPassword", value = "P@ssW0rd")  })  **public** **class** CreateUserServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** Connection conn;  **public** **void** init(ServletConfig config) {  **try** {  Class.*forName*("com.mysql.cj.jdbc.Driver");  conn = DriverManager.*getConnection*(config.getInitParameter("dbUrl"), config.getInitParameter("dbUser"), config.getInitParameter("dbPassword")); |

**Configure using web.xml**

No hardcoded values as values come from the web.xml file

|  |
| --- |
| <servlet>  <display-name>ReadUserServlet</display-name>  <servlet-name>ReadUserServlet</servlet-name>  <servlet-class>com.samsonmarikwa.user.servlets.ReadUserServlet</servlet-class>  <init-param>  <param-name>dbUrl</param-name>  <param-value>jdbc:mysql://localhost/mydb</param-value>  </init-param>  <init-param>  <param-name>dbUser</param-name>  <param-value>root</param-value>  </init-param>  <init-param>  <param-name>dbPassword</param-name>  <param-value>P@ssW0rd</param-value>  </init-param>  </servlet>  <servlet-mapping>  <servlet-name>ReadUserServlet</servlet-name>  <url-pattern>/readServlet</url-pattern>  </servlet-mapping> |

The init() method remains the same as the one when using annotations.

**Summary**

* Interface used to access init parameters which is available to a particular servlet

ServletConfig

* In the web.xml, what is the parent element of the init-param element

servlet

* Which annotation can be used inside the WebServlet annotation to configure init parameters?

@WebInitParam

|  |
| --- |
| **Servlet Context** |

**ServletConfig** and **ServletContext**, both are objects created at the time of [servlet](https://www.geeksforgeeks.org/introduction-java-servlets/) initialization and used to provide some initial parameters or configuration information to the servlet. But, the difference lies in the fact that information shared by ServletConfig is for a specific servlet, while information shared by ServletContext is available for all servlets in the web application.

**ServletContext:**

* ServletContext is the object created by Servlet Container to share initial parameters or configuration information to the whole application. There is one ServletContext for the entire application. Any servlet or JSP has access to the ServletContext which has data which has been put by another servlet or JSP, so the scope of the ServletContext is Application.
* Example:  
  Suppose, the name of one’s job portal is “NewWebsite.tg”. Showing the website name at the top of webpages delivered by different servlets, one needs to store the website name in every servlet inviting redundancy. Since the information shared by ServletContext can be accessed by every Servlet, it is better to go with ServletContext and retrieve the website name using getServletContext.getInitParameter(“Name”) whenever required.

<**context-param**>

<**param-name**>Website-name</**param-name**>

<**param-value**>NewWebsite.tg</**param-value**>

</**context-param**>

web.xml

Container

Servlet

injects

Initialization

**Retrieve ServletContext**

Once injected, the ServletContext can be retrieved in several ways.

1. **Init method with Config.**

The container sets the ServletContext into the ServletConfig when the servlet initializes.

public void init(ServletConfig config) {

ServletContext context = config.getServletContext();

1. **Init method without config**

The servlet inherits the getServletContext method from the Servlet class.

public void init() {

ServletContext context = getServletContext();

1. **In the service(…) methods that takes HttpServletRequest and HttpServletResponse**

ServletContext context = getServletContext();

**Using The ServletContext**

1. **Share and Manipulate Data**

You can set data in one servlet using **setAttribute()**

You can retrieve the data in another servlet using **getAttribute()**

You can remove data from another servlet or JSP using **removeAttribute()**

You can get all the attribute names into an array using **getAttributeNames()**

Enumeration<String> parameterNames = context.getInitParameterNames();

1. **To deal with Context params**
2. **To create the RequestDispatcher object for inter-servlet communication**
3. **To store information in to the server log files using the log() method**

This is rarely used since we used third-party libraries like log4j.

**Context Parameters**

Are name value pairs defined in the root element of the web.xml

<**context-param**>

<**param-name**>Website-name</**param-name**>

<**param-value**>NewWebsite.tg</**param-value**>

</**context-param**>

Summary

* Context-param are a child element of webapp
* Using which interface in the servlet API can we access the Context Parameters

ServletContext

* Context parameters can be accessed throughout the application

|  |
| --- |
| **Prepared Statement** |

A PreparedStatement is a child interface of the Statement interface.

A **PreparedStatement** is a pre-compiled SQL statement. Prepared Statement objects have some useful additional features than Statement objects. Instead of hard coding queries, PreparedStatement object provides a feature to execute a parameterized query.

**Advantages of PreparedStatement**

* When PreparedStatement is created, the SQL query is passed as a parameter. This Prepared Statement contains a pre-compiled SQL query, so when the PreparedStatement is executed, DBMS can just run the query instead of first compiling it like the Statement, which makes it fast.
* We can use the same PreparedStatement and supply with different parameters at the time of execution.
* An important advantage of PreparedStatements is that they prevent SQL injection attacks.

**Steps to use PreparedStatement**

1. Create Connection to Database

Connection myCon = DriverManager.getConnection(path,username,password);

1. Prepare Statement  
   Instead of hardcoding queries like,

select \* from students where age>10 and name ='Chhavi'

Set parameter placeholders(use question mark for placeholders) like,

select \* from students where age> ? and name = ?

PreparedStatement myStmt;

myStmt = myCon.prepareStatement(select \* from students where age> ? and name = ?);

1. Set parameter values for type and position (bind the values)

myStmt.setInt(1,10);

myStmt.setString(2,"Chhavi");

1. Execute the Query

ResultSet myRs= myStmt.executeQuery();

**Methods of PreparedStatement:**

* setInt(int, int): This method can be used to set integer value at the given parameter index.
* setString(int, string): This method can be used to set string value at the given parameter index.
* setFloat(int, float): This method can be used to set float value at the given parameter index.
* setDouble(int, double): This method can be used to set a double value at the given parameter index.
* executeUpdate(): This method can be used to create, drop, insert, update, delete etc. It returns int type.
* executeQuery(): It returns an instance of ResultSet when a select query is executed.

|  |
| --- |
| @WebServlet("/UpdateProductServlet")  public class UpdateProductServlet extends HttpServlet {  private static final long serialVersionUID = 1L;  Connection con;  PreparedStatement stmt;    public void init() {  try {  Class.forName("com.mysql.cj.jdbc.Driver");    ServletContext context = getServletContext();  String dbUrl = context.getInitParameter("dbUrl");  String dbUser = context.getInitParameter("dbUser");  String dbPassword = context.getInitParameter("dbPassword");  con = DriverManager.getConnection(dbUrl, dbUser, dbPassword);    stmt = con.prepareStatement("update product set name = ?, description = ?, price = ? where id = ?");    } catch (ClassNotFoundException | SQLException e) {  e.printStackTrace();  }  }    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  int id = Integer.parseInt(request.getParameter("id"));  String name = request.getParameter("name");  String description = request.getParameter("description");  int price = Integer.parseInt(request.getParameter("price"));    try {  stmt.setString(1, name);  stmt.setString(2, description);  stmt.setInt(3, price);  stmt.setInt(4, id);    int result = stmt.executeUpdate();    response.setContentType("text/html");  PrintWriter out = response.getWriter();  out.print("<b>"+result+" Product(s) updated</b>");  } catch (SQLException e) {  e.printStackTrace();  }  }    public void destroy() {  try {  stmt.close();  con.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Product Update</title>  </head>  <body>  <h2>Enter Product Details</h2>  <br />  <form action=*"UpdateProductServlet"* method=*"post"*>  Id: <input type=*"text"* name=*"id"* /><br /> Name: <input type=*"text"*  name=*"name"* /><br /> Description: <input type=*"text"*  name=*"description"* /><br /> Price: <input type=*"text"* name=*"price"* /><br />  <input type=*"submit"* name=*"Update Product"* />  </form>  </body>  </html> |

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"https://jakarta.ee/xml/ns/jakartaee"*  xmlns:web=*"http://xmlns.jcp.org/xml/ns/javaee"*  xsi:schemaLocation=*"https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-app\_5\_0.xsd http://xmlns.jcp.org/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*  id=*"WebApp\_ID"* version=*"5.0"*>  <display-name>PreparedStatement</display-name>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.jsp</welcome-file>  <welcome-file>default.htm</welcome-file>  </welcome-file-list>  <context-param>  <param-name>dbUrl</param-name>  <param-value>jdbc:mysql://localhost/mydb</param-value>  </context-param>  <context-param>  <param-name>dbUser</param-name>  <param-value>root</param-value>  </context-param>  <context-param>  <param-name>dbPassword</param-name>  <param-value>P@ssW0rd</param-value>  </context-param>  </web-app> |

|  |
| --- |
| **Inter Servlet Communication** |

Inter servlet communication is nothing but communication between servlets or request dispatching.

JSP

Servlet

JSP

Servlet

The RequestDispatcher interface is used to forward request to another servlet.

RequestDispatcher rd = request.getRequestDispatcher(uri);

The uri is where you want to go next.

Servlet 1 will forward the request to Servlet 2 for further processing and Servlet 2 will then send the response to the client

There are two ways to do the request dispatching

rd.forward(request, response);

Servlet 2

Servlet 1

Client

The second way is as follows

The container will merge the response from Servlet 1 and Servlet 2 and Servlet 1 will then send the response to the client

rd.include(request, response);

+

Servlet 2

Servlet 1

Client

**Usecase**

HomeServlet

LoginServlet

forward

include

Login.html

DB

**Summary**

Which of the following interfaces in the servlet API can be used to do inter servlet communication?

* RequestDispatcher

Which of the following is not a method used for inter servlet communication?

* Forward
* Next
* Include

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Login Page</title>  </head>  <body>  <form action=*"loginServlet"* method=*"post"*>  Email: <input type=*"text"* name=*"userName"* /><br />  Password: <input type=*"password"* name=*"password"* /><br />  <input type=*"submit"* value=*"Login"* />  </form>  </body>  </html> |

|  |
| --- |
| @WebServlet("/homeServlet")  **public** **class** HomeServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  PrintWriter out = response.getWriter();  response.setContentType("text/html");  out.print(request.getAttribute("message"));  }  } |

|  |
| --- |
| @WebServlet(urlPatterns = "/loginServlet", initParams = {  @WebInitParam(name="dbUrl", value="jdbc:mysql://localhost/mydb"),  @WebInitParam(name="dbUsername", value="root"),  @WebInitParam(name="dbPassword", value="P@ssW0rd")  })  **public** **class** LoginServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** Connection connection;  **private** PreparedStatement statement;  **private** ResultSet resultSet;    **public** **void** init(ServletConfig config) {  **try** {  Class.*forName*("com.mysql.cj.jdbc.Driver");    String dbUrl = config.getInitParameter("dbUrl");  String dbUsername = config.getInitParameter("dbUsername");  String dbPassword = config.getInitParameter("dbPassword");  connection = DriverManager.*getConnection*(dbUrl, dbUsername, dbPassword);    statement = connection.prepareStatement("select \* from user where email = ? and password = ?");  } **catch** (ClassNotFoundException | SQLException e) {  e.printStackTrace();  }    }  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {    String userName = request.getParameter("userName");  String password = request.getParameter("password");    **try** {  statement.setString(1, userName);  statement.setString(2, password);  resultSet = statement.executeQuery();    RequestDispatcher dispatcher = request.getRequestDispatcher("homeServlet");    **if** (resultSet.next()) {  request.setAttribute("message", "Welcome to InterServlet Communication " + userName);  dispatcher.forward(request, response);  } **else** {  dispatcher = request.getRequestDispatcher("login.html");  dispatcher.include(request, response);  }  } **catch** (SQLException e) {  e.printStackTrace();  }    }    **public** **void** destroy() {  **try** {  resultSet.close();  statement.close();  connection.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| **PreInitialization of Servlets** |

There are two types of Servlet initializations

* Lazy initialization
* Pre-initialization

**Lazy initialization**

When a Container initializes a servlet only when the first request comes from a client, that is called a Lazy initialization.

Servlet

Container

Web Client

Lazy Initialization

By default, all servlets are lazily initialized.

**Pre-Initialization**

A container will initialize the servlet before any request is received.

Container

Servlet

Pre Initialization

We can request a container to do pre-initialization by configuring load-on-startup tag under the servlet tag.

|  |
| --- |
| <servlet>  <display-name>CreateUserServlet</display-name>  <servlet-name> CreateUserServlet</servlet-name>  <servlet-class>com.samsonmarikwa.user.servlets.CreateUserServlet</servlet-class>  <load-on-startup> 1 </load-on-startup>  </servlet>  <servlet>  <display-name>ReadUserServlet</display-name>  <servlet-name>ReadUserServlet</servlet-name>  <servlet-class>com.samsonmarikwa.user.servlets.ReadUserServlet</servlet-class>  <load-on-startup> 2 </load-on-startup>  </servlet> |

The numeric value tells the container the priority of the servlet. The lower the number the priority of the servlet which means it gets started first.

**Uses Cases**

Web Services using CXF or Jersey. They use on servlet which is loaded from the startup and from that point knows which servlet to be get initialized by calling the service endpoints.

Spring MVC Framework also has the DispatcherServlet which should be loaded on startup.

**Lazy Initialization Example**

|  |  |
| --- | --- |
| @WebServlet("/preInitServlet")  **public** **class** InitServer **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **public** **void** init() {  System.***out***.println("Inside the init method");  }    **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.getWriter().write("From the pre init servlet");  }  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  }  } | Output when container starts, servlet not initialized |
| Sep 09, 2022 6:25:44 PM org.apache.catalina.core.StandardContext reload  INFO: Reloading Context with name [/ServletInitilization] has started  Sep 09, 2022 6:25:44 PM org.apache.catalina.core.StandardContext reload  INFO: Reloading Context with name [/ServletInitilization] is completed |
| When request is sent to container, servlet is initialized |
| INFO: Starting ProtocolHandler ["http-nio-8080"]  Sep 09, 2022 6:30:16 PM org.apache.catalina.startup.Catalina start  INFO: Server startup in [1122] milliseconds  Inside the init method |

When the container starts, none of the Servlet methods are executed. This indicates that the container does not initialize the Servlet. Only when we execute the servlet, we get the input which indicates that the Servlet is only loaded when there is a request from the browser.

**Pre-Initialization Example Using Annotation**

|  |  |
| --- | --- |
| @WebServlet(urlPatterns = "/preInitServlet", loadOnStartup = 0)  **public** **class** InitServer **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **public** **void** init() {  System.***out***.println("Inside the init method");  }    **protected** **void** doGet(HttpServletRequest req, HttpServletResponse res)  **throws** ServletException, IOException {  res.getWriter().write("From the pre init servlet");  }  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse res)  **throws** ServletException, IOException {  }  } | Servet is initalized when container starts when loadOnStartup is specified |
| Sep 09, 2022 6:19:13 PM org.apache.catalina.core.StandardContext reload  INFO: Reloading Context with name [/ServletInitilization] has started  Inside the init method  Sep 09, 2022 6:19:14 PM org.apache.catalina.core.StandardContext reload  INFO: Reloading Context with name [/ServletInitilization] is completed |
|  |

Executes the init() method when the Container starts, therefore pre-initializing the server. The integer determines the Servlet priority. 0 indicates a higher priority and the servlet will be initialized before all the other servlets.

**Pre-Initialization Example Using Web.xml**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"https://jakarta.ee/xml/ns/jakartaee"*  xmlns:web=*"http://xmlns.jcp.org/xml/ns/javaee"*  xsi:schemaLocation=*"https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-app\_5\_0.xsd"*  id=*"WebApp\_ID"* version=*"5.0"*>  <display-name>ServletInitialization</display-name>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.jsp</welcome-file>  <welcome-file>default.htm</welcome-file>  </welcome-file-list>  <servlet>  <servlet-name>InitServer</servlet-name>  <servlet-class>com.samsonmarikwa.Initilization.InitServer</servlet-class>  <load-on-startup>0</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>InitServer</servlet-name>  <url-pattern>/preInitServlet</url-pattern>  </servlet-mapping>  </web-app> |

Executes the init() method when the Container starts, therefore pre-initializing the server. The integer determines the Servlet priority. 0 indicates a higher priority and the servlet will be initialized before all the other servlets.

**Summary**

What is the name of the attribute on the WebServlet annotation that should be used for pre initialization?

* loadOnStartup
* loadOnInit
* loadOnStart
* preInit

Which of the following element can be used in the web.xml to configure pre initialization?

* load-on-start
* load-on-startup
* load-on-it
* load-fast

|  |
| --- |
| **Servlet Listeners** |

Servlet listeners are used for event handling in web applications. Once an event is invoked, the application can take appropriate action.

These events could be

Request creation and destruction

Session being created or destroyed

Context – a web application being loaded or unloaded from the container.

Async – we can also handle asynchronous events

To create a listener, we can implement the HttpSessionListener. The events specified above have specific interfaces just like the SessionListener.

class UserCountListener implement HttpSessionListener

Once we implement the class, we need to tell the container about the listener by marking it.

Annotation

@WebListener

Container

web.xml

<listener>

<listener-class> UserCountListener</listener-class>

When container starts, it scans the classpath and all the classes marked as listeners will be registered in the container as Listeners to be invoked when a certain event occurs

</listener>

Use Case

The term “**event**” refers to the occurrence of something. An event occurs when the state of an object changes. When these exceptions occur, we can conduct certain crucial actions, such as collecting total and current logged-in users, establishing database tables when deploying the project, building database connection objects, and so on. The javax.servlet and javax.servlet.http packages contain numerous Event classes and Listener interfaces.

|  |
| --- |
| **Filters** |

A Filter is a java class that can intercept a request from a client, do some processing and then pass on the request to a servlet or another filter for additional processing. A response from the servlet is handled in the same way before being passed back to the client.

A filter is pre-initialized by the container. Unlike the servlet, there is no lazy-initialization of filters. Filters are pre-initialized.

Post-processed logic applied

Browser

Container

SERVLET

JSP

F

I

L

T

E

R

Pre-processed logic applied

**Use Case**

Injection Attack Filter – this will look for string input with malicious content and the request can be rejected.

**Filter Chaining**

This is where a request is passed through two or more filters which perform different processes.

**Implementation**

@Filter annotation is used to mark a class as a filter. A web.xml file can also be used.

The life cycle of a filter is similar to that of a servlet.

Pre-Initialized by default

Container

Init(FilterConfig config)

doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

destroy()

The FilterChain has information on where the filter should send the pre-processed request, that is another filter or a servlet or JSP.

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"* xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"* version=*"4.0"*>  <servlet>  <description></description>  <display-name>FilterServletDemo</display-name>  <servlet-name>FilterServletDemo</servlet-name>  <servlet-class>com.samsonmarikwa.servlet.FilterServletDemo</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>FilterServletDemo</servlet-name>  <url-pattern>/filterDemoServlet</url-pattern>  </servlet-mapping>  <display-name>FiltersDemo</display-name>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.htm</welcome-file>  <welcome-file>default.jsp</welcome-file>  </welcome-file-list>  <filter>  <display-name>DemoFilter</display-name>  <filter-name>DemoFilter</filter-name>  <filter-class>com.samsonmarikwa.filters.DemoFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>DemoFilter</filter-name>  <url-pattern>/filterDemoServlet</url-pattern>  </filter-mapping>  </web-app> |

|  |
| --- |
| **package** com.samsonmarikwa.servlet;  //@WebServlet("/filterDemoServlet")  **public** **class** FilterServletDemo **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  response.getWriter().append("From the servlet\n");  }  } |

|  |
| --- |
| //@WebFilter("/filterDemoServlet") // "/\*" will make this filter available for all the Servlets in the application  // Filter order is as per filter-mapping entries in the web.xml  **public** **class** DemoFilter **extends** HttpFilter {    **public** **void** destroy() {  // **TODO** Auto-generated method stub  }  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {  PrintWriter out = response.getWriter();  out.print("Pre Servlet\n");  chain.doFilter(request, response);  out.print("Post Servlet\n");  }  **public** **void** init(FilterConfig config) **throws** ServletException {  // **TODO** Auto-generated method stub  }  } |

|  |
| --- |
| **Session Management** |

HTTP Statelessness

The server does not maintain a continuous connection whilst it handles the request and sends back the response.

request

Web Server

Web Client

response

Each time a request comes, the server opens a fresh connection and after sending the response it destroys the connection. HTTP protocol does not keep state of the connection. There is no need to remember anything from the previous connection.

**Advantages**

* Performance – No overhead – there is no need to maintain a state of the connection even where the client is not doing anything after sending the initial request.
* Scalability – The application can be deployed on several servers because the client does not need to be connected to the server that handled the initial request as there is no state from the previous connection.

**Disadvantages**

* The server cannot remember anything, for example, a shopping cart. Hence, we have to use session tracking.

**Steps for Session Management**

1. Create the session

HttpSession session = request.getSession();

1. Maintain data using the four attribute methods on HttpSession object across multiple user requests.
2. End the session

**How session management works?**

**Session Tracking**

To maintain state despite the Statelessness of HTTP.

request.getSession() is a method on the HttpServletRequest object which retrieves the session object in the request. If the session object is not found, the web server creates a unique HttpSession object which is written into the HttpServletRequest object and keeps the HttpSession object in memory in a key/value map, with the session id as the key and the session object as the value. When the server returns a response, it also includes the session id sent as a cookie in JSESSIONID cookie. Subsequent requests will have all the cookies generated by the server in the initial session. The server will see the JSESSIONID cookie which is used to retrieve the session object. This process of session tracking is important for HTTP as the protocol is stateless. In this way, we are able to create a stateful web application. This is necessary for applications such as e-commerce application which needs to match requests in created a Cart.

**What is a session?**

**Login to Logout** - activities that take place from the time of logging in to logging out are termed a session.

**Track user interaction** – for example when you access a ticket booking website where you do not login. The entire process of booking and checkout of the ticket is a session.

In Java, javax.servlet.http.HttpSession represents the session object.

In session tracking, there are two things that are key:

* Client / User Identification
* State Maintenance

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"https://jakarta.ee/xml/ns/jakartaee"*  xmlns:web=*"http://xmlns.jcp.org/xml/ns/javaee"*  xsi:schemaLocation=*"https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-app\_5\_0.xsd"*  id=*"WebApp\_ID"* version=*"5.0"*>  <servlet>  <servlet-name>sourceServlet</servlet-name>  <servlet-class>com.samsonmarikwa.servlets.sessionmanagement.SourceServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>sourceServlet</servlet-name>  <url-pattern>/sourceServletUri</url-pattern>  </servlet-mapping>  <servlet>  <servlet-name>targetServlet</servlet-name>  <servlet-class>com.samsonmarikwa.servlets.sessionmanagement.TargetServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>targetServlet</servlet-name>  <url-pattern>/targetServletUri</url-pattern>  </servlet-mapping>  </web-app> |

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>User Page</title>  </head>  <body>  <h1>Enter User Name:</h1>  <form method=*"post"* action=*"sourceServletUri"*>  Username: <input name=*"userName"* /><br />  <input type=*"submit"* value=*"send"* name=*"submitButton"* />  </form>  </body>  </html> |

|  |
| --- |
| **package** com.samsonmarikwa.servlets.sessionmanagement;  **import** java.io.IOException;  **import** java.io.PrintWriter;  **import** jakarta.servlet.http.HttpServlet;  **import** jakarta.servlet.http.HttpServletRequest;  **import** jakarta.servlet.http.HttpServletResponse;  **import** jakarta.servlet.http.HttpSession;  **public** **class** SourceServlet **extends** HttpServlet {    **private** **static** **final** **long** ***serialVersionUID*** = -8582813157012285897L;    **public** **void** doPost(HttpServletRequest request, HttpServletResponse response) {  String userName = request.getParameter("userName");    HttpSession session = request.getSession();  session.setAttribute("user", userName);    response.setContentType("text/html");  **try** {  PrintWriter out = response.getWriter();  out.print("<a href='targetServletUri'>Click Here To Get The User Name</a>");  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| **package** com.samsonmarikwa.servlets.sessionmanagement;  **import** java.io.IOException;  **import** java.io.PrintWriter;  **import** jakarta.servlet.http.HttpServlet;  **import** jakarta.servlet.http.HttpServletRequest;  **import** jakarta.servlet.http.HttpServletResponse;  **import** jakarta.servlet.http.HttpSession;  **public** **class** TargetServlet **extends** HttpServlet {    **private** **static** **final** **long** ***serialVersionUID*** = -5439760644476598855L;    **public** **void** doGet(HttpServletRequest request, HttpServletResponse response) {    HttpSession session = request.getSession();  String userName = (String) session.getAttribute("user");    response.setContentType("text/html");  **try** {  PrintWriter out = response.getWriter();  out.print("<h1>User Name Is: " + userName + "</h1>");  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

**Ending a Session**

**Explicit logout**

* Invalidate() HttpSession

This makes the container destroy the session associated with that user in memory.

**Session Expiry**

The WebContainer is responsible for destroying the session if the session is not active for a given time. By default, session expiry timeout in Tomcat is 30mins. We can override the default timeout.

* **Code**

session.setMaxInactiveInterval(600); // argument is in seconds, 600sec / 60min = 10mins

* **web.xml**

<session-config>

<session-timeout> 5 </session-timeout>

</session-config>

The time is in minutes

**Cookies**

Are name/value pairs of textual information that can be used to exchange data between the client (browser) and the Server using the HTTP headers. We can pass any data but usually cookies are used to maintain sessions between a Web Client and Web Container.

In Java EE, the name of the cookie used to maintain the session is jsessionid

jsessionid=1234

Web Client

Web Container

On the first request, the Web Container creates a unique id and puts it in a cookie, jsessionid. The cookie is sent to the web client as part of the response in the HTTP Headers. Subsequent requests will send back the cookies and the Web Container is able to read the session id which then allows it to retrieve the previous session object.

We rarely create cookies, but if it’s done, here is the code logic.

**Create Cookie**

Cookie cookie = new Cookie(“jsessionid”, “1234”);

**Set or Add a Cookie on the response**

response.addCookie(cookie);

**Retrieve Cookies from the Request**

Cookie[] cookies = request.getCookies(); will give you the array of cookies

cookies[0].getName(); will give you the name of the cookie

cookies[0].getValue(); will give you the value of the cookie

**URL Rewriting**

URL Rewriting takes place when cookies are disabled in the web client browser for some reason. If you want to exchange the session id, we have to fall back to URL writing. URL writing works only with url and not form fields.

**Final Code with the above items**

|  |
| --- |
| **package** com.samsonmarikwa.servlets.sessionmanagement;  **public** **class** SourceServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = -8582813157012285897L;  **public** **void** doPost(HttpServletRequest request, HttpServletResponse response) {  Cookie[] cookies = request.getCookies();  **if** (cookies != **null**) {  **for** (**int** i = 0; i < cookies.length; i++) {  System.***out***.println(cookies[i].getName());  System.***out***.println(cookies[i].getValue());  }  }  response.addCookie(**new** Cookie("securityToken", "1234567890"));  String userName = request.getParameter("userName");  HttpSession session = request.getSession();  session.setAttribute("user", userName);  response.setContentType("text/html");  **try** {  PrintWriter out = response.getWriter();  String urlWriting = "targetServletUri?sessionId=123"; // WebServer can automatically generate this url, with  // a unique id assigned to the sessionId  out.print("<a href='" + urlWriting + "'>Click Here To Get The User Name</a>");  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| **package** com.samsonmarikwa.servlets.sessionmanagement;  **public** **class** TargetServlet **extends** HttpServlet {    **private** **static** **final** **long** ***serialVersionUID*** = -5439760644476598855L;    **public** **void** doGet(HttpServletRequest request, HttpServletResponse response) {    Cookie[] cookies = request.getCookies();  **if** (cookies != **null**) {  **for** (**int** i = 0; i < cookies.length; i++) {  System.***out***.println(cookies[i].getName());  System.***out***.println(cookies[i].getValue());  }  }    HttpSession session = request.getSession();  String userName = (String) session.getAttribute("user");    response.setContentType("text/html");  **try** {  PrintWriter out = response.getWriter();  out.print("<h1>User Name Is: " + userName + "</h1>");  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| **Java Server Pages (JSP)** |

JSP is a technology in the JEE space that comes with an API and a Specification. The API is for the application developers and the specification for the JSP Container Creators. The specification is a set of rules that should be followed by the JSP container creators (such as Apache Tomcat, WebSphere, WebLogic, JBOSS, Glassfish etc).

Java Server Pages, are a dynamic web technology based on servlet container and Java EE specification which is used to generate dynamic web content in webpages. It was launched in the year 1999. It serves as a server-side technology based on various content formats such as XML or HTML or any other type of document contents. 

**Features of JSP:** Some important features of JSP are :

* It is an expression language for the server-side.
* It is easy to code as it allows tag-based programming.
* It is platform-independent, it can run anytime on any platform or any browser.
* It allows the building of dynamic web pages which helps to interact with the users in a real-time environment.
* It primarily connects with the server which provides an easy connection to the database.

request

JSP

Web

Client

response

JSP Container

Database

A JSP can do anything a servlet can do in an easier fashion whilst overcoming the limitations of a servlet.

**JSP and servlet differences**

Servlets are simply Java programs that run on a web server whilst JSP is simply a text document that contains two types of text: static text which is predefined and dynamic text which is rendered after server response is received.

| Key | Servlet | JSP |
| --- | --- | --- |
| Implementation | Servlet is Java code | JSP is in HTML language although Java code can also be embedded through JSTL or other languages required. |
| MVC | In MVC architecture servlets work as a controller. It receives the request, processes and sends back the response | JSP plays the role of view to render the response returned by the servlet |
| Request type | Servlets can accept and process all type of protocol requests. | JSP is compatible with HTTP request only. |
| Session Management | Session management is not enabled, the user has to enable it explicitly. | In JSP session management is automatically enabled. |
| Performance | Servlet is faster than JSP. | JSP is slower than Servlet because first the translation of JSP to java code is taking place and then compiles. |
| Modification reflected | Modification in Servlet is a time-consuming task because it includes reloading, recompiling and restarting the server for changes in our code to get reflected. | JSP modification is fast as we need to click the refresh button and code change would get reflected. |

**JSP Elements**

They are three types of JSP elements.

* Scripting elements

Allow us to embed Java code into a JSP page.

* Directives

Allow us to import other files to the JSP page

* Actions

Runtime instructions to the JSP container.

**JSP lifecycle methods and phases**

|  |  |
| --- | --- |
| **Methods** | **Phases** |
|  | Translation |
|  | Compilation |
|  | Instantiation |
| jspInit() | Initialization |
| \_jspService() | Servicing |
| jspDestroy() | Destruction |

helloworld\_jsp.class

JSP Container

+

Java Compiler

Helloworld\_jsp.java

Compilation

Container Generated helloworld\_jsp.java

Translation

JSP Container

Html, java, JSP Elements

Helloworld.jsp

The above process happens once.

**Implicit Objects**

|  |  |  |
| --- | --- | --- |
| Object Name | Type | Description |
| config | ServletConfig |  |
| request | HttpServletRequest | Each time a client requests a page the JSP engine creates a new object to represent that request.  The request object provides methods to get the HTTP header information including form data, cookies, HTTP methods etc. |
| response | HttpServletResponse | Just as the server creates the request object, it also creates an object to represent the response to the client.  The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes, etc. |
| session | HttpSession | The session object is used to track client session between client requests |
| application | ServletContext | This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the **jspDestroy()** method.  By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it. |
| page | Java.lang.Object | This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page.  The page object is really a direct synonym for the this object. |
| pageContext | Javax.servlet.jsp.PageContext | The pageContext object is used to represent the entire JSP page.  This object is intended as a means to access information about the page while avoiding most of the implementation details. |
| exception | Java.lang.Throwable | The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition. |
| out | Javax.servlet.jsp.JSPWriter | **out.print(dataType dt)**  Print a data type value  **out.println(dataType dt)**  Print a data type value then terminate the line with new line character.  **out.flush()**  Flush the stream. |

These Objects are the Java objects that the JSP Container makes available to the developers in each page and the developer can call them directly without being explicitly declared. JSP Implicit Objects are also called **pre-defined variables**.

**Scripting Elements**

Classified into three. They allow us to embed Java code into JSP.

**Declaration**

|  |  |
| --- | --- |
| <%!  int x;  int y;  void anyMethod() {  }  %> | Within the block you can put Java code |

**Expression**

|  |  |
| --- | --- |
| <%= a + b %>  <%= user.getName() %> | Any Java expression that evaluates to a value  The expression evaluates the expression and then sends the response back to the web browser |

**Scriptlet**

|  |  |
| --- | --- |
| <%  String num1 = request.getParameter(“num1”);  String num2 = request.getParameter(“num2”);  int result = num1 + num2;  %> | Any Java expression that evaluates to a value  The expression evaluates the expression and then sends the response back to the web browser. The code we put here goes into the \_jspService method that gets translated. |

|  |  |
| --- | --- |
| **Numbers.html** | **Addition.jsp** |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Addition</title>  </head>  <body>  <form action=*"addition.jsp"*>  <h2>Enter the numbers</h2>  Number1: <input type=*"text"* name=*"number1"* /><br/>  Number2: <input type=*"text"* name=*"number2"* /><br/>  <input type=*"submit"* name=*"submit"* />  </form>  </body>  </html> | <%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"*%>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Addition of two numbers</title>  </head>  <body>  <%  **int** num1 = Integer.parseInt(request.getParameter("number1"));  **int** num2 = Integer.parseInt(request.getParameter("number2"));  %>  Sum of <%=num1 %> and <%=num2 %> is <%= num1 + num2 %>  </body>  </html> |

**JSP Directives**

Three types of JSP Directives

1. **include**

include mechanism implements reusability. For example, a common file such as header.jsp, we can use the include directive and include the file for all the pages.

<%@include file=” “%>

header.jsp

home.jsp

login.jsp

1. **page**

The page directive has 13 attributes, most of which are not used due to becoming obsolete. The following are the attributes

**import** - allows us to import java packages into our JSP pages

**errorpage** – to point to an error page that handles errors in our entire application.

**iserrorpage** – this is a Boolean flag, which is true if the jsp file is used to display and error message.

**language**

**contenttype**

**session** – whether you want to create a new session

**extends** - if you want to extend another class other than the HttpServlet

**autoflush**

**info**

**pageencoding**

**isthreadsafe**

**buffer**

1. **taglib**

is a set of libraries. The taglib directives allows us to use the tags.

|  |  |
| --- | --- |
| numbers.html | addition.jsp |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Addition</title>  </head>  <body>  <form action=*"addition.jsp"*>  <h2>Enter the numbers</h2>  Number1: <input type=*"text"* name=*"number1"*/><br/>  Number2: <input type=*"text"* name=*"number2"*/><br/>  <input type=*"submit"* name=*"submit"* />  </form>  </body>  </html> | <%@ page language=*"java"*  contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"*%>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Addition of two numbers</title>  </head>  <body>  <%  **int** num1 = Integer.parseInt(  request.getParameter("number1"));  **int** num2 = Integer.parseInt(  request.getParameter("number2"));  %>    Sum of <%=num1 %> and <%=num2 %> is <%= num1 + num2 %>  </body>  </html> |

|  |  |
| --- | --- |
| openaccount.html | openaccount.jsp |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Open Account</title>  </head>  <body>  <form action=*"openaccount.jsp"* method=*"post"*>  Account Number:  <input type=*"text"* name=*"accno"* />  <br />  Last Name:  <input type=*"text"* name=*"lastname"* />  <br />  First Name:  <input type=*"text"* name=*"firstname"* />  <br />  Balance: <input type=*"text"* name=*"bal"* />  <br />  <input type=*"submit"* />  </form>  </body>  </html> | <%@ page import="java.sql.\*" language="java" contentType="text/html; charset=ISO-8859-1"  pageEncoding="ISO-8859-1"%>  <%!  Connection con;  PreparedStatement ps;    **public** **void** jspInit() {  **try** {  Class.forName("com.mysql.cj.jdbc.Driver");  con = DriverManager.getConnection(  "jdbc:mysql://localhost/mydb",  "root",  "P@ssW0rd");  ps = con.prepareStatement(  "insert into account value(?, ?, ?, ?)");  } **catch** (Exception e) {  e.printStackTrace();  }  }    **public** **void** jspDestroy() {  **try** {  ps.close();  con.close();  } **catch** (Exception e) {  e.printStackTrace();  }    }  %>    <%    **int** accno = Integer.parseInt(  request.getParameter("accno"));  String lastname = request.getParameter("lastname");  String firstname = request.getParameter("firstname");  **int** bal = Integer.parseInt(request.getParameter("bal"));    ps.setInt(1, accno);  ps.setString(2, lastname);  ps.setString(3, firstname);  ps.setInt(4, bal);    **int** result = ps.executeUpdate();    %>    <%@ include file="openaccount.html" %> |

Error Handling Example



|  |  |
| --- | --- |
| **errorgenerator.jsp** | **errorHandler.jsp** |
| <%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"* errorPage=*"errorHandler.jsp"* %>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Error Generator</title>  </head>  <body>  <%  String s = "123abc";    **int** num = Integer.parseInt(s);  %>  </body>  </html> | <%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"* isErrorPage=*"true"* %>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Error Page</title>  </head>  <body>  <p>Sorry, an error has occurred</p>  <%= exception.getMessage() %>  </body>  </html> |

**JSP Actions**

Also known as JSP tags are runtime instructions to the JSP container. There are two different types, namely Predefined and Custom tags.

**Predefined**

Include

Forward

Param

useBean

setProperty

getProperty

|  |
| --- |
| productDetails.html |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Product Details</title>  </head>  <body>  <form action=*"displayDetails.jsp"* method=*"post"*>  Product Id: <input type=*"text"* name=*"id"* /><br />  Product Name: <input type=*"text"* name=*"name"* /><br />  Product Description: <input type=*"text"* name=*"description"* /><br />  Product Price: <input type=*"text"* name=*"price"* /><br />  <input type=*"submit"* />  </form>  </body>  </html> |

|  |  |
| --- | --- |
| displayDetails.jsp | Product.java |
| <%@ page language=*"java"*  contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"*%>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Product Details</title>  </head>  <body>  <jsp:useBean id=*"product"*  class=*"com.samsonmarikwa.jsp.Product"*>  <jsp:setProperty  name=*"product"* property=*"\*"* />  </jsp:useBean>    Product Details  <br />  Id: <jsp:getProperty  property=*"id"* name=*"product"*/>  <br />  Name: <jsp:getProperty  property=*"name"* name=*"product"*/>  <br />  Description: <jsp:getProperty  property=*"description"* name=*"product"*/>  <br />  Price: <jsp:getProperty  property=*"price"* name=*"product"*/>  </body>  </html> | **package** com.samsonmarikwa.jsp;  **public** **class** Product {    **private** **int** id;  **private** String name;  **private** String description;  **private** **float** price;    **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getDescription() {  **return** description;  }  **public** **void** setDescription(String description) {  **this**.description = description;  }  **public** **float** getPrice() {  **return** price;  }  **public** **void** setPrice(**float** price) {  **this**.price = price;  }    } |

|  |
| --- |
| **MVC Design Pattern** |

MVC is a design pattern or framework that splits the web layer into three parts.

### **Model**

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

### **View**

The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

### **Controller**

Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

**Advantages**

Easier Maintenance

Parallel Development

|  |
| --- |
| **Creating Custom Tags** |

A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The Web container then invokes those operations when the JSP page's servlet is executed.

To create a custom tag we need three things:  
**1) Tag handler class**: In this class we specify what our custom tag will do when it is used in a JSP page. To implement the tag handler class, we extend the TagSupport which already implements the Tag interface or implement the Tag interface.

Life Cycle methods are

setPageContext()

doStartTag()

doEndTag()

release()

**2) TLD file**: Tag Lib descriptor file where we will specify our tag name, tag handler class and tag attributes. The container will know which Tag handler class corresponds to which tag name.  
**3) JSP page**: A JSP page where we will be using our custom tag.

|  |  |
| --- | --- |
| customtag.tld | Index.html |
| <?xml version="1.0" encoding="UTF-8"?>  <taglib>  <tlib-version>5.0</tlib-version>  <jsp-version>3.0</jsp-version>  <short-name>userinformation</short-name>  <info>This library displays user information</info>  <uri>http://samsonmarikwa.com</uri>  <tag>  <name>displayuser</name>  <tag-class>com.samsonmarikwa.customtags.ResultHandler</tag-class>  <info>This tag displays user information</info>  </tag>  </taglib> | <!DOCTYPE html>  <html>  <head>  <meta charset="ISO-8859-1">  <title>User Input</title>  </head>  <body>  <form action="userdetails.jsp">  Email: <input type="text" name="email" /><br />  <input type="submit" value="submit" />  </form>  </body>  </html> |

|  |  |
| --- | --- |
| Userdetails.jsp |  |
| <%@ taglib prefix="samson" uri="http://samsonmarikwa.com" %>  <%-- <%@ taglib prefix="samson" uri="WEB-INF/customtag.tld" %> --%>  <!DOCTYPE html>  <html>  <head>  <meta charset="ISO-8859-1">  <title>Display User Info</title>  </head>  <body>  <samson:displayuser />  </body>  </html> | package com.samsonmarikwa.customtags;  import java.io.IOException;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  import jakarta.servlet.ServletRequest;  import jakarta.servlet.jsp.JspException;  import jakarta.servlet.jsp.JspWriter;  import jakarta.servlet.jsp.tagext.Tag;  import jakarta.servlet.jsp.tagext.TagSupport;  public class ResultHandler extends TagSupport {  // instead of implementing Tag, we extend TagSupport  // This allows us to implement only the methods we need  Connection con;  PreparedStatement stmt;  public ResultHandler() {  try {  Class.forName("com.mysql.cj.jdbc.Driver");  con = DriverManager.getConnection("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");  stmt = con.prepareStatement("select \* from user where email = ?");    } catch (ClassNotFoundException | SQLException e) {  e.printStackTrace();  }  }    @Override  public int doStartTag() throws JspException {  ServletRequest request = pageContext.getRequest();  String email = request.getParameter("email");    try {  stmt.setString(1, email);  ResultSet resultSet = stmt.executeQuery();  JspWriter out = pageContext.getOut();  if (resultSet.next()) {  out.print("User Details are: <br/> First Nmae: ");  out.print(resultSet.getString(1));  out.print("<br/>Last Nmae: ");  out.print(resultSet.getString(2));  } else {  out.print("Invalid email entered");  }  } catch (SQLException | IOException e) {  e.printStackTrace();  }  return Tag.SKIP\_BODY; // we do not have a body in the request  }    @Override  public void release() {  try {  stmt.close();  con.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| JSTL - |

The JSP Standard Tag Library (JSTL) represents a set of tags to simplify the JSP development.

**Advantage of JSTL**

* Fast Development JSTL provides many tags that simplify the JSP.
* Code Reusability We can use the JSTL tags on various pages.
* **No need to use scriptlet tag** It avoids the use of scriptlet tag.

**JSTL Tags**

There JSTL mainly provides five types of tags:

|  |  |
| --- | --- |
| Tag Name | Description |
| Core tags | The JSTL core tag provides variable support, url management, flow control, etc. The url for the functions tag is <http://java.sun.com/jsp/jstl/core>. The prefix is of core tag is c. |
| Function tags | The functions tag provide support for string manipulation and string length. The URL for the functions tags is http://java.sun.com/jstl/functions and prefix is fn. |
| Formatting tags | The Formatting tags provide support for message formatting, number and date formatting, etc. The URL for the Formatting tags is **http://java.sun.com/jsp/jstl/fmt** and prefix is **fmt**. |
| XML tags | The XML tags provide flow control, transformation, etc. The URL for the XML tags is **http://java.sun.com/jsp/jstl/xml** and prefix is **x**. |
| SQL tags | The JSTL SQL tags provide SQL support. The URL for the SQL tags is **http://java.sun.com/jsp/jstl/sql** and prefix is **sql**. |

|  |
| --- |
| <%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jstl/core\_rt"* %>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Cout Demo</title>  </head>  <body>  <c:out value=*"*${10 + 9}*"* />  </body>  </html> |

# **Scope of JSP Objects**

Last modified on July 25th, 2014 by Joe.

The availability of a JSP object for use from a particular place of the application is defined as the scope of that JSP object. Every object created in a JSP page will have a scope. Object scope in JSP is segregated into four parts and they are page, request, session and application.

* **page**  
  ‘page’ scope means, the JSP object can be accessed only from within the same page where it was created. The default scope for JSP objects created using <jsp:useBean> tag is page. JSP implicit objects out, exception, response, pageContext, config and page have ‘page’ scope.
* **request**A JSP object created using the ‘request’ scope can be accessed from any pages that serves that request. More than one page can serve a single request. The JSP object will be bound to the request object. Implicit object request has the ‘request’ scope.
* **session**  
  ‘session’ scope means, the JSP object is accessible from pages that belong to the same session from where it was created. The JSP object that is created using the session scope is bound to the session object. Implicit object session has the ‘session’ scope.
* **application**A JSP object created using the ‘application’ scope can be accessed from any pages across the application. The JSP object is bound to the application object. Implicit object application has the ‘application’ scope.

|  |
| --- |
| JSTL |
| <%@page import=*"java.util.ArrayList"*%>  <%@page import=*"java.util.List"*%>  <%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jstl/core\_rt"* %>  <%@ taglib prefix=*"fmt"* uri=*"http://java.sun.com/jsp/jstl/fmt"* %>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Cout Demo</title>  </head>  <body>  <c:out value=*"*${10 + 9}*"* />  <c:set var=*"testScore"* value=*"*${10}*"* scope=*"session"* /> <!-- scope can be page, session, request, application -->  <c:out value=*"*${testScore}*"* />  <c:if test=*"*${testScore>=80}*"*>  <p>Your score is awesome ${testScore}</p>  </c:if>  <c:choose>  <c:when test=*"*${testScore >= 80 }*"*>  A Grade  </c:when>  <c:when test=*"*${testScore >= 60 && testScore < 80 }*"*>  B Grade  </c:when>  <c:when test=*"*${testScore >= 40 && testScore < 60 }*"*>  C Grade  </c:when>  <c:when test=*"*${testScore >= 20 && testScore < 40 }*"*>  D Grade  </c:when>  <c:otherwise>  Fail  </c:otherwise>  </c:choose>  <c:forEach var=*"i"* begin=*"1"* end=*"4"*>  <c:out value=*"*${i}*"* />  </c:forEach>  <%  List<String> studentNames = **new** ArrayList<>();  studentNames.add("Samson");  studentNames.add("Annet");  studentNames.add("Laureen");  studentNames.add("Walter");  request.setAttribute("studentNames", studentNames);  %>  <br />  <c:forEach var=*"studentName"* items=*"*${studentNames}*"*>  <c:out value=*"*${studentName}*"* />  <br />  </c:forEach>  <c:remove var=*"testScore"* /><br/>  After removal the value is: <c:out value=*"*${testScore}*"* />  <c:set var=*"accountBalance"* value=*"12345.4567"* />  <fmt:parseNumber var=*"i"* type=*"number"* value=*"*${accountBalance}*"*/>  <p>Amount is: <c:out value=*"*${i}*"* /></p>  <p>Formatted Number: <fmt:formatNumber value=*"*${accountBalance}*"* type=*"currency"* /></p>  <p>Formatted Number: <fmt:formatNumber value=*"*${accountBalance}*"* type=*"number"* /></p>  <p>Formatted Number: <fmt:formatNumber value=*"*${accountBalance}*"* type=*"number"* maxFractionDigits=*"2"* /></p>  <p>Formatted Number: <fmt:formatNumber value=*"*${accountBalance}*"* type=*"percent"* /></p>  <p>Formatted Number: <fmt:formatNumber value=*"*${accountBalance}*"* type=*"number"* pattern=*"#,###.##$"* /></p>  <c:set var=*"myDate"* value=*"28-09-1966"* />  <fmt:parseDate var=*"parsedDate"* value=*"*${myDate}*"* pattern=*"dd-MM-yyyy"* />  <p><c:out value=*"*${parsedDate}*"* /></p>  </body>  </html> |

|  |
| --- |
| Security |

Setting up Basic Authentication

|  |  |
| --- | --- |
| tomcat-users.xml |  |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <tomcat-users version=*"1.0"*  xmlns=*"http://tomcat.apache.org/xml"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://tomcat.apache.org/xml tomcat-users.xsd"*>  <role rolename=*"myuserrole"* />  <user username=*"myuser"* password=*"mypassword"* roles=*"myuserrole"* />  </tomcat-users> | Create users in the file tomcat-users.xml |

|  |  |
| --- | --- |
| simple.jsp |  |
| <%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"*%>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Secured JSP</title>  </head>  <body>  <b>Secured JSP</b>  </body>  </html> |  |

|  |  |
| --- | --- |
| Web.xml |  |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"https://jakarta.ee/xml/ns/jakartaee"*  xmlns:web=*"http://xmlns.jcp.org/xml/ns/javaee"*  xsi:schemaLocation=*"https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-app\_5\_0.xsd"*  id=*"WebApp\_ID"* version=*"5.0"*>  <security-constraint>  <web-resource-collection>  <web-resource-name>MyResources</web-resource-name>  <url-pattern>/\*</url-pattern>  </web-resource-collection>  <auth-constraint>  <role-name>myuserrole</role-name>  </auth-constraint>  </security-constraint>  <login-config>  <auth-method>BASIC</auth-method>  <realm-name>FILE</realm-name>  </login-config>  </web-app> | /\* all resources  For a single resource, you can put a name simple.jsp  or url pattern of the servlet  role-name we are taking it from the tomcat-users.xml file  auth-method can be FORM if using Form Based authentication  realm-name could be database, file etc. In this example, we put file because we are reading the username and password from the file |

Form Based Authentication

|  |  |
| --- | --- |
| Web.xml |  |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns=*"https://jakarta.ee/xml/ns/jakartaee"*  xmlns:web=*"http://xmlns.jcp.org/xml/ns/javaee"*  xsi:schemaLocation=*"https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakartaee/web-app\_5\_0.xsd"*  id=*"WebApp\_ID"* version=*"5.0"*>  <security-constraint>  <web-resource-collection>  <web-resource-name>MyResources</web-resource-name>  <url-pattern>/\*</url-pattern>  </web-resource-collection>  <auth-constraint>  <role-name>myuserrole</role-name>  </auth-constraint>  </security-constraint>  <login-config>  <auth-method>FORM</auth-method>  <realm-name>FILE</realm-name>  <form-login-config>  <form-login-page>/login.jsp</form-login-page>  <form-error-page>/error.jsp</form-error-page>  </form-login-config>  </login-config>  </web-app> | /\* all resources  For a single resource, you can put a name simple.jsp  or url pattern of the servlet  role-name we are taking it from the tomcat-users.xml file  auth-method can be FORM if using Form Based authentication  realm-name could be database, file etc. In this example, we put file because we are reading the username and password from the file |

|  |
| --- |
| Login.jsp |
| <%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"*%>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>LOGIN</title>  </head>  <body>  <form action=*"j\_security\_check"* method=*"post"*>  User Name: <input name=*"j\_username"* /><br />  Password: <input type=*"password"* name=*"j\_password"* /><br />  <input type=*"submit"* value=*"Submit"* />  </form>  </body>  </html> |

|  |
| --- |
| Error.jsp |
| <%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"*%>  <!DOCTYPE html>  <html>  <head>  <meta charset=*"ISO-8859-1"*>  <title>Error Page</title>  </head>  <body>  <h1>Please enter valid username and password</h1>  </body>  </html> |

|  |
| --- |
| Connection Pooling |

Connection pooling allows the container to create a set of JDBC connections when it starts up. Thee connections can then be used by servlets, jsps or any other web resource to do what they have to do and send the connection back once they are done with their work.

Major advantages

Performance – clients can use a ready made connection. The step of connecting to the database is gone.

Reuse – once a client is done, it returns the connection to the pool instead of closing it.

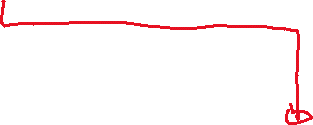
**Connection Pooling configuration**

There are two steps

* Copy Driver jar to Apache tomcat lib folder
* Configure resource element in the context.xml. This tells Tomcat about the database connection string. The file is under tomcat conf folder if using tomcat externally. In eclipse, press Ctrl Shift R and search for context.xml file.

|  |
| --- |
| Add the following line |
| <Resource name=*"myds"* auth=*"Container"* type=*"javax.sql.DataSource"* driverClassName=*"com.mysql.cj.jdbc.Driver"* url=*"jdbc:mysql://localhost/mydb"* username=*"root"* password=*"P@ssW0rd"* /> |

**Acquiring the Connection**



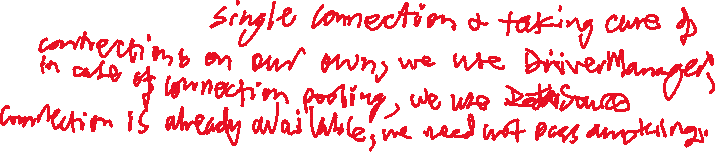
Connect to the naming service

Look up for the DataSource using JNDI method

Get the connection



|  |
| --- |
| 704574314, URL=jdbc:mysql://localhost/mydb, MySQL Connector/J is the output |
| **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  **try** {  Context context = **new** InitialContext();  DataSource ds = (DataSource) context.lookup("java:comp/env/myds");  Connection connection = ds.getConnection();  System.***out***.println(connection);  } **catch** (NamingException | SQLException e) {  e.printStackTrace();  }  } |



|  |
| --- |
| Advanced JDBC |

**Batch Processing**

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

**Steps for batching**

1. Add the dml statement

stmt.addBatch(String dml);

1. Execute the batch

int result[] = stmt.executeBatch();

Create Java project

Add the mysql driver jar file

Configure build path so the jar file is accessible.

Graphical user interface, application

Description automatically generated

|  |
| --- |
| Batch updates |
| **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.SQLException;  **import** java.sql.Statement;  **public** **class** BatchDAO {  **public** **static** **void** main(String[] args) {    **try** (Connection con = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");  Statement stmt = con.createStatement();  ) {  stmt.addBatch("insert into account values(10, 'Marikwa', 'Cynthia', 15000)");  stmt.addBatch("insert into account values(20, 'Kanakembizi', 'Ariel', 25000)");  stmt.addBatch("insert into account values(30, 'Kanakembizi', 'Tonderai', 15000)");    **int**[] result = stmt.executeBatch();    **for** (**int** i = 0; i < result.length; i++) {  System.***out***.println(result[i]);  }    } **catch** (SQLException e) {  e.printStackTrace();  }  }  } |

**ResultSetMetaData**

|  |
| --- |
| ResultsetMetaData – get data about the ResultSet |
| **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.ResultSetMetaData;  **import** java.sql.SQLException;  **import** java.sql.Statement;  **public** **class** MetadataDAO {  **public** **static** **void** main(String[] args) {  **try** (Connection con = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");  Statement stmt = con.createStatement();) {  ResultSet resultSet = stmt.executeQuery("select \* from account");  ResultSetMetaData resultSetMetaData = resultSet.getMetaData();  **int** columnCount = resultSetMetaData.getColumnCount();  **for** (**int** i = 1; i <= columnCount; i++) {  System.***out***.println(resultSetMetaData.getColumnName(i));  System.***out***.println(resultSetMetaData.getColumnTypeName(i));  }  } **catch** (SQLException e) {  e.printStackTrace();  }  }  } |

**Transaction Management**

A transaction is a logical unit of work in which all of it gets done or none of it.

An example is Money Transfer.

**Transaction Properties**

Every transaction should have 4 properties, ACID. The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

A – Automicity

Atomicity guarantees that all of the commands that make up a transaction are treated as a single unit and either succeed or fail together. This is important as in the case of an unwanted event, like a crash or power outage, we can be sure of the state of the database. The transaction would have either completed successfully or been rollbacked if any part of the transaction failed.  
Atomicity is also known as the ‘All or nothing rule’.

C – Consistency

Consistency guarantees that changes made within a transaction are consistent with database constraints. This includes all rules, constraints, and triggers. If the data gets into an illegal state, the whole transaction fails.

Going back to the money transfer example, let’s say there is a constraint that the balance should be a positive integer. If we try to overdraw money, then the balance won’t meet the constraint. Because of that, the consistency of the ACID transaction will be violated and the transaction will fail.

It refers to the correctness of a database.

I – Isolation

Isolation ensures that all transactions run in an isolated environment. That enables running transactions concurrently because transactions don’t interfere with each other.

For example, let’s say that our account balance is $200. Two transactions for a $100 withdrawal start at the same time. The transactions run in isolation which guarantees that when they both complete, we’ll have a balance of $0 instead of $100.

D - Durability

Durability guarantees that once the transaction completes and changes are written to the database, they are persisted. This ensures that data within the system will persist even in the case of system failures like crashes or power outages.

What is an ACID transaction example?

Let's continue with the banking example we discussed earlier in the article where money is being transferred from one account to another. Let's examine each of the ACID properties in this example:

* Atomicity: Money needs to both be removed from one account and added to the other, or the transaction will be aborted. Removing money from one account without adding it to another would leave the data in an inconsistent state.
* Consistency: Consider a database constraint that an account balance cannot drop below zero dollars. All updates to an account balance inside of a transaction must leave the account with a valid, non-negative balance, or the transaction should be aborted.
* Isolation: Consider two concurrent requests to transfer money from the same bank account. The final result of running the transfer requests concurrently should be the same as running the transfer requests sequentially.
* Durability: Consider a power failure immediately after a database has confirmed that money has been transferred from one bank account to another. The database should still hold the updated information even though there was an unexpected failure.

Controlling Transactions

Commit – tells the database the transaction boundary.

Rollback – used when an exception is thrown and we want to return everything to a prior state.

Savepoint – we can commit or rollback a transaction to a certain point.

In the JDBC, we control transactions by using the Connection interface.

|  |
| --- |
| ACID |
| **package** com.samsonmarikwa.transactionmanagement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.SQLException;  **import** java.sql.Statement;  **public** **class** TransactionDAO {  **public** **static** **void** main(String[] args) {  Connection con = **null**;  Statement stmt = **null**;  **try** {  con = DriverManager.*getConnection*("jdbc:mysql://localhost/mydb", "root", "P@ssW0rd");  stmt = con.createStatement();    con.setAutoCommit(**false**);  stmt.executeUpdate("update account set bal=bal-500 where accno=10");  stmt.executeUpdate("update account set bal=bal+500 where accno=20");  con.commit();  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  **try** {  con.rollback();  stmt.close();  con.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }  } |