|  |
| --- |
| **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

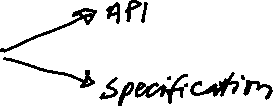
WEB-INF

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.

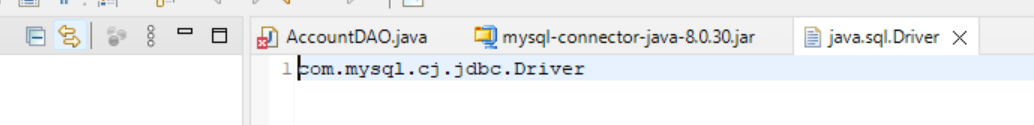
The Service Provider is installed in the form of extensions, a jar file which we place in the application classpath, the Java extension classpath or the user-defined classpath.

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.

Init(ServletConfig config)

Servlet

Container

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

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

**Configure using web.xml**

**Configure using Annotations**

@WebServlet(urlPatterns=”/addServlet”,

initParams={@WebInitParam(name=”dbUrl”, value=”jdbc:mysql://localhost/mydb”),

@WebInitParam(name=”dbUser”, value=”root”),

@WebInitParam(name=”dbPassword”, value=”test”)})