Introduction to REST

REST

Stands for **Representational State Transfer**.

Defined originally by Roy Fielding in his dissertation in 2000, REST is an *architectural style* that outlines communication between a client and server over the web.

Essentially for a web service to be RESTful it has to adhere to a set of guidelines or constraints.

A RESTful server should not retain information about the state of the client.

Clients communicate with the server through an interface that is standard in that it too follows another set of constraints: "defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state." - Roy Fielding

In a RESTful system, the server creates an object or resource and returns the *state* of that object (the values contained within the object) when requested by the client.

Exposing and Consuming RESTful API endpoints

API - Application Programming Interface

An API is a software intermediary that allows two applications to talk to each other.

For example, all airline ticket booking applications will use an API exposed by the airline company. Anytime a customer uses any such application and books a flight ticket, the application passes the passenger and flight booking information to the API. The booking API will process the data and book the ticket for the customer and the application will get a success response with booking details in return. The booking applications do not know and need not to know how the API works internally. All they are required to do is pass the booking information in a well-defined format to the API and wait for the response. Similarly, any application/mobile app can use an API or expose an API to other software.

A RESTful API (also known as a RESTful WebService) is a web service implemented using HTTP protocol and the REST Architectural Constraints.

RESTful APIs typically are built on top of HTTP.

Example - Exposing / Consuming REST API endpoints

Here, we setup a JSON server and publish a REST API. JSON Server is a simple project that helps you to setup a REST API with CRUD operations very fast. Following steps demonstrate how we can interact with the REST API through the HTTP endpoints.

Step 1 - Install the JSON Server using the `npm install -g json-server` command. JSON Server is available as a NPM package.

Step 2 - Create a new JSON file with name `db.json`. This file contains the JSON data which will be exposed by the REST API.

Below, the `db.json` file consists of one employee object which has three data sets assigned. Each employee object is consisting of four properties: `id`, `first_name`, `last_name` and `email`.

```
{
  "employees": [
    {
      "id": 1,
      "first_name": "Sebastian",
      "last_name": "Eschweiler",
      "email": "sebastianE@gmail.com"
    },
    {
      "id": 2,
      "first_name": "Steve",
      "last_name": "Palmer",
      "email": "steveP@gmail.com"
    },
    {
      "id": 3,
      "first_name": "Ann",
      "last_name": "Smith",
      "email": "annS@gmai;.com"
    }
  ]
}
```

Step 3 - Running the JSON Server using the `json-server --watch db.json` command. We need to pass over the file name containing our JSON structure - `db.json`- as a parameter. By using `watch` parameter we're making sure that the server is started in watch mode which means that it watches for file changes and updates the exposed API accordingly. Now, our JSON server is up and running in the port 3000.

The following HTTP endpoints are created automatically by the JSON server:

```
GET    /employees
GET    /employees/{id}
POST   /employees
PUT    /employees/{id}
PATCH  /employees/{id}
DELETE /employees/{id}
```

NOTE: You have to download the Postman before testing the REST API Endpoints with Postman.
Step 4 - Now, we are going to test the REST API Endpoints exposed by the JSON Server with Postman.

Send the HTTP GET request to http://localhost:3000/employees which shows the following result:

When you send the HTTP GET request with `id` parameter to http://localhost:3000/employees/1, you able see the following result:

When you send the HTTP POST request to http://localhost:3000/employees with JSON data on the Body, you able see the following result:

.

Similarly, you can try the remaining HTTP request methods. For all the POST, PUT, PATCH or DELETE requests you make, changes will be automatically saved to the `db.json` file.

Jackson Library

Working with Jackson API

The Jackson API is used to convert Java objects into JSON format to send in an HTML response. The JSON format is commonly used for transmitting data between a server and a client in a web application.

In this section, we'll discuss how we convert java objects into JSON using the Jackson API and send it in a servlet response.

Jackson ObjectMapper

The ObjectMapper API is used for data-binding. It uses reader/writer methods to perform the conversion between Java objects and JSON.

- **writeValue() method** - used to convert Java object to JSON.

Example:

```
ObjectMapper mapper = new ObjectMapper();
User user = new User();

//Object to JSON in String
String jsonInString = mapper.writeValueAsString(user);
```

- **readValue() method** - used to convert JSON to Java object.

Example:

```
ObjectMapper mapper = new ObjectMapper();
String jsonInString = "{'name' : 'Adam'}";

//JSON from String to Object
User user = mapper.readValue(jsonInString, User.class);
```

Example

- First, we need to add the Jackson dependency in our pom.xml file.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.11.0</version>
</dependency>
```

- Then, we create a POJO class for converting this class object into JSON.

```
public class User
{
   private Integer id;
   private String firstName;
   private String lastName;

   //Getters and setters
  }
```

- Servlets sends the JSON output as a response to the client using **ObjectMapper**.

```
public class HomeServlet extends HttpServlet {

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

                //converts the User object into JSON and sends the JSON data to the
Client as a response

                response.getWriter().write(

                        new ObjectMapper().writeValueAsString(

                                new User("1234", "John", "Adams")));

        }

}
```

References

- ObjectMapper Documentation

Exception and Error Handling

The Servlet API allows us to handle exceptions and errors caused during the execution of a servlet and still send a useful response to the user. The **deployment descriptor** file just needs to be configured to handle the exceptions/errors thrown by a servlet.

The **`<error-page>`** element used to specify the invocation of servlets in response to certain **exceptions** or **HTTP status codes**. The following elements are used within an `<error-page>` element to handle error or exception:

- `<error-code>` - used to specify a valid HTTP error code. *For example, 404, 403, 500, etc.*
- `<exception-type>` - used to specify a fully-qualified class name of a Java exception type. *For example,* *`javax.servlet.ServletException,java.io.IOException`*, *`java.lang.RuntimeException`*, *etc.*
- `<location>` - used to specify the location of the resource which is displayed to the user in case of an error. *This might be a servlet, an HTML page, a JSP page, or something else.*

Request Attributes for Errors/Exceptions:

Before the servlet container invokes the servlet to handle the exception, it sets some attributes in the request to get useful information about the exception. Some of these are:

- javax.servlet.error.status_code
- javax.servlet.error.servlet_name
- javax.servlet.error.exception
- javax.servlet.error.request_uri
- javax.servlet.error.exception_type
- javax.servlet.error.message

Note that while these appear to be fully-qualified class names, they are neither packages, classes, nor variables. They are the names of attributes, and are treated as Strings.

Example

The web.xml file for mapping an exception to a servlet.

```
<error-page>

      <exception-type>java.lang.ArithmeticException</exception-type>

     <location>/errorHandler</location>

</error-page>
```

The Servlet class which throws an exception:

```
@WebServlet(name = "testServlet", urlPatterns = {"/test"}, loadOnStartup = 1)

public class TestServlet extends HttpServlet {



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



        int i = 1 / 0;

    }

}
```

Here, we use @WebServlet annotation to declare a servlet.

The servlet class that handles the error caused by the above servlet:

```
@WebServlet(name = "errorHandlerServlet", urlPatterns = {"/errorHandler"}, loadOnStar
tup = 1)

public class ErrorHandlerServlet extends HttpServlet {



    @Override

    protected void doGet (HttpServletRequest request, HttpServletResponse response) t
hrows ServletException, IOException {
        PrintWriter out = response.getWriter();

        Exception exception = (Exception) req.getAttribute("javax.servlet.error.excep
tion");

        Class exceptionClass = (Class) req.getAttribute("javax.servlet.error.exceptio
n_type");

        Integer status_code = (Integer) req.getAttribute("javax.servlet.error.status_
code");

        String errorMessage = (String) req.getAttribute("javax.servlet.error.message"
);

        String requestUri = (String) req.getAttribute("javax.servlet.error.request_ur
i"

        String servletName = (String) req.getAttribute("javax.servlet.error.servlet_n
ame");

        out.println("Exception: " + exception);
        out.println("Exception Type: " + exceptionClass);
        out.println("HttpError Status code: " + status_code);
        out.println("ErrorMessage: " + errorMessage);
        out.println("Request URI: " + requestUri);
        out.println("Servlet Name: " + servletName);
    }

}
```

References

- Further Reading on Servlet Exception Handling

HttpSession API

The Servlet API provides **HttpSession Interface**, which provides a way to identify a user and to store information about that user. For the client's first request, the Servlet Container generates a **unique session ID** and gives it back to the client with a response. Thereafter, the client sends the session ID with each request to the server.

The **getSession()** method of the *HttpServletRequest* object returns a user's session. Any servlet can access the *HttpSession* object using getSession() method.

Example for creating the *HttpSession* object:

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

        HttpSession session = request.getSession();

}
```

The commonly used HttpSession Interface methods are listed below:

- **setAttribute(key,object)** - used to bind an object to the session, using the key specified.
- **getAttribute(String)** - used to retrieve a specific saved object from the session object, using its key.
- **removeAttribute(key)** - used to remove the object bound with the specified key from the session.
- **invalidate()** - destroys the session.
- **getId()** - returns the unique ID assigned to the session.
- **getCreationTime()**- returns the time when the session was created
- **getLastAccessedTime()** - returns the last time the client sent a request associated with the session
- **getMaxInactiveInterval()** - returns the maximum time interval, in seconds.
- **setMaxInactiveInterval(int interval)** - Specifies the time, in seconds, after servlet container will invalidate the session.

Example:

Create two servlets: a **SourceServlet** and a **TargetServlet** which will process data submit through a form in an HTML file called **user.html**.

1. Upon submitting the `userName` in the **user.html** form, the **SourceServlet will create a `session` object.
2. The **SourceServlet** will use the `session` object to `setAttribute` to the user, and then use the `PrintWriter` to produce a hyperlink to the **TargetServlet**.
3. The **TargetServlet** can retrieve the user by once again instantiating a `session` and then getting the `"user"` attribute.
4. By using `PrintWriter`, the **TargetServlet** will print username that the client initially input in the **user.html** form as well as more information about the session.

Create the HTML file `user.html`

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
        <h1>Enter User Name:</h1>
        <form method="post" action="sourceServlet">
                User Name :<input name="userName">
                           <input type="submit" value="send" name="submitButton">
        </form>
</body>
</html>
```

Create `SourceServlet.java`

```
@WebServlet("/sourceServlet")
public class SourceServlet extends HttpServlet {
        private static final long serialVersionUID = 1L;

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

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

                // instantiate a session objection
                HttpSession session = request.getSession();

                // set an attribute that can be retrieved by the next servlet
                session.setAttribute("user", username);

                // create a hyperlink to go to the next servlet which will process t
he request
```

```
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println("<a href='targetServlet'>Click Here to get the UserName<
/a>");
        }
}
```

Here we are using `@WebServlet` annotations to map our servlets.

Notice that the annotation for our sourceServlet correlates with the `action=` attribute of our `user.html` form.

Create `TargetServlet.java`

```
@WebServlet("/targetServlet")
public class TargetServlet extends HttpServlet {
        private static final long serialVersionUID = 1L;

        // this is called when the hyperlink from SourceServlet is clicked
        protected void doGet(HttpServletRequest request, HttpServletResponse respons
e) throws ServletException, IOException {

                // instantiate the session object and use the .getSession() method o
n the request
                HttpSession session = request.getSession();

                // retrieve the attribute from the session
                String username = (String)(session.getAttribute("user"));

                String sessionId = session.getId();

                long creationTime = session.getCreationTime();
                long lastAccessedTime = session.getLastAccessedTime();

                Date createDate= new Date(creationTime);
                Date lastAccessedDate= new Date(lastAccessedTime);
                // print the retrieved attribute using the PrintWriter
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println("<h1>Username is: " + username + " </h1>");

                out.println(" Your Session Infomation: <br/>");
                out.println("ID: " + sessionId  + "<br/>");
                out.println("Session Created Date: " + createDate + "<br/>");
                out.println("Session Created Time: " + creationTime + "<br/>");
                out.println("Last Accessed Date : " + lastAccessedDate + "<br/>");
                out.println("Last Accessed Time: " + lastAccessedTime + "<br/>");
        }
}
```

Right click on `user.html` and click Run On Server.

You should be able to enter a username and retrieve that information in your **TargetServlet** by accessing the data stored in the associated session.

Session Management in servlets

The HTTP protocol is a **stateless protocol**, which means no client information stored in the server. The server considers every request form the same client as a new, independent request. However, this means that a server cannot keep a user "logged in" naturally.

Instead, the client will have to re-identify itself with each request it sends. One workaround is to have the server create a **session** for each client request, which the client can re-associate itself with in each request by sending the **session ID**.

For example, when a user logs in to a website, a token proving a successful login will be associated with their session, and the session ID will be returned to the client. Whatever web page user visits after logging in, their browser will send that session ID with each request, and the server can check that session to validate that the user was successfully authenticated.

Session tracking is a mechanism that **servlets use to maintain state** about a series of requests from the same user across a period of time.

A session stores the **unique identification information about the client** that we can get for all requests that client makes. There are four different techniques used by the Servlet application for session management.

- Cookies
- Hidden form fields
- URL Rewriting
- The HttpSession API

References

- Session Management Further Reading

URL Rewriting

URL Rewriting is a process by which a **unique session ID gets appended to each request URL**, so the server can identify the user session.

URL Rewriting maintains the session and works even the user's browser doesn't support cookies. This makes it one of the ways in which we can provide a unique id in request and response, alongside implementing the Session Interface and Cookies Management.

Example

We will create an **HTML page** to capture a username value from the client, a **FirstServlet** which will print the username, and then provide the url pass control to **SecondServlet** using url rewriting.

Program Flow

Create `index.html` Form

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>HTML Form</title>
</head>
<body>
        <form action="FirstServlet" method="get">
                        Name:<input type="text" name="userName" /><br><br/>
                             <input type="submit" value="Submit" />
        </form>
```

```
</body>
</html>
```

Create `FirstServlet.java`

```
@WebServlet("/FirstServlet")
public class FirstServlet extends HttpServlet {


    public void doGet(HttpServletRequest request, HttpServletResponse response)
    {
        try {
            response.setContentType("text/html");


            PrintWriter out = response.getWriter();


            // request.getParameter takes the value from index.html file where name i
s "userName"
            String n = request.getParameter("userName");
            out.print("Welcome " + n);




            /** url rewriting is used for creating a session - it will redirect  you 
to SecondServlet page
                Notice that we have set the query parameter ?uname equal to the userN
ame parameter we retrieved from the request
            **/
            out.print("<a href='SecondServlet?uname=" + n + "'>visit</a>");


            out.close();
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

We apply URL rewriting in the FirstServlet when we pass control to the SecondServlet.

```
out.print("<a href='SecondServlet?uname=" + n + "'>pass control to secondServlet</a>"
);
```

Create `SecondServlet.java`

```
@WebServlet("/SecondServlet")
public class SecondServlet extends HttpServlet {


    public void doGet(HttpServletRequest request, HttpServletResponse response)
    {
        try {


            response.setContentType("text/html");


            PrintWriter out = response.getWriter();


            // use request.getParameter() to get the value from the url that we have
rewritten in  FirstServlet
            String n = request.getParameter("uname");
            out.print("Hello " + n);


            out.close();
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

The benefit of URL rewriting is that it doesn't depend upon cookies and will work whether cookies are enabled or disabled. Extra form submission is not required on all pages.

References

- Further reading on URL Rewriting

Cookies

A cookie is a key-value pair of information sent by the server to the client, which the client will store. The client (usually a web browser) can send this cookie in the HTTP request header for all subsequent requests until the cookie becomes invalid.

The Servlet container checks the request header for cookies, most commonly to get the session information from the cookie, which it uses it to retrieve the associated session data stored in the server.

We can classify the cookies into two types based on their expiry time:

- **Non-persistent cookie** - Cookie becomes expired when the user closes the browser.
- **Persistent Cookies** - Cookie expires only if the user logs out of the website. The cookie is stored on the browser even the user closes the browser each time.

Creating Cookies with Servlets

To send cookies to the client, we need to create a Cookie object, set the maximum age for the cookie, and place the cookie in the HTTP response header. The `Cookie(String name, String value)` constructor defined in the **javax.servlet.http.Cookie** class can be used to create a cookie with a specified name and value. We can use the `setMaxAge()` method to set the maximum age for the particular cookie in seconds. We can use the `response.addCookie()` method to place the cookie in the HTTP response header.

Example:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)throws Ser
vletException, IOException {


    //Creating a Cookie object

    Cookie cookie = new Cookie("name","Adam");


    // Set expiry date after 24 Hrs (86,400seconds)

    cookie.setMaxAge(86400);


    // Add the cookie in the response header

    response.addCookie(cookie);

}
```

Reading Cookies with Servlets

To read cookies, We need to create an array of javax.servlet.http.Cookie objects by calling the `getCookies()` method of HttpServletRequest. Then the `getName()` and `getValue()` methods used to access each cookie and associated value.

Example:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)throws Ser
vletException, IOException {


    //getting all the cookies
    Cookie cookies[]=request.getCookies();


    PrintWriter out = response.getWriter();
    for(Cookie c : cookies){
        out.println("Name: "+c.getName()+" & Value: "+c.getValue());
    }
}
```

References

- Further Reading on Cookies

Front Controller Design Pattern

The front controller design pattern provides a **single handler** for all the incoming requests for a resource in an application, and then **dispatches** the requests to the appropriate secondary handler for that type of request. The front controller may use other **helper** APIs to achieve the dispatching mechanism.

Front Controller - The Front controller is a single entry point for all requests, and routes incoming user requests. It delegates to a dispatcher to perform action and view management.

Dispatcher - A dispatcher is responsible for the action and view management, including locating and routing to the specific actions that will service a request, and finding the appropriate view.

Helper - We use Helper classes to break out specific features and make the application easier to build and maintain. They can be used for the retrieval of content, validation of user-entered information, processing of business logic, and data processing.

View - A view represents and displays information to the client - think an HTML/CSS/JS page. The view retrieves information from model objects. These model objects can be passed to the view from a Front Controller servlet through a request attribute, or by placement in the web application's session data.

The **benefits** of using the front controller design pattern:

- It provides centralized control for all requests, which helps for user tracking and security.
- It improves manageability, reusability, and role separation.

The Program flow of a Front Controller within a Web Application is as follows: The Front Controller uses the Dispatcher to delegate and identify which Helper can process the request and return the correct View.

Example

We will create a basic web application that allows a Customer to login to a basic `login.html` page.

The Web Application will have several packages to store our Models, Views, and Controllers.

1. Create a `Customer.java` class in a package called `com.revature.model`

```
package com.revature.model;


/* Main Customer POJO (bean) needed for example */
public class Customer {
        private int id;
        private String firstName;
        private String lastName;
        private String username;
        private String password;

        public Customer() {
                this.firstName = "";
                this.lastName = "";
                this.username = "";
                this.password = "";
        }
        public Customer(int id) {
                this();
                this.id = id;
        }

        public Customer(String username, String password) {
                this();
                this.username = username;
                this.password = password;
        }

        public Customer(int id, String firstName, String lastName, String username,
String password) {
                this.id = id;
                this.firstName = firstName;
                this.lastName = lastName;
                this.username = username;
                this.password = password;
        }

    // getters & setters and toString() method
}
```

2. Create a **Repository** package `com.revature.repository` which holds your repositories that follow the Data Access Object (DAO) design pattern. The DAO separates your business logic from your persistence layer (database).

```
package com.revature.repository;

import java.util.List;

import com.revature.model.Customer;

/* Contract interface that uses DAO design pattern rules that can be implemented
 * by many types of technologies like JDBC, Hibernate, MongoDB, etc.
 */
```

```
public interface CustomerRepository {
        public boolean insert(Customer customer);
        public boolean insertProcedure(Customer customer);
        public Customer select(Customer customer);
        public List<Customer> selectAll();
        public String getCustomerHash(Customer customer);
}
```

3. Create another class which will implement the `CustomerRepository` interface and establish connection to your database. Create your ConnectionUtil class in a separate package to follow best practices.

```
package com.revature.repository;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import org.apache.log4j.Logger;

import com.revature.model.Customer;
import com.revature.util.ConnectionUtil;

/* JDBC implementation for DAO contract for Customers data access */
public class CustomerRepositoryJdbc implements CustomerRepository {

        private static Logger logger = Logger.getLogger(CustomerRepositoryJdbc.class
);

        /*Singleton transformation of JDBC implementation object */
        private static CustomerRepository customerRepository;

        private CustomerRepositoryJdbc() {}

        public static CustomerRepository getInstance() {
                if(customerRepository == null) {
                        customerRepository = new CustomerRepositoryJdbc();
                }

                return customerRepository;
        }

        /* Regular insert statement for Customer */
        @Override
        public boolean insert(Customer customer) {
                try(Connection connection = ConnectionUtil.getConnection()) {
                        int statementIndex = 0;
                        String command = "INSERT INTO CUSTOMER VALUES(NULL,?,?,?,?)"
;

                        PreparedStatement statement = connection.prepareStatement(co
mmand);

                        //Set attributes to be inserted
                        statement.setString(++statementIndex, customer.getFirstName(
).toUpperCase());
                        statement.setString(++statementIndex, customer.getLastName()
.toUpperCase());
                        statement.setString(++statementIndex, customer.getUsername()
.toLowerCase());
                        statement.setString(++statementIndex, customer.getPassword()
);

                        if(statement.executeUpdate() > 0) {
                                return true;
                        }
                } catch (SQLException e) {
                        logger.warn("Exception creating a new customer", e);
                }
                return false;
        }

        /* Insert a customer using the stored procedure we created */
        @Override
        public boolean insertProcedure(Customer customer) {
                try(Connection connection = ConnectionUtil.getConnection()) {
                        int statementIndex = 0;

                        //Pay attention to this syntax
                        String command = "{CALL INSERT_CUSTOMER(?,?,?,?)}";

                        //Notice the CallableStatement
                        CallableStatement statement = connection.prepareCall(command
);

                        //Set attributes to be inserted
                        statement.setString(++statementIndex, customer.getFirstName(
).toUpperCase());
                        statement.setString(++statementIndex, customer.getLastName()
.toUpperCase());
                        statement.setString(++statementIndex, customer.getUsername()
.toLowerCase());
                        statement.setString(++statementIndex, customer.getPassword()
);

                        if(statement.executeUpdate() > 0) {
                                return true;
                        }
                } catch (SQLException e) {
                        logger.warn("Exception creating a new customer with stored p
rocedure", e);
                }
                return false;
        }

        /* Select Customer based on his username */
        @Override
        public Customer select(Customer customer) {
                try(Connection connection = ConnectionUtil.getConnection()) {
                        int statementIndex = 0;
                        String command = "SELECT * FROM CUSTOMER WHERE C_USERNAME = 
?";
                        PreparedStatement statement = connection.prepareStatement(co
mmand);
                        statement.setString(++statementIndex, customer.getUsername()
);
                        ResultSet result = statement.executeQuery();

                        while(result.next()) {
                                return new Customer(
                                                result.getInt("C_ID"),
                                                result.getString("C_FIRST_NAME"),
                                                result.getString("C_LAST_NAME"),
                                                result.getString("C_USERNAME"),
                                                result.getString("C_PASSWORD")
                                                );
                        }
                } catch (SQLException e) {
                        logger.warn("Exception selecting a customer", e);
                }
                return new Customer();
        }

        /* Select all customers */
        public List<Customer> selectAll() {
                try(Connection connection = ConnectionUtil.getConnection()) {
                        String command = "SELECT * FROM CUSTOMER";
                        PreparedStatement statement = connection.prepareStatement(co
mmand);
                        ResultSet result = statement.executeQuery();

                        List<Customer> customerList = new ArrayList<>();
                        while(result.next()) {
                                customerList.add(new Customer(
                                                result.getInt("C_ID"),
                                                result.getString("C_FIRST_NAME"),
                                                result.getString("C_LAST_NAME"),
                                                result.getString("C_USERNAME"),
                                                result.getString("C_PASSWORD")
                                                ));
                        }

                        return customerList;
                } catch (SQLException e) {
                        logger.warn("Exception selecting all customers", e);
                }
                return new ArrayList<>();
        }

        /* Get a customer hash consuming the user defined function we created */
        @Override
        public String getCustomerHash(Customer customer) {
                try(Connection connection = ConnectionUtil.getConnection()) {
                        int statementIndex = 0;
                        String command = "SELECT GET_CUSTOMER_HASH(?,?) AS HASH FROM
DUAL";
                        PreparedStatement statement = connection.prepareStatement(co
mmand);
                        statement.setString(++statementIndex, customer.getUsername()
);
                        statement.setString(++statementIndex, customer.getPassword()
);
                        ResultSet result = statement.executeQuery();

                        if(result.next()) {
                                return result.getString("HASH");
                        }
                } catch (SQLException e) {
                        logger.warn("Exception getting customer hash", e);
                }
                return new String();
        }
}
```

4. Your `ConnectionUtil.java` class will look like this:

```
package com.revature.util;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import org.apache.log4j.Logger;

/* Final utility class to obtain connections in a modular way */
public final class ConnectionUtil {

        private static Logger logger = Logger.getLogger(ConnectionUtil.class);

        /* Make Tomcat now which database driver to use */
        static {
                try {
                        Class.forName("oracle.jdbc.driver.OracleDriver");
                } catch (ClassNotFoundException e) {
                        logger.warn("Exception thrown adding oracle driver.", e);
                }
        }

        /* Get connection to JDBC */
        public static Connection getConnection() throws SQLException {
                String url = "jdbc:oracle:thin:@myrevaturerds.cgmoq4yzdcov.us-east-1
.rds.amazonaws.com:1521:ORCL";
                String username = "LOGIN_TEST_DB";
```

```
                  String password = "p4ssw0rd";


                  return DriverManager.getConnection(url, username, password);
          }
}
```

5. Create a **Service** layer between the controller and the model. Start with a `com.revature.service` package, and create an Interface called `CustomerService.java`.

This allows us to encapsulate all of the business logic that could be in the controller. This way, the only purpose of the controller is to forward and control the execution.

```
package com.revature.service;


import java.util.List;


import com.revature.model.Customer;


public interface CustomerService {


        public boolean registerCustomer(Customer customer);


        public boolean registerCustomerSecure(Customer customer);


        public List<Customer> listAllCustomers();


        public Customer authenticate(Customer customer);
}
```

6. Create an implementation of your `CustomerService.java` Interface in a class called `CustomerServiceAlpha`.

```
package com.revature.service;


import java.util.List;

import com.revature.model.Customer;
import com.revature.repository.CustomerRepositoryJdbc;

public class CustomerServiceAlpha implements CustomerService {

        /* Singleton */
        private static CustomerService customerService = new CustomerServiceAlpha();

        private CustomerServiceAlpha() { }

        public static CustomerService getInstance() {
                return customerService;
        }

        @Override
        public boolean registerCustomer(Customer customer) {
                return CustomerRepositoryJdbc.getInstance().insert(customer);
        }

        @Override
        public boolean registerCustomerSecure(Customer customer) {
                return CustomerRepositoryJdbc.getInstance().insertProcedure(customer
);
        }

        @Override
        public List<Customer> listAllCustomers() {
                return CustomerRepositoryJdbc.getInstance().selectAll();
        }

        @Override
        public Customer authenticate(Customer customer) {
                //Information on the database
                Customer loggedCustomer = CustomerRepositoryJdbc.getInstance().selec
t(customer);
```

```
                /*
                 * What we have stored in the database is the Username + Password ha
sh.
                 * We can't compare the blank password provided by the user, against
the hash.
                 * So we have to obtain the hash of the user input.
                 *
                 * If the hashes are the same, user is authenticated.
                 */
                if(loggedCustomer.getPassword().equals(CustomerRepositoryJdbc.getIns
tance().getCustomerHash(customer))) {
                        return loggedCustomer;
                }

                return null;
        }
}
```

7. Before we create a **Controller** Layer, let's create a package called `com.revature.ajax` which will hold the a `ClientMessage.java` that a controller could instantiate and return to the Client.

```
package com.revature.ajax;

public class ClientMessage {
        private String message;

        public ClientMessage() {}

        public ClientMessage(String message) {
                this.message = message;
        }

        public String getMessage() {
                return message;
        }
        public void setMessage(String message) {
                this.message = message;
        }
}
```

8. Next, create your **Controller** layer in a package called `com.revature.controller`. This will be an interface called `CustomerController.java` which we will implement later in another controller.

```
package com.revature.controller;

import javax.servlet.http.HttpServletRequest;

public interface CustomerController {

        /**
         * Registers the user.
         *
         * -> If the method is GET, return the registration view.
         * -> Else, return a message stating that registration was successful, or no
t.
         */
        public Object register(HttpServletRequest request);

        /**
         * Get all customers in the database.
         *
         * -> If it's GET with no parameters, then we return the view.
         * -> If it's GET with a parameter, then we return the list of users.
         */
        public Object getAllCustomers(HttpServletRequest request);
}
```

9. Create another controller called `CustomerControllerAlpha` which implements the `CustomerController` interface we just created.

```
package com.revature.controller;


import javax.servlet.http.HttpServletRequest;


import com.revature.ajax.ClientMessage;
import com.revature.model.Customer;
import com.revature.service.CustomerServiceAlpha;


public class CustomerControllerAlpha implements CustomerController {


        private static CustomerController customerController = new CustomerControlle
rAlpha();


        private CustomerControllerAlpha() {}


        public static CustomerController getInstance() {
                return customerController;
        }


        @Override
        public Object register(HttpServletRequest request) {
                if (request.getMethod().equals("GET")) {
                        return "register.html";
                }


                /* Logic for POST */
                Customer customer = new Customer(0, request.getParameter("firstName"
), request.getParameter("lastName"),
                                request.getParameter("username"), request.getParame
ter("password"));


                if (CustomerServiceAlpha.getInstance().registerCustomerSecure(custom
er)) {
                        return new ClientMessage("REGISTRATION SUCCESSFUL");
                } else {
                        return new ClientMessage("SOMETHING WENT WRONG");
                }
        }

        @Override
        public Object getAllCustomers(HttpServletRequest request) {
                Customer loggedCustomer = (Customer) request.getSession().getAttribu
te("loggedCustomer");

                /* If customer is not logged in */
                if(loggedCustomer == null) {
                        return "login.html";
                }

                /* Client is requesting the view. */
                if(request.getParameter("fetch") == null) {
                        return "all-customers.html";
                } else {
                /* Client is requesting the list of customers */
                        return CustomerServiceAlpha.getInstance().listAllCustomers()
;
                }
        }
}
```

10. Similarly, we will need to create a `LoginController.java` interface and implementation to handle login requests.

```
package com.revature.controller;

import javax.servlet.http.HttpServletRequest;

public interface LoginController {

        /**
         * If the method is GET, it will return the login view.
         *
         * If the method is POST.
         * -> If service layer returns null, we return a message
         * stating that authentication failed.
         *
         * -> Else, it will return the Customer information
         * (and store it in the session).
         */
        public Object login(HttpServletRequest request);

        /**
         * Invalidates the session and returns the login view.
         */
        public String logout(HttpServletRequest request);
}
```

11. Create the controller implementation in a class called `LoginControllerAlpha`.

```
package com.revature.controller;

import javax.servlet.http.HttpServletRequest;

import com.revature.ajax.ClientMessage;
import com.revature.model.Customer;
import com.revature.service.CustomerServiceAlpha;

public class LoginControllerAlpha implements LoginController {

        private static LoginController loginController = new LoginControllerAlpha();

        private LoginControllerAlpha() {}

        public static LoginController getInstance() {
                return loginController;
        }

        @Override
        public Object login(HttpServletRequest request) {
                if(request.getMethod().equals("GET")) {
                        return "login.html";
                }

                Customer loggedCustomer = CustomerServiceAlpha.getInstance().authent
icate(
                                        new Customer(request.getParameter("username
"),
                                                                request.getParame
ter("password"))
                                );

                /* If authentication failed */
                if(loggedCustomer == null) {
                        return new ClientMessage("AUTHENTICATION FAILED");
                }

                /* Store the customer information on the session */
                request.getSession().setAttribute("loggedCustomer", loggedCustomer);
                return loggedCustomer;
        }

        @Override
        public String logout(HttpServletRequest request) {
                //throw new RuntimeException("Something went wrong");

                /*
                 * If session.invalidate() doesn't work for you
                 */

                request.getSession().invalidate();
                return "login.html";
        }
```

```
}
```

12. Create your **Dispatchers** to delegate to particular controllers. This acts as a **Front Controller**. These are **Servlets** that redirect the user with the help of the logic incorporated in your controllers.

First create a `RequestHelper.java` class which will delegate to your DispatcherServlet (which we will create next).

```
package com.revature.request;


import javax.servlet.http.HttpServletRequest;


import com.revature.controller.CustomerControllerAlpha;
import com.revature.controller.LoginControllerAlpha;


public class RequestHelper {


        private RequestHelper() {}


        public static Object process(HttpServletRequest request) {
                switch(request.getRequestURI()) {
                case "/FrontController/login.do":
                        return LoginControllerAlpha.getInstance().login(request);
                case "/FrontController/logout.do":
                        return LoginControllerAlpha.getInstance().logout(request);
                case "/FrontController/register.do":
                        return CustomerControllerAlpha.getInstance().register(reques
t);
                case "/FrontController/getAll.do":
                        return CustomerControllerAlpha.getInstance().getAllCustomers
(request);
                default:
                        return "not-implemented.html";
                }
        }
}
```

13. Then create your `DispatcherServlet`.

```
package com.revature.request;


import java.io.IOException;


import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


import com.fasterxml.jackson.databind.ObjectMapper;


public class DispatcherServlet extends HttpServlet {


        private static final long serialVersionUID = 5244611926643604805L;


        protected void doGet(HttpServletRequest request, HttpServletResponse respons
e) throws ServletException, IOException {
                Object data = RequestHelper.process(request);


                /* If what the controllers return is a String, we forward to an HTML
file. */
                if(data instanceof String) {
                        String URI = (String) data;
                        request.getRequestDispatcher(URI).forward(request, response)
;
                }
                /* Else, we marshall the given POJO */
                else {
                        response.getWriter().write(new ObjectMapper().writeValueAsSt
ring(data));
                }
        }


        protected void doPost(HttpServletRequest request, HttpServletResponse respon
se) throws ServletException, IOException {
```

```
                doGet(request, response);
        }
}
```

14. In your **web.xml**, you can also map to error pages. It will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
  <display-name>FrontController</display-name>

  <welcome-file-list>
        <welcome-file>login.html</welcome-file>
  </welcome-file-list>

  <servlet>
        <servlet-name>DispatcherServlet</servlet-name>
        <servlet-class>com.revature.request.DispatcherServlet</servlet-class>
  </servlet>

  <servlet-mapping>
        <servlet-name>DispatcherServlet</servlet-name>
        <!-- * it's a Wildcard (which means "anything") -->
        <!-- .something it's called Extended Mapping -->
        <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- ERROR PAGES -->

  <!-- For 500 (Exceptions Thrown) -->
  <error-page>
        <exception-type>java.lang.Throwable</exception-type>
        <location>/oops.html</location>
  </error-page>
  <!-- For a specific status code -->
  <error-page>
        <error-code>404</error-code>
        <location>/404.html</location>
  </error-page>

  <!-- Session Objects will expire in 30 minutes -->
  <session-config>
        <session-timeout>30</session-timeout>
  </session-config>
</web-app>
```

15. You should have a `login.html`, `register.html`, `all-customers.html`, `404.html`, and an error page titled `oops.html` for any exceptions (as mapped in our **web.xml** page).

References

- Further Reading on FrontController Design Pattern

MVC Design pattern

The **M**odel **V**iew **C**ontroller Design pattern is an architectural pattern, used to design and create user interfaces and the structure of an application.

This pattern divides the application into three parts that are dependent and connected.

Model - Model used to represent the business layer of the application. It's not involved in the UI or presentation of the application. The model is the *data layer* which defines the business logic of the system and also represents the state of the application. The model objects retrieve and store the state of the model in a database.

View - The view presents the model's data to the user. The view is a *presentation layer* that represents the user interface. It displays the data fetched from the model layer by the controller and presents the data to the user whenever requests.

Controller - A controller is an intermediary between the model & a view. It receives the user requests from the view layer and processes them. The requests are then sent to model for data processing. Once they are processed, the data is again sent back to the controller and then displayed on the view.

Advantages - Multiple developers can simultaneously work on the model, controller and views. The MVC pattern enables a logical grouping of related actions on a controller together. The views for a specific model are also grouped. Models can have multiple views.

References

- MVC Example
- Oracle MVC Documentation