

Java SQL Adapter

Overview

This tutorial is a continuation of Java Adapter ([../../server-side-development/java-adapter/](#)) and assumes previous knowledge of the concepts described there.

Java adapters give developers control over connectivity to a back end. It is therefore the responsibility of the developer to ensure best practices regarding performance and other implementation details.

This tutorial covers an example of a Java adapter that connects to a MySQL back end to make CRUD (Create, Read, Update, Delete) operations on a users table, using REST concepts.

Prerequisite: This tutorial assumes knowledge of SQL.

Topics:

- Setting up the data source
- UserAdapterApplication
- UserAdapterResource
- Results
- Sample

Setting up the data source

The MobileFirst Server needs to be configured to connect to the MySQL server. Those configurations can be stored in the `server.xml` file.

To connect to a database, Java code needs a JDBC driver. For MySQL, the latest driver can be found on the Download Connector/J (<http://dev.mysql.com/downloads/connector/j/>) MySQL page, at <http://dev.mysql.com/downloads/connector/j/>.

This example uses the embedded application server of MobileFirst Studio.

```
1 <library id="MySQLLib">
2   <fileset dir="${shared.resource.dir}" includes="mysql-*.jar" />
3 </library>
4 <dataSource jndiName="jdbc/mobilefirst_training">
5   <jdbcDriver libraryRef="MySQLLib" />
6   <properties databaseName="mobilefirst_training"
7     password=""
8     portNumber="3306"
9     serverName="localhost"
10    user="root" />
11 </dataSource>
```

- The `library` tag specifies where to find the MySQL `.jar` file. In most cases, `${shared.resource.dir}` is **shared/resources** under the Liberty server root folder.

In MobileFirst Studio for Eclipse, select **Project Explorer > MobileFirst Development Server > shared > resources**.

- The `dataSource` tag specifies how to connect to the database. Write down the `jndiName` that you choose, because you will need it later.

UserAdapterApplication

`UserAdapterApplication` extends `MFPJAXRSApplication` and is a good place to trigger any initialization required by the adapter application.

```
1 | @Override
2 | protected void init() throws Exception {
3 |     UserAdapterResource.init();
4 |     logger.info("Adapter initialized!");
5 | }
```

UserAdapterResource

```
1 | @Path("/")
2 | public class UserAdapterResource {
3 |
4 | }
```

`UserAdapterResource` is where requests to the adapter are handled.

`@Path("/")` means that the resources will be available at the URL `http(s)://host:port/ProjectName/adapters/AdapterName/`.

Using DataSource

```
1 | static DataSource ds = null;
2 | static Context ctx = null;
3 | public static void init() throws NamingException {
4 |     ctx = new InitialContext();
5 |     ds = (DataSource)ctx.lookup("jdbc/mobilefirst_training");
6 | }
```

The `DataSource` is set as `static` so that it can be shared across all requests to the adapter. It is initialized in the `init()` method, which is called by the `init()` method of `UserAdapterApplication`, as described above.

The previously defined `jndiName` parameter is used to find the database configuration.

Create User

```

1  @POST
2  public Response createUser(@FormParam("userId") String userId, @FormParam("firstName") String first
3      Connection con = ds.getConnection();
4      PreparedStatement insertUser = con.prepareStatement("INSERT INTO users (userId, firstName, lastNa
5      try{
6          insertUser.setString(1, userId);
7          insertUser.setString(2, firstName);
8          insertUser.setString(3, lastName);
9          insertUser.setString(4, password);
10         insertUser.executeUpdate();
11         //Return a 200 OK
12         return Response.ok().build();
13     }
14     catch (SQLIntegrityConstraintViolationException violation) {
15         //Trying to create a user that already exists
16         return Response.status(Status.CONFLICT).entity(violation.getMessage()).build();
17     }
18     finally{
19         //Close resources in all cases
20         insertUser.close();
21         con.close();
22     }
23 }

```

Because this method does not have any `@Path`, it is accessible as the root URL of the resource. Because it uses `@POST`, it is accessible via HTTP POST only.

The method has a series of `@FormParam` arguments, which means that those can be sent in the HTTP body as `x-www-form-urlencoded` parameters.

It is also possible to pass the parameters in the HTTP body as JSON objects, by using `@Consumes(MediaType.APPLICATION_JSON)`, in which case the method needs a `JSONObject` argument, or a simple Java object with properties that match the JSON property names.

The `Connection con = ds.getConnection();` method gets the connection from the data source that was defined earlier.

The SQL queries are built by the `PreparedStatement` method.

If the insertion was successful, the `return Response.ok().build()` method is used to send a 200 OK back to the client. If there was an error, a different `Response` object can be built with a specific HTTP status code. In this example, a 409 Conflict error code is sent. It is advised to also check whether all the parameters are sent (not shown here) or any other data validation.

Important: Make sure to close resources, such as prepared statements and connections.

Get User

```

1  @GET
2  @Produces("application/json")
3  @Path("/{userId}")
4  public Response getUser(@PathParam("userId") String userId) throws SQLException{
5      Connection con = ds.getConnection();
6      PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId = ?");</pre>


```

7 try{
8 JSONObject result = new JSONObject();
9 getUser.setString(1, userId);
10 ResultSet data = getUser.executeQuery();
11 if(data.first()){
12 result.put("userId", data.getString("userId"));
13 result.put("firstName", data.getString("firstName"));
14 result.put("lastName", data.getString("lastName"));
15 result.put("password", data.getString("password"));
16 return Response.ok(result).build();
17 } else{
18 return Response.status(Status.NOT_FOUND).entity("User not found...").build();
19 }
20 }
21 finally{
22 //Close resources in all cases
23 getUser.close();
24 con.close();
25 }
26 }

```


```

This method uses `@GET` with a `@Path("/{userId}")`, which means that it is available via HTTP `GET /adapters/UserAdapter/{userId}`, and the `{userId}` is retrieved by the `@PathParam("userId")` argument of the method.

If the user is not found, the 404 `NOT FOUND` error code is returned.

If the user is found, a response is built from the generated JSON object.

Prepending the method with `@Produces("application/json")` makes sure that the Content-Type of the output is correct.

Get all users

This method is similar to `getUser`, except for the loop over the `ResultSet`.

```

1  @GET
2  @Produces("application/json")
3  public Response getAllUsers() throws SQLException{
4      JSONArray results = new JSONArray();
5      Connection con = ds.getConnection();
6      PreparedStatement getAllUsers = con.prepareStatement("SELECT * FROM users");
7      ResultSet data = getAllUsers.executeQuery();</p>
8      while(data.next()){
9          JSONObject item = new JSONObject();
10         item.put("userId", data.getString("userId"));
11         item.put("firstName", data.getString("firstName"));
12         item.put("lastName", data.getString("lastName"));
13         item.put("password", data.getString("password"));</p>
14         results.add(item);
15     }
16     getAllUsers.close();
17     con.close();
18     return Response.ok(results).build();
19 }

```

Update user

```

1  <br />
2  @PUT
3  @Path("/{userId}")
4  public Response updateUser(@PathParam("userId") String userId, @FormParam("firstName") String first
5      Connection con = ds.getConnection();
6      PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId = ?");
7      try{
8          getUser.setString(1, userId);
9          ResultSet data = getUser.executeQuery();
10         if(data.first()){
11             PreparedStatement updateUser = con.prepareStatement("UPDATE users SET firstName = ?, lastNan
12             updateUser.setString(1, firstName);
13             updateUser.setString(2, lastName);
14             updateUser.setString(3, password);
15             updateUser.setString(4, userId);
16             updateUser.executeUpdate();
17             updateUser.close();
18             return Response.ok().build();
19         } else{
20             return Response.status(Status.NOT_FOUND).entity("User not found...").build();
21         }
22     }
23     finally{
24         //Close resources in all cases
25         getUser.close();
26         con.close();
27     }
28 }

```

When updating an existing resource, it is standard practice to use `@PUT` (for HTTP PUT) and to use the resource ID in the `@Path`.

Delete user

```
1  @DELETE
2  @Path("/{userId}")
3  public Response deleteUser(@PathParam("userId") String userId) throws SQLException{
4      Connection con = ds.getConnection();
5      PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId = ?");
6      try{
7          getUser.setString(1, userId);
8          ResultSet data = getUser.executeQuery();
9          if(data.first()){
10             PreparedStatement deleteUser = con.prepareStatement("DELETE FROM users WHERE userId = ?");
11             deleteUser.setString(1, userId);
12             deleteUser.executeUpdate();
13             deleteUser.close();
14             return Response.ok().build();
15         } else{
16             return Response.status(Status.NOT_FOUND).entity("User not found...").build();
17         }
18     }
19     finally{
20         //Close resources in all cases
21         getUser.close();
22         con.close();
23     }
24 }
```

`@DELETE` (for HTTP DELETE) is used together with the resource ID in the `@Path`, to delete a user.

Results

Use the testing techniques described in Java Adapter ([../../server-side-development/java-adapter/](#)) to test your work.

Sample

Download the MobileFirst project (<https://github.com/MobileFirst-Platform-Developer-Center/JavaAdapters>) which includes the **UserAdapter** described here.

The project also includes a sample MySQL script in the **server** folder, which needs to be imported into your database to test the project.

The project does not include the MySQL connector driver, and does not include the **server.xml** configuration described above. Those steps need to be completed in order to use the sample.