

Java SQL Adapter

Overview

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

Java adapters give you 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 shows 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 by using REST concepts.

The tutorial covers the following topics:

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

Setting up the data source

MobileFirst Server needs to be configured to connect to the MySQL server. You can store the configuration settings in the `server.xml` file.

For Java to be able to connect to a database, a JDBC driver is necessary. For MySQL, you can find the latest driver from this [Connector/J](http://dev.mysql.com/downloads/connector/j/) (<http://dev.mysql.com/downloads/connector/j/>) MySQL page.

This example uses the underlying application server of MobileFirst Studio.

```
<br />
<library id="MySQLLib"><br />
  <fileset dir="${shared.resource.dir}" includes="mysql-*.jar" /><br />
>
</library></p>
<p><dataSource jndiName="jdbc/mobilefirst_training"><br />
  <jdbcDriver libraryRef="MySQLLib" /><br />
  <properties databaseName="mobilefirst_training"><br />
    password=""<br />
    portNumber="3306"<br />
    serverName="localhost"<br />
    user="root" /><br />
</dataSource><br />
```

- The `library` tag specifies where the MySQL `.jar` file can be found. In most cases, `${shared.resource.dir}` is the `shared/resources` folder under the Liberty server root folder. In MobileFirst Studio for Eclipse, the path is `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.

```
<br />
@Override<br />
protected void init() throws Exception {<br />
    UserAdapterResource.init();<br />
    logger.info("Adapter initialized!");<br />
}<br />
```

UserAdapterResource

```
<br />
@Path("/")<br />
public class UserAdapterResource {<br />
}<br />
```

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

`@Path("/")` means that the resources are available at this URL:

`http(s)://host:port/ProjectName/adapters/AdapterName/`

Using DataSource

```
<br />
static DataSource ds = null;<br />
static Context ctx = null;</p>
<p> public static void init() throws NamingException {<br />
    ctx = new InitialContext();<br />
    ds = (DataSource)ctx.lookup("jdbc/mobilefirst_training");<br />
>
}<br />
```

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 gets called by the `init()` method of `UserAdapterApplication`, as described above.

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

Create User

```

<br />
@POST<br />
public Response createUser(@FormParam("userId") String userId,<br />
    @FormParam("firstName") String firstName,<br />
    @FormParam("lastName") String lastName,<br />
    @FormParam("password") String password)<br />
    throws SQLException{</p>
<p> Connection con = ds.getConnection();<br />
    PreparedStatement insertUser = con.prepareStatement("INSERT INTO users (userId, firstName, lastNa
me, password) VALUES (?, ?, ?, ?)");</p>
<p> try{<br />
    insertUser.setString(1, userId);<br />
    insertUser.setString(2, firstName);<br />
    insertUser.setString(3, lastName);<br />
    insertUser.setString(4, password);<br />
    insertUser.executeUpdate();<br />
    //Return a 200 OK<br />
    return Response.ok().build();<br />
}<br />
catch (SQLIntegrityConstraintViolationException violation) {<br />
    //Trying to create a user that already exists<br />
    return Response.status(Status.CONFLICT).entity(violation.getMessage()).build();<br />
}<br />
finally{<br />
    //Close resources in all cases<br />
    insertUser.close();<br />
    con.close();<br />
}<br />
}<br />

```

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 arguments 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 this 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();` statement gets the connection from the data source that was defined in `Using DataSource`.

The SQL queries are built by using `PreparedStatement`.

If the insertion was successful, `return Response.ok().build()` 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, 409 Conflict is sent. It is better 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

```

<br />
@GET<br />
@Produces("application/json")<br />
@Path("/{userId}")<br />
public Response getUser(@PathParam("userId") String userId) throws SQLException{<br />
    Connection con = ds.getConnection();<br />
    PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId = ?");</p>
    <p> try{<br />
        JSONObject result = new JSONObject();</p>
        <p>    getUser.setString(1, userId);<br />
        ResultSet data = getUser.executeQuery();</p>
        <p>    if(data.first()){<br />
            result.put("userId", data.getString("userId"));<br />
            result.put("firstName", data.getString("firstName"));<br />
            result.put("lastName", data.getString("lastName"));<br />
            result.put("password", data.getString("password"));<br />
            return Response.ok(result).build();</p>
        <p>    } else{<br />
            return Response.status(Status.NOT_FOUND).entity("User not found...").build();<br />
        }</p>
        <p>    }<br />
        finally{<br />
            //Close resources in all cases<br />
            getUser.close();<br />
            con.close();<br />
        }</p>
    <p>    }<br />
}

```

This method uses `@GET` with a `@Path("/{userId}")`, which means 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, `404 NOT FOUND` is returned.

If the user is found, a response is built from the generated `JSONObject`.

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 that it loops over the `ResultSet`.

```

<br />
@GET<br />
@Produces("application/json")<br />
public Response getAllUsers() throws SQLException{<br />
    JSONArray results = new JSONArray();<br />
    Connection con = ds.getConnection();<br />
    PreparedStatement getAllUsers = con.prepareStatement("SELECT * FROM users");<br /
>
    ResultSet data = getAllUsers.executeQuery();<p>
<p> while(data.next()){<br />
    JSONObject item = new JSONObject();<br />
    item.put("userId", data.getString("userId"));<br />
    item.put("firstName", data.getString("firstName"));<br />
    item.put("lastName", data.getString("lastName"));<br />
    item.put("password", data.getString("password"));</p>
<p> results.add(item);<br />
    }</p>
<p> getAllUsers.close();<br />
    con.close();</p>
<p> return Response.ok(results).build();<br />
}<br />

```

Update user

```

<br />
@PUT<br />
@Path("/{userId}")<br />
public Response updateUser(@PathParam("userId") String userId,<br />
    @FormParam("firstName") String firstName,<br />
    @FormParam("lastName") String lastName,<br />
    @FormParam("password") String password)<br />
    throws SQLException{<br />
    Connection con = ds.getConnection();<br />
    PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId = ?");</p>
    >
    <p> try{<br />
        getUser.setString(1, userId);<br />
        ResultSet data = getUser.executeQuery();</p>
    <p> if(data.first()){<br />
        PreparedStatement updateUser = con.prepareStatement("UPDATE users SET firstName = ?, lastNam
e = ?, password = ? WHERE userId = ?");</p>
    <p> updateUser.setString(1, firstName);<br />
        updateUser.setString(2, lastName);<br />
        updateUser.setString(3, password);<br />
        updateUser.setString(4, userId);</p>
    <p> updateUser.executeUpdate();<br />
        updateUser.close();<br />
        return Response.ok().build();</p>
    <p> } else{<br />
        return Response.status(Status.NOT_FOUND).entity("User not found...").build();<br />
    }<br />
    }<br />
    finally{<br />
        //Close resources in all cases<br />
        getUser.close();<br />
        con.close();<br />
    }</p>
    <p> }<br />

```

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

Delete user

```

<br />
@DELETE<br />
@Path("/{userId}")<br />
public Response deleteUser(@PathParam("userId") String userId) throws SQLException{<br />
    Connection con = ds.getConnection();<br />
    PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId = ?");<p
>
<p> try{<br />
    getUser.setString(1, userId);<br />
    ResultSet data = getUser.executeQuery();</p>
<p> if(data.first()){<br />
    PreparedStatement deleteUser = con.prepareStatement("DELETE FROM users WHERE userId = ?");
<br />
    deleteUser.setString(1, userId);<br />
    deleteUser.executeUpdate();<br />
    deleteUser.close();<br />
    return Response.ok().build();</p>
<p> } else{<br />
    return Response.status(Status.NOT_FOUND).entity("User not found...").build();<br />
    }<br />
    }<br />
finally{<br />
    //Close resources in all cases<br />
    getUser.close();<br />
    con.close();<br />
    }</p>
<p> }<br />

```

@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 the Java Adapters (../../server-side-development/java-adapter/) tutorial to test your work.

Sample

Download the Studio project

(<http://public.dhe.ibm.com/software/products/en/MobileFirstPlatform/docs/v700/JavaAdaptersProject.zip>) which includes the **UserAdapter** described here.

The project also includes a sample MySQL script in the server folder. Import it into your database to test the project.

The project does not include the MySQL connector driver, and does not include the server.xml configuration file described in Setting up the data source. To use the sample, complete those steps first.