

Java SQL Adapter

Relevant to:



Native iOS



Native Android



Native Windows Phone 8



Native Windows 8 Universal



Hybrid

Overview

This tutorial is a continuation of [Java Adapter](#) and assumes previous knowledge of the concepts described there.

Java adapters provide free reign over connectivity to a backend. 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 backend to make CRUD (Create, Read, Update, Delete) operations on a `users` table, using REST concepts.

This tutorial assumes knowledge of SQL.

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.

For Java to be able to connect to a database, a JDBC driver needs to be used. For MySQL the latest driver can be found here: <http://dev.mysql.com/downloads/connector/j/>.

For this example, the underlying application server of MobileFirst Studio is used.

```
server.xml
<library id="MySQLLib">
  <fileset dir="${shared.resource.dir}" includes="mysql-*.jar" />
</library>

<dataSource jndiName="jdbc/mobilefirst_training">
  <jdbcDriver libraryRef="MySQLLib" />
  <properties databaseName="mobilefirst_training"
    password=""
    portNumber="3306"
    serverName="localhost"
    user="root" />
</dataSource>
```

- The `library` tag specifies where the MySQL .jar file can be found. In most cases `${shared.resource.dir}` is **shared/resources** under the Liberty server root folder. In MobileFirst Studio for Eclipse, this can be found in **Project Explorer > MobileFirst Development Server > shared > resources**.
- The `dataSource` tag specifies how to connect to the database. Write down the `jndiName` that you choose as it will be needed later.

UserAdapterApplication

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

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

UserAdapterResource

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

UserAdapterResource is where requests to the adapter are being handled.

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

Using DataSource

```
UserAdapterResource
static DataSource ds = null;
static Context ctx = null;

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

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

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

Create User

```
@POST
public Response createUser(@FormParam("userId") String userId,
                           @FormParam("firstName") String firstName,
                           @FormParam("lastName") String lastName,
                           @FormParam("password") String password)
                           throws SQLException{

    Connection con = ds.getConnection();
    PreparedStatement insertUser = con.prepareStatement("INSERT INTO users (userId,
firstName, lastName, password) VALUES (?, ?, ?, ?)");

    try{
        insertUser.setString(1, userId);
        insertUser.setString(2, firstName);
        insertUser.setString(3, lastName);
        insertUser.setString(4, password);
        insertUser.executeUpdate();
        //Return a 200 OK
        return Response.ok().build();
    }
    catch (SQLIntegrityConstraintViolationException violation) {
        //Trying to create a user that already exists
        return Response.status(Status.CONFLICT).entity(violation.getMessage()).build();
    }
    finally{
        //Close resources in all cases
        insertUser.close();
        con.close();
    }
}
```

Because this method does not have any @Path, it will be accessible as the root URL of the resource. Since it uses @POST, it will be 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 a JSON object, using `@Consumes(MediaType.APPLICATION_JSON)`, in which case the method will need a `JSONObject` argument, or a simple Java object with properties matching the JSON property names.

`Connection con = ds.getConnection();` will get the connection from the `DataSource` defined earlier.

The SQL queries are built 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 being sent. It is advised to also check if all the parameters are being sent (not shown here) or any other data validation.

It is important to make sure to close resources, such as prepared statements and connections.

Get User

```
@GET
@Produces("application/json")
@Path("/{userId}")
public Response getUser(@PathParam("userId") String userId) throws SQLException{
    Connection con = ds.getConnection();
    PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId
= ?");

    try{
        JSONObject result = new JSONObject();

        getUser.setString(1, userId);
        ResultSet data = getUser.executeQuery();

        if(data.first()){
            result.put("userId", data.getString("userId"));
            result.put("firstName", data.getString("firstName"));
            result.put("lastName", data.getString("lastName"));
            result.put("password", data.getString("password"));
            return Response.ok(result).build();
        } else{
            return Response.status(Status.NOT_FOUND).entity("User not
found...").build();
        }
    }
    finally{
        //Close resources in all cases
        getUser.close();
        con.close();
    }
}
```

This method uses `@GET` with a `@Path("/{userId}")`, which means it will be 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 we loop over the `ResultSet`.

```
@GET
@Produces("application/json")
public Response getAllUsers() throws SQLException{
    JSONArray results = new JSONArray();
    Connection con = ds.getConnection();
    PreparedStatement getAllUsers = con.prepareStatement("SELECT * FROM users");
    ResultSet data = getAllUsers.executeQuery();

    while(data.next()){
        JSONObject item = new JSONObject();
        item.put("userId", data.getString("userId"));
        item.put("firstName", data.getString("firstName"));
        item.put("lastName", data.getString("lastName"));
        item.put("password", data.getString("password"));

        results.add(item);
    }

    getAllUsers.close();
    con.close();

    return Response.ok(results).build();
}
```

Update user

```
@PUT
@Path("/{userId}")
public Response updateUser(@PathParam("userId") String userId,
                           @FormParam("firstName") String firstName,
                           @FormParam("lastName") String lastName,
                           @FormParam("password") String password)
                           throws SQLException{
    Connection con = ds.getConnection();
    PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId
= ?");

    try{
        getUser.setString(1, userId);
        ResultSet data = getUser.executeQuery();

        if(data.first()){
            PreparedStatement updateUser = con.prepareStatement("UPDATE users SET
firstName = ?, lastName = ?, password = ? WHERE userId = ?");

            updateUser.setString(1, firstName);
            updateUser.setString(2, lastName);
            updateUser.setString(3, password);
            updateUser.setString(4, userId);

            updateUser.executeUpdate();
            updateUser.close();
            return Response.ok().build();

        } else{
            return Response.status(Status.NOT_FOUND).entity("User not
found...").build();
        }
    } finally{
        //Close resources in all cases
        getUser.close();
        con.close();
    }
}
```

```
}
```

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

Delete user

```
@DELETE
@Path("/{userId}")
public Response deleteUser(@PathParam("userId") String userId) throws SQLException{
    Connection con = ds.getConnection();
    PreparedStatement getUser = con.prepareStatement("SELECT * FROM users WHERE userId
= ?");

    try{
        getUser.setString(1, userId);
        ResultSet data = getUser.executeQuery();

        if(data.first()){
            PreparedStatement deleteUser = con.prepareStatement("DELETE FROM users
WHERE userId = ?");
            deleteUser.setString(1, userId);
            deleteUser.executeUpdate();
            deleteUser.close();
            return Response.ok().build();
        } else{
            return Response.status(Status.NOT_FOUND).entity("User not
found...").build();
        }
    }
    finally{
        //Close resources in all cases
        getUser.close();
        con.close();
    }
}
```

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

Results

Use the testing techniques described in [Java Adapter](#) to test your work.

Sample

Download the [MobileFirst project](#) 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.