Java SQL Adapter

Overview

Java adapters give developers control over connectivity to a back end system. 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.

Prerequisites:

- Make sure to read the Java Adapters (../) tutorial first.
- This tutorial assumes knowledge of SQL.

Jump to:

- Setting up the data source
- Implemeting SQL in the adapter Resource class
- Results
- Sample

Setting up the data source

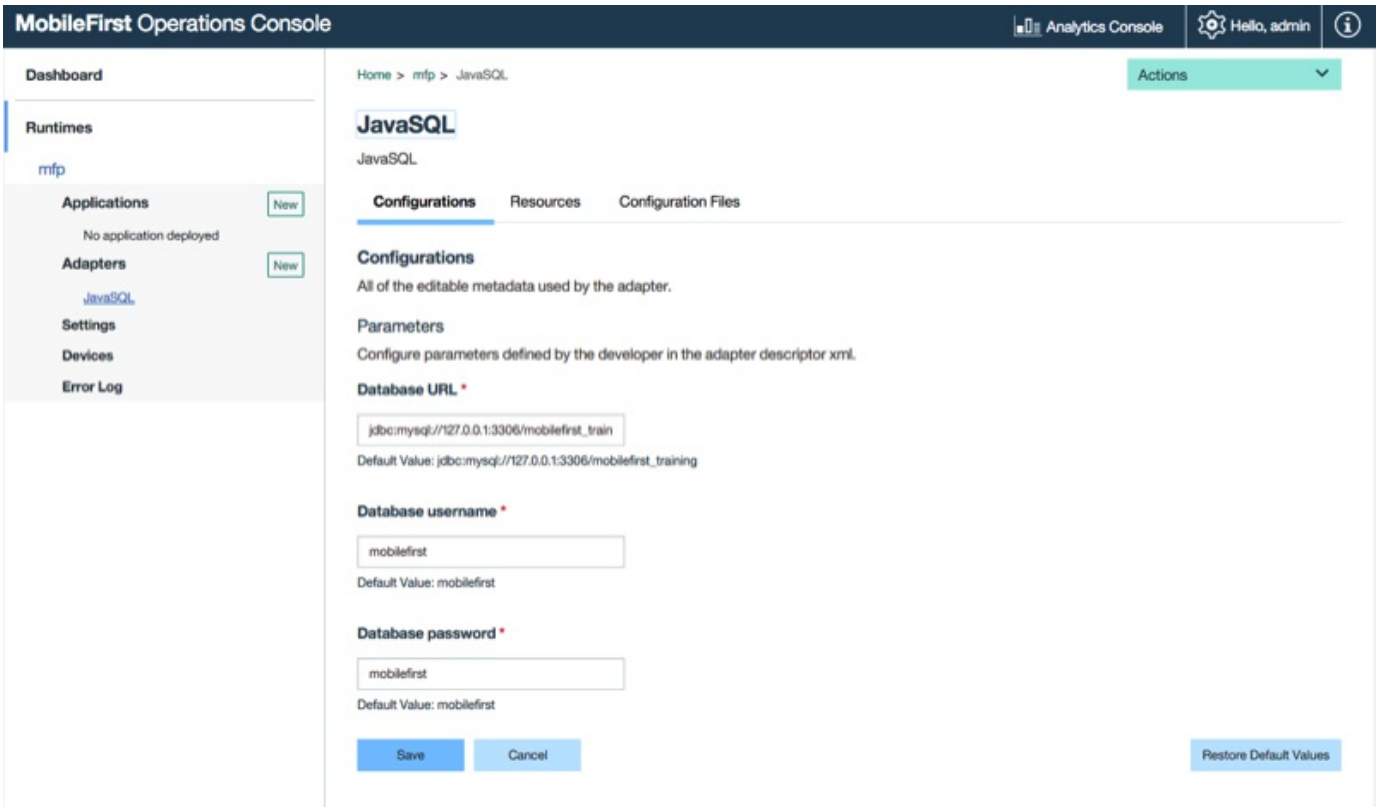
In order to configure the MobileFirst Server to be able to connect to the MySQL server, the adapter's XML file needs to be configured with **configuration properties**. These properties can later be edited through the MobileFirst Operations Console.

Edit the adater.xml file and add the following properties:

```
<mfp:adapter name="JavaSQL"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mfp="http://www.ibm.com/mfp/integration"
    xmlns:http="http://www.ibm.com/mfp/integration/http">

    <displayName>JavaSQL</displayName>
    <description>JavaSQL</description>

    <JAXRSApplicationClass>com.sample.JavaSQLApplication</JAXRSApplicationClass>

    <property name="DB_url" displayName="Database URL" defaultValue="jdbc:mysql://127.0.0.1:3306/mo
bilefirst_training"  />
    <property name="DB_username" displayName="Database username" defaultValue="mobilefirst"  />
    <property name="DB_password" displayName="Database password" defaultValue="mobilefirst"  />
</mfp:adapter>
```

> **Note:** The configuration properties elements must always be located *below* the `JAXRSApplicationClass` element.
> Here we define the connection settings and give them a default value, so they could be used later in the AdapterResource class.

You can then view and configure these properties in the MobileFirst Operations Console:



Implemeting SQL in the adapter Resource class

The adapter Resource class is where requests to the server are handled.

In the supplied sample adapter, the class name is `JavaSQLResource`.

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

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

Using DataSource

Define static variables to hold the database connection properties so they can be shared across all requests to the server:

```
private static BasicDataSource ds = null;
private static String DB_url = null;
private static String DB_username = null;
private static String DB_password  = null;
```

Since we are using configuration properties that can be configured during runtime in the console - we need to check their values each time we intend to connect to the database:

```
private boolean updatedProperties() {
    // Check if the properties were changed during runtime (in the console)
    String last_url = DB_url;
    String last_username = DB_username;
    String last_password  = DB_password;

    DB_url = configurationAPI.getPropertyValue("DB_url");
    DB_username = configurationAPI.getPropertyValue("DB_username");
    DB_password = configurationAPI.getPropertyValue("DB_password");

    return !last_url.equals(DB_url) ||
           !last_username.equals(DB_username) ||
           !last_password.equals(DB_password);
}
```

This method will be called each time we intend to connect to the database, and if the properties have been changed - we set the `DataSource` configuration properties again accordingly:

```
public Connection getSQLConnection(){
  // Create a connection object to the database
  Connection conn = null;
  if(updatedProperties() || ds == null){
    ds= new BasicDataSource();
    ds.setDriverClassName("com.mysql.jdbc.Driver");
    ds.setUrl(DB_url);
    ds.setUsername(DB_username);
    ds.setPassword(DB_password);
  }
  try {
    conn = ds.getConnection();
  } catch (SQLException e) {
    e.printStackTrace();
  }
  return conn;
}
```

Create User

Used to create a new user record in the database.

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

    Connection con = getSQLConnection();
    PreparedStatement insertUser = con.prepareStatement("INSERT INTO users (userId, firstName, lastNa
me, 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 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 = getSQLConnection();` 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

Retrieve a user from the database.

```
@GET
@Produces("application/json")
@Path("/{userId}")
public Response getUser(@PathParam("userId") String userId) throws SQLException{
    Connection con = getSQLConnection();
    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 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`.

```
@GET
@Produces("application/json")
public Response getAllUsers() throws SQLException{
    JSONArray results = new JSONArray();
    Connection con = getSQLConnection();
    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

Update a user record in the database.

```
@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 = getSQLConnection();
   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 = ?, lastN
ame = ?, 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();
   }

}
```

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

Delete a user record from the database.

```
@DELETE
@Path("/{userId}")
public Response deleteUser(@PathParam("userId") String userId) throws SQLException{
    Connection con = getSQLConnection();
    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 ID in the `@Path`, to delete a user.

Sample adapter

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/Adapters/tree/release80) the Adapters Maven project.

The Adapters Maven project includes the **JavaSQL** adapter described above.
Also included is an SQL script in the **Utils** folder, which needs to be imported into your database to test the project.

Sample usage

- Use either Maven or MobileFirst Developer CLI to build and deploy the adapter (../../creating-adapters/).
- To test or debug an adapter, see the testing and debugging adapters (../../testing-and-debugging-adapters) tutorial.