

# Java SQL Adapter

fork and edit tutorial (<https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/8.0/adapters/java-adapters/java-sql-adapter/index.md>) | report issue (<https://github.ibm.com/MFPSamples/DevCenter/issues/new>)

## 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
- Implementing SQL in the adapter Resource class
- Sample adapter

## 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 adapter.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/mobilefirst_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 AdapterApplication class.

# Implementing 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

When the adapter is deployed, or whenever the configuration is changed from the MobileFirst console, the adapter's `MFPJAXRSApplication`'s `init` method is called. This is a good place to load the connection properties (`../#configuration-api`) and create a `DataSource`.

```
public class JavaSQLApplication extends MFPJAXRSApplication{

    public BasicDataSource dataSource = null;

    @Context
    ConfigurationAPI configurationAPI;

    @Override
    protected void init() throws Exception {
        dataSource= new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl(configurationAPI.getPropertyValue("DB_url"));
        dataSource.setUsername(configurationAPI.getPropertyValue("DB_username"));
        dataSource.setPassword(configurationAPI.getPropertyValue("DB_password"));
    }
}
```

In the resource class, create a helper method to get an SQL connection. Use the `AdaptersAPI` to get the current `MFPJAXRSApplication` instance:

```
@Context
AdaptersAPI adaptersAPI;

public Connection getSQLConnection() throws SQLException{
    // Create a connection object to the database
    JavaSQLApplication app = adaptersAPI.getJaxRsApplication(JavaSQLApplication.class);
    return app.dataSource.getConnection();
}
```

## 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.

## Sample usage

- Import the SQL script to a database.
- Make sure that the `mobilefirst@%` user has all access permissions assigned.
- Use either Maven or MobileFirst Developer CLI to build and deploy the JavaSQL adapter (`../../creating-adapters/`).
- To test or debug an adapter, see the testing and debugging adapters (`../../testing-and-debugging-adapters`) tutorial.