How to develop a JDBC Form Store Binder

In this tutorial, we will following the guideline of developing a plugin to develop our JDBC Form Store Binder plugin. Please also refer to the very first tutorial How to develop a Bean Shell Hash Variable and also the following JDBC related plugin for more details steps.

- How to develop a JDBC Options Binder
- How to develop a JDBC Form Load Binder

1. What is the problem?

For integration purpose, we would like to store our form data to a different database table instead of Joget form data table.

2. What is your idea to solve the problem?

Joget Workflow has provided a plugin type called Form Store Binder Plugin. We will develop one to support JDBC connection and custom query to store form data.

3. What is the input needed for your plugin?

To develop a JDBC Store binder, we will need the JDBC connection setting and also the custom query to store the form data based the collected form data.

1. Datasource: Using custom datasource or Joget default datasource
2. Custom JDBC Driver: The JDBC driver for custom datasource
3. Custom JDBC URL: The JDBC connection URL for custom datasource
4. Custom JDBC Username: The username for custom datasource
5. Custom JDBC Password: The password for custom datasource
6. SQL Check Exist Query: The query to check whether an insert or update query should be execute.
7. SQL Insert Query: The query to insert form data.
8. SQL Update Query: The query to insert form data.
9. SQL Delete Query: The query to delete deleted form data when used as multirow binder.

We will have to support a syntax to inject the form data to the query. "{foreignKey}" can be used for Multi Rows storing.

We will also need to support a syntax to inject UUID value. In this case, we will use "{uuid}".

Example: INSERT INTO app_fd_test VALUES ({id}, {name}, {email}, {phone}, {foreignKey});

4. What is the output and expected outcome of your plugin?

All submitted data will store accordingly based on the check/insert/update query.

5. Is there any resources/API that can be reuse?

We can refer to the implementation of other available Form Store Binder plugins. Joget default datasource can be retrieve with AppUtil.getApplicationContext().getBean("setupDataSource").

6. Prepare your development environment

We need to always have our Joget Workflow Source Code ready and builded by following this guideline.

The following of this tutorial is prepared with a Macbook Pro and Joget Source Code version 5.0.0. Please refer to Guideline for developing a plugin for other platform command.

Let said our folder directory as following.

```
- Home
  - joget
    - plugins
    - jw-community
      -5.0.0
```

The "plugins" directory is the folder we will create and store all our plugins and the "jw-community" directory is where the Joget Workflow Source code stored.

Run the following command to create a maven project in "plugins" directory.

```
cd joget/plugins/
~/joget/jw-community/5.0.0/wflow-plugin-archetype/create-plugin.sh org.joget.tutorial jdbc_store_binder 5.0.0
```

Then, the shell script will ask us to key in a version for your plugin and ask us for confirmation before generate the maven project.

```
Define value for property 'version':  1.0-SNAPSHOT: : 5.0.0
[INFO] Using property: package = org.joget.tutorial
Confirm properties configuration:
groupId: org.joget.tutorial
artifactId: jdbc_store_binder
version: 5.0.0
package: org.joget.tutorial
Y: : y
```

We should get "BUILD SUCCESS" message shown in our terminal and a "jdbc_store_binder" folder created in "plugins" folder.

Open the maven project with your favour IDE. I will be using NetBeans.

7. Just code it!

a. Extending the abstract class of a plugin type

Create a "JdbcStoreBinder" class under "org.joget.tutorial" package. Then, extend the class with **org.joget.apps.form.model.FormBinder** abstract class.

To make it work as a Form Store Binder, we will need to implement **org.joget.apps.form.model.FormStoreBinder** interface. Then, we need to implement **org.joget.apps.form.model.FormStoreElementBinder** interface to make this plugin show as a selection in store binder select box and implement **org.joget.apps.form.model.FormStoreMultiRowElementBinder** interface to list it under the store binder select box of grid element.

Please refer to Form Store Binder Plugin.

b. Implement all the abstract methods

As usual, we have to implement all the abstract methods. We will using AppPluginUtil.getMessage method to support i18n and using constant variable MESSAGE_PATH for message resource bundle directory.

Implementation of all basic abstract methods

```
package org.joget.tutorial;

import org.joget.apps.app.service.AppPluginUtil;
import org.joget.apps.app.service.AppUtil;
import org.joget.apps.form.model.Element;
import org.joget.apps.form.model.FormBinder;
import org.joget.apps.form.model.FormData;
import org.joget.apps.form.model.FormRowSet;
import org.joget.apps.form.model.FormStoreBinder;
import org.joget.apps.form.model.FormStoreElementBinder;
import org.joget.apps.form.model.FormStoreMultiRowElementBinder;

public class JdbcStoreBinder extends FormBinder implements FormStoreBinder, FormStoreElementBinder,
FormStoreMultiRowElementBinder {

    private final static String MESSAGE_PATH = "messages/JdbcStoreBinder";

    public String getName() {
        return "JDBC Store Binder";
    }

    public String getVersion() {
        return "5.0.0";
    }

    public String getClassName() {
        return getClass().getName();
    }

    public String getLabel() {
        //support i18n
        return AppPluginUtil.getMessage("org.joget.tutorial.JdbcStoreBinder.pluginLabel", getClassName(),
MESSAGE_PATH);
    }

    public String getDescription() {
        //support i18n
        return AppPluginUtil.getMessage("org.joget.tutorial.JdbcStoreBinder.pluginDesc", getClassName(),
MESSAGE_PATH);
    }

    public String getPropertyOptions() {
        return AppUtil.readPluginResource(getClassName(), "/properties/jdbcStoreBinder.json", null, true,
MESSAGE_PATH);
    }

    public FormRowSet store(Element element, FormRowSet rows, FormData formData) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods,
choose Tools | Templates.
    }
}
```

Then, we have to do a UI for admin user to provide inputs for our plugin. In getPropertyOptions method, we already specify our Plugin Properties Options definition file is locate at "/properties/jdbcStoreBinder.json". Let us create a directory "resources/properties" under "jdbc_store_binder/src/main" directory. After create the directory, create a file named "jdbcStoreBinder.json" in the "properties" folder.

In the properties definition options file, we will need to provide options as below. Please note that we can use "@@message.key@@" syntax to support i18n in our properties options.

```
[{
    title : '@@form.jdbcStoreBinder.config@@',
    properties : [{
        name : 'jdbcDatasource',
        label : '@@form.jdbcStoreBinder.datasource@@',
        type : 'selectbox',
        options : [{
            value : 'custom',
            label : '@@form.jdbcStoreBinder.customDatasource@@'
```

```
        },{
            value : 'default',
            label : '@@form.jdbcStoreBinder.defaultDatasource@@'
        }],
        value : 'default'
    },{
        name : 'jdbcDriver',
        label : '@@form.jdbcStoreBinder.driver@@',
        description : '@@form.jdbcStoreBinder.driver.desc@@',
        type : 'textfield',
        value : 'com.mysql.jdbc.Driver',
        control_field: 'jdbcDatasource',
        control_value: 'custom',
        control_use_regex: 'false',
        required : 'true'
    },{
        name : 'jdbcUrl',
        label : '@@form.jdbcStoreBinder.url@@',
        type : 'textfield',
        value : 'jdbc:mysql://localhost/jwdb?characterEncoding=UTF8',
        control_field: 'jdbcDatasource',
        control_value: 'custom',
        control_use_regex: 'false',
        required : 'true'
    },{
        name : 'jdbcUser',
        label : '@@form.jdbcStoreBinder.username@@',
        type : 'textfield',
        control_field: 'jdbcDatasource',
        control_value: 'custom',
        control_use_regex: 'false',
        value : 'root',
        required : 'true'
    },{
        name : 'jdbcPassword',
        label : '@@form.jdbcStoreBinder.password@@',
        type : 'password',
        control_field: 'jdbcDatasource',
        control_value: 'custom',
        control_use_regex: 'false',
        value : ''
    },{
        name : 'check_sql',
        label : '@@form.jdbcStoreBinder.check_sql@@',
        description : '@@form.jdbcStoreBinder.check_sql.desc@@',
        type : 'codeeditor',
        mode : 'sql',
        required : 'true'
    },{
        name : 'insert_sql',
        label : '@@form.jdbcStoreBinder.insert_sql@@',
        description : '@@form.jdbcStoreBinder.insert_sql.desc@@',
        type : 'codeeditor',
        mode : 'sql',
        required : 'true'
    },{
        name : 'update_sql',
        label : '@@form.jdbcStoreBinder.update_sql@@',
        description : '@@form.jdbcStoreBinder.update_sql.desc@@',
        type : 'codeeditor',
        mode : 'sql',
        required : 'true'
    },{
        name : 'delete_sql',
        label : '@@form.jdbcStoreBinder.delete_sql@@',
        description : '@@form.jdbcStoreBinder.delete_sql.desc@@',
        type : 'codeeditor',
        mode : 'sql',
        required : 'true'
    }],
    buttons : [{
```

```
        name : 'testConnection',
        label : '@@form.jdbcStoreBinder.testConnection@@',
        ajax_url : '[CONTEXT_PATH]/web/json/app[APP_PATH]/plugin/org.joget.tutorial.JdbcStoreBinder/service?
action=testConnection',
        fields : ['jdbcDriver', 'jdbcUrl', 'jdbcUser', 'jdbcPassword'],
        control_field: 'jdbcDatasource',
        control_value: 'custom',
        control_use_regex: 'false'
    }]
}]
```

Same as JDBC Options Binder, we will need to add a test connection button for custom JDBC setting. Please refer to How to develop a JDBC Options Binder on the Web Service Plugin implementation.

Once we done the properties option to collect input and the web service to test the connection, we can work on the main method of the plugin which is store method.

```
    public FormRowSet store(Element element, FormRowSet rows, FormData formData) {
        Form parentForm = FormUtil.findRootForm(element);
        String primaryKeyValue = parentForm.getPrimaryKeyValue(formData);

        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            DataSource ds = createDataSource();
            con = ds.getConnection();

            //check for deletion
            FormRowSet originalRowSet = formData.getLoadBinderData(element);
            if (originalRowSet != null && !originalRowSet.isEmpty()) {
                for (FormRow r : originalRowSet) {
                    if (!rows.contains(r)) {
                        String query = getPropertyString("delete_sql");
                        pstmt = con.prepareStatement(getQuery(query));
                        int i = 1;
                        for (String obj : getParams(query, r, primaryKeyValue)) {
                            pstmt.setObject(i, obj);
                            i++;
                        }
                        pstmt.executeUpdate();
                    }
                }
            }

            if (!(rows == null || rows.isEmpty())) {

                //run query for each row
                for (FormRow row : rows) {
                    //check to use insert query or update query
                    String checkSql = getPropertyString("check_sql");
                    pstmt = con.prepareStatement(getQuery(checkSql));
                    int i = 1;
                    for (String obj : getParams(checkSql, row, primaryKeyValue)) {
                        pstmt.setObject(i, obj);
                        i++;
                    }
                    String query = getPropertyString("insert_sql");
                    rs = pstmt.executeQuery();
                    //record exist, use update query
                    if (rs.next()) {
                        query = getPropertyString("update_sql");
                    }
                    pstmt = con.prepareStatement(getQuery(query));
                    i = 1;
                    for (String obj : getParams(query, row, primaryKeyValue)) {
                        pstmt.setObject(i, obj);
                        i++;
                    }
                    pstmt.executeUpdate();
                }
```

```
            }
        } catch (Exception e) {
            LogUtil.error(getClassName(), e, "");
        } finally {
            try {
                if (rs != null) {
                    rs.close();
                }
                if (pstmt != null) {
                    pstmt.close();
                }
                if (con != null) {
                    con.close();
                }
            } catch (Exception e) {
                LogUtil.error(getClassName(), e, "");
            }
        }

        return rows;
    }

    /**
     * Used to replaces all syntax like {field_id} to question mark
     * @param query
     * @return
     */
    protected String getQuery(String query) {
        return query.replaceAll("\\{[a-zA-Z0-9_]+\\}", "?");
    }

    /**
     * Used to retrieves the value of variables in query
     * @param query
     * @param row
     * @return
     */
    protected Collection<String> getParams(String query, FormRow row, String primaryKey) {
        Collection<String> params = new ArrayList<String>();

        Pattern pattern = Pattern.compile("\\{([a-zA-Z0-9_]+)\\}");
        Matcher matcher = pattern.matcher(query);

        while (matcher.find()) {
            String key = matcher.group(1);

            if (FormUtil.PROPERTY_ID.equals(key)) {
                String value = row.getId();
                if (value == null || value.isEmpty()) {
                    value = UuidGenerator.getInstance().getUuid();
                    row.setId(value);
                }
                params.add(value);
            } else if ("uuid".equals(key)) {
                params.add(UuidGenerator.getInstance().getUuid());
            } else if ("foreignKey".equals(key)) {
                params.add(primaryKey);
            } else {
                String value = row.getProperty(key);
                params.add((value != null)?value:"");
            }
        }

        return params;
    }

    /**
     * To creates data source based on setting
     * @return
     * @throws Exception
     */
```

```
    protected DataSource createDataSource() throws Exception {
        DataSource ds = null;
        String datasource = getPropertyString("jdbcDatasource");
        if ("default".equals(datasource)) {
            // use current datasource
            ds = (DataSource)AppUtil.getApplicationContext().getBean("setupDataSource");
        } else {
            // use custom datasource
            Properties dsProps = new Properties();
            dsProps.put("driverClassName", getPropertyString("jdbcDriver"));
            dsProps.put("url", getPropertyString("jdbcUrl"));
            dsProps.put("username", getPropertyString("jdbcUser"));
            dsProps.put("password", getPropertyString("jdbcPassword"));
            ds = BasicDataSourceFactory.createDataSource(dsProps);
        }
        return ds;
    }
```

c. Manage the dependency libraries of your plugin

Our plugin is using dbcp, javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse class, so we will need to add jsp-api and commons-dbcp library to our POM file.

```
<!-- Change plugin specific dependencies here -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.0</version>
</dependency>
<dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
    <version>1.3</version>
</dependency>
<!-- End change plugin specific dependencies here -->
```

d. Make your plugin internationalization (i18n) ready

We are using i18n message key in getLabel and getDescription method. We also used i18n message key in our properties options definition as well. So, we will need to create a message resource bundle properties file for our plugin.

Create directory "resources/messages" under "jdbc_store_binder/src/main" directory. Then, create a "JdbcStoreBinder.properties" file in the folder. In the properties file, let add all the message keys and its label as below.

```
org.joget.tutorial.JdbcStoreBinder.pluginLabel=JDBC Binder
org.joget.tutorial.JdbcStoreBinder.pluginDesc=Used to store form data using JDBC
form.jdbcStoreBinder.config=Configure JDBC Binder
form.jdbcStoreBinder.datasource=Datasource
form.jdbcStoreBinder.customDatasource=Custom Datasource
form.jdbcStoreBinder.defaultDatasource=Default Datasource
form.jdbcStoreBinder.driver=Custom JDBC Driver
form.jdbcStoreBinder.driver.desc=Eg. com.mysql.jdbc.Driver (MySQL), oracle.jdbc.driver.OracleDriver (Oracle),
com.microsoft.sqlserver.jdbc.SQLServerDriver (Microsoft SQL Server)
form.jdbcStoreBinder.url=Custom JDBC URL
form.jdbcStoreBinder.username=Custom JDBC Username
form.jdbcStoreBinder.password=Custom JDBC Password
form.jdbcStoreBinder.check_sql=SQL SELECT Query
form.jdbcStoreBinder.check_sql.desc=Used  to decide an insert or update operation. Use syntax like {field_id}
in query to inject submitted form data.
form.jdbcStoreBinder.insert_sql=SQL INSERT Query
form.jdbcStoreBinder.insert_sql.desc=Use syntax like {field_id} in query to inject submitted form data.
form.jdbcStoreBinder.update_sql=SQL UPDATE Query
form.jdbcStoreBinder.update_sql.desc=Use syntax like {field_id} in query to inject submitted form data.
form.jdbcStoreBinder.delete_sql=SQL DELETE Query
form.jdbcStoreBinder.delete_sql.desc=Used to delete deleted form data in Grid element. Use syntax like {id} in
query to inject form data primary key.
form.jdbcStoreBinder.testConnection=Test Connection
form.jdbcStoreBinder.connectionOk=Database connected
form.jdbcStoreBinder.connectionFail=Not able to establish connection.
```

e. Register your plugin to Felix Framework

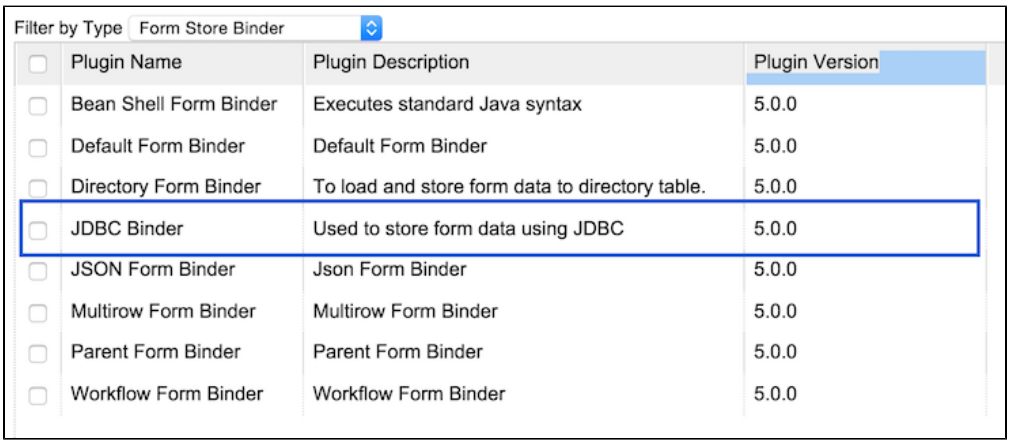
We will have to register our plugin class in Activator class (Auto generated in the same class package) to tell Felix Framework that this is a plugin.

```
    public void start(BundleContext context) {
        registrationList = new ArrayList<ServiceRegistration>();
        //Register plugin here
        registrationList.add(context.registerService(JdbcStoreBinder.class.getName(), new JdbcStoreBinder(),
null));
    }
```

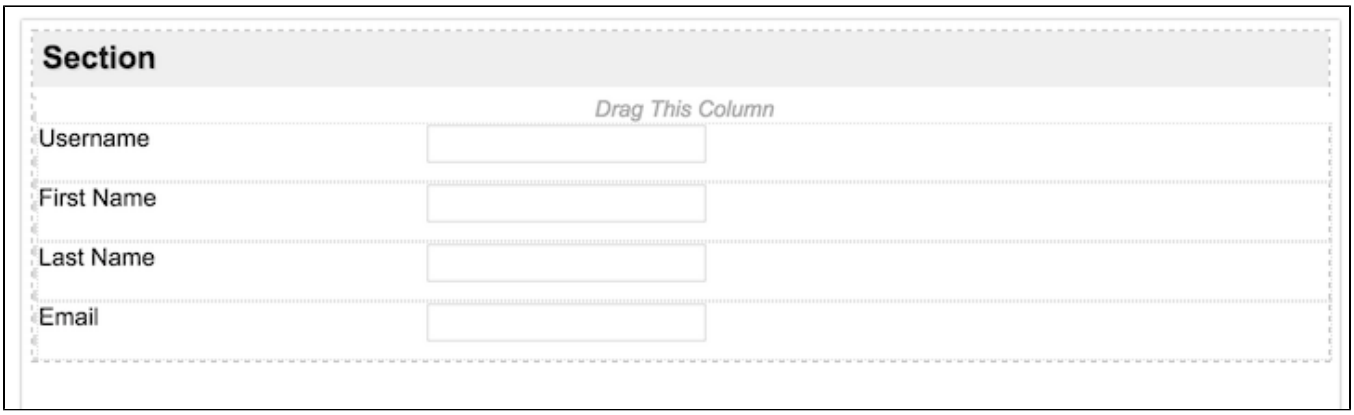
f. Build it and testing

Let build our plugin. Once the building process is done, we will found a "jdbc_store_binder-5.0.0.jar" file is created under "jdbc_store_binder/target" directory.

Then, let upload the plugin jar to Manage Plugins. After upload the jar file, double check the plugin is uploaded and activated correctly.



Let create a form to create and update user to dir_user table.



Then, configure the store binder of the form with the following query.

Check Select Query

```
select username from dir_user where username = {id}
```

Insert Query

```
insert into dir_user
(id, username, firstName, lastName, email, active)
values
({id}, {id}, {firstName}, {lastName}, {email}, 1)
```

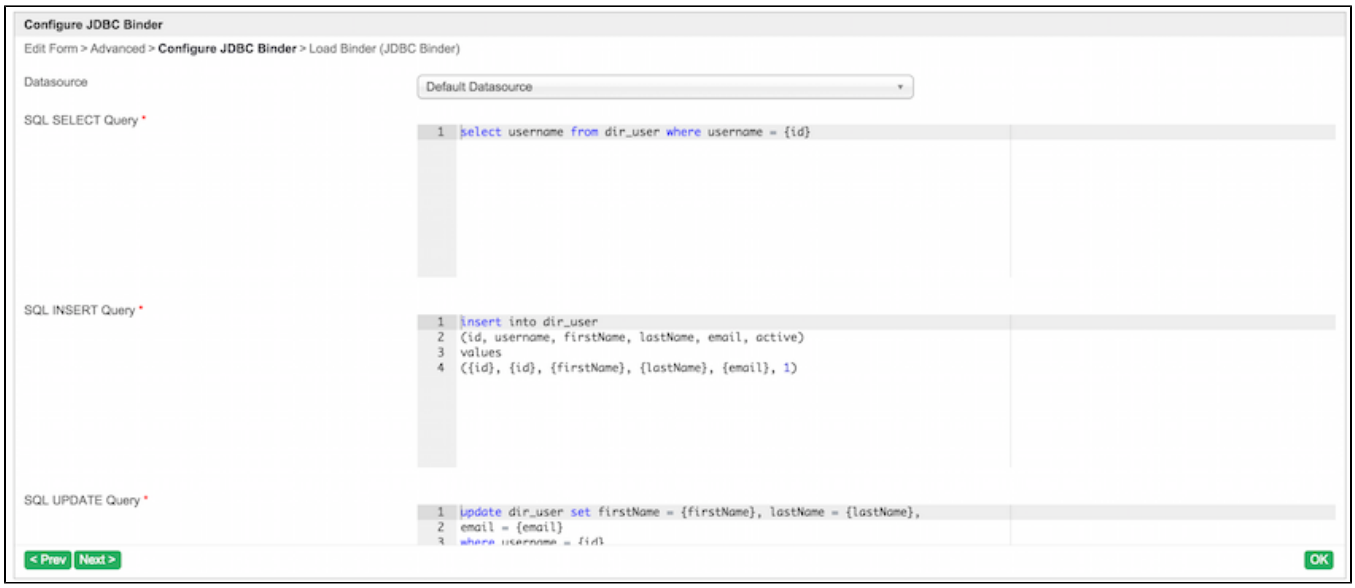
note: {uuid} can be used to generate a unique id

Update Query

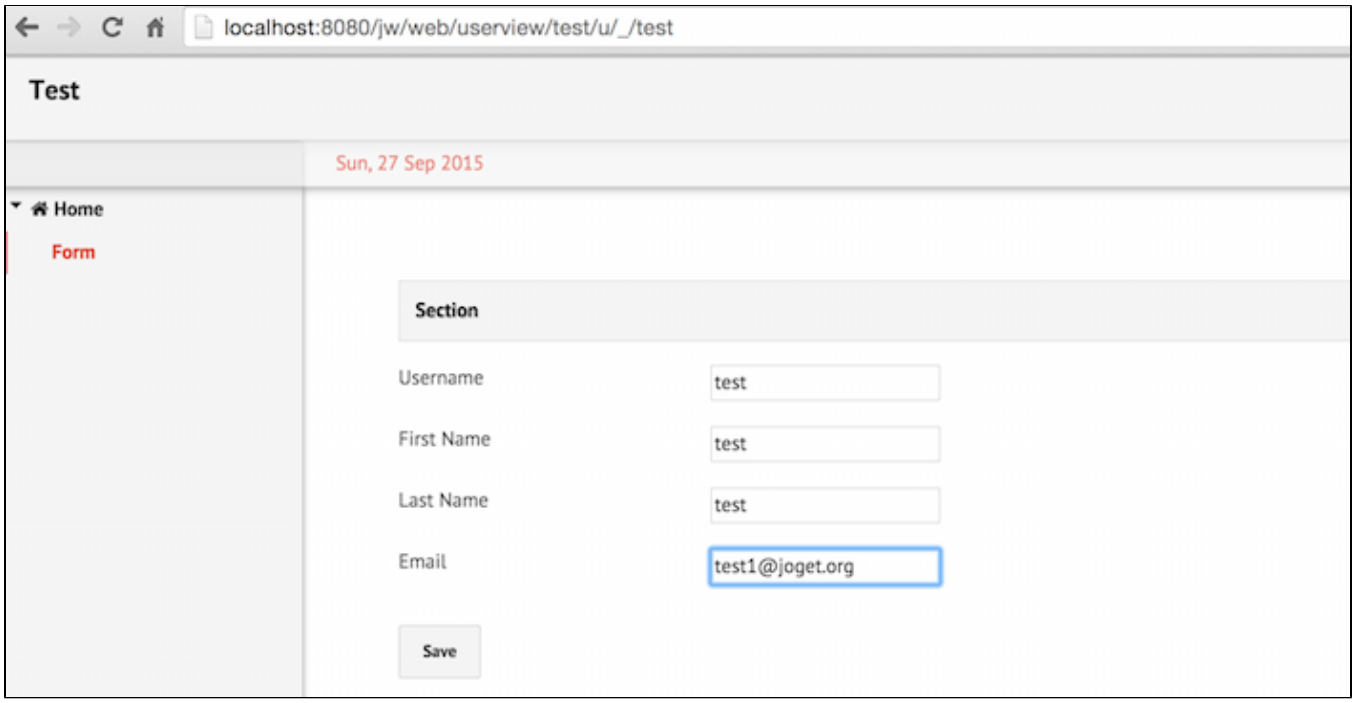
```
update dir_user set firstName = {firstName}, lastName = {lastName},
email = {email}
where username = {id}
```

Delete Query

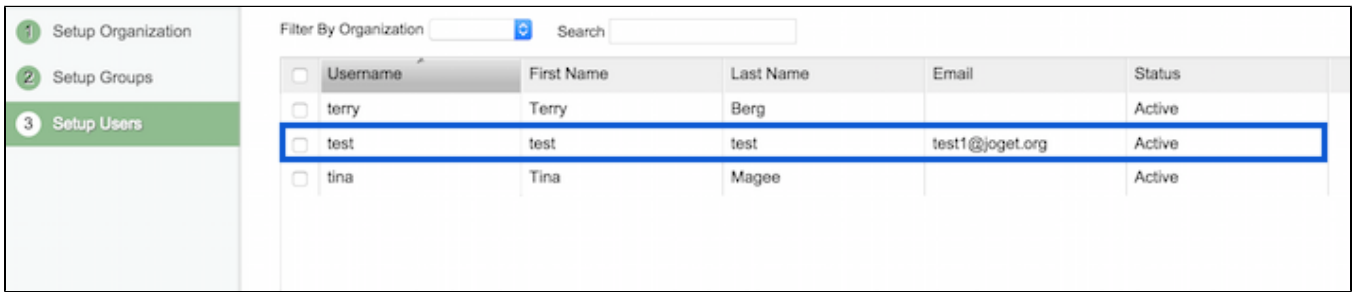
```
delete from TABLE_NAME where id = {id}
```



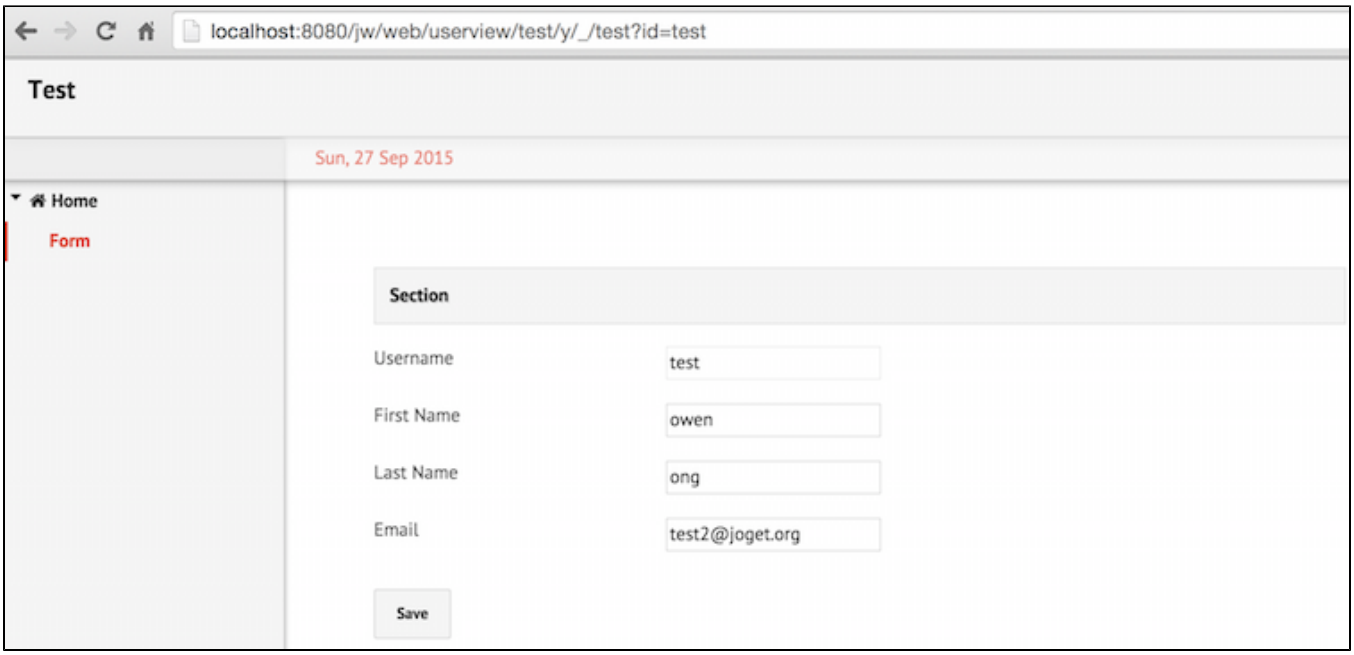
Now, let test to add a user.



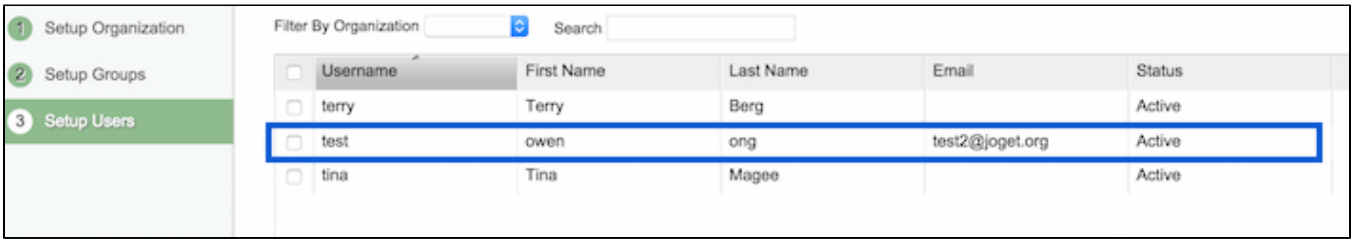
Check the user is created in dir_user table.



Let update the same record by pass the id in URL parameters.



Check the user is updated.



It works! Please remember test the other features of the plugin as well. 😃

8. Take a step further, share it or sell it

You can download the source code from jdbc_store_binder_src.zip

To download the ready-to-use plugin jar, please find it in http://marketplace.joget.org/.