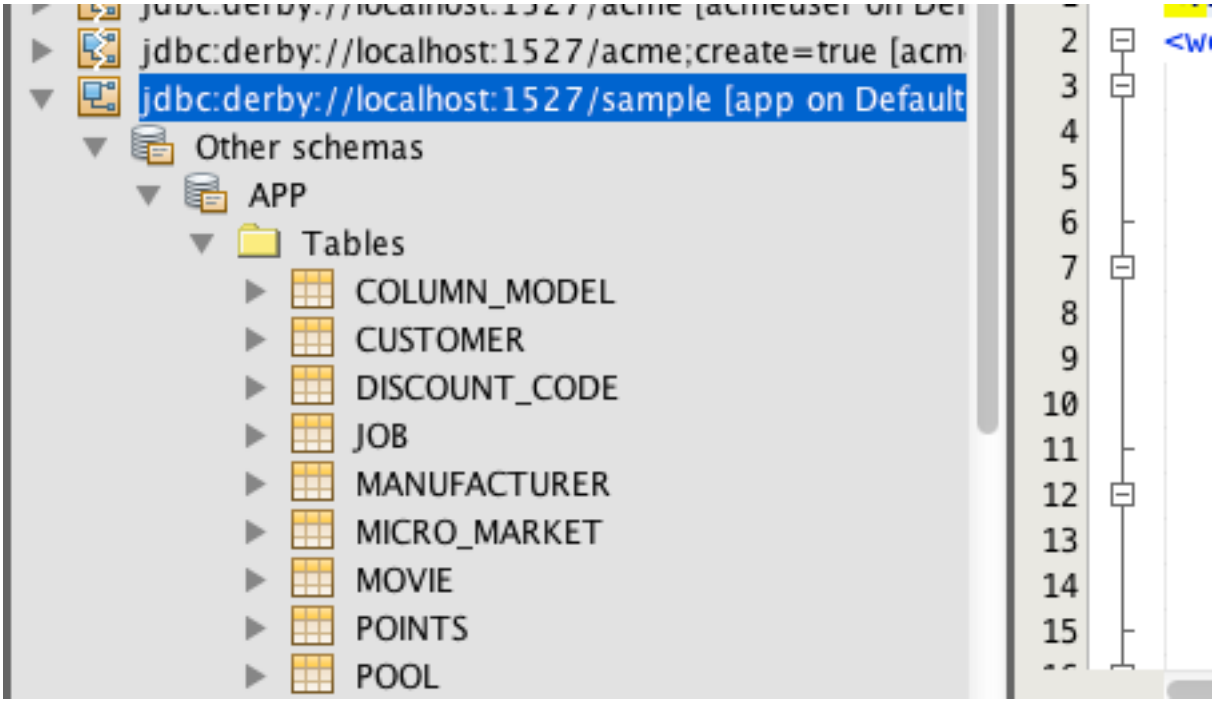
JSF Application from Scratch - Customers

In this tutorial, we will build a simple JSF application using the NetBeans IDE. This tutorial utilizes NetBeans 8.0.2 and GlassFish 4.1 for deployment. When downloading NetBeans, be sure to grab either the “Java EE” or “All” bundle in order to ensure that you have all required libraries.

** If not using NetBeans, please see the source archive for the Customers application, and copy/paste sources into your project.

Database

Once downloaded and installed, ensure that the Apache Derby database starts up when opening NetBeans. To check for a database connection to Derby, click on the Services tab and then expand Databases. There should be a connection for jdbc:derby://localhost:1527/sample. After double clicking the connection, expand the APP schema to see the table listing. If the APP user does not exist, or if the APP user does not have the CUSTOMER database table, then create the database by running the script https://github.com/juneau001/AcmePools/blob/master/sql/create_database-no-netbeans.sql in another Apache Derby user schema.



Application

Next, perform the following steps to create the application.

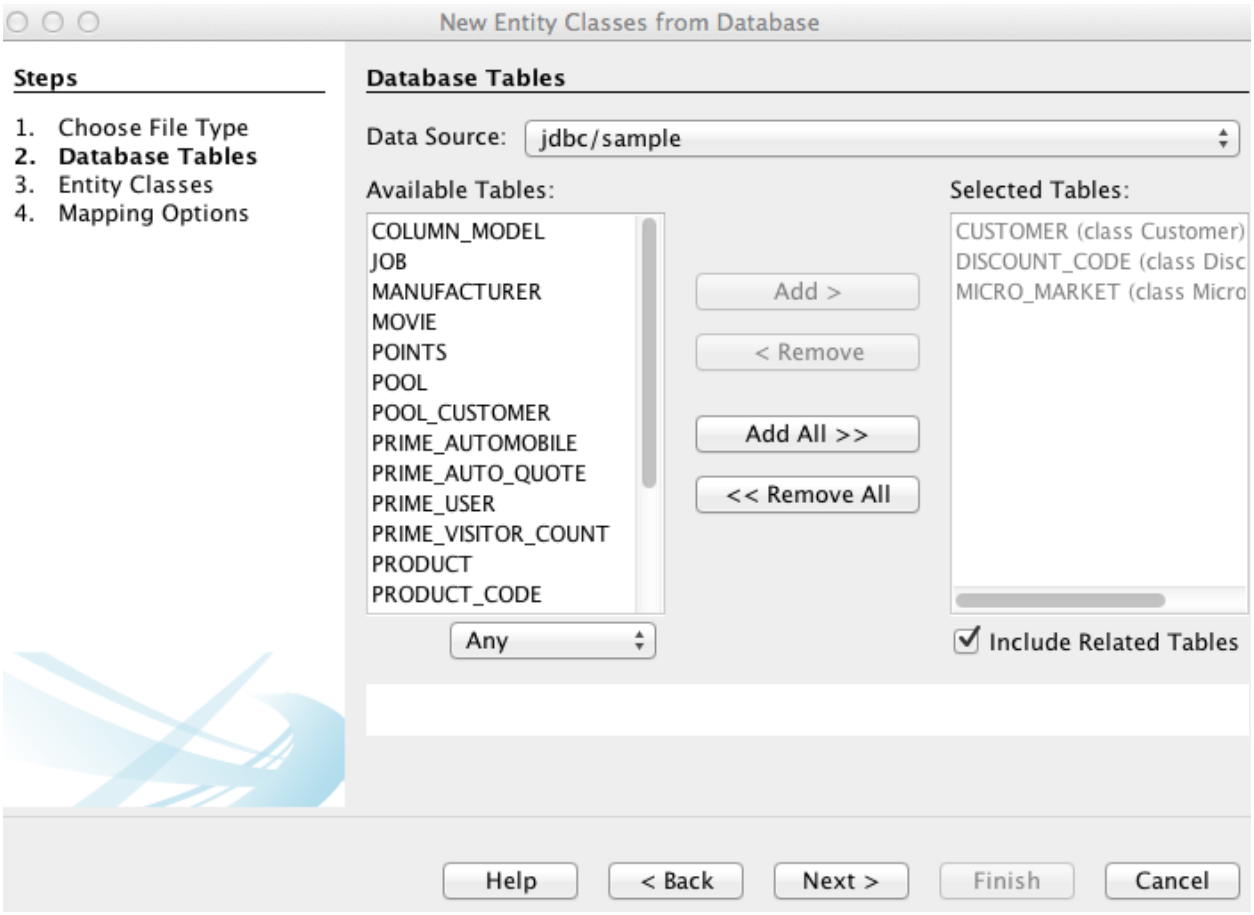
1) Create new “Maven Web Application” project within NetBeans. Fill in the following details:
   Project Name: Customers
   Project Location: Your Choice

Group ID: org.javaserverfaces
Version: 0.1
Package: org.javaserverfaces.customers

2) Settings:
Server: GlassFish 4.1
Java EE Version: Java EE 7

3) Create a new package that will be used for organization of the project's entity classes. Expand the "Source Packages" node, and right click on org.javaserverfaces.customers, select New..->Packages… Name the new package org.javaserverfaces.customers.entity. Click Finish.

4) Create an entity class for CUSTOMER database table. Right click on the newly created package, and select New… -> Entity Classes from Database. When the "New Entity Classes from Database" dialog opens, select the datasource "jdbc/sample" to connect to the sample database. This will populate the "Available Tables" list. Select CUSTOMER, and add it to the "Selected Tables" list. This will also cause the tables DISCOUNT_CODE and MICRO_MARKET to populate for referential purposes. Click next…accept all defaults within the next screen, and click "Finish". The entity classes for the database tables will be generated.



5) Create session bean EJBs for the entity classes. One may choose to create web services, rather than EJBs, but in this tutorial we will utilize the latter. Right-click on “Source Packages” and select “New” -> “Java Package”. Name the package org.javaserverfaces.customers.session, then click Finish. Right-click on the newly created package, and select “New” -> “Session Beans for Entity Classes” to open the “New Session Beans for Entity Classes” dialog. Select “org.javaserverfaces.entity.Customer” from the “Available Classes” list, and add it to the “Selected Classes” list. Doing so will cause the other two classes to move into the :Selected Classes” list as well. Click Next, accept all defaults on the next screen, and click “Finish”.

6) Create a JSF managed bean controller for the CustomerFacade. First, create a new package to contain the managed bean controllers by right-clicking on “Source Packages”, and select “New” -> “Java Package”. Name the package org.javaserverfaces.customers.jsf, then click Finish. Next, right-click on the newly created package and select “New” -> “JSF Managed Bean”, which will open the “New JSF Managed Bean” dialog. In the dialog, fill in the following:

Class Name: CustomerController
Name: customerController (CDI Name)
Scope: Session

Accept the remaining defaults and click “Finish.

The JSF Managed bean controller is the class that will communicate with the JSF view (web page). Typically, each view within an application contains a JSF managed bean controller class. The JSF view is able to interact with the public fields and methods of a managed bean controller via the use of Expression Language (EL).

7) Add a list of customers to the main view of the application. In order to facilitate this listing, we need to query the Customer entity for all records. To do so, let’s open the CustomerController JSF managed bean, and inject the CustomerFacade EJB into the controller by creating the following class field:

```
@EJB
CustomerFacade customerFacade;
```

Import the necessary classes, and then create a new private class field of type List<Customer>:

```
private List<Customer> customerList;
```

Import the necessary classes, and encapsulate the customerList field to create accessor methods (getter and setter) by right-clicking on the field name, and choosing “Refactor” -> “Encapsulate Fields”, and finally accepting the dialog defaults and choosing “Refactor”.

Lastly, create a public method which will be used for populating the customerList field. Name the method populateCustomerList(), and provide the following implementation:

```
public void populateCustomerList(){
    customerList = customerFacade.findAll();
```

}

We will be calling upon the populateCustomerList() method each time the JSF view containing the customer listing is loaded into the browser. In turn, the populateCustomerList() method will populate the customerList with the list of customers from the database, and they will then be displayed in the DataTable, which we will create next.

8) Create the front end table (*JSF DataTable*) to list the customers in the database. Begin by generating a new XHTML view within he "Web Pages" folder, and name the view customerList.xhtml. Delete the contents of the new view, and paste in the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <title>TODO supply a title</title>

    </h:head>
    <f:metadata>
        <f:viewAction action="#{customerController.populateCustomerList()}"/>
    </f:metadata>
    <h:body>
        <h1>Customer Listing</h1>
        <h:form id="customerListingForm">
            <h:dataTable id="customerListing" value="#{customerController.customerList}"
                         var="customer" border="1">
                <h:column id="name">
                    <h:outputText value="#{customer.name}"/>
                </h:column>
                <h:column id="email">
                    <h:outputText value="#{customer.email}"/>
                </h:column>
                <h:column id="phone">
                    <h:outputText value="#{customer.phone}"/>
                </h:column>
                <h:column id="address1">
                    <h:outputText value="#{customer.addressline1}"/>
                </h:column>
                <h:column id="address2">
                    <h:outputText value="#{customer.addressline2}"/>
                </h:column>
                <h:column id="city">
                    <h:outputText value="#{customer.city}"/>
                </h:column>
```

```
            <h:column id="state">
               <h:outputText value="#{customer.state}"/>
            </h:column>
            <h:column id="zip">
               <h:outputText value="#{customer.zip}"/>
            </h:column>
         </h:dataTable>
      </h:form>
   </h:body>
</html>
```

In the source above, the populateCustomerList() method, residing in the CustomerController JSF managed bean class, is initiated when the view is loaded via the JSF ViewAction that is located newer the top of the view. The JSF ViewAction must reside within an <f:metadata> tag.

The DataTable is populated via EL by obtaining the current list of users within the CustomerController's customerList field, as seen in bold. The var attribute of the DataTable provides a handle for each record within the List. Therefore, we can access each field of every customer record within the list, as seen within the <h:column> components that are contained within the DataTable.

9) Ensure that the project properties are correct. Right-click on the project in NetBeans IDE, and select "Properties". Click "Sources" menu and be sure that either Java 1.7 or 1.8 is selected as the Source/Binary format. Click "Compile" and be sure that JDK 1.7 or JDK 1.8 is selected as the Java Platform. Click "Run" and ensure that GlassFish is selected as the Server, and that Java EE 7 is selected as the Java EE Version. Exit the form by clicking "OK".

At this point, open the NetBeans project's WEB-INF directory, and see if a faces-config.xml file has been automatically generated for the project. NetBeans should automatically create this for us when we choose the Create JSF Pages for Entity Classes option (next section), but we can manually generate one by right-clicking on the project, and then choosing New…JavaServer Faces -> Faces Config.

10) Deploy and test the application. Click on web.xml, and update the welcome-file to "faces/customerList.xhtml" so that the new customer listing is displayed on start up. Right-click the project in NetBeans IDE, and select "Run" to compile, build, and deploy the project to GlassFish.

Creating a CRUD Application

Now that the entity classes have been created, and we have generated a simple view to display customers, lets create a CRUD application based upon the Customer entity. (*Note: If not using NetBeans, you can copy the sources from the Customers source archive into your application project.)* Generate the JSF views by right-clicking the project's Web Pages folder and choosing **New** and then **JSF Pages from Entity Classes**. When the dialog box opens, select each of the newly created entity classes, add them to the Selected Entity Classes list, and then choose **Next**.

Utilizing Standard JSF Views (Recommended for Testing JSF 2.3)

On the next screen, change the Session Bean package to org.javaserverfaces.customers.session, and change the JSF Classes Package to org.javaserverfaces.customers.jsf. Enter a / into the field labeled "JSF Pages Folder". Check the box that reads "Override Existing Classes". Select the "Choose Templates" option; and choose **Standard JavaServer Faces**. Click **Finish**, and NetBeans IDE will create three separate folders within the Web Pages folder, each named accordingly for their associated entity classes. Within each folder, the following JSF pages are created: `Create.xhtml`, `Edit.xhtml`, `List.xhtml`, and `View.xhtml`. NetBeans IDE also automatically generates JSF controllers and Enterprise JavaBeans (EJB) beans for each of the entity classes, and the newly generated pages are wired up to the controllers and ready for use.

Edit the web.xml, adding the following for the welcome-file:
<welcome-file>faces/customer/List.xhtml</welcome-file>

Run the application, and perform some tests to ensure that everything works properly.

Open each view, and inspect the EL that is used to access the fields and methods of the JSF managed bean classes. Each entity (Customer, DiscountCode, and MicroMarket) each has its own set of views, and therefore, each also has its own JSF managed bean controller class. CDI is used to access each of the JSF managed bean controller classes, and this is made possible via the @Named annotation on each of the JSF managed bean controller classes. The @Named annotation provides an accessor name for each class.

Utilizing a Third Party UI Library (PrimeFaces)

Begin by removing each of the view directories that were generated in the previous section. Next, Generate the JSF views by right-clicking the project's Web Pages folder and choosing **New** and then **JSF Pages from Entity Classes**. When the dialog box opens, select each of the newly created entity classes, add them to the Selected Entity Classes list, and then choose **Next**. On the next screen, change the Session Bean package to org.javaserverfaces.customers.session, and change the JSF Classes Package to org.javaserverfaces.customers.jsf. Enter a / into the field labeled "JSF Pages Folder". Check the box that reads "Override Existing Classes". Select the "Choose Templates" option; and choose **PrimeFaces**. Click **Finish**, and NetBeans IDE will create three separate folders within the Web Pages folder, each named accordingly for their associated entity classes. Within each folder, the following JSF pages are created: `Create.xhtml`, `Edit.xhtml`, `List.xhtml`, and `View.xhtml`. NetBeans IDE also automatically generates JSF controllers and Enterprise JavaBeans (EJB) beans for each of the entity classes, and the newly generated pages are wired up to the controllers and ready for use.

Run the application, and perform some tests to ensure that everything works properly. Open a view within the editor and notice that the PrimeFaces namespace has been imported, and all of the UI components are from the PrimeFaces library.

https://netbeans.org/kb/docs/web/jsf20-intro.html