Enterprise and Cloud Computing
Lab 4: Java EE application, and adding queries.

Lab sheets 2 and 3 developed web apps only. While these had many of the components of Java EE (entity classes, session beans), they were still web apps rather than a Java EE app, and so won't scale. This labs creates a Java EE app. The main differences are:

- Session beans and entity classes are in a separate module from the JSF /web tier code, and so can be deployed on a different node on the network.
- Session beans are running in a Java EE container, allowing the application to scale.

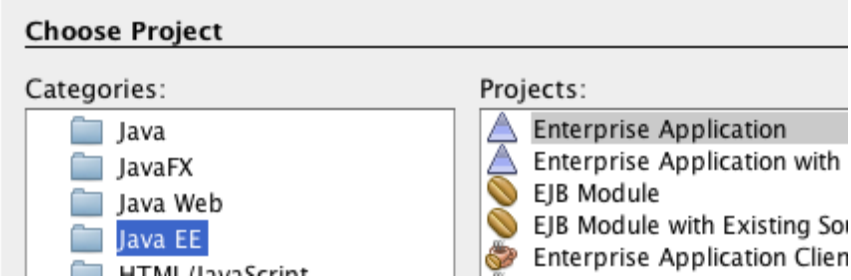
Customer List

Filter by state All states apply filter

Customer ID	Name	City	State
1	JumboCom	Fort Lauderdale	FL
2	Livermore Enterprises	Miami	FL
25	Oak Computers	Houston	TX
3	Nano Apple	Alanta	GA
36	HostProCom	San Mateo	CA
106	CentralComp	San Jose	CA
149	Golden Valley Computers	Santa Clara	CA
863	Top Network Systems	Redwood City	CA
777	West Valley Inc.	Dearborn	MI
753	Ford Motor Co	Dearborn	MI
722	Big Car Parts	Detroit	MI
409	New Media Productions	New York	NY
410	Yankee Computer Repair	New York	NY

The steps below will build a web tier to view the details of the Customer table in the sample derby database, adding a filter to view customers by state.

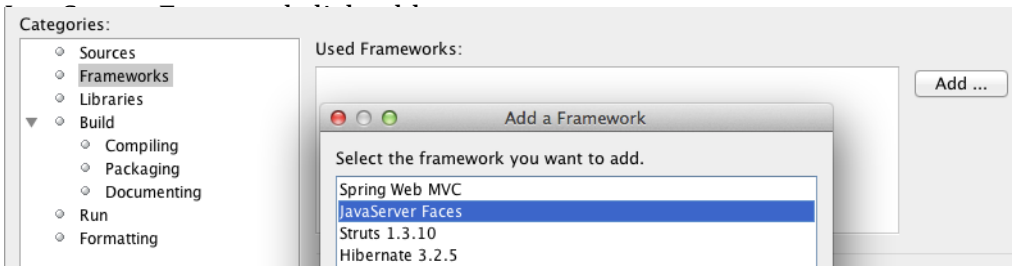
1. Start a new project in NetBeans selecting Java EE and Enterprise application (NOT web application).



2. Call the project **JEEapp**, leave all defaults and **click finish**. This creates three modules in Netbeans: JEEapp, JEEapp-ejb and JEEapp-war. JEEapp-ejb holds the business logic and JPA components (session beans and entity classes). JEEapp-war will contain the web tier (JSF pages and JSF backing beans). The entire application can be run from the **JEEapp node**. Look at the libraries for **JEEapp-war** – you should see JEEapp-ejb included, meaning classes from JEEapp-ejb are accessible to this web archive (war) module.

3. When creating the Java EE app, there wasn't an option to include the JSF framework, so the next step is to add JSF support to the web archive (JEEapp-war) module as follows:

Right click on JEEapp-war, select **properties**, under categories click on **Frameworks**, click on the **Add…** button on the right hand side, select

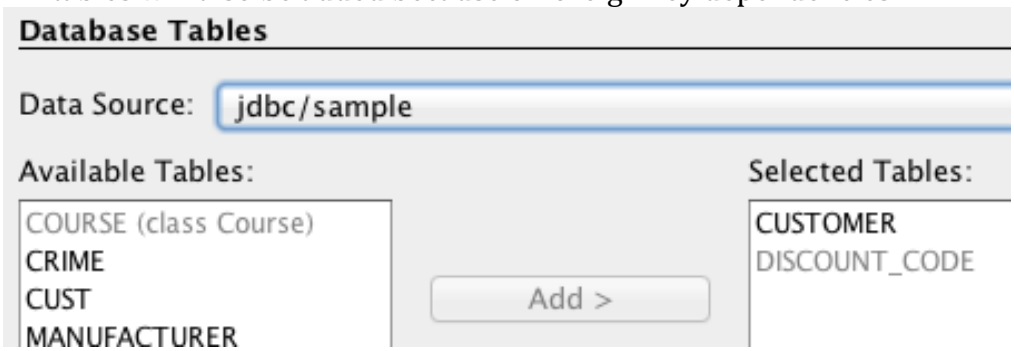


Run the application to check it is working up to this point. You should see the standard JSF welcome screen.

The following steps create the entity classes and session beans in the JEEabb-ejb module. Lab 3 created a database table from an entity class. This lab will create an entity class from an existing database table as follows:

4. Right click on **JEEapp-ejb**, and select **New > Entity Classes from Database**. If that option is not listed, you can get to it via **other>persistence**.

   Enter the following parameters:
   - **a.** Datasource **is jdbc/sample**
   - b. Select the customer table and click **add**. The discount and micro maret tables will also be added because of foreign key dependencies.



   - c. Click NEXT, and make the package **entity**. Leave other defaults as is, and **click finish**.

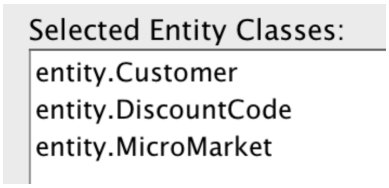
Open the code for the customer entity bean. You will see a number of named (static) queries to retrieve customers based on 'where clauses' for each attribute in the table, including one to select customer by state:
@NamedQuery(name = "Customer.findByState", query = "SELECT c FROM Customer c WHERE c.state = :state"). We will call this from a session bean in a later step.

Scroll down to the attribute declarations. Regular expressions have been generated for a number of attributes based on their database definitions. These can be used as validators:

```
// @Pattern(regexp="[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?", message="Invalid email")
```

The next step is to create session beans for the entity classes:

5. Right click on **JEEapp-ejb** again, and select **New > Session Beans for Entity Classes**. If that option is not listed, you can get to it via **other>Enterprise JavaBeans**.



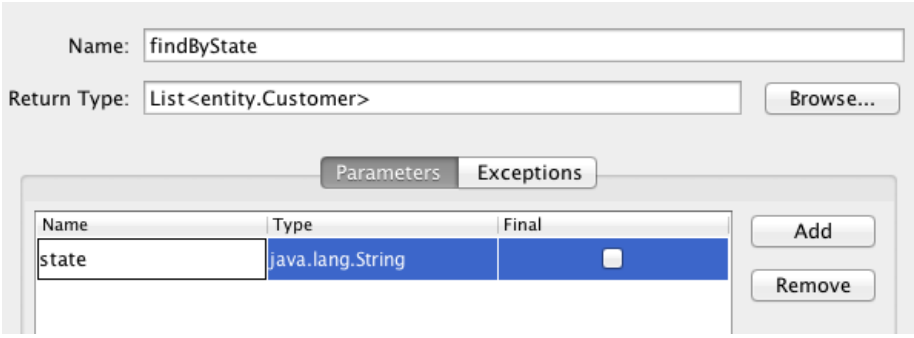
   a. Create session beans for the three entity beans listed.
   b. Put the session beans in a package called '**session**' and leave other defaults.
   c. Don't create interfaces for now.

   Click Finish.

This creates FOUR session beans: **AbstractFaçade** with the common methods for each of the session beans (CRUD methods, count, findall and findrange); **CustomerFacade** to link to the Customer entity class; **MicroMarketFaçad**e to link to the Micro Market customer class, and **DiscountFaçade** to link to the Discount entity class.

6. Open **CustomerFaçade**. We want to add a new method that will run a query based on a '**state**' entered into a filter on the user interface. Add a new business method to the session bean by right clicking in the code editor, select **Insert Code**, and select **Business Method**.

   a. Call the method **findByState**
   b. The return type is a list of customers: **List<entity.Customer>**
   c. Click **Add** to add a Parameter of type string called **state**



This creates the method signature that will accept a state name, and return a list of customers in that state.

Add the following code to the method, which calls the **findByState** named query (declared in the entity bean), as sets the query parameter to the method parameter **state**. The list of customers in that state will be returned.

```
if (state.isEmpty()) {
   Query query = em.createNamedQuery("Customer.findAll");
   return query.getResultList();
} else {
   Query query = em.createNamedQuery("Customer.findByState");
   query.setParameter("state", state);
   return query.getResultList();
}
```

Fix the imports (import java.util.List; import javax.persistence.Query;)

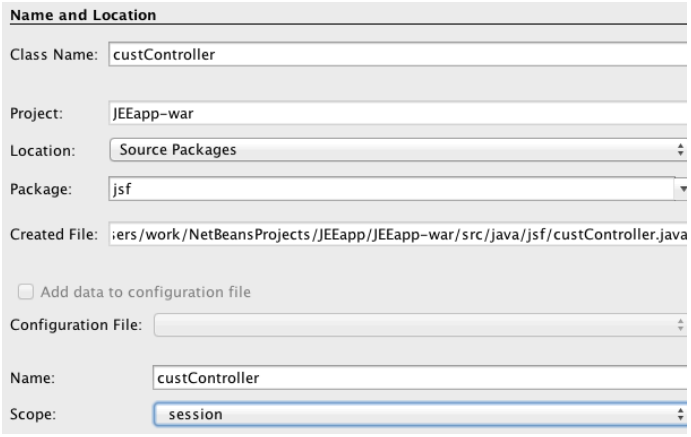
That's the business logic complete. The following steps return to the web tier (the WAR module) to create the front end:

7. Add the following code the index.**xhtml**:

```
<h1>Customer List</h1>
      <h:form >
            <h:panelGroup id="messagePanel" layout="block">
               <h:messages errorStyle="color: red" infoStyle="color: green"
layout="table"/>
            </h:panelGroup>

<!--the Filter -->
         <h:panelGrid columns="3">
               <h:outputLabel value="Filter by state" for="selectState" />
<h:selectOneMenu id="selectState" label="selectedState"
value="#{custController.stateFilter}">
               <f:selectItem itemLabel="All states" itemValue="all"/>
               <f:selectItem itemLabel="Florida" itemValue="FL"  />
               <f:selectItem itemLabel="California" itemValue="CA"/>
               <f:selectItem itemLabel="Texas" itemValue="TX"/>
               <f:selectItem itemLabel="NewYork" itemValue="NY"/>
               <f:selectItem itemLabel="Mississippi" itemValue="MI"/>
               <f:selectItem itemLabel="Georgia" itemValue="GA"/>
            </h:selectOneMenu>

 <!--the Filter's command button calling updateItems() to update the
datatable based on the selected state -->
<h:commandButton value="apply filter"
action="#{custController.updateItems()}"/>
```

```
</h:panelGrid>

<!-- checking there is data to display -->
<h:outputText escape="false" value="No customers were found"
rendered="#{custController.items.rowCount == 0}"/>

<h:panelGroup rendered="#{custController.items.rowCount > 0}">
<!-- the data table of customer details, create from the items object -->
<h:dataTable value="#{custController.items}" var="item" border="0"
cellpadding="2" cellspacing="0" rules="all" style="border:solid 1px">
            <h:column>
              <f:facet name="header">
                <h:outputText value="Customer ID"/>
              </f:facet>
              <h:outputText value="#{item.customerId}"/>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="Name"/>
              </f:facet>
              <h:outputText value="#{item.name}"/>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="City"/>
              </f:facet>
              <h:outputText value="#{item.city}"/>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="State"/>
              </f:facet>
              <h:outputText value="#{item.state}"/>
            </h:column>
          </h:dataTable>
        </h:panelGroup>
    </h:form>
```

8. The final step is to create the backing bean to manage the JSF page actions and values. Right click on the WAR module, and select **New > JSF managed bean**. If it is not listed, you can find it under **other > JavaServer Faces > JSF Managed Bean**



a. Call the bean **custController**.
b. Set the package to **JSF**

c. Change scope to session and **Click Finish.**

9. There are two attributes from the xhtml page which need to be included in the managed bean:
   a. The data model called **items**
   b. **selectState** - the filter input box.

   We also need to create a reference to the session bean.

   The code to declare these is as follows:

```
private DataModel items = null;
private String stateFilter="all";
@EJB
private session.CustomerFacade ejbFacade;
```

   Add set and get methods for each of the attributes, using insert code.

10. Change the getItems() method to call either findAll() or findByState()depending on what is selected in the filter as follows:

```
public DataModel getItems() {

if (getStateFilter().equals("all")) {
items = new ListDataModel(getEjbFacade().findAll());
    } else {
items = new
ListDataModel(getEjbFacade().findByState(getStateFilter()));
    }
return items;
 }
```

11. The is one more method to add. The filter command buttons needs to call an action that returns a string indicating what page to display. This action must also update the data model. The command button is currently calling **updateItems()**, so this method must be added to custController as follows:

```
public String updateItems() {
      items = getItems();     //update the data model
      return "index";         //return acstring indicating what page to
display
   }
```

The application is now ready to run.