



Profundización en Arquitectura de Software - Universidad de Antioquia

ARQUITECTURA DE SOFTWARE

LABORATORIO NRO 2: Introducción a JSF - Desarrollo de una aplicación Empresarial con JavaEE

Introducción

En este ejercicio se desarrollará una aplicación empresarial java que permitirá gestionar la información de una base de datos de clientes. El sistema validará además mediante un captcha el registro del formulario para realizar actualizaciones. La aplicación utilizará agrupamiento de conexiones (Pool de Conexiones), recurso Java Database Connectivity (JDBC), anotaciones de Java Persistence API (JPA), Enterprise JavaBeans (EJB), Unidad de Persistencia, Bean Gestionados (Managed Bean), JavaServer Faces (JSF) y la librería de componentes para JSF: PrimeFaces. Además, la gestión de las dependencias de las librerías y la construcción (builds) del proyecto serán a través de la herramienta Apache Maven.

I. Objetivo General

Desarrollar una aplicación web con JEE y Apache Maven para facilitar la gestión de dependencias y realizar la automatización de los builds del proyecto.

II. Objetivos Específicos

- Comprender el uso de Backing Beans para apoyar las transacciones complejas que se solicitan desde la vista.
- Aplicar y asimilar las funciones que ofrece el IDE NetBeans para la generación de componentes gráficos a partir de Entity Beans.
- Conocer y comprender más anotaciones JPA para simplificar la persistencia y el mapeo de una Base de Datos objeto-relacional.
- Entender el funcionamiento y configuración de Apache Maven y su importancia dentro del desarrollo del proyecto.
- Utilizar Bean Gestionados para la lógica de la aplicación y JSF para la vista del cliente.
- Comprender el manejo de JSF para la creación de una aplicación web.
- Conocer y utilizar las definiciones de la librería de componentes PrimeFaces que nos permiten representar componentes gráficos de una manera más intuitiva y simple.
- Conocer el funcionamiento y diferencias de las anotaciones de tipo Scoped que provee PrimeFaces para representar sus Managed Beans.
- Aplicar una arquitectura básica de 3 o N capas para el desarrollo de una aplicación web.
- Continuar aplicando y entender el uso de los patrones de diseño ORM, DTO y DAO.

III. Herramientas Empleadas

- A. NetBeans 8.2 o superior .
- B. Web Browser (Firefox, Chrome).
- C. Java EE 7 Web.
- D. JavaServer Faces 2.2.
- E. PrimeFaces 6.1.
- F. Apache Maven
- G. Enterprise JavaBeans.
- H. GlassFish Server 5.
- I. Mysql 5.2.
- J. API JPA.

IV. Arquitectura

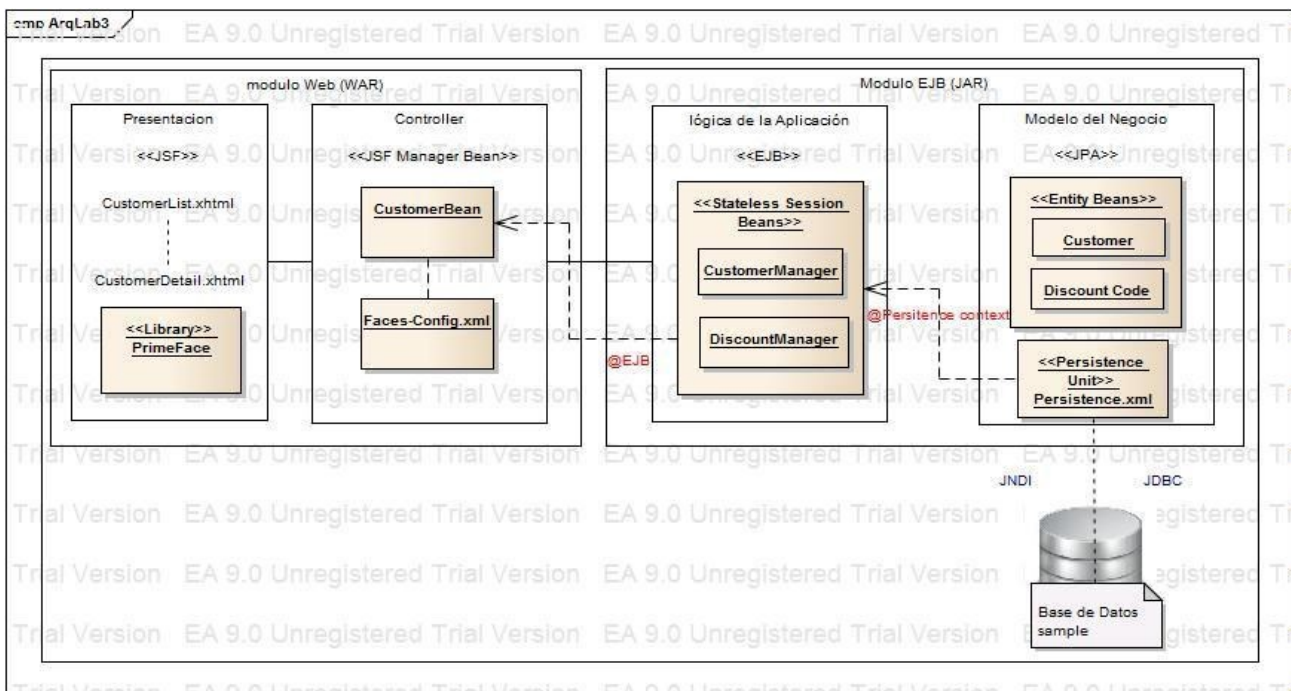


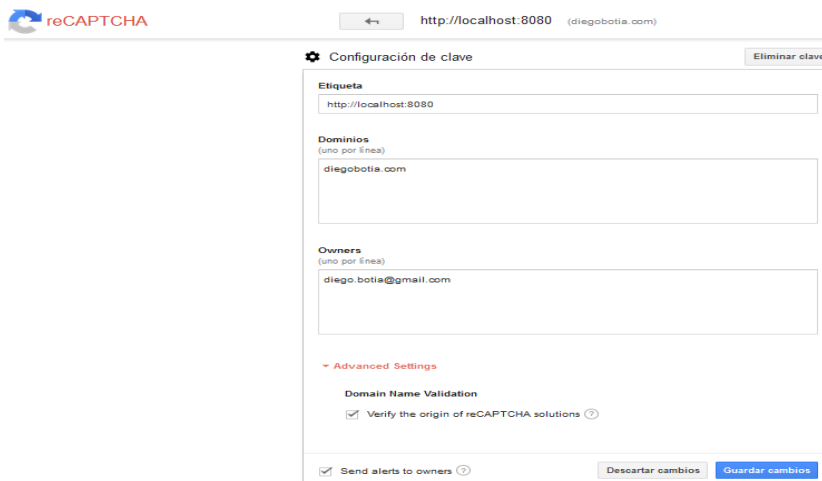
Figura 1: Arquitectura de la aplicación.

Al hacer uso de JavaServer Faces para desarrollar esta aplicación se adopta una arquitectura MVC. La **capa de Presentación** (formularios XHTML manipulados por Java, AJAX y PrimeFaces) de la aplicación, a la cual los **clientes accederán a través de sus navegadores**, La **capa controlador** que será la encargada de manejar, por medio de JSF Managed Beans o Backing Beans, las peticiones que haga el usuario a través de la capa web y la **capa de lógica del negocio** contendrá los métodos principales y se enlazarán a la **capa del modelo del dominio** a través de llamadas con el API JPA.

Procedimiento

Creando el Recaptcha

Primero, necesitamos una clave de API, por lo que vamos a <https://www.google.com/recaptcha/admin>. Para conseguir acceso a esta página necesitaremos estar conectados a una cuenta de Google. Pedirán registrar su website, así que le damos un nombre apropiado, ahora lista los dominios (por ejemplo diego.com donde el reCAPTCHA será usado. Los subdominios (como webdesign.diego.com y code.diego.com) son automáticamente tenidos en cuenta.



Con eso hecho nos darán una clave pública y su clave secreta:

Site key

Use this in the HTML code your site serves to users.

6LcePAATAAAAAGPRWgx90814DTjgt5sXnNbV5WaW

Secret key

Use this for communication between your site and Google. Be sure to keep it a secret.

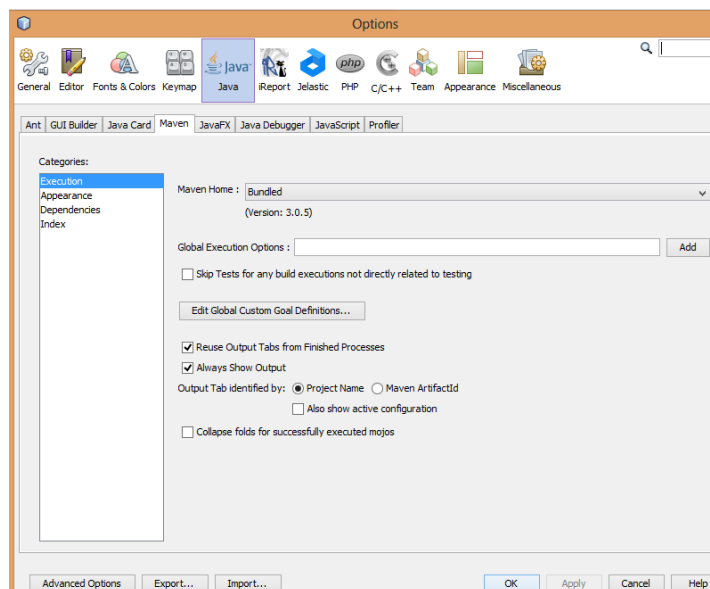
6LcePAATAAAAABjXaTsy7gwcbnbaF5XgJKwjSNwT

Para más detalles sobre configurar el reCAPTCHA podemos ir a <http://developers.google.com/recaptcha>.

Introducción a Maven

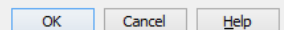
Muchos desarrolladores encuentran en Maven una herramienta que provee herramientas para la automatización de la gestión de dependencias y la construcción (builds) de proyectos. Netbeans da la facilidad para crear y administrar proyectos con Maven de una forma integrada y sencilla.

Netbeans embebe una copia de Maven para usar y crear proyectos. La versión actual de netbeans (8.2) integra la versión 3.0.5 de Maven. Puede ver la configuración general haciendo click en el menú **Tools → Options → Separador Java y pestaña Maven** (Ver la siguiente figura).



Después de que Ud cree el proyecto en Netbeans, automáticamente se abrirá el explorador de proyectos, y la lista de nodos disponibles del proyecto, los cuales dependerán del tipo de aplicación creada. Para la mayoría de los proyectos con Maven se crean arquetipos (archetypes), los cuales serán configurados en un archivo único llamado pom.xml. En el editor de POM provisto por netbeans podrá ver un gráfico de todos los artefactos usados por el proyecto Maven, incluyendo cada una de sus dependencias.

La línea de comandos no es amigable para el usuario, pero afortunadamente Netbeans permite el manejo de estos objetivos de una forma más sencilla. Para esto haga click derecho sobre el proyecto luego selección Properties. En el cuadro de dialogo que aparece haga click en el nodo Actions y vea o edite los objetivos de Maven (Por ejemplo Build Project, Clean Project, Test Project, y así sucesivamente).

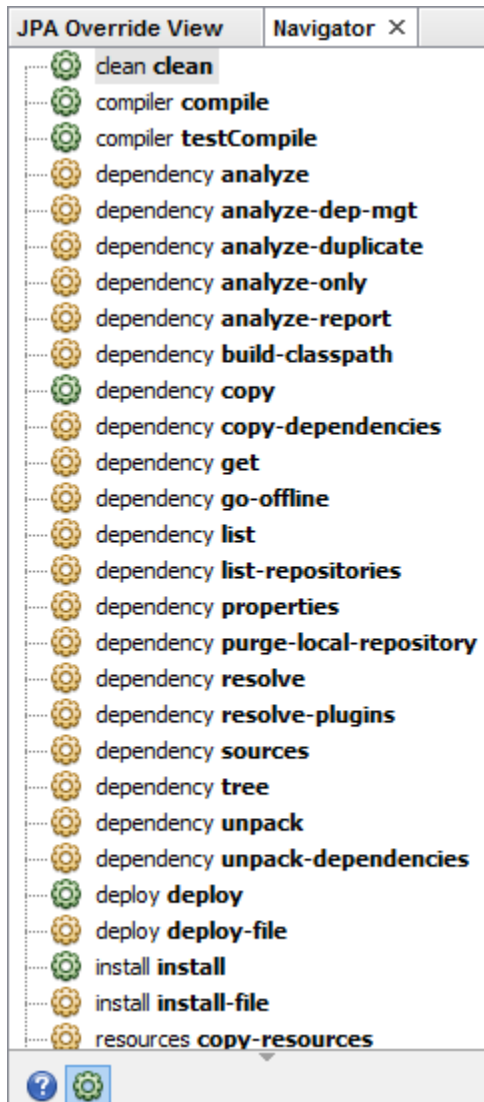


Profundización en Arquitectura de Software - Universidad de Antioquia

Estos objetivos son mapeados en las opciones de construcción estándar de Netbeans. Por ejemplo al hacer click derecho sobre el proyecto y seleccionar Build donde se ejecutará el objetivo de Maven install.

Invocando los objetivos en Maven

Dentro del explorador revise la ventana llamada Navigator que presenta una lista de los objetivos más comunes que son desplegados. Al hacer un doble click en cualquiera de estos objetivos se ejecutara directamente la instrucción en el proyecto actual. Puede hacer click en los iconos inferiores Show help goals y Show lifecycle bound goals para determinar el conjunto de objetivos que se presentan en el explorador.

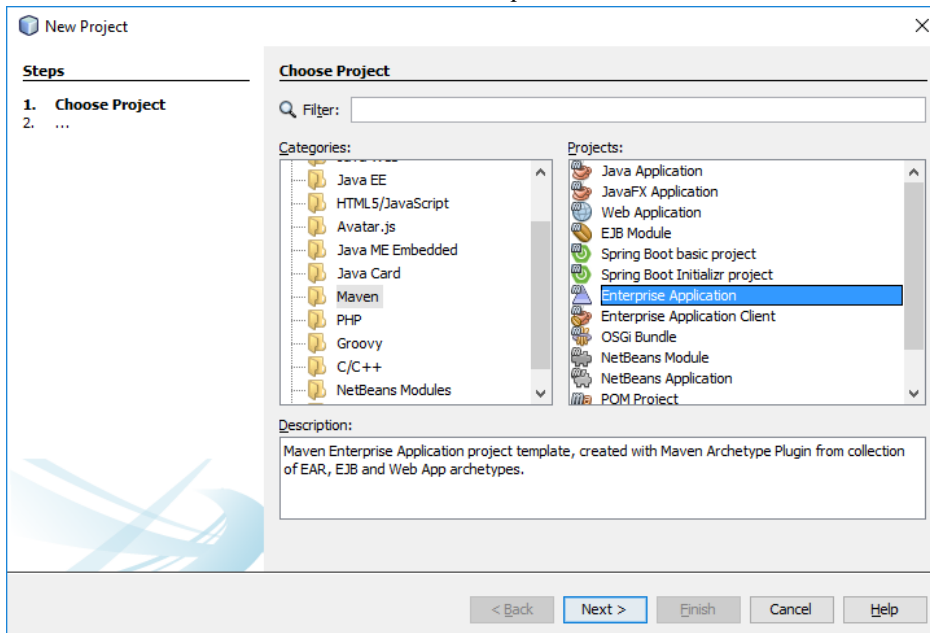


Creación de un proyecto tipo "Enterprise Application"

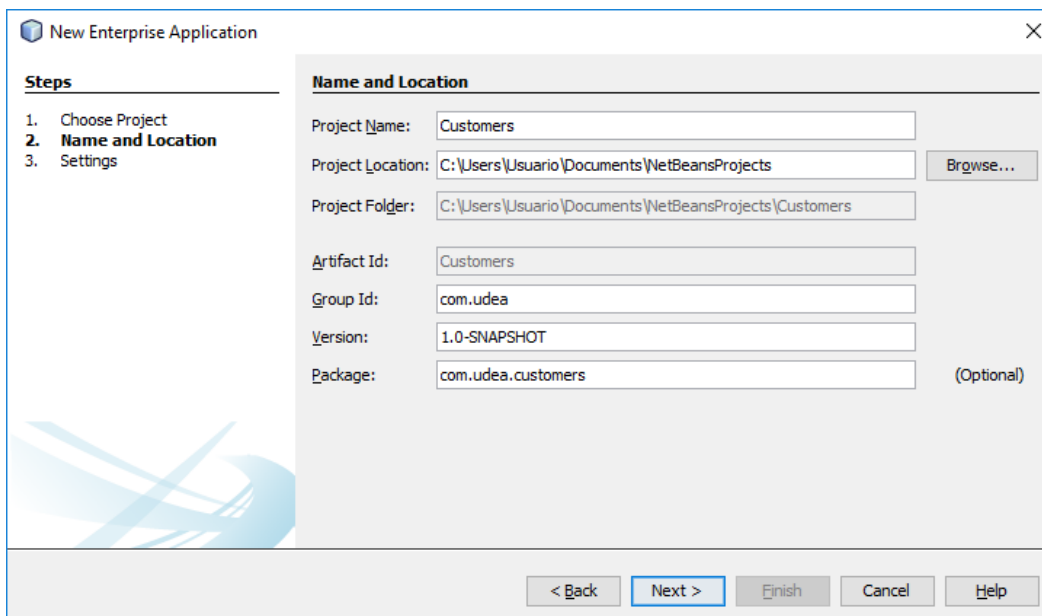
Es una clase especial de proyecto hecho en módulos diferentes: módulos EJB, módulos web, etc. Todos estos subproyectos son agrupados en un solo proyecto tipo empresarial, que se comporta como un solo contenedor. En Java EE se genera un archivo .ear (Enterprise ARchive), que permite hacer el despliegue de la aplicación más fácil.

Para crearlo en Netbeans vaya al menú File/New Project elegir la categoría Maven y elegir el tipo de proyecto como "Enterprise Application".

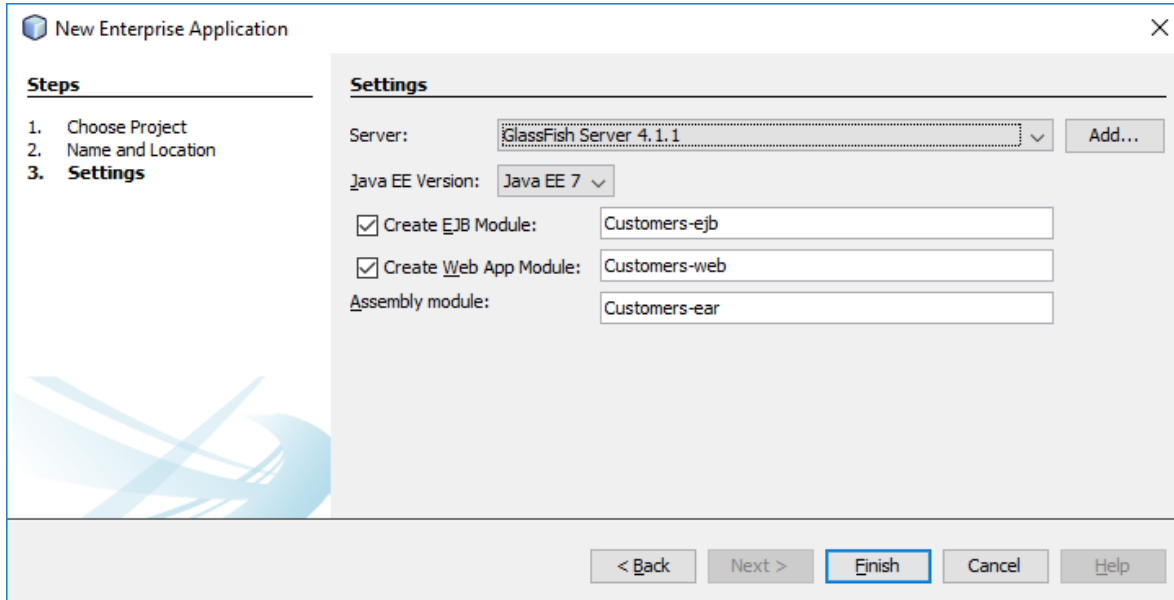
Profundización en Arquitectura de Software - Universidad de Antioquia



Haga click en "Next", y de un nombre al proyecto. Para este proyecto se utilizó el nombre Customers, como se muestra en la imagen. Ingrese también el Group Id, la versión de la aplicación que desee y el paquete principal por ejemplo **com.udea.customers**.



En la siguiente ventana, se debe elegir el servidor de aplicaciones, la versión del Java EE, etc.. Además debe dejar activado los distintos módulos que conformarán el proyecto.



New Enterprise Application

Steps

1. Choose Project
2. Name and Location
3. **Settings**

Settings

Server:

Java EE Version:

☒ Create EJB Module:

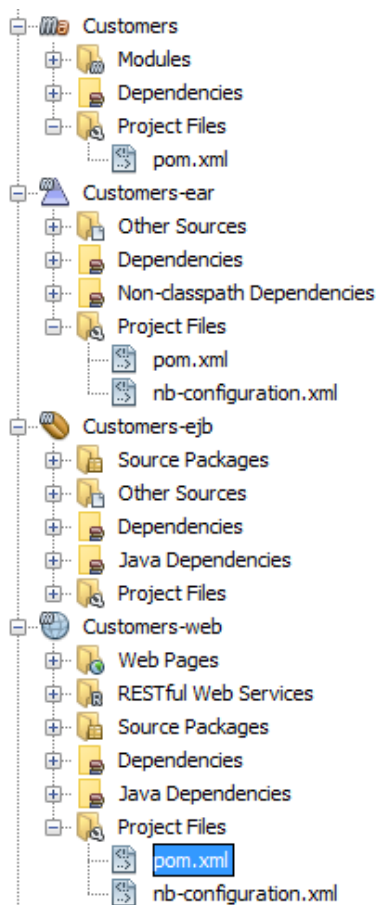
☒ Create Web App Module:

Assembly module:

< Back Next > **Finish** Cancel Help

Haga click en finalizar. Normalmente se verán 3 proyectos agregados a Netbeans:

1. Un proyecto con un triángulo azul, que sirve como contenedor para los dos proyectos siguientes, este es el proyecto .ear.
2. Un proyecto con un icono en forma de java bean (un grano de café), es donde se llevará a cabo la parte de la lógica de negocio de la aplicación, en este caso EJBs y Entity class.
3. Un proyecto con un icono en forma de mundo, este es el proyecto "Web" que contendrá los JSF managed beans, las páginas JSF, código javascript y los CSS.



Profundización en Arquitectura de Software - Universidad de Antioquia

Ahora se debe agregar las dependencias de cada una de las librerías a emplear en la aplicación por lo tanto abra el archivo **pom.xml** (Ubicado en el módulo Web) y agregue las siguientes dependencias

```
<dependency>
  <groupId>net.sf.barcode4j</groupId>
  <artifactId>barcode4j-light</artifactId>
  <version>2.0</version>
</dependency>

<dependency>
  <groupId>javax.enterprise</groupId>
  <artifactId>cdi-api</artifactId>
  <version>1.2</version>
</dependency>

<dependency>
  <groupId>commons-io</groupId>
  <artifactId>commons-io</artifactId>
  <version>2.0.1</version>
</dependency>

<dependency>
  <groupId>org.primefaces</groupId>
  <artifactId>primefaces</artifactId>
  <version>6.1</version>
</dependency>

<dependency>
  <groupId>net.glxn</groupId>
  <artifactId>qrgen</artifactId>
  <version>1.4</version>
</dependency>

<dependency>
  <groupId>com.google.zxing</groupId>
  <artifactId>core</artifactId>
  <version>2.0</version>
</dependency>

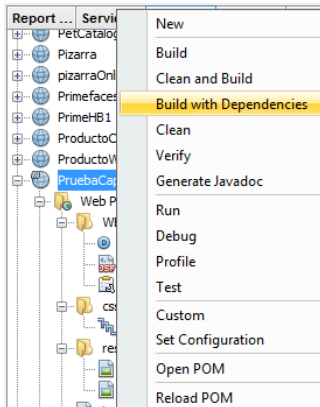
<dependency>
  <groupId>com.google.zxing</groupId>
  <artifactId>javase</artifactId>
  <version>2.2</version>
</dependency>

<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.5</version>
</dependency>

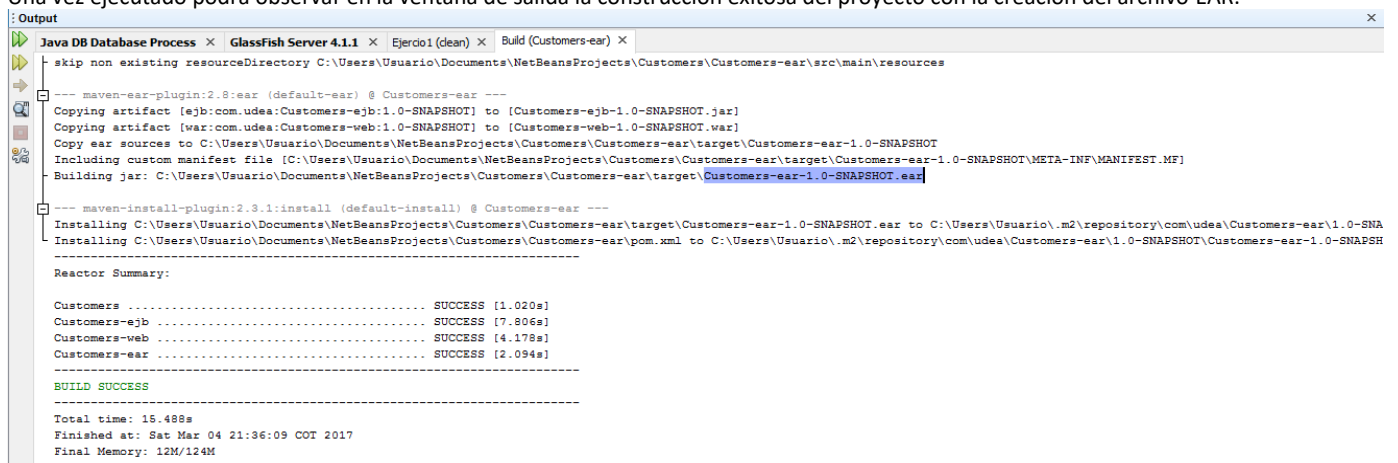
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.23</version>
</dependency>
```

Para que se agreguen las dependencias al proyecto desde el repositorio principal de Maven, debe hacer click derecho sobre el proyecto y seleccionar la opción Build with Dependencies.

Profundización en Arquitectura de Software - Universidad de Antioquia



Una vez ejecutado podrá observar en la ventana de salida la construcción exitosa del proyecto con la creación del archivo EAR.



Ahora edite el descriptor de despliegue (Web.xml) y cargue la llave pública y privada asociado al servicio Rcaptcha de Google. Recuerde que la etiqueta rcaptcha hace parte del conjunto de etiquetas definidas por primefaces.

Web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>faces/index.xhtml</welcome-file>
  </welcome-file-list>

```

```
<context-param>
  <param-name>primefaces.PUBLIC_CAPTCHA_KEY</param-name>
  <param-value>6LcWegoTAAAAAMCX4icYQrX1</param-value>
</context-param>

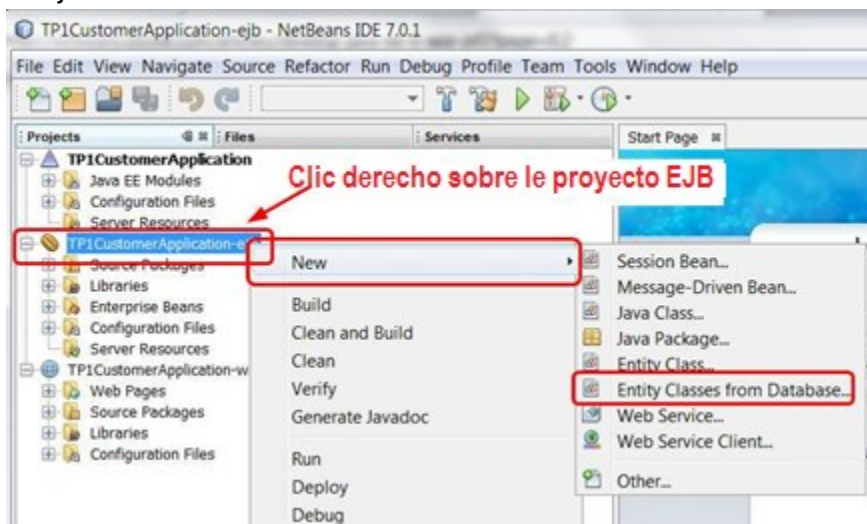
<context-param>
  <param-name>primefaces.PRIVATE_CAPTCHA_KEY</param-name>
  <param-value>6LcWegoTAAAAANoEcr7rG6n3q</param-value>
</context-param>

<listener>
  <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
</web-app>
```

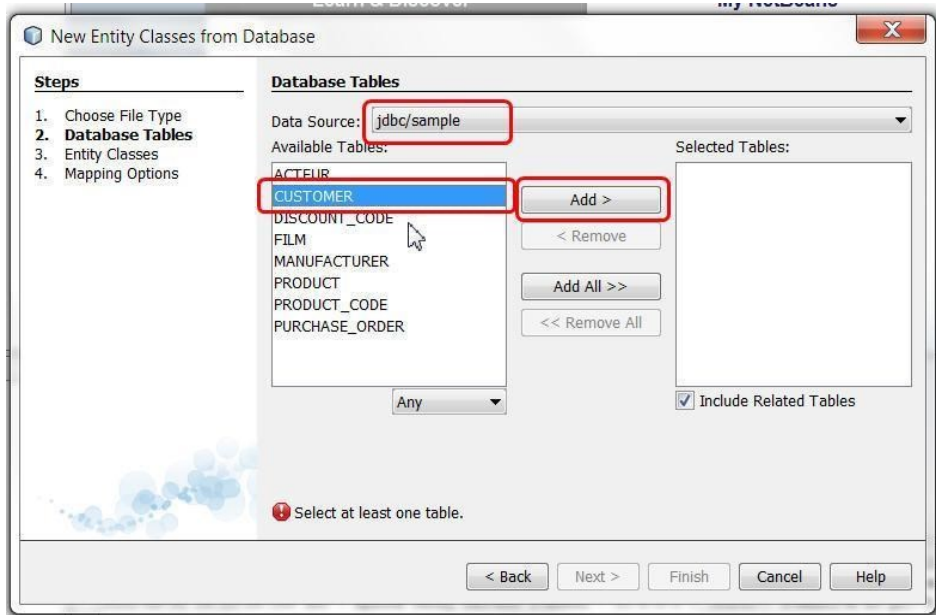
Crear una Entity class desde una Base de Datos existente

Java EE contiene un framework llamado JPA (Java Persistence API), en este framework las clases especiales llamadas "Entity class" corresponden a tablas en una base de datos. El mapeo entre estas clases y tablas se realiza usando herramientas ORM externas como Hibernate o EclipseLink, estas herramientas proporcionan algunos medios para generar automáticamente las Entity class desde la tabla modelo.

En el módulo EJB haga click derecho en el nombre del proyecto y luego seleccione **New → Other → Categoría Persistence → Entity class from database**.

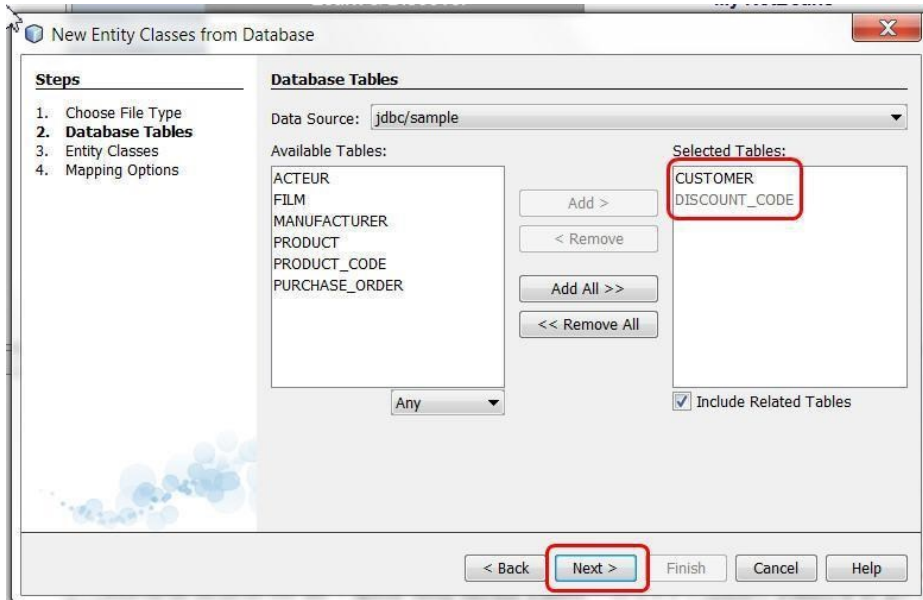


Se pedirá el nombre del JNDI (Java Naming and Directory Interface) de la base de datos. Elegir **"jdbc/sample"**, que es una base de datos que viene en el motor *JavaDB* incluida en Netbeans. Por lo tanto no hay necesidad de ejecutar un servidor de Base de Datos, ni crear a mano alguna base de datos, tablas, etc. Normalmente después de que se elige el nombre de la Base de Datos, en la lista de la izquierda aparecerá mostrando las tablas disponibles:



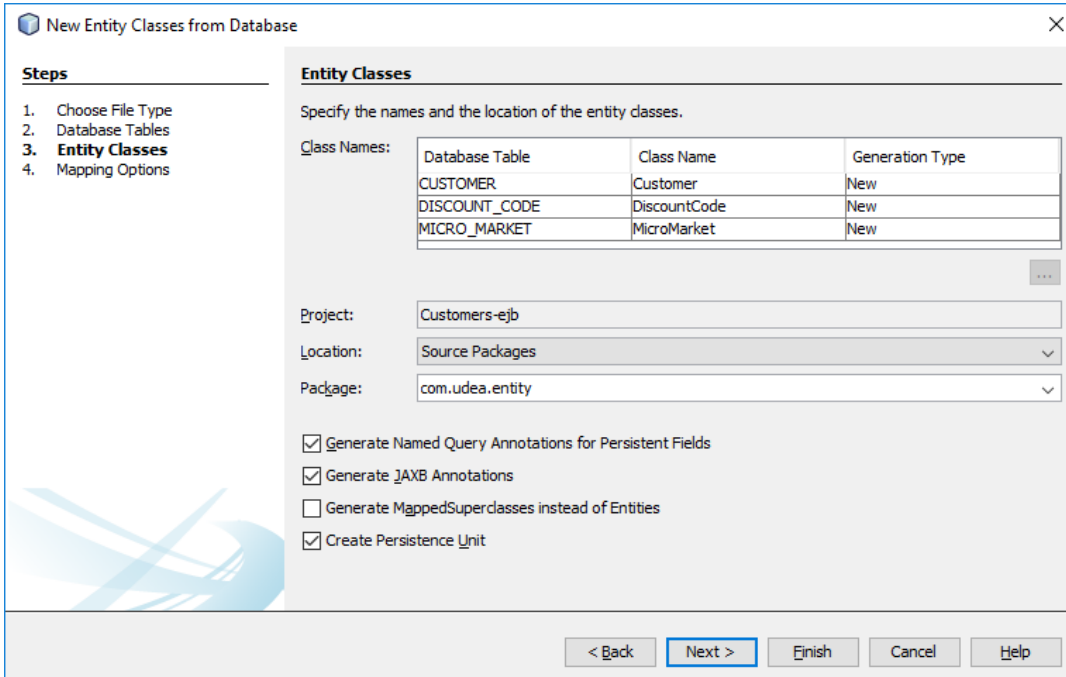
Elija la tabla "CUSTOMER" y de click en el botón "add", y observe que se agregarán tres tablas: CUSTOMER y DISCOUNT_CODE y MICROMARKET ya que hay una relación entre las tres tablas.

Nota: Si usted lo desea puede seleccionar solo las tablas que quiera relacionar haciendo click en la opción **Include Related Tables**.



Haga click en "Next", ahora se puede cambiar los nombres de las *Java entity classes* que se generarán, es recomendable dejar todos los valores por defecto. De un nombre al paquete y verifique que se encuentre seleccionado la opción de crear la unidad de persistencia como se muestra en la figura.

Profundización en Arquitectura de Software - Universidad de Antioquia



New Entity Classes from Database

Steps

1. Choose File Type
2. Database Tables
3. **Entity Classes**
4. Mapping Options

Entity Classes

Specify the names and the location of the entity classes.

Class Names:

Database Table	Class Name	Generation Type
CUSTOMER	Customer	New
DISCOUNT_CODE	DiscountCode	New
MICRO_MARKET	MicroMarket	New

Project: Customers-ejb

Location: Source Packages

Package: com.udea.entity

☒ Generate Named Query Annotations for Persistent Fields

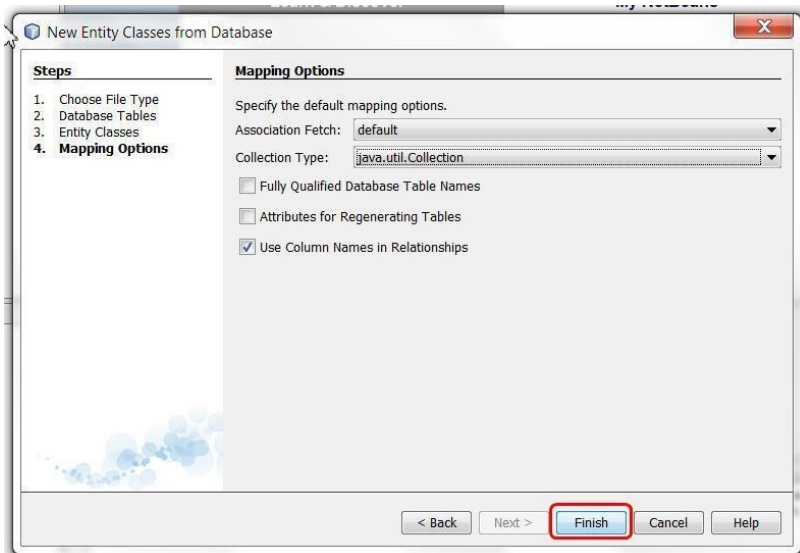
☒ Generate JAXB Annotations

☐ Generate MappedSuperclasses instead of Entities

☒ Create Persistence Unit

< Back Next > Finish Cancel Help

Haga click en "Next". Dejar todos los demás valores por defecto.



New Entity Classes from Database

Steps

1. Choose File Type
2. Database Tables
3. Entity Classes
4. **Mapping Options**

Mapping Options

Specify the default mapping options.

Association Fetch: default

Collection Type: java.util.Collection

☐ Fully Qualified Database Table Names

☐ Attributes for Regenerating Tables

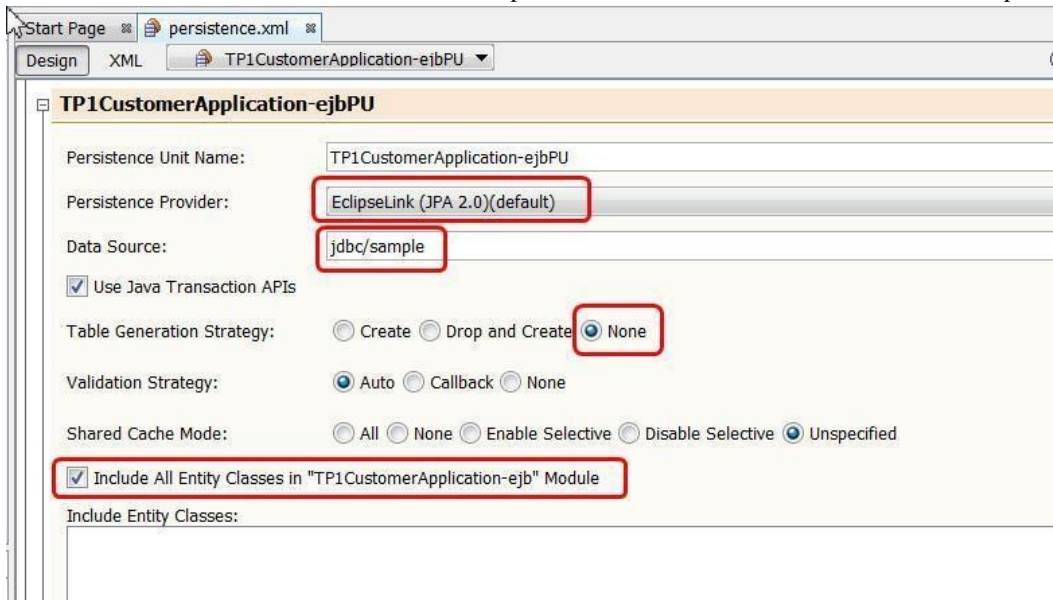
☒ Use Column Names in Relationships

< Back Next > Finish Cancel Help

Haga click en "Finish". Esto generará tres clases Java en un nuevo paquete llamado "com.udea.entity"

Además, un archivo llamado **persistence.xml** que aparece dentro de la carpeta "src/main/resources", el cual corresponde a la unidad de persistencia del proyecto. La unidad de persistencia es la responsable de la comunicación y transaccionalidad hacia la base de datos, en este objeto se generará todo el código SQL que está por debajo.

Si abre con el editor el archivo persistence.xml, aparece la siguiente ventana:



Start Page persistence.xml

Design XML TP1CustomerApplication-ejbPU

TP1CustomerApplication-ejbPU

Persistence Unit Name: TP1CustomerApplication-ejbPU

Persistence Provider: EclipseLink (JPA 2.0)(default)

Data Source: jdbc/sample

☒ Use Java Transaction APIs

Table Generation Strategy: ☐ Create ☐ Drop and Create ☒ None

Validation Strategy: ☒ Auto ☐ Callback ☐ None

Shared Cache Mode: ☐ All ☐ None ☐ Enable Selective ☐ Disable Selective ☒ Unspecified

☒ Include All Entity Classes in "TP1CustomerApplication-ejb" Module

Include Entity Classes:

En este archivo se encuentra que:

1. No se permite crear de nuevo las tablas (None)
2. Se usará el API de Java Transaction que permitirá hacer las transacciones con JDBC hacia la Base de Datos.
3. Esta unidad de persistencia trabaja con la base de datos **jdbc/sample**, usa EclipseLink como proveedor de persistencia el cual mapeará todas las entity class desde las tablas de la BD mediante el uso de un ORM (Mapeo Objeto Relacional).

También en la clase *customer.java* se deben agregar/verificar las siguientes líneas de código:

```
public String getDiscount() {return this.discountCode.getDiscountCode();}
public void setDiscount(String code) {this.discountCode=new DiscountCode(code);}

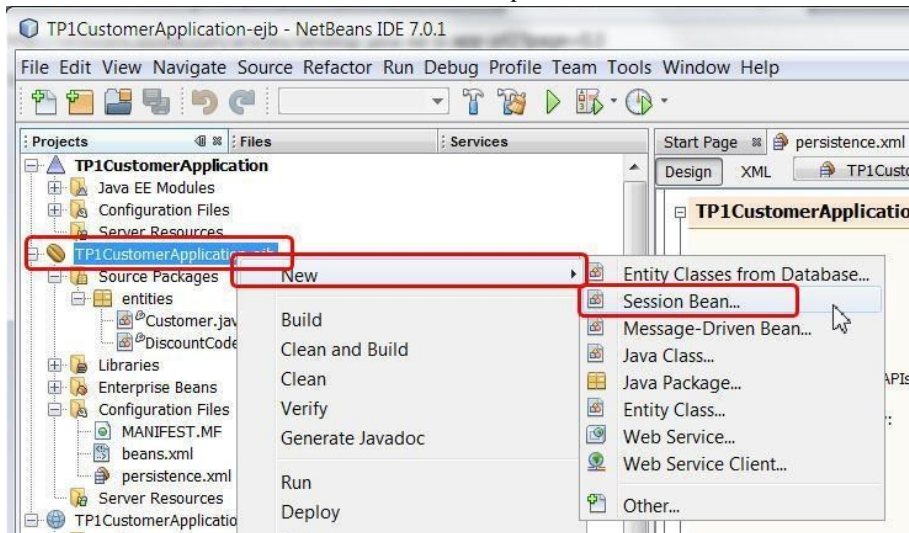
public String getZ() {return this.zip.getZipCode();}
public void setZ(String zip) {this.zip=new MicroMarket(zip);}
```

Estas líneas sirven para relacionar la clave foránea de la tabla CUSTOMER con la clave primaria de la tabla DISCOUNT_CODE y MICROMARKET.

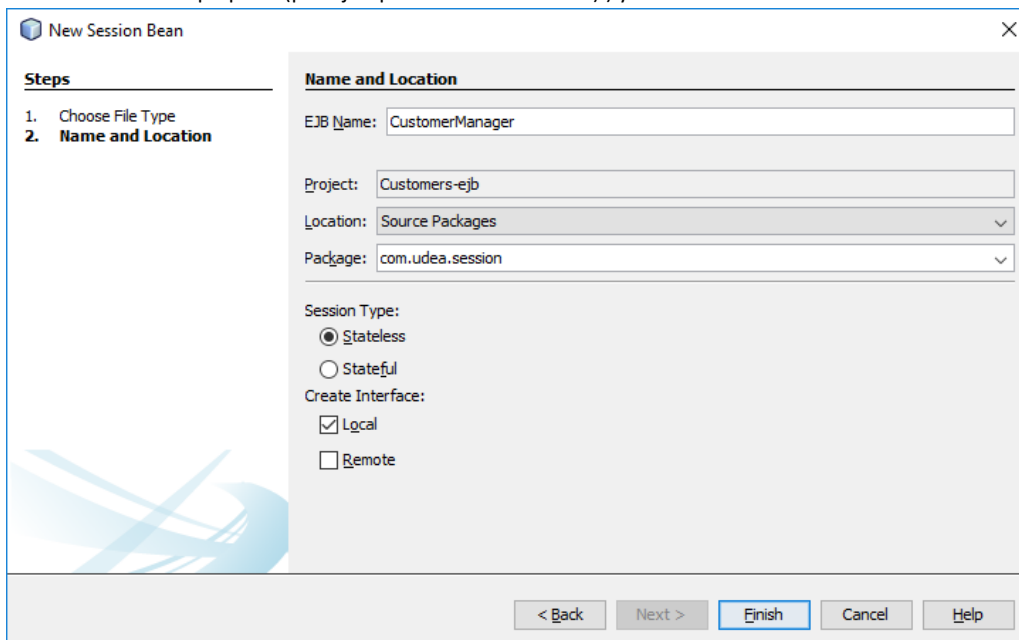
Creación de un Cliente Facade (Fachada)

Se añadirá al proyecto una clase DAO que expondrá sus métodos a través de un Facade, que se hará cargo de todas las operaciones CRUD sobre las entidades del cliente (Customer Entity), si se quiere insertar, actualizar, eliminar o buscar entidades en la base de datos **jdbc/sample** tendrá que utilizar esta clase. Los métodos del negocio serán implementados como una clase EJB de tipo *Stateless Session Bean*.

Haga click derecho en el proyecto EJB y seleccione **New → Other → Categoría Enterprise Java Beans → Session Bean**.



Dar un nombre al DAO, en general se podría llamar *CustomerFacade*, *CustomerDAO*, *CustomerManager* o *CustomerHandler* , luego darle un nombre al paquete (por ejemplo **com.udea.session**)) y si lo desea crear una interfaz Local.

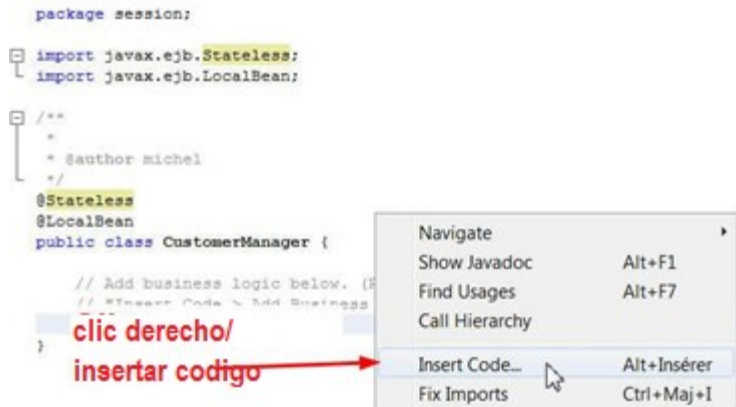


Nota: Repita el mismo proceso para crear los **Session Beans** llamados **DiscountCodeManager** y **ZipCodeManager**.

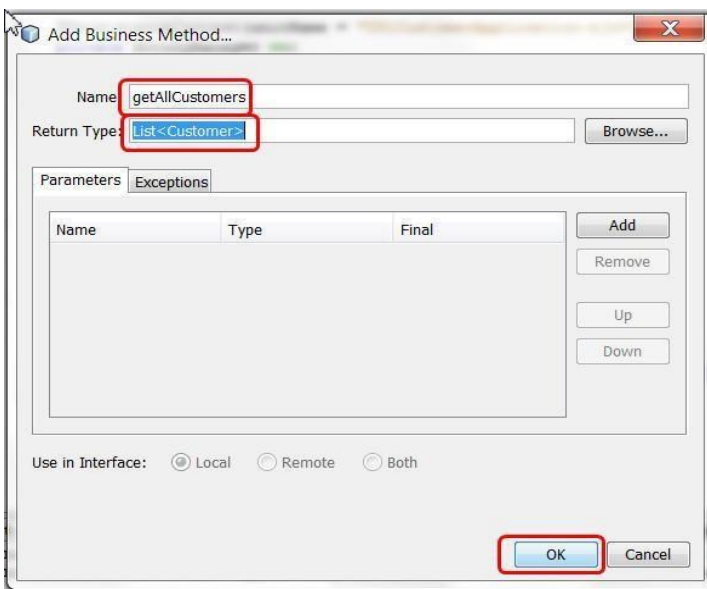
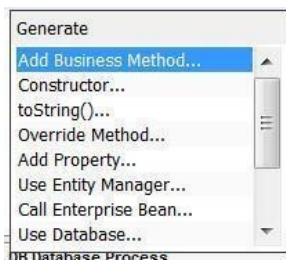
Añadir algunos métodos de negocio en el Facade

Haga doble click en la clase "*CustomerManager.java*" de manera que muestre el código fuente en el editor. Haga click derecho en algún lugar de la clase, y seleccione *insert code*.

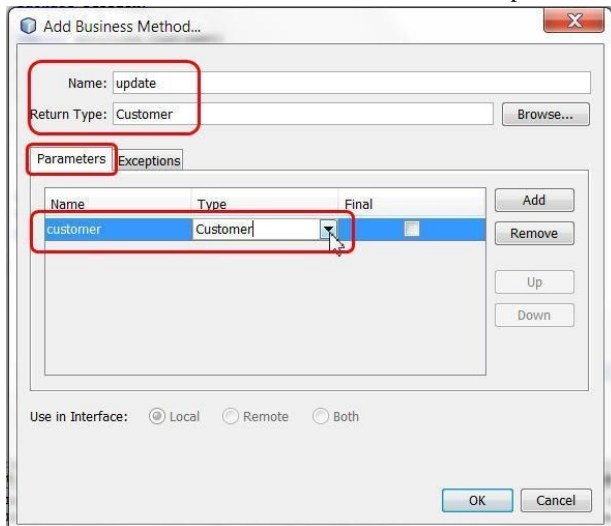
Profundización en Arquitectura de Software - Universidad de Antioquia



Se observará un menú, se seleccionamos "**Add Business Method**" y luego se agrega el nombre del método **getAllCustomers()** que devolverá una lista de clientes (Customer)



En la ventana del código debe añadir las importaciones que hagan falta (click en el icono amarillo pequeño o corregir importaciones, o dar Alt-Shift-I). Añada también el método **update (Customer c)** que permitirá realizar la actualización de un cliente.



Se debe obtener el siguiente código:

```
01. package session;
02.
03. import entities.Customer;
04. import java.util.Collection;
05. import javax.ejb.Stateless;
06. import javax.ejb.LocalBean;
07.
08. @Stateless
09. @LocalBean
10. public class CustomerManager {
11.
12.     public List<Customer> getAllCustomers() {
13.         return null;
14.     }
15.
16.     public Customer update(Customer customer) {
17.         return null;
18.     }
19.
20. }
```

Es posible escribir estos métodos a mano, pero Netbeans facilita el trabajo.

Observe que también aparecen los métodos expuestos en la Interface de la clase (**CustomerManagerLocal**) como se observa a continuación:

```
+ ...5 lines
package com.udea.session;

import java.util.List;
import javax.ejb.Local;
import com.udea.entity.Customer;

+ /**...4 lines */
@Local
public interface CustomerManagerLocal {

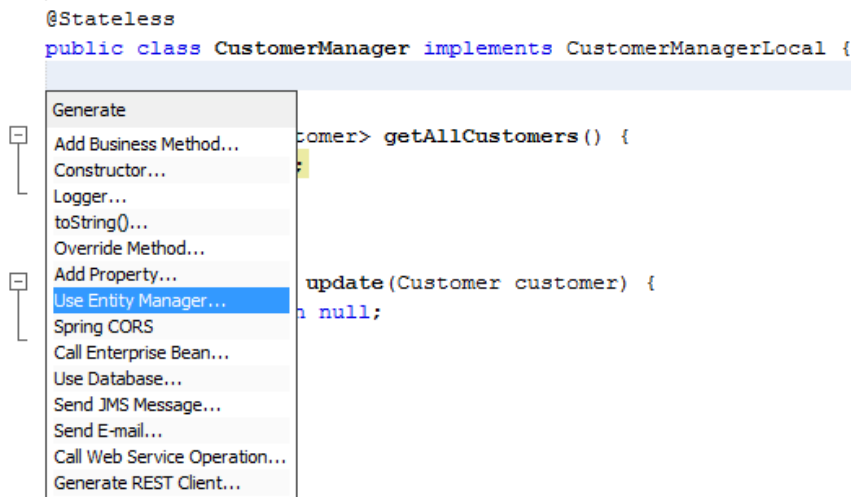
    public List<Customer> getAllCustomers();
    public Customer update(Customer customer);

}
```

Ahora se añadirá un el contexto de persistencia a la aplicación, utilizando una anotación que inyectará una variable de tipo *EntityManager*. Esta variable corresponde a la unidad de persistencia definida en el archivo **persistence.xml**, en este caso se utilizará

Profundización en Arquitectura de Software - Universidad de Antioquia

la variable para actualizar y realizar consultas a la Base de Datos. Haga click derecho sobre el editor de código luego seleccione **insert code** → **add entity manager**



Se agregará automáticamente el siguiente código:

```
01. @PersistenceContext(unitName = "TP1CustomerApplication-ejbPU") // the name can be different in your case
02. private EntityManager em;
03. public void persist(Object object) {
04.     em.persist(object);
05. }
```

La variable *em* no tiene que ser inicializada, ya que será "inyectada". El nombre justo después del parámetro *"unitName ="*, está también definido en el archivo *persistence.xml* file.

Agregue ahora a cada método el código anexo:

```
@Stateless
public class CustomerManager implements CustomerManagerLocal {

    @PersistenceContext(unitName = "com.udea_Customers-ejb_ejb_1.0-SNAPSHOTPU")
    private EntityManager em;

    @Override
    public List<Customer> getAllCustomers() {
        Query query= em.createNamedQuery("Customer.findAll");
        return query.getResultList();
    }

    @Override
    public Customer update(Customer customer) {
        return em.merge(customer);
    }
}
```

En el método **getAllCustomers()**, se ejecuta una "consulta nombrada", cuyo nombre es **"Customer.findAll"** (esta consulta se encuentra al principio del archivo *Customer.java*), y es definido como una consulta JPQL definida como **select c from Customer c**. Se debe recordar que para este caso como se está trabajando con objetos se debe retornar una lista de clientes (*Customer*) y no tuplas.

En el Segundo método *update(Customer customer)*... se actualiza el contenido de la Base de Datos mediante el objeto cliente que se pasa como parámetro; El método *em.merge(customer)* hace precisamente eso: "toma este objeto en memoria, mira si existe en la Base de Datos, a continuación actualiza la columna en la BD. Los objetos tiene una clave primaria (en este caso, mirar el atributo *id* en la clase *Customer.java*).

Con el objetivo de manejar la clase **DiscountCode** se insertará un **session Bean** de tipo **Stateless** para ayudar a administrar las relaciones entre la tabla *CUSTOMER* y *DISCOUNT*.

Profundización en Arquitectura de Software - Universidad de Antioquia

Se deben añadir los mismos métodos del negocio (getDiscountCodes), al finalizar el proceso se espera que el **Entity manager** posea el siguiente código:

```
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
- import javax.persistence.Query;

] /**...4 lines */
@Stateless
public class DiscountCodeManager implements DiscountCodeManagerLocal {

    @PersistenceContext(unitName = "com.udea_Customers-ejb_ejb_1.0-SNAPSHOTPU")
    private EntityManager em;

    @Override
] public List<DiscountCode> getDiscountCodes() {
    Query query= em.createNamedQuery("DiscountCode.findAll");
    return query.getResultList();
- }
}
```

Al igual que en caso anterior la **clase ZipCodeManager** quedará así:

```
import com.udea.entity.MicroMarket;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

] /**...4 lines */
@Stateless
public class ZipCodeManager implements ZipCodeManagerLocal {

    @PersistenceContext(unitName = "com.udea_Customers-ejb_ejb_1.0-SNAPSHOTPU")
    private EntityManager em;

    @Override
] public List<MicroMarket> getZipCodes() {
    Query query= em.createNamedQuery("MicroMarket.findAll");
    return query.getResultList();
}

}
```

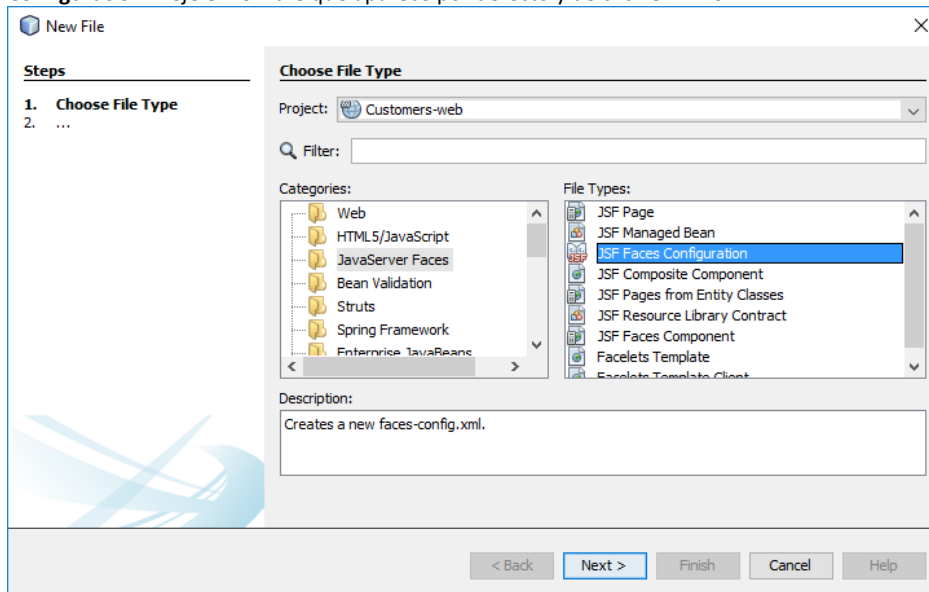
Creación del módulo Web

Para este caso se utilizará el framework JSF 2.2 junto con la librería Primefaces 5.2. Este soporte de dependencias ya se ha cargado previamente con Maven.

Creación del archivo faces-config.xml

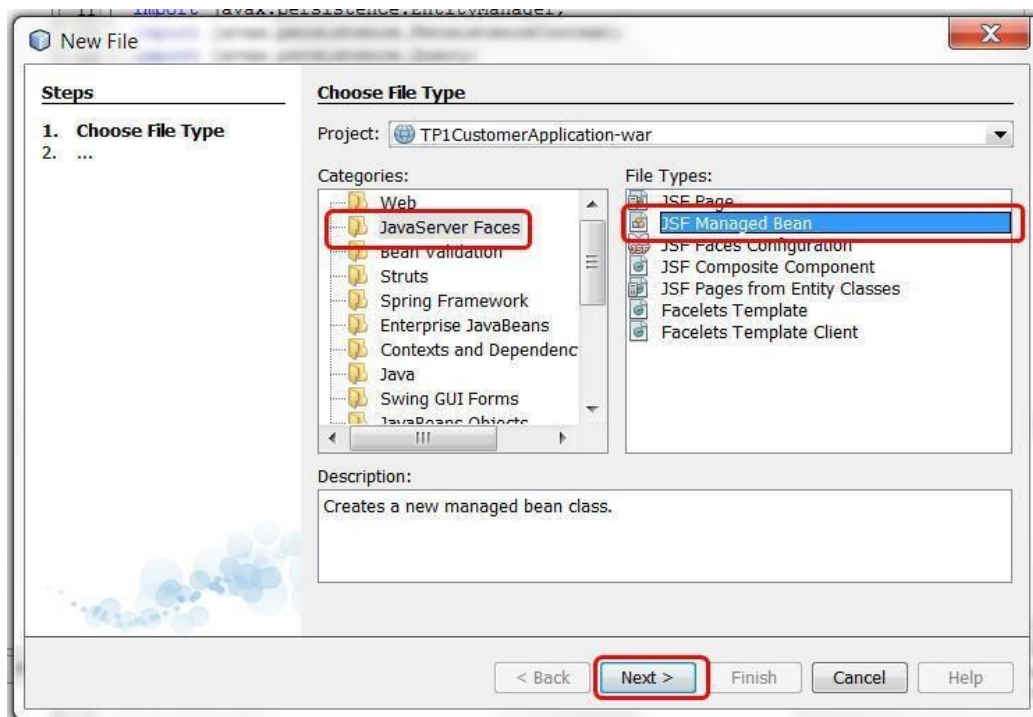
Para el modulo es necesario crear el archivo de configuración faces-config.xml, que permitirá registrar los controladores así como gestionar la navegabilidad entre las vistas.

Para crearlo haga click derecho sobre el modulo web y seleccione **New** → **Other** → categoría **JavaServer Faces** → **JSF Faces Configuration**. Deje el nombre que aparece por defecto y de click en Finish.



Creación de un JSF managed bean como un control

Las clases de este tipo permitirán actuar como los controladores de las vistas JSF (muy análogo a los Servlets con JSP vistos en el laboratorio anterior). Haga click derecho en el módulo web del proyecto y seleccione **New** → **Other** → Categoría **JavaServer Faces** → **JSF Managed Bean**



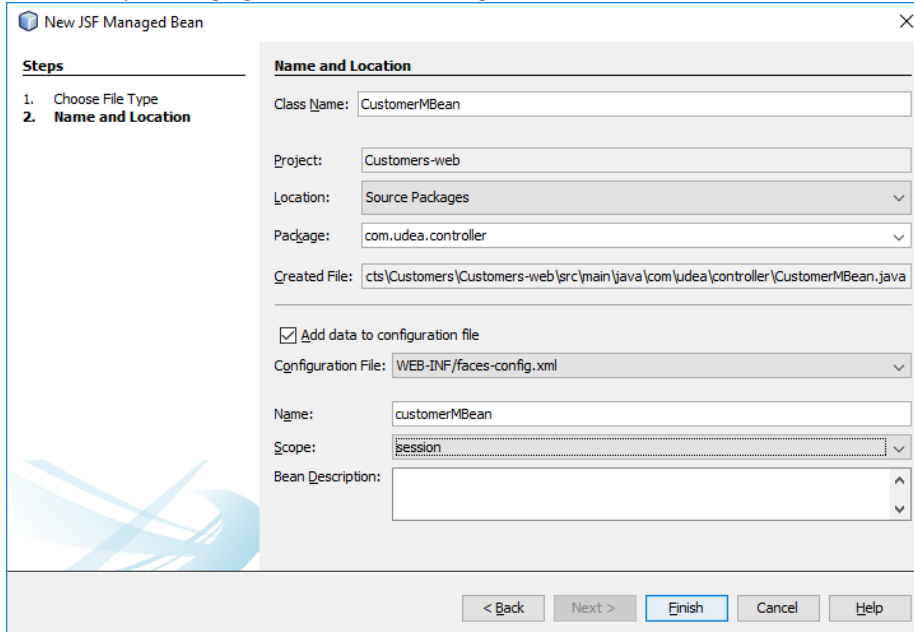
Colocar un nombre al *JSF bean*, que para este ejemplo recibe el nombre de **CustomerMBean** y será añadido al paquete llamado **com.udea.controller**. Además se eligió que el ámbito de la sesión HTTP (scope) será "session".

Nota: Debe investigar todos los tipos de scopes que soporta JSF. Más información en estos enlaces: <http://docs.oracle.com/javaee/6/tutorial/doc/girch.html> y http://puesenmiordenadorfunciona.blogspot.com.co/2011/10/jsf-20-managed-beans-iii_25.html

En este caso el ámbito **Session Scoped** significa que:

1. El bean durará lo que dure la sesión HTTP
2. Sus atributos/propiedades se almacenarán en la sesión HTTP, y se va a utilizar las propiedades de esta clase como modelo para la visualización y modificación en la página JSF.

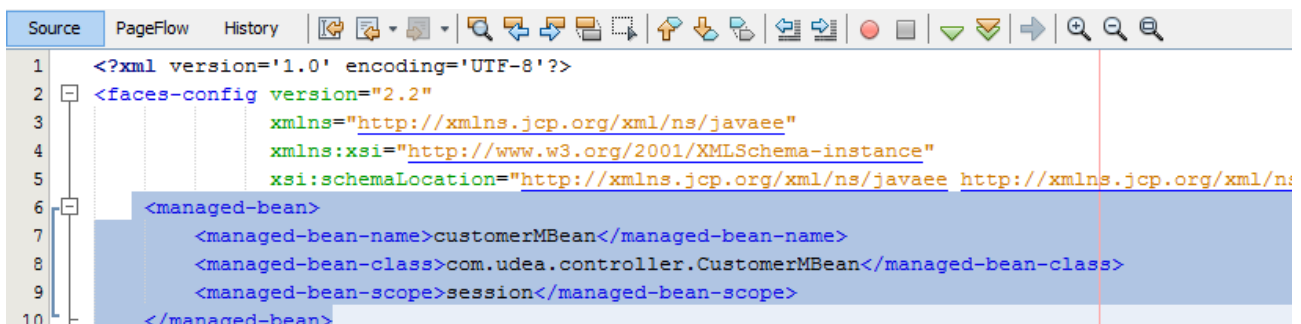
También es posible agregar el archivo **face-config.xml**, dando click en la casilla de habilitación **"Add data to configuration file"**.



Se debe tener un *CustomerMBean.java* que se parezca al siguiente:

```
01. package managedbeans;
02.
03. import javax.inject.Named;
04. import javax.enterprise.context.SessionScoped;
05. import java.io.Serializable;
06.
07. @Named(value = "customerMBean")
08. @SessionScoped
09. public class CustomerMBean implements Serializable {
10.
11.     public CustomerMBean() {
12.     }
13. }
```

Ahora puede observar dentro del archivo **faces-config.xml**, las siguientes etiquetas:



Funcionalidades del JSF Managed Bean

- Que esté listo para usar el *Customer Facade* que se escribió en el Proyecto EJB

Profundización en Arquitectura de Software - Universidad de Antioquia

- Añadir un atributo/propiedad del tipo Customer, que será usado cuando se edite o inserte un cliente en un formulario JSF.
- Añadir una lista de clientes que serán un modelo para mostrar en una tabla.
- Añadir algunos métodos controladores que se puedan llamar desde las páginas JSF.

Para llamar a un EJB por medio de una inyección, haga click derecho sobre el editor de código y seleccione *insert code* → *Call Enterprise Bean*. Seleccione todos los **facades** y observará su inserción por medio de la anotación **@EJB**.

```
package com.udea.controller;

import com.udea.entity.Customer;
import com.udea.entity.DiscountCode;
import com.udea.entity.MicroMarket;
import com.udea.session.DiscountCodeManagerLocal;
import com.udea.session.ZipCodeManagerLocal;
import java.io.Serializable;
import java.util.List;
import javax.ejb.EJB;
import javax.faces.model.SelectItem;

/**...4 lines */
public class CustomerMBean implements Serializable {

    @EJB
    private ZipCodeManagerLocal zipCodeManager;

    @EJB
    private DiscountCodeManagerLocal discountCodeManager;

    @EJB
    private com.udea.session.CustomerManagerLocal customerManager;
```

La anotación @EJB inyecta por ejemplo la variable *customerManager*, donde esta anotación actúa como un localizador de servicio y como una fábrica. Pero esta vez, el control se invierte, (lo llamamos "inversión de control" o IOC: el objeto correcto es construido por el contenedor EJB).

Añada ahora el siguiente código del JSF para completar la funcionalidad de las vistas:

```

22 public class CustomerMBean implements Serializable {
23
24     @EJB
25     private ZipCodeManagerLocal zipCodeManager;
26
27     @EJB
28     private DiscountCodeManagerLocal discountCodeManager;
29
30     @EJB
31     private com.udea.session.CustomerManagerLocal customerManager;
32     //Propiedades del modelo
33     private Customer customer; //para mostrar, actualizar o insertar en el formulario
34     private List<Customer> customers; //para visualizar en la tabla
35
36     public CustomerMBean() {}
37
38     //retorna una lista de clientes para mostrar en un datatable de JSF
39     public List<Customer> getCustomers(){
40         if((customers==null) || (customers.isEmpty()))
41             refresh();
42         return customers;
43     }
44     //Retorna el detalle de cada cliente en el formulario.
45     public Customer getDetails() {
46         return customer;
47     }
48     /**
49     * Action handler - llamado cuando en una fila de la tabla se haga click en ID
50     * @param customer
51     * @return
52     */
53     public String showDetails(Customer customer) {
54         this.customer = customer;
55         return "DETAILS"; // Permite enlazar a CustomerDetails.xml
56     }
57     /**
58     * Action handler - Actualiza el modelo Customer en la BD.
59     * Se llama cuando se presiona el boton update del formulario
60     * @return
61     */
62     public String update() {
63         System.out.println("###UPDATE###");
64         customer = customerManager.update(customer);
65         return "SAVED"; // Para el caso navegacional
66     }
67     /**
68     * Action handler - retorno hacia la vista de la lista de clientes
69     * @return
70     */
71     public String list() {
72         System.out.println("###LIST###");
73         return "LIST";
74     }
75     private void refresh() {
76         customers=customerManager.getAllCustomers();
77     }

```

```
//Para cargar correctamente los combobox del formulario Details
public javax.faces.model.SelectItem[] getDiscountCodes() {
    SelectItem[] options = null;
    List<DiscountCode> discountCodes = discountCodeManager.getDiscountCodes();
    if (discountCodes != null && discountCodes.size() > 0) {
        int i = 0;
        options = new SelectItem[discountCodes.size()];
        for (DiscountCode dc : discountCodes) {
            options[i++] = new SelectItem(dc.getDiscountCode(),
                dc.getDiscountCode() + " (" + dc.getRate() + "%)");
        }
    }
    return options;
}

public javax.faces.model.SelectItem[] getZipCodes() {
    SelectItem[] options = null;
    List<MicroMarket> zipCodes = zipCodeManager.getZipCodes();
    if (zipCodes != null && zipCodes.size() > 0) {
        int i = 0;
        options = new SelectItem[zipCodes.size()];
        for (MicroMarket dc : zipCodes) {
            options[i++] = new SelectItem(dc.getZipCode(), dc.getZipCode());
        }
    }
    return options;
}
}
```

Acá esta el código para copia

//Propiedades del modelo

private Customer customer; //para mostrar, actualizar o insertar en el formulario
private List<Customer> customers; //para visualizar en la tabla

public CustomerMBean() {}

//retorna una lista de clientes para mostrar en un datatable de JSF

```
public List<Customer> getCustomers(){
    if((customers==null) || (customers.isEmpty()))
        refresh();
    return customers;
}
```

//Retorna el detalle de cada cliente en el formulario.

```
public Customer getDetails() {
    return customer;
}
```

/**

* Action handler - llamado cuando en una fila de la tabla se haga click en ID

* @param customer

* @return

*/

```
public String showDetails(Customer customer) {
```

```
    this.customer = customer;
```

```
    return "DETAILS"; // Permite enlazar a CustomerDetails.xml
```

```
}
```

/**

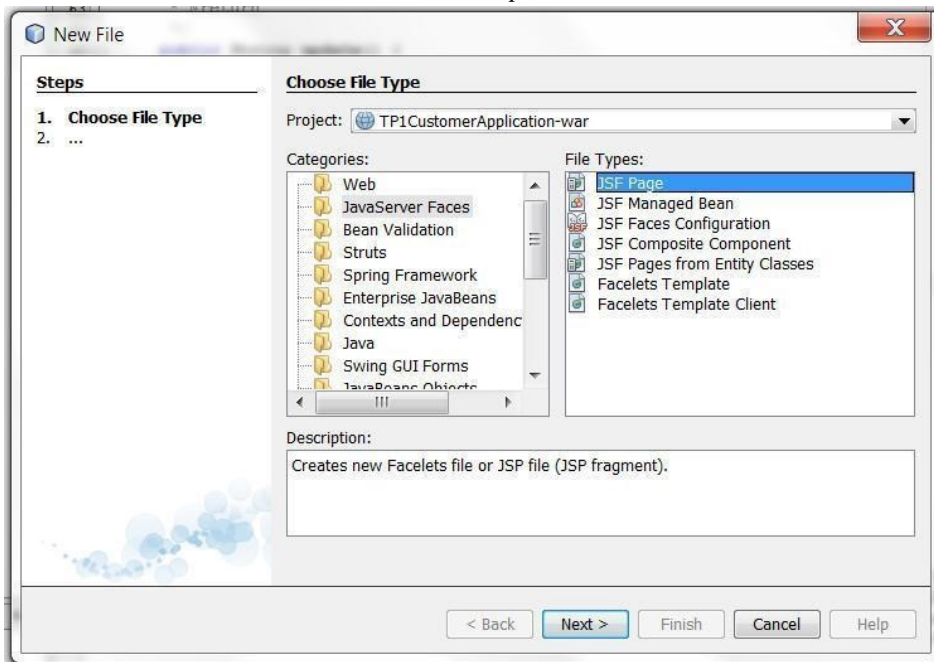

```
* Action handler - Actualiza el modelo Customer en la BD.
* Se llama cuando se presiona el boton update del formulario
* @return
*/
public String update() {
    System.out.println("###UPDATE###");
    customer = customerManager.update(customer);
    return "SAVED"; // Para el caso navegacional
}
/**
* Action handler - retorno hacia la vista de la lista de clientes
* @return
*/
public String list() {
    System.out.println("###LIST###");
    return "LIST";
}
private void refresh() {
    customers=customerManager.getAllCustomers();
}
//Para cargar correctamente los combobox del formulario Details
public javax.faces.model.SelectItem[] getDiscountCodes() {
    SelectItem[] options = null;
    List<DiscountCode> discountCodes = discountCodeManager.getDiscountCodes();
    if (discountCodes != null && discountCodes.size() > 0) {
        int i = 0;
        options = new SelectItem[discountCodes.size()];
        for (DiscountCode dc : discountCodes) {
            options[i++] = new SelectItem(dc.getDiscountCode(),
                dc.getDiscountCode() + " (" + dc.getRate() + "%)");
        }
    }
    return options;
}

public javax.faces.model.SelectItem[] getZipCodes() {
    SelectItem[] options = null;
    List<MicroMarket> zipCodes = zipCodeManager.getZipCodes();
    if (zipCodes != null && zipCodes.size() > 0) {
        int i = 0;
        options = new SelectItem[zipCodes.size()];
        for (MicroMarket dc : zipCodes) {
            options[i++] = new SelectItem(dc.getZipCode(), dc.getZipCode());
        }
    }
    return options;
}
```

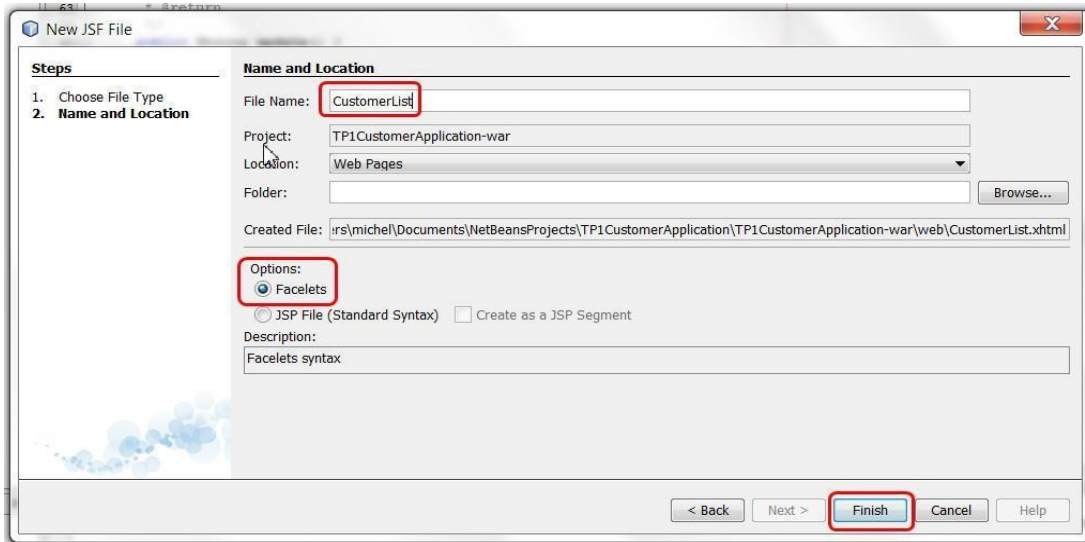
Añadir una página JSF para desplegar clientes en una tabla

En este ejercicio se crean dos vistas una que contendrá la tabla de datos (CustomerList.xhtml) y la segunda que será un formulario para realizar actualización de los datos (CustomerDetails.xhtml). Para crearlas haga click derecho sobre el modulo web y seleccione **New** →

Other -> Categoría Java Server Faces → JSF page.



Ahora haga click en "next", y luego debe colocar un nombre a la página, en este caso se asignará *CustomerList*.



Añadir un datatable JSF widget en la página JSF, y vincularlo con el modelo de "clientes"

Ahora activaremos la paleta de herramientas de Netbeans (menu Window/Palette o usar el atajo ctrl-shift-8). Abrir la sección JSF, y arrastre y suelte el icono "**JSF Data Table from Entity**" donde se indica en la figura dentro de la etiqueta `<h:body>` de la página:

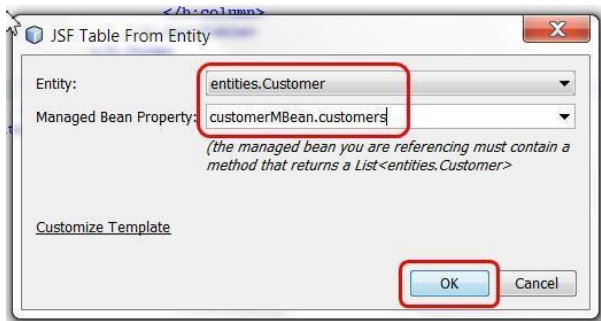


Aparece un cuadro de dialogo que pregunta cual entity class desea asociar con la vista. Seleccione a Customer cómo la entidad y seleccione customerMBean.customers como el nombre de las propiedades en el JSF bean que corresponde a este entity en el JSF bean.

Profundización en Arquitectura de Software - Universidad de Antioquia

Una propiedad es definida por estos getters and setters, y por eso se definió el método `getCustomers()` que devuelve una lista de clientes.

Un modelo de tipo "Customer" o "List<Customer>" en el JSF Bean, es una propiedad que se utiliza con los métodos de acceso `getCustomer()` o `getCustomers()`. Podemos asociar estos modelos a algunos componentes de vista como por ejemplo un datatable que retornan una colección) o para un formulario que retornan un solo Cliente en el momento de hacer una actualización.



Al hacer click en "Ok" se agregaran varias etiquetas en la página JSF.

Configuración de las vistas

Para mejorar la presentación de la vista, en el archivo **CustomerList.xhtml** se agregan los siguientes bloques de código.

En el primero se presenta un estilo llamado ".old", cuya función es poner de color verde una fila, posteriormente es llamada en otro bloque de código en una sentencia condicional explicada más adelante.

```
<style type="text/css">
    .old{
        background-color: #6CE26C !important;
        background-image: none !important;
        color: #000000 !important;
    }
</style>
```

```
<h:head>
    <title>Facelet Title</title>
</h:head>
<h:body>
    <h3>LISTADO DE CLIENTES</h3>

    <f:view>
        <h:form>
```

El siguiente bloque de código corresponde a la configuración del **dataTable** dinámico y que muestra los registros de la Tabla Customer de la BD conectado a la propiedad **customers** del controlador, la última línea es una sentencia condicional que coloca un color sobre la fila cuyo límite de crédito sea menor o igual a US\$60.000. También modifique las etiquetas que se asocian a **customerId** y agregue la etiqueta **<p:barcode>** el cual permite crear un código QR para fila.

```
<p:dataTable value="#{customerMBean.customers}"
    var="item"
    paginator="true"
    rows="10"
    scrollable="true"
    scrollWidth="150%"
    scrollHeight="550"
    rendered="true"
    resizableColumns="true"
    sortMode="multiple"
    emptyMessage="No hay datos"
    widgetVar="customerTable"
    rowStyleClass="#{item.creditLimit le 60000 ? 'old':null}" >
    <p:column headerText="CustomerId"
        sortBy="#{item.customerId}"
        filterBy="#{item.customerId}"
        filterMatchMode="contains" >
        <h:commandLink action="#{customerMBean.showDetails(item)}"
            value="#{item.customerId}" />
    </p:column>

    <p:column headerText="Qr Code" >
    <p:barcode value="#{item.customerId} #{item.creditLimit} #{item.email}"
        type="qr" height="150px" width="150px" />
    </p:column>
```

A continuación está el código:

```
<p:dataTable value="#{customerMBean.customers}" style="margin-bottom:20px"
    var="item"
    paginator="true"
    rows="10"
    scrollable="true"
    scrollWidth="150%"
    scrollHeight="950"
    rendered="true"
    resizableColumns="true"
    sortMode="multiple"
    emptyMessage="No hay datos"
    widgetVar="customerTable"
    rowStyleClass="#{item.creditLimit le 60000 ? 'old':null}" >
    <p:column headerText="CustomerId"
        sortBy="#{item.customerId}"
        filterBy="#{item.customerId}"
        filterMatchMode="contains" >
        <h:commandLink action="#{customerMBean.showDetails(item)}"
            value="#{item.customerId}" />
    </p:column>

    <p:column headerText="Qr Code" >
    <p:barcode value="#{item.customerId} #{item.creditLimit} #{item.email}"
        type="qr" height="150px" width="150px" />
    </p:column>
```

Profundización en Arquitectura de Software - Universidad de Antioquia

Como puede observar cuando hacemos click en customer Id, se llama un método JSF bean que hace que el valor actual de cliente se transmita a otra página JSF con un formulario lleno que muestra los detalles del cliente.

La línea agregada dice que cuando hacemos click en el Id, se llama el metodo `showDetails(Customer c)` definido en la clase JSF `managedBean`

```
/**
 * Action handler - llamado cuando en una fila de la tabla se haga click en ID
 * @param customer
 * @return
 */
public String showDetails(Customer customer) {
    this.customer = customer;
    return "DETAILS"; // Permite enlazar a CustomerDetails.xml
}
```

Este método hace 2 cosas:

1. Almacena el cliente correspondiente a la línea haciendo click en el modelo llamado "customer" (este atributo está en la sesión HTTP, por lo que será accesible incluso si cambiamos de una página a otra).
2. Reenvía a la página JSF `CustomerDetails.xhtml`, este obtendrá los valores almacenados en el modelo del paso 1 para prellenar el formulario que contiene los datos del cliente actual.

Para terminar de modificar la visualización del `discountCode`, si el valor mostrado en la tabla es "entities.DiscountCode[discountCode=M]", se modifica esta línea en la página:

01. `<h:outputText value="#{item.discountCode}"/>`

por esta:

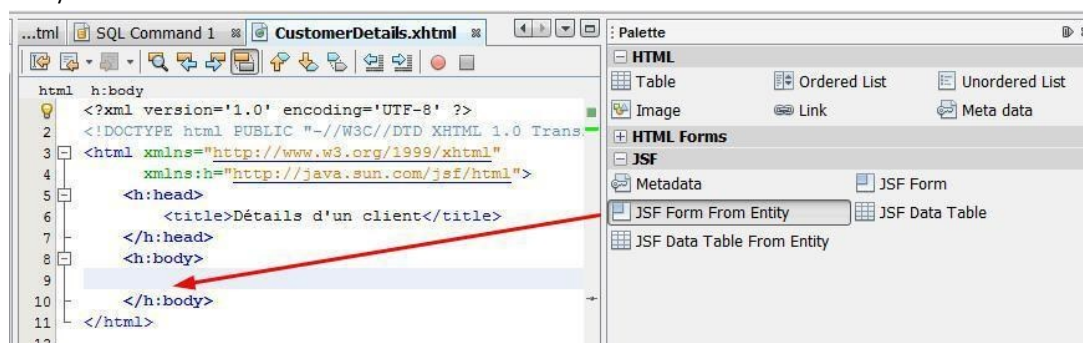
01. `<h:outputText value="#{item.discountCode.discountCode} : #{item.discountCode.rate}%" />`

El primer acceso a la propiedad corresponde a una llamada a `getDiscountCode().getDiscountCode()` de la instancia actual del cliente. El segundo corresponde a una llamada a `getDiscountCode().getRate()`, con lo cual ahora se podrá visualizar así :

DiscountCode
N : 0.00%
M : 11.00%
M : 11.00%

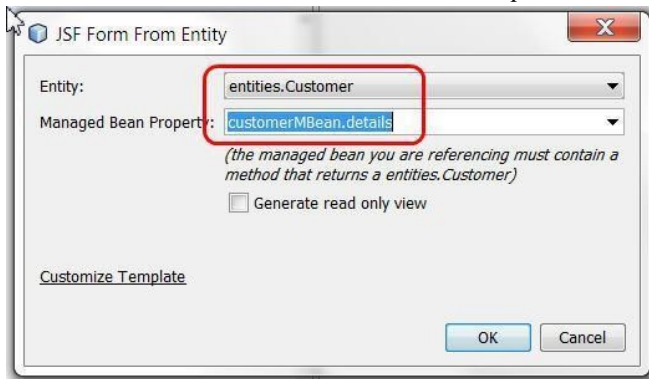
Añadir una página JSF para mostrar los detalles del cliente

1. Añadir una nueva página JSF para el proyecto web, y nombrarlo `CustomerDetails.xhtml`
2. Siguiendo el proceso de la primera vista arrastre en el cuerpo de esta página la herramienta de la paleta llamada "jsf form from entity".



3. Seleccione el nombre de la propiedad en el vean asociado a la entidad Customer.

Profundización en Arquitectura de Software - Universidad de Antioquia



Ahora seleccione la propiedad del Bean Gestionado llamado `customerMBean.details` que corresponde al método `getDetails()`. Haga ahora click en "Ok".

4 En la etiqueta `<h:form>` dentro del archivo **CustomerDetail.xhtml**, se agrega el siguiente bloque de código para que aparezca una lista desplegable correspondiente al descuento y al código ZIP, así como para cargar el control Captcha del formulario.

```
<p:outputLabel value="DiscountCode:" for="discountCode" />
<p:selectOneMenu id="discountCode" value="#{customerMBean.details.discount}" required="true"
    requiredMessage="The DiscountCode field is required.">
    <f:selectItems value="#{customerMBean.discountCodes}" />
</p:selectOneMenu>

<p:outputLabel value="Zip:" for="zip" />
<p:selectOneMenu id="zip" value="#{customerMBean.details.z}" required="true"
    requiredMessage="The Zip field is required.">
    <f:selectItems value="#{customerMBean.zipCodes}" />
</p:selectOneMenu>

<p:outputLabel value="Discount:" for="discount" />
<p:inputText id="discount" value="#{customerMBean.details.discount}" title="Discount" />
</p:panelGrid>
<h2>Primefaces - Captcha</h2>
<p:messages id="msg" />
<p:captcha id="captcha"></p:captcha>
<br/>
<h:commandButton id="back" value="Back" action="#{customerMBean.list()}"/>
<h:commandButton id="update" value="Update" action="#{customerMBean.update()}"/>

</h:form>
```

Este es el código:

```
<p:outputLabel value="DiscountCode:" for="discountCode" />
    <p:selectOneMenu id="discountCode" value="#{customerMBean.details.discount}" required="true"
        requiredMessage="The DiscountCode field is required.">
        <f:selectItems value="#{customerMBean.discountCodes}" />
    </p:selectOneMenu>

    <p:outputLabel value="Zip:" for="zip" />
    <p:selectOneMenu id="zip" value="#{customerMBean.details.z}" required="true"
        requiredMessage="The Zip field is required.">
        <f:selectItems value="#{customerMBean.zipCodes}" />
    </p:selectOneMenu>

    <p:outputLabel value="Discount:" for="discount" />
    <p:inputText id="discount" value="#{customerMBean.details.discount}" title="Discount" />
```

```

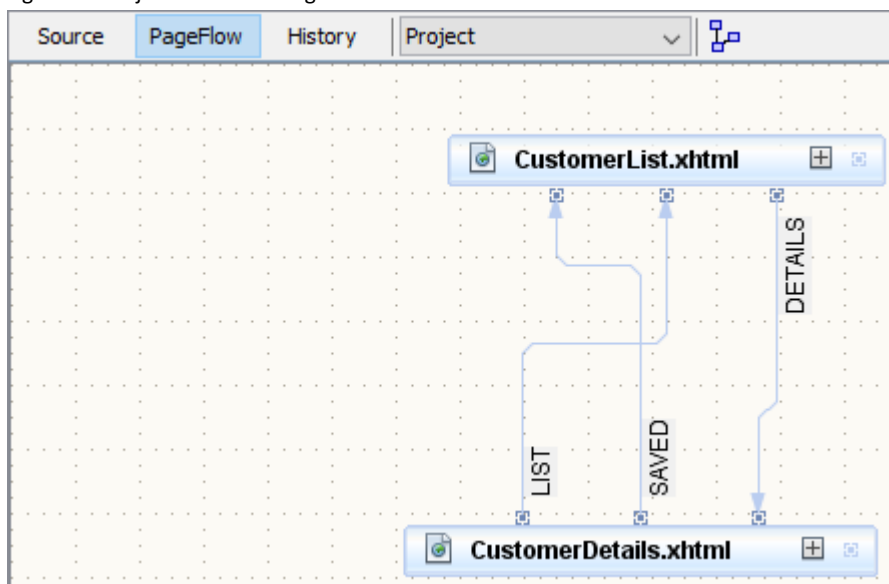
</p:panelGrid>
<h2>Primefaces - Captcha</h2>
<p:messages id="msg"/>
<p:captcha id="captcha"></p:captcha>
<br/>
<h:commandButton id="back" value="Back" action="#{customerMBean.list()}">
<h:commandButton id="update" value="Update" action="#{customerMBean.update()}">

```

Después de este paso ya han sido creadas las vistas correspondientes a cada uno de las entidades.

Crear la navegabilidad entre las vistas

Para formar el modelo navegacional abra el archivo **faces-config.xml** y vaya a la pestaña **Detail** y sobre las vistas generadas haga los siguientes flujos de casos navegacionales.



Si usted regresa al código XML observará que se han adicionado las siguientes etiquetas:

```

<navigation-rule>
  <from-view-id>/CustomerList.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>DETAILS</from-outcome>
    <to-view-id>/CustomerDetails.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>

<navigation-rule>
  <from-view-id>/CustomerDetails.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>LIST</from-outcome>
    <to-view-id>/CustomerList.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>SAVED</from-outcome>
    <to-view-id>/CustomerList.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>

```

Probar el proyecto

Ahora se probará el proyecto, para esto haga click derecho en el proyecto EAR (el que tiene un icono de triángulo azul), y seleccione "Ejecutar" - "Run". Esto debería mostrar la página CustomerList.xhtml por defecto, que se cargará en la URL: "




<http://localhost:8080/Customers/>"

También se puede cambiar la página de inicio predeterminada del proyecto mediante la edición del archivo web.xml, que se ha creado al inicio.

Podrá observar que aparece la siguiente vista en el web browser:

localhost:8080/Customers-web/ 67% Buscar

Listado de Clientes

Customerid	Qr Code	Name	Addressline1	Addressline2	City	State	Phone	Fax	Email	CreditLimit	DiscountCode
1		Jumbo Eagle Corp	111 E. Las Olivas Blvd	Suite 51	Fort Lauderdale	FL	305-555-018	305-555-0189	jumboeagle@gmail.com	100000	N : 0.00%
2		New Enterprises	9754 Main Street	P.O. Box 567	Miami	FL	305-555-014	305-555-0149	www.newexample.com	50000	M : 11.00%
25		Wren Computers	8989 Red Albatross Drive	Suite 9897	Houston	TX	214-555-013	214-555-0134	www.wrencomp.example.co	25000	M : 11.00%

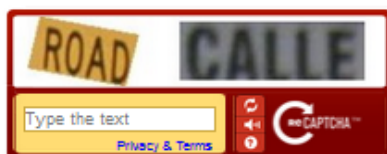
Arquitect... Arquitect... FT Multa a B... Desarroll...

localhost:8080/Customers-web/faces/CustomerList.xhtml;

Create/Edit

CustomerId: *	1
Name:	Jumbo Eagle Corp
Addressline1:	111 E. Las Olivas Blvd
Addressline2:	Suite 51
City:	Fort Lauderdale
State:	FL
Phone:	305-555-0188
Fax:	305-555-0189
Email:	jumboeagle@gmail.co
CreditLimit:	100000
DiscountCode: *	N (0.00%)
Zip: *	95035
Discount:	N

Primefaces - Captcha

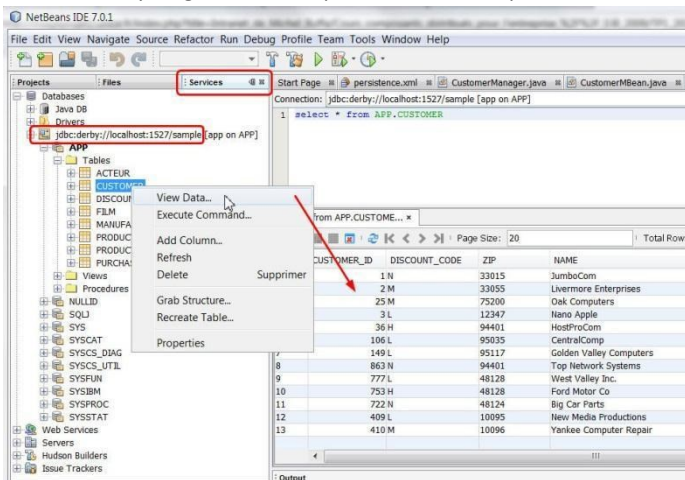


Back Update

Profundización en Arquitectura de Software - Universidad de Antioquia

Observe la funcionalidad de cada vista así como la generación de los códigos QR (Intente escanear alguno por medio de su smartphone para ver la correcta codificación de los datos), y el control Captcha.

Los datos visualizados provienen de una BD llamada **sample**. Abra la BD en el separador **Services**, observe la tabla CUSTOMER y haga click derecho y luego View Data, para ver los datos que se muestran en la página JSF.



V. Ejercicios propuestos

FECHA DE ENTREGA: 26 de octubre 2020

Realice una aplicación con MAVEN y Primefaces que simule una pasarela de pago. El sistema debe estar conformado mínimo de dos vistas. La primera vista contiene los datos del registro de la transacción incluyendo el nombre del cliente, el email, el número de la tarjeta de crédito, el código de seguridad CVV de la tarjeta (validar que sea de tres dígitos), el tipo de tarjeta de crédito (Visa, Mastercard, Diners, American Express) que se cargará automáticamente según la validación que se haga con el número de la tarjeta según los primeros 5 dígitos de la tarjeta así:

- American Express: Rango del 11111 al 22222
- Diners: Rango del 33334 al 44444
- Visa: Rango del 55555 al 66666
- Mastercard: Rango de 77777 al 88888

También se colocará el valor de la transacción (validar rangos de entre \$500 y \$10000) Fecha de Vencimiento de la tarjeta (MM/YYYY) El formulario deberá tener un control Captcha antes de realizar la transacción.

NOTA: Todos los campos de este formulario serán obligatorios. El segundo formulario presentará el resultado de la transacción y se presentará un código QR que contenga codificado los datos del id del cliente, el valor de la transacción y la marca de tiempo de la transacción. Recuerde que debe estar cargados los datos en una Base de Datos.

VII. Bibliografía Propuesta - Webgrafía

- [1] [Guía de Desarrollo Java EE.](#)
- [2] [Introducción a EJB.](#)
- [3] [Tutorial JSF.](#)
- [4] [Introducción a JSF.](#)
- [5] [JSF Servlet vs JSP](#)
- [6] [Guía PrimeFaces.](#)
- [7] [JSF Managed Beans.](#)