Java EE 6 Hands-on Lab
OTN Developer Day

Arun Gupta
blogs.oracle.com/arungupta, @arungupta
Oracle Corporation

Table of Contents

1.0 Introduction

The Java EE 6 platform allows you to write enterprise Java applications using much lesser code from its earlier versions. It breaks the “one size fits all” approach with Profiles and improves on the Java EE 5 developer productivity features tremendously. Several specifications like CDI, JSF 2, JAX-RS, JPA 2, and Servlets 3 make the platform more powerful by adding new functionality and yet simple to use. NetBeans, Eclipse, and IntelliJ provide extensive tooling for Java EE 6.

This hands-on lab will build a typical 3-tier end-to-end Web application using Java EE 6 technologies including JPA2, JSF2, CDI, EJB 3.1, JAX-RS 1.1, and Servlets 3. The development and deployment of the application will be performed using NetBeans and GlassFish 3.1.

1.1 Software Downloads

If this lab is performed as part of the Oracle Technology Network Developer Day then all the software is pre-installed on a Virtual Box instance. Otherwise you'll need to download and install the following software:

- JDK 6 or 7 from http://www.oracle.com/technetwork/java/javase/downloads/index.html.
- NetBeans 7.0+ “All” or “JavaEE” version from http://netbeans.org/downloads/index.html. This version includes a pre-configured GlassFish 3.1.

2.0 Problem Statement

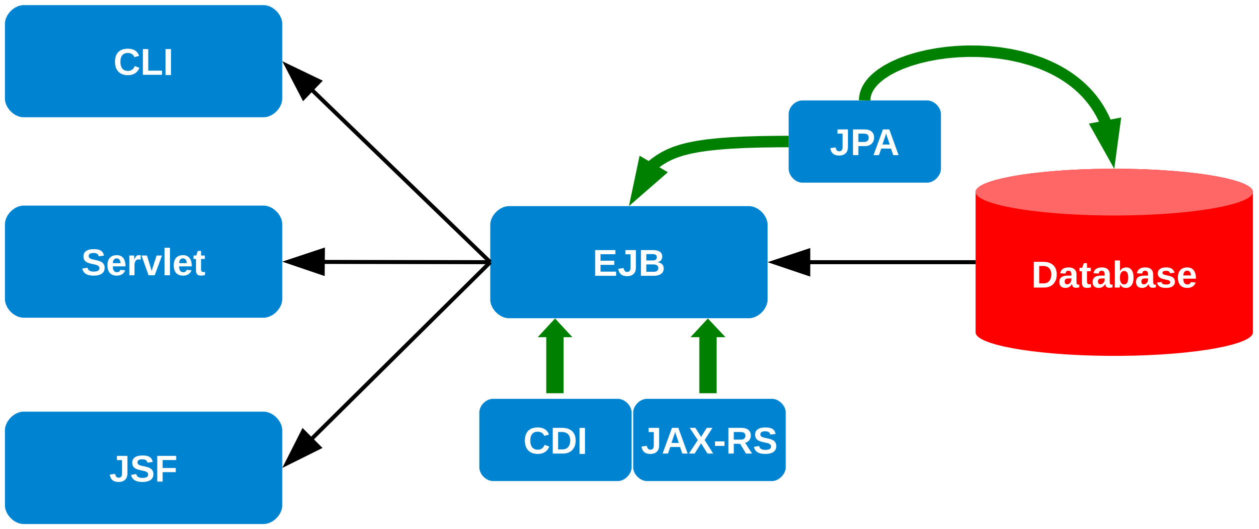
This hands-on lab builds a typical 3-tier Web application that displays customer information from a database table in a Servlet, a Web page, and as RESTful Web service using CLI.

The main steps are:

1. Generate JPA Entities from the database table
2. Refactor generated entities for a more intuitive O/R mapping

3. Create an EJB for querying the database
4. Create a Servlet for testing the EJB and displaying values from the database table
5. Enable CDI and make the EJB EL-injectable
6. Display the values in a JSF2/Facelets-based view
7. Expose JPA entities as a RESTful resource by using JAX-RS

A picture is worth a thousand words and so here is a block diagram of the application:

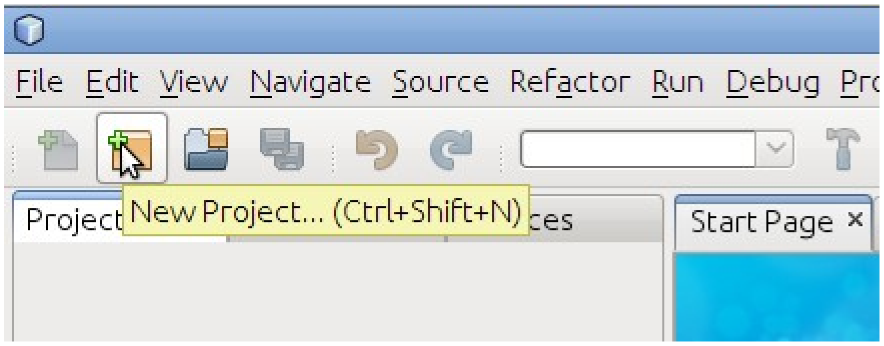


This application uses a sample database that comes pre-configured in both NetBeans “All” and “Java EE” downloaded bundled versions.

3.0 Build the template Web application

This section will build a template Web application.

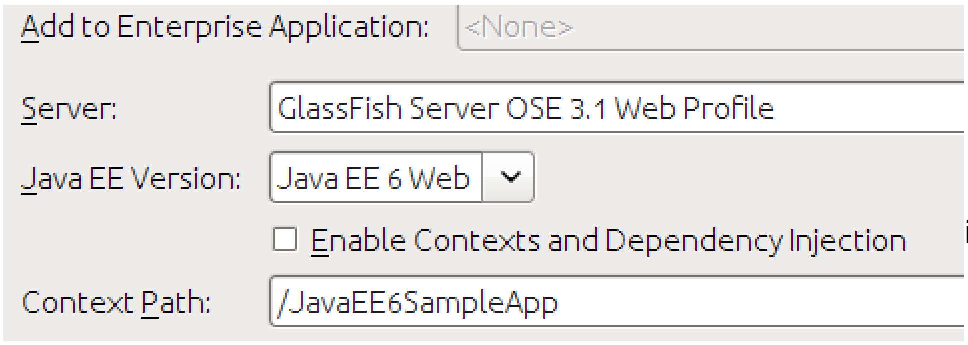
3.1 In NetBeans IDE, create a new Web application by selecting the “New Project” icon as shown.


File Edit View Navigate Source Refactor Run Debug
New Project... (Ctrl+Shift+N)
Project
Start Page

3.2 Choose “Java Web” as category and “Web Application” as Projects.

and click on “Next>”.

3.3 Specify the project name as “JavaEE6SampleApp” and click on “Next>”.



Add to Enterprise Application: <None>

Server: GlassFish Server OSE 3.1 Web Profile

Java EE Version: Java EE 6 Web

☐ Enable Contexts and Dependency Injection

Context Path: /JavaEE6SampleApp

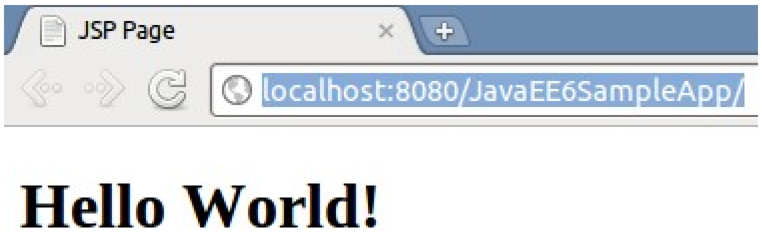
3.4 Choose the pre-configured GlassFish Server 3.1 as the Server. The actual server name may differ in your case, as in this case, if you have configured it externally. Ensure that the Java EE Version selected is “Java EE 6 Web” and click on “Finish”.

This generates a template Web project.

There is no “web.xml” in the WEB-INF folder as Java EE 6 makes it optional in most of the common cases.

3.5 Right-click on the project and select “Run”. This will start the chosen GlassFish server, deploy the Web application on the server, opens the default web browser, and displays “http://localhost:8080/JavaEE6SampleApp/index.jsp”. The default page looks like as shown here.

Note that even though the “index.jsp” is not displayed in the URL window, this file is displayed as default.



JSP Page

localhost:8080/JavaEE6SampleApp/

Hello World!

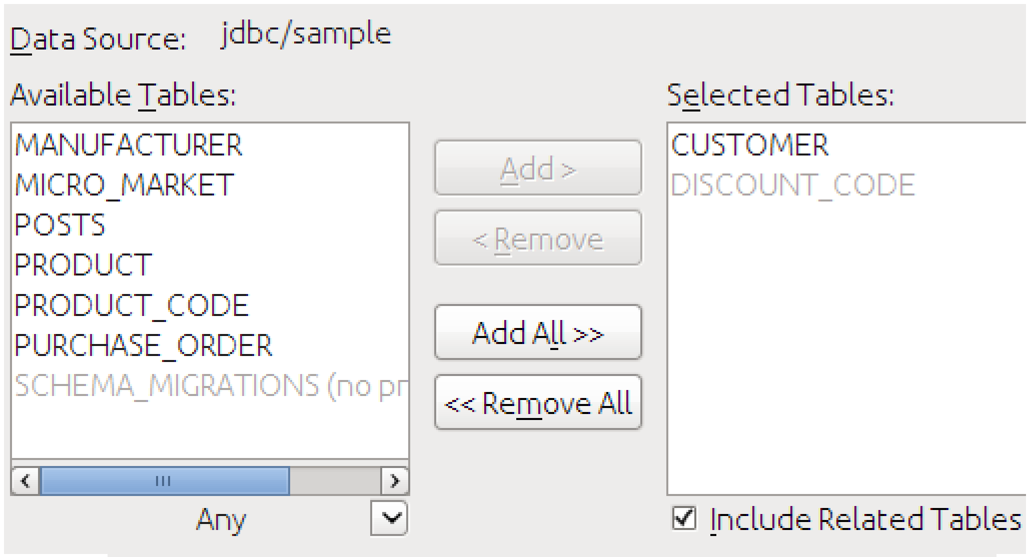
A display of this page ensures that the project is successfully created.

4.0 Generate JPA Entities

Java Persistence API (JPA) is a standard API that defines mapping between database tables and Java classes. These POJOs can then be used to perform all the database operations using Java Persistence Query Language (JPQL) which is SQL-like syntax and operates on the Java model instead.

This section will use the NetBeans wizards to generate JPA entities from a sample database and optionally customize them to be more intuitive for Java developers.

4.1 In the NetBeans IDE, right-click on the project and select “New”, “Other...”, “Persistence”, “Entity Classes from Database...”. Choose “jdbc/sample” as the Data Source from the drop down list box as shown. This will show all the tables from this data source.



Data Source: jdbc/sample

Available Tables:

- MANUFACTURER
- MICRO_MARKET
- POSTS
- PRODUCT
- PRODUCT_CODE
- PURCHASE_ORDER
- SCHEMA_MIGRATIONS (no pr

Selected Tables:

- CUSTOMER
- DISCOUNT_CODE

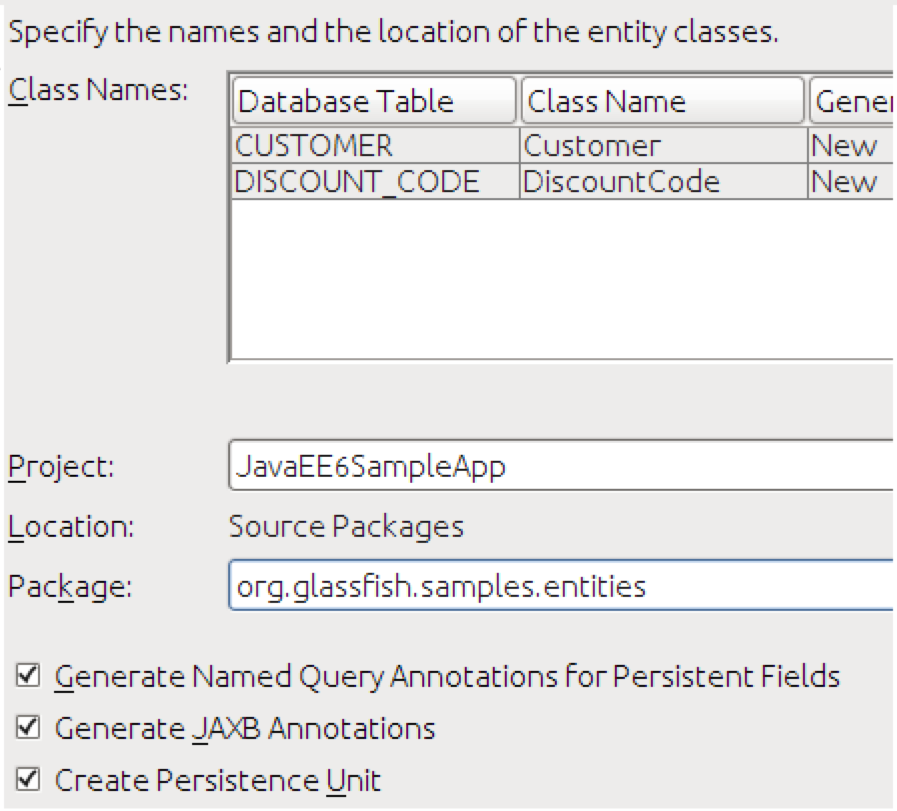
☑ Include Related Tables

4.2 Select “CUSTOMER” table from the “Available Tables” and click “Add>”. Notice that the “DISCOUNT_CODE” table is automatically selected because of the foreign key references and the selected “Include Related Tables” checkbox.

Click on “Next>”.

4.3 Enter the package name “org.glassfish.samples.entities” as shown.

The mapped class names are shown in the “Class Name” column and can be changed here, if needed.



Specify the names and the location of the entity classes.

Class Names:

Database Table	Class Name	Gener
CUSTOMER	Customer	New
DISCOUNT_CODE	DiscountCode	New

Project: JavaEE6SampleApp

Location: Source Packages

Package: org.glassfish.samples.entities

☑ Generate Named Query Annotations for Persistent Fields

☑ Generate JAXB Annotations

☑ Create Persistence Unit

Notice the following points:

- The first check box allows NetBeans to specify multiple @NamedQuery on the generated JPA entity that allows to query the database using pre-defined queries. New queries can easily be added as well following the “Don't Repeat Yourself” design pattern.

- The second check box ensures that the @XmlRootElement annotation is generated on the JPA entity class so that it can be converted to an XML representation easily by JAXB. This feature will be used later when the entities are exposed as a RESTful resource using JAX-RS.
- The third check box generates the required Persistence Unit required by JPA.

Click on “Finish” to complete the entity generation.

In the NetBeans IDE, expand “Source Packages”, “org.glassfish.samples.entities”, and double-click “Customer.java”.

Notice the following points:

- The generated class-level @NamedQuery annotations uses JPQL to define several queries. There is one query “Customer.findAll” that retrieves all rows from the database and one query, aka “findBy” query, for each field (which maps to a column in the table). Additional queries may be added here providing a central location for all your query-related business logic.
- BeanValidation constraints are generated on each field based upon the schema definition. These constraints are then used by the validator included in the JPA implementation before an entity is saved, updated, or removed from the database.
- The regular expression-based constraint may be used to enforce phone or zipcode in a particular format.
- The “discountCode” field is marked with @JoinColumn creating a join with the appropriate table.

4.5 Understand and Customize JPA entities (OPTIONAL)

This section will customize the generated JPA entities to make them more intuitive for a Java developer.

4.5.1 Edit “Customer.java” and change the class structure to introduce an Embeddable class for street, city, country, and zip fields as these are logically related entities.

4.5.2 Replace the following code:

```
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 10)
    @Column(name = "ZIP")
    private String zip;
```

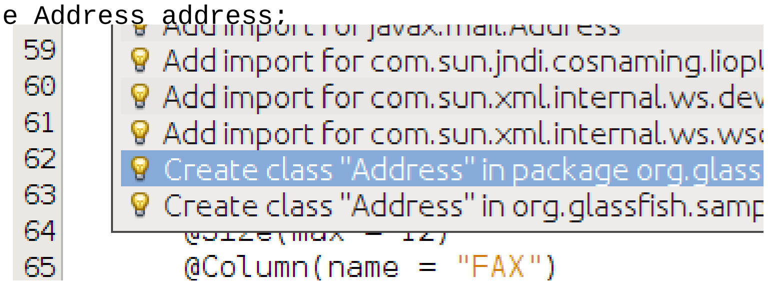
and

```
    @Size(max = 30)
    @Column(name = "ADDRESSLINE1")
    private String addressline1;
    @Size(max = 30)
    @Column(name = "ADDRESSLINE2")
    private String addressline2;
    @Size(max = 25)
    @Column(name = "CITY")
    private String city;
    @Size(max = 2)
    @Column(name = "STATE")
    private String state;
```

with

```
@javax.persistence.Embedded private Address address;
```

Use the yellow bulb in the left bar to create a new class in the current package as shown:



```
Add import for javax.mail.Address
Add import for com.sun.jndi.cosnaming.iiopl
Add import for com.sun.xml.internal.ws.dev
Add import for com.sun.xml.internal.ws.ws
Create class "Address" in package org.glass
Create class "Address" in org.glassfish.samp
@Size(max = 12)
@Column(name = "FAX")
```

Notice the following points:

- The two blocks of code above are not adjacent.
- Copy/pasting only the fields will show a red line under some of the methods in your entity but will be fixed in 4.5.5 below.
- @Embedded annotation ensures that this field's value is an instance of an embeddable class.

4.5.3 Change “Address” class so that it is a public class, annotated with @Embeddable such that it can be used as embeddable class later, and also implements the Serializable interface. The updated class definition is shown:

```
@javax.persistence.Embeddable
public class Address implements java.io.Serializable
```

4.5.4 In the “Address” class, paste the different fields code replaced earlier from “Customer.java” and add getter/setters for each field. These can be easily generated by going to the “Source”, “Insert Code...” menu, selecting “Getter and Setter...”, selecting all the fields, and clicking on “Generate”.

Fix all the imports by right-clicking in the editor and selecting “Fix Imports...”.

4.5.5 Make the following changes in “Customer.java”

i. Remove the getter/setter for the previously removed fields and add a new getter/setter for “address” field as:

```
public Address getAddress() { return address; }
public void setAddress(Address address) { this.address = address; }
```

ii. Change the different @NamedQuery to reflect the nested structure for Address by editing the queries identified by “Customer.findByAddressline1”, “Customer.findByAddressline2”, “Customer.findByCity”, “Customer.findByState”, and “Customer.findByZip” such that “c.addressline1”, “c.addressline2”, “c.city”, “c.state”, “c.zip” is replaced with “c.address.addressline1”, “c.address.addressline2”, “c.address.city”, “c.address.state”, and “c.address.zip” respectively.

Here is an updated query:

```
@NamedQuery(name = "Customer.findByZip", query = "SELECT c FROM Customer c
WHERE c.address.zip = :zip")
```

iii. Remove the following constructor:

```
public Customer(Integer customerId, String zip) {
    this.customerId = customerId;
    this.zip = zip;
}
```

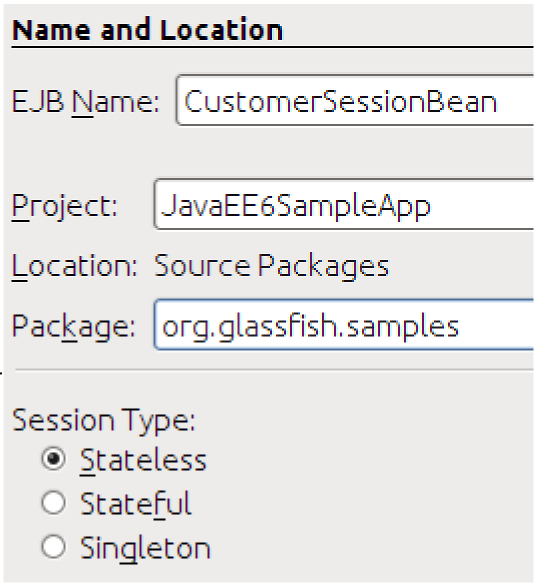
This constructor uses “zip” field which now exists in a different class. Alternately, you can comment it as well.

iv. Change the “toString()” method to the one shown below:

```
@Override
public String toString() {
    return name + "[" + customerId + "]";
}
```

This will ensure that the customer's name and unique identifier are printed when the “toString” method is invoked on the generated entity.

5.0 Create EJB for querying the database


Name and Location

EJB Name: CustomerSessionBean

Project: JavaEE6SampleApp

Location: Source Packages

Package: org.glassfish.samples

Session Type:
● Stateless
○ Stateful
○ Singleton

Java EE 6 defines a simplified definition and packaging for EJBs. Any POJO can be converted into an EJB by adding a single annotation (@Stateless, @Stateful, or @Singleton). No special packaging is required for EJBs as they can be packaged in a WAR file in the WEB-INF/classes directory.

This section will create an Enterprise JavaBean to query the database and show the simplicity of the Java EE 6 platform.

5.1 Create a new stateless EJB. Right-click on “org.glassfish.samples” package, select “New”, “Session Bean...”, specify the EJB Name as “CustomerSessionBean”, take all the defaults, and click on “Finish” as shown. This will create a stateless EJB.

5.2 EJBs are not re-entrant and so we can inject an instance of EntityManager as shown:

```
@PersistenceContext
EntityManager em;
```

Make sure to resolve the imports by right-clicking on the editor pane and selecting “Fix Imports”.

5.3 Add the following method in the EJB:

```
public List<Customer> getCustomers() {
   return (List<Customer>)em.createNamedQuery("Customer.findAll").getResultList();
}
```

This method uses a pre-generated @NamedQuery to retrieve all the customers from the database and return it as a List<Customer>. Remember to fix the imports.

That's all it takes to create an EJB – no deployment descriptors and no special packaging. In this case the EJB is packaged in a WAR file.

6.0 Create a Servlet for testing the EJB

Servlets can easily be defined using a POJO, with a single annotation, and no “web.xml” in most of the cases.

This section will create a Servlet and inject/invoke the EJB.

6.1 Right-click on the package “org.glassfish.samples”, select “New”, “Servlet...”. Enter the class name as “TestServlet” and click on “Finish”.

Expand “Web Pages”, “WEB-INF” and notice that no “web.xml” is generated for describing this Servlet as all the information is captured in the @WebServlet annotation.

```
@WebServlet(name = "TestServlet", urlPatterns = {"/TestServlet"})
public class TestServlet extends HttpServlet {
```

6.2 Invoke EJB business method from Servlet

Inject this bean into Servlet by using the following code as the first line in the TestServlet class:

```
@EJB CustomerSessionBean ejb;
```

Fix the imports.

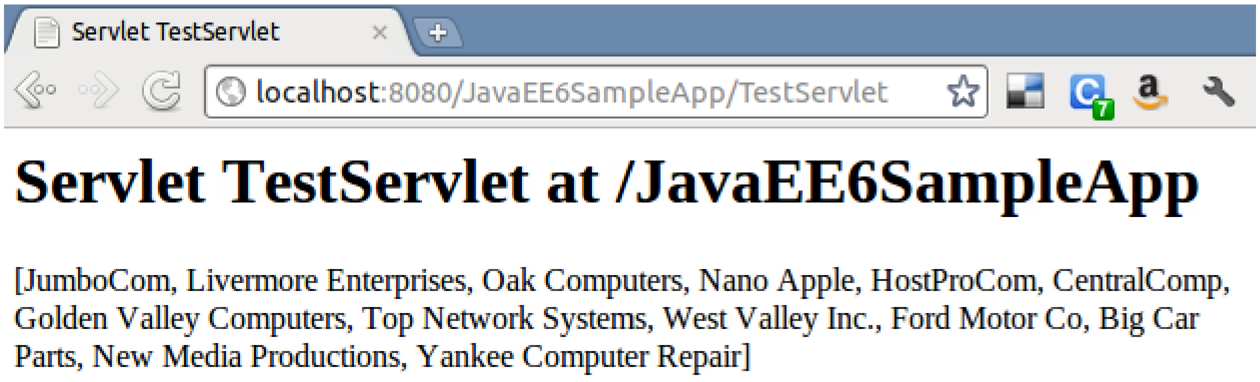
Un-comment the code in the “try” block of the “processRequest” method by removing the first and last the line in the try block and add code to invoke the business method. The updated “try” block looks like:

```
try {
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet TestServlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Servlet TestServlet at " + request.getContextPath () +
"</h1>");
    out.println(ejb.getCustomers());
    out.println("</body>");
    out.println("</html>");
} finally {
```

The business method invocation is highlighted in bold letters.

Save the file.

6.6 Right-click in the editor pane of “TestServlet.java”, select “Run File”, and click on “OK”. This loads the page at “http://localhost:8080/JavaEE6SampleApp/TestServlet” in a browser window and displays the output as:



Servlet TestServlet at /JavaEE6SampleApp

[JumboCom, Livermore Enterprises, Oak Computers, Nano Apple, HostProCom, CentralComp, Golden Valley Computers, Top Network Systems, West Valley Inc., Ford Motor Co, Big Car Parts, New Media Productions, Yankee Computer Repair]

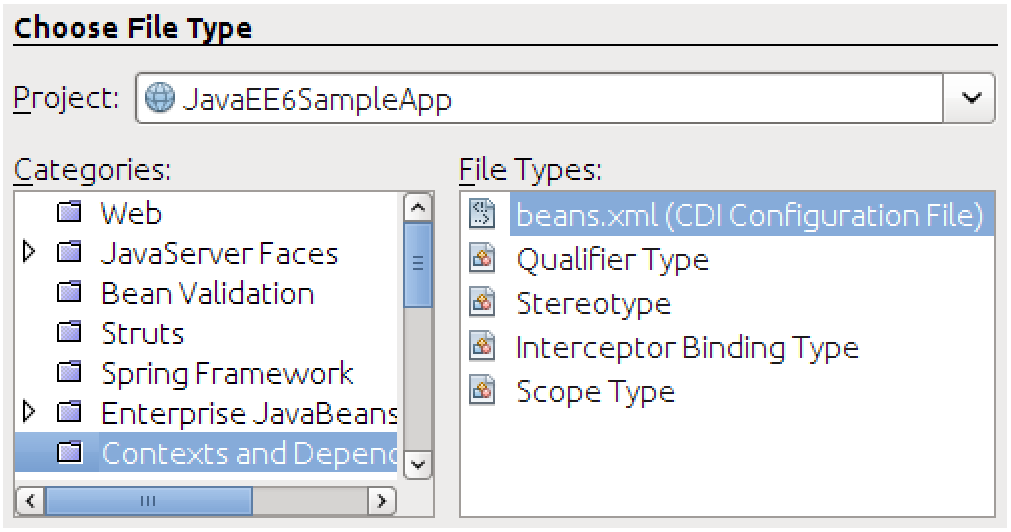
This shows that the rows from the database tables are retrieved successfully and displayed in the Servlet.

7.0 Enable CDI and make the EJB EL-injectable

The Contexts and Dependency Injection (CDI) is a new specification in the Java EE 6 platform and bridges the gap between the transactional and the Web tier by allowing EJBs to be used as the “backing beans” of JSF pages. This eliminates the need for any “glue code”, such as *JSF managed beans*, and there by further simplifying the Java EE platform.

In this section, we'll enable CDI for our project and then make it injectable in an Expression Language (EL), such as .XHTML pages that will be used later.

7.1 Right-click on the project, say “New”, “Other...”, choose “Contexts and Dependency Injection”, select “beans.xml (CDI Configuration File)” as shown:

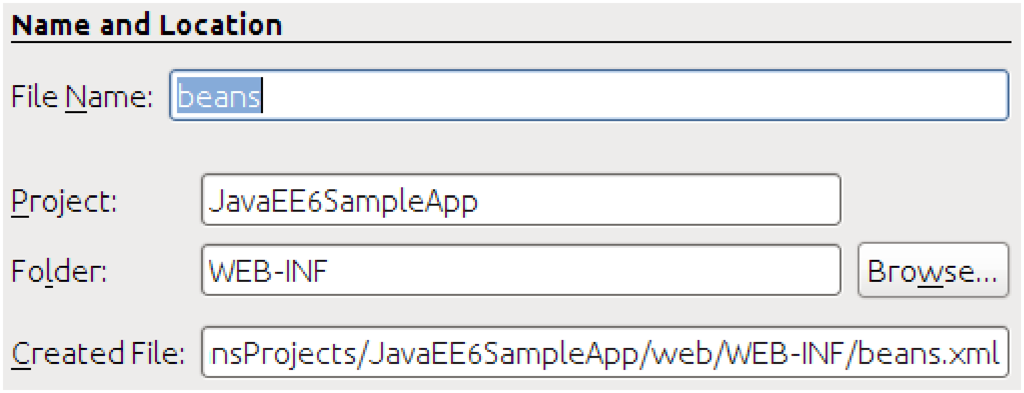

Choose File Type

Project: JavaEE6SampleApp

Categories:
- Web
- JavaServer Faces
- Bean Validation
- Struts
- Spring Framework
- Enterprise JavaBeans
- Contexts and Depend

File Types:
- beans.xml (CDI Configuration File)
- Qualifier Type
- Stereotype
- Interceptor Binding Type
- Scope Type

Click on “Next>”, take all the defaults as shown:


Name and Location

File Name: beans

Project: JavaEE6SampleApp

Folder: WEB-INF

Created File: nsProjects/JavaEE6SampleApp/web/WEB-INF/beans.xml

and click on “Finish”.

This generates an empty “beans.xml” file in the “WEB-INF” folder and ensures that all POJOs in the WAR file are available for injection.

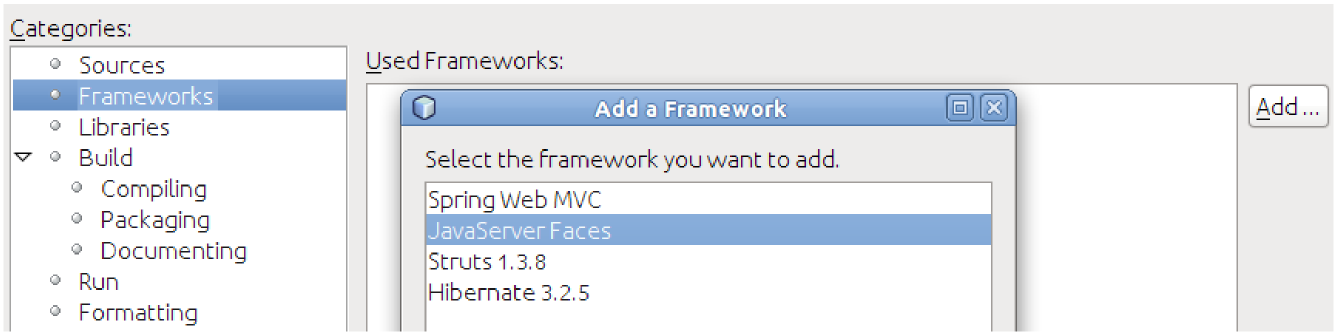
7.2 Add “@javax.inject.Named” CDI qualifier on the “CustomerSessionBean” class. This is a pre-defined CDI qualifier and ensures that the EJB can now be injected in an expression language.

8.0 Display the values in a JSF2/Facelets-based view

JavaServer Faces 2 allows Facelets to be used for view templates. This has huge benefits as Facelets are created using only XHTML and CSS and rest of the business logic is contained in the backing beans. This truly ensures that MVC architecture recommended by JSF can be easily enforced. Even though JSPs can still be used as view templates but Facelets are highly recommended with JSF2.

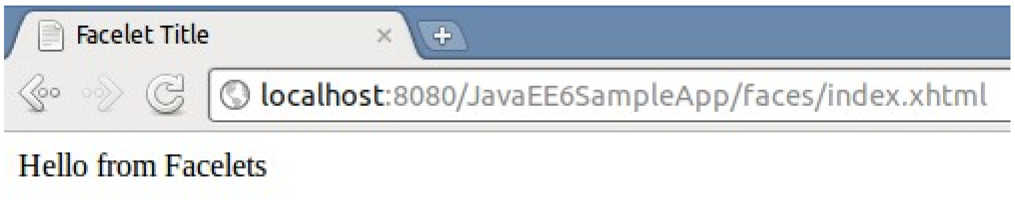
This section will add the JSF support to the application so that the values from the database table can be displayed in a nicely formatted way.

8.1 Right-click on the project, select “Properties”, “Frameworks”, “Add...”, and select “JavaServer Faces” as shown:



With JSF 2.0 implementation in GlassFish, if you are using any JSF-specific annotations then the framework is automatically registered by the underlying Web container. But since our application is not using any such annotation so we have to explicitly enable it. Click on “OK”, take the default configuration, and click on “OK” again.

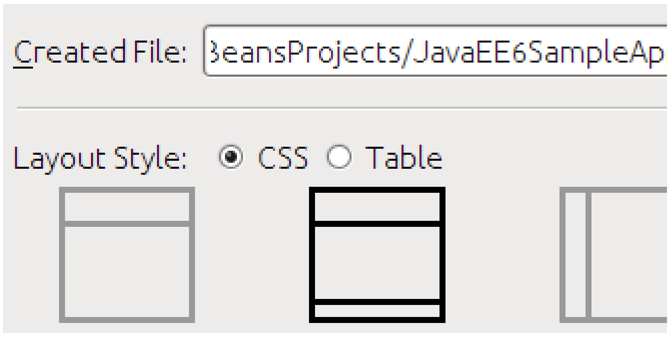
Registering the framework in this case will generate a “web.xml” and register the “FaceServlet” using the “/faces” URL pattern. It also generates “index.xhtml” page which can be verified by viewing “http://localhost:8080/JavaEE6SampleApp/faces/index.xhtml” with the following output:



```
Hello from Facelets
```

8.2 JSF2 allows to create XHTML/CSS-based templates that can be used for providing a consistent look-and-feel for different pages on your website. Lets create a template first and then use it for our web page.

Right-click on the project, select “New”, “Other”, “Java Server Faces”, “Facelets Template...”, change the name to “template”, click on “Browse...”, select the “WEB-INF” folder, and select the template as shown.


Created File: BeansProjects/JavaEE6SampleAp
Layout Style: ● CSS ○ Table

Click on “Finish”.

Notice the following points:

- This generates “template.xhtml” in the “WEB-INF” folder and two stylesheets in the “resources/css” folder.
- The “template.xhtml” page contains three <div>s with <ui:insert>s named “top”, “content”, and “bottom”. These are the placeholders for adding content to provide a consistent look-and-feel.
- Its a recommended practice to keep templates pages in the “WEB-INF” folder such that they are not accessible outside the web application.

8.3 In the generated “template.xhtml”, replace the text “Top” (inside <ui:insert name="top">) with:

```
<h1>Java EE 6 Sample App</h1>
```

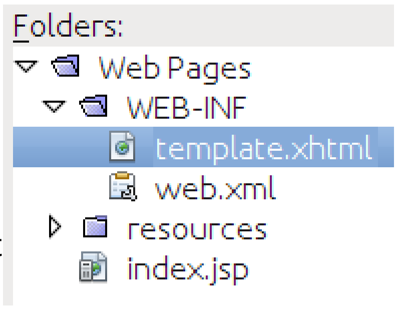
and replace the text “Bottom” with (inside <ui:insert name="bottom">) with:

```
<center>Powered by GlassFish!</center>
```

The “top” and “bottom” <div>s will be used in other pages in our application to provide a consistent look-and-feel for the web application. The “content” <div> will be overridden in other pages to display the business components.

8.4 Now lets re-generate “index.xhtml” to use this template. This page, called as “client page”, will use the header and the footer from the template page and override the required <div>s using <ui:define>. The rest of the section is inherited from the template page.

Delete “index.xhtml” by right-clicking and selecting “Delete”.



Right-click on “Web Pages”, select “New”, “Other”, “Java Server Faces” in Categories and “Facelets Template Client”... in File Types. Click on “Next>”.

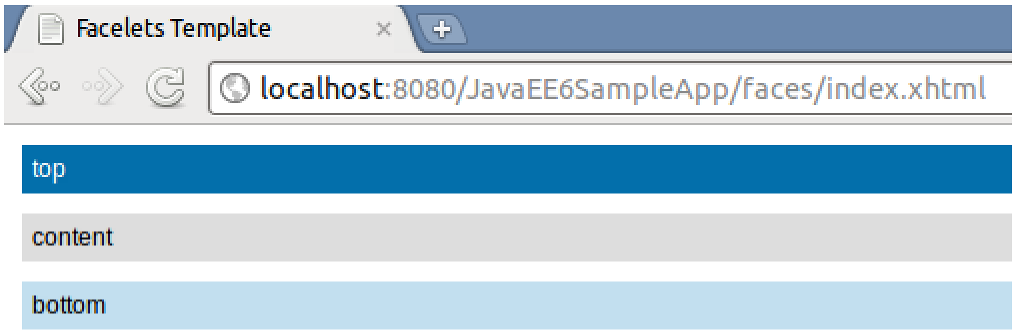
Enter the file name as “index”, choose the “Browse...” button next to “Template:” text box, select “template.xhtml” as shown, and click on “Select File”.

Click on “Finish”.

Notice the following points:

- The generated page has <ui:composition template='./WEB-INF/template.xhtml'> indicating that this page is using the template page created earlier.
- It has three <ui:define> elements with the exact same name as in the template. This allows specific sections of the template page to be overridden. The sections that are not overridden are picked up from the template.

8.5 Refresh “http://localhost:8080/JavaEE6SampleApp/faces/index.xhtml” to see the output as:



The output displays three sections from the .xhtml file as generated by the NetBeans wizard.

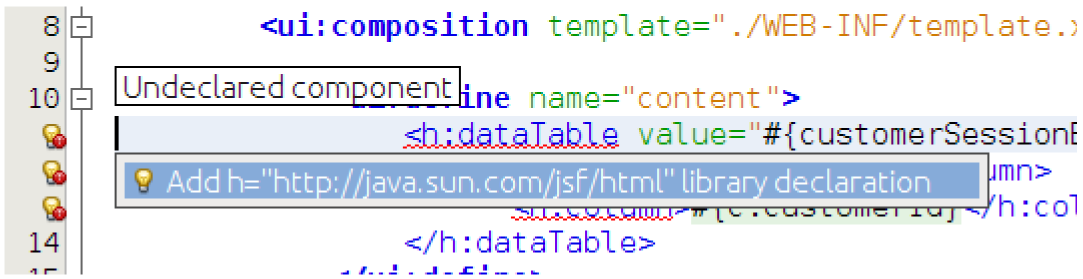
8.6 In “index.xhtml”, delete the <ui:define> element with name “top” and “bottom” as these sections are already defined in the template.

8.7 Replace the text “content” (inside <ui:define name="content">) with:

```
<h:dataTable value="#{customerSessionBean.customers}" var="c">
      <h:column>#{c.name}</h:column>
      <h:column>#{c.customerId}</h:column>
</h:dataTable>
```

This JSF fragment injects the EJB into the expression language, invokes its “getCustomers()” method, iterates through all the values, and then displays the name and id of each customer.

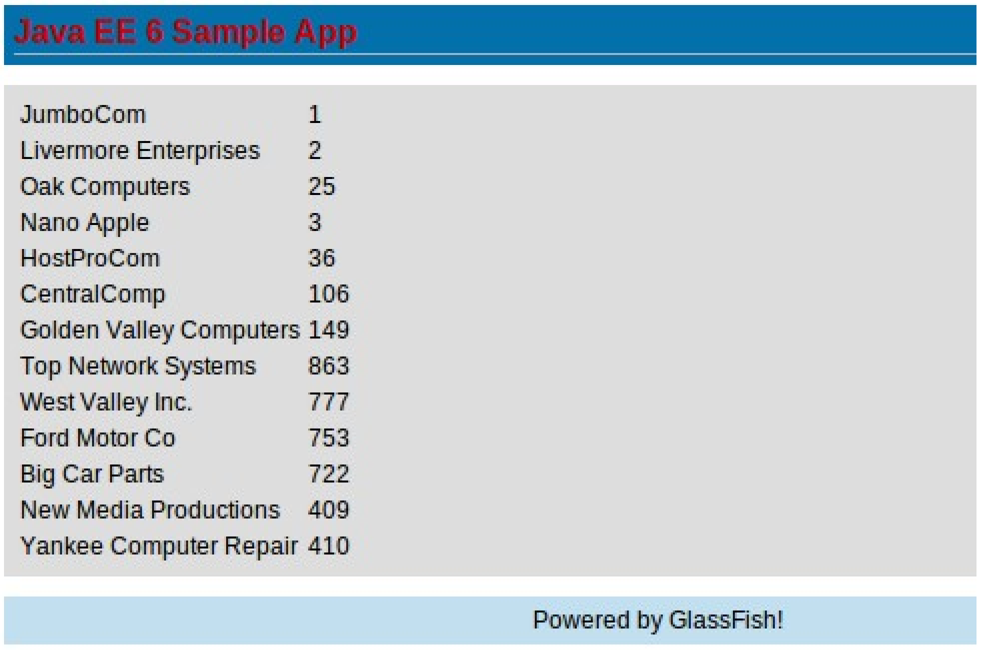
The “h” prefix used in the fragment is not referring to any namespace. This needs to be fixed by right-clicking on the yellow bulb as shown:



```
        <ui:composition template="./WEB-INF/template.
Undeclared component ine name="content">
                 <h:dataTable value="#{customerSession
 Add h="http://java.sun.com/jsf/html" library declaration
                 </h:dataTable>
```

and selecting the proposed fix.

8.8 Refreshing the page “http://localhost:8080/JavaEE6SampleApp/faces/index.xhtml” displays the result as shown.



Java EE 6 Sample App

JumboCom	1
Livermore Enterprises	2
Oak Computers	25
Nano Apple	3
HostProCom	36
CentralComp	106
Golden Valley Computers	149
Top Network Systems	863
West Valley Inc.	777
Ford Motor Co	753
Big Car Parts	722
New Media Productions	409
Yankee Computer Repair	410

Powered by GlassFish!

As you can see, the "top" and "bottom" sections are being inherited from the template and the "content" section is picked up from the "index.xhtml".

9.0 Expose JPA entities as a RESTful resource using JAX-RS

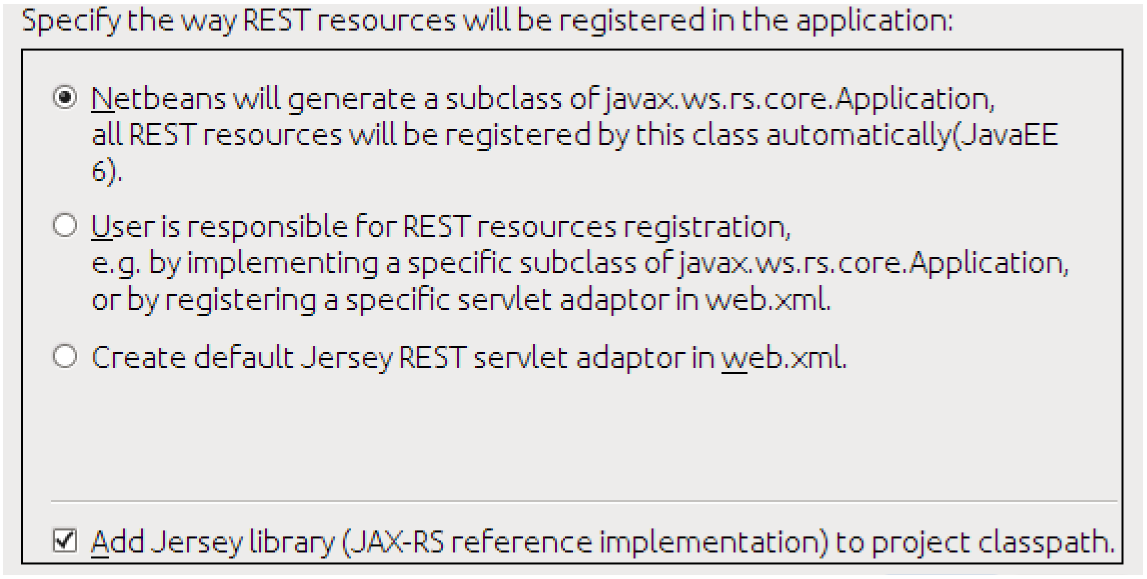
JAX-RS 1.1 defines a standard API to provide support for RESTful Web services in the Java platform. Just like other Java EE 6 technologies, any POJO can be easily converted into a RESTful resource by adding an annotation. JAX-RS 1.1 allows an EJB to be managed as a RESTful entity.

This section adds a method to our EJB which will then be invoked when the RESTful resource is accessed using the HTTP GET method.

9.1 In "CustomerSessionBean.java", add a @Path class-level annotation to promote EJB to a RESTful entity. The updated code looks like:

```
@Stateless
@LocalBean
@Named
@Path("/customers")
public class CustomerSessionBean {
```

The new annotation is highlighted in bold. The following window pops up as soon as you save this file:



Specify the way REST resources will be registered in the application:

[✓ Netbeans will generate a subclass of javax.ws.rs.core.Application, all REST resources will be registered by this class automatically(JavaEE 6).]

User is responsible for REST resources registration, e.g. by implementing a specific subclass of javax.ws.rs.core.Application, or by registering a specific servlet adaptor in web.xml.

Create default Jersey REST servlet adaptor in web.xml.

☑ Add Jersey library (JAX-RS reference implementation) to project classpath.

Take the default and click on “OK”.

Notice the following points:

- A new class that extends “javax.ws.rs.core.Application” is generated in the “org.netbeans.rest.application.config” package. This class registers the root URL for all the REST resources. The default base URL is “resources” as can be seen in the generated class.

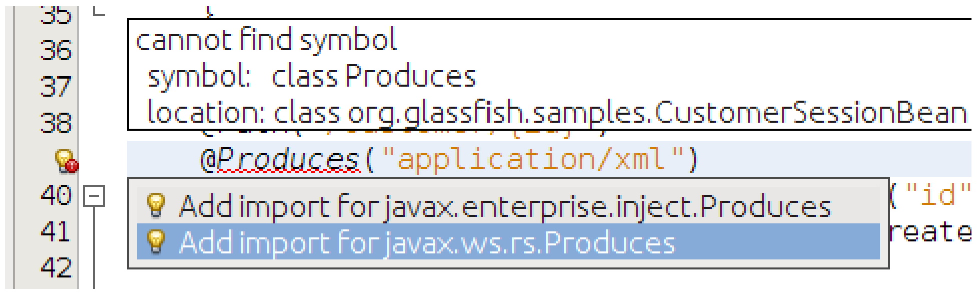
9.2 Add a new method in “CustomerSessionBean” and mark it such that this method is invoked whenever the REST resource is accessed using HTTP GET and the generated response is XML. Add the following method to “CustomerSessionBean”:

```
@GET
@Path("/customer/{id}")
@Produces("application/xml")
public Customer getCustomer(@PathParam("id")Integer id) {
    return
(Customer)em.createNamedQuery("Customer.findByCustomerId").setParameter("customerId
", id).getSingleResult();
}
```

Notice the following points:

- The “@GET” annotation ensures that this method is invoked when the resource is accessed using the HTTP GET method.
- The “@Path(“/customer/{id}”)” defines a sub-resource such that the resource defined by this method can be accessed at “customers/customer/{id}” where “id” is the variable part of the the URL and is mapped to the “id” parameter of the method.
- The “@Produces” annotation ensures that an XML representation is generated. This will work as @XmlRootElement annotation is already added to the generated entity.

Fix the imports by taking the default values for all except for “@Produces”. This annotation needs to be resolved from the “javax.ws.rs” package as shown.



```
cannot find symbol
 symbol:   class Produces
 location: class org.glassfish.samples.CustomerSessionBean
    @Produces("application/xml")
```

Add import for javax.enterprise.inject.Produces
Add import for javax.ws.rs.Produces

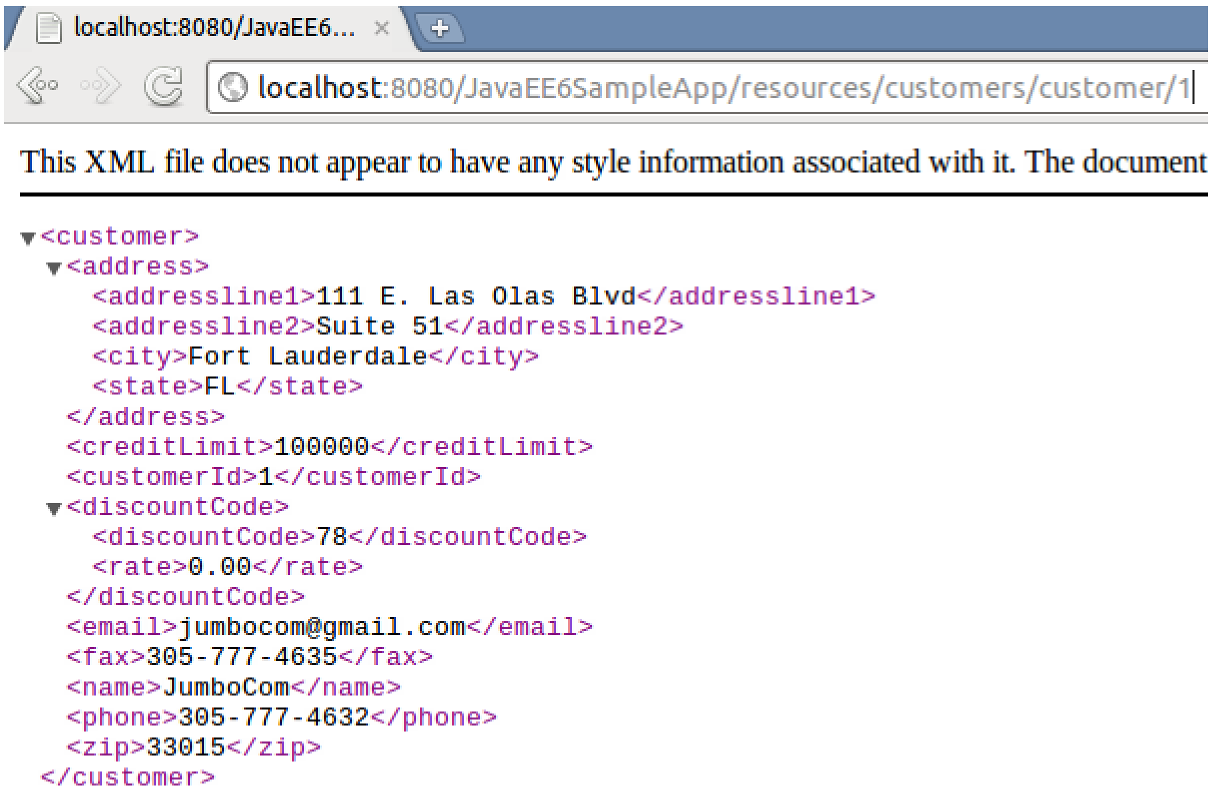
9.3 The RESTful resource is now accessible using the following format:

http://<HOST>:<PORT>/<CONTEXT-ROOT>/<ROOT-BASE-PATH>/<RESOURCE-PATH>/<SUB-RESOURCE-PATH>/<VARIABLE-PART>

In our case, the URL will look like:

http://localhost:8080/JavaEE6SampleApp/resources/customers/customer/{id}

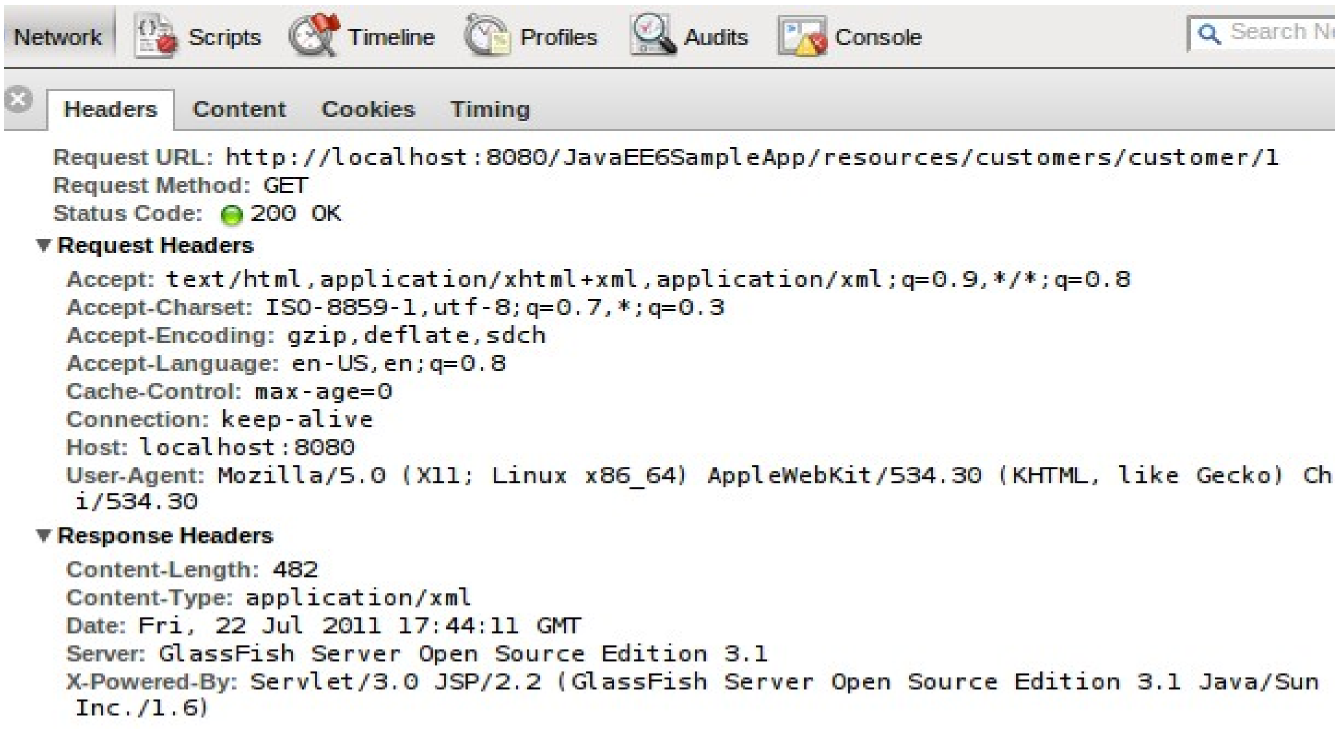
where {id} is the customer id shown in the JSF page earlier. So accessing "http://localhost:8080/JavaEE6SampleApp/resources/customers/customer/1" in a browser shows the output as:


localhost:8080/JavaEE6...
localhost:8080/JavaEE6SampleApp/resources/customers/customer/1

This XML file does not appear to have any style information associated with it. The document

```
▼<customer>
  ▼<address>
     <addressline1>111 E. Las Olas Blvd</addressline1>
     <addressline2>Suite 51</addressline2>
     <city>Fort Lauderdale</city>
     <state>FL</state>
   </address>
   <creditLimit>100000</creditLimit>
   <customerId>1</customerId>
  ▼<discountCode>
     <discountCode>78</discountCode>
     <rate>0.00</rate>
   </discountCode>
   <email>jumbocom@gmail.com</email>
   <fax>305-777-4635</fax>
   <name>JumboCom</name>
   <phone>305-777-4632</phone>
   <zip>33015</zip>
 </customer>
```

Invoking this URL in the browser is equivalent to making a GET request to the service. This can be verified by viewing the HTTP headers generated by the browsers as shown:



```
Request URL: http://localhost:8080/JavaEE6SampleApp/resources/customers/customer/1
Request Method: GET
Status Code: 200 OK
▼ Request Headers
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
  Accept-Encoding: gzip,deflate,sdch
  Accept-Language: en-US,en;q=0.8
  Cache-Control: max-age=0
  Connection: keep-alive
  Host: localhost:8080
  User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/534.30 (KHTML, like Gecko) Ch
   i/534.30
▼ Response Headers
  Content-Length: 482
  Content-Type: application/xml
  Date: Fri, 22 Jul 2011 17:44:11 GMT
  Server: GlassFish Server Open Source Edition 3.1
  X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1 Java/Sun
   Inc./1.6)
```

9.4 (OPTIONAL) Each resource can be represented in multiple formats. Change the “@Produces” annotation in the “CustomerSessionBean” from:

```
@Produces("application/xml")
```

to

```
@Produces({"application/xml", "application/json"})
```

This ensures that an XML or JSON representation of the resource can be generated. This can be easily verified by giving the following command (shown in bold) on a command-line:

arun@ArunUbuntu:~ **curl -H "Accept: application/json" http://localhost:8080/JavaEE6SampleApp/resources/customers/customer/1 -v**

```
* About to connect() to localhost port 8080 (#0)
*   Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /JavaEE6SampleApp/resources/customers/customer/1 HTTP/1.1
> User-Agent: curl/7.21.3 (x86_64-pc-linux-gnu) libcurl/7.21.3 OpenSSL/0.9.8o
zlib/1.2.3.4 libidn/1.18
> Host: localhost:8080
> Accept: application/json
>
```

```
< HTTP/1.1 200 OK
< X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1
Java/Sun Microsystems Inc./1.6)
< Server: GlassFish Server Open Source Edition 3.1
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Thu, 21 Jul 2011 05:43:41 GMT
<
* Connection #0 to host localhost left intact
* Closing connection #0
{"address":{"addressline1":"111 E. Las Olas Blvd","addressline2":"Suite
51","city":"Fort
Lauderdale","state":"FL"},"creditLimit":"100000","customerId":"1","discountCode":
{"discountCode":"78","rate":"0.00"},"email":"jumbocom@gmail.com","fax":"305-777-
4635","name":"JumboCom","phone":"305-777-4632","zip":"33015"}
```

Notice the following points:

- The command is shown in the bold letters
- Connection handshake is pre-pended “*”
- The HTTP request headers with “>”
- The HTTP response response with “<”
- The response in JSON format is at end of the message.

The “curl” utility for Windows-based machines can be downloaded from: http://curl.haxx.se/.

10.0 Conclusion

This hands-on lab created a typical 3-tier Java EE 6 application demonstrating the key features as:

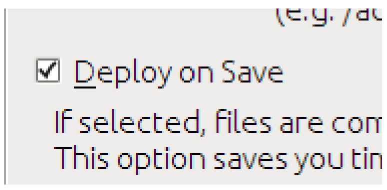
- Generate JPA Entities from the database table
- Refactor generated entities for a more intuitive O/R mapping
- Create an EJB for querying the database
- Create a Servlet for testing the EJB and displaying values from the database table
- Enable CDI and make the EJB EL-injectable
- Display the values in a JSF2/Facelets-based view
- Expose JPA entities as a RESTful resource by using JAX-RS

Hopefully this has raised your interest enough in trying out Java EE 6 applications using GlassFish and NetBeans.

Send us feedback at users@glassfish.java.net.

11.0 Troubleshooting

11.1 The project is getting deployed to GlassFish every time a file is saved. How can I disable/enable that feature ?



This feature can be enabled/disable per project basis from the Properties window. Right-click on the project, select “Properties”, choose “Run” categories and select/unselect the checkbox “Deploy on Save” to enable/disable this feature.

11.2 How can I show the SQL queries issued to the database ?

In NetBeans IDE, expand “Configuration Files”, edit “persistence.xml” and replace:

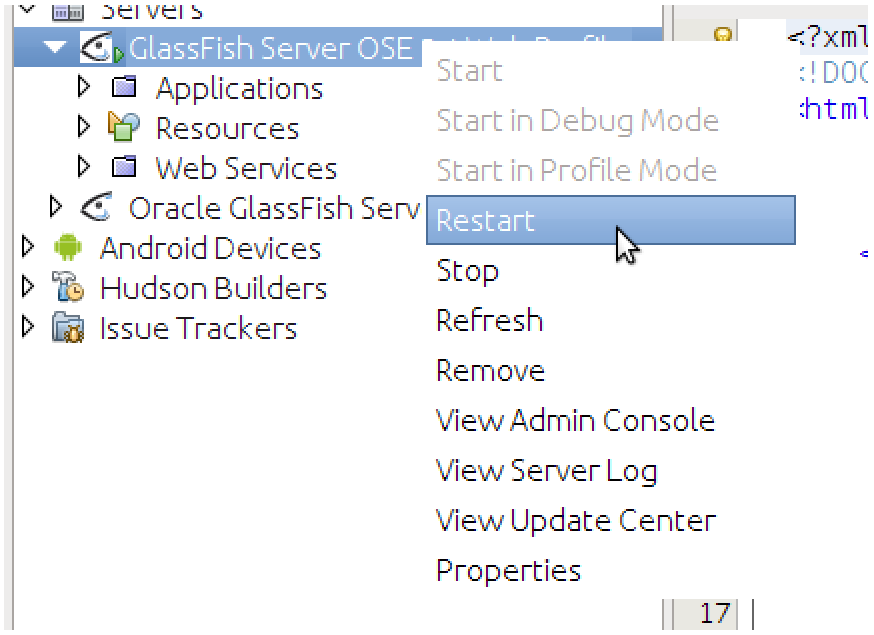
```
<properties/>
```

with

```
<properties>
    <property name="eclipselink.logging.level" value="FINE" />
</properties>
```

11.3 How can I start/stop/restart GlassFish from within the IDE ?

In the “Services” tab, right-click on “GlassFish Server 3.1”. All the commands to start, stop, and restart are available from the pop-up menu. The server log can be viewed by clicking on “View Server Log” and web-based administration console can be seen by clicking on “View Admin Console”.



12.0 Acknowledgments

This hands-on lab was graciously reviewed by the following GlassFish community members:

- Paulo Jeronimo
- Ajay Kemparaj
- Markus Eisele
- Santhosh Gandhe
- Filipe Portes
- Victor M. Ramirez

Thank you very much for taking time to provide the valuable feedback.

13.0 Completed Solutions

The completed solution is available as a NetBeans project at:

http://blogs.oracle.com/arungupta/resource/OTN-Developer-Day-2012-JavaEE6SampleApp.zip

Open the project in NetBeans, browse through the source code, and enjoy!