Java EE 6 Hands-on Lab
Feb 2012

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

Table of Contents

1.0 Introduction

The Java EE 6 platform allows you to write enterprise and web 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 Enterprise JavaBeans 3.1, JavaServer Faces 2.0, Java API for RESTful Web Services 1.1, Java Persistence API 2.0, Servlet 3.0, Contexts and Dependency Injection 1.0, and Bean Validation 1.0 make the platform more powerful by adding new functionality and yet keeping it 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, EJB 3.1, JAX-RS 1.1, Servlet 3, CDI 1.0, and Bean Validation 1.0. The application is developed using NetBeans 7 and deployed on Oracle WebLogic Server 12c.

The latest copy of this document is always available at https://blogs.oracle.com/arungupta/resource/javaee6-hol-weblogic.pdf.

1.1 Software Downloads

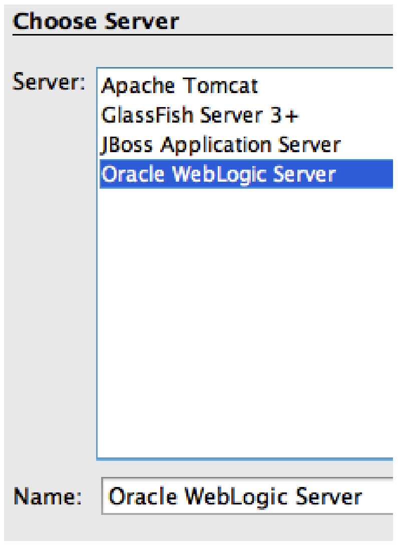
The following software need to be downloaded and installed:

1. JDK 6 or 7 from http://www.oracle.com/technetwork/java/javase/downloads/index.html.
2. NetBeans 7.1+ “All” or “JavaEE” version from http://netbeans.org/downloads/index.html.
3. Download and install Oracle WebLogic Server 12c from http://www.oracle.com/technetwork/middleware/fusion-middleware/downloads/index.html. The zip bundle is easiest to get started with.
4. Download Apache Derby database from http://db.apache.org/derby/releases/release-10.8.2.2.cgi and unzip.

1.2 Configure WebLogic 12c in NetBeans

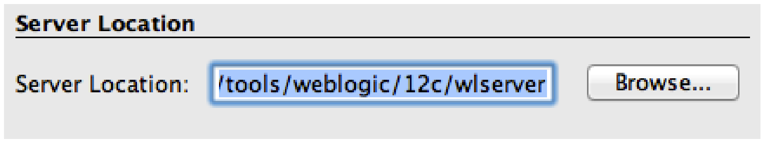
1. In the “Services” panel of NetBeans, right-click on “Servers” and select “Add Server ...”. Chose “Oracle WebLogic Server” and the default name as shown and click on “Next >”.

   If this is a new installation of NetBeans then you may see



a message “Activating Java Web and EE”.

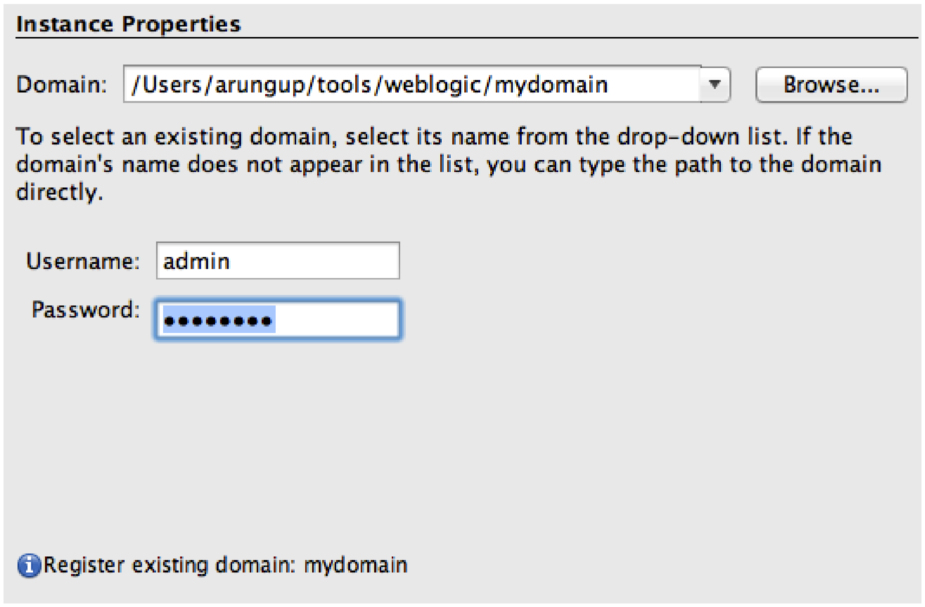
2. Click on the “Browse...” button and specify the location of WebLogic server as shown



and click on “Next >”.

Note, the directory name is “wlserver” and it exists inside the WebLogic installed directory.

3. If only one domain has been created (the default) then the directory location is automatically populated. Otherwise click on “Browse...” button and specify the location of the domain directory created during the installation process.



Domain: /Users/arungup/tools/weblogic/mydomain

To select an existing domain, select its name from the drop-down list. If the domain's name does not appear in the list, you can type the path to the domain directly.

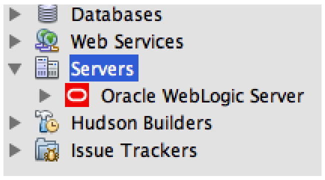
Username: admin

Password: •••••••••

Register existing domain: mydomain

Also specify the username and password for the domain and click on “Finish”

Note the informational message shown in the bottom-left corner of the dialog indicating successful registration of the domain. The completed installation will look like:

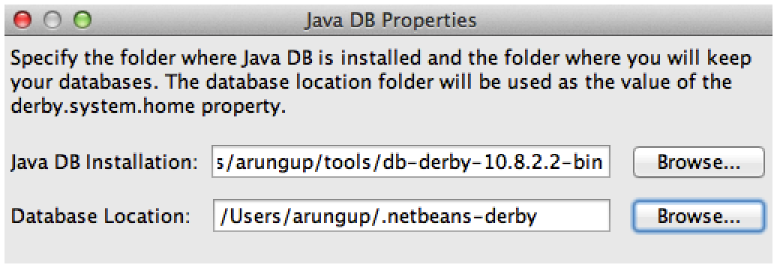


1.3 Configure JavaDB database in NetBeans

1. In the “Services” panel of NetBeans, expand “Databases”, right-click on “JavaDB”, and select “Properties...”.

2. Click on the first “Browse...” button and specify the location of the unzipped Apache Derby database.
3. In the second text box, specify the directory location as “<HOME>/.netbeans-derby”. Please make sure to specify the exact location of your home directory.

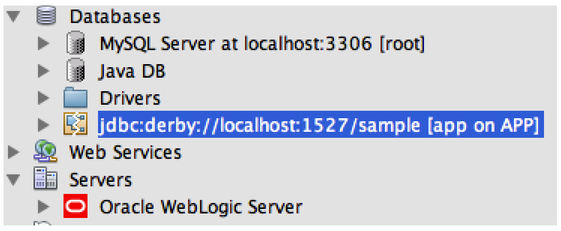
   After the values are entered the dialog box will look like as shown:



Specify the folder where Java DB is installed and the folder where you will keep your databases. The database location folder will be used as the value of the derby.system.home property.

Java DB Installation: s/arungup/tools/db-derby-10.8.2.2-bin

Database Location: /Users/arungup/.netbeans-derby


Databases
MySQL Server at localhost:3306 [root]
Java DB
Drivers
jdbc:derby://localhost:1527/sample [app on APP]
Web Services
Servers
Oracle WebLogic Server

Click on “OK”.

NetBeans creates the sample database and shows as following in the “Services” panel:

Right-click on the newly created database connection and select “Connect...”. This will start the JavaDB (or Apache Derby) database and expand the node. Expand “APP”, “Tables” to see the sample database structure.

1.4 Deploy Java DB JDBC driver to WebLogic

1. Stop WebLogic server if its already running by typing Ctrl+C in the console window that started the server.
2. Copy “derbyclient.jar” from the “lib” directory of the unzipped Apache Derby to “<DOMAIN>/lib” directory where <DOMAIN> is the location of your WebLogic domain directory.

2.0 Problem Statement

This hands-on lab builds a typical 3-tier Java EE 6 Web application that retrieves customer information from a database and displays it in a Web page. The application also allows new customers to be added to the database as well. The string-based and type-safe queries are used to query and add rows to the database. Each row in the database table is published as a RESTful resource and is then accessed programmatically. [TODO: Revisit what all patterns are covered] Typical design patterns required by a Web application like validation, caching, observer, partial page rendering, and cross-cutting concerns like logging are explained and implemented using different Java EE 6 technologies.

This application uses a sample database that comes pre-configured in NetBeans "All" and "Java EE" versions.

Disclaimer: This is a sample application and the code may not be following the best practices to prevent SQL injection, cross-side scripting attacks, escaping parameters, and other similar features expected of a robust enterprise application. This is intentional such as to stay focused on explaining the technology. Each of this topic can take its own hands-on lab and there is enough material available on the Internet. Its highly recommended to make sure that the code copied from this sample application is updated to meet those requirements.

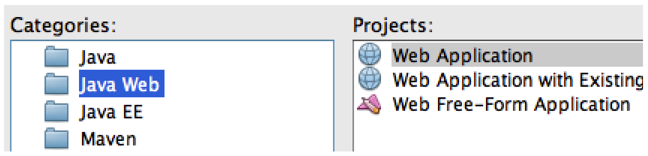
3.0 Build the Template Web Application

This section will build a template Web application using the NetBeans IDE.

3.1 In NetBeans IDE, create a new Web application by selecting the "File", "New Project".

3.2 Choose "Java Web" as Categories and "Web Application" as Projects.

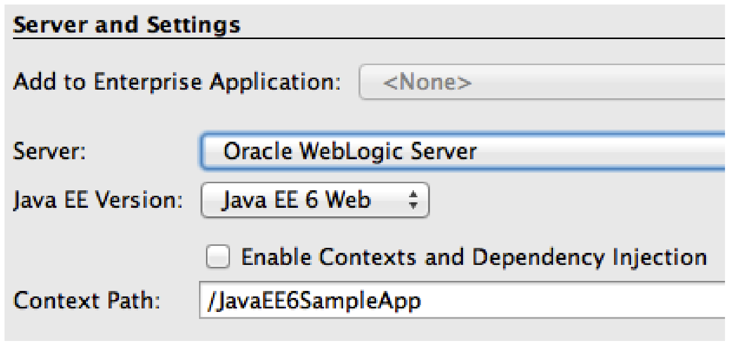
Click on "Next>".



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

3.4 Choose Oracle WebLogic Server as the Server.

The actual server name may differ based upon your preference. Ensure that the Java EE Version selected is "Java EE 6 Web" and click on "Finish".


Server and Settings

Add to Enterprise Application: <None>

Server: Oracle WebLogic Server

Java EE Version: Java EE 6 Web

Enable Contexts and Dependency Injection

Context Path: /JavaEE6SampleApp

This generates a template Web project.

3.5 Expand "Web Pages", "WEB-INF" and edit "weblogic.xml" by double-clicking on it. Change line number 9 from:

```
<enabled>true</enabled>
```

to

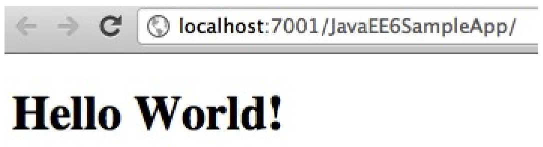
```
<enabled>false</enabled>
```

This ensures that “Fast Swap” is disabled for the development mode.

3.6 Right-click on the project and select “Run”. This will start the WebLogic server, deploy the Web application on the server, opens the default web browser, and displays “http://localhost:7001/JavaEE6SampleApp/index.jsp”.

The default page looks like as shown.

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



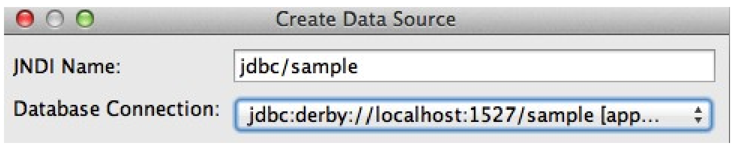
A display of this page ensures that the project is successfully created and WebLogic server has started successfully as well.

4.0 Generate JPA Entities from the Database Table

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 a string-based SQL-like syntax or a type-safe Criteria API. Both JPQL and Criteria API operate on the Java model instead of the database tables.

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

4.1 In NetBeans, right-click on the project and select “New”, “Entity Classes from Database...”. Choose “New Data Source...”.


Create Data Source

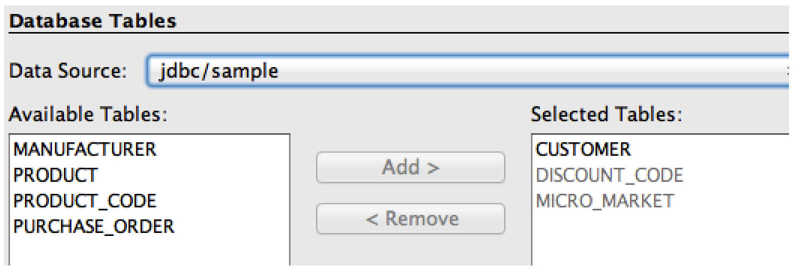
JNDI Name: jdbc/sample

Database Connection: jdbc:derby://localhost:1527/sample [app...

Specify the JNDI name as “jdbc/sample” and choose the pre-configured database connection as shown.

Click on “OK” and this will show all the tables from this data source.

4.2 Select “CUSTOMER” table from the “Available Tables” and click “Add>”. Notice that the “DISCOUNT_CODE” and “MICRO_MARKET” tables are automatically selected because of the foreign key references and the selected “Include Related Tables” checkbox (this is the default behavior).

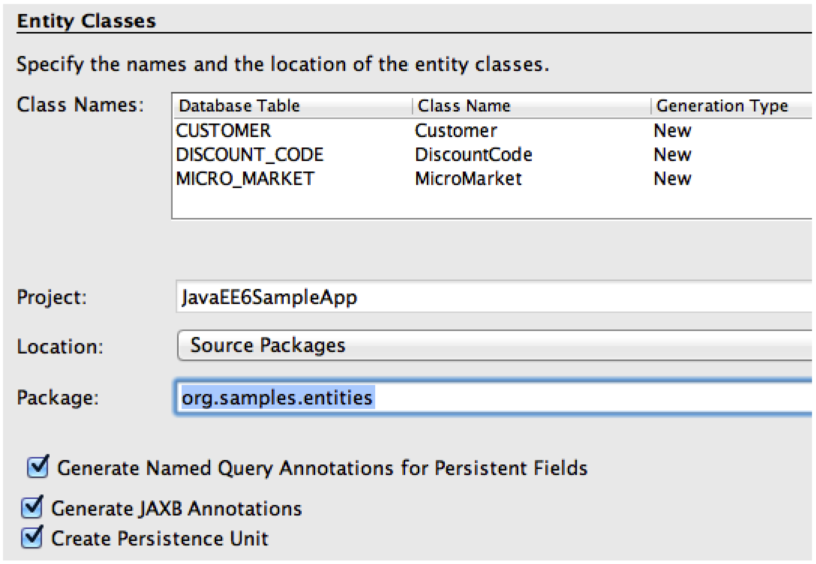


Data Source: jdbc/sample

Available Tables:	Selected Tables:
MANUFACTURER	CUSTOMER
PRODUCT	DISCOUNT_CODE
PRODUCT_CODE	MICRO_MARKET
PURCHASE_ORDER	

Click on “Next>”.

4.3 Enter the package name `org.samples.entities` as shown.



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: JavaEE6SampleApp

Location: Source Packages

Package: org.samples.entities

- [✓] Generate Named Query Annotations for Persistent Fields
- [✓] Generate JAXB Annotations
- [✓] Create Persistence Unit

The database table and the corresponding mapped class name is shown in the “Class Name” column and can be changed here, if needed.

Notice the following points:

- The first check box allows NetBeans to generate multiple `@NamedQuery` annotations on the JPA entity. These annotations provide pre-defined JPQL queries that can be used to query the database.
- 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 or JSON representation easily by JAXB. This feature is useful when the entities are published as a RESTful resource.
- The third check box generates the required Persistence Unit required by JPA.

Click on “Finish” to complete the entity generation.

In NetBeans, expand “Source Packages”, `org.samples.entities`, and double-click

`Customer.java`.

Notice the following points in the generated code:

- The generated class-level `@NamedQuery` annotations uses JPQL to define several queries. One of the queries is named "Customer.findAll" which maps to the JPQL that retrieves all rows from the database. There are several "findBy" queries, one for each field (which maps to a column in the table), that allows to query the data for that specific field. Additional queries may be added here providing a central location for all your query-related logic.
- The bean validation 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 `zip` and `discountCode` fields are marked with the `@JoinColumn` annotation creating a join with the appropriate table.

4.1 Walk-through the JPA Entities and Refactor to Simplify

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

4.1.1 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.

Replace the following code:

```
@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;
```

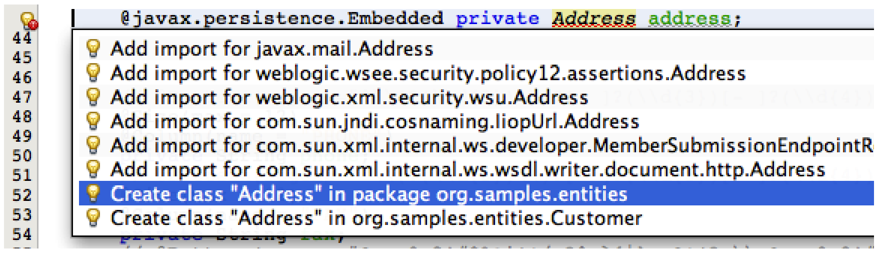
and

```
    @JoinColumn(name = "ZIP", referencedColumnName = "ZIP_CODE")
    @ManyToOne(optional = false)
    private MicroMarket zip;
```

with

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

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



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

Add import for javax.mail.Address
Add import for weblogic.wsee.security.policy12.assertions.Address
Add import for weblogic.xml.security.wsu.Address
Add import for com.sun.jndi.cosnaming.IiopUrl.Address
Add import for com.sun.xml.internal.ws.developer.MemberSubmissionEndpointR
Add import for com.sun.xml.internal.ws.wsdl.writer.document.http.Address
Create class "Address" in package org.samples.entities
Create class "Address" in org.samples.entities.Customer

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 later.
- The `@Embedded` annotation ensures that this field's value is an instance of an embeddable class.

4.1.2 Change `Address.java` so that it is a public class, annotate with `@Embeddable` such that it can be used as embeddable class, and also implement the `Serializable` interface. The updated class definition is shown:

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

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

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

4.1.4 Make the following changes in `Customer.java`:

i. Remove the getter/setter for the previously removed fields.

ii. Add a new getter/setter for “address” field as:

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

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

Here is one of the updated query:

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

iv. Change the implementation of the `toString` method as shown below:

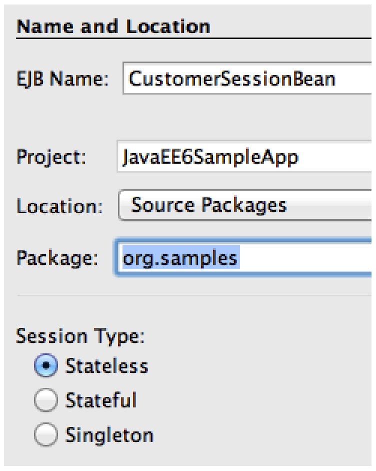
```
@Override
public String toString() {
    return name + "[" + customerId + "]";
}
```

This will ensure that the customer's name and unique identifier are printed as the default string representation.

5.0 Query the Database using EJB and Servlet

Java EE 6 provides 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 EJB to query the database and invoke it from the Servlet.



EJB Name: CustomerSessionBean

Project: JavaEE6SampleApp

Location: Source Packages

Package: org.samples

Session Type: [✓ Stateless] Stateful Singleton

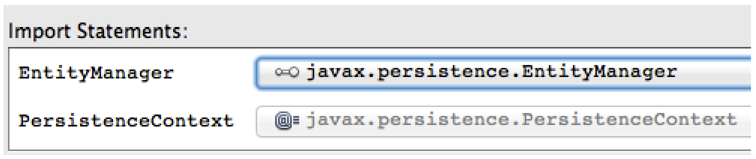
5.1 Create a new stateless EJB. Right-click on the project, select “New”, “Session Bean...”, specify the EJB Name as “CustomerSessionBean” as shown. Change the package name to “org.samples” and click on “Finish”.

This will create a stateless EJB. Notice the generated EJB does not implements an interface and this single class represents the EJB. This is a new feature of EJB 3.1.

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

```
@PersistenceContext
EntityManager em;
```

Resolve the imports by right-clicking on the editor pane and selecting “Fix Imports”. Make sure to change the package of EntityManager from the default to “javax.persistence.EntityManager” as shown.



Import Statements:

EntityManager	javax.persistence.EntityManager
PersistenceContext	javax.persistence.PersistenceContext

5.3 Add the following method in the EJB:

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

This code uses the injected `EntityManager` to query the database using a pre-generated `@NamedQuery` to retrieve all the customers.

Fix the imports by right-clicking in the editor pane and selecting “Fix 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.

In Java EE 6, a Servlet can be easily defined using a POJO, with a single annotation, and optional `web.xml` in most of the common cases.

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

🔍 Expand “Web Pages”, “WEB-INF” and notice that `web.xml` does not contain any information about the generated Servlet as that is now captured in the `@WebServlet` annotation.

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

5.5 Inject the EJB in the Servlet as

```
@EJB CustomerSessionBean bean;
```

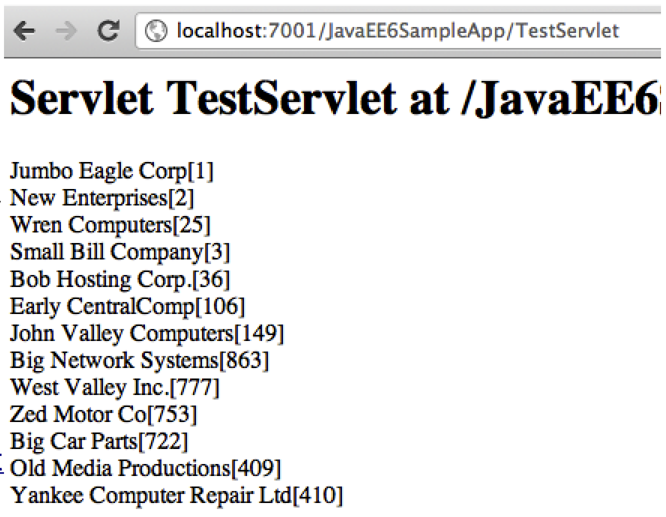
Invoke the `getCustomers` method in the `try` block. The updated `try` block looks like:

```
        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>");
            List<Customer> result = bean.getCustomers();
            for (Customer c : result)
                out.println(c + "<br>");
            out.println("</body>");
            out.println("</html>");
        } finally {
```

Resolve the imports by taking all default values.

Optionally, based upon the NetBeans version, you may have to un-comment the code in the `try` block of the `processRequest` method by removing the first and the last line in the `try` block.

5.6 Access "http://localhost:7001/JavaEE6SampleApp/TestServlet" in the browser and it looks like as shown.



localhost:7001/JavaEE6SampleApp/TestServlet

Servlet TestServlet at /JavaEE6

Jumbo Eagle Corp[1]
New Enterprises[2]
Wren Computers[25]
Small Bill Company[3]
Bob Hosting Corp.[36]
Early CentralComp[106]
John Valley Computers[149]
Big Network Systems[863]
West Valley Inc.[777]
Zed Motor Co[753]
Big Car Parts[722]
Old Media Productions[409]
Yankee Computer Repair Ltd[410]

6.0 Add Values to the Database using EJB, use Bean Validation Constraints

Bean Validation is a new specification added to the Java EE 6 platform. This specification defines an extensible validation constraints mechanism, with a pre-defined set of constraints, that can be specified on field, method, type, or annotations. It also defines a validation facility for the Java application developer.

The constraints can be specified on a JPA entity and the validation facility ensure these are met during the pre-persist, pre-update, and pre-remove lifecycle events. These constraints can also be used on “backing beans” for forms displayed by JSF. This ensures that the data entered in the form meets the validation criteria.

This section will explain how EJB can be used to add values to the database and Bean Validation constraints can be used to automatically validate the data.

6.1 Add a new method to the EJB as:

```
    public void addCustomer(Integer customerId,
            String name,
            Address address,
            String phone,
            String fax,
            String email,
            Integer creditLimit,
            DiscountCode discountCode) {
        Customer c = new Customer(customerId);
        c.setName(name);
        c.setAddress(address);
        c.setPhone(phone);
        c.setFax(fax);
        c.setCreditLimit(creditLimit);
        c.setDiscountCode(discountCode);
        em.persist(c);
    }
```

This method takes a few parameters and adds a new customer to the database by calling the `persist` method on the `EntityManager`.

Resolve the imports by taking default value for `DiscountCode` class but make sure `Address` class is imported from `org.samples.Address` instead of the default value.

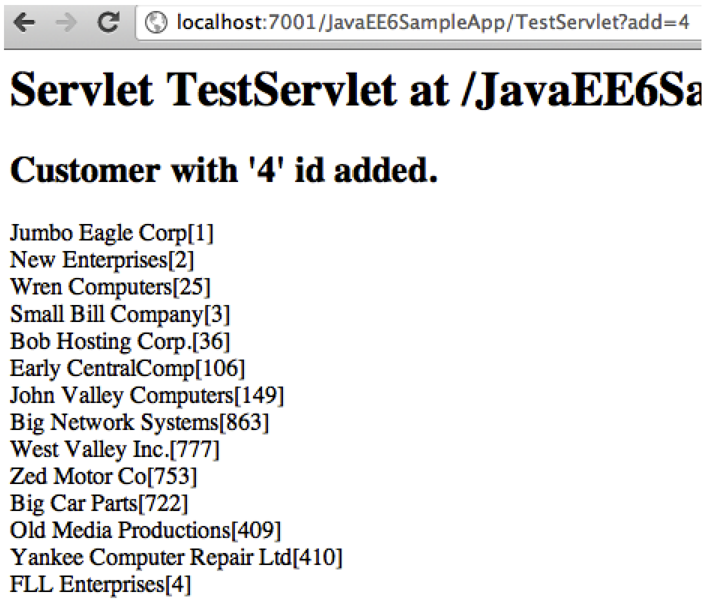
6.2 Add the following code to `TestServlet`'s `processRequest` method:

```
String id = request.getParameter("add");
if (id != null) {
       Address address = new Address();
       address.setAddressline1("4220, Network Circle");
       address.setCity("Santa Clara");
       address.setState("CA");
       MicroMarket zip = new MicroMarket();
       zip.setZipCode("95051");
       address.setZip(zip);
       bean.addCustomer(Integer.parseInt(id),
               "FLL Enterprises",
               address,
               "1234",
               "5678",
               "foo@bar.com",
               1000,
               new DiscountCode('H'));
       out.println("<h2>Customer with '" + Integer.parseInt(id) + "' id
added.</h2>");
}
```

This code fragment looks for the “add” parameter specified as part of the URL and then invokes the EJB's method to add a new customer to the database. The customer identifier is obtained as a value of the parameter and all other values are defaulted.

Once again, fix the imports by taking default values for `DiscountCode` and `MicroMarket` but make sure `Address` is imported from `org.samples.Address`.

6.3 Access “http://localhost:7001/JavaEE6SampleApp/TestServlet?add=4” in the browser and this will add a new customer (with id “4” in this case) to the database and displays the page as shown.


localhost:7001/JavaEE6SampleApp/TestServlet?add=4

Servlet TestServlet at /JavaEE6Sa

Customer with '4' id added.

```
Jumbo Eagle Corp[1]
New Enterprises[2]
Wren Computers[25]
Small Bill Company[3]
Bob Hosting Corp.[36]
Early CentralComp[106]
John Valley Computers[149]
Big Network Systems[863]
West Valley Inc.[777]
Zed Motor Co[753]
Big Car Parts[722]
Old Media Productions[409]
Yankee Computer Repair Ltd[410]
FLL Enterprises[4]
```

Notice the newly added customer is now shown in the list. The output may differ based upon where the code was added in the `processRequest` method.

The customer identifier is specified as part of the URL so its important to pick a number that does not already exist in the database.

6.4 One of the bean validation constraints mentioned in `Customer.java` is for the phone number (identified by the field phone) to be a maximum of 12 characters. Lets update the constraint such that it requires at least 7 characters to be specified as well. This can be done by changing the existing constraint from:

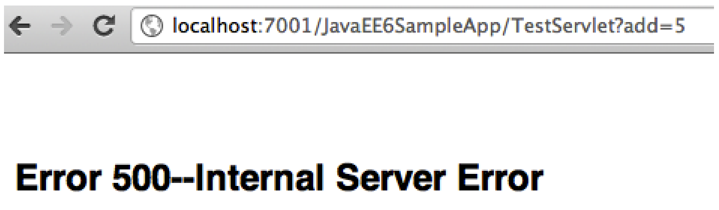
```
@Size(max = 12)
```

to

```
@Size(min = 7, max = 12)
```

Save the file and the project automatically gets re-deployed.

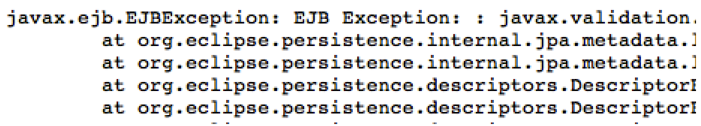
6.5 Access the URL "http://localhost:7001/JavaEE6SampleApp/TestServlet?add=5" in a browser and this tries to add a new customer to the database and shows the output as shown.



localhost:7001/JavaEE6SampleApp/TestServlet?add=5

Error 500--Internal Server Error

The WebLogic output log shows the following message:

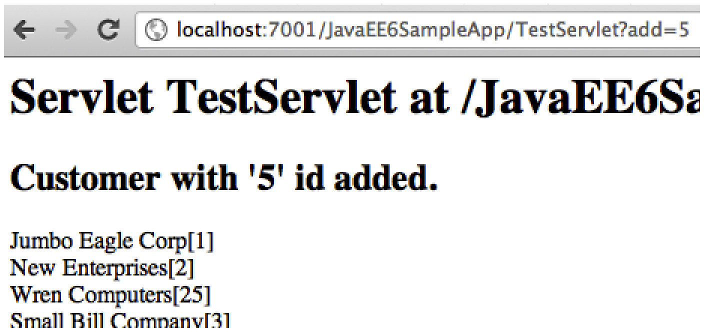


```
javax.ejb.EJBException: EJB Exception: : javax.validation.
        at org.eclipse.persistence.internal.jpa.metadata.
        at org.eclipse.persistence.internal.jpa.metadata.
        at org.eclipse.persistence.descriptors.Descriptor
        at org.eclipse.persistence.descriptors.Descriptor
```

```
Caused By: javax.validation.ConstraintViolationException: Bean Validation
constraint(s) violated while executing Automatic Bean Validation on callback
event:'prePersist'. Please refer to embedded ConstraintViolations for details.
```

This error message comes because the phone number is specified as a 4-digit number in the Servlet and so does not meet the validation criteria. This can be fixed by changing the phone number to specify 7 digits.

6.6 Edit `TestServlet.java`, change the phone number from "1234" to "1234567", and save the file. And access the URL "http://localhost:7001/JavaEE6SampleApp/TestServlet?add=5" again to see the output as shown.



localhost:7001/JavaEE6SampleApp/TestServlet?add=5

Servlet TestServlet at /JavaEE6Sa

Customer with '5' id added.

Jumbo Eagle Corp[1]
New Enterprises[2]
Wren Computers[25]
Small Bill Company[3]

The customer is now successfully added.

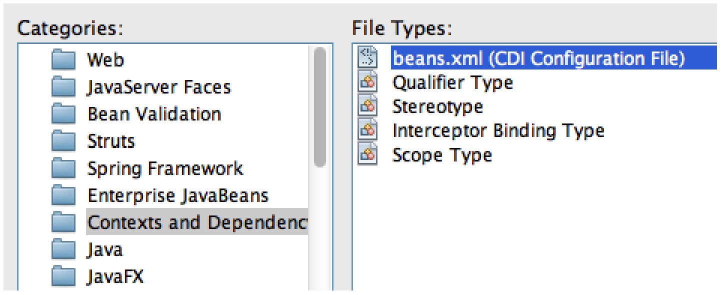
7.0 MVC using JSF2/Facelets-based view

JavaServer Faces 2 allows Facelets to be used as the 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 ensures that MVC architecture recommended by JSF can be easily achieved. The model is provided by the JPA entity, the view is served by JSF2, and controller is an EJB. Even though JSPs can still be used as view templates but Facelets are highly recommended with JSF2.

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 the JSF pages. This eliminates the need for any “glue code”, such as *JSF managed beans*, and there by further simplifying the Java EE platform.

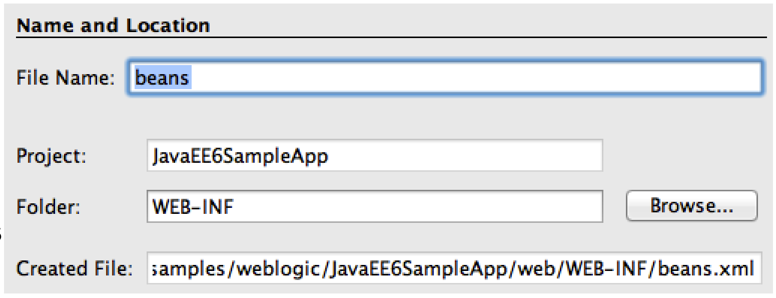
This section adds JSF support to the application and display the list of customers in a Facelets-based view. The EJB methods will be invoked in the Expression Language to access the database.

7.1 The CDI specification require `beans.xml` in the WEB-INF directory to enable injection of beans within a WAR file. This will allow to use the EJB in an Expression Language (EL), such as `.xhtml` pages that will be created later.



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

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


Name and Location

File Name: beans

Project: JavaEE6SampleApp

Folder: WEB-INF

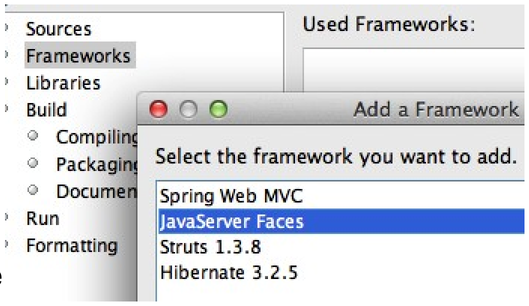
Created File: samples/weblogic/JavaEE6SampleApp/web/WEB-INF/beans.xml

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 (explained later) and ensures that the EJB can now be injected in an expression language.

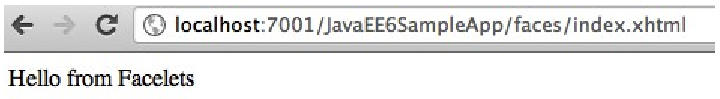
7.3 Right-click on the project, select “Properties”, “Frameworks”, “Add...”, and select “JavaServer Faces” as shown. Click on “OK”, take the default configuration, and click on “OK” again.



With JSF 2.0 implementation in WebLogic, if any JSF-specific annotations are used in the application 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.

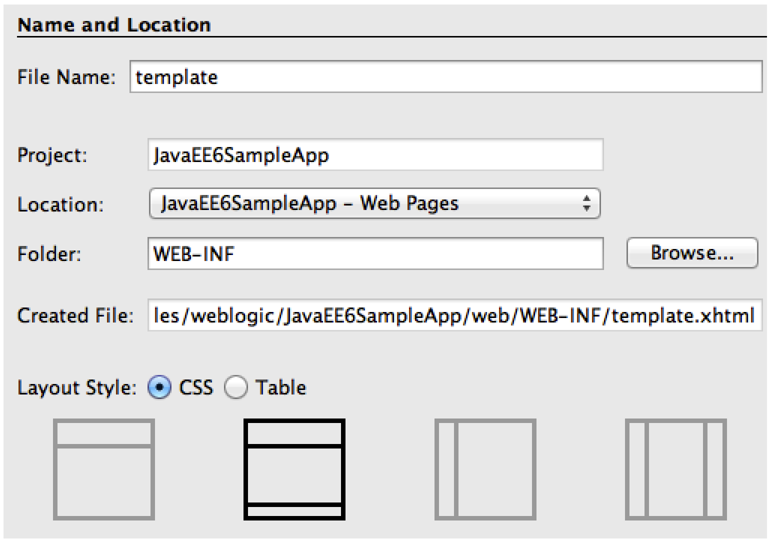
Adding the framework updates `web.xml` which registers the "FaceServlet" using the "/faces" URL pattern. It also generates `index.xhtml` page which can be verified by viewing "http://localhost:7001/JavaEE6SampleApp/faces/index.xhtml" with the output as shown.


localhost:7001/JavaEE6SampleApp/faces/index.xhtml

Hello from Facelets

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

Right-click on the project, select "New", "Other", "JavaServer Faces", "Facelets Template...", change the name to "template", click on "Browse...", select the "WEB-INF" folder, and select the template as shown. Click on "Finish".


Name and Location

File Name: template

Project: JavaEE6SampleApp

Location: JavaEE6SampleApp – Web Pages

Folder: WEB-INF

Created File: les/weblogic/JavaEE6SampleApp/web/WEB-INF/template.xhtml

Layout Style: [✓ CSS] Table

Notice the following points:

- This generates `WEB-INF/template.xhtml` and two stylesheets in the `resources/css` folder. NetBeans contains a pre-defined set of templates and additional ones can be created using XHTML/CSS.
- The `template.xhtml` page contains three `<div>`s with `<ui:insert>` named "top", "content", and "bottom". These are placeholders for adding content to provide a consistent look-and-feel.
- Its a recommended practice to keep the template pages in the `WEB-INF` folder to restrict their visibility to the web application.

7.5 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:

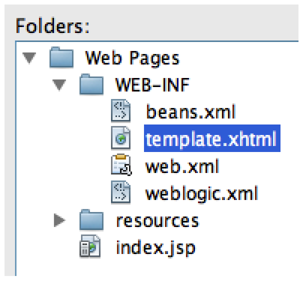
```
<center>Powered by WebLogic!</center>
```

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.

7.6 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.

First, delete `index.xhtml` by right-clicking and selecting “Delete”.

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



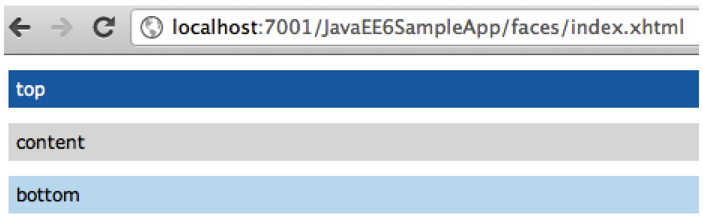
Enter the file name as “index”, choose the “Browse...” button next to “Template:” text box, select “template.xhtml” as shown, and click on “Select File”.

Click on “Finish”.

Notice the following points:

- The generated page, `index.xhtml`, 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, instead of `<ui:insert>` in the template, 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.

7.7 Refresh “http://localhost:7001/JavaEE6SampleApp/faces/index.xhtml” to see the output as:



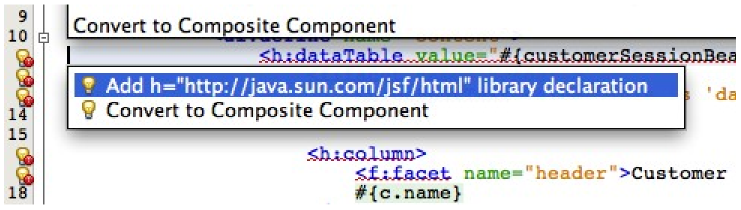
The output displays three sections from the `index.xhtml` file as generated by the NetBeans wizard.

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

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

```
<h:dataTable value="#{customerSessionBean.customers}" var="c">
    <f:facet name="header">
        <h:outputText value="Customer Table" />
    </f:facet>

    <h:column>
        <f:facet name="header">Customer Name</f:facet>
        #{c.name}
    </h:column>
    <h:column>
        <f:facet name="header">Customer ID</f:facet>
        #{c.customerId}
    </h:column>
</h:dataTable>
```

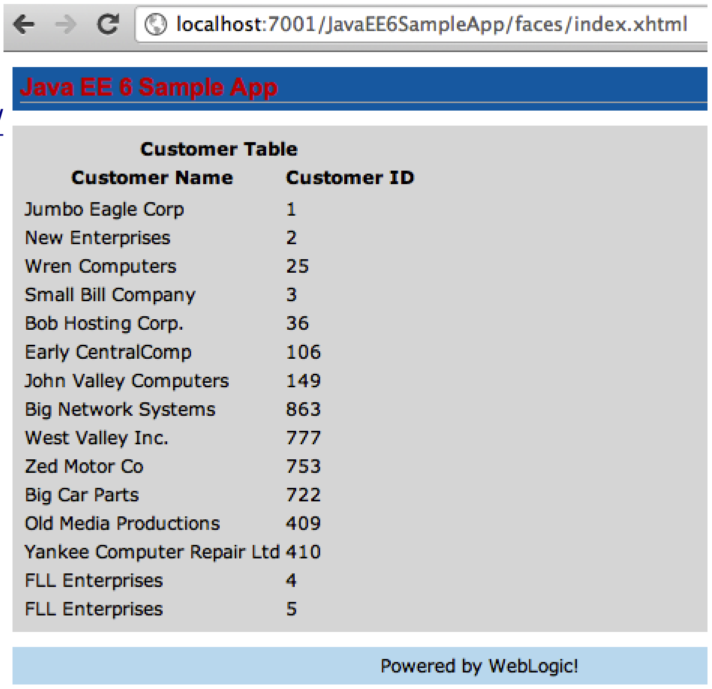
This JSF fragment injects `CustomerSessionBean` into the expression language, invokes its `getCustomers` method, iterates through all the values, and then displays the name and id of each customer. It also displays table and column headers.



```
Convert to Composite Component
        <h:dataTable value="#{customerSessionBea
 Add h="http://java.sun.com/jsf/html" library declaration
 Convert to Composite Component
            <h:column>
                <f:facet name="header">Customer
                #{c.name}
```

The `f` and `h` prefix used in the fragment is not referring to any namespace. This needs to be fixed by clicking on the yellow bulb as shown. Select the proposed fix. Repeat this fix for `f` prefix as well.

7.10 Refreshing the page “http://localhost:7001/JavaEE6SampleApp/faces/index.xhtml” displays the result as shown.



localhost:7001/JavaEE6SampleApp/faces/index.xhtml

Java EE 6 Sample App

Customer Table

Customer Name	Customer ID
Jumbo Eagle Corp	1
New Enterprises	2
Wren Computers	25
Small Bill Company	3
Bob Hosting Corp.	36
Early CentralComp	106
John Valley Computers	149
Big Network Systems	863
West Valley Inc.	777
Zed Motor Co	753
Big Car Parts	722
Old Media Productions	409
Yankee Computer Repair Ltd	410
FLL Enterprises	4
FLL Enterprises	5

Powered by WebLogic!

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

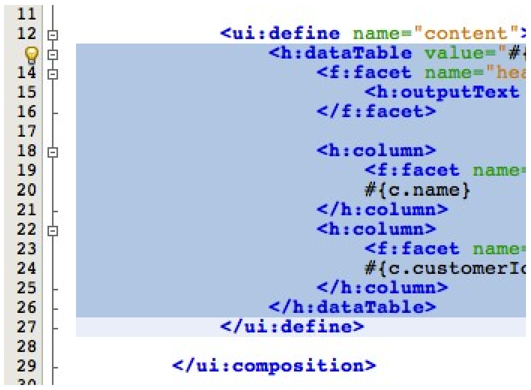
Additional pages can be added to this web application using the same template and thus providing consistent look-and-feel.

8.0 DRY using JSF2 Composite Component

The JSF2 specification defines a *composite component* as a component that consists of one or more JSF components defined in a Facelet markup file that resides inside a resource library. The composite component is defined in the *defining page* and used in the *using page*. The defining page defines the metadata (or parameters) using `<cc:interface>` and implementation using `<cc:implementation>` where `cc` is the prefix for the `http://java.sun.com/jsf/composite` namespace.

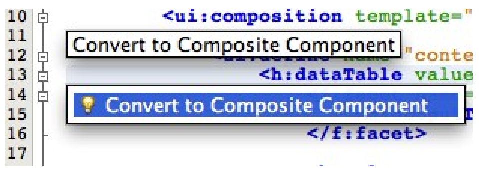
This section will convert a Facelets markup fragment from `index.xhtml` into a composite component.

8.1 In `index.xhtml`, select the `<h:dataTable>` fragment as shown.



```
        <ui:define name="content">
            <h:dataTable value="#
                <f:facet name="hea
                    <h:outputText
                </f:facet>

                <h:column>
                    <f:facet name=
                    #{c.name}
                </h:column>
                <h:column>
                    <f:facet name=
                    #{c.customerIc
                </h:column>
            </h:dataTable>
        </ui:define>

    </ui:composition>
```

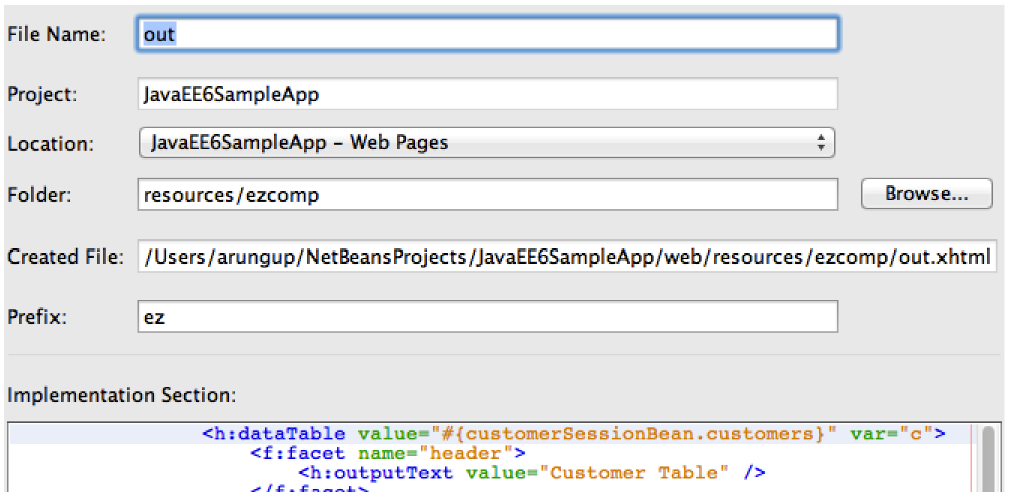
Click on the yellow bulb and select the hint.



```
                <ui:composition template="
                                         "conte
                    <h:dataTable value
                        </f:facet>
```

Convert to Composite Component
Convert to Composite Component

8.2 Take the defaults as shown.



File Name: out

Project: JavaEE6SampleApp

Location: JavaEE6SampleApp – Web Pages

Folder: resources/ezcomp

Created File: /Users/arungup/NetBeansProjects/JavaEE6SampleApp/web/resources/ezcomp/out.xhtml

Prefix: ez

Implementation Section:

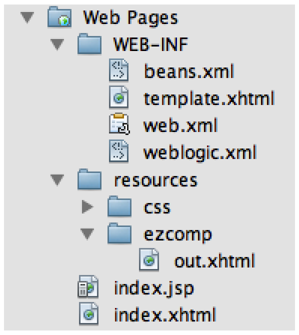
```
            <h:dataTable value="#{customerSessionBean.customers}" var="c">
                <f:facet name="header">
                    <h:outputText value="Customer Table" />
```

and click on “Finish”.

The updated `index.xhtml`, the using page, looks like:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ez="http://java.sun.com/jsf/composite/ezcomp">

    <body>

        <ui:composition template="./WEB-INF/template.xhtml">

                <ui:define name="content">
                    <ez:out/>
            </ui:define>

        </ui:composition>
```

It has a new namespace `http://java.sun.com/jsf/composite/ezcomp` with a prefix `ez`. This namespace is the standard namespace defined by the JSF2 specification suffixed with `ezcomp`. This `ezcomp` is a new directory created in the standard `resources` directory defined by the JSF2 specification. The tag name for the new composite component is the same as the defining page file name. The updated directory structure looks like as shown.


Web Pages
WEB-INF
beans.xml
template.xhtml
web.xml
weblogic.xml
resources
css
ezcomp
out.xhtml
index.jsp
index.xhtml

The entire fragment inside `<h:dataTable>` is replaced with `<ez:out>`. This allows a collection of JSF components to be extracted into a composite component following the Dont-Repeat-Yourself (DRY) design pattern and enables to use `<ez:out>` for printing the list of customers instead of repeating that entire code fragment. It also allows developers to author new components without any Java code or XML configuration.

The defining page `out.xhtml` looks like:

```
<!-- INTERFACE -->
    <cc:interface>
    </cc:interface>

    <!-- IMPLEMENTATION -->
    <cc:implementation>
        <h:dataTable value="#{customerSessionBean.customers}" var="c">
            <f:facet name="header">
                <h:outputText value="Customer Table" />
            </f:facet>

            <h:column>
                <f:facet name="header">Customer Name</f:facet>
                #{c.name}
            </h:column>
            <h:column>
```

```
                <f:facet name="header">Customer ID</f:facet>
                #{c.customerId}
            </h:column>
        </h:dataTable>

    </cc:implementation>
</html>
```

The `<cc:interface>` defines metadata that describe the characteristics of this component, such as supported attributes, facets, and even attach points for event listeners. `<cc:implementation>` contains the markup substituted for the composite component, `<h:dataTable>` fragment from `index.xhtml` in this case.

The `<cc:interface>` is generated in the page but is empty and may be made option in a subsequent release of the JSF specification.

Refreshing "http://localhost:7001/JavaEE6SampleApp/faces/index.xhtml" shows the same result as before.

Note: In some unknown cases you may have to deploy the project explicitly otherwise h namespace prefix in `out.xhtml` is not resolved correctly.

9.0 Declarative Ajax to Retrieve Partial Values from the database

The JSF2 specification provides standard support for Ajax. It exposes a standard JavaScript API primarily targeted for use by frameworks as well as the JSF implementation itself. It also provides a declarative approach that is more convenient for page authors.

This section will create a new Facelet page to retrieve the list of customers from the database as their name is typed in the input box using the declarative `<f:ajax>`. It will also show how only some elements of the page can be refreshed, aka partial page refresh.

9.1 Create a new Facelets Client Page by right-clicking on the project, select "New", "Facelets Template Client ...", give the file name as "list", select the template "WEB-INF/template.xhtml", take all other defaults and click on "Finish".

9.2 Remove the `<ui:define>` elements with name "top and "bottom".

9.3 Replace the code in `<ui:define>` element with "content" name to:

```
<h:form>
        <h:inputText value="#{customerName.value}">
                <f:ajax event="keyup" render="custTable"
                        listener="#{customerSessionBean.matchCustomers}"/>
        </h:inputText>
        <h:dataTable var="c" value="#{customerSessionBean.cust}" id="custTable">
                <h:column>#{c}</h:column>
        </h:dataTable>
</h:form>
```

This code displays an input text box and binds it to the “value” property of “CustomerName” bean (to be defined later). The `<f:ajax>` tag attaches Ajax behavior to the input text box. The meaning of each attribute is explained in the table below:

Attribute	Purpose
`event`	Event of the applied component for which the Ajax action is fired, `onkeyup` in this case.
`render`	List of components that will be rendered after the Ajax action is complete. This enables “partial page rendering”.
`listener`	Method listening for the Ajax event to be fired. This method must take `AjaxBehaviorEvent` as the only parameter and return a `void`.

The `<h:dataTable>` is the placeholder for displaying the list of customers.

Make sure to fix the namespace/prefix mapping for `h` and `f` by clicking on the yellow bulb in the left side bar.

Note that the partial view is still rendered on the server. The updated portion is sent to the browser where the DOM is updated.

9.4 Add a new Java class by right-clicking on the project, selecting “New”, “Java Class...”, give the name as “CustomerName”. Change the code to:

```
@Model
public class CustomerName {
    private String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

The `@Model` annotation is a CDI stereotype that is a simplified way of saying that the bean is both `@Named` and `@RequestScoped`. A handful of stereotypes are already pre-defined in the CDI specification and new ones can be easily defined.

Its important to mark this bean request-scoped otherwise a new instance of this bean is created for every injection request.

9.5 Add the following method to `CustomerSessionBean.java`:

```
public void matchCustomers(AjaxBehaviorEvent evt) {
    CriteriaBuilder builder = em.getCriteriaBuilder();
    CriteriaQuery criteria = builder.createQuery(Customer.class);
    // FROM clause
    Root root = criteria.from(Customer.class);

    // SELECT clause
    criteria.select(root);

    // WHERE clause
    Predicate condition = builder.like(root.get(Customer_.name),
            "%" + name.getValue() + "%");
    criteria.where(condition);

    // FIRE query
    TypedQuery query = em.createQuery(criteria);

    // PRINT result
    cust = query.getResultList();

    return;
}
```

This method uses JPA2 `CriteriaQuery` API, a type-safe API to author queries. Note the following points:

- The method takes `AjaxBehaviorEvent` as the parameter as this is requirement for the `listener` attribute of `<f:ajax>`.
- The JPA2 specification defines a canonical metamodel. NetBeans uses “EclipseLink Canonical Metamodel Generator” to generate these metamodel classes when the project is built. This is pre-configured in “Project Properties”, “Libraries”, “Processor”.
- The `Criteria` query specifies a WHERE clause using the JPA2 metamodel. The clause narrows down the search results where the customer's name consists of the string mentioned in the input text box.

Fix the imports by taking all defaults.

9.6 Inject the customer name as:

```
@Inject CustomerName name;
```

And resolve the import once again by taking defaults.

9.7 Add a new field as:

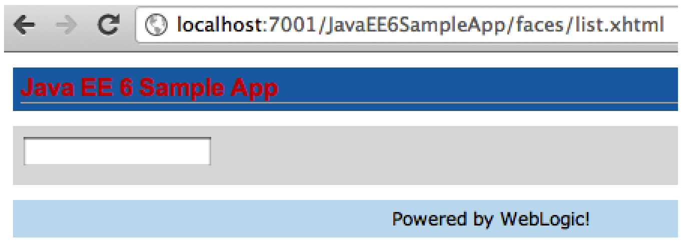
```
private List<Customer> cust;
```

This field is used to return the partial list of customers meeting the criteria. Add a getter for it as:

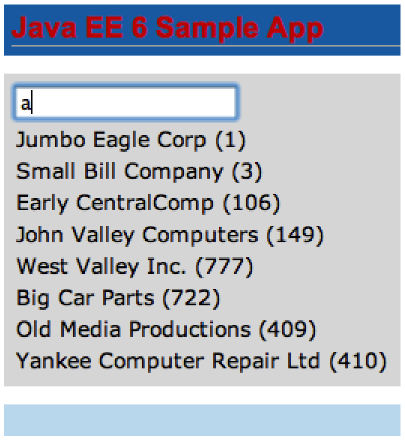
```
    public List<Customer> getCust() {
        return cust;
    }
```

9.8 Save all the files and open "http://localhost:7001/JavaEE6SampleApp/faces/list.xhtml" in the browser. The default output looks like as shown.

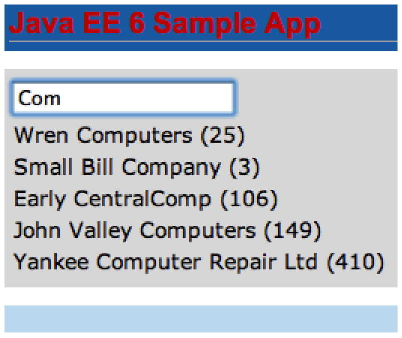
This is because the listener method in EJB is invoked on the `keyup` event. If no criteria is specified in the text, then no event is fired, the business method is not invoked, and the empty data table is shown.



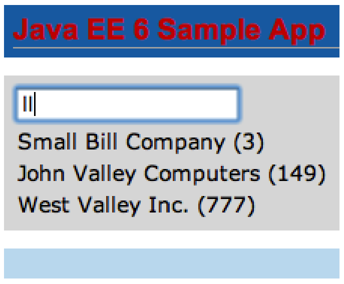
If you type "a" in the text box then the list of customers is narrowed down to the names that contain "a" as shown. Some other sample results are shown as well.



```
Jumbo Eagle Corp (1)
Small Bill Company (3)
Early CentralComp (106)
John Valley Computers (149)
West Valley Inc. (777)
Big Car Parts (722)
Old Media Productions (409)
Yankee Computer Repair Ltd (410)
```



```
Wren Computers (25)
Small Bill Company (3)
Early CentralComp (106)
John Valley Computers (149)
Yankee Computer Repair Ltd (410)
```



```
Small Bill Company (3)
John Valley Computers (149)
West Valley Inc. (777)
```

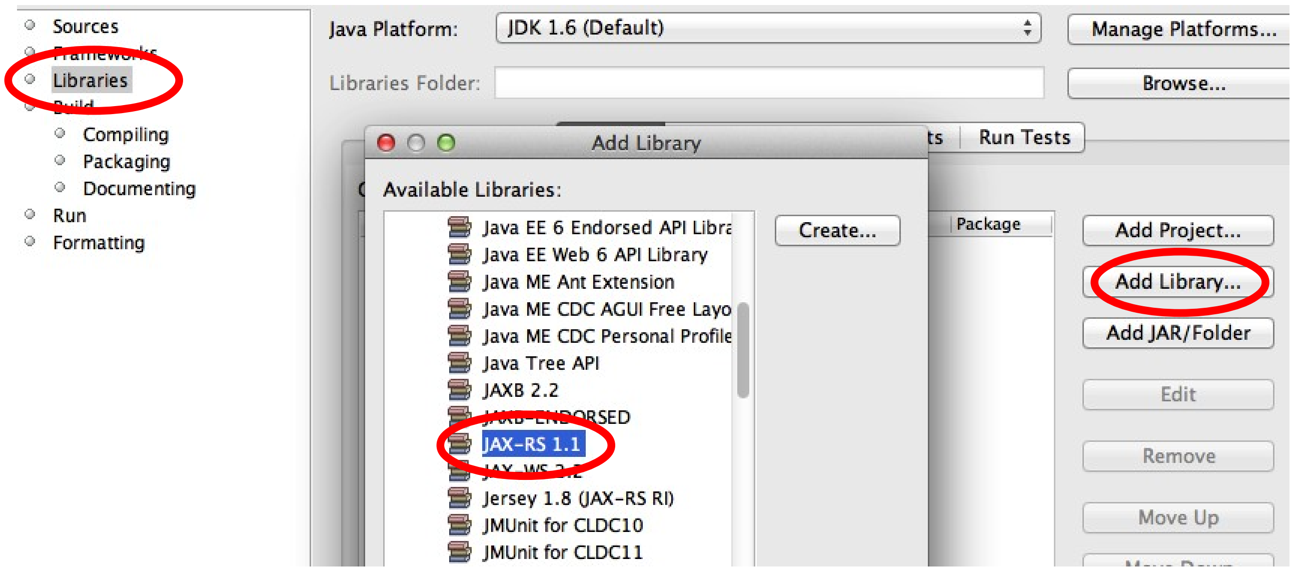
Notice, all of this is only refreshing the `<h:dataTable>` and there by showing "partial page refresh" or "partial page rendering".

10.0 Publish EJB as a RESTful Resource using JAX-RS

The JAX-RS 1.1 specification 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 the `@Path` annotation. JAX-RS 1.1 allows an EJB to be published as a RESTful entity.

This section publishes an EJB as a RESTful resource. A new method is added to the EJB which will be invoked when the RESTful resource is accessed using the HTTP GET method.

10.1 Add the JAX-RS library to the project so that all the classes can be correctly resolved. Right-click on the project, select “Properties”, “Libraries”, “Add Library...”, scroll down and select “JAX-RS 1.1” as shown.



Click on “OK”.

There is no need to package the library so unselect “Package” and then click on “OK” again.

10.1 In `CustomerSessionBean.java`, add a class-level `@Path` annotation to publish EJB as a RESTful entity. The updated code looks like:

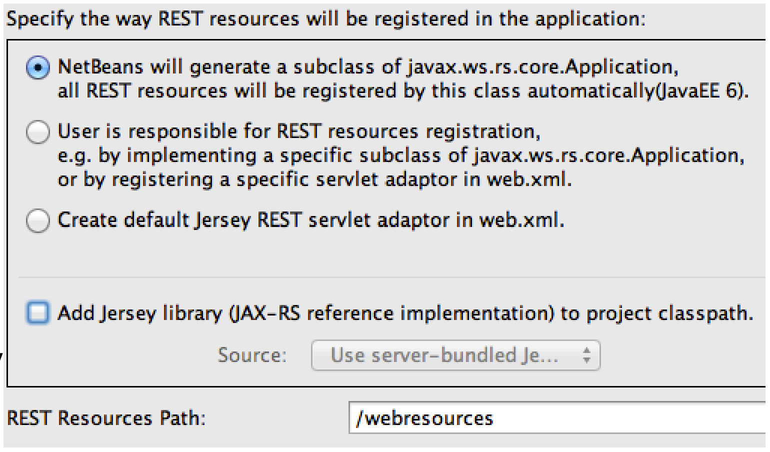
```
@Stateless
@LocalBean
@Named
@Path("/customers")
```

```
public class CustomerSessionBean {
```

The new annotation is highlighted in bold. Resolve the imports by clicking on the yellow bulb and selecting `javax.ws.rs.Path`, this is the default as well.

The window shown on the right pops up as soon as you save this file.

The "REST Resources Path" is the base URI for servicing all requests for RESTful resources in this web application. The default value of "/resources" is already used by the `.xhtml` templates and CSS used by JSF. So change the value to "/webresources".



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.

Source: Use server-bundled Je...

REST Resources Path: /webresources

Also there is no need to bundle Jersey (implementation of JAX-RS) libraries with the project so unselect that check box as shown.

Click on "OK".

Notice that a new class that extends `javax.ws.rs.core.Application` is generated in the `org.netbeans.rest.application.config` package. This class registers the base URI for all the RESTful resources provided by `@Path`. The updated base URI is "webresources" as can be seen in the generated class.

10.2 Add the following method to `CustomerSessionBean.java`:

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

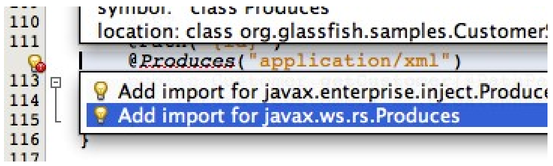
This method is invoked whenever the REST resource is accessed using HTTP GET and the expected response type is XML.

Notice the following points:

- The `@GET` annotation ensures that this method is invoked when the resource is accessed using the HTTP GET method.

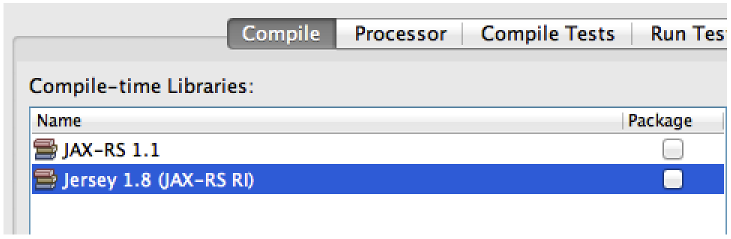
- The `@Path("{id}")` defines a sub-resource such that the resource defined by this method can be accessed at `customers/{id}` where "id" is the variable part of the URI and is mapped to the "id" parameter of the method as defined by `@PathParam` annotation.
- The `@Produces` annotation ensures that 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.



```
location: class org.glassfish.samples.Customer
    @Produces("application/xml")
Add import for javax.enterprise.inject.Produc
Add import for javax.ws.rs.Produces
```

Note: If you are using NetBeans 7.1, then because of a bug the Jersey libraries are bundled even though they've been unselected earlier. This can be worked around by right-clicking on the project, selecting "Properties", "Libraries", and unselecting "Jersey 1.8 (JAX-RS RI)" as shown.



Compile | Processor | Compile Tests | Run Tes

Compile-time Libraries:

Name	Package
JAX-RS 1.1	☐
Jersey 1.8 (JAX-RS RI)	☐

Click on "OK".

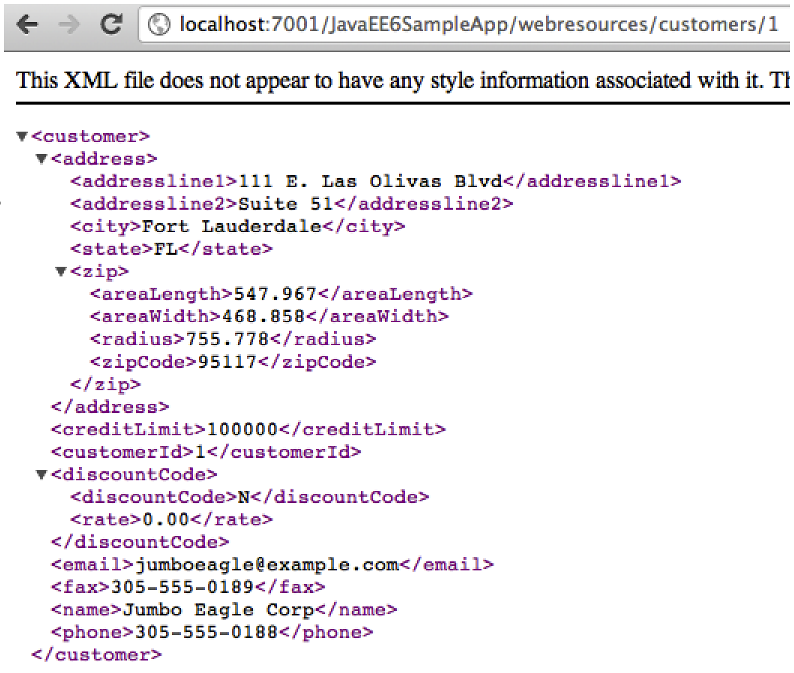
Make sure to deploy the project again by right-clicking on the project and selecting "Deploy".

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

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

The <CONTEXT-ROOT> is the context root of the web application. The <SUB-RESOURCE-URI> may be optional in some cases. The <VARIABLE-PART> is the part that is bound to the parameters in Java method.

In our case, the URI will look like:



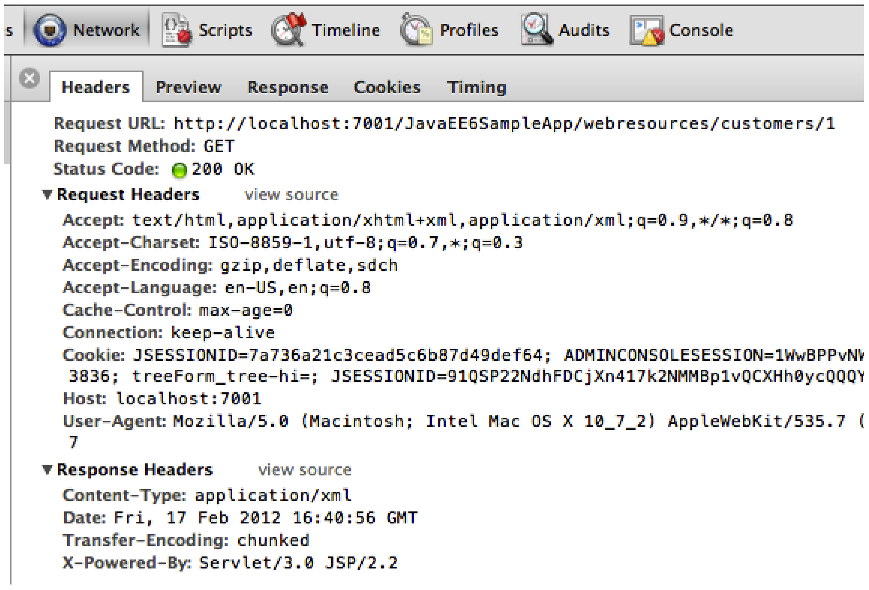
localhost:7001/JavaEE6SampleApp/webresources/customers/1

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

```
▼<customer>
  ▼<address>
     <addressline1>111 E. Las Olivas Blvd</addressline1>
     <addressline2>Suite 51</addressline2>
     <city>Fort Lauderdale</city>
     <state>FL</state>
    ▼<zip>
       <areaLength>547.967</areaLength>
       <areaWidth>468.858</areaWidth>
       <radius>755.778</radius>
       <zipCode>95117</zipCode>
     </zip>
   </address>
   <creditLimit>100000</creditLimit>
   <customerId>1</customerId>
  ▼<discountCode>
     <discountCode>N</discountCode>
     <rate>0.00</rate>
   </discountCode>
   <email>jumboeagle@example.com</email>
   <fax>305-555-0189</fax>
   <name>Jumbo Eagle Corp</name>
   <phone>305-555-0188</phone>
 </customer>
```

http://localhost:7001/JavaEE6SampleApp/restful/customers/{id} where `{id}` is the customer id shown in the JSF page earlier. So accessing "http://localhost:7001/JavaEE6SampleApp/restful/customers/1" in a browser displays the output as shown.

Accessing this URI 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 ("Tools", "Developer Tools" in Chrome).



```
Request URL: http://localhost:7001/JavaEE6SampleApp/webresources/customers/1
Request Method: GET
Status Code: ● 200 OK
▼Request Headers      view source
  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
  Cookie: JSESSIONID=7a736a21c3cead5c6b87d49def64; ADMINCONSOLESESSION=1WwBPPvN
   3836; treeForm_tree-hi=; JSESSIONID=91QSP22NdhFDCjXn417k2NMMBp1vQCXHh0ycQQQ
  Host: localhost:7001
  User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_2) AppleWebKit/535.7 (
   7
▼Response Headers      view source
  Content-Type: application/xml
  Date: Fri, 17 Feb 2012 16:40:56 GMT
  Transfer-Encoding: chunked
  X-Powered-By: Servlet/3.0 JSP/2.2
```

10.4 Each resource can be represented in multiple formats. Change the `@Produces` annotation in `CustomerSessionBean.java` from:

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

to

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

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

curl -H"Accept: application/json" http://localhost:7001/JavaEE6SampleApp/webresources/customers/1 -v

```
* About to connect() to localhost port 7001 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 7001 (#0)
> GET /JavaEE6SampleApp/webresources/customers/1 HTTP/1.1
> User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/7.21.4 OpenSSL/0.9.8r zlib/1.2.5
> Host: localhost:7001
> Accept: application/json
>
< HTTP/1.1 200 OK
< Date: Fri, 17 Feb 2012 18:03:01 GMT
< Transfer-Encoding: chunked
< Content-Type: application/json
< X-Powered-By: Servlet/3.0 JSP/2.2
<
* Connection #0 to host localhost left intact
* Closing connection #0
{"address":{"addressline1":"111 E. Las Olivas Blvd","addressline2":"Suite 51","city":"Fort
Lauderdale","state":"FL","zip":
{"areaLength":"547.967","areaWidth":"468.858","radius":"755.778","zipCode":"95117"}},"creditLimit":"100000","c
ustomerId":"1","discountCode":
{"discountCode":"N","rate":"0.00"},"email":"jumboeagle@example.com","fax":"305-555-0189","name":"Jumbo
Eagle Corp","phone":"305-555-0188"}
```

Notice the following points:

- The command is shown in the bold letters.
- Connection handshake is pre-fixed "*".
- The HTTP request headers are pre-fixed with ">" and response headers with "<".
- The response in JSON format is at the end of the message.

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

11.0 Inject Beans using CDI Qualifiers

The Contexts and Dependency Injection is a new specification in the Java EE 6 platform. The CDI specification provides a type-safe approach to dependency injection. The specification promotes "strong typing" and "loose coupling". A bean only specifies the type and semantics of other beans it depends upon, only using the Java type system with no String-based identifiers. The bean requesting injection may not be aware of the lifecycle, concrete implementation, threading model, or other clients requesting injection. The CDI runtime finds the right bean in the right scope and context and then injects it in the requestor. This loose coupling makes code easier to maintain.

The CDI specification defines "Qualifiers" as a means to uniquely identify one of the multiple implementations of a bean to be injected. The specification defines certain pre-defined qualifiers (`@Default`, `@Any`, `@Named`, `@New`) and allows new qualifiers to be defined easily.

This section shows how one or many implementations of a bean can be injected using CDI Qualifiers.

11.1 Right-click on the project, select “New”, “Other...”, “Java”, “Java Interface...”, give the name as “Greeter” and choose the package name as “org.samples”. Click on “Finish”.

11.2 Add the following method to this interface:

```
public String greet(Customer customer);
```

This interface will be used to greet the customer.

Resolve the imports.

11.3 Add an implementation of the customer by adding a new class with the name “NormalGreeter”. Change the generated class to look like:

```
public class NormalGreeter implements Greeter {

    @Override
    public String greet(Customer customer) {
        return "Hello " + customer.getName() + "!";
    }
}
```

This class implements the `Greeter` interface and provide a concrete implementation of the method.

11.4 In `TestServlet.java`, inject a Greeter as:

```
@Inject Greeter greeter;
```

Note, this is injecting the interface, not the implementation.

Resolve the imports.

11.5 Change the code in the `processRequest` method which was printing the list of customers from:

```
            for (Customer c : result) {
                out.println(c + "<br>");
            }
```

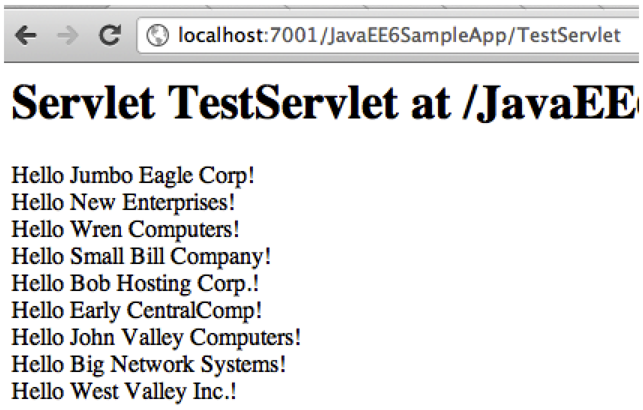
to

```
            for (Customer c : result) {
                out.println(greeter.greet(c) + "<br>");
            }
```

This code now greets each customer using `Greeter` interface instead of just printing their name. Behind the scene, the CDI runtime looks for an implementation of this interface, finds `NormalGreeter`, and injects it.

The default scope of the bean is `@Dependent` and is injected directly. For non-default scopes, a proxy to the bean is injected. The actual bean instance is injected at the runtime after the scope and context is determined correctly.

11.6 Access the page "http://localhost:7001/JavaEE6SampleApp/TestServlet" in a browser to see the output as shown.



Servlet TestServlet at /JavaEE

```
Hello Jumbo Eagle Corp!
Hello New Enterprises!
Hello Wren Computers!
Hello Small Bill Company!
Hello Bob Hosting Corp.!
Hello Early CentralComp!
Hello John Valley Computers!
Hello Big Network Systems!
Hello West Valley Inc.!
```

Lets add an alternative implementation of Greeter that greets customers based upon their credit limit.

11.7 Right-click on the project, select "New", "Java Class...", give the name as "PromotionalGreeter". Change the implementation to:

```
public class PromotionalGreeter extends NormalGreeter {

    @Override
    public String greet(Customer customer) {
        String greeting = super.greet(customer);

        if (customer.getCreditLimit() >= 100000)
            return "You are super, thank you very much! " + greeting;

        if (customer.getCreditLimit() >= 50000)
            return "Thank you very much! " + greeting;

        if (customer.getCreditLimit() >= 25000)
            return "Thank you! " + greeting;

        return greeting;
    }
}
```

This method returns a different greeting method based upon the credit limit. Notice, this class is extending `NormalGreeter` class and so now the `Greeter` interface has two implementations in the application.

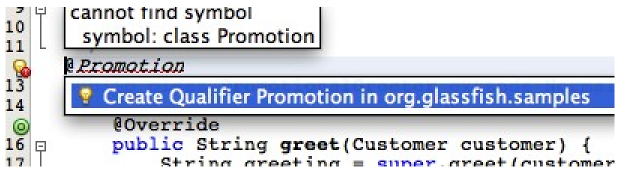
Resolve the imports. As soon as you save this file, the NetBeans IDE tries to deploy the project but fails with the following error:

```
Caused By: org.jboss.weld.exceptions.DeploymentException: WELD-001409 Ambiguous
dependencies for type [Greeter] with qualifiers [@Default] at injection point
[[field] @Inject org.samples.TestServlet.greeter]. Possible dependencies [[Managed
Bean [class org.samples.NormalGreeter] with qualifiers [@Any @Default], Managed
Bean [class org.samples.PromotionalGreeter] with qualifiers [@Any @Default]]]
```

The error message clearly explains that the `Greeter` interface has two implementations, both with the default set of qualifiers. The CDI runtime finds both the implementations equally capable for injection and gives an error message explaining the “ambiguous dependencies”.

Lets resolve this by adding a qualifier on one of the implementations.

11.8 Add `@Promotion` as a class-level annotation on `PromotionalGreeter.java`. Click on the yellow bulb and take the suggestion to create a CDI Qualifier.



```
cannot find symbol
  symbol: class Promotion
@Promotion
    Create Qualifier Promotion in org.glassfish.samples
    @Override
    public String greet(Customer customer) {
        String greeting = super.greet(customer
```

This generates the boilerplate code required for the `@Promotion` qualifier as:

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Promotion {
}
```

As you can see, the CDI qualifier is a Java annotation, that itself is annotated with the `@javax.inject.Qualifier` meta-annotation.

The project gets successfully deployed after this file is saved. This happens correctly now because one of the implementations of the `Greeter` interface (`PromotionalGreeter`) is more qualified than the other (`NormalGreeter`) and so the default implementation (`NormalGreeter`) can be injected without any ambiguities.

Refreshing the page “http://localhost:7001/JavaEE6SampleApp/TestServlet” shows the same the output as earlier.

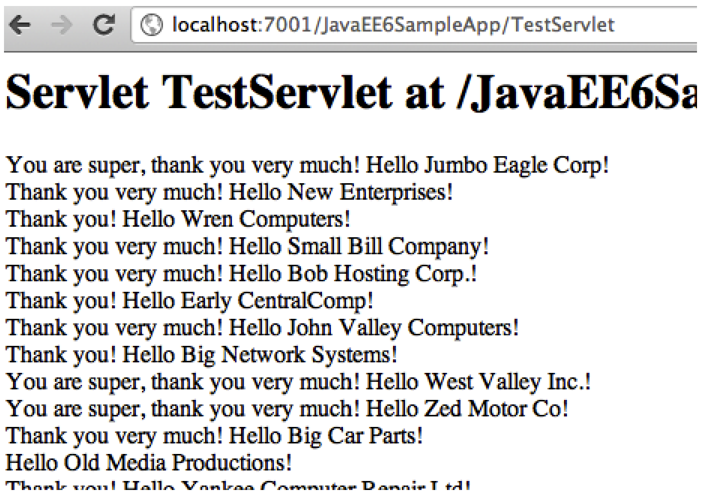
11.9 Change the injection of `Greeter` to:

```
@Inject @Promotion Greeter greeter;
```

Save the file, refreshing the page "http://localhost:8080/JavaEE6SampleApp/TestServlet" displays the output as shown.

The updated greeting message shows that the `PromotionalGreeter` is injected instead of the default one (`NormalGreeter`).

The CDI qualifiers ensures that there is no direct dependency on any (possibly many) implementations of the interface.



```
localhost:7001/JavaEE6SampleApp/TestServlet

Servlet TestServlet at /JavaEE6Sa

You are super, thank you very much! Hello Jumbo Eagle Corp!
Thank you very much! Hello New Enterprises!
Thank you! Hello Wren Computers!
Thank you very much! Hello Small Bill Company!
Thank you very much! Hello Bob Hosting Corp.!
Thank you! Hello Early CentralComp!
Thank you very much! Hello John Valley Computers!
Thank you! Hello Big Network Systems!
You are super, thank you very much! Hello West Valley Inc.!
You are super, thank you very much! Hello Zed Motor Co!
Thank you very much! Hello Big Car Parts!
Hello Old Media Productions!
```

12.0 Cross-cutting Concerns using CDI Interceptors

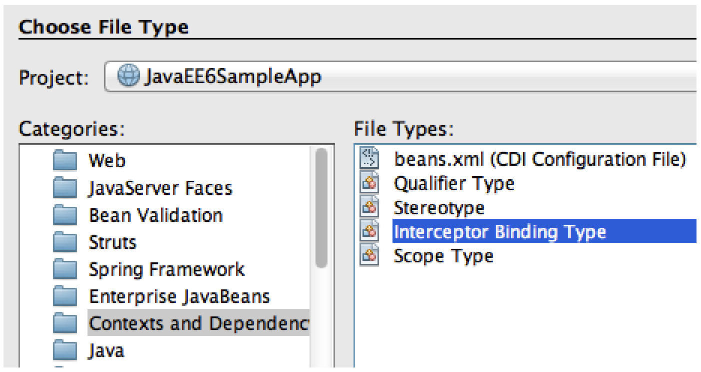
The Interceptors do what they say – they intercept on invocation and lifecycle events on an associated target class. They are used to implement cross-cutting concerns like logging and auditing. The Interceptors specification is extracted from the EJB specification into a new specification so that they can more generically applied to a broader set of specifications.

The CDI specification enhances the basic functionality by providing an annotation-based approach to binding interceptors to beans. The target class may specify multiple interceptors thus forming a chain of interceptors.

This section will explain how to add an interceptor that intercepts all the business method invocations and logs them.

12.1 Right-click on the project, select "New", "Other...", "Contexts and Dependency Injection", "Interceptor Binding Type" as shown.

Click on "Next>".



Choose File Type

Project: JavaEE6SampleApp

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

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

12.2 Give the class name as "Logging" and choose the package name as "org.samples", and click on "Finish".

This defines the *interceptor binding type* which is specified on the target class. This binding can have multiple implementations which can be enabled or disabled at deployment using `beans.xml`.

The generated binding consists of the following code:

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface Logging {
}
```

12.3 Implement (or bind) the interceptor by creating a POJO class and naming it "LoggingInterceptor". Change the class definition to:

```
@Interceptor
@Logging
public class LoggingInterceptor {

    @AroundInvoke
    public Object intercept(InvocationContext context) throws Exception {
        System.out.println("BEFORE: " + context.getMethod());
        Object result = context.proceed();
        System.out.println("AFTER: " + context.getMethod());

        return result;
    }
}
```

Resolve the imports.

Notice the following points in the code:

- This class has the `@Logging` annotation which is the binding type generated earlier.
- It has the `@Interceptor` annotation marking this class to be an interceptor implementation.
- The `@AroundInvoke` annotation defines the method that will intercept the business method invocation on the target class and require the signature as shown in the code. An interceptor implementation cannot have more than one `@AroundInvoke` method.
- The `InvocationContext` parameter provides information about the intercepted invocation and provide operations to control the behavior of the chain invocation.
- This method prints a log message, with the name of the business method being invoked, before and after the business method execution.

12.4 The interceptors may be specified on the target class using the `@Interceptors` annotation which suffers from stronger coupling to the target class and not able to change the ordering globally. The recommended way to enable interceptors is by specifying them in `beans.xml`.

Add the following fragment to `beans.xml` to enable the interceptor:

```
    <interceptors>
        <class>org.samples.LoggingInterceptor</class>
    </interceptors>
```

12.5 The interceptors may be specified at a class- or a method-level. Lets intercept all invocations of the `getCustomers` method in `CustomerSessionBean.java`. This can be done by adding the following annotation right above the method:

```
@Logging
```

This is the same binding type that we created earlier.

12.6 Access the URL "http://localhost:7001/JavaEE6SampleApp/TestServlet" in a browser which invokes `CustomerSessionBean.getCustomers` method and displays the familiar output as seen earlier.

The interesting part is shown in the WebLogic log as shown:

```
BEFORE: public java.util.List org.samples.CustomerSessionBean.getCustomers()
<Feb 17, 2012 7:32:34 PM CET> <Notice> <EclipseLink> <BEA-2005000> <2012-02-17
19:32:34.695--ServerSession(274985990)--EclipseLink, version: Eclipse Persistence
Services - 2.3.2.v20111125-r10461>
<Feb 17, 2012 7:32:34 PM CET> <Notice> <EclipseLink> <BEA-2005000> <2012-02-17
19:32:34.695--ServerSession(274985990)--Server: 12.1.1.0>
<Feb 17, 2012 7:32:34 PM CET> <Notice> <EclipseLink> <BEA-2005000> <2012-02-17
19:32:34.704--ServerSession(274985990)--
JavaEE6SampleAppfile:/Users/arungup/samples/weblogic/JavaEE6SampleApp/build/web/WE
B-INF/classes/_JavaEE6SampleAppPU login successful>
AFTER: public java.util.List org.samples.CustomerSessionBean.getCustomers()
```

It shows "BEFORE" and "AFTER" string along with the business method name and return type. These statements are printed from the interceptor implementation. The other statements show the log statements from the persistence layer.

13.0 Conclusion

This hands-on lab created a typical 3-tier Java EE 6 application using Java Persistence API (JPA), Servlet, Enterprise JavaBeans (EJB), JavaServer Faces (JSF), Java API for RESTful Web Services (JAX-RS), Contexts and Dependency Injection (CDI), and Bean Validation. It also explained how typical design patterns required in a web application can be easily implemented using these technologies.

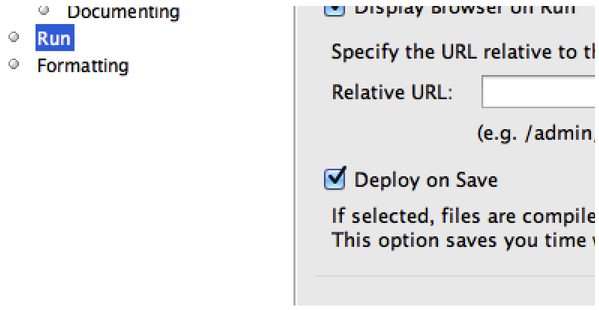
Hopefully this has raised your interest enough in trying out Java EE 6 applications using WebLogic and NetBeans.

Send us feedback at TBD.

14.0 Troubleshooting

14.1 The project is getting deployed to WebLogic 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.



Specify the URL relative to t

Relative URL:

(e.g. /admin

☑ Deploy on Save

If selected, files are compile
This option saves you time

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

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

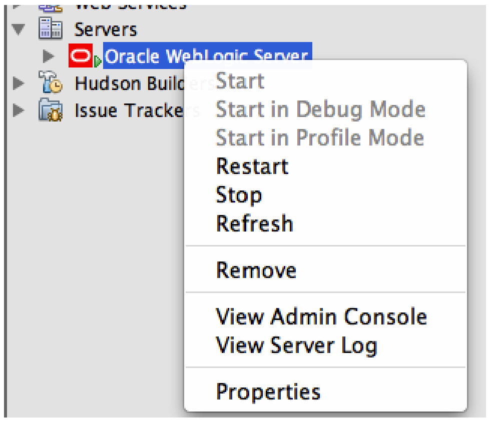
```
<properties/>
```

with

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

14.3 How can I start/stop/restart WebLogic 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".



15.0 Completed Solutions

The completed solution is available as a NetBeans project at:

TBD

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