Java EE 6 Hands-on Lab
Feb 2012

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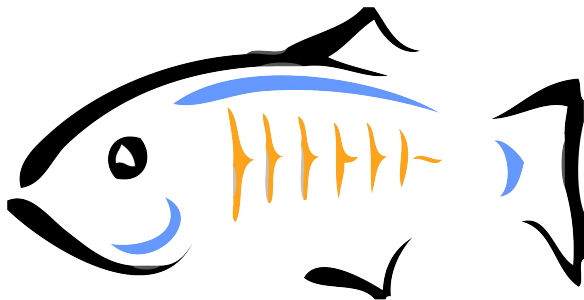
Table of Contents

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 GlassFish 3.1.1.

The latest copy of this document is always available at https://blogs.oracle.com/arungupta/resource/javaee6-hol-feb2012.pdf.

1.1 Software Downloads

The following software need to be downloaded and installed:

- JDK 6 or 7 from http://www.oracle.com/technetwork/java/javase/downloads/index.html.
- NetBeans 7.0.1+ "All" or "JavaEE" version from http://netbeans.org/downloads/index.html. This version includes a pre-configured GlassFish 3.1.1.

If you want to use Oracle WebLogic Server 12c for development and deployment of this application then please follow the instructions at TBD.

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. 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 both NetBeans "All" and "Java EE" downloaded bundled 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>".
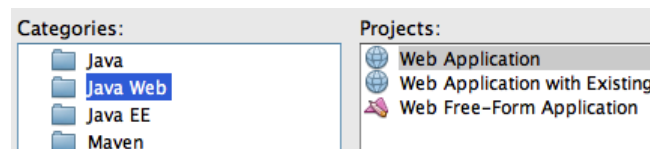
3.3 Specify the project name as "JavaEE6SampleApp" and click on "Next>".

3.4 Choose 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".

Hello World!

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.
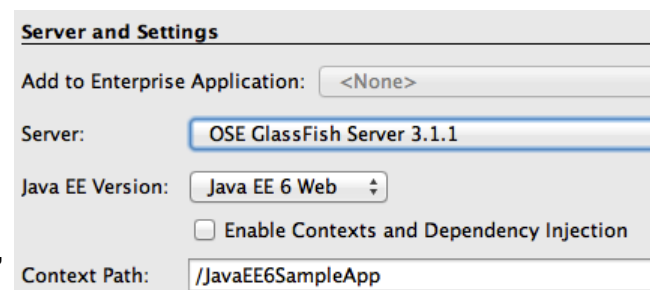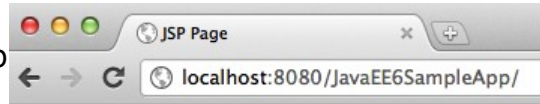
A display of this page ensures that the project is successfully created and GlassFish has started successfully as well. The GlassFish server comes bundled with the JavaDB and is pre-configured in NetBeans to start the database server as well. This can be changed by expanding the "Servers" tree in the "Services" tab, right-clicking on the chosen server, selecting the properties, and selecting/unselecting the checkbox that says "Start Registered Derby Server".
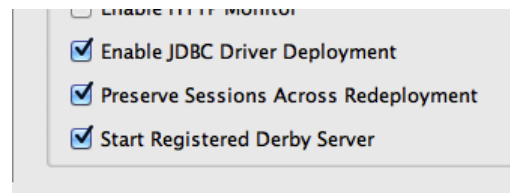
4.0 Generate JPA Entities from the Database Table

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", "Other...",
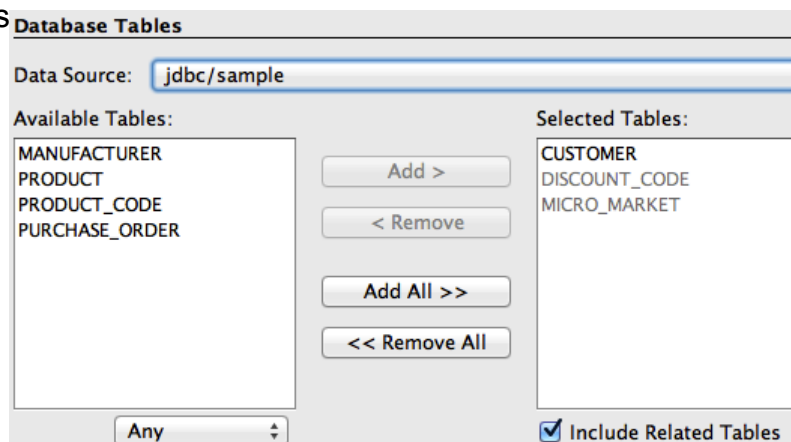
"Persistence", "Entity Classes from Database...". Choose "jdbc/sample" as the Data Source from the drop down list box as shown. This will show all the tables from this data source.

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.

Click on "Next>".

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

The database table name 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.glassfish.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:

Notice the following points:
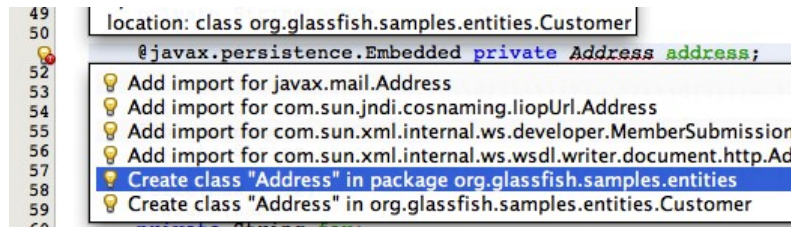
- The two blocks of code above are not adjacent.

- 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 from a Servlet

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

This section will create a Servlet and invoke the @NamedQuery to query the database.

5.1 Right-click on the project, select "New", "Servlet...". Enter the class name as "TestServlet", package as org.glassfish.samples, 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 {
```

5.2 Inject EntityManagerFactory in Servlet by adding the following code right after the class declaration:

```
@PersistenceUnit EntityManagerFactory emf;
```

Resolve the imports by clicking on the yellow bulb.

5.3 Add the code to use the injected `EntityManagerFactory` to query the database using a pre-generated `@NamedQuery` to retrieve all the customers.

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 = (List<Customer>)emf.
            createEntityManager().
            createNamedQuery("Customer.findAll").
            getResultList();
    for (Customer c : result)
        out.println(c + "<br>");
    out.println("</body>");
    out.println("</html>");
} finally {
```

Since the Servlets are re-entrant and `EntityManager` is not thread safe, it needs to be obtained from an `EntityManagerFactory` during each method invocation. This is resolved when the database specific code is moved to an EJB later.

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.
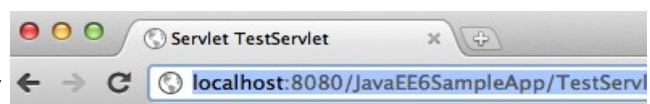
🔍 Notice that no transactions are started or committed before/after the database operation as this is a read-only operation and does not have the demands of a real-life application. An explicit transaction has to be created and committed (or rollbacked) if either create, update, or delete operations are performed. In addition, JPA2 also provides support for optimistic and pessimistic locking by means of specified locking modes.

Servlet TestServlet at /JavaE

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]

5.4 Right-click in the editor pane, select "Run File", choose the defaults, and click on OK. This displays "http://localhost:8080/JavaEE6SampleApp/TestServlet" page in the browser and looks like as shown.

6.0 Active Redeploy to Preserve Sessions

GlassFish provides support for preserving session state across redeployments. This feature is called as *Active Redeploy* and works for HTTP sessions, stateful EJBs, and persistently created EJB timers.

This section will show how Active Redeploy for HTTP sessions, together with NetBeans auto-deploy, boosts productivity for Java EE 6 application developers.

6.1 In `TestServlet.java`, add the following code before the database access code:

```
int count;
if (request.getSession().getAttribute("count") == null) {
    count = 0;
} else {
    count = (Integer) request.getSession().getAttribute("count");
}
request.getSession().setAttribute("count", ++count);
out.println("<h2>Timestamp: " + new Date() + "<br>");
out.println("Times: " + request.getSession().getAttribute("count") +
        "</h2>");
```

Fix the imports by taking the defaults.

This code stores a session attribute identifying the number of times a page has been accessed.

Refresh the page "http://localhost:8080/JavaEE6SampleApp/TestServlet" in the browser to see the output as shown.

← → C ⊙ localhost:8080/JavaEE6SampleApp/TestServlet

Servlet TestServlet at /JavaEE6SampleApp

Timestamp: Sun Nov 06 22:10:31 CET 2011
Access count: 1

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

The "Timestamp" and "Access count" is displayed on the page.

The output from the Servlet may look slightly different based upon where you copy/pasted the code.

6.2 Edit `Customer.java`, change the implementation of the `toString` method to:

```
return name + " (" + customerId + ")";
```

Notice, "[" are replaced with " (". Save the file.

The NetBeans IDE auto-deploys a Java EE 6 project with every file save. This can be configured in "Project Properties", "Run", and by selecting/unselecting the checkbox "Deploy on Save" as shown.
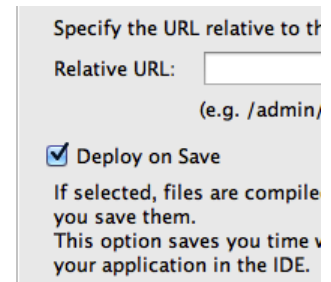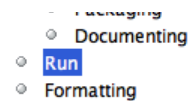
The updated output looks like as shown.

Notice, that even though the output from the `toString` is correctly updated, indicating the app was redeployed successfully, the session access count was incremented by one. This shows that the session attributes were preserved across the redeployment.

This boosts developer's productivity because HTTP session, which is where typically application's login information might be stored, is not invalidated if your application is redeployed. This is an optional feature and can be disabled by viewing the properties of the GlassFish server in the "Services" tab of NetBeans and unselecting "Preserve Sessions Across Redeployment".
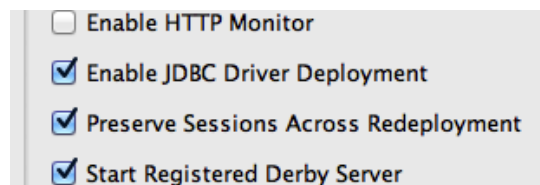
7.0 Query the Database using EJB and Refactor the Servlet

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.

7.1 Create a new stateless EJB. Right-click on "org.glassfish.samples" package, select "New", "Session Bean...", specify the EJB Name as "CustomerSessionBean" as shown. Take all the defaults, 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.

7.2 EJBs are not re-entrant and so we can inject an instance of `EntityManager`, as opposed to `EntityManagerFactory` in the Servlet, as shown:

```
@PersistenceContext
EntityManager em;
```

Resolve the imports by right-clicking on the editor pane and selecting "Fix Imports".

7.3 Add the following method in the EJB:

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

The implementation here is similar to the code used in the Servlet earlier.

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.

7.4 The database access code in the Servlet can now be removed and all of that functionality can be delegated to this EJB. This can be done by replacing the following code from the Servlet:

```
@PersistenceUnit EntityManagerFactory emf;
```

with

```
@EJB CustomerSessionBean bean;
```

And replacing the following code:

```
        List<Customer> result = (List<Customer>)emf.
            createEntityManager().
            createNamedQuery("Customer.findAll").
            getResultList();
```

with

```
List<Customer> result = bean.getCustomers();
```

Notice how an EJB is easily injected using the `@EJB` annotation and is then used to invoke the business method.

Refreshing "http://localhost:8080/JavaEE6SampleApp/TestServlet" displays the output as shown.

← → C ☉ localhost:8080/JavaEE6SampleApp/TestServlet

Servlet TestServlet at /JavaEE6

Timestamp: Sun Nov 06 23:01:13 CET 201
Access count: 3

This is almost the same output as shown earlier, and that's the whole point! EJBs are better suited for database access as all methods are implicitly transactional. This will be more apparent when we add code to insert values in the database table.

Jumbo Eagle Corp (1)
New Enterprises (2)
Wren Computers (25)
Small Bill Company (3)
Bob Hosting Corp. (36)

Using an EJB, instead of Servlet, will not require to start and commit a transaction explicitly.

The only difference from the previous output is the updated timestamp and the access count has increased by 1.

8.0 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.

8.1 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.glassfish.samples.Address` instead of the default value.

8.2 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.glassfish.samples.Address`.

8.3 Access "http://localhost:8080/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.

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.

8.4 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.

8.5 Access the URL "http://localhost:8080/JavaEE6SampleApp/TestServlet?add=5" in a browser and this tries to add a new customer to the database and shows the output as shown.

The output shows the "Access count" incremented by 1 but the list of customers is not shown. Instead the GlassFish output log shows the following message:

```
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.

8.6 Edit `TestServlet.java`, change the phone number from "1234" to "1234567", and save the file. And access the URL "http://localhost:8080/JavaEE6SampleApp/TestServlet?add=5" again to see the output as shown.

The customer is now successfully added.

9.0 Type-safe Critieria Query using JPA2 Metamodel

The JPA2 specification defines Metamodel classes to capture the meta-model of the persistent state and relationships of the managed classes of a persistence unit. The canonical metamodel classes can be generated statically using an annotation processor following the rules defined by the specification.

This abstract persistence schema is used to author the type-safe queries using Critieria API, rather than use of the string-based approach of the Java Persistence Query Language (JPQL).

This section creates type-safe query using JPA2 Criteria API.

9.1 Right-click on the project and select "Clean and Build" to generate metamodel classes for the JPA entities. The project explorer looks like as shown.

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" as shown.

Here is how the generated `Address_.java` class looks like:

```java
package org.glassfish.samples.entities;

import javax.annotation.Generated;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;
import org.glassfish.samples.entities.MicroMarket;

@Generated(value="EclipseLink-2.2.0.v20110202-r8913", date="2011-11-07T16:13:45")
@StaticMetamodel(Address.class)
public class Address_ {

    public static volatile SingularAttribute<Address, String> addressline2;
    public static volatile SingularAttribute<Address, MicroMarket> zip;
    public static volatile SingularAttribute<Address, String> addressline1;
    public static volatile SingularAttribute<Address, String> state;
    public static volatile SingularAttribute<Address, String> city;

}
```

This class provides the type and cardinality of each field in the `Address` entity.

9.2 Add the following code to `CustomerSessionBean.java`:

```java
public List<Customer> getCustomers2() {
    CriteriaBuilder builder = em.getCriteriaBuilder();
    CriteriaQuery criteria = builder.createQuery(Customer.class);

    // FROM clause
    Root root = criteria.from(Customer.class);

    // SELECT clause
    criteria.select(root);

    // No WHERE clause

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

    // PRINT result
    return query.getResultList();
}
```

This method returns the list of all customers from the database, just like `getCustomers` method earlier, but uses type-safe Criteria API to query the database.

The generated metamodel will be used later.

🔍 Note that all the information is specified without any string-based identifiers.

Fix the imports by taking all the defaults.

9.3 In "TestServlet.java", replace the code:

```
List<Customer> result = bean.getCustomers();
```

with

```
List<Customer> result = bean.getCustomers2();
```

Notice the new EJB function is now invoked.

Fix the imports.

9.4 Refresh the page "http://localhost:8080/JavaEE6SampleApp/TestServlet" in browser to see an output like:

This confirms that there is no change in the output, as expected.

10.0 Caching using Singleton EJB and @Schedule to Pre-fetch Data

The EJB 3.1 specification introduces `@Singleton` annotation that creates a single instance of an EJB per-application and per-JVM. By default, all methods are thread-safe and transactional. The specifications also simplifies how to schedule callback methods using calendar-based expressions. These methods can be invoked at a specified time, after a specified elapsed duration, or at recurring intervals.

This section adds a singleton EJB that can be used to pre-fetch data from the database at recurring intervals.

10.1 Right-click on "org.glassfish.samples" package, select "New", "Session Bean...", type the name as "CacheSingletonBean", and type as "Singleton" as shown.

And click on OK.

10.2 Add the following class-level annotation:

```
@javax.ejb.Startup
```

This will ensure that this singleton bean is loaded eagerly during the application startup.

10.3 Add the following code to the generated class:

```
@EJB CustomerSessionBean bean;
List<Customer> customers;

public List<Customer> getCustomers() {
    return customers;
}

@Schedule(hour="*", minute="*", second="*/30")
public void preFetchCustomers() {
    System.out.println(new Date() + " Fetching customers");
    customers = bean.getCustomers();
}
```

Fix the imports.

This code is using previously created `CustomerSessionBean` to get the list of customers. This method has a `@Schedule` annotation that takes cron-like syntax to specify the recurring interval at which the method needs to be invoked.

The GlassFish server log will show messages like:

```
INFO: Tue Nov 08 00:06:00 CET 2011 Fetching customers
INFO: Tue Nov 08 00:06:30 CET 2011 Fetching customers
INFO: Tue Nov 08 00:07:00 CET 2011 Fetching customers
INFO: Tue Nov 08 00:07:30 CET 2011 Fetching customers
INFO: Tue Nov 08 00:08:00 CET 2011 Fetching customers
```

The log messages show that customer cache is updated every 30 seconds, as defined by `@Schedule`.

The application can now use this singleton bean, instead of `CustomerSessionBean`, to get the list of customers such as:

```
@EJB CacheSingletonBean bean;

. . .

bean.preFetchCustomers();
```

Note, `bean.getCustomers()` cannot be invoked as it may return `null` if the timer event has not occurred yet at least once.

You may consider commenting `@Schedule` annotation to avoid flooding the server log with an output from the `preFetchCustomers` method.

11.0 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.

11.1 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.
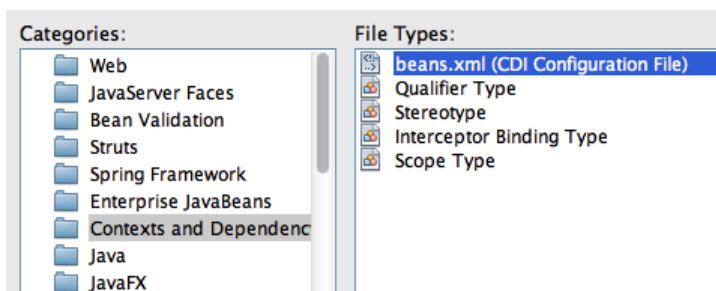
Click on "Next>", take all the defaults as shown.

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.

11.2 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.

11.3 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 GlassFish, if any JSF-specific annotations are used in the application then the framework is automat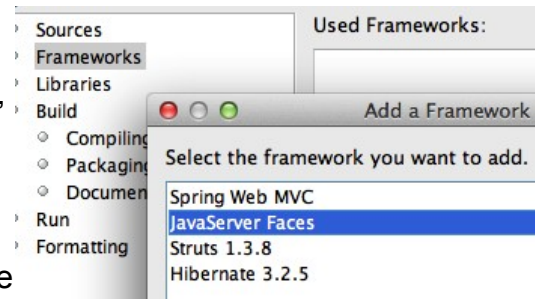ically registered by the underlying Web container. But since our application is not using any such annotation so we have to explicitly enable it.

Adding the framework in this case will generate a `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:8080/JavaEE6SampleApp/faces/index.xhtml" with the output as shown.

11.4 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".

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.

11.5 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 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.

11.6 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.

11.7 Refresh "http://localhost:8080/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.

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

11.9 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.

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.

11.10 Refreshing the page "http://localhost:8080/JavaEE6SampleApp/faces/index.xhtml" displays the result as shown.

🔍 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.

12.0 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.

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

Click on the yellow bulb and select the hint.
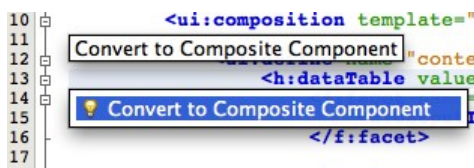
12.2 Take the defaults as shown.

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 standar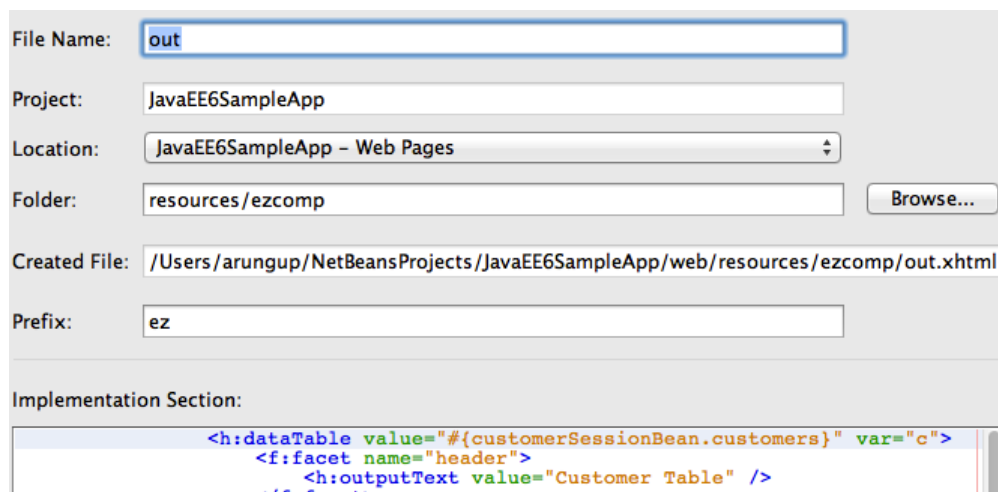d `resources` directory defined by the JSF2 specification. The tag name for the new composite component is the same as the defining page file name. The updated directory structure looks like as shown.

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:8080/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.

13.0 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.

13.1 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".

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

13.3 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.

13.4 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.

13.5 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();
}
```

This method is similar to `getCustomers2` added earlier. The two differences are:

- The method takes `AjaxBehaviorEvent` as the parameter as this is requirement for the `listener` attribute of `<f:ajax>`.

- The Criteria query specifies a WHERE clause using the JPA2 metamodel that was generated earlier. 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. Make sure `javax.persistence.criteria.Predicate` class is imported, this is not the default.

13.6 Inject the customer name as:

```
@Inject CustomerName name;
```

And resolve the import again by taking defaults.

13.7 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;
}
```

13.8 Save all the files and open "http://localhost:8080/JavaEE6SampleApp/faces/list.xhtml" in the browser. The default output looks very familiar as shown.

This is because no criteria is specified in the text and so the complete list 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.

Notice, all of this is only refreshing the `<h:dataTable>` and there by showing "partial page refresh" or "partial page rendering".
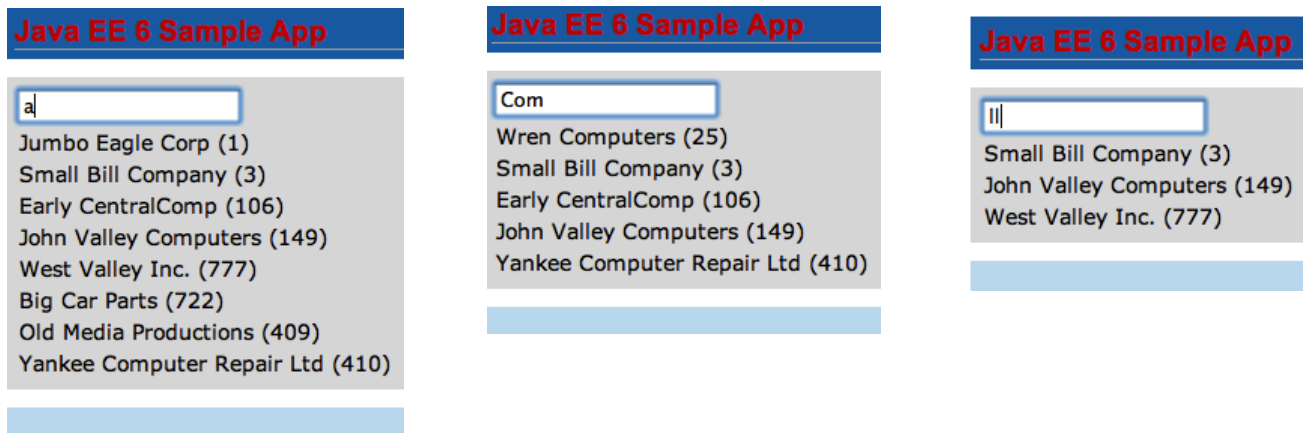
14.0 Publish EJB as a RESTful Resource using JAX-RS

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.

14.1 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`.

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 "/restful" and 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 "restful" as can be seen in the generated class.

14.2 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.

14.3 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:
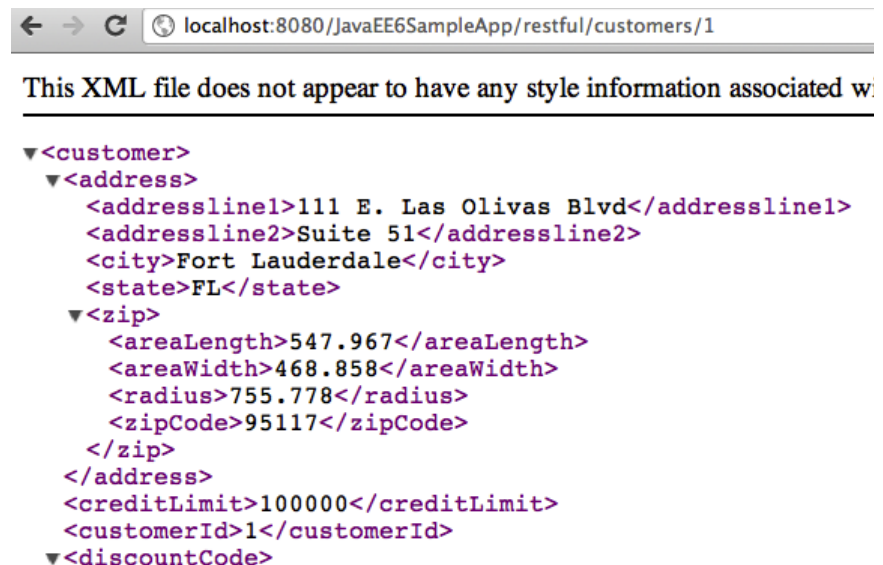
http://localhost:8080/JavaEE6SampleApp/restful/customers/{id} where {id} is the customer id shown in the JSF page earlier. So accessing "http://localhost:8080/JavaEE6SampleApp/restful/customers/1" in a browser displays the output as shown.

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).

```
rces   Network   Scripts   Timeline   Profiles   Audits   Console
       Headers   Preview   Response   Cookies   Timing
       Request URL: http://localhost:8080/JavaEE6SampleApp/restful/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
       Host: localhost:8080
       User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_2) AppleWebKit/53
     ▼ Response Headers     view source
       Content-Length: 601
       Content-Type: application/xml
       Date: Mon, 14 Nov 2011 14:32:22 GMT
       Server: GlassFish Server Open Source Edition 3.1.1
       X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3
```

14.4 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:8080/JavaEE6SampleApp/restful/customers/1 -v
* About to connect() to localhost port 8080 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 8080 (#0)
> GET /JavaEE6SampleApp/restful/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:8080
> Accept: application/json
>
< HTTP/1.1 200 OK
< X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.1 Java/Apple Inc./1.6)
< Server: GlassFish Server Open Source Edition 3.1.1
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Tue, 08 Nov 2011 14:45:52 GMT
<
* 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":"78","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/.

15.0 Jersey Client API to Access the RESTful Resource

Jersey is the Reference Implementation of JAX-RS. The Jersey Client API is an easy-to-use and high-level Java API that can help you write clients for any HTTP-based RESTful Web service.

◎◄ This section will create a client that will access the RESTful resource published at the `/customers/{id}` path.

15.1 Right-click on the project, select "New", "Java Class...", give the class name as "RESTfulClient", take all other defaults, and click on "Finish".

15.2 Add the following code to the generated class:

```
    public static void main(String[] args) {
        String BASE_URL =
"http://localhost:8080/JavaEE6SampleApp/restful/customers/";
        Client client = Client.create();
        WebResource resource = client.resource(BASE_URL + "1");
        resource.addFilter(new LoggingFilter());
        resource.get(String.class);

        resource.accept(MediaType.APPLICATION_JSON).get(String.class);
    }
```

🔍 Notice the following points:

- This code creates a new Jersey client runtime instance. This is an expensive object

and so must be saved if no other configuration is required to access the resources.
- A new `WebResource` is created per RESTful resource that is capable of building requests to send to the resource and processing responses returned from the resource. Each `WebResource` follows a builder pattern to specify the required MIME type in the response, `application/json` in this case.
- A `LoggingFilter` is added to the resource to print the request and response messages.
- The code then requests a default and JSON representation of the resource in two separate calls.

Fix the imports by taking all the defaults.

15.3 Right-click on the file and select "Run File" to see the following output:

```
run:
Nov 8, 2011 10:15:43 PM com.sun.jersey.api.client.filter.LoggingFilter log
INFO: 1 * Client out-bound request
1 > GET http://localhost:8080/JavaEE6SampleApp/restful/customers/1

Nov 8, 2011 10:15:43 PM com.sun.jersey.api.client.filter.LoggingFilter log
INFO: 1 * Client in-bound response
1 < 200
1 < Date: Tue, 08 Nov 2011 21:15:43 GMT
1 < Content-Length: 601
1 < Content-Type: application/xml
1 < X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.1
Java/Apple Inc./1.6)
1 < Server: GlassFish Server Open Source Edition 3.1.1
1 <
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
><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><discoun
tCode>78</discountCode><rate>0.00</rate></discountCode><email>jumboeagle@example.c
om</email><fax>305-555-0189</fax><name>Jumbo Eagle Corp</name><phone>305-555-
0188</phone></customer>

Nov 8, 2011 10:15:43 PM com.sun.jersey.api.client.filter.LoggingFilter log
INFO: 2 * Client out-bound request
2 > GET http://localhost:8080/JavaEE6SampleApp/restful/customers/1
2 > Accept: application/json

Nov 8, 2011 10:15:43 PM com.sun.jersey.api.client.filter.LoggingFilter log
INFO: 2 * Client in-bound response
2 < 200
2 < Transfer-Encoding: chunked
2 < Date: Tue, 08 Nov 2011 21:15:43 GMT
2 < Content-Type: application/json
2 < X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.1
Java/Apple Inc./1.6)
2 < Server: GlassFish Server Open Source Edition 3.1.1
2 <
```

```
{"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","customerId":"1","discountCode":
{"discountCode":"78","rate":"0.00"},"email":"jumboeagle@example.com","fax":"305-
555-0189","name":"Jumbo Eagle Corp","phone":"305-555-0188"}
```

```
BUILD SUCCESSFUL (total time: 1 second)
```

The output shows request and response messages to/from the resource. The first pair of messages is pre-fixed with "1" and shows the default XML response coming back. The second pair of messages is pre-fixed with "2" and shows the "Accept" header going from client to endpoint for content-negotiation with server. The server understands the expected format and returns a JSON response.

16.0 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.

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

16.2 Add the following method to this interface:

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

This interface will be used to greet the customer.

Resolve the imports.

16.3 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.

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

```
@Inject Greeter greeter;
```

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

Resolve the imports.

16.5 Add the following code in the `processRequest` method:

```
String greet = request.getParameter("greet");
if (greet != null) {
        Customer customer = bean.getCustomers().get(Integer.parseInt(greet));
        out.println(greeter.greet(customer) + "<br><br>");
}
```

For convenience, this code was added right before the list of customers is printed.

This code reads a parameter "greet" from the URL, uses its value to extract the customer from the database, and invokes the injected `Greeter` interface on it. 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.

16.6 Access the page "http://localhost:8080/JavaEE6SampleApp/TestServlet?greet=0" in a browser to see the output as shown. Note, the value of the parameter is not the customer

identifier, its the order of customer in the list returned by the database.

The output from the recently added statement is highlighted with a red circle.

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

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

```java
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:

```
Error occurred during deployment: Exception while loading the app : WELD-001409
Ambiguous dependencies for type [Greeter] with qualifiers [@Default] at injection
point [[field] @Inject org.glassfish.samples.TestServlet.greeter]. Possible
dependencies [[Managed Bean [class org.glassfish.samples.PromotionalGreeter] with
qualifiers [@Any @Default], Managed Bean [class
org.glassfish.samples.NormalGreeter] 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.

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

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:8080/JavaEE6SampleApp/TestServlet?greet=1" shows the same the output as earlier.

16.9 Change the injection of `Greeter` to:

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

Save the file, refreshing the page "http://localhost:8080/JavaEE6SampleApp/TestServlet?greet=0" displays the output as shown.

The updated greeting message shows that the `PromotionalGreeter` is injected instead of the default one (`NormalGreeter`). The updated greeting is highlighted with a red circle.

Accessing "http://localhost:8080/JavaEE6SampleApp/TestServlet?greet=6" shows a different greeting message, based upon the credit limit of the customer, as shown.

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

17.0 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.

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

Click on "Next>".
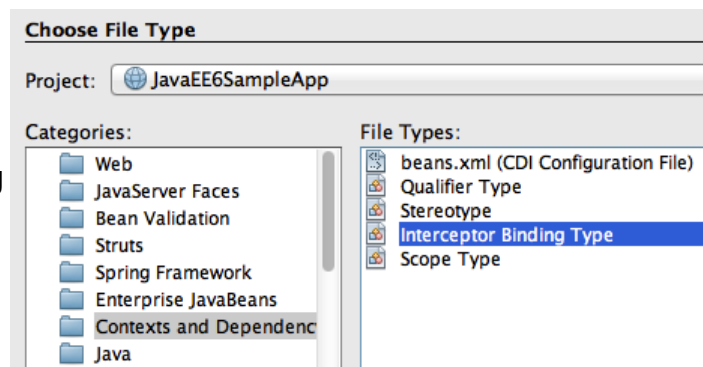
17.2 Give the class name as "Logging" and choose the package name as "org.glassfish.samples", and click on "Finish".

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 {
}
```

17.3 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.

17.4 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.glassfish.samples.LoggingInterceptor</class>
</interceptors>
```
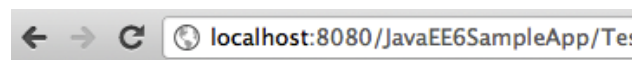
17.5 The interceptors may be specified at a class- or a method-level. Lets intercept all invocations of the `getCustomers2` method in `CustomerSessionBean.java`. This can be done by adding the following annotation right above the method:

```
@Logging
```

This is the same binding type that we created earlier.

17.6 Access the URL

localhost:8080/JavaEE6SampleApp/Te:

Servlet TestServlet at /Jav

Timestamp: Wed Nov 09 21:41:32

Access count: 13

41

Jumbo Eagle Corp (1)
New Enterprises (2)
Wren Computers (25)
Small Bill Company (3)
Bob Hosting Corp (36)

"http://localhost:8080/JavaEE6SampleApp/TestServlet" in a browser which invokes `CustomerSessionBean.getCustomers2` method and displays the output as shown. This is very similar output as we've seen earlier.

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

```
INFO: BEFORE: public java.util.List
org.glassfish.samples.CustomerSessionBean.getCustomers2()
INFO: AFTER: public java.util.List
org.glassfish.samples.CustomerSessionBean.getCustomers2()
```

It shows "BEFORE" and "AFTER" string along with the business method name and return type. These statements are printed from the interceptor implementation.

18.0 Observer Pattern using CDI Events

The CDI events follows the *Observer* pattern. Event producers raise events that are consumed by observers. The observers can optionally specify a combination of "selectors" using Qualifiers. The event object can carry state from producer to consumer. Also the observers can be notified immediately, or wait until the end of the current transaction.

This section will explain how to produce/observe a CDI event. The `TestServlet` will produce an event if a new customer is added to the database. The `CacheSingletonBean` will observe the event and refresh the cache out of schedule.

18.1 Right-click on the project, select "New", "Java Class...", and give the name as "NewCustomerAdded". Add the following code to this generated class:

```
private int id;

public NewCustomerAdded(int id) {
    this.id = id;
}

public int getId() {
    return id;
}
```

This `id` field is used to communicate the customer identifier that is added to the database. Alternatively, it can also pass the complete `Customer` object and then `CacheSingletonBean` can update the cache without querying the database.

18.2 Inject an event in the "TestServlet" as:

```
@Inject Event<NewCustomerAdded> newCustomerEvent;
```

Resolve the imports by choosing the class `javax.enterprise.event.Event` as shown. Note, this is not the default value.

18.3 In the `TestServlet.processRequest` method, add the following code fragment:

```
newCustomerEvent.fire(new NewCustomerAdded(Integer.parseInt(id)));
```

at the end of the `if` block that adds a new customer to the database. The customer identifier is passed as the event state. The `if` block's updated fragment looks like:

```
    out.println("<h2>Customer with '" + Integer.parseInt(id) + "' id
added.</h2>");
    newCustomerEvent.fire(new NewCustomerAdded(Integer.parseInt(id)));
}
```

The newly added code is highlighted in bold.

18.4 Add the following method in the `CacheSingletonBean.java`:

```
public void onAdd(@Observes NewCustomerAdded event) {
    System.out.println("Processing the event: " + event.getId());
    preFetchCustomers();
}
```

This method listens for the `NewCustomerAdded` event published from any producer, `TestServlet` in our case.

18.5 Accessing the URL "http://localhost:8080/JavaEE6SampleApp/TestServlet?add=6" in a browser, adds a new customer to the database, and displays the output as shown.

This confirms that the customer is added to the database.

The interesting part is shown in the GlassFish server log as:

```
INFO: Processing the event: 6
```

This shows that the event was produced by `TestServlet` and observed/consumed by `CacheSingletonBean`.

19.0 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 GlassFish and NetBeans.

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

20.0 Troubleshooting

20.1 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.
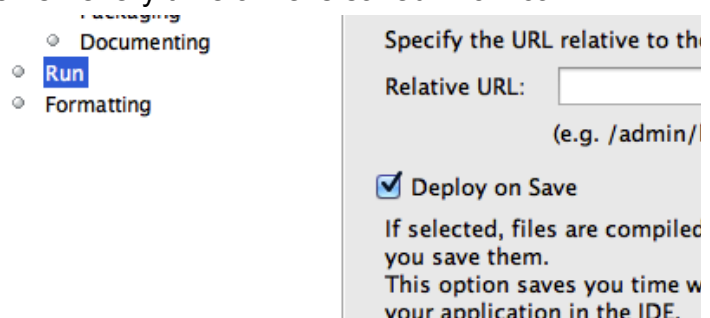
20.2 How can I show the SQL queries issued to the database ?

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

```
<properties/>
```
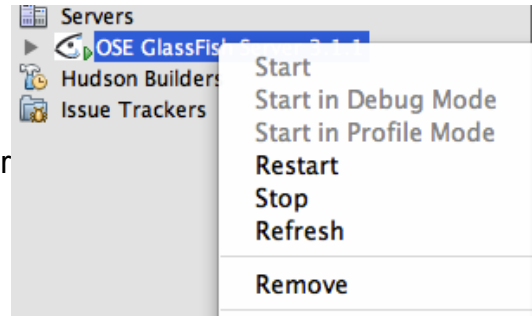
with

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

20.3 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".

20.4 How to recreate the NetBeans sample database ?

1. Find the location of database in "Services" tab of NetBeans, right-clicking on "JavaDB". The default location is .netbeans-derby in your home directory. Close NetBeans and delete this directory.

2. Remove .netbeans directory in your home directory.

3. Start NetBeans and the database will be recreated and new records will deployed into Sample database.

20.5 I accidentally closed the GlassFish output log window. How do I bring it back ?

In "Services" tab of NetBeans, expand "Servers", choose the GlassFish node, and select "View Domain Server Log".
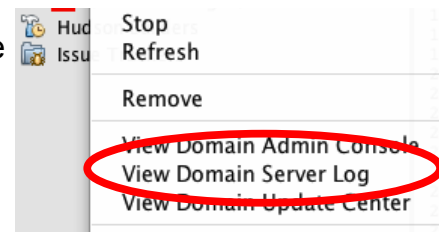
20.6 The method with `@Schedule` are not printing the output on desired intervals.

The GlassFish output window may be showing the following log message:

```
keepstate is true and will not create new auto timers during
deployment.
```

This can be fixed if you right-click on the project and select "Deploy", right-click again now select "Run".

20.7 The namespace prefix for the composite component is not getting resolved.

This may happen if the namespace prefix was manually specified during the composite component creation. In this case you need to add the following namespace prefix mapping to the .xhtml document:

```
xmlns:ez="http://java.sun.com/jsf/composite/ezcomp"
```

20.8 I clicked `Cancel` when the wizard for specifying the base URL for the RESTful resources showed. How do I generate the class now ?

Right-click on "org.glassfish.samples" package, select "New", "Java Class...", specify the name as "ApplicationConfig". Change the generated source code to:

```
@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends javax.ws.rs.core.Application {
}
```

20.9 The database server is not getting started.

Here are a few items to check for:
- In "Services" tab, right-click on "JavaDB" and click on Properties. It shows the location of JavaDB and the directory where the sample database is installed. Make sure both the directories exist.
- Manually start the database by right-clicking on "JavaDB" and selecting "Start Server".
- Refer to 20.4 if the sample database has no records.

21.0 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!

22.0 Completed Solutions

The completed solution is available as a NetBeans project at:

https://blogs.oracle.com/arungupta/resource/JavaEE6SampleApp-Nov2011.zip

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