JSF2 + Primefaces3 + EJB3 & JPA2 Integration Project

April 15, 2012 henk53 Go to comments Leave a comment

Software developer Eren Avşaroğulları recently wrote an interesting blog posting about integrating JSF2, PrimeFaces3, Spring3 and Hibernate and a similar version without Spring.

In this blog posting I would like to present a third variant where EJB3 is used instead of Spring. Additionally, in true open source spirit I also present several code changes that hopefully improve the original code a little (some are taken from the comments on Eren's article). What I did not do is add features or change the structure. This is as much as possible kept the same so the implementations remain comparable.

I'll adhere to the same steps as the Spring blog uses.

Used Technologies:

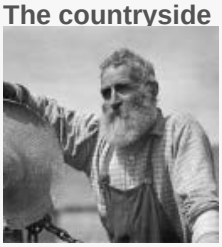
JDK 1.6.0_31
Java EE 6 (JBoss AS 7.1.1, or Glassfish 3.1.2, or TomEE, or ...)
PrimeFaces 3.1.1
H2 1.3.166

STEP 1: CREATE MAVEN PROJECT

Instead of a maven project, I created a dynamic web project in Eclipse:

RSS feed

The countryside



The land and scenery of a rural area, cows and sheep everywhere, the occasional chicken and somewhere there is a crazy old farmer who's deeply involved with IT.

Search

Recent Posts

- The curious case of JBoss AS 7.1.2 and 7.1.3
- Popularity of UI technology on Devoxx '12
- Is there such a thing as the NoMock movement?
- The state of @DataSourceDefinition in Java EE
- Is WebLogic 12c a heavy-weight enterprise solution?

Archives

- January 2013
- November 2012
- September 2012
- June 2012
- May 2012
- April 2012
- October 2011

Categories

- java
- java ee
- Uncategorized

Meta

- Reg
- Log
- Entr
- Com
- Blog

Twitte

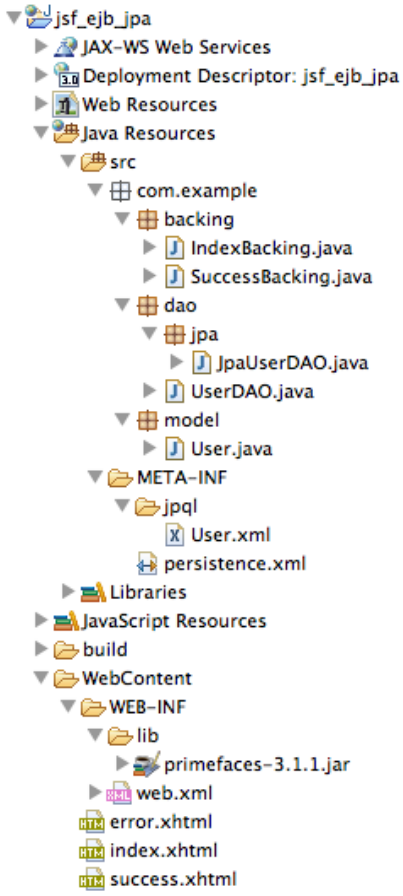
Follow

Follow "henk53"

Get every new post delivered to your Inbox.

Enter your email address

Sign me up



STEP 2: CREATE USER TABLE

Automatically created by JPA (see below)

STEP 3: LIBRARIES

Most libraries (JSF, EJB, JPA, Bean Validation, etc) are already part of Java EE 6 and put on the class-path by an Eclipse Java EE 6 Runtime Server (I used the one from JBoss tools). I downloaded PrimeFaces and H2 separately and put it in WEB-INF/lib (for Glassfish, H2 needs to be put in [GLASSFISH_HOME/lib]). If you do use Maven, I guess this would be in pom.xml:

```
<dependencies>
    <!-- Java EE 6 dependency -->
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>6.0</version>
        <scope>provided</scope>
   </dependency>

    <!-- Primefaces dependency -->
    <dependency>
        <groupId>org.primefaces</groupId>
         <artifactId>primefaces</artifactId>
        <version>3.1.1</version>
    </dependency>
</dependencies>
```

STEP 4: CREATE USER MODEL CLASS

The model is in the core the same as Eren's original. Following the advice from Clean Code, I removed the comments since they basically said the same thing that the code was already saying. I also removed the *@Column* annotations, since all the names already exactly corresponded to the target table. The nullable and unique attributes are only used for schema generation, which wasn't used in the original. For the *@Id* annotation, nullable and unique is

already implied (it's a PK) and for the other columns an annotation that's actually processed at run-rime is much more convenient. I choose to use *@Size* here, which implies not null and at the same time makes specifying this same constraint in the view later on redundant.

For the *toString* method I made a small improvement using *StringBuilder* instead of the unnecessarily synchronized *StringBuffer* and the builder pattern.

```
1  package com.example.model;
2
3  import javax.persistence.Entity;
4  import javax.persistence.Id;
5  import javax.persistence.Table;
6  import javax.validation.constraints.Size;
7
8  /**
9   *
10  * This entity represents the human user.
11  *
12  * @since 12 Apr 2012
13  * @version 1.0.0
14  *
15  */
16 @Entity
17 public class User {
18
19     private int id;
20     private String name;
21     private String surname;
22
23     @Id
24     public int getId() {
25         return id;
26     }
27
28     public void setId(int id) {
29         this.id = id;
30     }
31
32     @Size(min = 5)
33     public String getName() {
34         return name;
35     }
36
37     public void setName(String name) {
38         this.name = name;
39     }
40
41     @Size(min = 5)
42     public String getSurname() {
43         return surname;
44     }
45
46     public void setSurname(String surname) {
47         this.surname = surname;
48     }
49
50     @Override
51     public String toString() {
52         return new StringBuilder()
53             .append("id : ").append(getId())
54             .append(", name : ").append(getName())
55             .append(", surname : ").append(getSurname())
56             .toString();
57     }
58 }
```

STEP 5: CREATE USER MANAGED BEAN CLASS

I made some bigger changes here. In Eren's version there's a managed bean that backs both the *index* and *success* views. I think it's a better practice to split those up into one bean per page, adhering to the rule that a backing bean should back one page only. The names of the beans were thus changed to reflect the page they're backing, instead of having some service like name. There's also no need to explicitly specify the bean's name in the annotation. There's a default, which is good enough here.

I also made the beans *@ViewScoped*, since there's no need to do any re-creation of data after postbacks (e.g. when validation of user entered data fails) and in case of the list page (success.xhtml) to reload the data from the DB every time the table is sorted or paged (which isn't used in this example btw).

A more severe issue was that the original code used the scatter/gather anti-pattern. Here, the

properties of the user were scattered into properties of the backing bean and upon saving gathered again in the model object. In JSF there is no need for this. A backing bean can directly return the model object and UI components can directly bind to those properties.

Navigation rules were also done away with in favor of implicit navigation. Navigation rules may have their place, but often you're better off using the simpler implicit navigation, and I think this is one of those cases. I also removed the explicit navigation to a general error page. This can already be handled application wide by Java EE's error-page mechanism. Therefor I just let the exception propagate. If a specific message would have to be shown on screen via a faces message for specific exceptions from the EJB bean, a try/catch would of course still be the best mechanism.

The smallest, but main point of this entire variant of Eren's version, is that an EJB bean is injected instead of a Spring bean.

Finally, here too I removed comments that didn't told us anything that the code wasn't already telling.

index.xhtml backing:

```
1  package com.example.backing;
2
3  import javax.ejb.EJB;
4  import javax.faces.bean.ManagedBean;
5  import javax.faces.bean.ViewScoped;
6
7  import com.example.dao.UserDAO;
8  import com.example.model.User;
9
10 /**
11  *
12  * This backing bean backs index.xhtml, which allows adding a new User.
13  *
14  * @since 12 Apr 2012
15  * @version 1.0.0
16  *
17  */
18 @ViewScoped
19 @ManagedBean
20 public class IndexBacking {
21
22     private User user = new User();
23
24     @EJB
25     private UserDAO userDAO;
26
27     /**
28      * Adds a user to persistent storage
29      *
30      * @return String - navigation to the success page
31      */
32     public String addUser() {
33         userDAO.add(user);
34         return "success?faces-redirect=true";
35     }
36
37     /**
38      * Resets the user data back to the initial values.
39      *
40      */
41     public void reset() {
42         user = new User();
43     }
44
45     public User getUser() {
46         return user;
47     }
48
49 }
```

success.xhtml backing:

```
1  package com.example.backing;
2
3  import java.util.List;
4
5  import javax.annotation.PostConstruct;
6  import javax.ejb.EJB;
7  import javax.faces.bean.ManagedBean;
8  import javax.faces.bean.ViewScoped;
9
10 import com.example.dao.UserDAO;
11 import com.example.model.User;
12
13 /**
```

```
 *
 * This backing bean backs success.xhtml, which shows a list of all users.
 *
 * @since 12 Apr 2012
 * @version 1.0.0
 *
 */
@ViewScoped
@ManagedBean
public class SuccessBacking {

    private List<User> users;

    @EJB
    private UserDAO userDAO;

    @PostConstruct
    public void init() {
        users = userDAO.getAll();
    }

    public List<User> getUsers() {
        return users;
    }

}
```

STEP 6: CREATE IUserDAO INTERFACE

I took over most of the DAO interface, but made some small changes again.

For starters, I don't think the interface should mention that it's an interface, so I removed the *I* from the name. Continuing with naming, the DAO is already about entity *User* so I simplified the method names by removing this term from them.

In Eren's code the methods all had the *public* access modifier, but for interfaces this is already the default and can thus be safely removed. Finally, there was again comment that said the same as the code was saying, so I removed that too.

```
package com.example.dao;

import java.util.List;

import com.example.model.User;

/**
 *
 * Interface for DAOs that know how to perform persistence operations for User
entities.
 *
 * @since 12 Apr 2012
 * @version 1.0.0
 *
 */
public interface UserDAO {

    void add(User user);

    void update(User user);

    void delete(User user);

    User getById(int id);

    List<User> getAll();
}
```

STEP 7: CREATE UserDAO CLASS

The main meat of this variant of the example application is a DAO implemented with EJB instead of Spring. Since the implementation class uses a specific technology, I named this after the technology. Since both JPA and EJB are used, one possible name could have been *EjbJpaUserDAO*, but feeling this was a bit over-the-top I settled for *JpaUserDAO* instead.

With respect to the JPA code being used, I took advantage of typed queries, where the original code used untyped queries with casts and raw collection types. For the *getAll* query I used a named query in a separate XML file, instead of embedding JPQL inline. To keep the example short, embedding JPQL would certainly be preferable, but I feel JPQL is best at home in its own file.

```java
package com.example.dao.jpa;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import com.example.dao.UserDAO;
import com.example.model.User;

@Stateless
public class JpaUserDAO implements UserDAO {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public void add(User user) {
        entityManager.persist(user);
    }

    @Override
    public void update(User user) {
        entityManager.merge(user);
    }

    @Override
    public void delete(User user) {
        entityManager.remove(
            entityManager.contains(user) ? user : entityManager.merge(user)
        );
    }

    @Override
    public User getById(int id) {
        return entityManager.find(User.class, id);
    }

    @Override
    public List<User> getAll() {
        return entityManager.createNamedQuery("User.getAll", User.class)
                            .getResultList();
    }

}
```

META-INF/jpql/User.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_2_0.xsd">

    <named-query name="User.getAll">
        <query>
            SELECT
                _user
            FROM
                User _user
        </query>
    </named-query>

</entity-mappings>
```

STEP 8: CREATE IUserService INTERFACE

STEP 9: CREATE UserService CLASS

Here I made the opposite decision as I made for the separate JPQL file and decided to skip the service. As in Eren's example the service doesn't add any code and just wraps the DAO, I felt it wasn't necessary here. Normally having a separate service layer, where services aggregate multiple DAOs and perform additional tasks other than just plain persistence (like e.g. validation) is a good thing of course.

STEP 10: CREATE applicationContext.xml

The file *applicationContext.xml* is a Spring specific artifact and thus doesn't appear in this Java EE

example. Nevertheless, some elements have Java EE equivalents and need to be created as well.

The first three entries in Eren's *applicationContext.xml* concern the declaration and injection of Spring beans. In Java EE this is done via annotations and doesn't need to be declared in an XML file.

We do need a datasource though. In case of JBoss AS, the simplest thing for an example would be to use the default datasource (which points to an internal embedded database), but to stay closer to Erin's example we'll define a datasource ourselves. And this brings us to a sore point in Java EE.

In the traditional Java EE philosophy, a datasource is akin to an environment variable. It's defined completely outside the application and the application only refers to it by name and expects nothing more than the *javax.sql.(XA)DataSource* interface.

This is all fine and well for decoupling and giving AS implementations the opportunity to introduce fancy optimizations, but it necessarily means a Java EE application has to be packaged with unresolved dependencies. Eventually someone has to resolve those, which means a traditional Java EE application needs to be accompanied by some kind of document intended to be read and interpreted by humans, who will then need to create all those dependencies in the local environment. If a new version of the application is shipped, some kind of document needs to be checked for changes, and someone has to apply those changes to the local environment.

Obviously some people are a big fan of this approach, but it's not suited for everyone. Recognizing this, Java EE 6 introduced an alternative where the datasource is defined inside an application (just like Spring does). For this an annotation can be used (which might be suited for development/testing), an entry in web.xml (web apps) or an entry in application.xml (EAR apps).

Unfortunately, at the moment this approach hasn't been fully adopted yet. JBoss for instance is apparently a big fan of the traditional approach and it seems that it has only grudgingly implemented the standard embedded datasource definition. It didn't really worked at all in JBoss AS 6, and still doesn't work completely in the latest JBoss AS 7 (it only works when transactions are turned off) and only seems to be recommended for testing and development. It's also not really clear where the necessary JDBC driver has to be located. The spec is silent about this. As it turns out, Glassfish required the driver to be in *glassfish/lib*, while JBoss AS 7 allows the driver to be in WEB-INF/lib.

Despite the not entirely optimal situation, I'll use the standard Java EE 6 datasource definition for this example. For serious production work the reader is advised to look into the implementation specific ways to do the same.

Finally, the Spring *applicationContext.xml* defines the *SessionFactory*, which directly corresponds to a *Persistence Unit* in Java EE. This is defined in META-INF/persistence.xml. Note that schema creation is not standardized in JPA, so it contains properties for both Hibernate, EclipseLink and OpenJPA to autogenerate the SQL schema. If you create the schema yourself, you obviously don't need those.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="example">

        <jta-data-source>java:app/MyApp/myDS</jta-data-source>

        <mapping-file>META-INF/jpql/User.xml</mapping-file>

        <properties>
            <!-- Hibernate -->
            <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="create-drop" />

            <!-- EclipseLink  -->
            <property name="eclipselink.ddl-generation" value="create-tables" />
            <property name="eclipselink.ddl-generation.output-mode"
value="database" />

            <!-- OpenJPA -->
            <property name="openjpa.jdbc.SynchronizeMappings"
value="buildSchema(ForeignKeys=true)"/>
        </properties>

    </persistence-unit>
</persistence>
```

STEP 11: CREATE faces-config.xml

In JSF *faces-config.xml* is optional, and since in Java EE all other elements are by default integrated with it and we didn't used explicit navigation rules, I could leave it out.

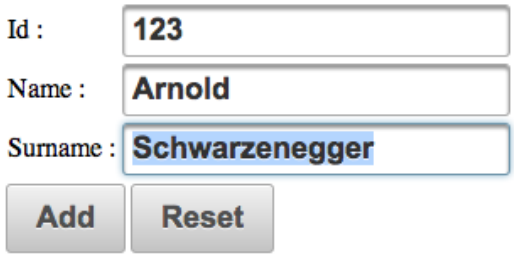
STEP 12 : CREATE web.xml

In Java EE, web.xml is optional too if you're happy with all the defaults. In this case I wanted to set an explicit welcome- and error page, as well remap the FacesServlet from it's default of .jsf to .xhtml. Furthermore in a web application this is the place where the above discussed datasource definition needs to be placed.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
>

    <display-name>jsf_ejb_jpa</display-name>

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

    <error-page>
        <error-code>500</error-code>
        <location>/error.xhtml</location>
    </error-page>

    <servlet>
        <description>The JSF Servlet</description>
        <servlet-name>facesServlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>facesServlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>

    <!-- Following param only needed for MyFaces (TomEE, etc) -->
    <context-param>
        <param-name>org.apache.myfaces.SERIALIZE_STATE_IN_SESSION</param-name>
        <param-value>false</param-value>
    </context-param>

    <data-source>
        <name>java:app/MyApp/myDS</name>
        <class-name>org.h2.jdbcx.JdbcDataSource</class-name>
        <url>jdbc:h2:~/mydb;DB_CLOSE_DELAY=-1</url>
        <user>sa</user>
        <password>sa</password>
        <!--For JBoss AS 7.1.1 must be false.
            For JBoss AS 7.1.2/7.1.3 (EAP 6.0.0/6.0.1) must be true
        -->
        <transactional>false</transactional>
        <max-pool-size>10</max-pool-size>
        <min-pool-size>5</min-pool-size>
        <max-statements>0</max-statements>
    </data-source>

</web-app>
```

STEP 13: CREATE index.xhtml

I made a few small changes here. Instead of the *table* tag I used the somewhat higher level *panelGrid*. I also removed the explicit *Integer* converter, and since the minimal length constraint is already on the model now (and JSF is capable of reading that) the *validateLength* validator wasn't needed either.




```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p="http://primefaces.org/ui">

    <h:head>
        <title>Welcome to the JSF EJB JPA example project</title>
    </h:head>

    <h:body>
        <h:form>

            <h:panelGrid columns="3">

                <h:outputLabel for="id" value="Id : " />
                <p:inputText id="id" value="#{indexBacking.user.id}">
                    <p:ajax event="blur" update="idMsg" />
                </p:inputText>
                <p:message id="idMsg" for="id" display="icon" />

                <h:outputLabel for="name" value="Name : " />
                <p:inputText id="name" value="#{indexBacking.user.name}">
                    <p:ajax event="blur" update="nameMsg" />
                </p:inputText>
                <p:message id="nameMsg" for="name" display="icon" />

                <h:outputLabel for="surname" value="Surname : " />
                <p:inputText id="surname" value="#{indexBacking.user.surname}">
                    <p:ajax event="blur" update="surnameMsg" />
                </p:inputText>
                <p:message id="surnameMsg" for="surname" display="icon" />

            </h:panelGrid>

            <p:commandButton id="addUser" value="Add" action="#
{indexBacking.addUser}" ajax="false" />
            <p:commandButton id="reset" value="Reset" action="#
{indexBacking.reset}" ajax="false" />

        </h:form>
    </h:body>
</html>
```

STEP 14: CREATE success.xhtml

Again a few changes here. For starters *h:outputText* wasn't needed (and if it was needed could be collapsed). I placed the EL expression directly on the Facelet. Also, a PrimeFaces column has a *headerText* attribute, which is less verbose than a Facet, so I used that.


```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p="http://primefaces.org/ui">

    <h:head>
        <title>List of all users</title>
    </h:head>

    <h:body>
        USERs:
        <h:form>
            <p:dataTable id="users" var="user" value="#{successBacking.users}"
style="width: 10%">
                <p:column headerText="ID">
                    #{user.id}
```

```
                </p:column>

                <p:column headerText="Name">
                    #{user.name}
                </p:column>

                <p:column headerText="Surname">
                    #{user.surname}
                </p:column>
            </p:dataTable>
        </h:form>
    </h:body>

</html>
```

STEP 15: CREATE error.xhtml

On Eren's version of this page a form was placed, but this isn't needed for simple text output so I removed it. I also removed the *h:outputText* tag again and changed the text shown, since it's really about any error and not just about transaction errors.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">

    <h:head>
        <title>Application error</title>
    </h:head>

    <h:body>
        Error has occurred!
    </h:body>

</html>
```

STEP 16 : DEPLOY PROJECT

After the **JSF_EJB_JPA** project is deployed to a Java EE 6 implementation (JBoss AS, Glassfish, etc), the index page can be opened via the following URL: *http://localhost:8080/jsf_ejb_jpa*

That's it! Thanks again to Eren Avşaroğulları for his inspiring series of articles and I hope my contribution is useful to someone.

Share this:

Uncategorized ejb, glassfish, java, jboss, jpa, jsf, programming

Leave a comment Trackback

Trackbacks (2) Comments (8)

oussema TOUKEBRI

April 17, 2012 at 12:54 pm

Reply

is it correct that you called DAO from Managed Bean and ommited BO even it is a call this broke 3 Layer Disign Patern

henk53

April 18, 2012 at 9:10 pm

Reply

Oussema, I'm not entirely sure what you mean with "omitted BO". Do you mean I omitted the Business Object? This one actually is still there, it's the *User* class.

What I omitted is the service. I'm a big fan of layering, and as I explained a service is important when it contains additional business logic and calls out to multiple DAOs and/or other services. Also, a not necessarily entirely friendly (remote) client would of course get a service and not directly the DAO.

In this specific case (CRUD), the service was fully wrapping the DAO and adding nothing. That's why I felt it could be left out in this particular case. Another consideration is that in JPA the EntityManager is already a kind of DAO. Not exactly maybe, but close to some. So close, that those same people feel that in JPA an explicit DAO is also not needed.

I do feel a DAO is needed/convenient, but the DAO in this code example is thus already more akin to a service than to a DAO in the traditional sense.

Specifically, the 3 layers are there. The first layer is the UI (presentation), which has the Facelet and the backing bean, the second tier is the DAO, and the third tier is the actual database, which is H2 in my example.

Mohamed Abd El-Naby

July 18, 2012 at 1:25 pm

Reply

you are greaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaat many thanks

Van

September 17, 2012 at 9:14 am

Reply

JSF makes java web programming easier. Is there a way to use EJB with Tomcat?

henk53

September 20, 2012 at 8:10 pm

Reply

Yes, EJB can be added to Tomcat. One example is OpenEJB.

But it's easier to download TomEE. That's Tomcat, but with OpenEJB and JSF (MyFaces) already added.

Asme

January 16, 2013 at 9:11 pm

Reply

hi, can you please CRUD function too i only see create option, no delete, no

update

goodguy

April 22, 2013 at 2:54 pm

Reply

Hello,
Thanks for the inspiring tutorial Would anyone provide a link to the source code please ?
Much appreciated

Ejb3

June 18, 2013 at 9:27 pm

Reply

Yes. We appreciate so much if you publish a sourcecode.
Thanks. I need make a test..

Leave a Reply

Is WebLogic 12c a heavy-weight enterprise solution?

Reply to Comparing Java Web Frameworks