Ohioedge J2eeBuilder Toolkit

Advanced Developer's Guide

By Sandeep Dixit

March 30, 2003

Introduction

Ohioedge J2eeBuilder Toolkit is a library of generic implementations of J2EE core patterns and practices such as Value Object, Business Delegate, Service Locator, Connection Factory, Intercepting Filter, Front Controller, Service To Worker, Dispatcher View, View List Handler, Composite View, Session Façade, Authentication, and Session Management essential for building Web-based applications. Ohioedge J2eeBuilder also includes a set of generic Ant scripts for that make packaging of components and composing of applications a breeze. *Why spend months of development time programming common, standard J2EE infrastructure code? Why spend thousands of dollars on proprietary J2EE IDE when you can go the open source route? Spend time and money learning J2EE and not on how to use vendor-specific IDEs.*

Building applications using Ohioedge J2eeBuilder Toolkit consists of the following three steps:

1. Write your business component by extending abstract classes and/or interfaces from org.j2eebuilder packages.
2. Plug-in the business component into your application by adding the component's Unified Component Definition (UCD) into the application's J2eeBuilder Application Definition (JAD) file - j2ee-builder.xml
3. Using Ant scripts, create the component libraries, create the component distribution, and compose the application EAR.

Through out this guide, I will refer to the Employee business component to explain how to use Ohioedge J2eeBuilder for building Web-based, EJB-JavaBean-JSP tier J2EE applications. In order to create an environment similar to the one I've referred in this guide, follow the instructions given in the next couple of sections and download, install and configure the necessary files.

Download

Get the latest distribution of Ohioedge J2eeBuilder Toolkit from http://j2eebuilder.sourceforge.net. The distribution file is named using the nomenclature below:

- **oefnd***<Version>*-*<Applicatin Server>*-*<Database>*.zip

oefnd is Ohioedge Foundation business application, an organization management application built using J2eeBuilder. It consists of Organization, State, Employee, and Name Prefix/Suffix/Title business components.

Installation and setup

Extract the distribution on your drive. The extracted directory structure should look similar to the one shown in Figure below.

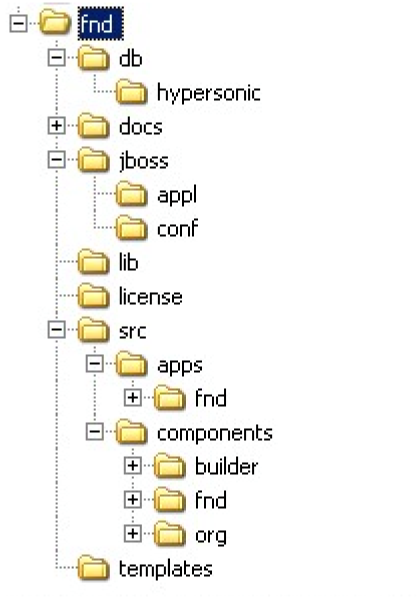


Figure 1. Distribution directory structure

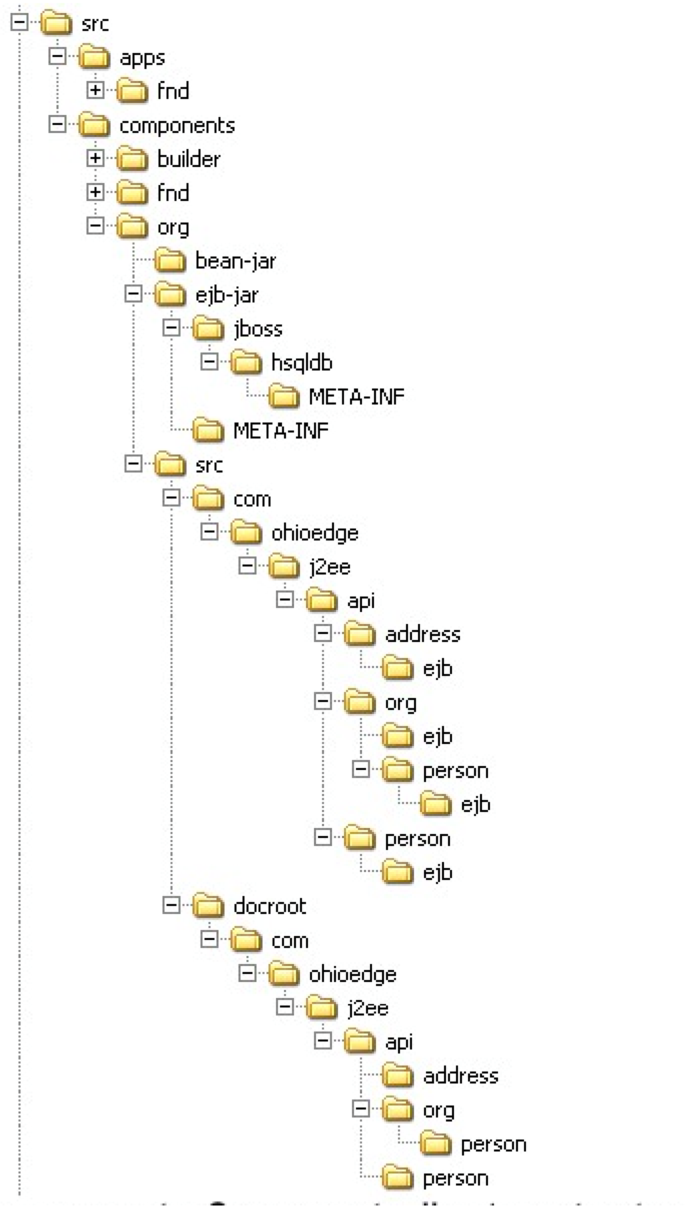
Ohioedge J2eeBuilder is driven by Apache's Ant tool. So it is important that you thoroughly familiarize yourself with the directory structure and the Ant build files. As shown in Figure 1, the source code directory 'src' contains 'apps' and 'components' directories. The 'apps' directory contains the source code and build scripts specific to the application ('fnd.') The 'components' directory contains the J2eeBuilder directory – 'builder', an <application-component> directory ('fnd' in Figure 1) and one directory for every J2eeBuilder business component ('org' in Figure 1.)

Before we move forward, let's first review frequently used definitions in the context of this document.

- ‘**a business component**’ is a generic reference to a business functionality that is complete in it self. For example, an employee or a name title of a person. It has nothing to do with the source code. It is purely a functional reference.
- ‘**an EJB component**’ means just that, an Enterprise JavaBean. It is a reference to the source code that makes that EJB.
- ‘**a JavaBean component**’ means just that, a JavaBean. It is a reference to the source code that makes that JavaBean.
- ‘**a JSP component or a JSP page**’ means just that, a JSP. It is a reference to the source code that makes that JSP.
- ‘**a three-tier business component**’ is a reference to the collection of EJB, JavaBean, and JSP components of a single business component. For example, collectively EJB, JavaBean and JSP components of an employee business component is referred as a three-tier employee component. Each three-tier business component has a UCD.
- ‘**a J2eeBuilder business component**’ is a reference to a collection of one or more three-tier business components. Each J2eeBuilder business component consists of,
  1. One JavaBean library (.jar) consisting of all its JavaBean components. One MANIFEST.INF file listing the contents of the JavaBean library.
  2. One EJB library (.jar) consisting of all its EJB components. One MANIFEST.INF file listing the contents of the EJB library.
  3. One EJB (.jar) consisting of one EJB definition (ejb.xml) file, its JavaBean library JAR, its EJB library JAR and one MANIFEST.INF file with class path references to other required JavaBean and/or EJB library JARs.
  4. One docroot directory that holds all the JSP pages.
  5. One Ant script (build.xml) for creating the libraries and packaging the created libraries, manifest files, ejb.xml, and any application server specific definition files into an EJB module. Note: An EJB module is not same as an EJB component. One EJB module consists of one or more EJB components. And by the way, the terminology of EJB module and EJB component is used through out the EJB world. It is not specific to our context here.

  For example, in the download you will see that the Organization, State, Employee, Name Prefix, Name Suffix, and Name Title three-tier components are grouped under the Organization J2eeBuilder component. This is similar to the EJB module and EJB component difference mentioned above. Except that in addition to EJB components, a J2eeBuilder business component also includes JavaBean and JSP components.

Figure below shows an expanded structure of ‘src/components/org’ directory:



```
src
    apps
        fnd
    components
        builder
        fnd
        org
            bean-jar
            ejb-jar
                jboss
                    hsqldb
                        META-INF
                META-INF
            src
                com
                    ohioedge
                        j2ee
                            api
                                address
                                    ejb
                                org
                                    ejb
                                    person
                                        ejb
                                person
                                    ejb
                docroot
                    com
                        ohioedge
                            j2ee
                                api
                                    address
                                    org
                                        person
                                    person
```

Figure 2. Components <Component> directory sturcture

__Building applications__

1. Write your business component by extending abstract classes and/or interfaces from org.j2eebuilder packages

1.1 Writing an EJB

Employee EJB is implemented as a Local EJB (Local and LocalHome interfaces).
First let's look at the local interface of Employee bean:
Employee Local Interface:

```
package com.ohioedge.j2ee.api.org.person.ejb;
public interface Employee extends javax.ejb.EJBLocalObject,
          org.j2eebuilder.util.ejb.ValueObjectHandler,
          org.j2eebuilder.model.ejb.Signature {
}
```

Wait a minute, there is nothing in this class! Yes. That's right.
org.j2eebuilder.util.ejb.ValueObjectHandler interface defines getDataVO() and setDataVO(ValueObject vo) methods which is all required of any EJB remote/local interface! This is a Value Object J2EE pattern.

Let's look at the Employee EJB implementation class and see how the value object pattern is generalized and how to use it. Below is the relevant code from EmployeeEJB

EmployeeEJB.class:

```
package com.ohioedge.j2ee.api.org.person.ejb;
public abstract class EmployeeEJB
          extends org.j2eebuilder.model.ejb.SignatureAbstract
          implements EntityBean {

/**
*         load readable properties from this object into the ValueObject class which is
*         provided as an input parameter.
*         @param - primaryKey
*         @param - reference of this instance
*         @param - ValueObject class
*/
public org.j2eebuilder.util.ValueObject getDataVO()
throws org.j2eebuilder.util.ejb.ValueObjectHandlerException  {
          return org.j2eebuilder.util.ejb.ValueObjectHandlerHelper.getDataVO(
                    this.ctx.getPrimaryKey(),
                    this,
                    com.ohioedge.j2ee.api.org.person.EmployeeBean.class);
}
```

```
/**
*         copy updateable properties from ValueObject to this object
*/
public void setDataVO(org.j2eebuilder.util.ValueObject ValueObject,
            org.j2eebuilder.ComponentDefinition componentDefinition,
Integer mechanismID)
throws org.j2eebuilder.util.ejb.ValueObjectHandlerException {
                        org.j2eebuilder.util.ejb.ValueObjectHandlerHelper.setDataVO(
                                    this, ValueObject, componentDefinition);
                        setLastModifiedOn(new java.sql.Timestamp(
 (new java.util.Date()).getTime()));
                        setLastModifiedBy(mechanismID);
}
```

That's it! Again, not much code here. Simply copy and paste these two methods into your EJB implementation class and you are done with the value object pattern. Only thing that will be specific to your EJB implementation class is the ValueObject class parameter (in our case - com.ohioedge.j2ee.api.org.person.EmployeeBean.class) of the getDataVO() method. Ohioedge J2eeBuilder core libraries take care of 1) the loading of data into the value object from ‘this’ EJB instance’s CMP fields, in case of the getDataVO() method, and 2) updating 'this' EJB instance's CMP fields with the values provided by the passed in value object, in case of the setDataVO(ValueObject vo) method.

As you must have figured out by now, com.ohioedge.j2ee.api.org.person.EmployeeBean extends the org.j2eebuilder.util.ValueObject interface. J2eeBuilder’s value object handler mechanism is designed to handle/manipulate any JavaBean that extends the org.j2eebuilder.util.ValueObject interface. This makes it generic. We will take a look at com.ohioedge.j2ee.api.org.person.EmployeeBean in the next section. Before that, lets first wrap-up our EJB implementation class discussion. As you must have also noticed, besides the ValueObjectHandler interface, both Employee Local and EJB implement/extend org.j2eebuilder.model.ejb.Signature and org.j2eebuilder.model.ejb.SignatureAbstract interface/class respectively. The signature interface/abstract class defines/implements getter/setter methods of the createdOn, createdBy, lastModifiedOn, and lastModifedBy CMP fields. If you decide to have these four fields into your every EJB (and thus underlying database table) use it, else don't. It is entirely up to you. However having these four fields into your EJB (and database table) will come handy down the road for searching records by creation/modification dates or in general auditing of data. I *recommend* having these four fields in every EJB. Below is the code of Signature interface:

```
package org.j2eebuilder.model.ejb;
public interface Signature extends java.io.Serializable {
  java.sql.Timestamp getCreatedOn();
  Integer getCreatedBy();
  java.sql.Timestamp getLastModifiedOn();
  Integer getLastModifiedBy();
}
```

Before we proceed to the next section on how to write JavaBeans, a quick note on getName() and getDescription(); I *recommend* having 'name' and 'description' CMP fields into every EJB (a recommendation that I didn't follow while writing Employee EJB. Oops!) Why? Well, the ValueObject interface extends the ListElement interface, an interface used by J2eeBuilder's value list pattern, and the ListElement interface requires getName() and getDescription() methods. In most cases, I think it is lot easier to calculate and store the values of these fields one time during the creation of an object rather than calculating them dynamically every-time an object is accessed. In any case, you have to implement these two methods in your JavaBeans (ValueObject). If you don't, the output generated by the list mechanism would look something like:

```
null, null
null, null
null, null
.
.
.
```

Get the picture? I will discuss the value list pattern in Appendix A.

NOTE: I plan to move the ValueObject interface/abstract class out of the org.j2eebuilder.util package in to the org.j2eebuilder.view package in the next release. This is definitely a TO-DO. Please pardon me until then.

1.2 Writing a JavaBean

Now let's look at com.ohioedge.j2ee.api.org.person.EmployeeBean. As you have seen in the section above, it implements the ValueObject interface and thus is used as a value object for getting data in and out of EJB. In the next section you will see that it is also used as a ‘useBean’ in Employee JSP pages. This is how Ohioedge J2eeBuilder threads EJB, JavaBean, and JSP tiers together.

EmployeeBean.class:

```
package com.ohioedge.j2ee.api.org.cust;
public class EmployeeBean extends org.j2eebuilder.view.ValueObjectImpl {
  // gettter/setter method
.
.
.
 // search method
public Collection search(Integer orgID, String criteria) {
Collection col = new java.util.HashSet();
        try {
                OrganizationManagerHome home = (OrganizationManagerHome)
                        ServiceLocatorBean.getInstance().getHome(
                        "ejb/OrganizationManager", OrganizationManagerHome.class);
                if (criteria != null) {
                        OrganizationManager organizationManager = home.create();
                        col = organizationManager.findColOfEmployeeVOByLastName(
orgID, criteria);
                }
        } catch(Exception e) {
        log.error(this.getClass().getName()+".search():" + e.toString());
        }
        return col;
}
```

This is how a minimum JavaBean should look like. If you don't know what the getter and setter methods are, you probably qualify as a project manager. The search method is required to return a collection of value objects. Here also, Ohioedge J2eeBuilder makes your life easier by providing a value object factory class - ValueObjectFactory. The value object factory class provides a method for converting a collection of Remote/Local objects (returned by EJB Home finder methods) into a collection of value objects. I will discuss the value object factory class in Appendix B.

Take a look at OrganizationManager, a facade session bean. As shown in the example above, it is used to call the findColOfEmployeeVOByLastName() method. Again, it is simply a session façade that calls the finder/select methods of the Employee EJB. Below is the findColOfEmployeeVOByLastName () method for your reference. Without getting into any details, I would like to point out the use of ServiceLocatorBean in this example. I will be covering the service locator pattern later in Appendix C.

```
public Collection findColOfEmployeeVOByLastName(Integer orgID, String name)
```

```
throws SessionException, RemoteException      {
        try {
                EmployeeHome home = (EmployeeHome)
                        ServiceLocatorBean.getInstance().getLocalHome(
                        "ejb/Employee", EmployeeHome.class);

                Collection col = home.findByLastName(orgID, name);
                return ValueObjectFactory.getInstance().getCollectionOfVO(col);
        } catch (Exception re)          {
                throw new SessionException(re.getMessage());
        }
}
```

That’s all there is to writing a JavaBean component. You must be wondering where in the world are create, delete and update methods? Well, they don’t have to exist! Ohioedge J2eeBuilder takes care of these methods for you! By extending the ValueObjectImpl interface, every JavaBean component inherits these methods. You don't have to write these methods unless of course you need to overwrite them for some reason. This WILL save you tons of code and months of development time. I guarantee it! This is J2eeBuilder’s business delegate pattern. I will discuss it further in Appendix D.

That's all there is to writing a JavaBean component. Only the getter/setter and search(){} methods. No create, update, and delete methods!

1.3 Writing JSP

For every component, Ohioedge J2eeBuilder Toolkit requires you to write at least two JSP pages: - 1) a controller JSP and 2) a data maintenance JSP. Let's look at a controller JSP first.

Controller JSP

Employee.jsp

```
<%@ page contentType="text/html;charset=ISO-8859-1"%>
<%@ page import=" java.sql.*, java.util.*" %>
<%@ include file="/com/ohioedge/j2ee/ApplicationLicense.jsp" %>
<% String componentController = (String)request.getAttribute("componentController"); %>
<% String componentControllerAlias = (String)request.getAttribute("componentControllerAlias"); %>
<JSP:useBean id="employeeBean" scope="session" class="com.ohioedge.j2ee.api.org.person.EmployeeBean"/>
<HTML>
<BODY>
<%
        // keep originalValue of submit
        String originalValue = (String)request.getAttribute("submit");
        // this is value before it goes in bean
        if (originalValue == null || originalValue.equals("Home")) {
```

```
                        out.println("Default page is not defined.");
            } else if (originalValue.equals("Search")) {
                        // display resultset or include listPage here
                        String criteria = (String)request.getAttribute("criteria");
                        Collection rs = null;
                        if (criteria != null) {
                                    rs = employeeBean.search(
sessionBean.getOrganizationID(), criteria);
                        }
                        if (rs != null) {
                                    request.setAttribute("resultset", rs);
                                    %>
                                    <JSP:include page="/ListDefaultAlias" flush="true">
                                    </JSP:include>
                                    <%
                        }
            }
%> <!-- end of if successful -->
 <BODY>
<HTML>
```

Employee.jsp is the controller JSP page of Employee component. What is a controller JSP any way? When a user clicks on a button or a link of a J2eeBuilder-based application's Web page, this is what happens, the request first goes to a view controller (Appendix E.) Once the requested action/command is identified and executed, the control is then passed on to a generic controller JSP - ViewControllerHelper.jsp. The generic controller JSP then passes the control to the controller JSP of the component from where the request originated. Why does a component need to have its own controller.jsp? Because you may want to customize certain things for a component, such as, default home page, menu page, custom searches, etc. As you can see in Employee.jsp, if the value of 'submit' is 'Search', the request ends up at Employee.jsp calling Employee JavaBean's search method. The returned result is passed onto ListDefaultAlias, a default List JSP. The list mechanism is covered later in Appendix A. A component's controller JSP is where its search methods are called. Why have I used JSP? Isn't JSP only for presentation-tier? Well, I have found it a lot easier to use JSP pages for this kind of coding rather than using JavaBean. As you can see, there is not much code in a controller JSP. Mainly it is the 'if-then-else' kind of logic. An important note, don't get confused between a controller JSP and a view controller. A view controller does the handling and processing of requests, whereas a controller JSP gives developers additional capability to further customize the response received from the generic controller JSP.

Now let's take a look at Employee Data Maintenance JSP – EmployeeMaintain.jsp. This JSP page should be fairly straightforward. It is the JSP page where Employee JavaBean's data members (fields) are exposed to users for data management.

Maintenance JSP

EmployeeMaintain.jsp:

```
<JSP:useBean id="employeeBean" scope="session" class="com.ohioedge.j2ee.api.org.person.EmployeeBean"/>
<JSP:setProperty name="employeeBean" property="*" />
<JSP:useBean id="stateBean" scope="session" class="com.ohioedge.j2ee.api.address.StateBean"/>
<HTML>
<BODY>
<FORM ACTION=<%= request.getAttribute("componentControllerAlias") %> METHOD=POST
ENCTYPE="application/x-www-urlencoded">
```

Look at the form action - <%= request.getAttribute("componentControllerAlias") %>. 'componentControllerAlias' is the called component's alias (here, Employee.ctrl.) A component alias is defined in j2ee-builder.xml. Also see the 'id' attribute of JSP:useBeans. It is also defined in a component's UCD.

With this, we are done writing EJB, JavaBean and JSP components. Now let's see how to write UCD and JAD.

2. Plug-in the business component into your application by adding the component's Unified Component Definition (UCD) into the application's J2eeBuilder Application Definition (JAD) file - j2ee-builder.xml

Employee UCD:

```
<component>
  <name>Employee</name>
  <description>Employee</description>
  <controller>/com/ohioedge/j2ee/api/org/person/Employee.jsp</controller>
  <servlet-path>Employee.ctrl</servlet-path>
  <data>/com/ohioedge/j2ee/api/org/person/EmployeeMaintain.jsp</data>
  <menu>/com/ohioedge/j2ee/api/org/person/EmployeeMenu.jsp</menu>
  <JSP-declaration>
    <useBean>
      <id>employeeBean</id>
      <scope>session</scope>
      <class-name>com.ohioedge.j2ee.api.org.person.EmployeeBean</class-name>
    </useBean>
    <useBean>
      <id>employeeBean1</id>
      <scope>page</scope>
      <class-name></class-name>
    </useBean>
  </JSP-declaration>
  <bean-property>
    <property-name>employee</property-name>
  </bean-property>
  <ejb>
    <home>com.ohioedge.j2ee.api.org.person.ejb.EmployeeHome</home>
    <remote>com.ohioedge.j2ee.api.org.person.ejb.Employee</remote>
    <entity>com.ohioedge.j2ee.api.org.person.ejb.EmployeeEJB</entity>
    <jndi>ejb/Employee</jndi>
    <primary-key>
      <class-name>com.ohioedge.j2ee.api.org.person.ejb.EmployeePK</class-name>
      <field-name>employeeID</field-name>
    </primary-key>
    <attribute>
      <name>organizationID</name>
      <type>Integer</type>
    </attribute>
      .
      .
      .
    <attribute>
      <name>zip</name>
      <type>String</type>
    </attribute>
    <write-attributes>
      <attribute-name>employeeName</attribute-name>
      <attribute-name>dunsNumber</attribute-name>
        .
        .
        .
      <attribute-name>zip</attribute-name>
    </write-attributes>
    <factory-methods>
      <factory-method>
        <name>create</name>
        <type>create</type>
        <method-params>
          <method-param>
            <param-seq>1</param-seq>
            <param-value>
              <component-name>Employee</component-name>
              <useBean-id>employeeBean</useBean-id>
              <useBean-class-method>getOrganizationID</useBean-class-method>
            </param-value>
          </method-param>
          .
          .
          .
        </method-params>
      </factory-method>
    </factory-methods>
</ejb>
</component>
```

Fnd Application JAD:

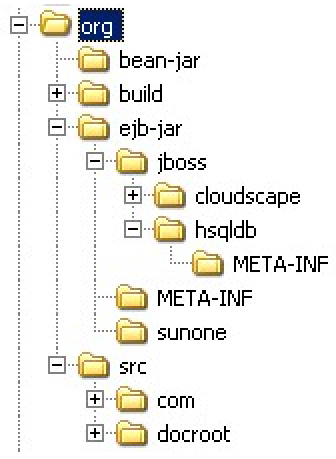
```
<?xml version="1.0" encoding="UTF-8"?>
<application>
        <name>Ohioedge Foundation Application</name>
        <description>Ohioedge Foundation Application consists of basic organizational components such as
organization, state, employee, name prefix, name suffix, and name title.</description>
        <layout>/com/ohioedge/j2ee/fnd/Layout.jsp</layout>
        <component>…</component>
        .
        .
        .
</application>
```

3. Using Ant scripts, create component libraries and package them into the application's ear.

At the core of how components are packaged and applications are composed is the J2eeBuilder directory structure. Below are the three key directories of J2eeBuilder: -

src/components/<component>

All files of a component are in components/<component> directory. Figure 3 below shows org component's directory structure.



```
org
    bean-jar
    build
    ejb-jar
        jboss
            cloudscape
            hsqldb
                META-INF
        META-INF
        sunone
    src
        com
        docroot
```

Figure 3

. The directory structure of the org component

- *org/bean-jar/beanlib-manifest.txt*: - <component>-beanlib.jar's manifest file that lists the content of the jar.
- *org/ejb-jar/ejblib-manifest.txt*: - <component>-ejblib.jar's manifest file that lists the content of the jar.
- *org/ejb-jar/META-INF/manifest.txt*: - ejb.jar's manifest file that includes Class-path references to <component>-beanlib.jar and <component>-ejblib.jar.
- *org/ejb-jar/META-INF/ejb.xml*: - The generic EJB definition file. This file is required by the J2EE specification and has nothing to do with application servers.
- *org/ejb-jar/<application server>/<database>/META-INF:* - The application server

and database specific EJB definition files. In case of JBoss 3.0.4, it is *jbosscmp-jdbc.xml*

- *org/build*: - Used by the build process.
- *org/build/classes*: - Contains compiled classes
- *org/build/lib*: - Contains <component>-beanlib.jar and <component>-ejblib.jar.
- *org/src/*<java source code>: - Contains .java source code
- *org/src/docroot*: - Contains .jsp source code
- *org/build.xml:-* Ant build script for the packaging of component. Every component's build script is identical in every aspect except for its classes to be included in bean & EJB libraries and the class-path attribute of the manifest file of its EJB. Note: In the next release, I plan to eliminate this duplication by moving this script under src/components directory and modifying it to accept three variable values for bean classes, EJB classes, and the class-path attribute. src/<component>/build.xml would call this generic src/components/build.xml by passing these three variables.

In order to add your component to the library, follow these steps: -

Copy the content of one of the existing components into the src/components/<your_component> directory of the component you are creating. Then modify/add the files below for your component: -

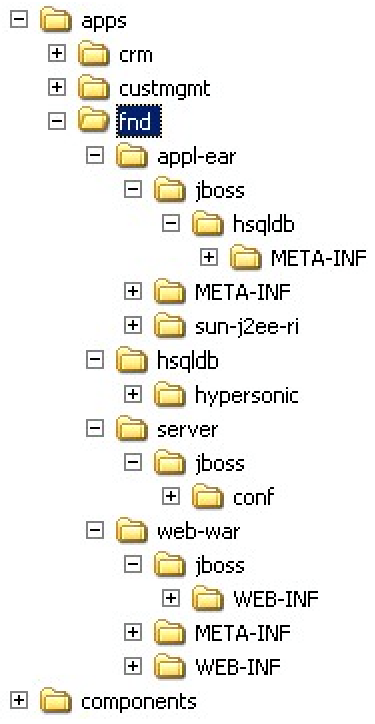
- *src/components/<your_component>/bean-jar/beanlib-manifest.txt*
- *src/components/<your_component>/ejb-jar/ejblib-manifest.txt*
- *src/components/<your_component>/ejb-jar/META-INF/manifest.txt*
- *src/components/<your_component>/ejb-jar/META-INF/ejb.xml*
- *src/components/<your_component>/ejb-jar/<application server>/<database>/META-INF*
- *src/components/<your_component>/src/<java source code>*
- *src/components/<your_component>/src/docroot*
- *src/components/<your_component>/build.xml*

src/components/<application>

For every application, there is components/<application> directory that contains all source files specific to the application. These typically include a session bean and look & feel JSP pages. The structure of src/components/<application> directory is same as src/components/<component>.

src/apps/<application>

For every J2eeBuilder application, there is *src/apps/<application>* directory that contains application EAR and WAR related files, application database files, and any application server specific configuration files. Let's look at apps/<application> directory structure of ‘fnd.’



```
apps
    crm
    custmgmt
    fnd
        appl-ear
            jboss
                hsqldb
                    META-INF
            META-INF
            sun-j2ee-ri
        hsqldb
            hypersonic
        server
            jboss
                conf
        web-war
            jboss
                WEB-INF
            META-INF
            WEB-INF
components
```

- *fnd/appl-ear/META-INF/application.xml*: - Application EAR’s definition file required by the J2EE specification.
- *fnd/appl-ear/META-INF/manifest.txt*: - Application EAR’s manifest file that lists the content of the EAR file.
- *fnd/appl-ear/<application server>/<database>/META-INF*: - Contains definition files required by the *<application server>* application server, configured for the *<database>* database. In case of JBoss, configured for hsqldb, it is *jboss.xml*.
- *fnd/appl-ear/<database>*: - Actual database files used by the application.

- *fnd/appl-ear/server/<application server>*: - Contains configuration files of the *<application server>* application server.
- *fnd/web-war/META-INF*: - Contains the configuration files of application WAR.
- *fnd/ web-war /WEB-INF*: - Contains the manifest file of application WAR.
- *fnd/ web-war /<application server>/WEB-INF*: - Contains the *<application server>* specific configuration files of application WAR. In case of JBoss, it is *jboss-web.xml*.

As explained earlier in the src/components/<component> section, an efficient way to build your application is to copy an existing application directory structure under your application directory structure and then use it as a base to write your application specific files by modifying the existing files.

Summary

Ohioedge J2eeBuilder Toolkit is an open source library of generic implementations of core J2EE patterns and Ant scripts for packaging and composing J2EE components and applications. At the core of Ohioedge J2eeBuilder is a well-thought-out 'generic' directory structure that enables building of J2EE business components and applications for any open source or proprietary J2EE-compatible application server, without ever loosing the generic-ness and open-source nature of your code.

Ohioedge J2eeBuilder Toolkit is a great way to stay with J2EE specification and not get locked into any vendor-specific J2EE development environment. Spend time learning J2EE and not how to use vendor-specific J2EE environment.

Yes you can with J2EE…

Have fun building J2ee applications!

Sandeep Dixit
Chief Architect
Ohioedge
1246 West 70th Street
Cleveland, Ohio 44102

Appendix A: List mechanism

Ohioedge J2eeBuilder Toolkit provides a generic list mechanism. Here is how it works. ValueObject interface extends ListElement interface. The collection returned by the finder methods is a collection of value objects and thus a collection of list elements. Same is true with getDataVO() method. The returned object is of a ValueObject type and thus of a ListElement type. The collection is displayed by List JSP pages. They are under components/builder/src/docroot/components/<your_component>/j2eebuilder/view directory.

Appendix B: ValueObjectFactory

Below are two relevent methods from ValueObjectFactory.class
- Collection ValueObjectFactory.getInstance().getCollectionOfVO(col);
- ValueObject ValueObjectFactory.getInstance().getDavaVO(ValueObjectHandler);

Appendix C: ServiceLocatorBean

ServiceLocatorBean is a JavaBean that provides creation and locating of Home, LocalHome, JNDI, JDBC Connection, etc. services.

Appendix D: BusinessDelegateManager

BusinessDelegateManager is a generic method invoker. The <ejb-ref> and <ejb-local-ref> references of all EJB components must be added to the EJB definition of BusinessDelegateManager.

Appendix E: ViewController

ViewController is the front view controller of your application.

Appendix F: Security Components

J2eeBuilder includes a set of generic security components as below:

Role

The *Role* business component specifies a security role.

Example:

Name	**Description**
Supervisor	User with a Supervisor role
Manager	User with a Manager role
Worker	User with a Worker role

Privilege

The *Privilege* business component specifies a security privilege.

Example:

Name	**Description**
Approve	Privilege to approve an assignment
Sign-off	Privilege to sign-off an assignment
Approve	Privilege to approve an activity
OnHold	Privilege to put activity schedule on hold
Release	Privilege to release activity schedule
Route	Privilege to route activity schedule
Assign	Privilege to assign assignments
Schedule	Privilege to schedule activities
Originator	Privilege to originate or create activities

RolePrivilege

Dependency: **Role**, **Privilege**.

The *RolePrivilege* business component specifies the relationship between a role and a privilege.

Example:

Role	Privilege
Supervisor	Assign: Supervisor can assign Assignments to other Mechanisms Approve: Supervisor can approve an Activity or Assignment signed-off by other Mechanisms Sign-off: Supervisor can sign-off an Assignment Originator: Supervisor can create or originate an Activity
Manager	Schedule: Manager can schedule an Activity to be sent to the next ActivityType Route: Manager can route an Activity to the next ActivityType OnHold: Manager can schedule an Activity to be put on-hold until, certain business conditions are met to release the Activity Release: Manager can schedule an Activity to be released after all the business conditions for that Activity are met
Worker	Sign-Off: Worker can sign-off an Assignment

Group

Dependency: GroupType.

The *Group* business component specifies a group object used in an entitlement definition. In the J2eeBuilder security structure, a mechanism belongs to a group and thus inherits that group's entitlements.

Example:

Name	Description
ApplAdmin	Application administrator
OrgAdmin	Organization administrator
ReportUser	Read-only user

GroupType

A group type is a category of group based on certain attributes. For example, group types could be top-security, general-security, etc.

Entitlement

An entitlement is a definition of a role a group has over an entitlement object. An entitlement object must implement the HierarchyVO interface. This allows all children of an entitlement object to inherit its entitlements.

GroupMechanism

A group mechanism is a definition of a group a mechanism belongs to. By belonging to a group, the mechanism inherits that group's all entitlements.

Figure below illustrates the UML of Generic Security Components along with the UML of Component-level Security Structure.

Appendix G: Component-level Security

J2eeBuilder component-level security is built by integrating the generic security components with the following component schema components:

Component

A component business object holds schema of all business objects. For example:

Name	**Description**
NamePrefix	NamePrefix

A component business object implements the HierarchyVO interface.

ComponentStatusType

The ComponentStatusType business component represents statuses a component is aware of. For example, "create," "update," etc. could be the statuses a component is aware of.

ComponentStatus

The ComponentStatus business object represents a status type associated with a component.

ComponentStatusTypePrivilege

The ComponentStatusTypePrivilege schema component is the link between generic security components and component-level security structure. The ComponentStatusTypePrivilege schema component represents the privileges that are associated with status types. Only a mechanism that ultimately inherits a privilege associated with a component status type is authorized to apply that status type on components.

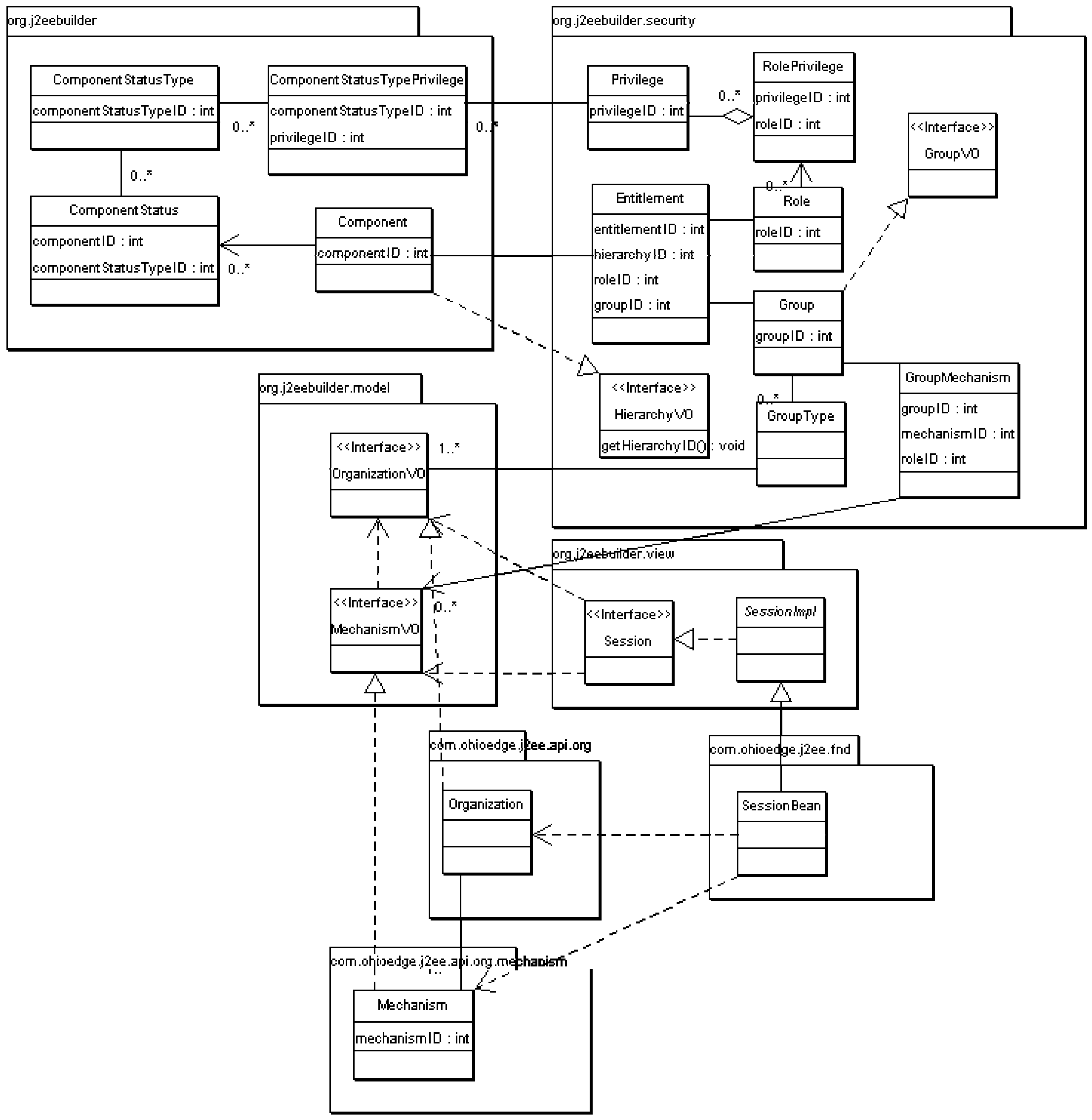


Figure 4. J2eeBuilder Security Model

Appendix H: License

J2eeBuilder licensing mechanism consists of a license generator JavaBean (LicenseGenerator.class,) a license validator JavaBean (LicenseValidator.class) and a license record maintainer J2eeBuilder component (EJB, JavaBean, and JSPs) License. The license source code is under src\components\builder directory and org.j2eebuilder.license package.

The license generator is used for creating the following license files: -

- public.key: A public key is used to validate a message signed by the private key
- private.key: A private key is used to sign a message
- msg.signature: A signature generated by a private key by signing a message.

In order to create your own license, do the following steps:

STEP 1. Create license files by running the license generator program (command_prompt>*java -cp .;build\lib\builder-beanlib.jar* ***org.j2ee builder.license.LicenseGenerator***). It will take you through the following prompts. Values entered are shown in bold. To use default value, just hit <enter>.

1. Enter Licensee Name[Ohioedge]: **MyCompany<enter>**

You should see a response: "You entered licenseeName:MyCompany"

2. Enter License Expiration Date[2003-12-31]: **2010-12-31<enter>**

You should see a response: "You entered expirationDate:2010-12-31"

3. Enter number of organizations[1]: **10<enter>**

You should see a response: "You entered numberOfOrganizations:10"

4. Enter number of users[1000]: **100000<enter>**

You should see a response: "You entered numberOfUsers:100000"

5. Enter the directory location where the license files will be created. You need to have read/write privilege on this directory. [C:\ohioedge\crm\license]: **<enter>**

You should see a response: "You entered directory:C:\ohioedge\crm\license

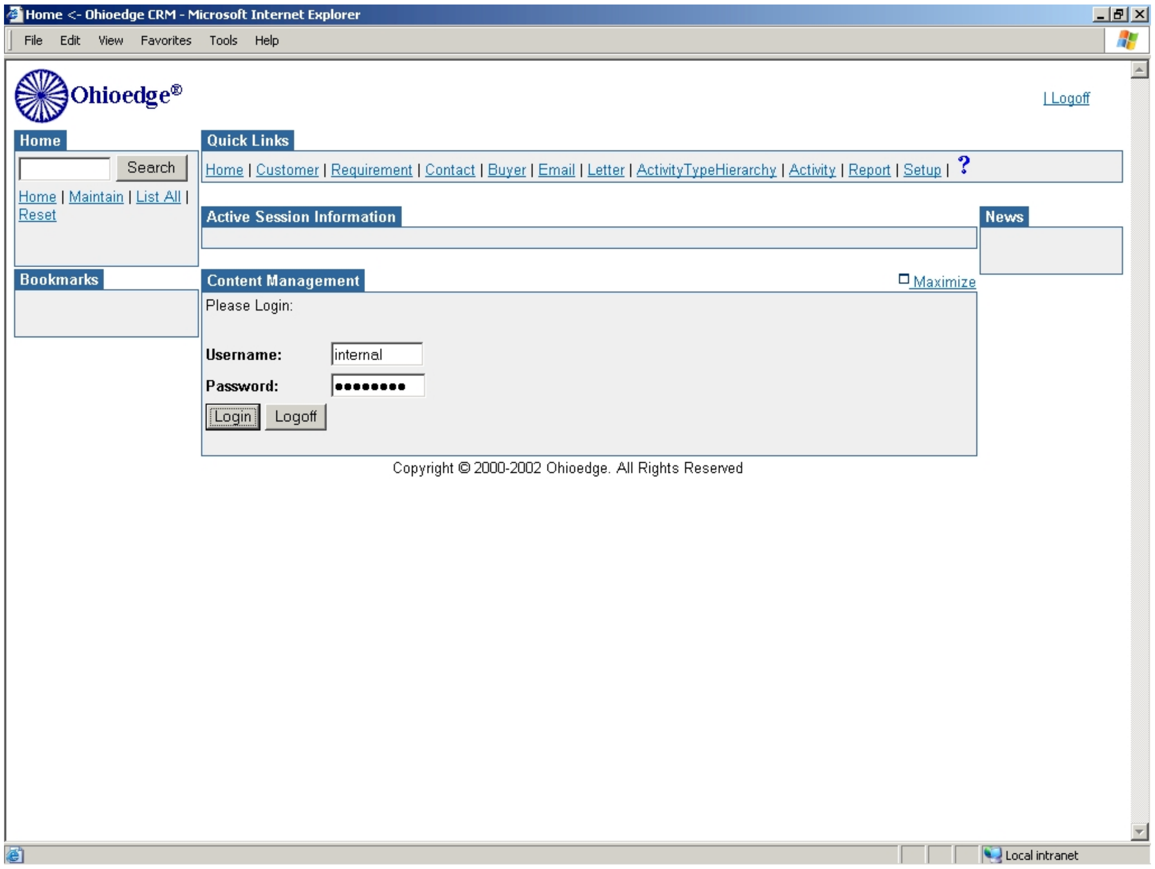
1. Generating message...Complete.

2. Creating license files...Complete.

public.key, private.key and msg.signature files are created in C:\ohioedge\crm\license directory."

STEP 2. Update the license record using J2eeBuilder License component (part of J2eeBuilder library and comes with every J2eeBuilder application, such as J2eeOrganization, J2eeCustomer, and Ohioedge CRM)

1. Login as shown in Figure below.


Home <- Ohioedge CRM - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Ohioedge®
| Logoff
Home
Search
Home | Maintain | List All | Reset
Quick Links
Home | Customer | Requirement | Contact | Buyer | Email | Letter | ActivityTypeHierarchy | Activity | Report | Setup | ?
Active Session Information
News
Bookmarks
Content Management
Maximize
Login
Logoff
Local intranet

Please Login:

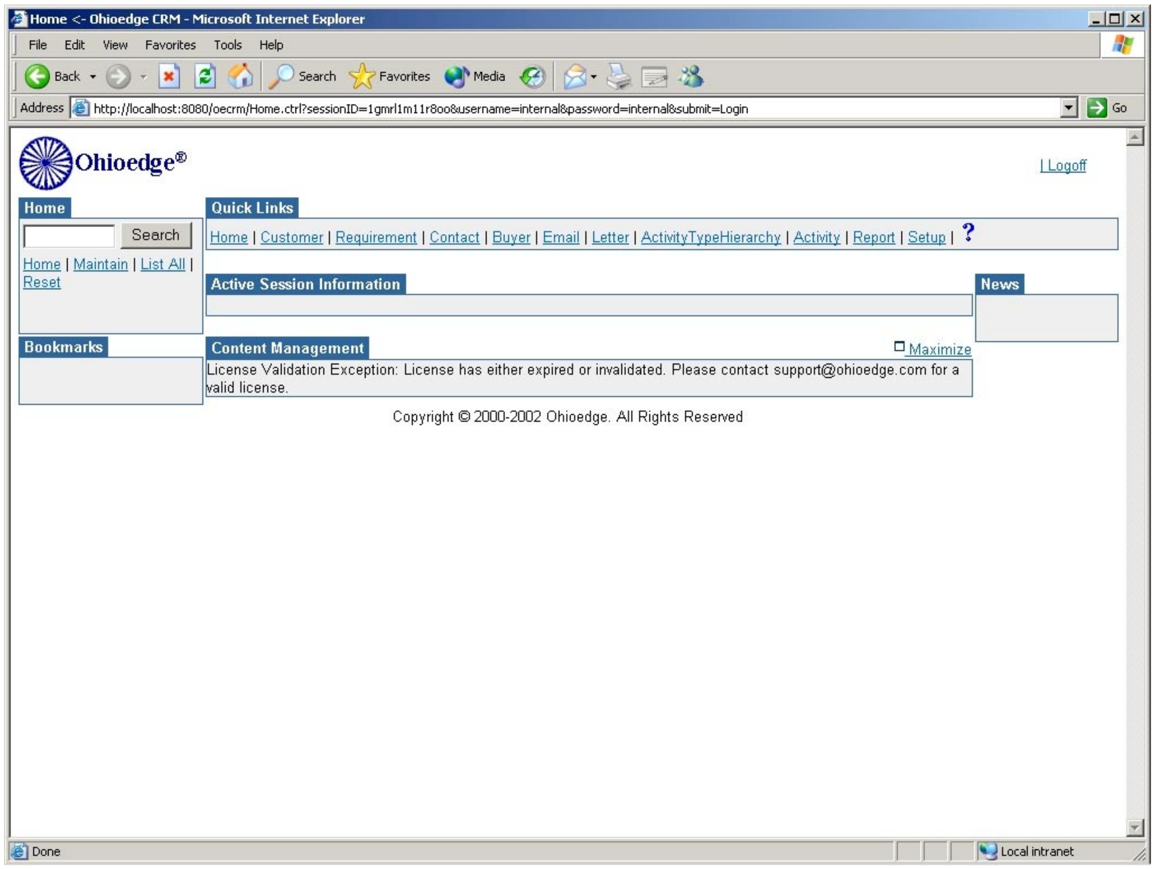
Username: internal

Password: ●●●●●●●●●

Copyright © 2000-2002 Ohioedge. All Rights Reserved

Figure 5. J2eeBuilder application login.

2. As shown in Figure 5, you should see either, a License Validation Exception message if your license record data does not match with the license keys or a License Expiration Exception message if the current date is later than the expiration date of the license record.

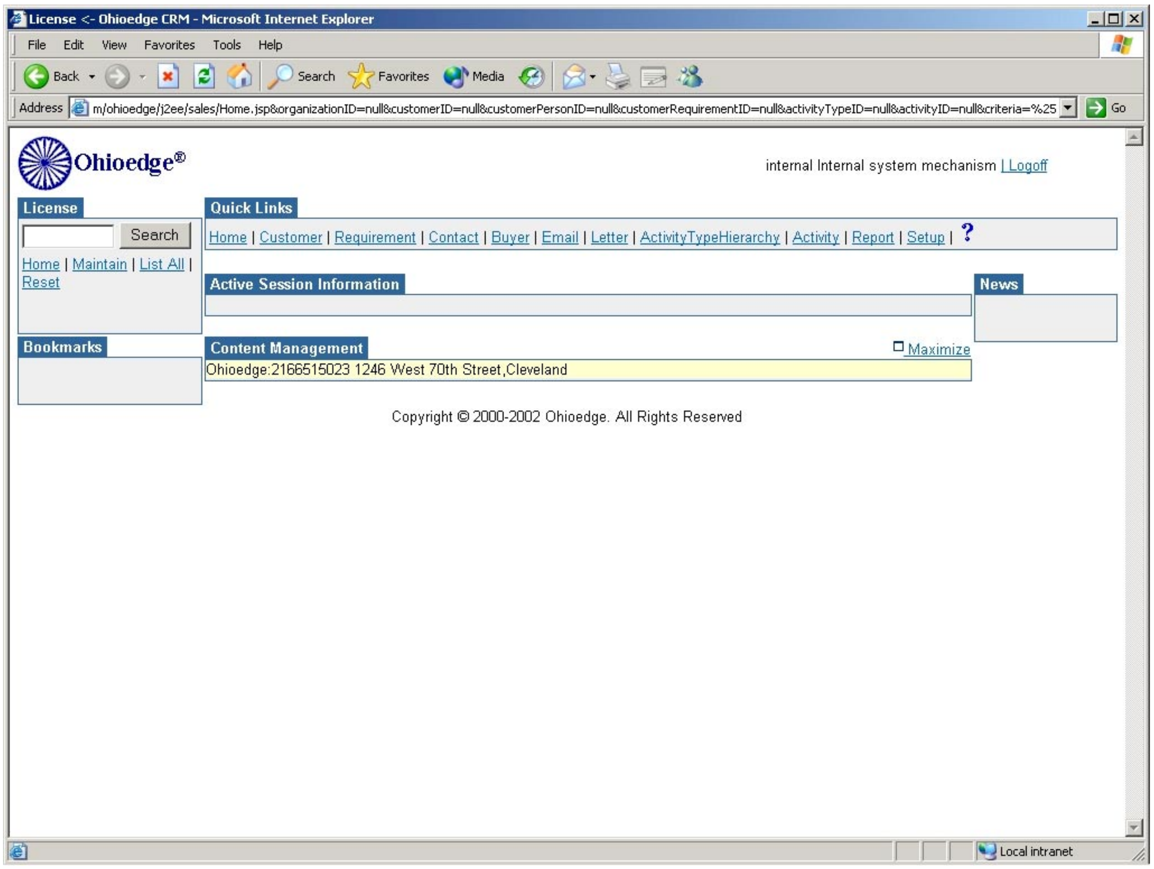

Home <- Ohioedge CRM - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/oecrm/Home.ctrl?sessionID=1gmrl1m11r8oo&username=internal&password=internal&submit=Login
Go
Ohioedge®
| Logoff
Home
Search
Home | Maintain | List All | Reset
Quick Links
Home | Customer | Requirement | Contact | Buyer | Email | Letter | ActivityTypeHierarchy | Activity | Report | Setup | ?
Active Session Information
News
Bookmarks
Content Management
Maximize
Done
Local intranet

License Validation Exception: License has either expired or invalidated. Please contact support@ohioedge.com for a valid license.

Copyright © 2000-2002 Ohioedge. All Rights Reserved

Figure 6. License Exception

3. Click on the ListAll link in the top-left portlet (Home). As shown in Figure below, you should see the license record in the content management portlet.

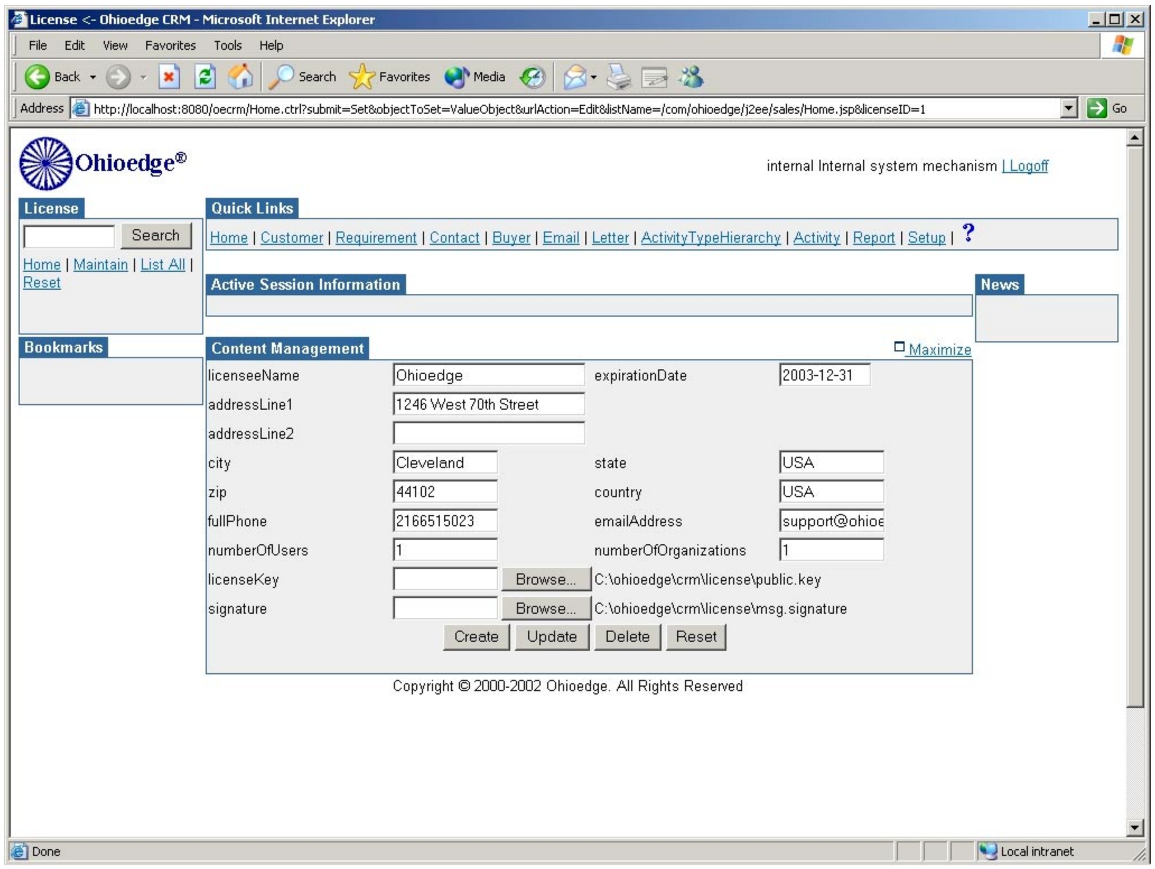

License <- Ohioedge CRM - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address m/ohioedge/j2ee/sales/Home.jsp&organizationID=null&customerID=null&customerPersonID=null&customerRequirementID=null&activityTypeID=null&activityID=null&criteria=%25
Go
Ohioedge®
internal Internal system mechanism | Logoff
License
Search
Home | Maintain | List All | Reset
Quick Links
Home | Customer | Requirement | Contact | Buyer | Email | Letter | ActivityTypeHierarchy | Activity | Report | Setup | ?
Active Session Information
News
Bookmarks
Content Management
Maximize
Local intranet

Ohioedge:2166515023 1246 West 70th Street,Cleveland

Copyright © 2000-2002 Ohioedge. All Rights Reserved

Figure 7

4. Click on the record displayed in the content management portlet, it should open-up the record for maintenance as shown in Figure below.


License <- Ohioedge CRM - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go
Ohioedge®
License
Search
Home | Maintain | List All | Reset
Bookmarks
Quick Links
Home | Customer | Requirement | Contact | Buyer | Email | Letter | ActivityTypeHierarchy | Activity | Report | Setup | ?
Active Session Information
News
Content Management
Maximize
Browse...
Create Update Delete Reset
Done
Local intranet

http://localhost:8080/oecrm/Home.ctrl?submit=Set&objectToSet=ValueObject&urlAction=Edit&listName=/com/ohioedge/j2ee/sales/Home.jsp&licenseID=1

internal Internal system mechanism | Logoff

licenseeName	Ohioedge	expirationDate	2003-12-31
addressLine1	1246 West 70th Street		
addressLine2			
city	Cleveland	state	USA
zip	44102	country	USA
fullPhone	2166515023	emailAddress	support@ohioe
numberOfUsers	1	numberOfOrganizations	1
licenseKey		Browse...	C:\ohioedge\crm\license\public.key
signature		Browse...	C:\ohioedge\crm\license\msg.signature

Copyright © 2000-2002 Ohioedge. All Rights Reserved

Figure 8. License record maintenance.

5. Update this record with the EXACT values you used in Step 1 above while creating the license files.

- LicenseeName -> licenseeName
- expirationDate -> expirationDate
- numberOfOrganizations -> numberOfOrganizations
- numberOfUsers -> numberOfUsers
- licenseKey -> Full name (including path) of public.key
- signature -> Full name (including path) msg.signature

Click on ‘Update’. It should come up with a message – Transaction was successfully completed.

6. Logoff by clicking on the logoff link in the top portlet.
7. Close the browser.
8. Open the browser and login. You should see a message – Transaction was

successfully completed.

You have successfully created and setup the license for your J2eeBuilder application.