Introducing EJB-CMP/CMR, Part 1 of 2

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Introduction 2
2. Application design 7
3. Example 1: Creating the entity bean 12
4. Example 1: Defining CMP fields 13
5. Example 1: Finder and Create methods 17
6. Example 1: Creating the session bean 20
7. Example 1: Creating the client application 29
8. Example 1: Compiling and packaging your application 31
9. Example 2: Finder methods and EJB-QL 43
10. Summary 47
11. Feedback 48

Section 1. Introduction

Introduction to CMP/CMR

This tutorial is designed to introduce you to Container-Managed Persistence (CMP) and Container-Managed Relationships (CMR) in Enterprise JavaBeans 2.0 (EJB). These features are particular to EJB entity beans that are typically long-lived as compared to session beans that are normally transitory.

Enterprise JavaBeans (EJB) 2.0 extends the earlier version 1.1 by adding advanced support for entity beans as follows:

- updated container-managed persistence for entity beans
- support for container-managed relationships
- EJB-Query Language (EJB-QL) for portable *select* and *find* query methods defined in the deployment descriptor
- The addition of local interfaces and local home interfaces to optimize access from other beans in the same container.

If you want to buy or sell components, you will most likely want a layer of persistence in your components to work cross-platform on application servers (for example, IBM WebSphere, BEA WebLogic, JBoss/Tomcat, etc.) and persistence storage systems (for example, Oracle, DB2, etc.). You do not have to write low-level Java Database Connectivity (JDBC) calls in your EJBs to add these features, a great reducer of time and complexity. Once you get the hang of CMP/CMR, it is faster to write entity beans using this technology, than using low-level JDBC inside of bean-managed persistence (BMP) beans.

Should I care about CMP/CMR?

What does this mean to you? Well, for starters, you do not have to write low-level JDBC calls and, you do not have to write code to manage relationships. It is all built into the EJB framework. Your interface to relationships is through the pervasive `java.util.Collection` and `java.util.Set` which most EJB developers are already familiar with. Very cool!

This additional feature includes support for JavaBeans component patterns for persistent fields, inside of the entity bean. Thus, instead of making your class variables public -- which has always felt strange to me -- you create get and set methods following the JavaBean's standard naming pattern we all know and love.

I can't stress this point enough. Since EJB 2.0 containers will support the most common SQL databases (and other data stores as well), you can write components that work with many types of databases. This makes it easier to sell components that require persistent storage. For example, you can sell components that will work in an IT department that uses Oracle or a shop that uses DB2. Thus, instead of writing low-level JDBC calls using SQL native to a particular database, you will use EJB-QL to create finder and select methods, and describe relationships in deployment descriptors.

Simply put, CMP/CMR is the missing link in cross-platform component creation. CMP/CMR will spur the growth of the enterprise level component marketplace. In addition, CMP/CMR is easier to use then low-level JDBC calls. CMP/CMR corrects many of the foibles, and missing functionality of earlier versions of CMP. There are many persistence frameworks, none are available on as many application server platforms as EJB CMP/CMR!

What do I need to know for this tutorial?

The example code in this tutorial is written to work with any J2EE compliant application server that supports EJB 2.0. The example code endeavors to be compliant; thus, all example code was deployed on the J2EE reference implementation that ships with Java 2 SDK, Enterprise Edition 1.3. The example code should deploy to your application server of choice by just modifying the Ant build scripts and corresponding deployment descriptors as long as your application server support EJB 2.0, and therefore, supports CMP/CMR.

This tutorial assumes that you are familiar with Java technology and to some extent EJB; although, you do not have to be an expert. Since I will be covering deployment descriptors, which are written in XML, you should have a rudimentary knowledge of XML. If you are not familiar with EJB, I recommended that you read *Enterprise JavaBeans Fundamentals*, an developerWorks tutorial written by Richard Monson-Haefel and Tim Rohaly (See Resources on page 47). This is an excellent tutorial written by great authors. Even if you do not read this tutorial word for word, I suggest you at least use it as a reference.

Although not a prerequisite per se, knowledge of `Ant`, an XML-based, open source build system similar to make, will be helpful to understand the build scripts presented in the examples.

You do not need knowledge of JDBC since there will be no low-level calls in this tutorial, but basic knowledge of SQL and relational database theory is required.

What will this tutorial series cover?

Instead of writing a giant tutorial that would take days to go through. I have split the tutorial into three parts that can each be finished in an hour or so. You could finish each tutorial during a lunch break, so get a sandwich and a beverage, and get started.

EJB 2.0 added a lot of features and functionality, this tutorial focuses on CMP/CMR. Thus, this tutorial assumes you have a background with EJB, and entity beans. You don't have to be an EJB expert to follow along. The tutorials cover local interfaces, deployment descriptor CMP, CMR fields and relationship elements. I will also cover the full range of relationship types as follows:

- One-to-one
- One-to-many
- Many-to-many

The relationships in the example also cover both unidirectional support and bidirectional support. The relationships are defined in XML deployment descriptors.

In the first tutorial, you get a taste of CMP/CMR and EJB-QL, then I will delve into an example of a simple EJB 2.0 style CMP entity bean. Part of the example demonstrates simple EJB-QL to create a finder method without a Java implementation. This tutorial is just to get you acclimated to the terminology and technology. and it adds a teaser example. The next part, the second tutorial, does the heavy lifting.

In the second tutorial, I will build on the first example to eventually cover each type of relationship and each type of direction (both unidirectional and bidirectional form). Each example also has a client that demonstrate accessing the relationships to add, remove and change related members. Lastly, cascade-delete is demonstrated with consequences for both the one-to-one relationship example and the one-to-many relationship.

In the third tutorial, I will demonstrate advance EJB-QL to build finder and select methods. This tutorial uses the relatioships I built in the last tutorial to show the ends and outs of EJB-QL.

Each example includes source code as follows: Deployment Descriptor, Implementation class files, Interface class files, Home Interface class files and an Ant build script. For these examples I will use the J2EE reference implementation which ships with Cloudscape RDBMS system.

In a future follow-on article to these tutorials I will port the final example to IBM Websphere plus DB2 and JBOSS plus HypersonicSQL to show how the CMP 2.0 entity beans can be deployed to many environments: from light-weight single user databases to industrial strength transaction servers utilizing full scalable behemoth databases! This article will disucss some of the pitfalls of CMP 2.0 implementation variants.

Now that I have stated what the tutorial will cover, let's briefly go over what I'm not going to cover. I will not cover (in any detail) support for transactions, as this is not new

to EJB 2.0. Also, since this article is focused on CMP-CMR, there is no need to cover Message-driven beans or session beans. (Session beans and transactions will be presented but not explained in any detail as they are covered in detail in many other sources see the Resources on page 47 section for more detail.)

Remember you don't have to be an EJB expert to get value out of this tutorial, but some details will be left for explanation by other resources. Also, I do not cover EJB-QL in great detail, although, it is covered somewhat as it is needed for the examples. Expect a follow-up article or tutorial covering EJB-QL in depth.

Tools you will need for this tutorial

This tutorial is chock-full of example EJBs not to mention deployment descriptors and sample clients. Thus, you will need the Java SDK 1.3 or higher and a J2EE compliant application server, the reference implementation will do. Here is a list of requisite tools:

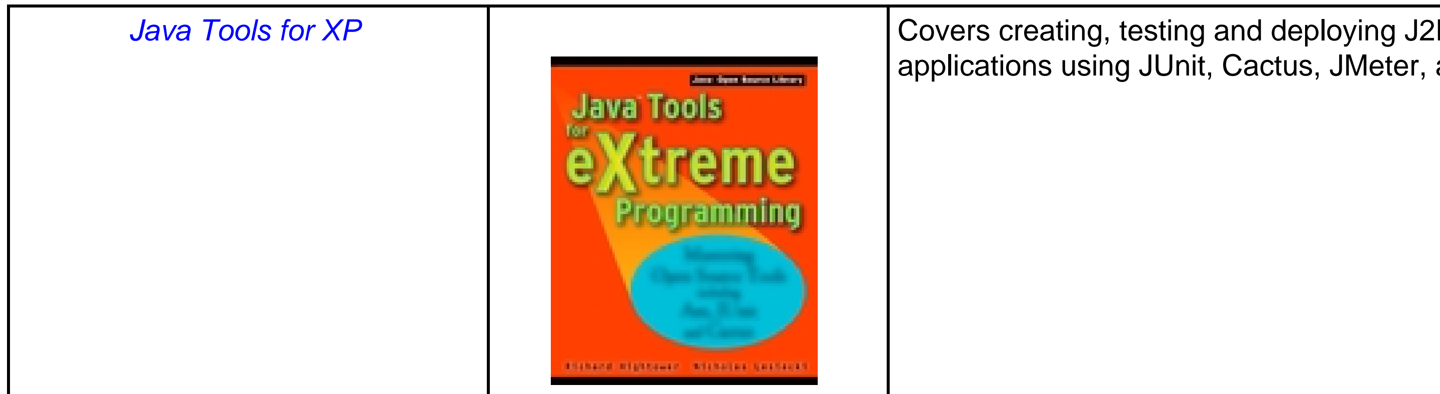
- A text editor. An IDE is okay too.
- A Java development environment, for example, the Java SDK 1.3 or higher.
- A persistence datastore, that is, likely a SQL compliant database.
- A JDBC driver for your persistence datastore of choice.
- An J2EE application server that is EJB 2.0 compliant.
- The Ant build system. Used to build and package the examples.

As stated above, all of the examples, were done in the J2EE reference implementation, which is freely available as part of the J2EE SDK 1.3.

About the Author

Rick Hightower, Director of Development at *eBlox*, has over a decade of experience as a software developer. He leads the adoption of new processes like Extreme Programming, and technology adoption like adoption of CMP and CMR.

Rick's publications include *Java Tools for eXtreme Programming* , which covers deploying and testing J2EE projects (published by *John Wiley*); contributions to *Java Distributed Objects* (published by Sams); and several articles in *Java Developer's Journal.*



Java Tools for XP		Covers creating, testing and deploying J2 applications using JUnit, Cactus, JMeter, a

Also expect to see his book on Jython from Addison Wesley in the near future.

Rick has also taught classes on developing Enterprise JavaBeans, JDBC, CORBA, Applets as CORBA clients, etc.

Rick is a software developer at heart who specializes in software development tools, creating frameworks and tools, enforcing processes, and developing enterprise applications using J2EE, XML, UML, CORBA, JDBC, SQL, and Oracle technologies.

A self-proclaimed technology masochist, Rick often enjoys working with cutting-edge technologies. "Expect to bleed a lot if you are on the cutting edge", is one of Rick's favorite sayings. On a recent project, Rick and others adopted EJB CMP/CMR as there persistence layer of choice before the EJB 2.0 specification was finished.

Many a night, they doubted their decision, but it all worked out in the end. Rick states: "We did not want to use a non-J2EE compliant persistence solution, since we want the freedom of porting our application to other J2EE application servers, and the possibility of selling components that we created. In the end, it looks like going with EJB CMP/CMR was a good decision, but there was some pain along the way as we were using the specification a lot as our main form of documentation, and we were one of the first users of this implementation of CMP/CMR."

"We were able to help the vendor by reporting problems. Some of the problems were our perception of the specification versus the vendors perception of the specification. The vendor was usually right (that said we were credited with reporting and at times fixing quite a few bugs)."

"One of the advantages of this decision was our team got to work with a solid implementation of CMP/CMR a long time before it was available anywhere else. We now use EJB CMP-CMR for all projects. I am a big EJB CMP-CMR fan! This new feature will drive the adoption of EJB even further."

Section 2. Application design

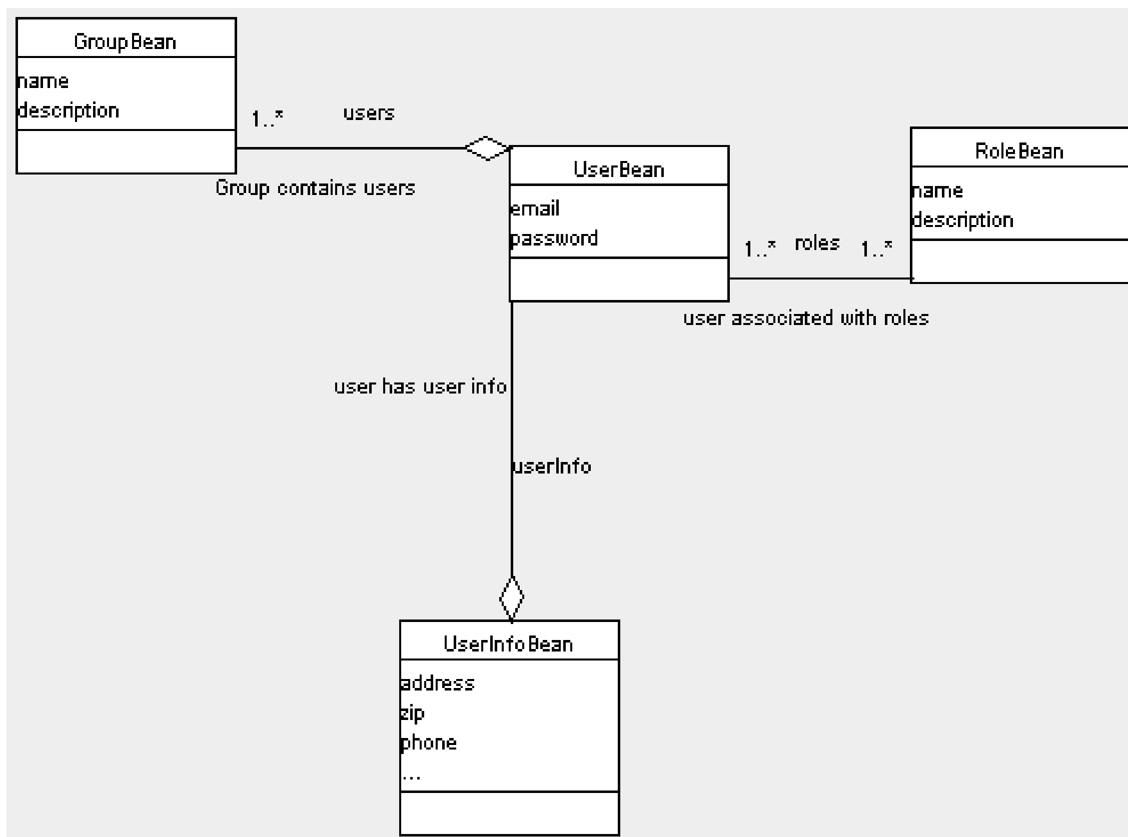
Application design: Entity design

This tutorial uses an example based on a fictitious authentication subsystem for an online content management system with the following capabilities:

- log users into the system
- authenticate users are in certain roles
- allow user to be organized into groups to allow group operations
- store user information like address and contact information
- manage roles, users, groups

I picked this domain because most developers are familiar with it to some degree I want to spend the content of this tutorial covering the technology rather the domain of the example.

Figure 1: User, Group, Role, and UserInfo entities in the example



An overview of the entities in the system in *Figure 1* shows that there are four distinct entities: *User*, *Group*, *Role*, and *UserInfo*. Each of these entities have the following three relationships:

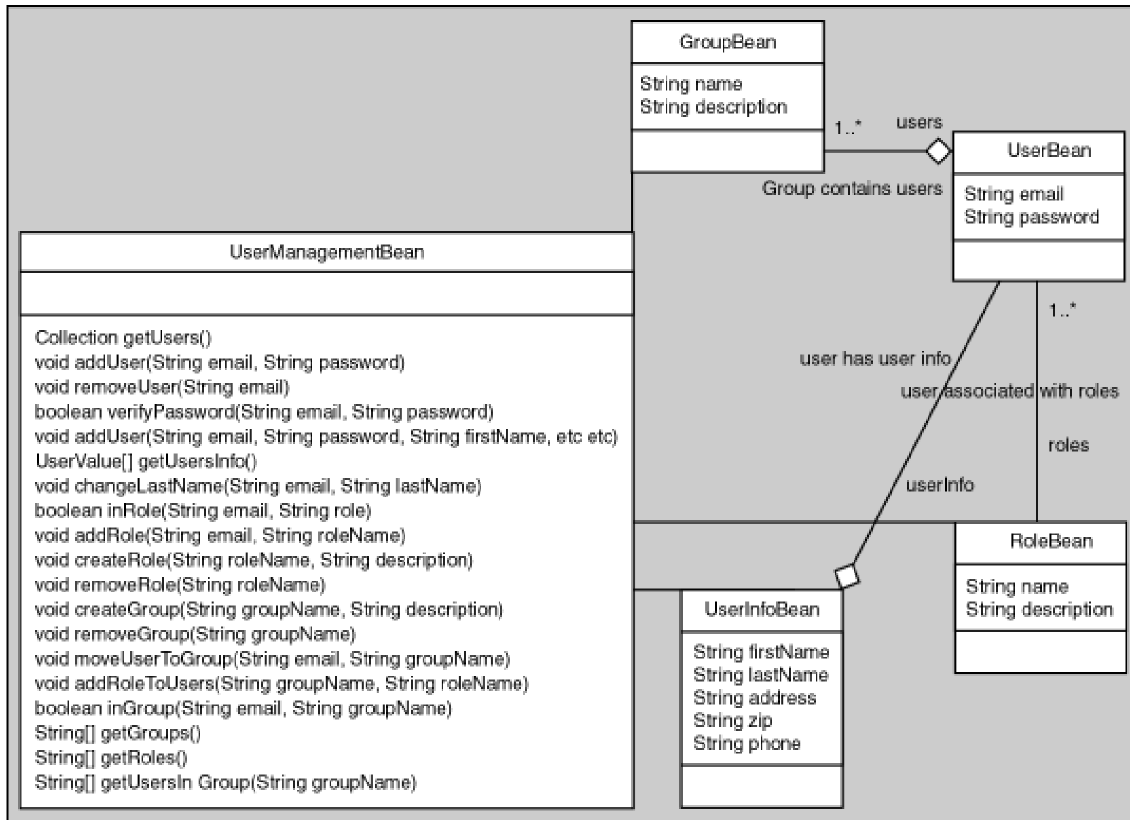
- Users are associated with Roles (many-to-many)
- A User has UserInfo (one-to-one)
- A Group contains Users (one-to-many)

Each of these entities would likely have its own table in a relational database as well as a table for any of the many-to-many relationships. Thus, this example application consists of five tables, as modelled on using a relational database.

Application design: Client access system via Session bean

All of the entity beans in this example are local entity beans. Local beans can not be access by remote clients. They can not be returned from a method either. Thus all access to the entities in this example is done through a session bean that acts as a mediator between the client and the entity beans in this example (see).

Figure 2: Diagram of operations and associations



As you can see from the above diagram, the `UserManagement` session bean has access to all other beans. It functions as a facade into the authentication system.

Figure 3: Block Diagram of `UserManagement` being accessed by client application

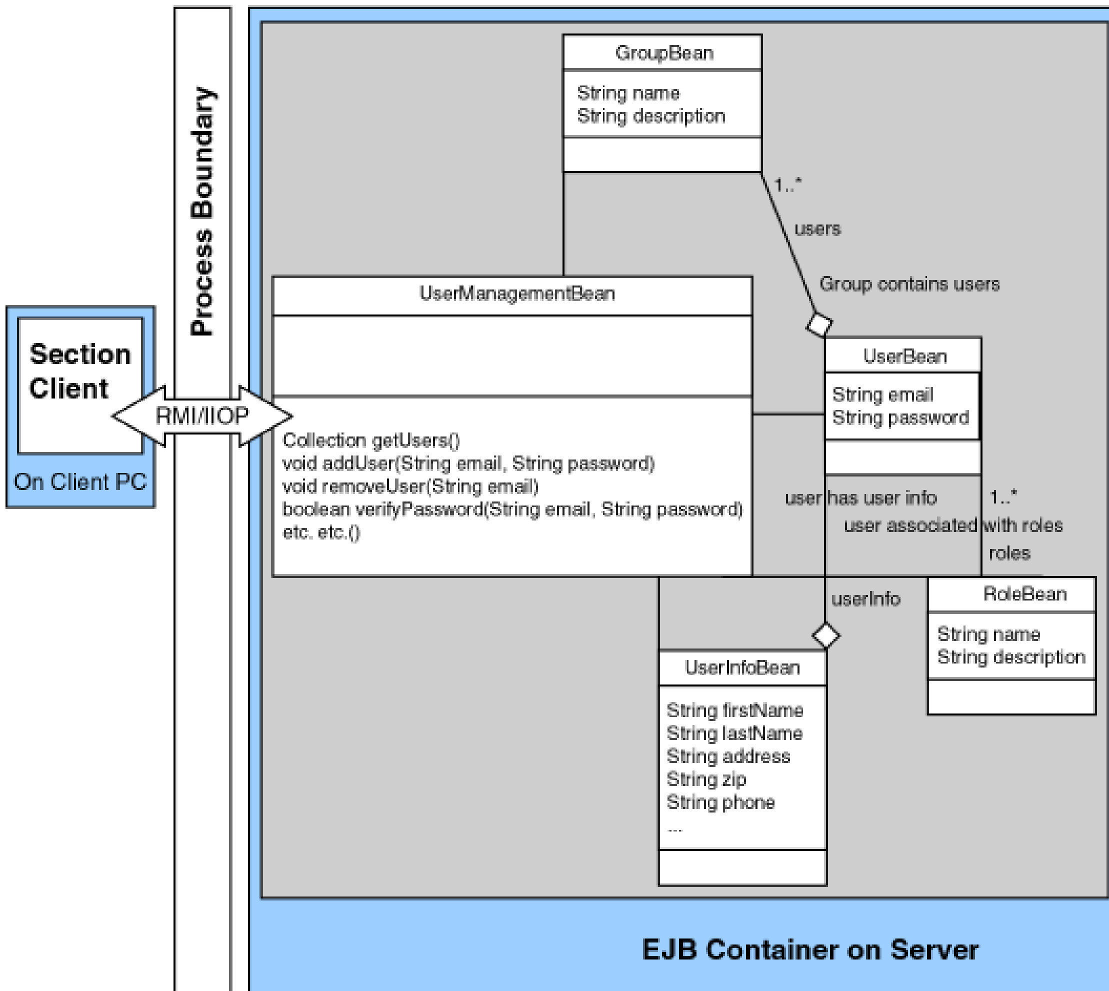


Figure 3 shows that the client only access the `UserManagementBean`. It does not directly access any of the other beans in this example. Also, notice that the client is in another address space; actually it can be on another computer half-way across the world. The client accesses the session bean, `UserManagementBean`, remotely.

Overview of Examples

There is one thing I hate when reading a tutorial:examples in the tutorial that are too complex. I can't stand getting bogged down in details that aren't necessary to understand the technology. I prefer examples that are just complex enough to show the concept under review.

I also like to look at the source code and compile and run it as soon as possible. The

example is not real until you can get it running, and you can look over the code. With this in mind, I divided the tutorial examples into a incrementally increasing complex example, each with automated build files. Thus when you first go over CMP, you do not have to worry about EJB-QL, etc. Since the tutorials get incrementally more complex, I suggest that you do them in the right order.

The examples are as follows:

- The first example introduces you to the concepts of CMP including defining CMP fields in the deployment descriptor. This example also shows how to reference an enterprise bean from another bean, and how to reference an enterprise bean from a client. This example only has two beans: the `UserBean` and the `UserManagementBean`. The example is in this first tutorial, part one
- The second example adds a `findAll()` method to the home interface using EJB-QL. This is just a slight change from the last example to demonstrate EJB-QL. It has the same number of beans. This example is also in the current tutorial, part one.
- The third example covers adding the `UserInfoBean` and creating a one-to-one bidirectional relationship between the `UserBean` and the `UserInfoBean`. This example is in the second tutorial, part two.
- In the fourth example, I cover adding the `RoleBean`, and creating a many-to-many, unidirectional, relationship between `UserBean`s and `RoleBeans`. This example is in the second tutorial, part two.
- In the fifth example, I cover adding the `GroupBean`, and creating a one-to-many, bidirectional relationship, between `GroupBean` and UserBeans. This example is in the second tutorial, part two.
- The last example extends the fifth and adds EJB-QL to demonstrate naviagating collections and sets of objects. This example is in the third tutorial, part three.

The entire tutorial is code-centric and every part of the code is covered: client, entity beans and session bean.

Section 3. Example 1: Creating the entity bean

CMP Fundamentals: An overview by example

The first example introduces you to the fundamental concepts of CMP including the following:

- defining CMP fields in the deployment descriptor
- how to reference an enterprise bean from another bean,
- how to reference an enterprise bean from a client.

In this example, I will create three things:

- An entity bean (`UserBean`) that uses container-managed persistence.
- A session bean that accesses the entity bean (`UserManagementBean`).
- A client that access the entity bean (Section2Client).

`UserBean` (entity bean)

The `UserBean` represents a user in the system. The `UserBean`, like all enterprise beans, needs three Java source files as follows: a home, an interface, and an implementation. Thus, you need to define the following code:

- Local Interface (`LocalUser`)
- Local Home Interface (`LocalUserHome`)
- Entity Bean Class (`UserBean`)

The source code for this example can be found at *cmpCmr/section2/src*. To compile this example go to *cmpCmr/section2* and type `ant` at the command prompt. A sample *.ear* will be created in the *cmpCmr/section2/final* directory. If you are using the reference implementation, you can deploy this *.ear* with the `deploytool` program.

Section 4. Example 1: Defining CMP fields

Defining CMP fields in the local interface

The first step is to define CMP fields. A CMP field is a field you want the EJB container to persist. An entity bean has CMP fields that are virtual fields. Unlike CMP 1.0, in CMP 2.0 you don't define member variables for CMP fields. Instead, using the JavaBeans property naming convention, you define virtual getter and setter methods corresponding to name of the fields that you want to persist. You add these abstract methods in both the interface and the implementation. This concept is demonstrated in the following code:

```
package com.rickhightower.auth;

import javax.ejb.EJBLocalObject;

public interface LocalUser extends EJBLocalObject {

    public String getEmail();
    public String getPassword();

}
```

Since you only defined getters for the `UserBean`'s e-mail and password CMP fields, they are both considered read-only fields. Conversely, the implementation entity bean must defined both getters and setter methods for the properties, even though the properties are read-only. This is covered in the next panel.

Defining CMP fields in the entity bean class (implementation)

Unlike the local interface the implementation entity bean class must defined both getters and setter methods for the CMP fields, even though the CMP-fields are read-only. The getter and setter methods for the CMP fields are abstract methods since there implementation will be defined by the EJB container. The code for the implementation class is as follows:

```
package com.rickhightower.auth;

public abstract class UserBean implements EntityBean {

    public abstract String getEmail();
```

```
    public abstract void setEmail(String email);

    public abstract String getPassword();
    public abstract void setPassword(String password);


}
```

Notice that the entity bean is declared abstract, and that it defines abstract getters and setter methods for its CMP fields. (Note that the ellipsis is used to abbreviate the listing to the parts of interest.)

It is likely that you will have to map the entity to a database table and CMP fields to columns that table. The next panel shows a sample table that could be used in conjunction with this entity.

Sample table for `UserBean`

The CMP fields in the `UserBean` may correspond to field in a relational table as defined by the following SQL DDL:

```
CREATE TABLE TBL_USER (
    email varchar (50)  PRIMARY KEY,
    password varchar (50) NOT NULL
)
```

Note that since the mapping from entities to tables is left up to each individual EJB container, I will not cover the specifics of the mapping in this tutorial. (This will be covered in a follow up article that ports these examples to WebLogic and JBoss.) Instead I use the default mapping provided by the J2EE reference implementation.

Defining CMP fields in the deployment descriptor

Each one of the CMP fields has to be defined in the deployment descriptor as follows:

```
<cmp-field>
  <field-name>password</field-name>
</cmp-field>
```

```
<cmp-field>
  <field-name>email</field-name>
</cmp-field>
```

Although the CMP field definition in the deployment descriptor is specified by the EJB specification, there is not a standard way to map entity beans to SQL tables and CMP fields to SQL columns. (I hope this is addressed in the next version of the EJB Specification.)

The mapping is left up to the discretion of the EJB container implementation. This will likely change in a future release of EJB. Since the implementation of the reference J2EE application server does not allow easy mapping from entity beans to SQL tables, the examples use the mappings defined by using the `deploytool`.

You will not have to perform mappings. The mappings are provided by the included Ant build scripts.

Defining the primary key field in the deployment descriptor

Every entity bean must have a primary key. The `UserBean`'s primary key is the e-mail CMP field. Thus, primary key field and its type must be specified in the deployment descriptor just like any entity bean as follows:

```
<prim-key-class>java.lang.String</prim-key-class>

<primkey-field>email</primkey-field>
```

Deployment descriptor

The rest of the deployment descriptor is what you would expect. Notice that the deployment descriptor uses `<cmp-version>2.x</cmp-version>` to denote the version of CMP desired. This was added to accommodate EJB containers that support both CMP 1.0 and CMP 2.0 style container managed persistence. The complete deployment descriptor is as follows:

Listing 1: Deployment descriptor for UserBean

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN' 'ht

<ejb-jar>
  <display-name>user-mgmt-beans</display-name>
  <enterprise-beans>
    <entity>
      <display-name>UserBean</display-name>
      <ejb-name>UserBean</ejb-name>

      <local-home>com.rickhightower.auth.LocalUserHome</local-home>
      <local>com.rickhightower.auth.LocalUser</local>
      <ejb-class>com.rickhightower.auth.UserBean</ejb-class>

      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>

      <reentrant>True</reentrant>
      <cmp-version>2.x</cmp-version>

      <abstract-schema-name>UserBean</abstract-schema-name>

      <cmp-field>
        <description>no description</description>
        <field-name>password</field-name>
      </cmp-field>
      <cmp-field>
        <description>no description</description>
        <field-name>email</field-name>
      </cmp-field>

      <primkey-field>email</primkey-field>


    </entity>

  </enterprise-beans>

</ejb-jar>
```

Section 5. Example 1: Finder and Create methods

The Create method

Thus, far I havecovered defining the CMP fields. Another key feature CMP 2.0 defines your `findByPrimaryKey()` method. Notice that `ejbFindByPrimaryKey()` is absent from the entity bean class as it is defined by the EJB container. The finder method, `findByPrimaryKey()` must be declared in the home interface (listed below).

Lastly, lets cover the ability to create UserBeans. The `UserBean` entity class defines two create methods. `ejbCreate()` and `ejbPostCreate()`. Since the e-mail and password CMP fields are read only (you don't have setter methods for e-mail and password CMP fields in the local interface), it is prudent to allow them to be initialized using the `ejbCreate()` method as shown below.

```
package com.rickhightower.auth;

public abstract class UserBean implements EntityBean {

   public String ejbCreate(String email, String password)
                                                  throws CreateException {
         setEmail(email);
         setPassword(password);
         return null;
    }

   public void ejbPostCreate(String email, String password) { }


}
```

The Create method

One thing that you will notice is that the implementation of these methods is virtually empty. Also notice, that the setEmail, setPassword methods that the `ejbCreate()` method calls are abstract; thus, they are defined by the EJB container during deployment.

Remember that the `ejbCreate()` method gets called before the container inserts the row into the database and the `ejbPostCreate()` gets called after the container inserts the row into the database. Since the container, manages the creation of the entity bean, the `ejbCreate()` method returns *null*. The corresponding create method

in the home interface is listed as shown below:

```
package com.rickhightower.auth;

import javax.ejb.EJBLocalHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface LocalUserHome extends EJBLocalHome {

    public LocalUser create (String email, String password)
                                        throws CreateException;

    public LocalUser findByPrimaryKey (String email)
                                        throws FinderException;

}
```

The Create method

Notice that the arguments for the home interfaces create methods match the arguments for the `ejbCreate()` and `ejbPostCreate()` as any veteran EJB developer would expect. The full code listing for the entity bean class is listed as follows:

Listing 2: The code for UserBean

```
package com.rickhightower.auth;


import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.CreateException;


public abstract class UserBean implements EntityBean {

    public String ejbCreate(String email, String password)
                                              throws CreateException {
        setEmail(email);
        setPassword(password);
        return null;
    }

    public void ejbPostCreate(String email, String password) { }

    public abstract String getEmail();
    public abstract void setEmail(String email);
```

```
    public abstract String getPassword();
    public abstract void setPassword(String password);

    public void setEntityContext(EntityContext context){ }
    public void unsetEntityContext(){ }
    public void ejbRemove(){ }
    public void ejbLoad(){ }
    public void ejbStore(){ }
    public void ejbPassivate(){ }
    public void ejbActivate(){ }

}
```

Now you're done with the real work of the first example. That is it. That is all you have to do to write a CMP entity bean; the rest of Example 1 just shows how to access, and package this bean for use on a network.

The CMP fields will be managed by the container. All you had to do is represent the CMP fields with virtual getters and setters methods.

In addition the `findByPrimaryKey()` method is completely defined by the container. You had to define the application-specific create methods, but the implementations of the create methods were virtually empty!

As I stated earlier, CMP entity beans are typically local beans. You can not access local beans directly from the remote client. Thus, next you define a session bean that is used to access this entity bean.

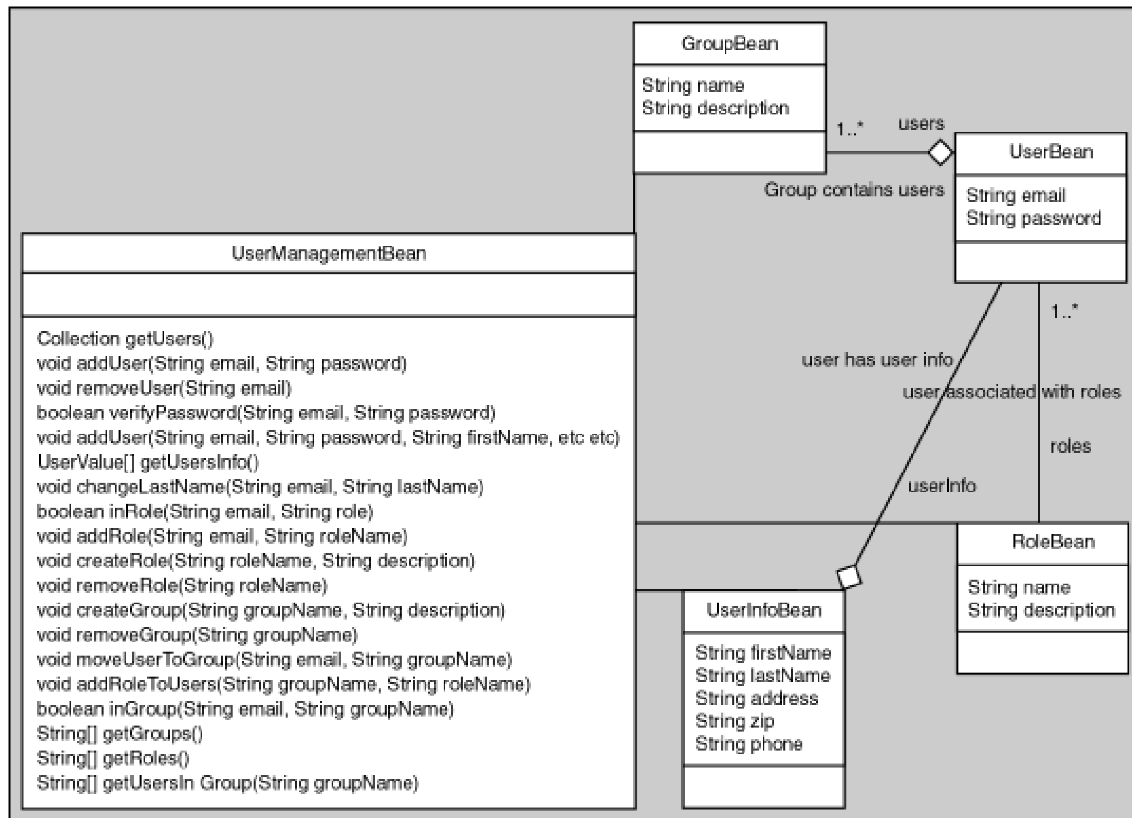
Section 6. Example 1: Creating the session bean

UserManagementBean (session bean)

The client of all of your CMP effort is the UserManagementBean. For all intents and purposes, this is a regular session bean.

The UserManagementBean has operations for creating, removing, and verifying the password of users. It is the only class, thus far, that will directly access the UserBean.

Figure 4: Diagram of UserManagement operations and associations



UserManagementBean referencing the UserBean

In order for UserManagementBean to manager Users, it will have to find the UserBean's home interface.

UserManagement accesses the UserBean entity by looking up UserBean's home interface as follows:

```
package com.rickhightower.auth;


import javax.naming.Context;
import javax.naming.InitialContext;

import javax.naming.NamingException;

public class UserManagementBean implements SessionBean {


   private LocalUserHome getUserHome() throws NamingException {
        Context initial = new InitialContext();
        return (LocalUserHome) initial.lookup("java:comp/env/ejb/LocalUser");
   }
```

Notice that the getUserHome() method is a private helper method that returns a reference to the UserBean's home interface (LocalUserHome). In order for the UserManagementBean to have access to the UserBean's home interface the following had to be added to the UserManagementBean's deployment descriptor within the session element as follows:

```
<session>

      ...
      <ejb-local-ref>
        <ejb-ref-name>ejb/LocalUser</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>com.rickhightower.auth.LocalUserHome</local-home>
        <local>com.rickhightower.auth.LocalUser</local>
        <ejb-link>userEntity.jar#UserBean</ejb-link>
      </ejb-local-ref>
        ...
```

Notice that the ejb-local-ref defines the reference to the UserBean. (The UserBean is contained in an *.jar* file called *userEntity.jar* that I will cover in detail in Compiling and packaging your application on page 31 .)

UserManagementBean grabbing the UserBean's home interface

The UserManagementBean stores the reference to the home interface as a private

member variable. The session bean looks up the entity bean during creation and activation in the `ejbCreate()` and ejbActivate methods as follows:

```
package com.rickhightower.auth;

public class UserManagementBean implements SessionBean {

    private  LocalUserHome userHome = null;

    public void ejbCreate() throws CreateException {

        try {
            userHome = getUserHome();
        } catch (NamingException e) {
            throw new CreateWrapperException(
                "Unable to find local user home in ejbCreate", e);
        }
    }

    public void ejbActivate() {

        try {
            userHome = getUserHome();
        } catch (NamingException e) {
            throw new EJBException(
                "Unable to find local user home in Activate", e);
        }
    }

}
```

Thus the home interface is always available to the business methods of the session bean.

`UserManagementBean` business methods in remote interface

The business methods of the `UserManagementBean` are defined succinctly in the remote interface of the `UserManagementBean` as follows:

```
package com.rickhightower.auth;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface UserManagement extends EJBObject {
```

```
    public void addUser(String email, String password) throws RemoteException;
    public void removeUser(String email) throws RemoteException;
    public boolean verifyPassword(String email, String password)throws RemoteException;
}
```

UserManagementBean business methods implementation

All EJB veterans should realize that the implementation of these methods are in the implementation Session bean class as follows:

```
package com.rickhightower.auth;

public class UserManagementBean implements SessionBean {


    public void addUser(String email, String password) {

        try {
            LocalUser user = userHome.create(email, password);
        } catch (CreateException e) {
            throw new EJBException
                        ("Unable to create the local user " + email, e);
        }

    }

    public void removeUser(String email) {
        try {
            userHome.remove(email);
        } catch (RemoveException e) {
            throw new EJBException("Unable to remove the user " + email, e);
        }
    }

    public boolean verifyPassword(String email, String password){

        try {
            LocalUser user = userHome.findByPrimaryKey(email);
            return user.getPassword().equals(password);
        } catch (FinderException e) {
            throw new EJBException
                            ("Unable to create the local user " + email, e);
        }

    }

}
```

UserManagementBean business method decomposition

Notice that the addUser method takes an e-mail, and a password argument. The method then calls the LocalUserHome's create method as follows:

```
public void addUser(String email, String password) {

        LocalUser user = userHome.create(email, password);
```

The removeUser method uses the e-mail argument passed to it to call the LocalUserHome's remove method to remove the UserBean from the container, that is, delete the row in the database corresponding to the UserBean. Since the e-mail address is the primary key, it can be used to uniquely identify the entity and remove it with the home interface's remove method. Note that you did not define the remove method in the home interface -- it was inherited from javax.ejb.EJBLocalHome interface and it was implemented by the EJB container.

```
public void removeUser(String email) {

        userHome.remove(email);
```

The verifyPassword method uses the LocalUserHome's findByPrimaryKey() method to look up the UserBean instance associated with the e-mail primary key. The verifyPassword method then uses the CMP password field of the entity to compare to the password that was passed as an argument to determine if they match as follows:

```
public boolean verifyPassword(String email, String password){

        LocalUser user = userHome.findByPrimaryKey(email);
        return user.getPassword().equals(password);
```

The business methods of the session bean use the UserBean to manage the instances of the user bean (add, remove), and to verify the password is the same as the one passed. As far as the UserManagementBean session bean is concerned at this point, there is nothing special about the UserBean entity bean.

`UserManagementBean` complete listing

There is nothing really special about the `UserManagement` session bean. It references the `UserBean` and invokes its home method to create and remove users. It also uses the `LocalUser` interface to verify passwords.

The full source and deployment descriptor, interface and class files are listed below.

Listing 3: Deployment Descriptor Listing for `UserManagementBean` (ejb-jar.xml)

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN' 'ht

<ejb-jar>
  <display-name>main-user-mgmt</display-name>
  <enterprise-beans>
    <session>

      <display-name>UserManagementBean</display-name>
      <ejb-name>UserManagementBean</ejb-name>

      <home>com.rickhightower.auth.UserManagementHome</home>
      <remote>com.rickhightower.auth.UserManagement</remote>
      <ejb-class>com.rickhightower.auth.UserManagementBean</ejb-class>

      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>

      <ejb-local-ref>
        <ejb-ref-name>ejb/LocalUser</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>com.rickhightower.auth.LocalUserHome</local-home>
        <local>com.rickhightower.auth.LocalUser</local>
        <ejb-link>userEntity.jar#UserBean</ejb-link>
      </ejb-local-ref>

  </enterprise-beans>

</ejb-jar>
```

Note that the deployment descriptor for all examples is shown minus security elements, and assembly elements.

Listing 4: The Complete Remote Interface for `UserManagementBean`

```
package com.rickhightower.auth;

import javax.ejb.EJBObject;
```

```
import java.rmi.RemoteException;

public interface UserManagement extends EJBObject {

    public void addUser(String email, String password) throws RemoteException;
    public void removeUser(String email) throws RemoteException;
    public boolean verifyPassword(String email, String password)throws RemoteException;
}
```

`UserManagementBean` complete listing (cont'd)

The following shows the implementation code for the home Interface and the Entity bean for UserManagementBean.

Listing 5: The Complete Home Interface for `UserManagementBean`

```
package com.rickhightower.auth;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface UserManagementHome extends EJBHome {

    UserManagement create() throws RemoteException, CreateException;
}
```

Listing 6: The complete entity bean implemetation for `UserManagementBean`.

```
package com.rickhightower.auth;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import javax.ejb.CreateException;
import javax.ejb.RemoveException;
import javax.ejb.FinderException;
import javax.ejb.EJBException;

import javax.naming.Context;
import javax.naming.InitialContext;

import javax.naming.NamingException;

import com.rickhightower.util.CreateWrapperException;
```

```
public class UserManagementBean implements SessionBean {

    private  LocalUserHome userHome = null;



    public void addUser(String email, String password) {

        try {
            LocalUser user = userHome.create(email, password);
        } catch (CreateException e) {
            throw new EJBException
                        ("Unable to create the local user " + email, e);
        }

    }

    public void removeUser(String email) {
        try {
            userHome.remove(email);
        } catch (RemoveException e) {
            throw new EJBException("Unable to remove the user " + email, e);
        }
    }

    public boolean verifyPassword(String email, String password){

        try {
            LocalUser user = userHome.findByPrimaryKey(email);
            return user.getPassword().equals(password);
        } catch (FinderException e) {
            throw new EJBException
                            ("Unable to create the local user " + email, e);
        }

    }

    public void ejbCreate() throws CreateException {

        try {
            userHome = getUserHome();
        } catch (NamingException e) {
            throw new CreateWrapperException(
                "Unable to find local user home in ejbCreate", e);
        }
    }

    public void ejbActivate() {

        try {
            userHome = getUserHome();
        } catch (NamingException e) {
            throw new EJBException(
                "Unable to find local user home in Activate", e);
        }
    }
```

```
    public void ejbPassivate() {

        userHome = null;
    }

    public void ejbRemove() {}
    public void setSessionContext(SessionContext sc) {}

    private LocalUserHome getUserHome() throws NamingException {
        Context initial = new InitialContext();
        return (LocalUserHome) initial.lookup("java:comp/env/ejb/LocalUser");
    }

}
```

Section 7. Example 1: Creating the client application

The `UserManagement` client

In the interest of completeness you have to have a client that can exercise your example. The client in this application does the following:

- Looks up the `UserManagement` session bean using JNDI
- Adds two user to the system using the addUser method of the `UserManagementBean`
- Verifies that one of the user's password is "mypassword" using the `verifyPassword()` method of the `UserManagementBean`
- Then removes the two users using the removeUser method of the `UserManagementBean`

The complete code listing for the client is listed below.

In order for the application client to access the session bean, it needs an ejb-ref element entry in its deployment descriptor (*application-client.xml*) shown in the example column to the left below the home interface listing.

Notice the manner that you access the session bean from the client is similar to the way the `UserManagement` session bean access the User entity.

Listing 7: The `UserManagement` Client code

```
/** Client code */
package com.rickhightower.client;

import com.rickhightower.auth.UserManagement;
import com.rickhightower.auth.UserManagementHome;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class Section2Client {

   public static void main(String[] args) {
       try {
           Context initial = new InitialContext();
           Object ref =
                 initial.lookup("java:comp/env/ejb/UserManagement");
           UserManagementHome home =
            (UserManagementHome)PortableRemoteObject
                              .narrow(ref,UserManagementHome.class);
```

```
            /* Create the users--Rick and Nick */
            UserManagement userMgmt = home.create();
            userMgmt.addUser("rick@rickhightower.com", "mypassword");
            userMgmt.addUser("nick@rickhightower.com", "oxford");

            /* Verify Rick's Password */
            boolean login = false;
            login = userMgmt.verifyPassword("rick@rickhightower.com",
                                                          "mypassword");

            System.out.println("Login =" + login);

            /* Remove the users--Rick and Nick */
            userMgmt.removeUser("rick@rickhightower.com");
            userMgmt.removeUser("nick@rickhightower.com");

        } catch (Exception e) {
            System.err.println("MESSAGE:" + e.getMessage());
            e.printStackTrace(System.err);
        }
    }

}
```

Listing 8: The `UserManagement` Client deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application-client PUBLIC '-//Sun Microsystems, Inc.//DTD
J2EE Application Client 1.3//EN'
'http://java.sun.com/dtd/application-client_1_3.dtd'>

<application-client>
  <display-name>Section2Client</display-name>
  <ejb-ref>
    <ejb-ref-name>ejb/UserManagement</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.rickhightower.auth.UserManagementHome</home>
    <remote>com.rickhightower.auth.UserManagement</remote>
    <ejb-link></ejb-link>
  </ejb-ref>
</application-client>
```

In addition, if you want the *.jar* file that contains the client to be executable, then you have to specify client's class file name as the main class of the *.jar* file in the manifest file (*MANIFEST.MF*) as follows:

```
Manifest-Version: 1.0
Main-Class: com.rickhightower.client.Section2Client
```

Section 8. Example 1: Compiling and packaging your application

Compiling and packaging your application

Now that I have shown all the code to the CMP enabled entity bean, let's look at how to package and actually deploy it. To facilitate deployment, I have created an Ant build script that does the following:

- compiles all of the source code
- packages the enterprise beans into two *.jar* files: one for the session bean (*userMgmt.jar*) and one for all of the entities (*userEntity.jar*)
- packages the client code into a *.jar* file (*client-userMgmt.jar*)
- Uses SQL DDL to create sample database tables
- Packages the *.jar* files into an *.ear* file (*app.ear*)

Note that an*.ear* file is an Enterprise Archive file. Basically it is a file that can contain other *.jar* files and *.war* files. *Ear* files can be read and processed by using the version of `deploytool` that ships with the reference implementation.

The to compile and package your application you will need to setup your environment. Setup you environment as follows:

```
set USR_ROOT=c:
set JAVA_HOME=%USR_ROOT%\jdk1.3
set J2EE_HOME=%USR_ROOT%\j2sdkee1.3
set ANT_HOME=%USR_ROOT%\tools\Ant
PATH=%PATH%;%ANT_HOME%\bin;%J2EE_HOME%\bin
```

The above example assumes you are using a Windows box and that you have installed the Java SDK in *c:\jdk1.3*, the J2EE SDK in *c:\j2sdkee1.3* and that you have installed the Ant build application in *c:\tools\ant*. Create a batch file similar to the one listed above and adjust the directory location accordingly. Modification will have to be made for developers on other platforms. (I expect any Unix jocks to translate the above easily into thier shell script of choice so I wont insult thier intelligence by adding directions.)

Deploying your application with `deploytool`

To run the Ant build script go to cmpCmr/seciton2 and type `ant package` on the command prompt. Using Ant with no arguments will display a message describing all of

the options to run the Ant script shown as follows:

Output of running Ant build script with no arguments

```
It is best to set the J2EE_HOME environment variable
before running this build script.

To compile the example use Ant as follows:

ant compile

To compile and package the examples use the following:

ant package

To create sample tables for this example use the following:

ant createTables
```

The last step (SQL DDL) is more for EJB implementation that have robust entity to SQL table mapping, which the reference implementation does not. I left this in there just in case you want to port the examples to your application server of choice. It is just starter SQL DDL code and only does the first table as the reference implementation creates the tables for you.

Once you use the Ant build script to compile the source, package the classes into jars, and package the jars into an *.ear* file, then you will want to open the *.ear* file with deploytool which is described in the next section.

Deploying your application with `deploytool`

Most application server's ship with deployment tools. These tools typically allow you to deploy your application and to map entity beans and CMP fields to tables and columns in a database. The reference implementation is no different. It ships with a deploy tool called `deploytool`.

The rest of this panel assumes you will be using the reference implementation and Ant to compile, package, and deploy your application. Make adjustments for your application server where necessary. For notes on how to setup your environment see the previous section.

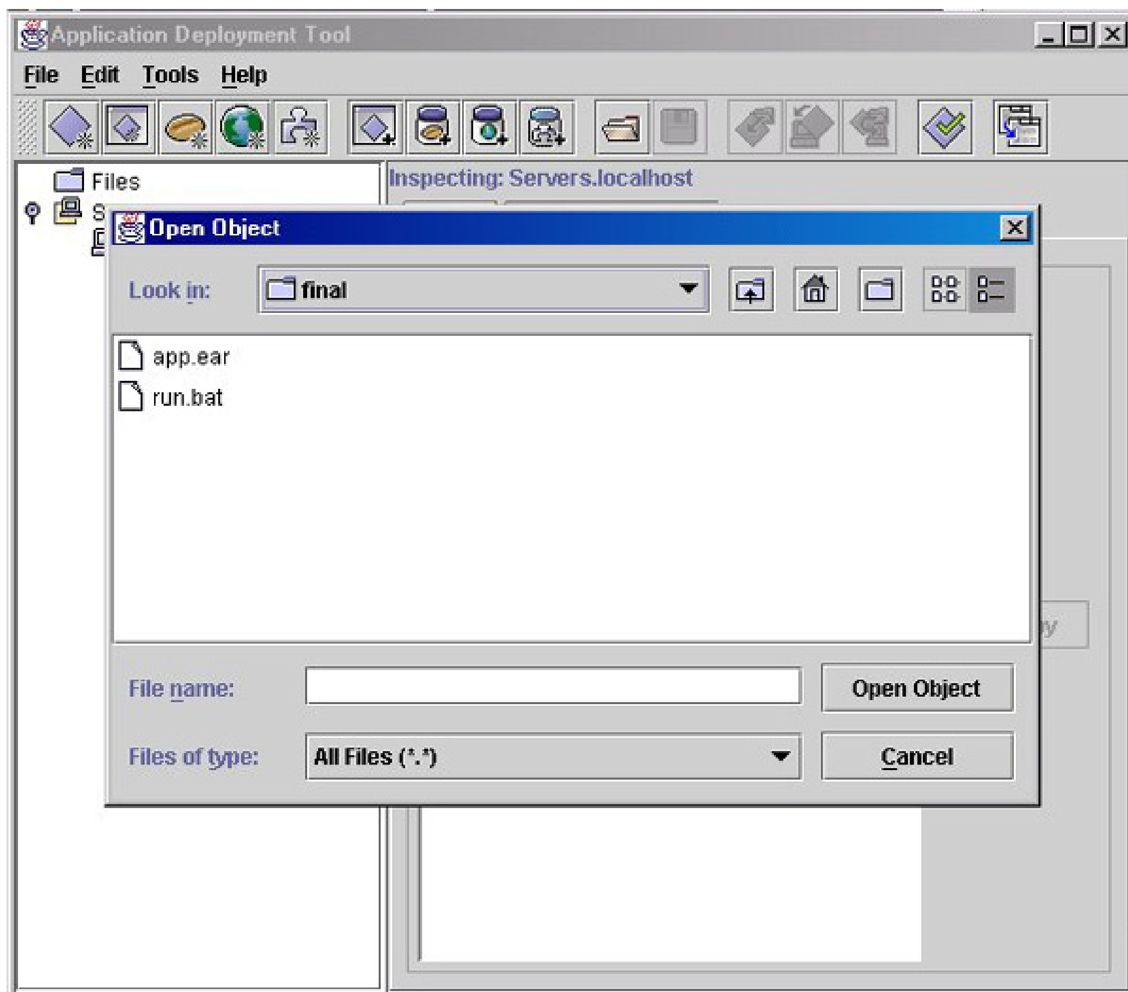
In order to deploy the *.ear* file, you will need to run the J2EE reference implementation, and the Cloudscape RDBMS system. Once the J2EE application server and Cloudscape are running you will want to run the deploy tool described as follows:

```
rem calls a batch file that defines proper environment variables
call setenv
cd j2sdkee1.3
cd bin
start "CLOUDSCAPE" cloudscape -start
start "J2EE SERVER" j2ee -verbose
start "DEPLOY TOOL" deploytool
```

Deploying your application with `deploytool`

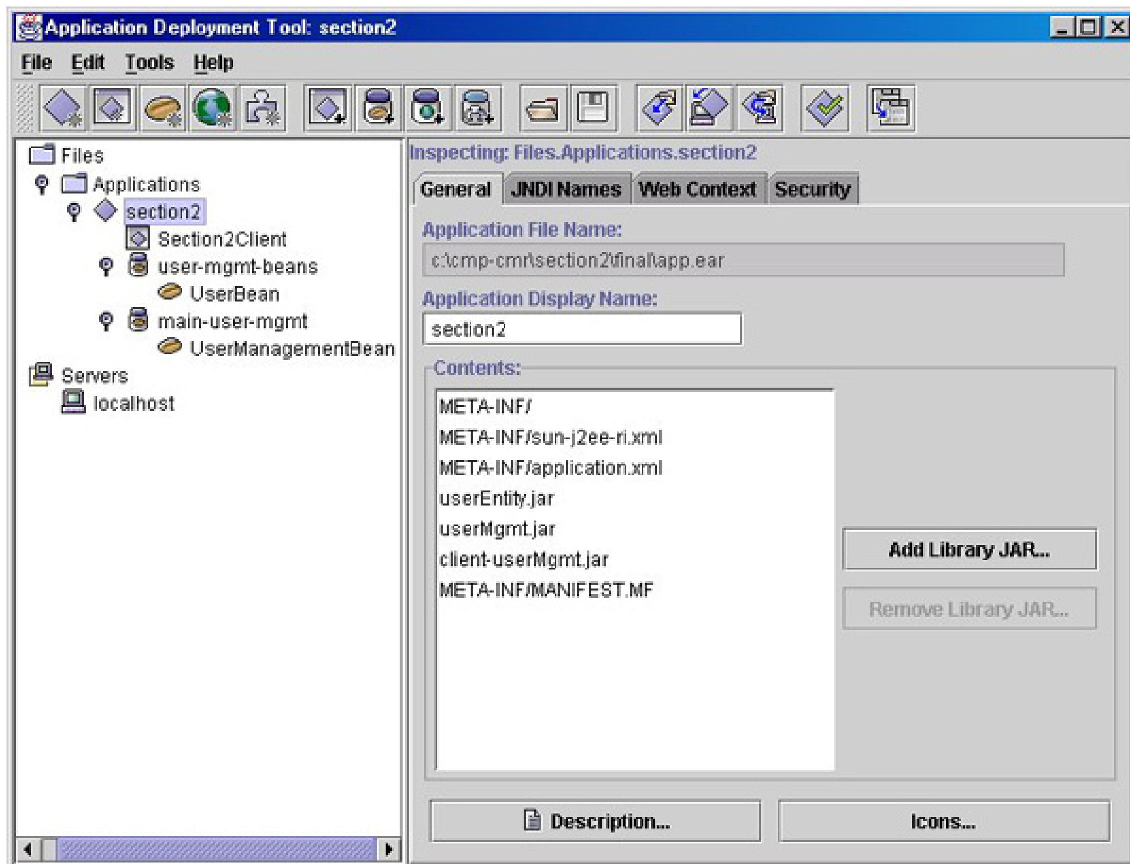
Once everything is running, you can use `deploytool` to open the *.ear* file (app.ear) located in *cmpCmr/section2/final* as pictured below.

Figure 5: Opening the EAR file with `deploytool`



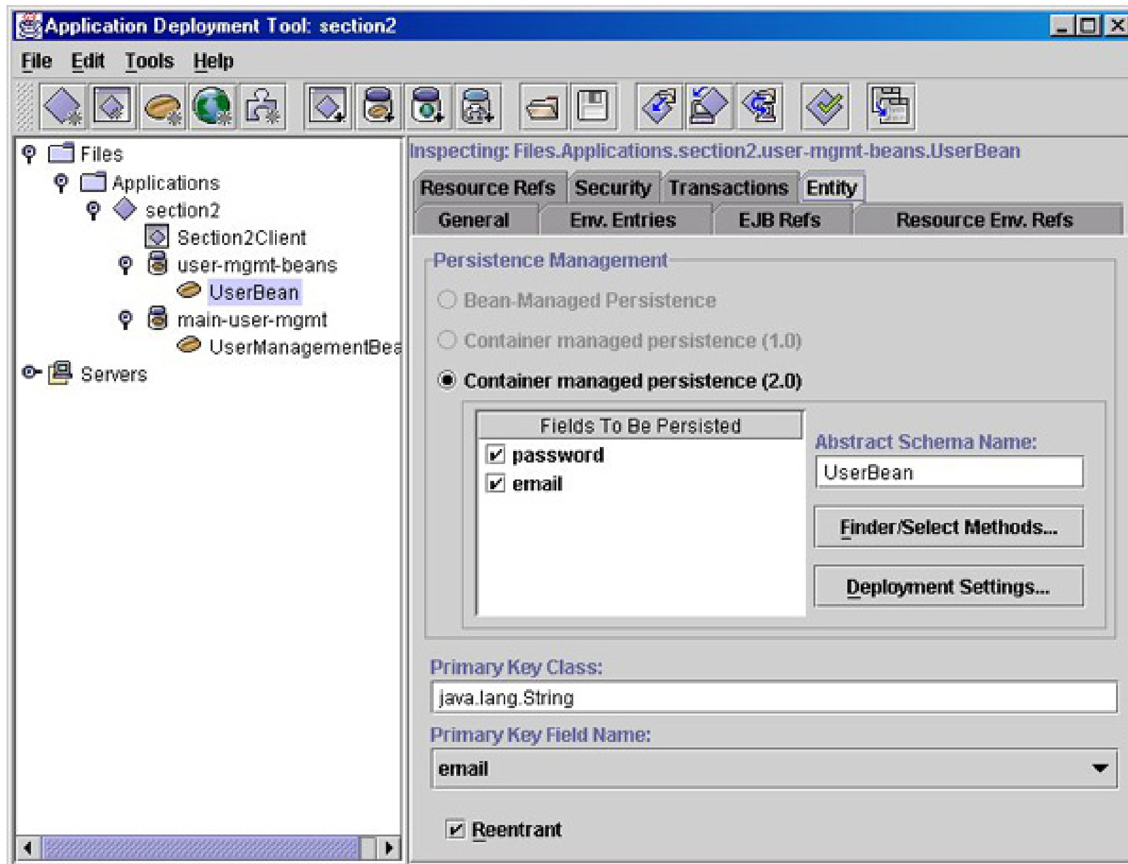
Once `deploytool` opens the *.ear* file you can browse it. Please look over the *.ear* file and see that the *.jar* file for the client, the session, and the entity bean are contained within the *.ear* file pictured as follows:

Figure 6: Displaying *.jar* files in the *.ear* file with `deploytool`



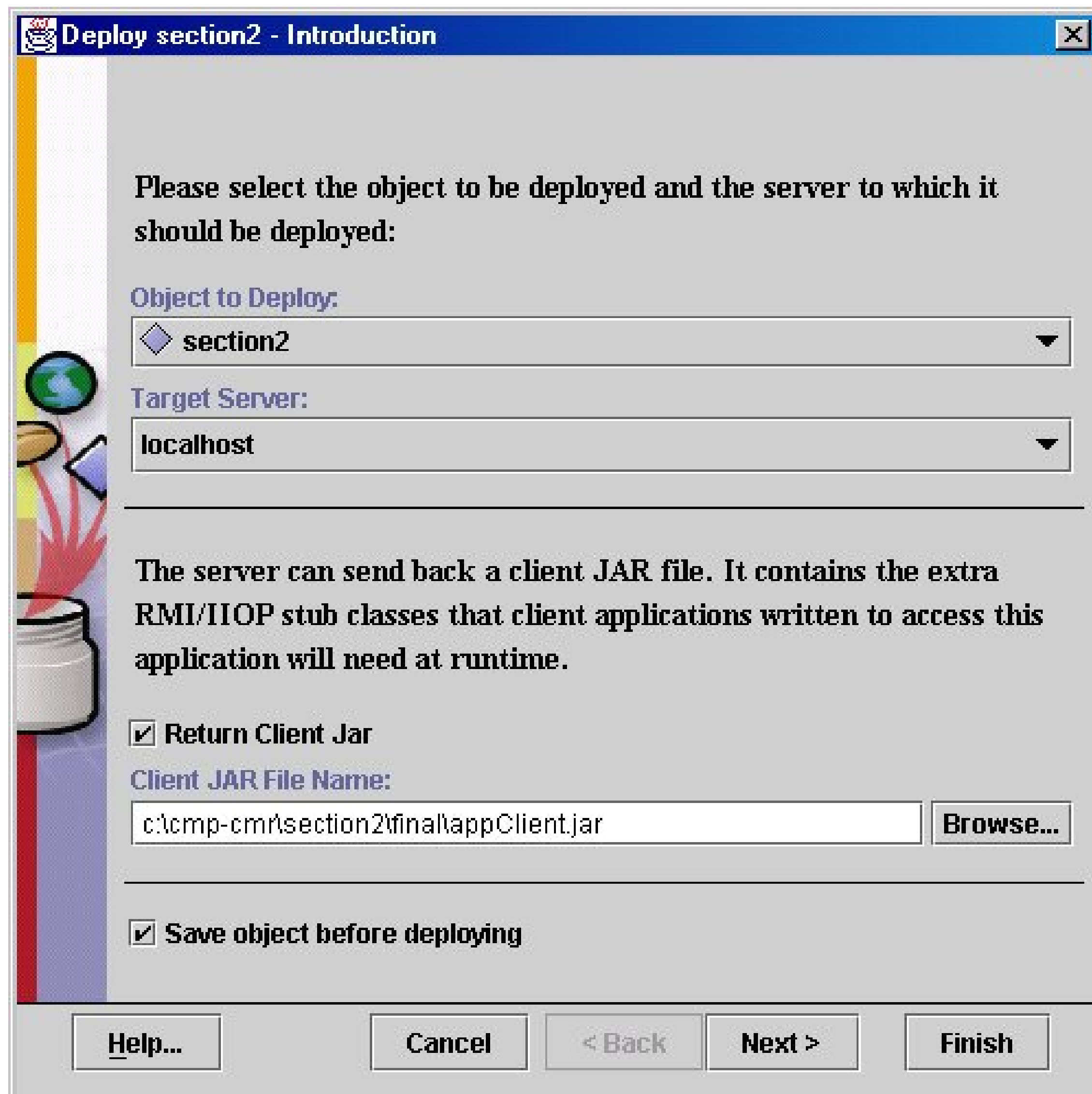
Also notice that if you select the `UserBean` in the tree view and then select the Entity tab in the detail view that you can see the CMP fields of the `UserBean` pictured as follows.

Figure 7: Displaying CMP fields in the `UserBean` with `deploytool`



Once you are done browsing *.ear* file then you can deploy it. To deploy the *.ear* file go to the tools menu and select the **Deploy...** menu item. On the first step of the deployment wizard check the checkbox **Return Client Jar**, then hit the finish button to deploy the *.ear* file to the container. (Note the reference implementation must be running see the J2EE SDK documents for more details -- also see note below on setting up you environment.)

Figure 8: Deploying the *.ear* file with `deploytool`



Notice that the `deploytool` shows the *.ear* file as deployed in the Servers/localhost element in the tree view. Also notice that the `deploytool` deposited a client application *.jar* file in the *cmpCmr/section2/final* directory. The next step is to actually run the client, which is described in the next section.

Deploying your application with `deploytool`

To run the client go to *cmpCmr/section2/final* on the command prompt and enter the following on the command prompt.

```
runclient -client app.ear -name Section2Client-textauth
```

The default username and password for the reference implementation clients is *guest* and *guest123* respectively. The session client output would look like the following:

```
C:\cmp-cmr\section2\final>runclient -client app.ear -name Section2Client -textauth
Initiating login ...
Username = null
Enter Username:guest
Enter Password:guest123
Binding name:`java:comp/env/ejb/UserManagement`
class type com.rickhightower.auth._UserManagementHome_Stub
Login =true
Unbinding name:`java:comp/env/ejb/UserManagement`
```

Details on Ant build script

For completeness, the Ant build script is included below. If you have not interest in this topic, you can safely skip this panel and go on with the rest of the tutorial without skipping a beat.

Note from the Author

I found it quite necessary to create an Ant build script as I wanted to incrementally add complexity to the application. At first, I tried to to use just the `deploytool`. But I found after each change that I had to repeat a 50 step process to redeploy the application. I also figured that after each step the reader would have to repeat the same 50 step process. Thus I created an Ant build script to automate the task of building the example into an *.ear* file. The key to make this work was to put the files that the `deploytool` created back into the META-INF directories of the various subcomponents. Note 50 steps may seem like a slight exaggeration, and maybe it is, but even 12 steps executed in order is very time consuming and error prone.

Read the description elements of the Ant build script to get an idea of what it is doing. Since the Ant build script is in simple XML and you already get the gist of what it is doing the build script is somewhat easy to follow.

Listing 9: The Ant build script

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="about" name="cmp-cmr">
    <property environment="myenv" />

    <property name="driver" value="com.jnetdirect.jsql.JSQLDriver"/>
    <property name="driver-classpath" value="/cvs/lib/JSQLconnect/JSQLConnect.jar"/>
```

```
    <property name="password" value="cruelsummer"/>
    <property name="userid" value="user1"/>
    <property name="url" value="jdbc:JSQLConnect://CANIS_MAJOR:1433/database=auth"/>
    <property name="j2ee-lib" value="${myenv.J2EE_HOME}/lib/j2ee.jar"/>
    <property name="outdir" value="/tmp/cmp-cmr"/>
    <property name="final_output" value="./final"/>


        <!-- relative to outdir -->
    <property name="client" value="${outdir}/ejb-jar-client"/>
    <property name="build" value="${outdir}/ejb-jar"/>
    <property name="meta-inf" value="${build}/META-INF"/>
    <property name="dist" value="${outdir}/dist"/>
    <property name="lib" value="${outdir}/lib"/>
    <property name="jar_name_user" value="userEntity.jar"/>
    <property name="jar_name_user_mgmt" value="userMgmt.jar"/>
    <property name="app_ear" value="app.ear"/>



    <target description="prepare the output directory." name="prepare">
        <mkdir dir="${build}"/>
        <mkdir dir="${lib}"/>
        <mkdir dir="${meta-inf}"/>
        <mkdir dir="${client}"/>
        <mkdir dir="${dist}"/>
        <mkdir dir="${final_output}"/>
    </target>



    <target name="dropTables">

        <sql classpath="${driver-classpath}" driver="${driver}"
                      password="${password}" url="${url}" userid="greek">

            DROP TABLE TBL_USER

        </sql>

    </target>

    <target name="createTables">

        <sql classpath="${driver-classpath}" driver="${driver}"
                      password="${password}" url="${url}" userid="greek">

            CREATE TABLE TBL_USER (
                email varchar (50)  PRIMARY KEY,
                password varchar (50) NOT NULL
            )

        </sql>

    </target>

    <target depends="prepare"
                             description="compile the Java source."
```

```
                         name="compile">
        <echo> J2EE lib is set to ${j2ee-lib}</echo>

        <javac destdir="${build}" srcdir="./src">
            <classpath>
                <pathelement location="${j2ee-lib}"/>
                <pathelement location="${junit-lib}"/>
            </classpath>
        </javac>
    </target>

    <target depends="compile"
                   description="package the Java classes into a jars."
                         name="package">

        <copy todir="${client}">
             <fileset dir="${build}">
                <patternset id="Client">
                    <exclude name="**/*Bean*"/>
                </patternset>
             </fileset>
        </copy>

                  <!-- jar the client -->
        <jar basedir="${client}" jarfile="${lib}/client.jar"/>

                  <!-- jar the entity bean -->
        <jar jarfile="${dist}/${jar_name_user}">
                  <fileset dir="./src/META-INF/Entity" />
             <fileset dir="${build}">
                <patternset id="UserEntity">
                    <include name="**/*LocalUser*"/>
                    <include name="**/*UserBean.class"/>
                </patternset>
             </fileset>
        </jar>

                  <!-- jar the session bean -->
        <jar jarfile="${dist}/${jar_name_user_mgmt}">
             <fileset dir="./src/META-INF/Session" />
             <fileset dir="${build}">
                <patternset id="UserMgmt">
                    <include name="**/*LocalUser*"/>
                    <include name="**/*UserManagement*.class"/>
                    <include name="**/*CreateWrapperException.class"/>
                </patternset>
             </fileset>
        </jar>

                  <!-- jar the client as per refernce implementation -->
        <jar jarfile="${dist}/client-${jar_name_user_mgmt}">
             <fileset dir="./src/META-INF/App-Client" />
             <fileset dir="${build}">
                <patternset id="App-Client">
                    <include name="**/*Client*"/>
                    <include name="**/*UserManagement*.class"/>
                    <exclude name="**/*Bean*"/>
                </patternset>
```

```
            </fileset>
        </jar>

                 <!-- jar the jars into an ear -->
        <jar jarfile="${final_output}/${app_ear}">
             <fileset dir="./src/META-INF/Application" />
             <fileset dir="${dist}">
                <patternset id="Application">
                    <include name="**/*.jar"/>
                </patternset>
             </fileset>
        </jar>
    </target>

    <target name="clean" depends="prepare"
                          description="clean up the output directory.">
        <delete dir="${outdir}" />
             <delete dir="${final_output}" >
           <patternset id="clean" >
                 <include name="**/*.ear"/>
                 <include name="**/*.jar"/>
                </patternset>
             </delete>
    </target>


<target name="about" >
        <echo>
It is best to set the J2EE_HOME environment variable
before running this build script.

To compile the example use Ant as follows:

ant compile

To compile and package the examples use the following:

ant package

To create sample tables for this example use the following:

ant createTables

        </echo>
</target>

</project>
```

Details on application deployment descriptor for *.ear* file

Also for completeness, here is the *.ear* file deployment descriptor that lists all of the

modules that is uses as follows:

Listing 10: The *.ear* file deployment descriptor (application.xml)

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN'
 'http://java.sun.com/dtd/application_1_3.dtd'>

<application>
  <display-name>section2</display-name>
  <description>Example tutorial written for IBM developerWorks by www.rickhightower.com</c

  <module>
    <ejb>userEntity.jar</ejb>
  </module>
  <module>
    <ejb>userMgmt.jar</ejb>
  </module>

  <module>
    <java>client-userMgmt.jar</java>
  </module>


</application>
```

Notes on porting the application to other EJB containers

Note that although this example uses the references implementation, it should not take that much effort to modify the build script and deployment descriptors slightly to accommodate any compliant EJB 2.0 container.

The closer the EJB container is to compliance the easier it will be to perform the conversion.

Also note that you have to change the JDBC driver and JDBC URL property to match your database, if you want to use the build script. Thus all you may have to really change is the following:

```
    <property name="driver-classpath" value="/cvs/lib/JSQLconnect/JSQLConnect.jar"/>
    <property name="password" value="cruelsummer"/>
    <property name="userid" value="user1"/>
    <property name="url" value="jdbc:JSQLConnect://CANIS_MAJOR:1433/database=auth"/>
```

At this point you should have a working version of the example application. Next, you will add finder methods with EJB-QL.

Section 9. Example 2: Finder methods and EJB-QL

Finder methods and EJB-QL

The Enterprise JavaBeans Query Language (EJB-QL), a subset of SQL92, allows the definition of finder methods in the deployment descriptor for entity beans with container-managed persistence. EJB-QL is a powerful little language that is easy to learn and use. Like SQL, EJB-QL is easy to learn, but takes some time to master.

EJB-QL was extended to navigate over CMR relationships. The EJB container is responsible for translating EJB-QL queries into the query language of the persistent datastore, for example, SQL-92 for IBM DB2. This translation results in creating finder methods that are more portable. This allows enterprise component vendors to sell components that work on more target platforms. The third tutorial in this series will cover EJB-QL in detail. Here is a small taste.

Creating a `findAll()` method for `UserBean`

Finder methods allow entities to be found based on a criteria. Finder methods are defined in the home interface. Prior to EJB 2.0 it was up to the developer to create his own finder methods (or a propietary EJB vendor solution). The developer would declare a finder method in the home interface let's say `findAll()` then the developer would define the corresponding definition of the finder method in the bean implementation for example `ejbFindAll()`.

Listing 11: Home interface for `findAll()`

```
/** Home interface now has findAll method **/
package com.rickhightower.auth;

import java.util.Collection;

public interface LocalUserHome extends EJBLocalHome {

    public Collection findAll() throws FinderException;
}
```

With EJB 2.0 CMP, you merely define the finder method definition in the deployment descriptor using EJB-QL. You still have to declare the finder method in the home interface. Note that the finder method `public Collection findAll() throws FinderException` is declared in the home.

Listing 12: Deployment descriptor for `findAll()` query

```
<entity>
  <display-name>UserBean</display-name>
  <ejb-name>UserBean</ejb-name>

  <query>
    <description></description>
    <query-method>
      <method-name>findAll</method-name>
      <method-params />
    </query-method>
    <ejb-ql>select Object(theUser) from UserBean as theUser</ejb-ql>
  </query>

</entity>
```

Also note that the finder method `findAll()` is defined using the query element. The query element defines the method name, and method parameters in this case `findAll()` and no parameters. Note that the body of the ejb-ql element defines the actual query `select Object(theUser) from UserBean as theUser`. This syntax is very similar to SQL.

Next I will demonstrate using the `findAll()` method from the session bean by implmenting a `getUsers()` method.

Adding a `getUsers()` methods to the `UserManagementBean`

To use the home interfaces `findAll()` method, you add a `getUsers()` method to the `UserManagementBean`. The implementation of `getUsers()` in the `UserManagementBean` class calls the `findAll()` method. It then iterates through the collection of `LocalUser` objects and puts their email property in an ArrayList. The `getUsers()` method cannot simply return the `Collection` sent by `findAll()` since a local bean can not be sent remotely.

Listing 13: Interface for `getUsers()`

```
/** Add the getUsers method to the interface */

public interface UserManagement extends EJBObject {
    public Collection getUsers() throws RemoteException;

}
```

Listing 14: Implementation for `getUsers()`

```
/** Add the getUsers method to the implementation */

public class UserManagementBean implements SessionBean {

    public Collection getUsers(){

         ArrayList userList = new ArrayList(50);
         Collection collection = userHome.findAll();
         Iterator iterator = collection.iterator();
         while(iterator.hasNext()){
             LocalUser user = (LocalUser)iterator.next();
             userList.add(user.getEmail());
         }
         return userList;

    }
```

Using the `getUsers()` methods from the client

The client code invokes the `getUsers()` method. It then takes the collection of emails that uniquely identify a user returned from the `getUsers()` method and prints them out to standard out.

```
public class Section2Client {

   public static void main(String[] args) {

           Collection collection = userMgmt.getUsers();
           Iterator iterator = collection.iterator();
           while(iterator.hasNext()){
                String email = (String)iterator.next();
                System.out.println("user id=" + email);
           }

   }

}
```

Compiling and deploying the `findAll()` EJB-QL example

The same technique can be used to the build, package, and deploy the findAll example as was used to build, package, and deploy the first example which can be found at the on page panel. All of the source and build file for this example can be found at *cmpCmr/section2.2*.

Be sure to run `ant clean` to delete the old examples intermediate build files.

Section 10. Summary

Whew!

You are done with the first two examples. All the other examples are based on these first two examples.

I have covered the definition of the CMP Entity bean; creating a Session bean that references and uses the Entity bean; and creating a client that references and uses the Session bean. Then I extended the Entity bean by adding a finder method to its home interface. The implemenation of the finder method was defined with EJB-QL in the entity beans deployment descriptor.

If you made it this far, congratulations! You are at the top of the hill. It is all easy riding down-hill from here.

The next tutorial in this series will cover defining container managed relationships.

Resources

- A good *tutorial on EJB CMP/CMR and EJB-QL*
- The *J2EE tutorial* from Sun
- *Developer's Guide to Understanding EJB 2.0 (updated by Rick Hightower)*
- *Enterprise JavaBeans fundamentals*

Books:

- *Mastering Enterprise JavaBeans (2nd Edition)* by Ed Roman, Scott W. Ambler, Tyler Jewell, Floyd Marinescu

  The EJB Encyclodpedia!
- *Enterprise JavaBeans (3rd Edition)* by Richard Monson-Haefel

  Get this one too!
- *Java Tools for Extreme Programming* by Richard Hightower, Nicholas Lesiecki

  Covers building and deploying J2EE applications with EJBs.

Section 11. Feedback

Feedback

Please let us know whether this tutorial was helpful to you and how I could make it better. I'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.