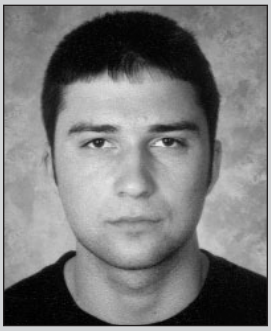
Persist Data for Your J2EE Applications with Less Effort Using Entity Beans with Container-Managed Persistence (EJB CMP)

Christian Fecht and Svetoslav Manolov

Christian Fecht, Java Server Technology Group, SAP AG



Svetoslav Manolov, Java Server Technology Group, SAP Labs Bulgaria

(complete bios appear on page 166)

If you are a Java programmer, then you are likely no stranger to the following scenario: you need to persist Java objects in a relational database, and you are facing the chore of once again writing the SQL code for storing the objects to and loading them from the database — a tedious and time-consuming process. Like any developer, you'd much rather be focusing on the actual application than wallowing around in the nitty-gritty details of accessing a database from Java.

What you are wishing for is an easy way to bind Java objects to their persistent representations in the database in a transparent and (mostly) automated way — a concept known as "transparent object persistence." There are two standard technologies available in the Java world that provide such persistence for Java objects: entity beans with container-managed persistence (CMP entity beans) and Java Data Objects (JDO). CMP entity beans are a part of the Enterprise JavaBeans (EJB) specification (itself part of the J2EE standard), and provide transparent persistence at the EJB component level. JDO, which is essentially a persistence enhancement to the Java language, provides transparent persistence for ordinary Java objects.

But what about persisting Java objects in the SAP world? Usually this means using SQLJ or JDBC to write all the necessary SQL code for moving the object representation to and from the database — until now. SAP Web Application Server (SAP Web AS) 6.40, the core of SAP NetWeaver, supports both CMP entity beans and JDO as part of its overall persistence offering. Both JDO and CMP entity beans run on top of the Open SQL Engine in SAP Web AS, and automatically benefit from the portability and built-in enhancements of Open SQL for Java, including table buffering, statement pooling, and SQL tracing. This

article is the first in a series that will provide you with insight into these two object persistence technologies for Java. This article introduces you to CMP entity beans; a future article will focus on JDO.

Be aware that EJB in general, and CMP entity beans in particular, are broad topics, and we cannot cover all the details here. The goal of this article, then, is to introduce you to the fundamentals of CMP entity beans, and, using an example, give you a feel for what it means to develop and use them. By the end of this article, you will be able to develop your first CMP entity beans with SAP NetWeaver Developer Studio[1] and run them in SAP Web AS 6.40, so that you can provide transparent persistence for the EJB components that make up your custom J2EE applications.

> ✓ ***Note!***
>
> *This article assumes a solid knowledge of Java and persistence,[2] and some basic knowledge of EJB and XML.*

Before diving into the details of EJB CMP development, we start with a gentle, but thorough introduction to transparent object persistence and EJB technology in general.

Transparent Object Persistence

Since Java is an object-oriented programming language, Java programmers tend to think, and model their application domains, in terms of objects. An object has a "state" that is represented at any given point in time by its instance fields and their corresponding values. "Persisting" an object basically means storing the relevant state of the object in an external data store, typically a relational database. Later on, the persisted state can be retrieved from the data store — to populate a Java object, for example. With JDBC or SQLJ alone, the Java programmer must write all the SQL code necessary to move the state of the object back and forth between the data store and the Java object. For example, to save an Employee object, you would write code to insert the employee's instance fields — first name, last name, salary, and so on — into the data store. To load an Employee object from the data store, you would have to write additional code that first fetches the stored values of the employee's instance fields from the data store, then instantiates an Employee object in the Java virtual machine (JVM), and finally populates the newly created Employee instance with the retrieved data.

Transparent object persistence strives to automate all the tasks necessary to persist objects in a data store, so that binding Java objects to persistent data in a data store is as easy, seamless, and automated as possible. Transparent object persistence allows developers to work with persistent Java objects directly in their code, just as they would work with any other application objects. Keep in mind that with persistence, there is a difference between the persistent entity in the data store and the Java object representing that entity in the JVM — the persistent entity in the data store is the real thing. It represents an entity, such as an Employee object, from the application domain, and it lives in the data store until it is explicitly removed from it. The Java object, on the other hand, allows for an object-oriented view of the persistent entity in the data store. Transparent object persistence makes the persistent entity and its Java object representation appear as a single logical object from the application programmer's perspective.

In the next sections, we'll look at some of the key features of transparent object persistence: transparent synchronization, transparent loading, and transparent object-relational mapping.

---

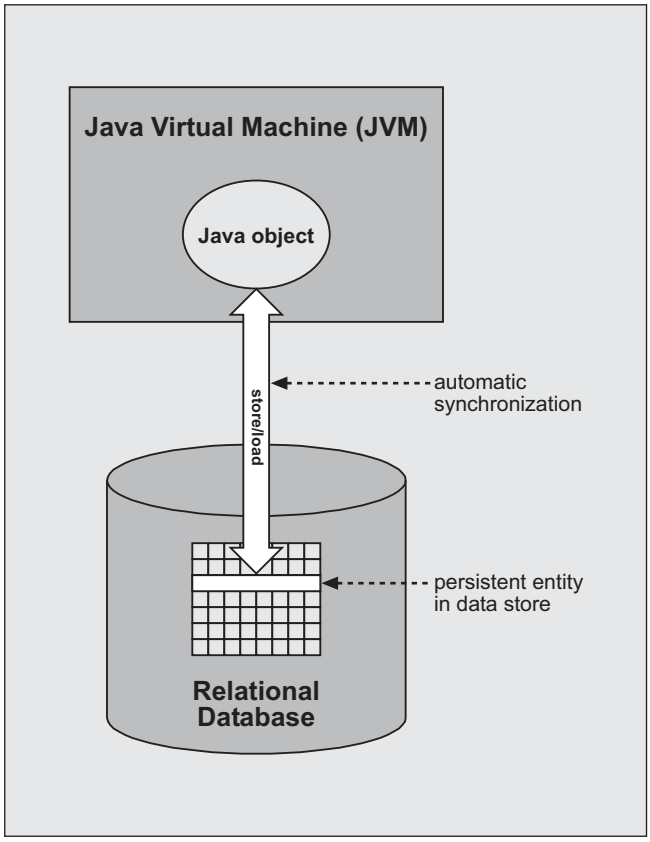
[1] See the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" in the May/June 2004 issue of *SAP Professional Journal* for more on this new IDE for Java applications.

[2] For a detailed introduction to SAP Web AS support for persistence, see the article "A Guided Tour of the SAP Java Persistence Framework — Achieving Scalable Persistence for Your Java Applications" in the May/June 2004 issue of *SAP Professional Journal*.

Transparent Synchronization

During a transaction, the state of the Java object may be different than the state of the persistent entity it represents. This is a situation that can occur, for example, because the value of one of the Java object's persistent fields has been changed in an application. With transparent synchronization, however, once the transaction is committed, all changes to the Java object are automatically tracked and written back to the persistent entity in the data store. Analogously, when the Java object is accessed for the first time in the transaction, its state is fetched from the persistent entity in the data store. This means that the states of the persistent entity and its Java object representation are automatically kept in sync. **Figure 1** is a graphical representation of this process.

Figure 1 The Transparent Synchronization Process



Transparent Loading

All persistent entities that an application needs are transparently loaded from the data store and represented by Java objects in the JVM. It does not matter if an application iterates over the result of a query or navigates through a graph of persistent Java objects; whenever it accesses a persistent Java object, the object will have been instantiated by some magic in the JVM, and will contain the current state of the persistent entity it represents. For an application programmer, it will appear as if all persistent entities in the data store have been loaded into the JVM, although only a very small subset of the persistent entities may have been loaded and represented by Java objects.

Transparent Object-Relational Mapping

In SAP Web AS, the data store is a relational database. Therefore, persistent Java objects must be mapped to relational structures (hence the term "object-relational mapping"). Typically, an object is mapped to a table row, as shown in Figure 1. The persistent fields of the object are mapped to table columns, and references between objects are represented by foreign key relationships in the database. With transparent object persistence, the programmer doesn't need to write the code that implements the object-relational mapping, but must specify the mapping in a declarative way (more on this later in the article).

With transparent object persistence, there is a clean separation between the persistence logic and the actual business logic. Although the business logic deals with persistence-specific code (transaction demarcation, creation/deletion of persistent Java objects, etc.), the business logic is not mixed up with persistence implementation code. Moreover, the business logic works with persistent Java objects without having to know where and how they are stored.

Now that you have a solid understanding of how transparent object persistence works, we'll turn our attention to Enterprise JavaBeans. The next section provides you with a brief overview of the Enterprise JavaBeans technology and how it works.

A Quick and Gentle Introduction to Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) comprises the standard technology for building business components in the Java world. EJBs are part of the Java 2 Platform, Enterprise Edition (J2EE), which includes more than 10 other technologies[3] and over 4,000 APIs that supplement the 5,000 classes included in the main Java platform, J2SE (Java 2 Platform, Standard Edition). The richness of J2EE provides an excellent infrastructure for the development of business applications, and its benefits derive not only from its variety of technologies, but mainly from the integration and compatibility between them.

EJBs are reusable binary software components that consist of code (Java class files) and declarative descriptions (XML deployment descriptors). By changing the deployment descriptors (more on this later in the article), it is possible to customize an EJB component without having to touch and recompile the code. Furthermore, EJBs address a number of specific needs of distributed business applications: (1) they provide persistence capabilities, as well as the transaction management and security services needed by almost all business applications; (2) the EJB components that make up the actual application can reside on different servers, and are accessed remotely in a transparent way.

Types of EJBs: Entity, Session, and Message-Driven

EJB components come in three flavors: entity, session, and message-driven. Entity beans (the type of bean we'll focus on in this article) are fine-grained objects that represent persistent data in a data store (typically a relational database). For example, an entity bean *EmployeeBean* could be used to represent an employee of a company. Session beans are more coarse-grained objects that implement behavior and processes. A session bean *HRSesBean* could implement typical HR processes, such as the hiring or promoting of employees. While session beans do not represent persistent data themselves, they can manipulate persistent data. Message-driven beans serve as asynchronous message "consumers" and provide the integration between the EJB architecture and messaging systems like Java Message Service (JMS) providers.

How EJBs Work

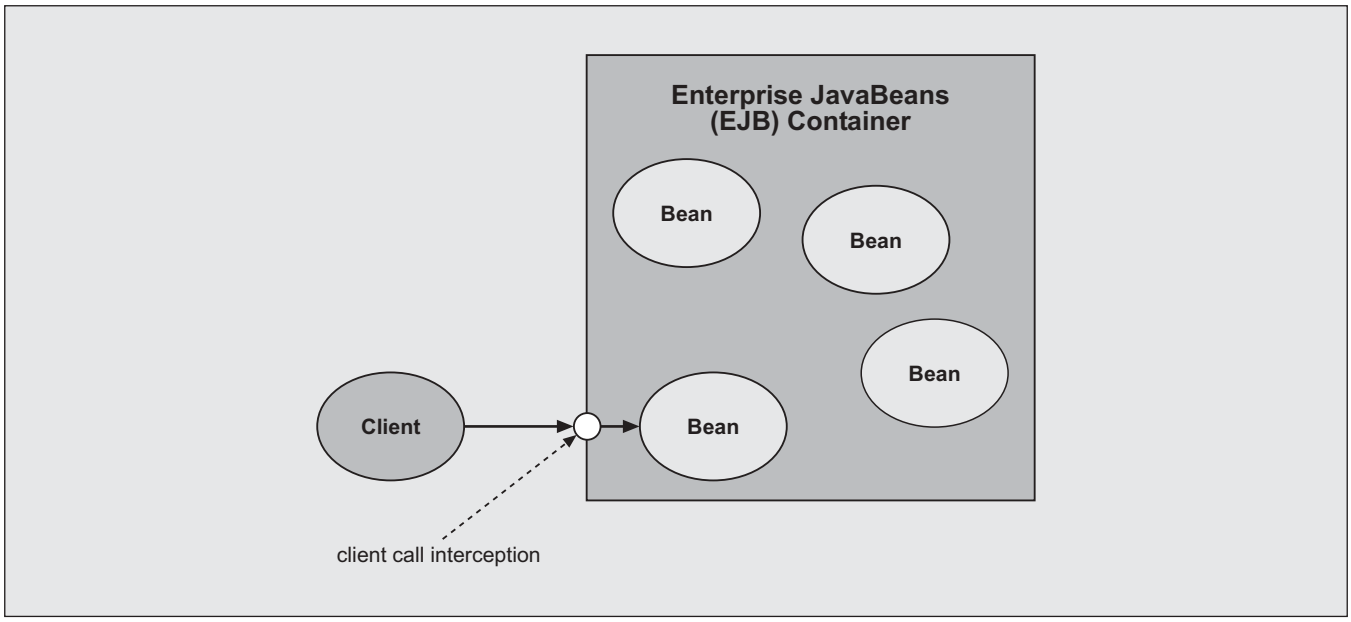
The software life of a bean starts with the "enterprise bean provider," an application developer who produces beans that implement business logic. Individual beans are then assembled into an "EJB module," which will constitute the actual EJB application that is finally deployed to a server. Next, the "EJB container," which not only provides the tools for deploying the EJB application, but also manages and controls all aspects of a bean's life once the bean has been deployed, enters the scene. The container provides a runtime environment for the bean, informs the bean about important lifecycle events, and, last but not least, makes the bean available to clients. An "EJB client" makes use of the bean and the functionality provided by the bean. In most cases, the client is a Web component (a Web Dynpro[4] or servlet/JSP component) or another bean (e.g., a session bean when using the session façade design pattern; more on this later). The client looks up the bean from the Java Naming and Directory Interface (JNDI) context in which it was stored by the EJB container.

A bean exposes its functionality to clients via Java interfaces (more on these in a moment). However, when a client calls a method on these interfaces, it does not talk to the bean directly — all client calls are intercepted and mediated by the EJB container. This interception allows the EJB container to "hook into" the client call and provide services such as persistence, transaction management, and security to both the bean provider and the client in a transparent way. The classes used for interception are generated at application deployment by the EJB container's

[3] JavaServer Pages (JSP), Java Message Service (JMS), Java Transaction API (JTA), etc.

[4] Web Dynpro is SAP's development environment for creating professional Web user interfaces for business applications. It is model-driven and consists of a powerful toolset and a sophisticated runtime.

Figure 2 The EJB Container Intercepts the Client Call to the Bean



deployment tools. At runtime, the interceptor object acts as a "bodyguard" between the client and the actual bean instance, as shown in **Figure 2**.

The Structure of an EJB

A bean consists of:

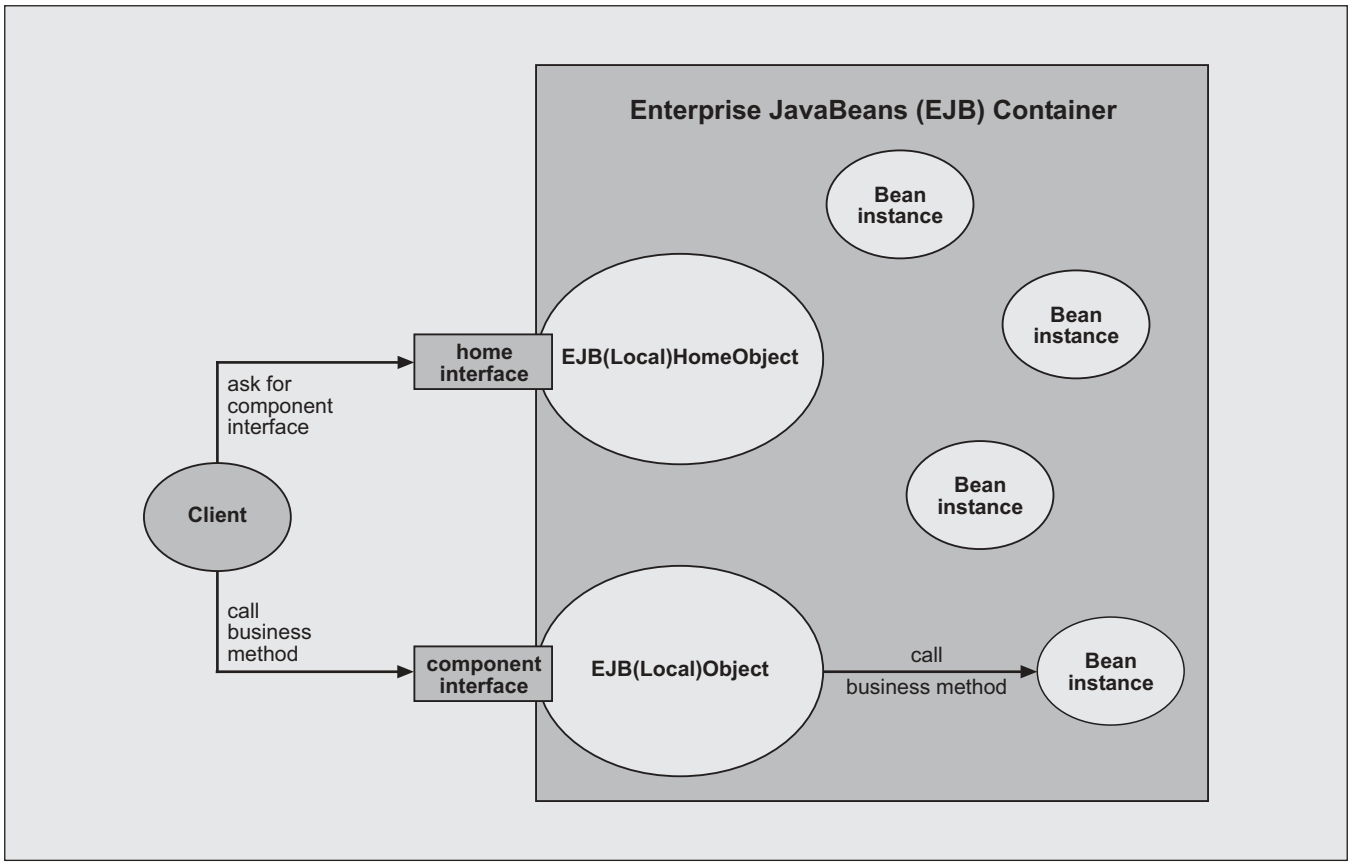
- Home and component interfaces, through which it is accessed by its clients
- A bean class containing the business logic implementation
- A deployment descriptor containing the metadata that describes the component

A bean exposes its business methods to the client through its component interface. The business methods in the component interface are implemented by the bean class. The instances of the bean class are called "bean instances." Bean instances are managed by the EJB container and cannot be directly accessed by the client.

Each deployed bean has a "home" that is represented by a home object and accessed by clients through the home interface. The home object's job is to hand out references to that bean's component interface via the home interface. When a bean is deployed, the EJB container instantiates the bean's home object and registers it with JNDI. A client that wants to use a bean first performs a JNDI lookup of the bean's home object using its JNDI-registered name, and then asks the home object for a reference to the bean's component interface. Through this reference, the client can now do what it really wants to do — call business methods on the bean.

There are two different kinds of clients: remote and local. A local client is located in the same JVM as the beans used by the client. In this case, method calls on the beans are just ordinary Java method calls. A remote client, on the other hand, may reside in a different JVM, or even on a server other than the one hosting the beans it wants to use. In this case, the client communicates with the beans via Remote Method Invocation (RMI) method calls — objects are passed by value, arguments and the return value must be serializable, etc. Accordingly, there are two versions of the component and home interfaces: local and remote.

Figure 3 Implementing the Home and Component Interfaces



Since the EJB container intercepts all client calls on the component interface, the bean class cannot implement the component interface in the way a Java class implements a Java interface. So how, then, are component interfaces implemented? The actual implementation of a component interface is provided by a class that the EJB container generates at deployment time. The instances of this container-generated class, called "EJB objects," intercept the client calls, provide transaction and security management, and delegate the calls to the bean instances. An EJB object acts as the "bodyguard" between the client and the actual bean instance. The home interface is implemented analogously by "EJB home objects," which are similarly generated by the EJB container during deployment. **Figure 3** shows the relationships between the client, the component and home interfaces, the bean instances, and the container-generated EJB objects and EJB home objects.

So how does the EJB container know what to do when its container-generated class intercepts a method call on a bean? This information is conveyed to the EJB container through the deployment descriptor. The standard deployment descriptor (according to the EJB specification) is an XML file called *ejb-jar.xml*. It contains some basic information needed to define EJB components and several optional properties that describe special features (e.g., transaction and security management), the use of external resources, and so on. EJB container vendors also provide vendor-specific deployment descriptors containing metadata specific either to the application or to the EJB container. The EJB container uses the information from the deployment descriptors to provide the environment for the EJB components.

As we mentioned earlier in the EJB discussion, there are three types of Enterprise JavaBeans: entity,

session, and message-driven. In the next section, we will take a closer look at the type of EJB that enables you to persist data for your J2EE applications — entity beans.

EJB Entity Beans for Persisting Java Objects

Entity beans are an object-oriented way of looking into a data store. In the world of EJB components, their task is to represent data in a persistent data store — i.e., they provide object persistence to application programmers in a component-based form. Almost always, as is the case with SAP Web AS 6.40, the persistent data store is a relational database, and, consequently, entity beans map to relational database rows (refer back to Figure 1). With object persistence, you have to distinguish between the real thing in the database — the "persistent entity" — and its Java object representation in the JVM. The persistent entity is represented in the JVM by a bean instance, that is, by an instance of the entity bean's bean class. A bean instance is essentially a pure data object, representing the current state of a persistent entity using its instance fields. The fields of the bean class that actually represent the persistent entity attributes are called "persistent fields." The business methods implemented in the bean class manipulate the state of that persistent entity by modifying its persistent fields.

In this article, we will call the EJB objects of entity beans "entity objects," since they represent the client view of a persistent entity. When a client invokes a business method on a particular entity object, this will affect the entity represented by the object.

Entity objects are uniquely identified by a special object called the "primary key," which consists of one or more persistent fields corresponding to the primary key columns in the database table.

Types of Persistence: Container-Managed and Bean-Managed

Entity beans can have bean-managed persistence (BMP) or container-managed persistence (CMP). The difference between the two types lies in the responsibility for the persistent data synchronization.

With BMP entity beans, the bean provider (i.e., the application developer) implements the database access and writes the SQL code required to load and store the persistent state of the entity object, usually with either JDBC or SQLJ. The only benefits the developer gains from using BMP entity beans is an object-oriented view of the persistent data, and the advantage that the bean is notified by the container when it is time to load and store the persistent state of the entity.

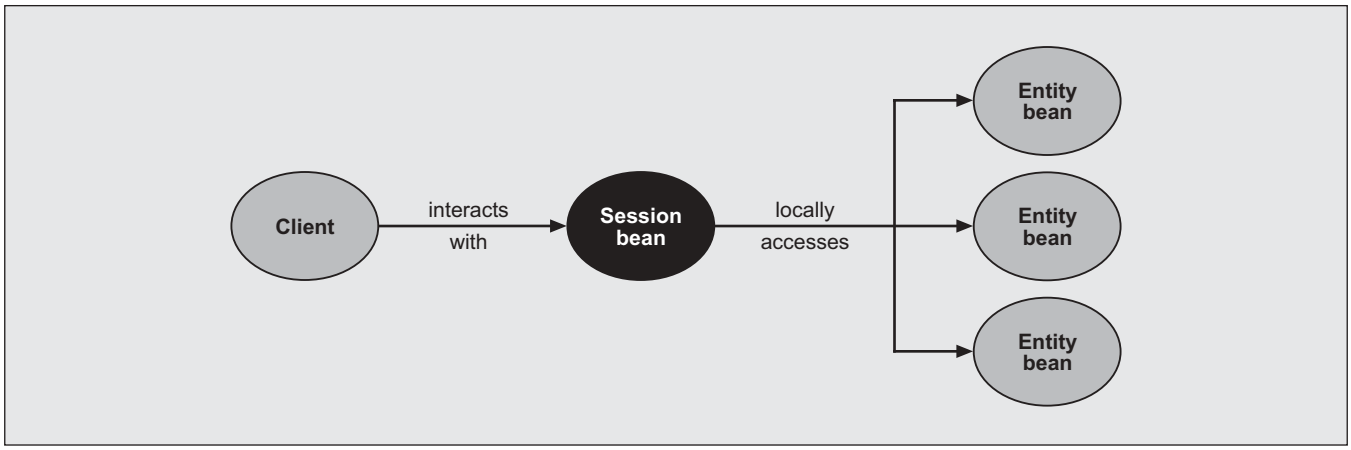
CMP entity beans, on the other hand, provide real transparent object persistence, and are the type of entity bean we will develop in this article. The entire persistence management mechanism is provided by the EJB container, and is transparent to the application developer. Using CMP entity beans requires less custom code development than BMP entity beans, and no SQL coding at all, further reducing the potential for code-induced problems.

Using a Session Façade Bean with Entity Beans

In well-defined EJB applications, entity beans are usually combined with session beans. The session bean implements a business process, and when it needs to access data in the database, it uses entity beans. The client of the application never interacts directly with the entity beans; it interacts only with the session bean, which in turn accesses the entity beans through their local home/component interfaces.

You're probably thinking, "What about the remote interfaces?" There is a consensus in the J2EE community that you should never access an entity bean through its remote interfaces. The rationale behind this recommendation is efficiency, or, in other words, the negative effect on performance caused by the

Figure 4 The Session Façade Design Pattern



fine-grained remote access of the entity bean's persistent attributes. The idea of only using local interfaces of entity beans is captured by the "session façade design pattern" (see **Figure 4**), which is one of the most important design patterns for dealing with entity beans. Therefore, for the rest of this article we will ignore remote home/component interfaces and consider only local home/component interfaces.

> ✓ ***Tip***
>
> *Never provide remote home or remote component interfaces for your entity beans.*

We're now ready to put our background knowledge to work and turn our attention to the actual bean development tasks. Over the course of the rest of this article, we will show you how and to what extent EJB entity beans with container-managed persistence provide transparent object persistence to Java programmers. In the next section, we'll get started by defining an example CMP entity bean.

Defining CMP Entity Beans

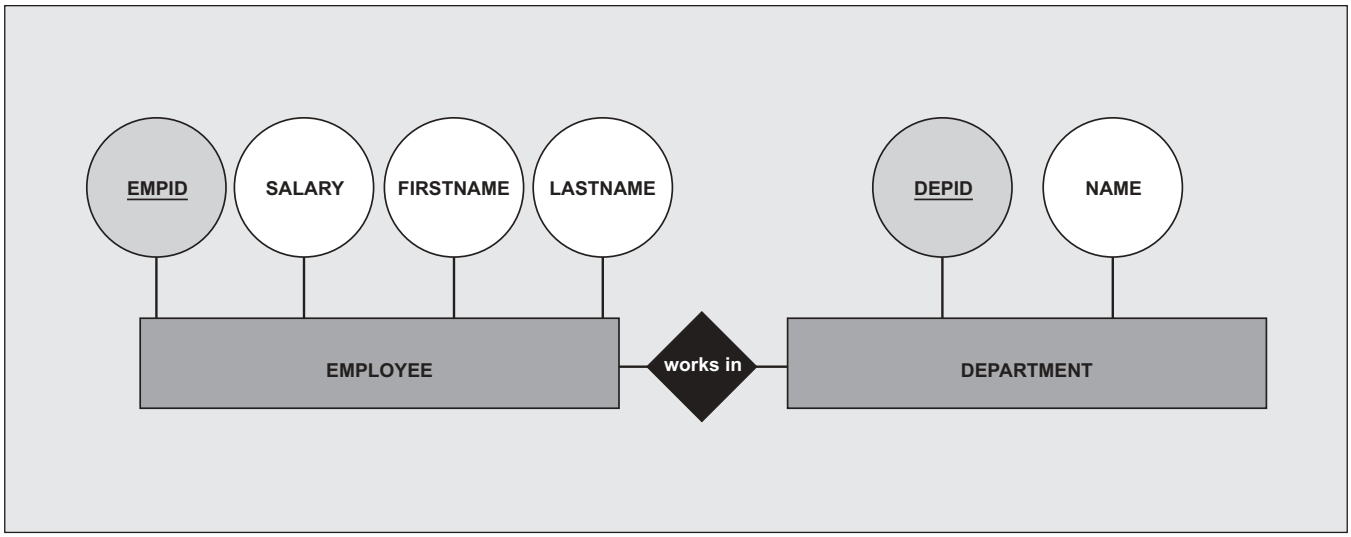
When you start developing CMP entity beans, you first must have a well-defined database model that your entity components will represent. Usually each database entity is represented by a separate entity bean, and relationships between persistent entities are represented as relationships between the corresponding entity beans. For this reason, an entity-relationship model of the persistent data is very useful when designing a CMP entity bean application.

To illustrate the CMP entity bean development process, we will describe the implementation of a simple example J2EE enterprise application consisting of two entities — Employee and Department — that contain a number of attributes. **Figure 5** shows a database model for the example, with the primary key fields *EMPID* and *DEPID* highlighted.

The relationship between the Department and Employee entities is "one-to-many" — that is, each employee can work in a single department, although a department may comprise an arbitrary number of employees. This custom J2EE application will allow the creation, removal, and update of employees and departments. A client of the application will also be able to view all existing departments, all employees, and all employees from a certain department.

To implement this model, we are going to develop two CMP entity beans in a one-to-many relationship. Then, we will create a session bean that accesses the CMP entity beans. Here are the tasks we will need to complete to create our example:

Figure 5 An Example Database Model with Two Entities



- Implement the bean classes for the entity beans.
- Define the component and home interfaces for the entity beans.
- Edit the XML deployment descriptors needed to build the application.
- Create a session bean that will access the entity beans to implement the application's business logic.

Since the standard *ejb-jar.xml* deployment descriptor does not provide information for the object-relational mapping of the CMP entity beans, this information must be provided by an auxiliary deployment descriptor. The SAP-specific object-relational mapping descriptor is called *persistent.xml*. We will specify how our persistent data should be represented in the database using the SAP NetWeaver Developer Studio object-relational mapping wizard, which will store all information in a *persistent.xml* descriptor and pack it into the application's archive.

Implement the Bean Classes for the Entity Beans

What makes a simple Java class a bean class is the implementation of the *javax.ejb.EntityBean* interface. **Figure 6** shows the declaration of the bean class for the Employee entity bean. Follow these steps to implement a bean class:

1. Create a new entity bean.
2. Define persistent fields in the bean class.
3. Define the persistent relationship fields in the bean class.

Figure 6 Declaring a Bean Class for the Employee Bean

```
public abstract EmployeeBean implements javax.ejb.EntityBean {
…
}
```

Figure 7 Defining Persistent Fields for the Employee Bean Class

```
/* CMP field employeeId */
public abstract Integer getEmployeeId();
public abstract void setEmployeeId(Integer employeeId);

/* CMP field firstName */
public abstract String getFirstName();
public abstract void setFirstName(String firstName);

/* CMP field lastName */
public abstract String getLastName();
public abstract void setLastName(String lastName);

/* CMP field salary */
public abstract BigDecimal getSalary();
public abstract void setSalary(BigDecimal salary);
```

4. Implement the *EntityBean* interface callback methods.

Step 1: Create a New Entity Bean

SAP NetWeaver Developer Studio provides a wizard for creating a new entity bean:

1. From the SAP NetWeaver Developer Studio main screen, choose *File* → *New* → *Other*....
2. On the left pane of the first wizard page, choose *J2EE* → *EJB*.
3. Choose *Next*.
4. Enter a name for the new entity bean and select the project in which it should be added.[5]
5. In the *Bean Type* dropdown list, select *Entity Bean*.
6. Choose *Finish*.

[5] In SAP NetWeaver Developer Studio, the application source code and its components are housed in a "project." For a detailed introduction to SAP NetWeaver Developer Studio, and using its wizards to create applications and EJBs, see the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" in the May/June 2004 issue of *SAP Professional Journal*.

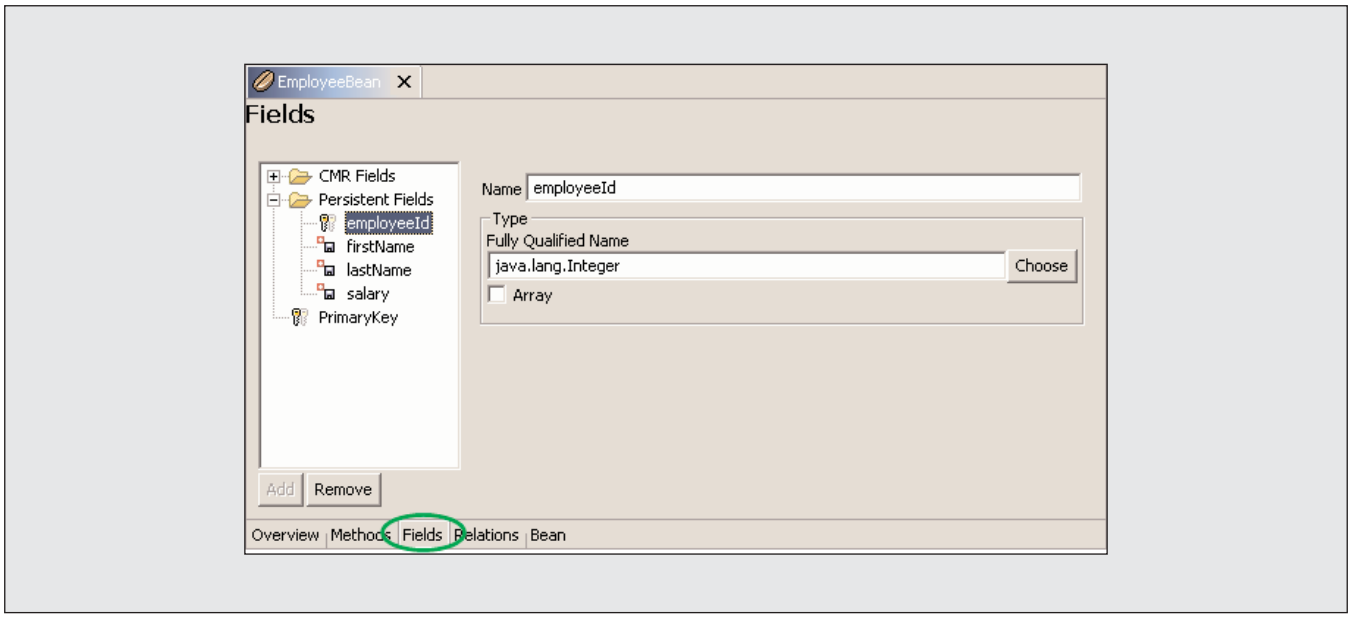
Step 2: Define Persistent Fields in the Bean Class

Bearing in mind the attributes of the persistent entity your component is to represent (refer back to Figure 5), you will be able to define the persistent fields of your component. Usually each attribute is represented by a persistent field of the CMP entity bean. Be careful with the types of the persistent fields, which must match the domain of the corresponding attributes of the persistent entity. The persistent fields of a CMP entity bean (CMP fields) are virtual fields — in the bean class they are defined by pairs of abstract get and set accessor methods, as shown in the snippet of code for the Employee bean in **Figure 7**.

> ✓ ***Tip***
>
> *Do not define persistent fields as instance variables in your bean class. Instead, use the abstract get and set methods to access and manipulate the persistent fields.*

If you use SAP NetWeaver Developer Studio, you do not need to write Java code — you simply specify the names and types of your persistent fields, and the

Figure 8 Defining Persistent Fields Using SAP NetWeaver Developer Studio



corresponding get and set accessor methods will be automatically generated. Here is the procedure:

1. From the *J2EE Explorer* pane, choose the CMP entity bean whose persistent fields you want to create.
2. Choose the *Fields* tab.
3. Add, remove, or edit the persistent fields of your CMP entity bean (see **Figure 8**).
4. Choose the persistent field that will be the primary key field of your component.

Note that the type of the *employeeId* field, which is the primary key field for the Employee bean, is the class *java.lang.Integer*. Using the Java primitive type *int* would result in an error, because the EJB specification requires that the primary key type for an entity bean be a serializable Java class.

Step 3: Define the Persistent Relationship Fields in the Bean Class

Once the persistent fields are ready, you can create relationships between your entity components. The CMP entity bean model allows the use of binary relationships; that is, relationships between two entity beans. If you want to declare such a relationship, the only thing you need to do in your bean class is define a virtual container-managed relationship (CMR) field — the code in **Figure 9** shows the

Figure 9 Defining the Relationship Between Entity Beans

```
// CMR field department
public abstract DepartmentLocal getDepartment();
public abstract void setDepartment(DepartmentLocal department);
```

CMR field defined in the Employee bean for the relationship between the Employee and Department entity beans. The return type of the get method and the parameter of the set method must be the local component interface of the CMP entity bean connected to your bean.

With SAP NetWeaver Developer Studio, choose the *Relationships* tab, where you can add, remove, and edit the container-managed relationships of your bean (see the sidebar below for more on what's happening behind the scenes).

The implementation of the get and set methods for accessing the persistent fields (both CMP and CMR fields) is done by the EJB container in a subclass of the bean class, which is generated at deployment time. By implementing the abstract get and set methods, the EJB container controls any accesses (reads or writes) to the persistent fields and thus manages the synchronization of the persistent entities. Synchronization is accomplished by the automatic loading of entities from the database the first time they are accessed by a transaction, and the automatic storing back at the end of the transaction if they have been changed.

Managing the Relationships Between CMP Entity Beans: Behind the Scenes

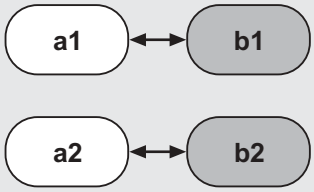
While SAP NetWeaver Developer Studio makes defining relationships quite easy and natural, we know that behind the curtain, the EJB container is doing a lot of work to make our lives easier. Here, we will explore in detail what the EJB container does in order to manage the entity bean relationships. This will help you to choose the most suitable relationships when you design and develop your CMP entity bean applications.

Multiplicity Constraints and Cascading Deletes

"Managing relationships" means adhering to several rules that comprise the concept of "referential integrity." These rules are always observed by the EJB container, no matter what type of operations the application performs on its container-managed relationships. When an operation is executed with a container-managed relationship (CMR) field, the EJB container implicitly does all the subsequent work to complete the operation and make it logically correct. A single operation can trigger an implicit update to all entities related to the modified one. The most important rules — the ones that provoke the EJB container to execute the implicit operations — are maintaining multiplicity constraints and automatic cascading deletes:

- **Multiplicity constraints:** Binary relationships used in the entity-relationship model can be one-to-one, one-to-many, and many-to-many. The same separation is used in the object model of CMP entity beans. The one-to-one and one-to-many CMP entity relationships require constraints that correspond to the foreign key constraints managed by the databases. These multiplicity constraints are maintained by assignment rules. According to these rules, assigning a CMR field triggers additional changes to the related entity objects. The goal is to preserve the multiplicity of the relationship.

  Let's suppose we have a one-to-one relationship between entity beans A and B and the entities a1 and a2 are related to the entities b1 and b2 (see the diagram to the right).

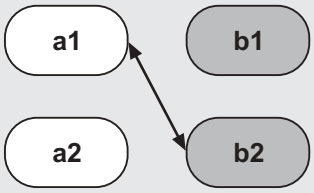


Step 4: Implement the EntityBean Interface Callback Methods

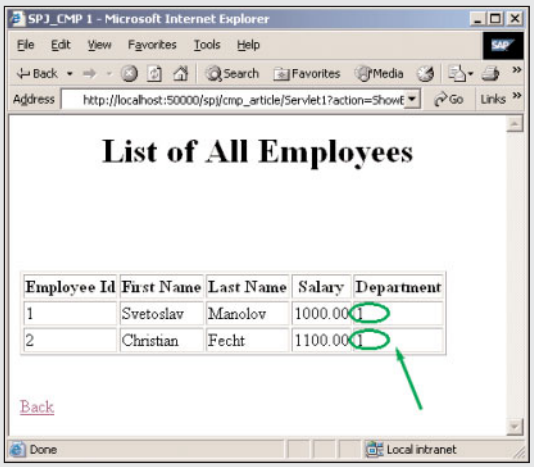
The entity bean class must also implement or inherit from its superclasses the implementation of the *EntityBean* interface methods. The *EntityBean* interface methods are callback methods that the EJB container invokes on the bean instance at specific points in time. The EJB container creates and manages bean instances according to a special component "contract," which is called "the lifecycle of the bean instance." Implementing these callback methods enables the EJB container to notify you at critical points during the lifecycle of the bean instance[6] and to perform certain operations. For example, you might have fields that are not persistent themselves, but contain data that is calculated from persistent fields. You must initialize these calculated fields using the *ejbLoad()* method, which is invoked by the EJB container just after it loads the persistent fields from the database.

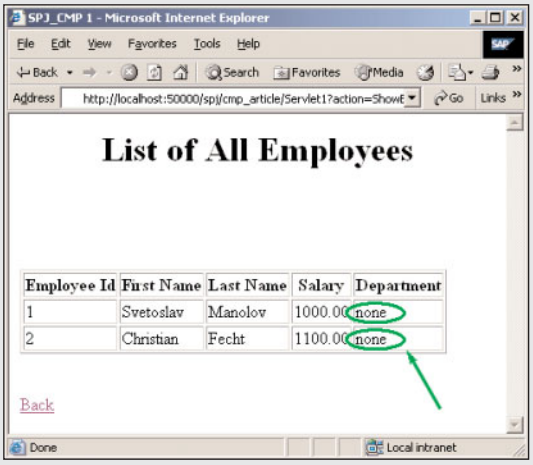
[6] Right after the state is loaded from the database; just before the state is written to the database; and just before the persistent entity is deleted.

Assigning the CMR field of a1 by the a1.setB(b2) operation also affects the other entities. That single operation results in updating the CMR fields of the four entities (see the diagram to the right).



To maintain the multiplicity constraints, there are rules not only for assignment, but also for the removal of entity objects. After removing an entity object, all entities that refer to it are implicitly affected. For example, when a department is removed, all employees working in that department are updated. Their department CMR fields are set to null, since they cannot work in a department that does not exist (see the screenshots below).





Suppose that in your use case you did not want to have an existing employee without a department. Then you would expect all employees that belong to a removed department to also be removed. In this situation, you need cascading deletes.

- **Cascading deletes:** A cascading delete removes all entity objects that refer to a removed entity

(continued on next page)

(continued from previous page)

object; the cascading delete is an automatic operation performed by the EJB container. Such functionality is useful when the lifetime of one or more entity objects depends on the lifetime of another entity object.

To switch the cascading delete option on, you need to specify the cascade-delete element, which is part of the relationship definition in the ejb-jar.xml deployment descriptor:

```
<ejb-relationship-role>
  <ejb-relationship-role-name>Employee works in Department
  </ejb-relationship-role-name>
  <multiplicity>Many</multiplicity>
  <cascade-delete/>
  <relationship-role-source>
    <ejb-name>EmployeeBean</ejb-name>
  </relationship-role-source>
  <cmr-field>
    <cmr-field-name>department</cmr-field-name>
  </cmr-field>
```

✓ ***Tip***

In most cases, EntityBean interface methods can safely be ignored using an empty implementation:

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

Define the Component and Home Interfaces for the Entity Beans

We next need to define the component interfaces and the home interfaces for the entity beans, which clients will use to access the beans.

Recall that a bean exposes its business logic to its clients through the component interface, so you can include almost any method you want in the component interface. In most cases, the component interface of a CMP entity bean will contain an assortment of get and set methods for the bean's CMP and CMR fields. The home interface typically contains methods that are needed for creating and finding CMP entity beans.

Defining the Component Interface

The main consideration in designing a CMP entity bean application is which persistent data manipulations are to be encapsulated in entity bean business methods, and which are to be moved to the session bean that accesses the entity bean (according to the session façade design pattern). You can encapsulate several accesses to the persistent data in a single business method of your entity bean. For example,

Navigating Relationships — Unidirectional vs. Bidirectional

The relationships between CMP entity beans are either bidirectional or unidirectional. Bidirectional relationships can be navigated in both directions if both CMP beans have get and set accessor methods for manipulating the relationship. Unidirectional relationships can be navigated in one direction only, that is, only one side of the relationship has get and set accessor methods. These relationships are typically used to restrict the visibility of a relationship and allow only one of the participants to manage it. The navigability of relationships is independent from the representation of the persistent entities. You can choose the navigability of a relationship according to the object-oriented design of your application.

Relationships vs. Dependent Value Objects

The EJB specification allows you to have a simple persistent field mapped to several attributes of the corresponding persistent entity. A dependent value class is a concrete class that contains persistent fields that are usually used together. For example, it is more convenient to have an encapsulated Address object containing country, town, street, etc., when the application's use cases always manipulate whole addresses rather than accessing separate fields from the address. The Address object should be defined as a dependent value class and should be accessed as a single persistent field, instead of developing a separate CMP entity bean component for it.

Figure 10 Exposing the Accessor Methods of the Persistent Fields in the Component Interface

```
public interface EmployeeLocal extends EJBLocalObject {

     public Integer getEmployeeId();

     public String getFirstName();
     public void setFirstName(String firstName);

     public String getLastName();
     public void setLastName(String lastName);

     public BigDecimal getSalary();
     public void setSalary(BigDecimal salary);

     public DepartmentLocal getDepartment();
     public void setDepartment(DepartmentLocal department);
}
```

you can define an *increaseSalary* business method that first reads the employee's salary, and then updates it.

In our example (see **Figure 10**), we choose to expose only the accessor methods of the persistent fields in the component interface and to manipulate

Figure 11 Initializing the State of an Entity Object

```
public EmployeeLocal create(
     int empId,
     String firstName,
     String lastName,
     BigDecimal salary,
     DepartmentLocal department)
     throws CreateException;
```

the persistent data in those fields via fine-grained calls from the session bean.

> ✓ ***Tip***
>
> *Do not forget that methods declared in the component interface are related to a single entity.*

> ✓ ***Tip***
>
> *You cannot expose the set accessor methods for the primary key CMP fields in the component interface of the entity bean. Once initialized, the primary key for an entity bean cannot be changed.*

You're probably wondering what will happen to the application's performance as the session bean, implementing the application's business logic, executes multiple method calls for retrieving the persistent attributes of a single employee. Don't worry — using local interfaces for your CMP entity beans makes the business method calls very efficient.

Defining the Home Interface

The bean's local home interface serves as a "factory" for entity objects of a concrete type. For example, the *EmployeeLocalHome* interface defines methods for the client to create, remove, and find Employee entity objects (we'll look at these methods in action later in the article):

- **Create methods:** Create methods are used to create new CMP entity bean objects and their persistent representations; that is, they insert new records in the data store (the relational database). The return type of a create method is the entity bean's component interface. At runtime, the result of a create method is an *EJBLocalObject* representing the newly created persistent entity. The arguments of the create methods are typically used to initialize the state of the created persistent entity, as shown in **Figure 11**.
- **Finder methods:** Finder methods are used by clients to locate entity objects (see **Figure 12**). They search persistent entities that fulfill a certain criterion. Their return type is either the entity bean's component interface or a *java.util.Collection* object. At runtime, they return either a single *EJBLocalObject* or a collection of *EJBLocalObjects* that represent the found persistent entities.
- **Remove methods:** Clients can remove a CMP entity bean and its persistent representation by calling the remove method of the local home interface and passing the identity object of the entity they want to remove, as shown in **Figure 13**. As a result, the corresponding database record will be removed from the persistent data store.

Figure 12 Locating an Entity Object

```
public EmployeeLocal findByPrimaryKey(Integer primKey)
     throws FinderException;

public Collection findAllEmployees() throws FinderException;

public Collection findEmployeesByMinSalary(BigDecimal minSalary)
     throws FinderException;
```

Figure 13 Removing an Entity Object

```
void remove(Object primaryKey) throws RemoveException, EJBException;
```

> ✓ ***Tip***
>
> *There is no need to define a remove method in your home interface. The remove method is defined in the javax.ejb.EJBLocalHome interface, which is a superinterface for all local home interfaces.*

In the local home interface of your CMP entity bean, you can also define methods that supply business logic that is not specific to a single CMP entity bean. This is useful when you want to have business methods that execute queries involving more than one entity. These methods are called "home methods," and usually deal with a group of entities rather than a single entity. The home methods can also serve as utility methods; that is, they may provide auxiliary functionality other than manipulation of the persistent data.

We're now ready with the Java code of our CMP entity beans.

Edit the XML Deployment Descriptors

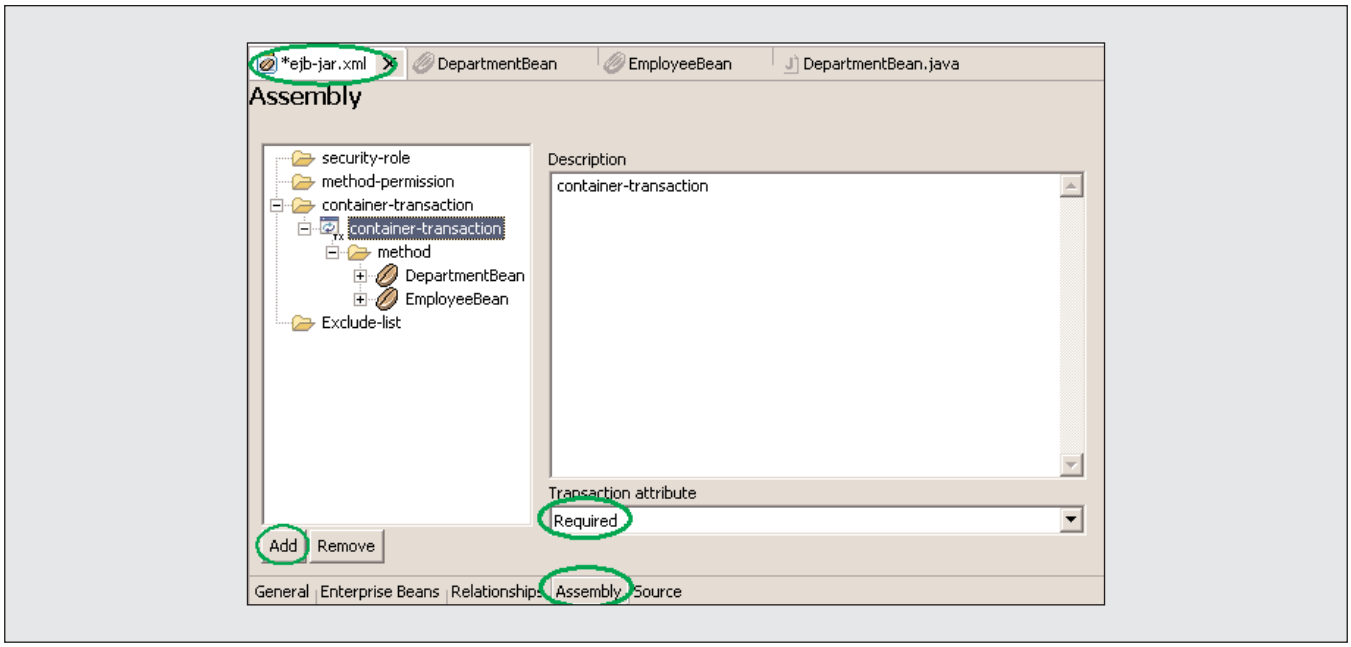
We next need to describe the CMP entity beans in XML deployment descriptors (the standard *ejb-jar.xml* file and the SAP-specific *persistent.xml* file), which will be used to assemble the application.

The Standard Deployment Descriptor (ejb-jar.xml)

The *ejb-jar.xml* file contains two basic types of information. The first describes the structure of the beans; the second describes how those beans are assembled into the application. The metadata describing the structure of the CMP entity beans is automatically generated by SAP NetWeaver Developer Studio. When you use the SAP NetWeaver Developer Studio wizards to define the name of your bean, its interfaces and bean class, its primary key, and its CMP and CMR fields, it automatically generates not only the Java code, but the XML descriptor as well.[7]

[7] The XML source in a deployment descriptor file can be changed manually. The download available at **www.SAPpro.com** explains the semantics of the essential *ejb-jar.xml* elements in detail.

Figure 14 Defining Transactional Support for the Entity Beans



The next step is to define transactional support for the CMP entity beans, which is the most important metadata needed for assembling the beans into an application.

Entity beans always use container-managed transaction (CMT) demarcation. That is, the EJB container starts and completes all transactions, and the bean provider (the application developer) just chooses the most suitable behavior for each business method. The *container-transaction* element from the *ejb-jar.xml* file specifies how the container must manage the transaction scopes for the bean methods. You can choose a transaction attribute and a list of methods to which this attribute will be applied. You can also choose a "whole bean," which is a convenient way of selecting all of its business methods. This is what we do with *DepartmentBean* and *EmployeeBean*. In SAP NetWeaver Developer Studio:

1. Open the *ejb-jar.xml* editor (**Figure 14**).

2. From the right-hand pane, choose the *Assembly* tab.

3. Select the *container-transaction* node.

4. Choose *Add*. From the dialog box that appears, choose the enterprise beans for which you want to specify a transaction attribute.

The transaction attribute for the selected beans and methods can be chosen from the dropdown list at the lower right of Figure 14. Since entity beans deal with persistent data from a transactional data store, their business methods should be executed in a transactional context. For such purposes, *Required* is the most useful transaction attribute, and it means that the methods require execution in a transaction context. If the caller has such a context, the EJB container propagates this context to the invoked method. Otherwise, the EJB container automatically starts a new transaction prior to delegating the call to the actual implementation in the bean instance.

The SAP-Specific Object-Relational Mapping Descriptor (persistent.xml)

Our example relational database schema consists of two database tables: TMP_EMPLOYEE, represented by *EmployeeBean*, and TMP_DEPARTMENT, represented by *DepartmentBean* (see **Figure 15**).

Figure 15 The Database Tables of the Example Relational Database Schema

Field Name	Type	Primary Key	Nullable	Foreign Key
TMP_DEPARTMENT				
DEPID	INTEGER	Yes	No	No
NAME	CHAR(30)	No	No	No
TMP_ EMPLOYEE				
EMPID	INTEGER	Yes	No	No
FIRSTNAME	VARCHAR(30)	No	No	No
LASTNAME	VARCHAR(30)	No	No	No
SALARY	DECIMAL(10, 2)	No	No	No
DEPID	INTEGER	No	Yes	Yes

Figure 16 Specifying the Object-Relational Mapping

```
<ejb-name>EmployeeBean</ejb-name>
<table-name>TMP_EMPLOYEE</table-name>
<field-map
    key-type="PrimaryKey">
    <field-name>employeeId</field-name>
    <column>
        <column-name>EMPID</column-name>
    </column>
</field-map>
<field-map
    key-type="NoKey">
    <field-name>firstName</field-name>
    <column>
        <column-name>FIRSTNAME</column-name>
    </column>
</field-map>
<field-map
    key-type="NoKey">
    <field-name>lastName</field-name>
    <column>
        <column-name>LASTNAME</column-name>
    </column>
</field-map>
<field-map
    key-type="NoKey">
    <field-name>salary</field-name>
    <column>
        <column-name>SALARY</column-name>
    </column>
</field-map>
```

When assembling your CMP entity beans in an application, you must specify their object-relational mapping, as shown in **Figure 16** for the Employee entity bean. This information is stored in the

SAP-specific object-relational mapping descriptor *persistent.xml*. This descriptor contains all the information needed by the EJB container to manage the automatic synchronization of your persistent data: the mapping between entity beans and database tables, between persistent fields and database columns, and between entity relationships and database table relationships.

The object-relational mapping can easily be created using the SAP NetWeaver Developer Studio *persistent.xml* editor. This editor allows you to browse the database tables already defined in data dictionary projects and to map your entity beans and their persistent fields to the corresponding database tables and columns.[8]

Create a Session Bean That Will Access the Entity Beans

Before an application can work with an entity bean, it first needs to get a reference to the entity bean's component interface via its home interface. According to the session façade design pattern, entity beans should be accessed solely through a session bean. Here, we'll examine what a session façade bean for our *EmployeeBean* and *DepartmentBean* entity beans could look like, and what has to be done to use the entity beans from within it. Let's assume that we've followed the previously outlined steps and created a stateless session bean *HRSesBean* that provides (through its own home/component interfaces) all the functionality a client needs to work with employees and departments: creating new employees and new departments, listing all employees, deleting an employee, moving an employee from one department to another, and so on. The session bean *HRSesBean* is part of the same J2EE enterprise application as the *EmployeeBean* and the *DepartmentBean* entity beans.

[8] For further discussion of this topic, see the article "A Guided Tour of the SAP Java Persistence Framework — Achieving Scalable Persistence for Your Java Applications" in the May/June 2004 issue of *SAP Professional Journal*.

The session bean needs to declare that it's going to make use of the two entity beans. Since it will access the entity beans through their local home/component interfaces, the session bean has to declare local EJB references to both *EmployeeBean* and *DepartmentBean*.

> ✓ ***Tip***
>
> *When you define the session bean façade, don't forget to declare local EJB references to all the entity beans you are going to use from within the session bean. It's good practice to prefix the locally declared JNDI names for the entity beans with "ejb/".*

Technically, the reference is declared in *ejb-jar.xml* by including <*ejb-local-ref*> declarations in the declaration of *HRSesBean* (see **Figure 17**).

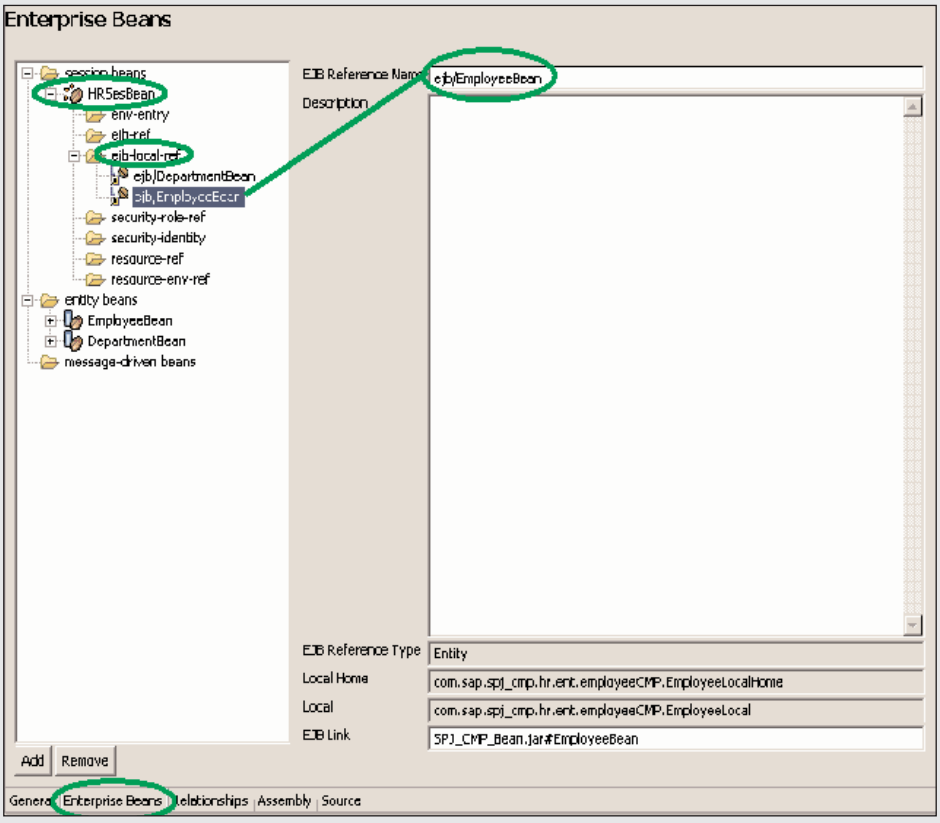
Of course, you don't have to edit the *ejb-jar.xml* file manually in order to declare a reference to another bean. The *ejb-jar.xml* editor in SAP NetWeaver Developer Studio provides a convenient way to declare such references. Just add a new *ejb-local-ref* entry to *HRSesBean*, as shown in **Figure 18**. From the dialog showing all available beans, choose *EmployeeBean* and *DepartmentBean*, and declare the local name of the reference as *ejb/EmployeeBean* and *ejb/DepartmentBean*, respectively.

What is the purpose of declaring a local EJB reference *ejb/EmployeeBean*? Remember that the session bean *HRSesBean* needs to get a reference to the local home interface of the *EmployeeBean* before it can access its local component interface and start working with it. Recall further that each bean has its own local JNDI context. By declaring a local EJB reference to *EmployeeBean* with the name *ejb/EmployeeBean*, a reference to the local home interface of the entity bean *EmployeeBean* is put into the JNDI context *java:comp/env* under the full name

Figure 17 Declaring Local EJB References to the Entity Beans in ejb-jar.xml

```
<enterprise-beans>
  <session>
    <ejb-name>HRSesBean</ejb-name>
    …
    <ejb-local-ref>
      <ejb-ref-name>ejb/EmployeeBean</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <local-home>package.EmployeeLocalHome</local-home>
      <local>package.EmployeeLocal</local>
      <ejb-link>EmployeeBean</ejb-link>
    </ejb-local-ref>
    …
  </session>
```

Figure 18 Declaring a Local Bean Reference Using SAP NetWeaver Developer Studio



java:comp/env/ejb/EmployeeBean. The bottom line is that the session bean *HRSesBean* can simply look up the entity bean *EmployeeBean* under the name *java:comp/env/ejb/EmployeeBean*.

Figure 19 Starting the Bean Class for the Session Bean

```
public class HRSesBean implements SessionBean {
     private EmployeeLocalHome employeeHome;
     private DepartmentLocalHome departmentHome;
…
}
```

Figure 20 Implementing the setSessionContext Method

```
public void setSessionContext(SessionContext context) {
   try {
     Context ctx = new InitialContext();
     employeeHome = (EmployeeLocalHome)
                     ctx.lookup(java:comp/env/ejb/EmployeeBean");
   } catch (NamingException ex) {
      throw new EJBException("Could not find
                              java:comp/env/ejb/EmployeeBean");
   }
   …
}
```

The session bean *HRSesBean* should acquire the references to the home interfaces of *EmployeeBean* and *DepartmentBean* in the lifecycle method *setSessionContext* and store them for later use in private attributes. The lifecycle method *setSessionContext* is invoked automatically by the EJB container at the very beginning of a bean's life. It is invoked immediately after the bean instance has been instantiated, and allows the bean to acquire resources that it needs for the rest of its bean life and that are independent from the client the bean will be serving. The bean class for *HRSesBean* will typically start as shown in **Figure 19**.

The class *HRSesBean* has to implement the interface *SessionBean* that declares (among other methods) the lifecycle method *setSessionContext*. Furthermore, it declares two private fields *employeeHome* and *departmentHome* that hold references to the home interface of *EmployeeBean* and *DepartmentBean*, respectively. The implementation of the *setSessionContext* method is shown in **Figure 20**. It looks up the *EmployeeBean*'s home in the session bean's local JNDI context and stores a reference to it in the instance field *employeeHome*.

Our session bean *HRSesBean* is ready to work with the entity beans in one of its business methods. Since the business methods access a transactional data store through the entity beans, the business methods should be executed in a well-defined transactional context. This is best achieved by simply using container-managed transactions (CMTs) for the session bean *HRSesBean* and declaring all business methods to have the transaction attribute *Required* (see Figure 14). This can be readily done with the SAP NetWeaver Developer Studio *ejb-jar.xml* editor. The advantage of required CMTs is that the EJB container will ensure that the business method always executes within the context of an active server-controlled transaction.

With the entity and session beans created, and the necessary edits made to the XML deployment descriptors, we can now assemble the J2EE application using the SAP NetWeaver Developer Studio deployment wizards. For details on the specific steps involved, see the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" in the May/June 2004 issue of *SAP Professional Journal*. Over the remaining sections of this article, we will look at what happens behind the scenes when a client accesses our example application and our previous development work springs into action.

Manipulating Entity Beans and Their Persistent Entities

When a client accesses our HR session bean — for example, to hire a new employee — the session bean in turn uses the methods in the *DepartmentBean* and *EmployeeBean* entity bean component and home interfaces to manipulate the underlying persistent entities in the relational database — i.e., to create, update, and delete persistent entities and to search for entities that fulfill a given search criterion.

Creating New Entity Beans and Their Persistent Entities

The session bean creates an entity bean — together with its corresponding persistent entity in the data store — by calling one of the create methods in the bean's home interface, which returns a reference to the bean's component interface. The parameters passed to the create method are typically used to initialize the newly created entity bean (see **Figure 21**).

DepartmentBean has one create method that takes the ID and the name of the department to be created as parameters and returns a reference to the component interface of the newly created department. In the example in Figure 21, we first create a new department with an ID of *0* and the name *R&D*. We then create a new employee who works in this department. The create method for employees takes several parameters: the employee's ID (*0*), first name (*John*), last name (*Smith*), salary (variable *salary*), and department (variable *department*). Note that the persistent representations of the newly created department and of the newly created employee are not immediately created in the data store. The actual creation of the persistent objects in the data store is delayed until the commit time of the current transaction.

Let's take a moment and look in more detail at what happens when a bean is created. Recall that a bean consists of the bean instance and the EJB entity object that is inseparably associated with an object that represents the bean's object identity. Upon invocation of a create method in the home interface, the EJB container either takes a bean instance from the pool or instantiates a new bean instance, and then invokes the corresponding *ejbCreate* method on that bean instance. The purpose of the *ejbCreate* method is to initialize the CMP fields of the bean, and to (at least) initialize the primary key CMP fields. For each create method in the bean's home interface, there must be corresponding *ejbCreate* and *ejbPostCreate* methods in the bean class (that is, methods that have

Figure 21 Initializing the Newly Created Entity Bean

```
DepartmentLocal department = departmentHome.create(0, "R&D");
EmployeeLocal employee     = employeeHome.create( 0,
                                                  "John", "Smith",
                                                  salary,
                                                  department);
```

Figure 22 The ejbCreate Method

```
public Integer
         ejbCreate(
           int empId,
           String firstName,
           String lastName,
           BigDecimal salary,
           DepartmentLocal
               department)
             throws
               CreateException {

     // Initialize the CMP fields
     setEmployeeId(new
       Integer(empId));
     setFirstName(firstName);
     setLastName(lastName);
     setSalary(salary);
     return null;
}
```

the same parameter signature). These *ejbCreate* and *ejbPostCreate* methods have to be implemented by the bean provider (i.e., the application developer). In **Figure 22**, you see the *ejbCreate* method from *EmployeeBean*, which initializes the CMP fields of *EmployeeBean* by calling the abstract set methods.

> ✓ ***Tip***
>
> *Always return null in ejbCreate methods.*

After the *ejbCreate* method has been invoked, the EJB container can retrieve the values of the primary key CMP fields, construct the identity object, and associate it with the bean's EJB object. It should be clear by now that during the execution of *ejbCreate*, no identity is available, so the *ejbCreate* methods should only be used for the initialization of CMP fields; CMP fields, as opposed to CMR fields, don't rely on the object identity. CMR fields have to be initialized in *ejbPostCreate* methods. The *ejbPostCreate* method is called after the corresponding *ejbCreate* method has been called. The identity of the bean is available within the *ejbPostCreate* method. The code in **Figure 23** shows the *ejbPostCreate* method of *EmployeeBean*. *EmployeeBean* has only one CMR field (*department*), which is initialized by calling its abstract set method *setDepartment*.

Figure 23 The ejbPostCreate Method

```
public void ejbPostCreate(
     int empId,
     String firstName,
     String lastName,
     BigDecimal salary,
     DepartmentLocal department)
     throws CreateException {

     // Initialize the CMR field
        department
     setDepartment(department);
}
```

> ✓ ***Tip***
>
> *Never initialize CMR fields in ejbCreate methods. Always initialize them in ejbPostCreate methods.*

Updating Entity Beans and Their Persistent Entities

The session bean updates an entity bean by calling the abstract set methods of its CMP and CMR fields. In the code in **Figure 24**, we change the last name of an employee to *Miller* and change the department she works in to some "other department."

Updates are actually triggered by clients invoking methods in the bean's component interface. All updates performed on entity beans are automatically tracked by the EJB container and written back to the database at commit time.

Figure 24 Updating an Entity Bean

```
employee.setLastName("Miller");
employee.setDepartment(
  otherDepartment);
```

Deleting Entity Beans and Their Persistent Entities

There are two ways the session bean can delete an entity bean (see **Figure 25**):

- By invoking the remove method on the bean's component interface
- By passing a reference to the bean's identity object to the remove method in the bean's home interface

The persistent entity is not removed from the database until the transaction is committed. Before the persistent entity is removed, the container invokes the *ejbRemove* method on the bean instance to notify the bean about the forthcoming removal. Thus the bean can perform additional operations such as updating other beans. For beans that are in a relationship, one can declaratively specify "cascading deletes," so that all dependent objects are also deleted (see the sidebar on pages 150-153).

Figure 25 Deleting an Entity Bean

```
employee.remove();

// or alternatively:

employeeHome.remove(employeePKObj);
```

Finding Entity Beans and Their Persistent Entities

The session bean finds existing entity beans in the data store that meet a given search criterion using query methods. There are two kinds of query methods: finder methods and select methods. The main difference between these two is that finder methods are exposed to clients, whereas select methods can only be used internally in the bean class. For the sake of brevity, we will focus on finder methods.[9]

Finder methods are exposed to clients through the bean's home interface, as shown in **Figure 26**.

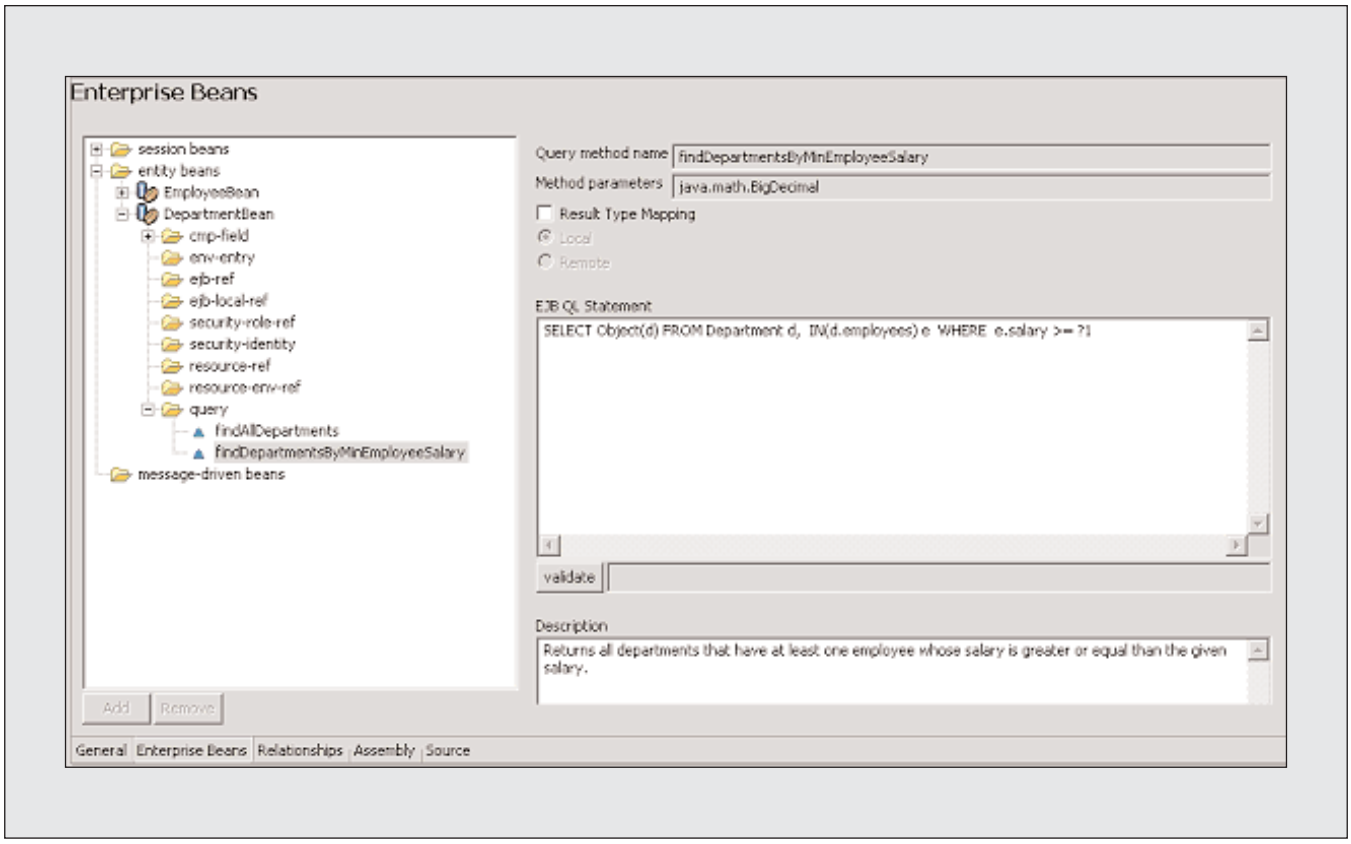
We declare some finder methods in the local home interface of *EmployeeBean*. A finder method has to specify the exception *javax.ejb.FinderException* in its *throws* clause. It can take parameters and it basically returns a collection of entity beans, or, to be more precise, a collection of references to the beans'

[9] You can find more on select methods in the NetWeaver documentation and in many EJB books.

Figure 26 Using Finder Methods

```
public interface EmployeeLocalHome extend EJBLocalHome {
…
  EmployeeLocal findByPrimaryKey(Object pk)
                                  throws FinderException;
  Collection    findAllEmployees()  throws FinderException;
  Collection    findEmployeesByLastName(String lastName)
                                  throws FinderException;
  Collection    findEmployeesByDepartment(String depName)
                                  throws FinderException;
}
```

Figure 27 The EJBQL Query for a Finder Method



component interfaces. Note that a finder method can only return beans of its own kind — a finder method in the employee home can only return employees; it cannot return department beans or strings.

Finder methods that are expected to find more than one bean are called "multi-object finders" (for example, *findEmployeesByLastName*) and must have *java.util.Collection* as the return type. Finder methods that will return at most one bean are called "single-object finders" (for example, *findByPrimaryKey*) and can simply use the bean's component interface as the return type.

Finder methods are declared in the bean's home interface. But where and how are their semantics (that is, the actual search conditions) specified and implemented? Apart from putting the finder declaration into the home interface, the bean provider (the application developer) has to declare the finder in the

✓ ***Note!***

Every entity bean has to declare at least the special single-object finder findByPrimaryKey in its home interface. This finder takes a reference to a primary key object as a parameter and returns the corresponding bean. Method findByPrimaryKey is implemented automatically by the EJB container.

ejb-jar.xml deployment descriptor, where the name and the signature of the finder are declared. The finder's semantics are declaratively specified by means of the EJB Query Language (EJBQL). That's all the bean provider has to do.

> ✓ ***Note!***
>
> *Don't put anything about finders in your bean class code.*

During deployment, the EJB container uses the home interfaces to figure out what the finders are and to relate them to the finder declarations in the deployment descriptor. Given the finder declarations together with the EJBQL query, the EJB container then generates the complete implementation of the finder method, including all necessary database access code. You can easily declare finder methods with the SAP NetWeaver Developer Studio EJB editor. The actual EJBQL query for a finder method is specified in the *ejb-jar.xml* editor (see **Figure 27**).

EJBQL allows you to write queries for your finders in a portable and SQL-like style. With EJBQL, you don't have to know anything about the real database where your beans will finally run. For the search condition, you simply use your object model. During deployment, the EJB container will generate the real database access code for your queries. In our case, with a relational database, the EJBQL queries are translated by the container into semantically equivalent SQL queries. Let's take a look at a simple EJBQL query, shown in **Figure 28**. This query returns all employees whose salary is greater than or equal to a specific salary that is given as the value of the parameter *?1*.

Figure 28 A Simple EJBQL Query

```
SELECT Object(e) FROM Employee e
  WHERE e.salary >= ?1
```

As you can see in Figure 28, a query consists of three parts:

- The **SELECT clause**, which specifies the kinds of things the query will return. *SELECT Object(e)* says that the query will return *e* beans, whatever *e* is. Formally, *e* is called an "identification variable."
- The **FROM clause**, in which the domain of the query is declared. Here, we tell the container where we want it to search for beans. This is done by declaring the identification variables that are used in the SELECT clause. *FROM Employee e* simply says that we want to search for *EmployeeBean* beans.
- The **WHERE clause**, which is optional, lets you define extra search criteria for the beans you are looking for. By using *WHERE e.salary >= ?1*, you are saying that you are not interested in all employees, only in those whose salary fulfills the given condition.

A query can have parameters (*?1*, *?2*, *?3*, etc.) that correspond to the parameters of the finder method. When a finder method is invoked, the underlying query is executed with its parameters bound to the current values of the corresponding parameters of the finder.

> ✓ ***Note!***
>
> *You may have observed that we don't use the actual bean name EmployeeBean in the EJBQL query in Figure 28. Instead we use Employee to refer to EmployeeBean. Each CMP entity bean has a second name, its abstract schema name, which you have to use in EJBQL queries to refer to the bean. The abstract schema name is declared in the ejb-jar.xml deployment descriptor as part of the bean's declaration.*

In the WHERE clause *WHERE e.salary >?= ?1* in Figure 28, we use the dot notation *e.salary* in the

Figure 29 Navigating to a Related Bean

```
SELECT Object(e)
  FROM Employee e, Department d
  WHERE e.department.name >= ?1
```

Figure 30 Using the IN Operator

```
SELECT Object(d)
  FROM Department d,
    IN(d.employees) e
  WHERE e.salary >= ?1
```

query to navigate to and access the CMP field *salary* of the *EmployeeBean*, denoted by *e*. With the dot notation, you can make up paths and navigate in an object-oriented way. In the example in **Figure 29**, we even navigate to another related bean. The single-valued CMR field *department* is used to navigate from *Employee to Department*.

The path used in navigation must end up in a CMP field. You can only navigate from one bean to another through single-valued CMR fields. In the case of collection-valued CMR fields, you have to use the *IN* operator, which lets you declare a variable that ranges over the elements of the collection. With the query in **Figure 30**, we search for all departments that have at least one employee whose salary is greater than or equal to a given salary.

By using the *IN* operator we declare the identification variable *e* that ranges over the elements of the collection *d.employees*.

Conclusion

This article has aimed to provide you with a general introduction to CMP entity beans and the transparent object persistence provided by the SAP Web AS 6.40 EJB container. By following the steps for developing CMP entity beans, getting acquainted with the EJB development tools in SAP NetWeaver Developer Studio, and looking behind the curtain to see entity beans in action, you have glimpsed the powerful world of CMP entity beans. We hope we have given you the knowledge and the confidence to delve more deeply into this area and to discover some of the more interesting features of CMP entity beans — they will prove to be of great help to you in lots of situations by reducing your custom-coding workload. Try it for yourself and see!

Christian Fecht studied computer science at the University of Saarland in Saarbruecken, Germany, where he received his Ph.D. in 1997. He joined SAP in 1998 as a member of the Business Programming Languages group, where he was responsible for the ABAP runtime environment, in particular the ABAP Objects garbage collector. Christian is currently a member of the Java Server Technology group, where he works in the area of Java persistence, with a special focus on object persistence. He can be reached at christian.fecht@sap.com.

Svetoslav Manolov graduated with a degree in computer science from Sofia University, Bulgaria, after completing a master's degree thesis on persistence management. He joined SAP Labs Bulgaria when it was founded in 2000. Svetoslav is a member of the Java Server Technology group, where he works in the area of EJB and Java persistence, and currently leads the EJB team. He can be reached at svetoslav.manolov@sap.com.

