PRAMATI

The Promise of Perfect Persistence from EJB 2.0

Enterprise JavaBeans ™ (EJB) is a component-based architecture for modeling business entities and implementing business rules as distributed objects, and is a part of the Java™ 2 Platform, Enterprise Edition (J2EE™). The architecture abstracts the user from the complexities of building scalable, transactional and distributed applications by providing a suite of common services. Such services include transactions, persistence, security and distribution.

EJB brings together the benefits of component-based server side development with role-based development. Therefore, it was no surprise that when EJB standards were introduced by Sun Microsystems in 1998, the industry and developer community immediately accepted them. In the last two years, EJB has matured as a business layer framework with each new release. EJB 2.0 was released on September 17, 2001 and is a significant improvement over the earlier EJB 1.1, with emphasis on a re-architected Container Managed Persistence model that takes application designers closer to real-world entity relationships.

How EJB 2.0 removes bottlenecks in EJB 1.1

Let us look at the shortcomings of the EJB 1.1 standard and the solution provided by EJB 2.0 to solve these issues.

Container Managed Persistence (CMP)

The CMP model in EJB 1.1 had performance issues. The standards mandated that the bean state be loaded and stored using the `ejbLoad()` and `ejbstore()` callbacks before the start and end of a transaction by the EJB Container. This led to expensive database hits that negated the advantages of data caching.

Restrictions on the type of entity bean instance variables (primitive, serializable, home and remote) made modeling complex business data models cumbersome and difficult.

The EJB 1.1 CMP model was suitable only for coarse-grained entity beans. Since entity beans could only be exposed as remote objects, fine- grained persistence models were inefficient. There was no standard to specify queries for finder<methods> that made porting between EJB Containers an arduous task.

Solution: EJB 2.0 specifies Container Managed Relationships that allow the creation of fine-grained entity beans that can have relationship capabilities.

Asynchronous Processing

Asynchronous Processing leads to higher throughput and is an essential attribute of a Framework. EJB 1.1 did not provide integrated support for asynchronous processing. Though JMS was available as a separate standard that enabled beans to send JMS messages, receipt of messages was unreliable. Since bean methods could only be invoked as callback from the EJB Container, they could not make any blocked calls and thus both forms of method receipts were illegal.

Solution: EJB 2.0 specifies Message Driven Bean (MDB). MDBs are components that are triggered by messages and execute business logic asynchronously.

Interoperability

EJB 1.1 did not mandate Interoperability as a requirement for the EJB Container. Thus applications deployed on EJB Containers from different vendors could not collaborate.

Solution: EJB 2.0 model specifies the *Container Managed Persistence Model* which enhances support to complex business models, local beans and standard queries. Asynchronous processing is enabled through MDBs, which integrate Java Messaging Service with EJB. The new standard also specifies interoperability requirements for transactions, security, naming and invocation.

Powerful persistence model for Entity Beans

EJB 2.0 introduces the concept of "abstract persistence schema" for defining CMP in entity beans. Container Managed Field (CMF) is not *explicitly defined,* instead *implied* in virtual fields through a set of abstract accessory methods.

Below is a code snippet for an EJB 1.1 CMP entity bean. This bean has three container-managed fields.

The following example illustrates this.

```
public class AccountBean implements EntityBean {
   // Container managed fields
   public long account_number;
   public java.lang.String customer_name;
   public double balance;
   // Business Methods
   public void credit ( double amount ) {
         balance += amount;
   }
}
```

The same bean written to EJB 2.0 specifications would look like this:

```
public abstract class AccountBean implements EntityBean {
   // Virtual Container managed Fields
   public abstract long getAccount_number();
   public abstract  void  setAccount_number(long account_number);

   public abstract java.lang.String getCustomer_name();
   public abstract void  setCustomer_name(String customer_name);

   public abstract double getBalance();
   public abstract void  setBalance(double  balance);
         // Business Method
         public void credit ( double amount ) {
         double balance = getBalance();
         balance += amount;
         setBalance(balance);
   }
```

```
}
```

Each CMF has an abstract 'getter' and 'setter' method to imply the field. In the EJB 2.0 implementation, instead of directly updating the balance, the implementation retrieves the balance using the 'getter' and then resets the balance using the 'setter'. The Bean class itself is abstract and the EJB Container provides the implementation of the abstract class.

Since the EJB container provides the implementation for the 'getter' method, it gets a callback when the bean is retrieving a CMF. It can then '**Lazy Load**' the field if required. The real benefit comes from the container's implementation of the 'setter' method. For example, the user would not want the image of an employee loaded whenever the employee record is accessed. The image can be **lazily loaded** when required.

It provides '**Dirty Detection**' of the bean state that has been changed by any business function. The EJB 2.0 standard also relaxes the requirement for calling `ejbLoad()` and `ejbStore()`.

Combining the two, we get a container capable of detecting a change in state and hitting the database only when required. Since most operations are read operations, this provides a significant performance boost, as there are no trips to the database if the data is cached. The container can choose to delay the loading of CMFs till they are actually accessed.

Local Beans

Before EJB 2.0, all beans available across VMs were distributed. High cost of distribution has always been associated with the Bean, since they need not be accessed directly across VM, yet they have to be exported and distributed. To minimize this distribution cost, the beans were modeled as coarse-grained objects. EJB 2.0 introduces Local Beans using which the developer can specify them as Local and incur no distribution cost on them. To better understand this concept, here is a simple design problem.

Design Options with EJB 1.1 Model

Business Model contains `Order` Entity Bean and `ShippingAddress` Entity Bean.

This option involves unnecessary cost of distribution for `ShippingAddress`. It also exposes `ShippingAddress` directly to the Client view, which is not a good design. In case the container does not support intra VM call optimizations, the interaction between `Order` and `ShippingAddress` will be through the sockets, thus making the whole process very expensive.

Business Model contains `Order` as an Entity Bean and `ShippingAddress` as a Serialized Object

This is a coarse-grained design where the `Order` aggregates the `ShippingAddress`. This would eliminate the problems in the above design, but serialization and de-serialization would be an expense for the `ShippingAddress` Object. Further more, since the object will be stored in a serialized form, `ShippingAddress` specific queries will not be possible.

Design Options with EJB 2.0 Model

Model `Order` as an Entity Bean and `ShippingAddress` as a Local Entity Bean

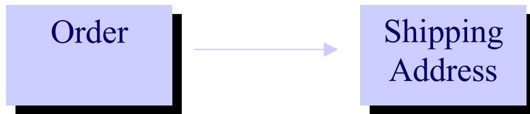

Figure 1: Modeling Order and Shipping Address as an Entity Bean

This would solve both set of problems plaguing the above two models. Local Beans are identical to Remote Beans with the only difference that they are available on the VM where they are co-located. There is no cost of distribution. Local Beans allows for fine- grained persistence modeling without any overheads.

Container Managed Relationships (CMR)

CMRs enable the framework to define complex business models. As mentioned earlier, representing complex business models as entity beans was difficult and cumbersome in the EJB 1.1 era. To explain CMRs, let us extend the business problem a bit further. This is a very common business model comprising of an `Order`, `ShippingAddress`, `LineItem` and `Product`. Let us again explore the design options in EJB 1.1 and EJB 2.0.

Relationships can be unidirectional or bi-directional and can have different cardinalities such as One to One, One to Many, Many to One and Many to Many.

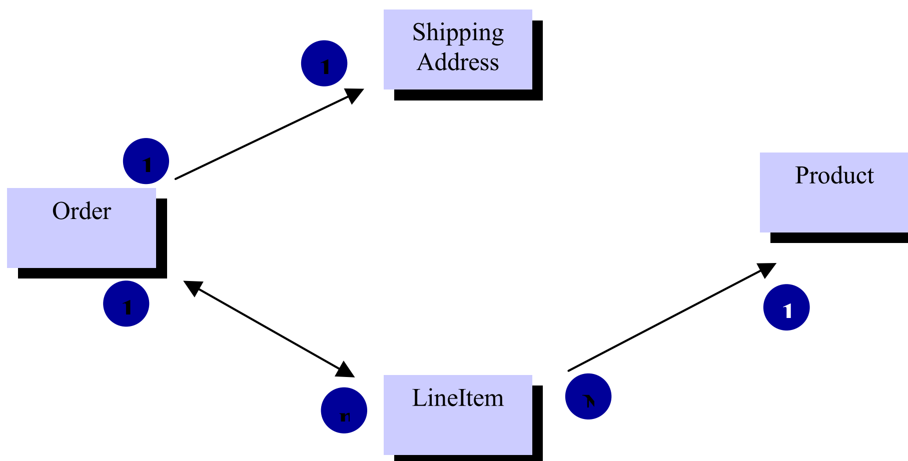

Figure 2: Entity-Relationship diagram, showing cardinality.

Design Option in EJB 1.1

The only option in EJB 1.1 is to use bean managed persistence instead of CMPs. This is because EJB 1.1 beans can have only primitive types, serializable, home and remote as CMFs. There was no standard support for collections, though some vendors provide proprietary extensions. Thus, there is no design option using CMP in EJB 1.1.

Design Option in EJB 2.0

EJB 2.0 extends CMFs to relationships. CMRs, like fields, are defined using a set of Abstract Accessor methods. The return type of the 'getter' method is a local interface of the related bean or a *Collection* interface in case the cardinality is more than one. Similarly, the argument to the 'setter' method is either a Local Interface or a Collection.

EJB Query Language (EJB QL)

For all those who have worked on projects involving porting EJB components from one server to another would know that porting finder queries is a virtual minefield. Although EJB components are portable across servers, since the EJB 1.1 specifications did not standardize the syntax for specifying the queries, EJB container vendors evolved their own representations. EJB 2.0 standardizes these queries with EJB QL, a specification language that defines queries using the CMP Data Model (abstract persistence schema). EJB QL is compiled to a target persistence store language, such as SQL, during deployment.

Why EJB QL?

SQL definition is based on the relational schema whereas the EJB QL is based on the Object Schema. Consider an SQL query pertaining to the examples we have discussed.

SQL Query to find pending orders:

```
SELECT order_id FROM order AS o.lineitems AS l WHERE o.order_id = l.order_id AND
l.shipped = 'FALSE'
```

The corresponding EJB QL on the `OrderBean` will look this:

```
FROM OrderBean o, l IN o.lineItems WHERE l.shipped = 'FALSE'
```

In EJB QL, queries are specified as objects. The deployment environment determines the persistence schema. It could be a relational database or even an object database. Defining queries using persistence store format would again make the queries non-portable. EJB QL, as is evident from the example, is syntactically similar to SQL and fairly simple.

ejbSelect Methods

Finder queries can return only the Remote type or a Collection of Remote Objects of that Bean due to the limitation of their return type. This limitation is removed by another feature of the EJB 2.0 CMP model with ejbSelect methods. These are again queries specified using the EJB QL, but can return any type. These queries are however not exposed to the client view and can only be called by the business methods.

The EJB 2.0 solution

`OrderBean` and `Product` have an independent existence and need to be directly accessed. The two are modeled as Entity Beans with both Remote and Local Interfaces. `LineItem` and `ShippingAddress` are largely dependent on the order and need not be accessed directly. Whenever they are accessed, in most cases, they will be done in the context of an Order. They are modeled as Entity Beans with local interfaces alone.

The `OrderBean` has two CMR fields

- One-to-many bidirectional relationship with `LineItem`
- One-to-One Unidirectional relationships with `ShippingAddress`

They are represented as follows:

```
public abstract OrderBean extends Entity Bean {
// Virtual Files <cmp-fields>
public abstract Long getOrderID();
public abstract void setOrderID(Long orderID);
// Virtual Fields <cmr-fields>
public abstract Address getShippingAddress();
public abstract void setShippingAddress (Address address);
public abstract Collection getLineItems();
public abstract void setLineItems
(Collection  lineItems);
}
```

Note: The return type for `getLineItem` is a collection indicating a Many to Many relationship with the target.

The Shipping has One to One unidirectional Relationship with `Order`. It is interesting to note that since the relationship is not navigable from the Shipping address, there are no CMR fields in the Shipping Address.

```
public abstract LineItem extends Entity Bean {
// Virtual Fileds <cmp-fields>
public abstract Long getAddressID();
public abstract void setAddressID(Long addressID);

public abstract String getStreetName();
public abstract void setStreetname(String streetName);
// NO CMR Fields
}
```

The code for `LineItem` bean has two CMR fields – one for a bidirectional relationship with `Order` and other with the `Product`.

```
public abstract LineItem extends Entity Bean {

// Virtual Fields <cmp-fields>
public abstract Long getLineNumber();
public abstract void setLineNumber(Long lineNumber);
// Virtual Fields <cmr-fields>
public abstract Order getOrder();
public abstract void setOrder (Order order);
// Virtual Fields <cmr-fields>
public abstract Order getProduct();
public abstract void setProduct (Product product);
}
```

Finally we have the code for the Product Bean:

```
public abstract OrderBean extends Entity Bean {
// Virtual Fields <cmp-field>
public abstract Long getProductID();
```

```
public abstract void setProductID(Long orderID);
public abstract String getProductCategory();
public abstract void setProductCategory (String  category);

// NO - Relationship Fields
}
```

Since this Product Entity Bean has no relationships navigable from itself, the bean has no CMR Fields.

Summary

EJB 2.0 brings developers closer to real-world business models by enabling inter-entity relationships and local beans, and further enhancing application performance by extending support to dirty detection and lazy loading in a non-proprietary way. With the introduction of EJB QL, developers can author queries at the time of packaging components, independent of the target server platform.

Comparison

Feature	EJB 1.1	EJB 2.0
Container Managed Persistence (CMP)	Performance Issues: Database hits & disadvantages of data caching. No design option using CMP in EJB 1.1.	Introduction of Container Managed Relationships (CMR) to create fine-grained entity beans with relationship capabilities. CMR enables relationships due to introduction of One to One, One to Many, Many to One and Many to Many
Asynchronous Processing	Inadequate support	Support through MDB
Interoperability	No Interoperability as a requirement for EJB container thus disabling collaboration from different vendors.	Inclusion of the *Container Managed Persistence Model* supports complex business models, local beans.
Local Beans	Distribution of Beans – High distribution cost	Minimized distribution cost due to beans being modeled as coarse-grained objects enabling developers to specific beans as local.
Standards querying	No standards available	EJB 2.0 specifies standards for specifying queries with the introduction of EJBQL
Finder Queries	Limitation of Return type	Limitation removed by ejbSelect method