Design enterprise applications with the EJB 3.0 Java Persistence API

JPA provides a standard OR mapping solution, formerly the domain of third-party frameworks

Borys Burnayev

March 28, 2006

This article presents a Java™ Platform, Enterprise Edition (Java EE 5) design approach that leverages Enterprise JavaBeans™ (EJB) 3.0's new Java Persistence API (JPA). JPA provides a standard object-relational mapping solution that bypasses the need to rely on third-party frameworks such as Hibernate. You'll see details of a sample application that validates this approach and illustrates key design decisions.

The long-awaited next version of the Java EE 5 is knocking at the door (see Resources for links to the specification and preview release). Its many new capabilities include a revamped EJB architecture, one of whose prominent features is the JPA. Loaded with both in- and out-of-container persistence options, JPA brings J2EE architects a number of new design choices. This article focuses on the design of in-container applications relying on an EJB container for enterprise services, such as transactions and security.

I'll use the familiar PetStore application to give you a test-drive that demonstrates JPA's promise and how it might challenge traditional J2EE design patterns. The application is relatively trivial, so rather than provide exhaustive implementation details, I'll illustrate noteworthy design aspects with code excerpts (see Download for a link to the complete source code). This article assumes that you're familiar with basic EJB 3.0 and object-relational (OR) mapping concepts (see Resources for more information on both subjects).

Design overview

The example PetStore application is a Web-based, e-commerce application that realizes the following use cases:

- Browse products
- Find products
- Maintain account
• Maintain cart
• Create order

The application is designed as a multitier Java EE application with three major logical tiers:

• The **presentation tier** (not the focus of this article) uses the Struts framework.
• The **service tier** is a slim service facade that delegates all the work to its collaborators. The purpose of the service tier is to separate service provision from service implementation.
• The **data access tier** is a set of coarse-grained Data Access Objects (DAOs) (see Resources) implemented as stateless session beans. They rely on the Java persistence entity manager for their persistence needs.

The application domain model is represented by EJB 3.0 entity beans and is used for communication between tiers. When a domain object leaves the data access tier, it gets detached from the entity manager. It needs to be reattached when it reenters the data access tier.

Annotations is a feature of Java 5 that seems to be used extensively, and JPA is no exception. Annotations can be used to specify OR mapping -- something you see all over the available documentation and tutorials -- and the PetStore application uses them for that purpose too. However, it's worth mentioning you can also specify OR mapping by means of mapping files. The OR Mapping section later in this article discusses and compares these two alternatives.

I developed and deployed the PetStore application in the JBoss application server (see Resources). I performed most of the development with a commercial database and then migrated the application back end to the PostgreSQL database. (The OR Mapping section includes a discussion of potential effects of database migration you should aware of when using JPA.)

One of the goals of this case study was to come up to a design that allows for a highly testable implementation. As you'll see in the Testing section, you can use a range of testing techniques to test the PetStore application.

PetStore's design leverages the fact that it is a regular Web application. The main benefit is that all the tiers can run within the same JVM, eliminating the need for component distribution. This article's Remoting section outlines an approach to adding remoting capabilities to the application.

### The service tier

The service tier is designed as a service facade. It is implemented by a single stateless session bean, PetStoreService. The bean wholly depends on its collaborators for the services it provides.

Because the simplified PetStore requirements are limited to retrieving the data from, and storing it to, the database, the only collaborators are DAOs. Real applications can invoke Web services, access other applications via RMI/IIOP or resource adapters, generate e-mail messages, and so on. All these capabilities call for additional collaborators.

Collaborators can be injected by means of @EJB or @Resource annotations (shown in Listing 1) or through a @PostConstruct method (shown in Listing 2):
Listing 1. Injecting collaborators with @EJB

```java
@EJB(beanName = "AccountDao")
AccountDao accountDao;
```

Listing 2. Injecting collaborators with @PostConstruct

```java
MessageSource messageSource;
@PostConstruct
public void init() {
    messageSource = new MessageSourceImpl("exceptions");
}
```

The key factor in selecting the testing strategy for the bean implementation class is the fact that the class fully relies on its collaborators to provide its services. This means that the class's interaction with the collaborators is what needs to be validated. As you'll see in the Testing section, a mock-objects approach perfectly matches the goal.

The data access tier

The data access tier is designed as a set of coarse-grained DAOs. The DAOs are implemented as stateless session beans, one bean per logical domain: AccountDao, OrderDao, and ProductDao.

Each bean has the entity manager injected into it:

```java
@PersistenceContext(unitName = "manager1")
protected EntityManager em;
```

This is the application's most persistence-aware tier. It uses the all-new Enterprise JavaBeans Query Language (EJB QL) extensively (see Resources). All the persistence-related actions take place in this tier. For example:

```java
profile = (UserProfile) em.createQuery("from UserProfile up where up.login = :login").setParameter("login", login).getSingleResult();
```

And here's another example:

```java
em.persist(account);
```

The fact that the classes are persistence-aware demands an in-container testing strategy, which the Testing section describes.

The domain model

You can think of JPA as the successor to well-known transparent persistence technologies such as JDO and Hibernate. Although transparent persistence can be considered an add-on service that can be applied to persistence-ignorant Plain Old Java Objects (POJOs), JPA does impose a few restrictions on domain objects.
First, you usually want to have a (surrogate) object identifier that maps to the primary key of the corresponding database table:

```java
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "AccountSeq")
@Column(name = "account_sysid")
public Integer getAccountId() {
    return this.accountId;
}
```

Second, in an online transaction processing (OLTP) environment, you want a version field/JavaBeans property for optimistic concurrency control:

```java
@Column(name = "row_version")
@Version
public int getVersion() {
    return version;
}
```

Finally, if you choose to go with annotation-style mappings, mapping annotations will be spread around your code.

Another restriction arises from the fact that the domain objects can be accessed both in the tier where they are managed by the entity manager and in all the other tiers where they are detached. Moreover, an object can be new or removed. In the general case, the behavior of an object depends on its persistent state. For example, suppose the following line of code executes within the realm of entity manager:

```java
int size = attachedProduct.getItems().size();
```

The items collection populates with items for the given product, and the value of `size` variable is greater than 0. If the same line of code executes in the presentation tier, the value of `size` variable is 0 if the default lazy fetching configuration for one-to-many associations is in use. In other words, there is no way to tell a lazy object apart from an eager one outside of the realm of the entity manager (and it really doesn't matter within). The way to address the issue is to enforce project guidelines and conventions.

You can apply the same reasoning to business methods within domain objects. They should be ready for invocation in any tier, whether within the realm of the entity manager or not. That's why it might make sense to limit the number of business methods in domain objects to a necessary minimum.

All that having been said, the domain objects still retain a lot of their POJO nature. This means that you can test the business methods using plain old Java tests (POJTs), a term introduced in Expert One-on-One J2EE Development without EJB (see Resources.)

**OR mapping**

OR mapping is a crucial element of an application designed around JPA. It directly affects the way the entity manager populates the domain objects. As a result, a mapping change can be felt as
far as the presentation tier. You can produce especially nasty effects by changing fetch type or cascade type.

As I mentioned earlier, you have two ways to define the mapping: metadata (annotations) and a mapping file. Although the metadata approach is heavily promoted, you should be mindful of the baggage it carries. The approach essentially entangles two logical layers of an application: the domain model and the mapping information. Being separate, these logical layers require separate testing using different techniques. The metadata approach doesn't hamper the testability of the layers per se. Rather, it causes two layers to appear to be one, which might or might not be a problem depending on a range of factors.

One factor that can affect the choice of a mapping approach is the project team structure. In a small project with a few developers, a small number of tables (fewer than 100 as a rule of thumb) and no dedicated mapping person, taking the annotation approach for defining mappings is probably the best bet because working with annotations is generally faster. For medium-sized and large projects that have a dedicated mapping person or team, a smarter choice is to go with a mapping-files approach. This approach reduces resource contention and adds another degree of freedom to the development process. The metadata-based mapping approach proved to be more time-efficient for the PetStore application.

The goal of an OR mapping layer is to shield the rest of the application from the effects of changes in the underlying database. When I migrated the PetStore application back end to PostgreSQL, the mapping layer did not require any changes at all. This can be attributed to the fact that both the original database and PostgreSQL support sequences, so the primary key generation strategy stayed intact. In the general case, you should expect some rework in the area of the mapping related to object ID handling.

Thorough test coverage of the mapping is of paramount importance. You must cover all the relationship mappings to make sure fetching behavior and transitive persistence is tested. You can leverage out-of-container use of JPA to perform the task (more on that in the next section).

**Testing**

The key element in the service tier's design is the focus on the collaboration with underlying tiers for providing the requested services. This allows for a reliable testing strategy that uses dynamic mock objects. I used the EasyMock framework (see Resources) to implement the testing approach.

The DAO layer has strong database cohesion. That's why reliable testing requires some sort of in-container strategy with database access. Although it's a no-brainer with remote EJBs, a meaningful approach appropriate for local beans required some consideration. The aggravating factors here were:

- The need for an in-container testing facade
- The fact that the domain objects are not serializable, so all the verifications need to happen within the container
The JBoss Embeddable EJB3 container (see Resources) (in the beta stage at the time of this writing) proved to be a good fit. It can be started from within a unit test, so all the code runs within the same JVM. The in-container testing with the embeddable container achieved its goal but was relatively slow, with the container startup time being around 30 seconds. Perhaps this can be attributed to the relatively early state of the product or can be improved with a proper configuration.

I tested the business methods of domain model classes with POJTs. Other testing techniques are not necessary or appropriate for these classes.

The OR mapping is a major layer that needs exhaustive test coverage. The layer is well aware of the database, so neither mock objects nor POJT techniques should be applied. Instead, you can leverage JPA's out-of-container capabilities. I used this strategy to test the PetStore OR mapping layer.

You should keep in mind the importance of covering associations and transitive persistence behavior in the tests. This way you can notice a change of fetch type or cascade type attributes early and take an appropriate action.

**Remoting**

The key feature of the PetStore application design is its local nature. All the application's logical tiers are intended to run within the same JVM, an approach that has many benefits. See *Expert One-on-One J2EE Development without EJB* (see Resources) for an excellent discussion.

Nevertheless, it's worth noting that you can add the remoting capabilities to the application quite easily by means of a remoting facade. The remoting facade, as opposed to the service facade I described earlier, exposes a remote interface and has two responsibilities: converting the domain objects to/from serializable data transfer objects (DTOs) (see Resources) and calling the appropriate method on the service facade.

You can implement this approach with a remote stateless session bean. The only hurdle is creating an additional layer of DTOs and converting them to/from domain objects. However, you need it to ensure a clean interface and achieve a looser coupling with the remote clients.

**Conclusion**

EJB 3.0 and JPA will undoubtedly be major selling points for Java EE 5. They bring the standards-bound Java community competitive advantages in some areas and put it on par with the competitors in others. (Admittedly, in some areas no standards-based approach exists now.)

The Spring Framework (see Resources) has been EJB's major competitor in the enterprise space for the last few years. The EJB 3.0 specification addresses many of the issues that facilitated the rise of Spring. As it stands now, EJB 3.0 arguably provides a better development experience than Spring does -- the most noteworthy advantage being that it doesn't require configuration files.

JPA provides a standard OR mapping solution that's well-integrated into an EJB 3.0-compliant container. JPA's progenitors will continue to thrive, but their raw use in business applications will
probably diminish. Implementing JPA-compliant entity managers seems to be the likely direction of these technologies.

At the time of this writing, the EJB 3.0 specification is in its Proposed Final Draft stage (see Resources). Some outstanding issues with the preview implementations relate to JPA:

- The JPA specification in its current form does not define read-only entity beans. This is puzzling, because this feature is supported for EJB 2.1 specification-compliant entity beans. The Spring framework does support read-only transactions.
- The pluggable persistence provider concept remains an undelivered promise.
- A standard optimistic concurrency exception -- `OptimisticLockException` -- appeared for the first time in the EJB 3.0 Proposed Final Draft. Until the persistence providers implement it, you'll need to use a provider-specific exception, such as Hibernate's `StaleObjectStateException`, for detecting an optimistic concurrency problem. For now, this ties your implementation to a particular persistence provider.

A bigger issue with the Java EE umbrella specification has nothing to do with JPA. It relates to integration between the Web and EJB containers. Spring still has a major competitive advantage in this area. JBoss's Seam project (see Resources) tries to solve the problem in a custom way. The Caucho Resin application server (see Resources) tries to extend container boundaries and supports the use of `@EJB` annotation in the Web container. The hope is that Java EE 5.1 will address the tier-integration issue to give us a comprehensive and standard dependency injection approach.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PetStore</td>
<td>j-ejb3jpapetstore.zip</td>
<td>149KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2006  
**Trademarks**  
(www.ibm.com/developerworks/ibm/trademarks/)