An Original Implementation of EJB 3.0 Persistence

Experiences for Implementers and Users

Overview

Object / Relational mapping tools have had a long and difficult childhood, but at last some standards are starting to emerge that may finally solve this thorny problem, as exemplified by the EJB 3.0 persistence specification that makes up part of JSR 220: Enterprise Java Beans 3.0. This standard was driven by three successful providers of O/R solutions: KODO from SolarMetric (and now BEA), TopLink from Oracle and Hibernate from JBoss. There are notable differences between these frameworks: KODO is based on JDO, TopLink has a long history pre-dating Java itself, and Hibernate is probably responsible for the recent surge in confidence in O/R mapping. However, while the adopted standard clearly borrows from all these sources, it is not necessarily clear that it will work when other existing O/R frameworks attempt to support it. This discussion paper looks at a set of different starting assumptions that were used to develop an alternative mapping framework, and looks at how compatible the result is with the prevailing consensus expressed by the EJB3.0 persistence specification.

The paper concludes by demonstrating that an implementation of the EJB specification is indeed possible even with these differing assumptions and approach. However, and perhaps most interestingly, it highlights areas in which the specification:

- Leaves room for interpretation and therefore may suffer from portability issues between persistence providers.
- Dictates the implementation of features that were not felt necessary when taking a different set of starting assumptions.
- Has limitations in its ability to expose useful features of an alternative framework.

Yet another O/R mapping layer

You may well be thinking at this point “haven’t we solved this problem yet?” Why should you read on or consider any of the points made in this paper? The reason put forward is that the design of the O/R mapping approach considered in this discussion starts from a different place than the solutions used as a basis for the EJB standard. As the name Object/Relational implies, these popular solutions focus on mapping from Object to Relational, answering the question ‘how do I make these Java/C++/C#

objects that I've written persistent so that I can write them to, and read them back from a relational database?'. In this paper we answer the converse question 'how do I lay an object model over this existing relational database schema so that I can read and write Objects using a model that is decoupled from the underlying data?'. It is an approach used by "SQLMap" solutions such as iBatis and this change of emphasis has far reaching effects on some of the design decisions made.

Another point of interest to those considering the broader applicability of the EJB standard is that the framework developed here uses full-on code generation as its approach to the solution: a design that has recently fallen somewhat out of favour with O/R communities with increased focus on the absolute requirement to be able to persist plain objects (POJOs in Java). While the EJB specification does specify that the model be described as POJOs, it does not specifically preclude the use of code generation. Nevertheless, current indications seem to suggest that most solutions being worked on will rely more heavily on introspection. Here we attempt to show that a solution dependent principally on code generation is still possible with EJB 3.0 persistence.

This discussion has relevance not only to implementers of the EJB API, but also to clients of it, since it provides insight into potential cross compatibility issues and possible directions for future development.

Working Assumptions

The design started with the three key design assumptions described below. These are not the only assumptions that could form the basis of a mapping solution for relational to object data, but have been chosen because experience shows that they result in a relational to object mapping solution that works in practise. Those who are aware of the EJB 3.0 persistence specification will see that a number of the statements made in this list of assumptions are at odds with it. As we shall see, it is possible to reconcile these differences, but the way this was done and the implications of it explain the motivation for writing this paper.

> ☞ **Working Assumption**
> Start the design from a UML class model that makes no concessions to the data store. Get the data from an existing database schema over which you have limited control.

When mapping objects to a database you will generally derive your database schema from the object model. For example, you might choose to have one object map to one table and a relationship always be represented by a linking table with two foreign keys in it. Object/Relational frameworks often permit some flexibility in the mapping choices made allowing you to, for example, override the way inheritance hierarchies are mapped or the names of the relational table columns. Working in the other direction however, you aren't always given a database model that matches the logical model that you would like to be working with, but you should not have to compromise that model to fit the data model you have been given. Examples of where you may have limited control over your data model are:

- You are working with legacy data. In this context, a legacy database might be defined as as any non-trivial data store that is being used in a production environment and on which one or more systems depend. Such data stores are often less than ideally structured, if not because they were badly designed or maintained, then because of the inevitable requirements creep causing bolt-on enhancements during the lifetime of a project. It is generally very costly and impractical to restructure such a database to fit the model you would like to be working with, but an R/O mapping tool should permit you to perform this transformation.
- You need to denormalize your database model for performance. One common development practise when working with databases is to start by creating a fully normalized representation of the object data in the first cut of the code, and then to denormalize selected parts of the model to tune the performance at a later stage. This strategy works because the relational model guarantees us that we can switch between normalized and denormalized representations just by writing queries and therefore the badly performing queries can be replaced by better performing ones against the revised schema, but returning the same results. The strategy would not work if each normalization of the object model results in a change in the corresponding logical model.
- You want to use a problem-specific data representation. There are a number of patterns for solving specific data problems that result in database schemas that are quite far removed from the logical model. For example, star schemas, often used in data warehouses do not produce a particularly intuitive logical model by mapping each table to an object. Similar problems exist with temporal or versioned data in which some or all previous versions of the data are retained.
- You want to map only a part of a large database. A large organization often relies on large reference data stores or data warehouses. Applications that

need to use this data may only be interested in a small subset of the available data: the object model you lay over your data store in order to make use of the data in one context may be simpler than the object model required to fully represent the complexity of the reference data.

- You want to map information from several data stores into a single object model. The data sources for an object model may be split horizontally, with different subsets of the same data in different locations, for example, regional data stored in regional databases. Alternatively, the data may be split vertically, so that different objects, or even different attributes of the same object need to be read from different data stores. Clearly it may be difficult to permit update to database in situations like this, but there are many applications where the ability to update all of the data is not a requirement.
- Your object model may represent a summary view of the data in the database. For example, you may have a database of sales records, but you want to work with an object model that only represents sales totals. If your object model matches the database, you could put methods on these objects to provide the calculations, but this involves reading a lot of data out of the database into memory. Relational databases are actually very good at this kind of calculation and an R/O framework should support the ability to map the results from this calculation into objects.
- You do not have ownership of your data model. It is a relatively common situation in a large organization that the application developers who are interested in representing the data as an object model are in a different part of the organization to the database administrators whose job it is to design and manage the database. Sometimes application developers do not even have direct control over the design of the database schema, but even if they do, there may be policies in place (such as mandatory use of stored procedures for database access and update) that make it difficult to work with object/relational mapping frameworks that require the use of SQL that is generated by the framework.

In summary, there are many reasons in real world situations in which you need to start from the relational model and then build an object model over the top that is loosely coupled to it.

> ☞ **Working Assumption**
> The visible contractual boundary between Relational space and Object space is the Query Result Set.

A Query Result Set in this context is a tabulated set of results from a query consisting of a list of named attributes, followed by a list of rows with each attribute having a single scalar value for each row. A programmatic construct that represents a Query Result Set in Java is the java.sql.ResultSet interface.

This profound statement is at the core of this R/O mapping philosophy: that the Query Result Set serve this function is perhaps an obvious statement, since by definition all interaction with a relational database is in this form. The important part of the statement is that the boundary is visible and contractual. This means that:

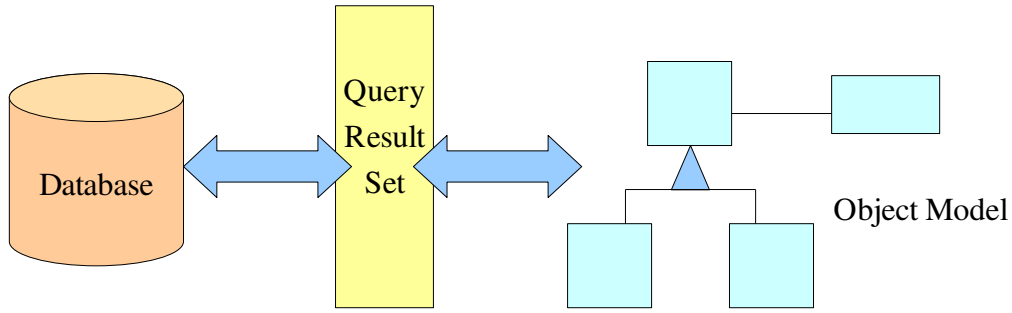
- The framework does not *rely* on an "object query language" which writes SQL and maps the results *behind the scenes.* That is not to say that an object query language is not very useful in easing the task of querying data for the developer, but the task of doing the mapping of data on the one hand and converting between the object query language and the query that will run against the database with its mapping information on the other, are separate and distinct problems.
- The framework does not rely on "dependent queries" that are quietly executed by the framework for example child object fetch queries or implicit lazy loading. Again, there are cases where it is useful be able to ease the job of the programmer by adding these features, but they are not an integral part of the mapping.
- You can map any result set into an object model including result sets that return information about multiple types of object, for example, loading both sides of a many-to-one or many-to-many relationship at the same time. This approach overcomes the so-called n+1 problem in mapping object relationships.

There are a number reasons for basing our mapping strategy on the Query Result Set:

- We are not throwing away any of the power of the relational algebra. Since this approach to writing queries won out over the navigational approach so long ago, those data retrieval solutions that were rejected back in the 70's can seem like fresh ideas today even though the reason for their rejection is as valid today as it was then.
- The concept of a Query Result Set is well supported in high level languages with, for example, JDBC or ODBC. Using the Query Result Set in this way permits us to leverage this work and gain instant compatibility with a broad range of RDMBSs.

- It's easy to see what is happening when you are researching unexpected problems in your application. When you don't get the objects you expected mapped into memory or the database does not get updated in the way you anticipated, you can look at what has passed between the two as easily expressed and viewed tables of data. Also, you have total control over what queries get executed, when. Lazy-loading strategies, as the name perhaps implies, give you a simple way of getting to your data without having to think too hard about what data you are going to need up front, but can result in unexpected performance problems that are difficult to track down and resolve.
- We can easily merge data from non-relational sources such as flat text files, as long as they can be represented as a query result set.

In order to making the strategy work we need a way to map from the relational model to a query result set and back again as well as a way to map from the logical object model to a query result set and back again.



Mapping from a relational database to a result set is simple: we just use an SQL query. Such result sets represent the result of a relational expression and as such may well be logically updateable. Being logically updateable means that you can write the necessary insert/update/delete query or queries given only the information about the values in the record set row to be updated. Some JDBC drivers are capable of doing this for simple joins where the primary/foreign key relationship is known to the database, but if the framework provides a way to customize the query for each case we can go beyond these restrictions. The concept of the updateable result set gives us the ability to read from and write to our relational database through a Query Result Set.

On the other side of the result set mapping process, going from a database result set to the logical object model uses a set of simple maps attached to the result set: one map for each object that should be built from it. This might say, for example that column

‘transaction_time_dt’ maps to attribute ‘transactionTime’ in object ‘Sale’. This map may be different for each distinct kind of result set. If we choose a subset of these mapped columns as the ‘key’ to define object uniqueness, we have a way of mapping objects from the result set whilst ensuring that we get no duplicates. Mapping from the object back to a result set can use the same maps to go the other way.

Combining these two mappings we can now effectively map bidirectionally between the database and our object model, and it has been shown in practise, when combined with a visual tool to assist with the mapping process to be an approach that very quick to build and lightning fast to execute.

- **Working Assumption**
  The design of R/O (and O/R) mapping is a problem that is separate and distinct from the design of an object or query results cache and an object query language.

The effort that goes into the design of an R/O mapping framework should not include creating an object cache in order to get acceptable performance in the real world nor is it a good idea to invent a new query language to make the framework viable. Caches and object query languages are useful tools, but they are problems that are distinct from R/O mapping.

When objects have been read from the database they occupy memory space in the process in which they are running. For efficiency it would be good to keep objects in memory for the lifetime of a process in order to avoid having to read them from the database next time they are used. However, the available memory is usually much less than the full size of the database being accessed so this is often impractical. Using a second level cache, however, it may still be possible to get significant performance improvements by storing a good proportion of the most often used objects in memory and others in a fast-access backing store such as local disk.

These considerations have meant that many O/R mapping tools (for example TopLink, from Oracle) have an integrated cache that can perform this job automatically.

The problem with caches is that they are a very difficult problem to solve well for all situations. For example

- What exactly do you cache, objects or query results?
- How to you ensure that the cache is always up-to-date?

- How can you write an efficient algorithm to check for cache hits?
- How do you build a cache that performs equally well for data that will be sparsely populated and data that can benefit from some form of bulk load driven by locality of reference assumptions?
- What do you do if you need to distribute the cache across several machines?

For this reason, it is argued here that the cache should be a separate, not integral part of the O/R solution.

There are also strong and very valid motivations for writing an object query language:

- SQL is has a nasty syntax with many pitfalls for the unwary.
- It is tricky (though not impossible) to write polymorphic SQL queries, yet these are important to object oriented systems and a key feature of object oriented databases.
- With knowledge of the object model, which is a given in any O/R mapping solution, it is possible to massively automate the task of writing these queries.
- An object cache may rely on a level of query abstraction in order to check for cache hits.

However, it is argued here that the construction of this is a separate problem from the data mapping that should be the core function of an R/O mapping solution.
Specifically:

- Making another query language part of the persistence solution is coupling the success of the persistence solution to the success of the query language, where they could be separate.
- There are many bad things about SQL, but the problem is in its syntax and the fact that every database provider has its own nuances, rather than what it is trying to do (relational algebra). Object query languages, while certainly making it simpler to write queries that access objects, lack SQL's ability to transform the data in ways not foreseen at initial design time. This is a very important consideration when mapping the kinds of legacy data store that are the target of an R/O framework.

In summary, the assumption for the framework that is the subject of this document is that while caches and object query languages are in general a good idea, they should be external to the R/O mapping framework. External caches or SQL generators can be plugged in, but they are not an integral or indispensable part of the core design.

Hydrate: An R/O Implementation

The R/O mapping framework that exemplifies the arguments put forward in this paper is Hydrate (http://hydrate.sourceforge.net). This is the open source, LGPL licensed, fully functioning and commercial strength R/O implementation that was built using the above working assumptions. Here we look at a simple example that was used to test the ease with which the framework could be made to meet the new EJB 3.0 persistence specification.

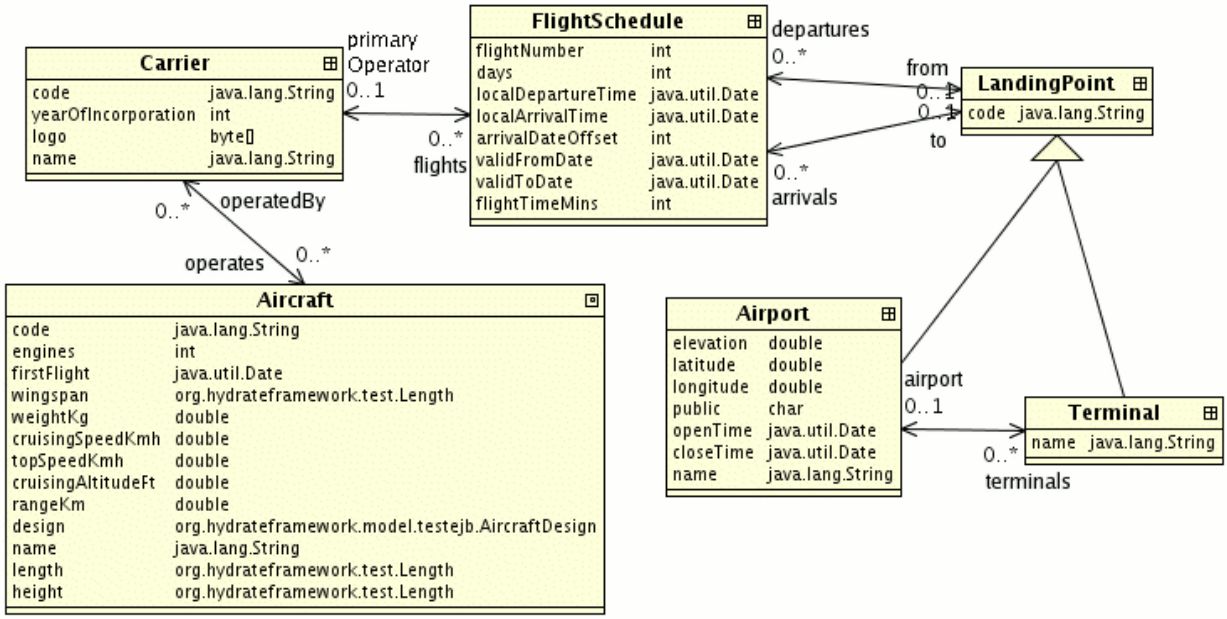
The Goal

The goal for this investigation was not to have Hydrate fully support all of the specification, rather to implement an extensive enough cross-section of it to understand firstly whether it was even possible, and secondly where any differences would show up, in particular:

- Perceived ambiguities that may result in potential incompatibilities between persistence providers.
- Parts of the specification that were not covered by the features already supported by Hydrate.
- Features of Hydrate that would be difficult to expose given the EJB 3.0 specification.

Test Model

A simple model was used in this exercise:



The model represents airline flight schedules including times of flight schedules with details of the carrier for each scheduled flight. The model also includes details of the aircraft operated by each carrier as well as the departing and arriving landing points for the flights, which may be either an airport or a terminal in an airport.

This model was chosen because it is simple enough to represent an easily understood problem to be solved but complex enough to include a few different relationship types including one to many and many to many relationships as well as simple inheritance. It was also chosen because the information to populate this model is actually widely available from a variety of on-line data sources, but in a format that is, in the case of most airlines or airports, only partially complete and for which a complete representation would require the merging of data from multiple data sources, each with a slightly different representation of the data model. In this sense it is exactly the kind of problem that Hydrate was written to solve in that you are starting with the data model over which you have little control and laying over the top of it an object model that you want to work with to solve a particular business problem.

The Challenges Posed With This Model

While this simple example does not require a full implementation of the standard, the model is complex enough to go beyond the most basic exigences of the API. It includes:

- Attributes with native types, serializable types e.g. java.lang.String, native type arrays and compatible object types.
- Date types where the temporal type (date, time or time-stamp) needs to be defined.
- Enumerated types.
- One-to-Many, Many-to-One and Many-to-Many relationships.
- Bidirectional relationships, that is relationships between objects that are navigable in both directions.
- Multiple relationships between the same object (potentially ambiguous).
- An inheritance hierarchy.
- Single-valued and composite keys.
- A key that includes an object reference (many-to-one relationship) as part of the key.

The Tests

The goal of the investigation was to consider the persistence problem in isolation and to this end all of the tests were written using ‘standalone EJB 3.0 persistence’, that is

working with the persistence provider outside of the EJB container. Tests were set up to perform the following basic operations:

- Initialize the persistence provider and obtain an entity manager instance.
- Start a transaction.
- Create a tree of unmanaged objects conforming to the above model.
- Use the entity manager to persist the objects thus created.
- Commit the transaction.
- Obtain another entity manager instance.
- Read back some of the objects by key and confirm that the data had been persisted correctly.

The following is a code snippet from the test cases:

```
public void testEntityManager() throws Exception
{
    EntityManagerFactory emf = new PersistenceProviderImpl().
            createEntityManagerFactory("schedules", null);
    EntityManager em = emf.createEntityManager();
    EntityTransaction tm = em.getTransaction();

    tm.begin();

    Airport stanstead = new Airport("STN", "London Stanstead");
    stanstead.setElevation(348);
    stanstead.setLatitude(51.885);
    stanstead.setLongitude(0.235);
    stanstead.setOpenTime(dfTime.parse("05:00"));
    stanstead.setCloseTime(dfTime.parse("23:00"));
    stanstead.setPublic(true);

    em.persist(stanstead);
    .
    .
    .

    Carrier ryanAir = new Carrier();
    em.persist(ryanAir);
    ryanAir.setCode("FR");
    ryanAir.setName("Ryanair");
    ryanAir.setYearOfIncorporation(1985);
    ryanAir.setLogo(byteArray("ryanair.gif"));

    FlightSchedule sched = new FlightSchedule();
    sched.setPrimaryOperator(ryanAir);
    sched.setFlightNumber(9804);
    sched.setDays(0x7f);
    .
    .
    em.persist(sched);
```

```
    Aircraft a320 = new Aircraft();
    a320.setCode("A320");
    a320.setName("Airbus A320");
    .
    .
    em.persist(a320);
    ryanAir.getAircrafts().add(a320);

    tm.commit();

    em.close();
}

public void testEntityManagerRead() throws Exception {
    EntityManagerFactory emf = new PersistenceProviderImpl().
            createEntityManagerFactory("schedules", null);
    EntityManager em = emf.createEntityManager();

    Carrier ryanAir = em.find(Carrier.class, "FR");
    assertNotNull(ryanAir);

    FlightSchedule sched = em.find(FlightSchedule.class,
            new FlightSchedulePK(ryanAir, 9804));

    assertNotNull(sched);
    assertTrue(sched.getFrom().toString().
            equals("Airport: London Stanstead"));

    em.close();
}
```

Building Using the Reference Implementation

The first step in the exercise was to implement the tests using TopLink the specification reference implementation. The above model was converted into Java POJOs and annotated according to the persistence specification, for example:

```
package org.hydrateframework.model.testejb;

import java.util.Collection;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;

@Entity
public class Carrier {
    private String m_code = null;
    private String m_name = null;
    private int m_yearOfIncorporation = Integer.MIN_VALUE;
    private byte[] m_logo = null;
    private Collection<Aircraft> m_aircrafts = null;
```

```
    private Collection<FlightSchedule> m_flights = null;

    public Carrier() {
    }

    @Id
    public String getCode() {
        return m_code;
    }
    public void setCode(String x) {
        m_code = x;
    }

    public String getName() {
        return m_name;
    }
    public void setName(String x) {
        m_name = x;
    }

    public int getYearOfIncorporation() {
        return m_yearOfIncorporation;
    }
    public void setYearOfIncorporation(int x) {
        m_yearOfIncorporation = x;
    }

    public byte[] getLogo() {
        return m_logo;
    }
    public void setLogo(byte[] x) {
        m_logo = x;
    }

    @ManyToMany
    public Collection<Aircraft> getAircrafts()  {
        return m_aircrafts;
    }
    public void setAircrafts(Collection<Aircraft> c) {
        m_aircrafts = c;
    }

    @OneToMany(mappedBy="primaryOperator")
    public Collection<FlightSchedule> getFlights()  {
        return m_flights;
    }
    public void setFlights(Collection<FlightSchedule> c) {
        m_flights = c;
    }
}
```

Two unexpected difficulties were encountered in setting up the annotations for this model. Firstly, the model calls for a composite primary key for the `FlightSchedule` object (`primaryOperator` and `flightNumber`) that

includes an object reference (`primaryOperator`) as one of its constituents. In the database, this reference is represented as the key of that object (`primaryOperator_code`), but the preferred representation of our Java POJOs only includes the object reference:

```
@Id
@ManyToOne
public Carrier getPrimaryOperator() {
    return m_primaryOperator;
}
public void setPrimaryOperator(Carrier x) {
    m_primaryOperator = x;
}
```

This declaration of this part of the key does not appear to work as is with the reference implementation, which reports a runtime error with the above code. There are of course workarounds: abandoning the composite key for a surrogate key is one, but this demands the exposure of an additional object attribute that is anachronistic to the object interface as well as potentially affecting the performance of queries by requiring additional joins in commonly used queries. Reference to Internet sources suggests replacing the @Id annotation above using the following approach:

```
@Id
@Column(name="PRIMARYOPERATOR_CODE", nullable=false,
    updatable=false, insertable=false)

public String getOperatorCode() {
  return m_primaryOperator.getCode();
}

public void setOperatorCode(String code) {
}

@ManyToOne
public Carrier getPrimaryOperator() {
    return m_primaryOperator;
}
public void setPrimaryOperator(Carrier x) {
    m_primaryOperator = x;
    setOperatorCode(x.getCode());
}
```

This approach is effective though, as with the surrogate key approach it adds an additional attribute to the object interface that probably doesn't belong there.

The second problem was encountered in specifying the primary key of the `LandingPoint` hierarchy. It was expected that it would be possible to declare the primary key at the base of the object hierarchy (`LandingPoint`), but the key had to be declared separately at each of the `Airport` and `Terminal` levels. As discussed in the conclusions, this unexpected behaviour is an example of how different implementations of the specification might behave differently with the same annotation information.

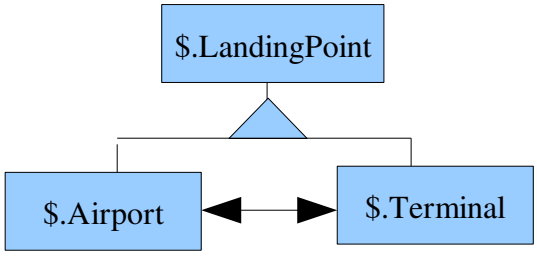
Ultimately, a working set of annotations was applied to the Java POJOs and a MySQL database was set up with a schema compatible to the above object descriptions. The TopLink essentials implementation was used in standalone mode and the tests described above were executed.

All tests passed successfully.

Switching Persistence Providers to Hydrate

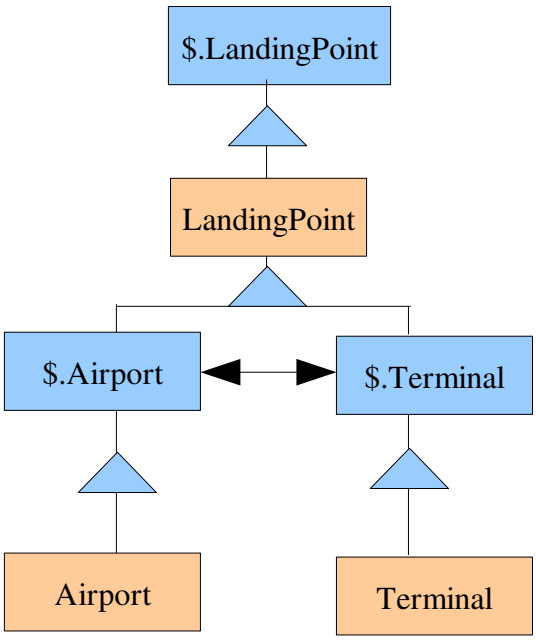
The motivation for this paper was to publish the experience in implementing the EJB 3.0 persistence standard in a persistence framework that focuses on mapping from the relational model to objects rather than the other way around. To conform to the standard in a way that would pass the tests, the following steps had to be completed:

- Implement the `javax.persistence` interfaces: `PersistenceProvider`, `EntityManagerFactory`, `EntityManager` and `EntityTransaction`.
- Parse the core annotations on the supplied POJOs to construct the internal representation of the object model.
- Write code to support the range of data types required by the tests.
- Write the code generators that would generate java sources and compile them on the fly.
- Perform bytecode enhancement on the original class files to fit them into the new inheritance hierarchy.



It is worth highlighting here the approach taken in using a code generation approach to support the EJB standard, since this demonstrates how an approach that relies principally on code generation can still be used with EJB 3.0. The original class hierarchy for the LandingPoint, Airport, Terminal inheritance tree was declared using POJOs as shown above.

The code generation and bytecode manipulation modified this tree as follows:



In the modified tree, the package name of the original objects was changed so that it included an additional '$', and the generated objects were written so that they matched the package and class name of the original objects. In this way, when the original code creates objects using the 'new' operator, it compiles against the original POJO classes, but when it executes, the actual classes created are instances of the code generated objects that are configured to derive from the classes that were originally defined.

A similar package name change and class override was required for the composite primary key class for the `FlightSchedule` object, although with Hydrate it was possible to specify the object reference (`primaryOperator`) to be a direct part of that composite key.

After implementation of these changes, the tests were shown to be working using Hydrate as a persistence provider.

Experiences

Ambiguities

The specification does seem to leave some scope for interpretation and these ambiguities in its precise scope are important as they highlight areas in which there may be incompatibilities between different implementations of the standard. There are numerous examples where it is possible to achieve the same thing in two different ways and there is a risk that different persistence providers will use different mechanisms. These ambiguities fall into two distinct categories.

Firstly, there are those associated with the way the annotations are supported, and the completeness with which they are supported. This means that an annotated persistence unit that works with one persistence provider is unlikely to work seamlessly with another. In this exercise, examples of these incompatibilities showed up:

- In the ability to use object references as part of a composite key (or not).
- As a difference in the requirement on where to specify the primary key for the objects in an inheritance hierarchy.
- In the case of identifiers used in the database (all upper case vs. mixed case).
- Where using join columns with keys that contained object references.

It is likely that these differences have just scratched the surface of potential cross portability issues and the lesson is that if you need guaranteed portability across providers, then you need to (1) write tests for all target providers and (2) suffer the limitations of the 'least common denominator'.

The second category where there is scope for ambiguity is in the database representation of the mapped objects. This is probably of most concern to enterprise application developers with long-lived data repositories who need to maintain independence from the underlying data persistence provider. In fact there is perhaps a dichotomy in the standard in this regard: while there are a number of annotations that provide direct control and hints about how objects should be mapped to a database, the standard also has to accommodate object databases as back ends. Given this fact, it a fair bet that a database created using one persistence provider will need to undergo a data migration effort when switching to a second provider, that relies on a clear understanding of these assumptions for both old and new providers.

Additional Features of EJB 3.0

The first, and most obvious gap in what was already offered by Hydrate is the object query language (EJBQL). One of the working assumptions behind Hydrate explicitly excludes an object query language as an integral part of the data mapping exercise. Instead, Hydrate relies on mapping the results of each query into objects in the persistence unit and this process is partially, but not fully supported by the concept of SQL Mapping annotations. In fact the absence of an OQL for Hydrate is not as serious a problem as it may at first seem: the process of writing a compiler for this kind of language is well understood and this, coupled with information already read from the other annotations can easily be used to go from the object query written in EJBQL to an SQL query combined with mapping information. In reality the component that does this would be largely decoupled from the rest of the persistence provider (apart from the assumptions referred to above about the database representation of mapped objects). It is relatively easy to imagine an EJBQL->SQL parsing component that is largely pluggable into different persistence providers.

The second significant gap in the services offered by Hydrate is its lack of container integration. This did not create problems in passing the above tests as they rely on an out-of-container implementation. Clearly though, this integration would have to happen to complete the work.

The remainder of gaps that would need to be plugged in Hydrate mostly seem to fall into two camps: supporting more complex Java Object representations and permitting some degree of control over mapping to the underlying database schema. Examples of the first include:

- Embedded objects. These are useful in representing composite types, for example a ‘currency’ type which might of consist of a currency symbol and an amount. Interestingly, however, it seems that this is the preferred way of representing a composite key though for this author this treatment of composite keys is something of an anti-pattern.
- Collections (to-many references) as maps and sorted lists. In Hydrate all collections are actually sets and it was something of a concious decision to exclude support for other collection types and maps. This is because sorted and indexed representations are usually just applicable to one way of using the object model rather than being an intrinsic part of the model. In the example, a list of departure schedules for a landing point could be ordered by date, or carrier and which is more useful depends on the way the model is being used.

Examples of control over mapping include:

- Different strategies for inheritance mapping
- Secondary tables that allow objects to be explicitly spread across more than one table in the database.

Yet again, it would be a relatively simple task to implement these features, though some were consciously excluded from the Hydrate implementation and the importance of others is not so great in the R/O mapping case with its focus on getting data from the relational model.

Original Features Found in Hydrate

There are a few features of Hydrate that have proved challenging to expose via the persistence API. This is unsurprising given the genesis of the EJB standard as an Object-to-Relational mapping standard, but it is worthwhile looking at a few of them here to see what might be missing from an extended standard that would also cater for better mapping in the relational to object direction.

The first gap not covered by the specification is in the provision of multiple alternate keys for an object. This is a prime example of a consideration that is important to R/O mapping and not so important to O/R mapping. In the O/R case, starting from objects you get to define which key you use. Going in the other direction you need to put up with whatever key or keys you have in your data and the best one to use may vary depending on the source of your data, especially when dealing with legacy information. In this context it is imperative to be able to handle data referenced or loaded via alternate keys. In Hydrate, each object has one or more keys, exactly one of which is designated the primary or default key. The only thing special about the primary key is that it is the one that gets used when no other key is specified. Keys are used to determine object uniqueness when data is read from the database.

Perhaps the most significant Hydrate feature for which there is undersupport in the specification is for mapping result sets into objects, the need for which is a core tenet of the mapping approach. Again, this is an example of something that is more important in the R/O case, since you are likely to be loading data from many different and disparate data sources. The standard does provide for SQL Result Set Mappings but these fall short of what is offered by Hydrate in the following areas:

- In Hydrate a result set map consists of a list of object mappings (supported) each mapping specifying a key to use (not supported) and a list of column to attribute name maps (supported).
- Each object mapping has a predicate that determines from the data available in a particular row whether it should try to build an object or not. This can be a simple comparison as is used for object type discriminator columns or something more complex.
- Relationships are mapped either implicitly via the inclusion of a foreign key in the mapped information for an object (supported, but not for many-to-many relationships) or explicitly via the nomination of objects to join to (not supported)
- In mapping objects, not all attributes need to be mapped (not supported).
- Mappings work in both directions: the result set is assumed to be updateable so that the same mapping is used (at least conceptually), to insert, update and delete objects in the database.

In general, EJB persistence sees EJBQL as the primary way of accessing data and the SQL Result Set Mappings provided by the persistence API are something of an afterthought and part of a separate implementation. As such, there may be things that cannot be done with result set mappings, such as mapping many-to-many relationships. The working assumptions of Hydrate mean that while EJBQL can still be the primary way that data is accessed for most cases from the perspective of the user, it should be built on top of the result set mappings, in the same way as a high-level programming language builds on a set of underlying libraries. This means that you can always kick down to this raw form of data access in the same way as a programmer in a high level language is free to reimplement unsatisfactory library classes if performance or mapping complexity considerations demand it.

Other features supported by Hydrate that cannot be exposed directly through the persistence API include:

- Enumerated types where the in-memory enumerated value is referred to with a different name than its code in the database. For example in a Person table, it is common practise to use a character value for the Sex ('M' or 'F') whereas you would probably prefer to refer to this sort of thing in code as Sex.Male and Sex.Female.
- Not set and Not Loaded values. All attributes (even native types) can have a special state of 'not set' (equivalent to Null in the database) and 'Not Loaded' (where the attribute wasn't read from the database). These states have been

shown to be very useful in handling null values from databases and partially loading objects from the database (or loading different parts of an object from different data sources).

- Dynamic SQL queries - like JSR220's EJBQL, Hydrate extends the SQL statement to provide the concept of named parameters (the `NamedParameterStatement`), and provides a very useful implementation of this interface that can dynamically add parts of the where clause as parameter values are supplied – similar, but more powerful than the queries offered by mapping tools like iBatis. Thus the same query can be used to select all data from a table, ranges of data, or specific rows since the parameters not supplied to the query when it was run will be dropped out of the where clause.
- Hydrate provides a handy GUI for mapping the results of any query, but particularly these dynamic SQL queries, into the objects in the model. This GUI creates an XML file as output that defines the SQL query text, the elements of the where clause and the Result Set Mappings that need to be applied to populate attributes and link the resultant objects. This tool could be adapted to help write SQLResultSetMapping annotations, but as noted above, some functionality would be lost.
- Automated reverse fixup of bidirectional references. In EJB persistence, a reference is always 'owned' by one side or the other which creates difficulty where it is visible in both directions. In Hydrate a reference is owned by neither object it simply has an independent multiplicity and navigability in each direction. As such a many-to-one relationship is no different from a one-to-many relationship in the other direction. For references that are owned it is easy to say that the owned side determines how changes will be reflected in the database, but the owned side is generally an arbitrary concept and keeping both sides of a bidirectional relationship consistent can be quite tricky to get right. This is particularly the case when you consider the cascading ramifications of any such change on the previously referenced object. Hydrate does this all automatically, both when populating the references in the first place and when they are updated through the object's public interface.
- XML Schemas: though properly not part of a pure O/R mapping framework, Hydrate supports the mapping of objects in the persistence unit into one or more XML schemas and back again. Since the annotations provide all the information necessary to perform this operation efficiently and accurately, it does represent a possible future extension for EJB.

Notwithstanding these differences, a fully working version of the tests was successfully implemented using Hydrate as a basis. A copy of the source code for the tests and the implementation can be found by downloading the latest Hydrate package from http://sourceforge.net/project/showfiles.php?group_id=146051

Overall Conclusion

In conclusion, this exercise has demonstrated that a mapping framework that starts from a completely different set of assumptions that are on the face of it incompatible with EJB 3.0 persistence, can be adapted relatively easily to conform to that specification. This statement does however come with a few caveats.

- There seems to be significant scope for interpretation of the specification, and this means that early adopters are unlikely to be able to switch seamlessly between persistence providers unless they are severely restricted to the most basic usage of the interfaces.
- A mapping framework that is based around significantly different assumptions from those used to drive the EJB 3.0 persistence development should anticipate firstly some additional work to fully meet the standard and secondly to have to provide some hooks to the underlying framework in order to leverage its full capabilities.

Appendix

About the Author and the Framework

The author is a commercial Java programmer of some 7 years' experience having previously spent a similar time as C++ programmer. The framework described in this paper was developed in Java over the past 5 years.

References

The JSR 220 (Enterprise Java Beans 3.0) - Linda DeMichiel et al.
http://www.jcp.org/en/jsr/detail?id=220
EJB 3.0 Focus: Ease of Development – Linda DeMichiel
http://www.c-sq.com/data/maru/2004/20050309/LindaSeminar_6.pdf
Getting Started with EJB 3.0 Persistence out-of-container using the Reference Implementation – Lucas Jellema
http://technology.amis.nl/blog/?p=962
Banish Your Resistence to Persistence with EJB 3.0 Persistence API
http://www.devx.com/Java/Article/30161
Standards-based Object Persistence FAQ (from BEA)
http://www.solarmetric.com/Software/Kodo/persistence.php
The Foundations of Object Relational Mapping – Mark Fussell
http://www.chimu.com/publications/objectRelational/index.html
Mapping Objects to Relational Databases
The Joy of Legacy Data
Why Data Models Shouldn't Drive Object Models (and Vice Versa)
– Scott Ambler
http://www.agiledata.org/essays/mappingObjects.html
http://www.agiledata.org/essays/legacyDatabases.html
http://www.agiledata.org/essays/drivingForces.html