Evan Byrne

December 4, 2012

Valkyrie: A Simple ORM

After developing websites and other software for the past seven years, I can say with confidence that Object-Relational Mappers are one of the most useful tools that a software developer can have at his disposal. Object-Relational Mappers, or ORMs, are so useful that they are often an integral part of popular software development frameworks. For the purpose of personal education I decided to try and design and program my own simple ORM for the Java programming language which interfaces with SQLite 3 databases. Creating an ORM is no easy task though. Going through the process of creating the Valkyrie ORM has not only taught me about the complexities of ORMs, but has also given me valuable practical experience with the Java programming language. In this paper I will summarize all of my work on the Valkyrie ORM, give example usage of the different components, and explore some of the obstacles that made development difficult.

Object-Relational Managers (which are called ORMs for short) are used to convert database data to some sort of object-oriented API, and vice-versa. Typically ORMs are used to make interfacing with relational databasing systems less difficult. Because relational databasing systems are usually pretty complex and feature-rich, the ORMs which are used to interface with them tend to be complex as well. However, the usefulness of ORMs have made them well worth the effort of designing and developing, so there are quite a few Java ORMs in existence. A couple notable Java ORMs that support SQLite include: OrmLite (1) and Hibernate (2).

Valkyrie is a Java ORM which I designed to interface with SQLite 3 databases (3). The main reason that I decided to create Valkyrie was personal education. By working through the process of creating a simple Java ORM, I have gained valuable software design and development experience with the Java programming language. I had previously had very minimal experience working with Java, most of which was in a classroom environment, so Valkyrie gave me practical knowledge of working with a strictly-typed programming language that was new to me. Because my main objective was to gain practical experience, I would say that Valkyrie was clearly a success.

While working on Valkyrie as a whole I tried very hard to keep it both simple and well-organized. One example of the simplicity of Valkyrie is the installation. After adding the sqlite4java JRE (4) to the project build path, using Valkyrie in an application is as simple as just dropping the `com/beakerstudio/valkyrie` directory into the project's source folder, and then importing whatever classes that are needed. A great deal of effort was put into making Valkyrie very easy to use, while still being fully-functional, and the final results reflect this.

After installation, the first thing that needs to be done is establish a SQLite 3 database connection. I thought of various ways to handle this automatically, but in the end I decided that it would be best to allow the developer to manually open and close the database connection. To open and close a database connection, just use the `open()` and `close()` class methods on `com.beakerstudio.valkyrie.Connection` like so:

```
Connection.open("testdb");
// Make database queries...
Connection.close();
```

After a database connection has been established, then database queries can be made. Once the program is done running queries, then the database connection must be closed. In order to make queries against a SQLite 3 database, the developer may either bypass the ORM completely and manually prepare and execute SQL queries, or he/she may define model classes and use the various methods of the model classes to manipulate the database. Since bypassing the ORM defeats the purpose of using it in the first place, I won't spend any time discussing how to manually prepare and execute SQLite queries.

Models in Valkyrie are simply classes that extend `com.beakerstudio.valkyrie.Model` and are used to interact with the database. Each model class represents a SQLite table in the database. A model that maps to the `user` SQLite table could look like this:

```
class User extends com.beakerstudio.valkyrie.Model {
    @Column(primary=true)
    public Integer id;

    @Column
    public String name;

    @Column
    public String email;

    public String foobar;
}
```

Each field marked with the `@Column` annotation represents a column in the database table. So in the above example `id`, `name`, and `email` are fields in the `user` SQLite table, but `foobar` is not, because it does not have the `@Column` annotation. Additionally, the `id` field has the `primary` option set to `true`, which means that the `id` is the primary key for the SQLite table.

Once created, a model class can be instantiated, and then those instances can be used to manipulate the database. All Valkyrie models have the `create_table()` method, which generates the table schema and then creates the table in the SQLite database. This is a particularly useful feature, because it frees the software developer from having to manually create the SQLite tables in addition to creating the model classes. Also, the `drop_table()` method is available, which of course drops the table from the database.

As with most ORMs, performing basic CRUD operations with Valkyrie is a very straightforward process. A fully populated model instance can be thought of as one-to-one mapping to a SQLite table row. So naturally an insert query for the `User` model created previously might look something like this:

```
User u = new User();
u.id = 1;
u.name = "Evan Byrne";
u.email = "evantbyrne@gmail.com";
u.insert();
```

The above operation is logically equivalent to the following SQL:

```
INSERT INTO "user" ("id", "name", "email") VALUES
     ('1', 'Evan Byrne', 'evantbyrne@gmail.com')
```

If any column was to be omitted (remain null), then that column would not be present in the inserted row. Once data has been inserted into the database, then it can be selected, updated, and deleted.

Selection of data can be accomplished in a number of ways. The first way is to use the `get()` method, which uses the table's primary key to retrieve a single row from the database. The following code would be used to grab the row inserted in the above example:

```
User evan = new User();

evan.id = 1;

evan.get();
```

Note that `get()` populates the model instance that it's called from. This means that the `evan` instance above would contain the populated `name` and `email` fields. The second way to select data is by using the appropriately named `select()` method, which returns an instance of `com.beakerstudio.valkyrie.sql.Select`. The `Select` class contains many methods which can be method-chained together to modify the query. Of all these methods, `fetch()` runs the query and returns a vector of model instances. In its simplest form, usage of `select()` looks like this:

```
User u = new User();

Vector<User> results = u.select().fetch();
```

The above is logically equivalent to the following SQL:

```
SELECT * FROM "user"
```

The select query can be modified by using various methods, such as `eql()`, which can be used like so to add `WHERE` clauses:

```
results = u.select().eql("name", "Evan Byrne").fetch();
```

Which would be equivalent to this:

```
SELECT * FROM "user" WHERE "name" = 'Evan Byrne'
```

In addition to the = comparison operator Valkyrie also supports <>, <, <=, >, and >=. Although `IS NULL` and `IS NOT NULL` aren't currently supported, they could be added without too much effort. The reason they aren't supported is simply because I've not had time to add them yet. Also, because my main goal while working on Valkyrie was self-education, adding every possible feature wasn't necessary. In addition to conditional operators, Valkyrie also supports `LIMIT` and `OFFSET` via the `limit()` and `offset()` methods, which can also be method-chained.

To remove rows from the database, all Valkyrie models have the `delete()` method, which works in a very similar way to `get()`. However, instead of only using the primary key to determine which rows will be affected by the query, `delete()` uses every populated row in the model. Take the following for example:

```
User u = new User();

u.name = "Evan Byrne";

u.email = "evantbyrne@gmail.com";

u.delete();
```

This is equivalent to running the following SQL:

```
DELETE FROM "user" WHERE "name" = 'Evan Byrne'

     AND "email" = 'evantbyrne@gmail.com'
```

Each populated column field adds an additional `WHERE` clause to the query. It's also possible to run a plain `DELETE FROM "user"` by leaving all of the column fields null.

Update queries in Valkyrie are a hybrid between `get()` and `delete()`, because they use the primary key to select rows, but then use the other columns to update the row values. To update a row populate the primary key, populate any columns which need to be updated, and then use the `save()` method, like so:

```
User u = new User();

u.id = 1;

u.email = "byrne1et@cmich.edu";

u.save();
```

Which is logically equivalent to this SQL:

```
UPDATE "user" SET "email" = 'byrne1et@cmich.edu'

     WHERE "id" = '1'
```

By using `get()`, `select()`, `delete()`, and `save()`; it's possible to effectively manage simple table structures. Although there are features missing which would be useful for more diverse table structures, the basic CRUD functionality is all there, so it would be straightforward to add additional functionality onto that area.

Once CRUD operations were working properly, the next step was to add one-to-many and many-to-many relationship support. Up until this point, implementation of different features were pretty easy and straightforward. However, adding foreign key support was a significant challenge. The first challenge associated with implementing foreign keys was that the field data type for the column couldn't just be a simple `String` or `Integer` object. Instead I had to create two new data types: `ForeignKey` and `HasMany`. In the following example, the two models form a one-to-many relationship by using these two custom data types:

```
public class Article extends Model {
     @Column(primary=true)
     Integer id;

     @Column(type="com.foo.bar.models.Comment",
          field="article")
     HasMany<Comment> comments;
}

public class Comment extends Model {
     @Column(primary=true)
     Integer id;

     @Column(type="com.foo.bar.models.Article")
     ForeignKey<Article> article;
}
```

In the above example, there is a one-to-many relationship between the `Article` and `Comment` models. Each article has many comments. Note that the `type` option on the `Column` annotations are set to the canonical name of the other model class, and the `field` option is set to the name of the foreign key.

Both of the foreign key data types have unique functionality which makes them very useful. Instances of `ForeignKey` class have the ability to leverage the functionality of the model that they point to, because they actually store an instance of that class. This can be seen below:

```
Article a1 = new Article();
a1.id = 123;
a1.insert();

Comment c1 = new Comment();
c1.id = 987;
c1.article = a1;
c1.insert();

Comment c2 = new Comment();
c2.id = 987;
c2.get();
Article a2 = c2.article;
```

In the above, the `a2` instance contains an `id` field populated with the value `123`. This makes it easy to use methods such as `get()`, `delete()`, and `update()`.

The second foreign key data type - `HasMany` - contains a few useful methods for inserting, selecting, and deleting child models. The usage of these methods can be seen in the following example, which continues the example above:

```
//...
// Select
Vector<Comment> a1_comments = a1.comments.select().fetch();

// Insert
Comment c3 = new Comment();
c3.id = 567;
a1.comments.insert(c3);

// Delete
a1.comments.delete(c1);
```

Like I said earlier, implementing foreign key functionality proved to be pretty difficult though. This was mainly because of the classes' heavy reliance on Java's reflection API. On more than one occasion I found myself spending a great deal of time trying to figure out which combination of reflection methods would yield the desired results.

After adding one-to-many support, the next step was to implement support for many-to-many table relationships. Like with foreign keys, this required the creation of a new data type, which turned out to be the `ManyToMany` class. In order to define a many-to-many relationship between two models, two things have to happen. The first thing that needs to happen is a middleman model must be defined. This model will need `ForeignKey` fields which point to both of the other models' primary key fields. A middleman model for a many-to-many relationship between the `User` and `Group` models might look something like this:

```
public class Membership extends Model {
     @Column(type="com.foo.bar.models.User")
     public ForeignKey<User> user;

     @Column(type="com.foo.bar.models.Group")
     public ForeignKey<Group> group;
}
```

Once the middleman model has been created, then each of the other models need to have `ManyToMany` fields defined:

```
public class User extends Model {
     //...
     @Column(type="com.foo.bar.models.Group",
          middleman="com.foo.bar.models.Membership")
     public ManyToMany<Group> groups;
}
```

```
public class Group extends Model {
	//...
	@Column(type="com.foo.bar.models.User",
		middleman="com.foo.bar.models.Membership")
	public ManyToMany<User> users;
}
```

After the middleman model is defined and the `ManyToMany` fields have been added, then operations can be run on the many-to-many relationship. Operations include adding models to the relationship, removing models from the relationship, and selecting related models. An example of the many-to-many relationship in action can be seen below:

```
User u1 = new User();
u1.id = 123;
u1.get();

Group g1 = new Group();
g1.id = 987;
g1.get();

// Add
u1.groups.add(g1); //or g1.users.add(u1)
```

```
// Select
Vector<Group> u1_groups = u1.groups.select().fetch();
Vector<User> g1_users = g1.users.select().fetch();


// Remove
u1.users.remove(g1); //or g1.users.remove(u1);
```

Behind the scenes `ManyToMany` fields work very similarly to `HasMany` fields. The main difference is that `ManyToMany` operations often involve multiple other models. This means in order to get select queries to work I had to add support for basic SQLite joins. Because join support was needed, implementing the `select()` method on `ManyToMany` took a great deal of time and effort. I'm glad that I took the time to fully implement many-to-many relationships though, because the end result was fantastic.

Development of Valkyrie was test-driven from the beginning. To test Valkyrie, I used the JUnit unit testing library. Unit tests were written for just about every feature of Valkyrie before their respective functionality was actually implemented. For example: before I created the sql.Select class, I created some basic unit tests, which tested to make sure the various method were generating the correct SQL. Creating the tests before implementation not only helped reduce bugs, but also helped speed up development by giving me a way to reliably check that each section of my code was doing what I thought it was. This in turn allowed me to narrow down the scope of problems, which allowed me to spend less time searching for problems in the code, and more time improving it.

Although I believe that my unit tests were good, if I had more time, then I would do a couple extra things to make them even better. The first thing that I would have liked to see done was the automatic deletion of the test SQLite database before any tests were run. Currently if a single unit test fails, then tables aren't dropped in the SQLite database. This causes the next run of tests to fail when they try to create SQLite tables. In order to prevent further tests from failing after a single failed test, the test database file has to be manually deleted. I'm not sure how I could get around this issue, but it's definitely something that I should look into for the future.

The second thing that I would like to see, which would improve my tests, is to simply have more of them. Right now the unit tests cover all the functionality of the library, but there simply isn't a large enough data set being used to ensure Valkyrie works in all intended use-cases. The solution to this would be to slowly add more tests over time, until it eventually would be satisfactorily complete.

In addition to the two small shortcomings of the unit tests, there are a couple of other things that need to be done before Valkyrie can be considered a complete SQLite 3 ORM. The most prominent thing that Valkyrie is lacking is support for more SQLite 3 data types. Right now Valkyrie only supports `TEXT` and `INTEGER` data types, which was enough for my own purposes. However, support for `DATE`, `DATETIME`, `TIMESTAMP`, `DECIMAL`, and other formats would need to be added if Valkyrie was to be used in any sort of production environment.

The second most prominent thing that Valkyrie is missing is support for configuring keys and indexes in the models. Right now the primary key can be defined, but other keys or indexes cannot. I imagine that this isn't a huge issue for most developers who are using SQLite, but it's still something that I believe should be in any complete ORM.

Despite a few minor omissions, I would say that Valkyrie was a success. Not only did I manage to create an ORM with all of the major components, but I also learned a great deal by working on the project. Going through the process of creating an ORM has given me valuable experience with the Java programming language and SQL databases in general. I plan on using what I learned from Valkyrie on future software development projects, including a content management system that I'm currently in the process of working on.

Works Referenced

1. OrmLite: http://ormlite.com
2. Hibernate: http://www.hibernate.org
3. Valkyrie: https://github.com/evantbyrne/valkyrie
4. sqlite4java: http://code.google.com/p/sqlite4java