Advanced Programming

Lab Sheet 18 – ORM & DAOs

Introduction

Although there are a large number of software systems that access databases directly using JDBC, developers sometimes find the task of writing boilerplate code to marshall objects into the database for storage, and to unmarshall data back from the database into objects for use in the project cumbersome. In recent years, it has become fairly common to see systems use an Object-Relational Mapping (ORM) library to handle some of this work. You may have heard of the hibernate [1], eclipselink [2], or openJPA [3] libraries, which all perform this task in a similar manner.

ORM libraries have something of a mixed reputation. Critics point out that they tend to be quite large and bloated, that code accessing a database via an ORM is significantly slower than code that accesses the database directly [4], and that the complex web of dependencies introduced increases the potential for bugs and security vulnerabilities [5]. Supporters point out that developer time is expensive, that there could still be bugs or security vulnerabilities in custom-written JDBC code, and that ORMs can make it easier to switch the database used by the application at a later stage [6].

In this lab session, you will gain familiarity with the ORMLite library [7] for Object-Relational Mapping. ORMLite is considerably less bloated than some other ORM libraries, and has far fewer dependencies. It has an easy-to-understand API, and supports SQLite – with which you are already familiar. You will learn how classes can be modified for use with an ORM library, how you can use an ORM library to perform standard database operations, and you will also gain some experience with Data Accessor Objects (DAOs), which are a common way to organise database code in larger applications with a class responsible for the loading/saving/updating/retrieving of objects of a specific type. I have uploaded a demonstration project to moodle to help you get started.

Importing the Code

The initial version of the program is available to download on Moodle as an eclipse project, exported as a .zip file. You will need to import the project into eclipse to start work. You can import it into your eclipse workspace by selecting File → Import, and then selecting "Existing Projects Into Workspace". You will see a dialog similar to that depicted in Fig. 1, below.

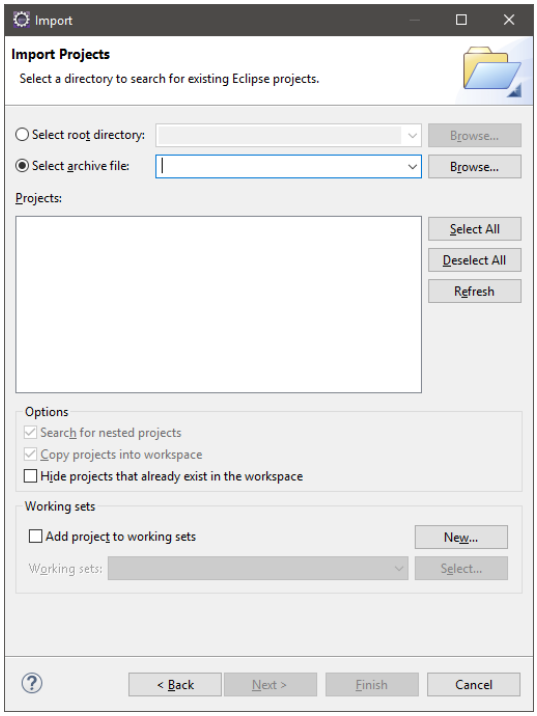

Figure 1: Eclipse Import Projects Dialog

On this dialog, you should make sure "Select archive file" is selected, then browse to the zip you downloaded from Moodle. Make sure that the "ORMLab" project is ticked, before clicking the "Finish" button at the bottom of the dialog. You will see the project in the left-hand panel in eclipse, along with your previous projects.

The ORMLab project has just three classes. The *Person* and *Pet* classes are what we call *POJOs*, or Plain Old Java Objects. They're simple classes with properties, getters and setters that adopt all the conventions you are used to. The ORMLite library requires us to do two things for objects of these classes to be persisted to the database:

1. The class must have a default, no-parameter constructor (it can have others too, though)

2. The class and its fields must be *annotated* to tell the library what needs saving, and where

The first requirement is easy. You'll have noticed that I normally include a no-parameter constructor in all of the classes I give you, and this is a good habit to get into. Quite a few libraries (not just ORMs) out there require that classes have a default constructor, and they're quick and easy to write.

The second requirement is a little trickier. You have seen annotations before, they're the @Tags that are sometimes placed above classes or methods (like @Override when using inheritance, or @Test in JUnit). The ORMLite library has a fair number of annotations, but two are particularly important:

1. The @DatabaseTable annotation, which marks a class with the name of the SQL table to use

2. The @DatabaseField annotation, which marks a property as needing saving to the database

Each of the annotations has one or more parameters, but fortunately they're self-explanatory.

The ORMLab project's Main class is a sample program that connects to the database, inserts some records, and performs some queries using the ORMLite library. I left some inline comments to help you to find each of those things in the class.

Task 1: Understanding The Existing Code

There's quite a lot to understand here, so don't rush! A sensible place to start is with the POJOs, in which the majority of the code is standard getters, setters and constructors. The Person class is the simplest, with annotations specifying the table name, the properties that will become columns, and a special annotation to tell ORMLite that the id property should be a primary key with auto_increment.

The Pet class is slightly more complex, because each Pet has an owner. This is a standard Many-to-One relationship (An owner can have many pets, but each pet has just one owner) that we would normally implement with a foreign key in the Pet class's database table. The ORMLite library does exactly this, but we use a special *foreign* parameter on the owner field to tell it to do so.

Run the program, and open the generated SQLite database using DB browser on your machine. Take a look at the SQL tables and data. Is everything as you would have done if you created the database tables yourself?

Now let's turn our attention to the Main class. This is less self-explanatory, with quite a few classes and methods from the ORMLite library called which you will not have seen before. The first call to an ORMLite method is:

```
ConnectionSource cs = new JdbcConnectionSource(connectionString);
```

This is the code that connects to the database. You can think of the ConnectionSource it as functionally equivalent to the JDBC's Connection class, which you used previously. This ConnectionSource is passed to the methods that create database tables, and the methods that create the DAOs, which we create with:

```
Dao<Person,Integer> personDAO = DaoManager.createDao(cs, Person.class);
Dao<Pet,Integer> petDao = DaoManager.createDao(cs, Pet.class);
```

Data Accessor Objects (DAOs) are a commonly-used design for separating database code from the rest of the application, to help keep things neat and tidy. If using this design, and application features a DAO object for each of the classes that will be loaded/saved from a database/file. The DAO is responsible for saving, loading, finding, updating and deleting objects of its associated type from the relevant data store. The DAOs created by ORMLite have various features, including several ways to find records. My personal favourite of these methods is named *Query By Example* (QBE).

Many different ORM libraries feature some form of Query By Example, which allows developers to query a database by supplying the ORM with an instance of the object that should be retrieved when making the query as an example of what it should find. This example object can have some of the class's properties set, and the ORM will search for records of that type with matching values. In the ORMLab project, the first QBE looks like:

```
Person example = new Person();
example.setSurname("Bloggs");
List<Person> results = personDAO.queryForMatching(example);
```

The *example* object is a Person, and the only property set on the example object is a surname. The ORM will search through the Person class's database table for all rows with the specified surname, returning them in a convenient list. The list will be empty if there are no matching records.

Feel free to play around with the project to help you understand what's going on. You won't be extending the existing project in the ret of the lab, so it doesn't matter if you break it. I would also recommend commenting out the line which disables logging for a while so that you can see the (very detailed) trace of what the ORM library is doing.

Task 2: Using an ORM in Your Own Program

Whenever I pick up an old Pokemon game for some nostalgic entertainment, I struggle to remember which Pokemon learn which moves in which games, not to mention when or how. Fortunately, there are some incredibly useful websites to help me! Pokemondb is one example. I would like you to create a program that, for a single generation, allows you to:

- Find all of the Pokemon that can learn a specific move
- Find all of the moves that can be learned by a specific Pokemon

You should create a small SQLite database (with 5-10 Pokemon and 20-30 moves), with appropriate data for each entity. I would recommend keeping track of the name/types/abilities/descriptions of Pokemon, and the name/description/type/category/power/acc/default pp of moves. I would recommend creating the Java classes first, and letting ORMLite create the tables for you, before using DB Browser to add the data to the tables.

Because this relationship is Many-to-Many, you will need to have an extra class and DAO to link the two tables. The ORMLite documentation includes a sample project featuring such a relationship, so that you can see how this is done.

Extension Task: Pokemon Finder

Sometimes I find myself in a situation where I *really* want a Pokemon that can learn a specific combination of moves. Ideally, this would be one that shares a type with any of the moves that are damage-dealing. Add in functionality to help me find my perfect "toxic, substitute, leech seed, double team" partner.

References

1. Hibernate Project, “Hibernate. Everything Data”. Available: https://hibernate.org
2. The Eclipse Foundation, “EclipseLink”. Available: https://www.eclipse.org/eclipselink/
3. The Apache Software Foundation, “Apache OpenJPA”. Available: https://openjpa.apache.org
4. J. Yang, C. Yan, P. Subramaniam, S. Lu and A. Cheung, “How not to Structure Your Database-Backed Web Applications: A Study of Performance Bugs in the Wild,” 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018, pp. 800-810, doi: 10.1145/3180155.3180194.
5. Pulse Security Ltd., “ORM, huh, What is it good for?” Available: https://pulsesecurity.co.nz/articles/ORM-SQLi
6. Halpin, T., “Object-Role Modeling: Principles and Benefits.” International Journal of Information System Modeling and Design (IJISMD), 2010, 1(1), 33-57. doi: 10.4018/jismd.2010092302
7. G. Watson, “OrmLite – Lightweight Object Relational Mapping”. Available: https://ormlite.com