Projet Home Plans : Base de données et ORM

Antoine Carpentier

November 9, 2014

1 Introduction

Dans cette section, je discuterai le choix, l'installation et la configuration de la base de données et de l'ORM

1.1 Choix

1.1.1 Base de données

J'ai choisi la base de données SQLite parce qu'elle fournit une implémentation standard et fournie du language SQL, parce qu'elle est open-source et multi-plateforme mais surtout parce que la base de données est stockée sous forme d'un fichier donc il n'y a pas besoin d'installer un serveur de base de données et tout le monde aura la même base de données pendant le développement (si on push le fichier .db).

1.1.2 ORM

J'ai choisi ORMlite comme ORM parce qu'il est compatible avec un grand nombre de base de données donc on peut changer de base de données en rajoutant juste un driver et parce qu'il est moins complexe qu'Hibernate mais aussi riche en fonctionnalités. Le seul problème est qu'il ne gère pas les relations many-to-many (*-* en UML) et qu'il faudra gérer ca nous-mêmes mais ca ne sera pas très compliqué. Il faut juste créer une classe intermédiaire comme on ferait dans une base de données.

1.2 Installation et Configuration

1.2.1 Base de données

Optionnellement (pour utiliser la base de données sans passer par Java), il faut installer sqlite3 avec le gestionnaire de paquet de votre distribution. Pour Windows, il y apparemment une DLL à installer sur http://www.sqlite.org/download.html mais je suis dans l'impossibilité de tester. Dites-moi si ca marche. Il n'y a rien à configurer avec SQLite. ORMlite créera le fichier tout seul.

1.2.2 ORM

J'ai rajouté 3 fichiers dans le dossier lib (que vous devez rajouter dans votre projet Eclipse en tant qu' External JARs)

- "ormlite-core-4.48.jar" est le jar principal de ORMlite
- "ormlite-jdbc-4.48.jar" sert à établir la connection entre ORMlite et JDBC (le gestionnaire de connexion aux base de données de Java)
- "sqlite-jdbc-3.8.7.jar" est le driver SQLite pour JDBC. C'est ce fichier qu'il faut remplacer si on veut utiliser une autre base de données

Si vous ajoutez ces 3 fichiers, normalement ca fonctionne.

1.3 Test

Dans le dossier test, j'ai rajouté 3 fichiers qui permettent de tester si la base de données et l'ORM fonctionnent. Lancez simplement TestMain et dites-moi si vous avez des erreurs. Sinon, c'est que ca fonctionne. Vous pouvez vérifier par vous-même en lançant "sqlite3 test.db" à la racine du projet et en faisant des select en SQL :

```
select * from test_orms;
select * from test_manys;
select * from test_onetoones;
```

2 Utilisation

Toutes les fonctionnalités dont je parle ici sont illustrés par des exemples dans le dossier test. ORMlite utilise les annotations Java. Il faut les placer au dessus des déclarations des classes et des membres. Je met pour chaque sous-section un lien vers la documentation d'ORMlite. Vous pouvez trouver des bons exemples en plus sur leur site `http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_7.html#Examples` Je vous invite à tout lire avant de commencer parce qu'il y a des subtilités.

2.1 Base de données

Pour obtenir la connexion à la base de données, j'ai créé une classe Database qui possède une méthode getConnectionSource() qui renvoie une connexion unique vers la base de données. Je prévois de rajouter une méthode pour charger un fichier de configuration pour pouvoir modifier facilement la configuration de la connexion. Je prévois également de la rendre thread-safe, soit en utilisant un monitor, soit en donnant une connexion à chaque thread (mais dans ce cas-la il faudra faire attention à la concurrence).

```
JdbcConnectionSource connectionSource = Database.getConnectionSource();
```

2.2 Classes et tables

Pour lier une classe à une table dans la base de données, il faut faire :

```
@DatabaseTable {tableName = "tests"}
public class Test {

}
```

Pour rajouter un membre et le lier à une colonne dans la table :

```
@DatabaseTable {tableName = "tests"}
public class Test {
        @DatabaseField
        private String aTestMember;
}
```

Pour les membres, les deux options les plus importants sont

- generatedId qui spécifie que le membre est l'identifiant unique dans la table et qu'il est incrémenté automatiquement
- canBeNull qui spécifie si le membre peut être NULL dans la base de données ou non

Donc ca pourrait donner

```
@DatabaseTable {tableName = "tests"}
public class Test {
        @DatabaseField (generatedId = true)
        private int id_test;
        @DatabaseField (canBeNull = false)
        private String aTestMember;
}
```

La doc :

- `http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_2.html#Local-Annotations`

2.3 Sauver et charger dans/de la base de données

Pour sauver et charger de la base de données, il faut un objet en plus de la classe de base Java appelé DAO (Data Access Object) qui va utiliser les annotations de la section précédente pour interagir avec la base de données. Avec ORMlite, on fait

```
// Le premier paramètre du template est la classe pour laquelle on veut
// un DAO et le deuxième est le type de l'id de cette classe
Dao<Test, Integer> daoTest = new DaoManager.createDao(connectionSource,
```

Ceci crée un DAO pour la classe Test qu'on peut ensuite utiliser :

```
Test test = new Test();
daoTest.create(test); // crée test dans la base de données
daoTest.update(test); // met a jour test dans la base de données
daoTest.delete(test); // supprime test dans la base de données
```

Plein d'autres méthodes sont disponibles pour faire des requêtes complexes et pour créer/mettre à jour/supprimer plusieurs objets en même temps.

La doc :

- http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_2.html#DAO-Setup
- http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_5.html#DAO-Methods

2.4 Créer automatiquement une table si elle n'existe pas

Pour l'instant, tout ça ne fonctionne pas parce que les tables ne sont pas crées dans la base de données. Par chance, on peut dire à ORMlite de tout créer tout seul en se basant sur les annotations. Pour cela, il suffit de faire (une seule fois dans le code)

```
TableUtils.createTableIfNotExists(connectionSource, Test.class);
```

C'est surtout utile pour le moment parce qu'on est en développement. Une fois qu'on passe en production, on enlève tout ça et on garde simplement la base de données finale.

La doc :

- http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_2.html#TableUtils

2.5 Lier les classes entres elles (foreign key dans la base de données)

Une des forces d'un ORM, c'est qu'il peut faire lui-même les liens entre les tables dans la base de données. Pour ça on lui spécifie simplement les relations entre les classes avec des annotations.

```
@DatabaseTable(tableName = "tests")
public class Test {
        @DatabaseField(generatedId = true)
        private int id_test;
        @DatabaseField(canBeNull = false)
        private String aTestMember;

        // C'est ici que ca se passe !
        @DatabaseField(canBeNull = false, foreign = true)
```

```
        private OtherTest otherTest;
}
```

Ici, la class Test possède une référence vers un objet OtherTest, tout va être créé automatiquement dans la base de données et l'objet OtherTest sera automatiquement chargé quand on le référence. Si un objet Test possède un objet OtherTest, ca veut dire qu'un même objet OtherTest peut être référencé par plusieurs objets Test différents. C'est une relation one-to-many. Donc il pourrait avoir besoin de référencer une collection de ces objets Test.

Voici le code de OtherTest

```
@DatabaseTable (tableName = "other_tests")
        @DatabaseField (generatedId = true)
        private int id_othertest;

        // C'est ici que ca se passe !
        // le paramètre eager détermine si les objets sont chargés
        // directement ou seulement à la première utilisation
        @ForeignCollectionField(eager = true)
        private ForeignCollection<Test> tests;
```

Attention, il y a une subtilité : lorsqu'on crée les tables avec TableUtils (section précédente), il faut les créer dans le bon ordre pour ne pas référencer une table qui n'existe pas encore ! Dans l'exemple, il faut créer d'abord OtherTest et puis Test puis Test référence OtherTest.

Si on veut créer une relation one-to-one (c'est à dire qu'un certain objet OtherTest ne référence qu'un objet Test et inversément, i.e deux objets Test différents ne peuvent pas référence le même objet OtherTest), il faut rajouter une contrainte unique sur la foreign key :

```
@DatabaseTable (tableName = "tests")
public class Test {
        @DatabaseField (generatedId = true)
        private int id_test;
        @DatabaseField (canBeNull = false)
        private String aTestMember;

        // C'est ici que ca se passe !
        @DatabaseField (canBeNull = false, foreign = true, unique = true
        private OtherTest otherTest;
```

}

Je verrai plus tard comment faire facilement les relations many-to-many puisque ORMlite n'offre pas de support pour ca

La doc :

- `http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_2.html#Foreign-Objects`
- `http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_2.html#Foreign-Collection`

3 To do

- Rajouter loadConfig et le threading dans Database
- Améliorer ce tuto ??
- Trouver un moyen de simplifier les relations many-to-many
- Créer des query personnalisées pour stocker les .obj en BLOB