ORM Lite Package

Version 4.3
November 2010

Gray Watson

ORMLite

Version 4.3 – November 2010

ORMLite provides a lightweight Object Relational Mapping between Java classes and SQL databases – see http://en.wikipedia.org/wiki/Object-relational_mapping. There are certainly more mature ORMs which provide this functionality including Hibernate and iBatis. However, the author wanted a simple yet powerful wrapper around the JDBC functions and Hibernate and iBatis are significantly more complicated with many dependencies.

ORMLite supports JDBC connections to MySQL, Postgres, Microsoft SQL Server, H2, Derby, HSQLDB, and Sqlite and can be extended to additional ones relatively easily. ORMLite also supports native database calls on Android OS. There are also initial implementations for DB2 and Oracle although the author needs access to each of these database types to tune the support. Contact the author if your database is not supported.

To get started quickly with ORMLite, see the code examples down in the `com.j256.ormlite.examples` package test classes in the Java sources jar down in `src/test/java`. They contain a couple different examples with working code. There is also a HTML version of this documentation – see http://ormlite.sourceforge.net/javadoc/ormlite-core/doc-files/ormlite.html.

Gray Watson http://256.com/gray/

1 Getting Started

1.1 Downloading ORMLite Jar

To get started with ORMLite, you will need to download the jar file. It is available on the central maven repository (`http://repo1.maven.org/maven2/com/j256/ormlite/`) or from Sourceforge (`http://sourceforge.net/projects/ormlite/files/`).

Users that are connecting to SQL databases via JDBC connections will need to download the ormlite-jdbc-X.X.jar file. For use with Android applications, you should download the ormlite-android-X.X.jar instead. ORMLite does not have any required direct dependencies. See the section on external dependencies for information about other packages that you may want to use. See Section 7.6 [Dependencies], page 37. The code works with Java 5 or later.

1.2 Configuring a Class

The following is an example class that is configured to be persisted to a database using ORMLite annotations. The `@DatabaseTable` annotation configures the Account class to be persisted to the database table named `accounts`. The `@DatabaseField` annotations map the fields on the Account to the database columns with the same names.

The name field is configured as the primary key for the database table by using the `id = true` annotation field. Also, notice that a no-argument constructor is needed so the object can be returned by a query. For more information see the class setup information later in the manual. See Section 2.1 [Class Setup], page 7.

```
@DatabaseTable(tableName = "accounts")
public class Account {

    @DatabaseField(id = true)
    private String name;
    @DatabaseField
    private String password;

    public Account() {
        // ORMLite needs a no-arg constructor
    }
    public Account(String name, String password) {
        this.name = name;
        this.password = password;
    }
    public String getName() {
        return name;
    }
    public String getPassword() {
        return password;
```

```
        }
    }
```

1.3 Configuring a DAO

A typical Java pattern is to isolate the database operations in Database Access Objects or DAO classes. Each DAO provides create, delete, update, etc. type of functionality and specializes in the handling a particular persisted class. To set up a DAO, you will need a DAO interface and an implementation class. ORMLite provides a base DAO interface and a base implementation class. The following is an example DAO interface corresponding to the Account class from the previous section of the manual:

```
    /** Account DAO which has a String id (Account.name) */
    public interface AccountDao extends Dao<Account,String> {
        // empty wrapper, you can add additional DAO methods here
    }
```

The following is the example implementation class:

```
    /** JDBC implementation of the AccountDao interface. */
    public class AccountDaoImpl extends BaseDaoImpl<Account,String>
     implements AccountDao {
        public AccountDaoImpl(ConnectionSource connectionSource) throws SQLException {
            super(connectionSource, Account.class);
        }
    }
```

You are not *required* to create a DAO class for every one of your persisted objects. You can use the `createDao` static method on the `BaseDaoImpl` class to create a DAO class without having to define one. For example:

```
    Dao<Account, String> accountDao =
      BaseDaoImpl.createDao(connectionSource, Account.class);
    Dao<Order, Integer> orderDao =
      BaseDaoImpl.createDao(connectionSource, Order.class);
```

More information about setting up the DOAs is available later in the manual. See Section 2.3 [DAO Setup], page 13.

1.4 Client Code Example

The client code in this section demonstrates how to use the classes presented in the previous two sections. The code uses H2 as a test database instance. You will need to add the H2 jar file to your classpath if you want to run the example as-is. *NOTE:* Android users should see the Android specific documentation later in the manual. See Chapter 5 [Use With Android], page 27.

The client performs the following steps.

- It creates a connection source which handles connections to the database.
- It instantiates a AccountDaoImpl.

- The `accounts` database table is created. This step is not needed if the table already exists.

```
public class AccountApp {

    public static void main(String[] args) throws Exception {

        // this uses h2 by default but change to match your database
        String databaseUrl = "jdbc:h2:mem:account";
        // create a connection source to our database
        ConnectionSource connectionSource =
            new JdbcConnectionSource(databaseUrl);

        // instantiate the dao
        AccountDaoImpl accountDao = new AccountDaoImpl(connectionSource);

        // if you need to create the 'accounts' table make this call
        TableUtils.createTable(connectionSource, Account.class);
```

Once we have configured our database objects, we can use them to create an Account, persist it to the database, and query for it from the database by its ID:

```
        // create an instance of Account
        Account account = new Account();
        account.setName("Jim Coakley");

        // persist the account object to the database
        // it should return 1 for the 1 row inserted
        if (accountDao.create(account) != 1) {
            throw new Exception("Failure adding account");
        }

        // retrieve the account
        Account account2 = accountDao.queryForId("Jim Coakley");
        System.out.println("Account: " + account2.getName());

        // close the connection source
        connectionSource.close();
    }
}
```

You should be able to get started using ORMLite by this point. To understand more of the functionality available with ORMLite, continue on with the next section. See Chapter 2 [Using], page 7.

2 How to Use

2.1 Setting Up Your Classes

To setup your classes to be persisted you need to do the following things:

1. Add the @DatabaseTable annotation to the top of each class. You can also use @Entity.
2. Add the @DatabaseField annotation right before each field to be persisted. You can also use @Column and others.
3. Add a no-argument constructor to each class with at least package visibility.

2.1.1 Adding ORMLite Annotations

Annotations are special code markers have have been available in Java since version 5 that provide meta information about classes, methods, or fields. To specify what classes and fields to store in the database, ORMLite supports either its own annotations (`@DatabaseTable` and `@DatabaseField`) or the more standard annotations from the `javax.persistence` package. See Section 2.1.2 [Javax Persistence Annotations], page 10. Annotations are the easiest way to configure your classes but you can also configure the class using Java code or Spring XML. See Section 7.2 [Class Configuration], page 31.

With ORMLite annotations, for each of the Java classes that you would like to persist to your SQL database, you will need to add the `@DatabaseTable` annotation right above the `public class` line. Each class marked with one of these annotations will be persisted into its own database table. For example:

```
@DatabaseTable(tableName = "accounts")
public class Account {
...
```

The `@DatabaseTable` annotations can have an optional `tableName` argument which specifies the name of the table that corresponds to the class. If not specified, the class name, all lowercase, is used by default. With the above example each `Account` object will be persisted as a row in the `accounts` table in the database. If the `tableName` was not specified, the `account` table would be used instead.

Additionally, for each of the classes, you will need to add a `@DatabaseField` annotation to each of the *fields* in the class that are to be persisted to the database. Each field is persisted as a column of a database row. For example:

```
@DatabaseTable(tableName = "accounts")
public class Account {

    @DatabaseField(id = true)
    private String name;

    @DatabaseField(canBeNull = false)
    private String password;
    ...
```

In the above example, each row in the `accounts` table has 2 columns:

- the `name` column which is a string and also is the database identity (id) of the row
- the `password` column, also a string which can not be null

The @DatabaseField annotation can have the following fields:

`columnName`
: String name of the column in the database that will hold this field. If not set then the field name, all lowercase, is used instead.

`dataType`
: The type of the field as the DataType class. Usually the type is taken from Java class of the field and does not need to be specified. This corresponds to the SQL type. See Section 2.2 [Persisted Types], page 12.

`defaultValue`
: String default value of the field when we are creating a new row in the table. Default is none.

`width`
: Integer width of array fields – usually for strings. Some databases do not support this unfortunately. Default for those that do is 255.

`canBeNull`
: Boolean whether the field can be assigned to null value. Default is true. If set to false then you must provide a value for this field on every object inserted into the database.

`id`
: Boolean whether the field is the id field or not. Default is false. Only one field can have this set in a class. Id fields uniquely identity a row. If you don't have it set then you won't be able to use the query, update, refresh, and delete by ID methods. Only one of this, `generatedId`, and `generatedIdSequence` can be specified. See Section 2.9.1 [Id Column], page 19.

`generatedId`
: Boolean whether the field is an auto-generated id field. Default is false. Only one field can have this set in a class. This tells the database to auto-generate a corresponding id for every row inserted. When an object with a generated-id is created using the `Dao.create()` method, the database will generate an id for the row which will be returned and set in the object by the create method. Some databases require sequences for generated ids in which case the sequence name will be auto-generated. To specify the name of the sequence use `generatedIdSequence`. Only one of this, `id`, and `generatedIdSequence` can be specified. See Section 2.9.2 [GeneratedId Column], page 19.

`generatedIdSequence`
: String name of the sequence number to be used to generate this value. Same as `generatedId` but you can specify the sequence name to use. Default is none. Only one field can have this set in a class. This is only necessary for databases which require sequences for generated ids. If you use generatedId instead then the code will auto-generate a sequence name. Only one of this,

`id`, and `generatedId` can be specified. See Section 2.9.3 [GeneratedIdSequence Column], page 20.

`foreign`

Boolean setting which identifies this field as corresponding to another class that is also stored in the database. Default is false. The field can not be a primitive type. The other class must have an id field (either `id`, `generatedId`, or `generatedIdSequence`) which will be stored in this table. When an object is returned from a query call, any foreign objects will *just* have the id field set. See Chapter 4 [Foreign Objects], page 25.

`useGetSet`

Boolean that says that the field should be accessed with get and set methods. Default is false which instead uses direct field access via Java reflection. This may be necessary if the object you are storing has protections around it.

NOTE: The name of the get method *must* match getXxx() where Xxx is the name of the field with the first letter capitalized. The get *must* return a class which matches the field's exactly. The set method *must* match setXxx(), have a single argument whose class matches the field's exactly, and return void. For example:

```
@DatabaseField(useGetSet = true)
private Integer orderCount;

public Integer getOrderCount() {
  return orderCount;
}

public void setOrderCount(Integer orderCount) {
  this.orderCount = orderCount;
}
```

`unknownEnumName`

If the field is a Java enumerated type then you can specify the name of a enumerated value which will be used if the value of a database row is not found in the enumerated type. If this is not specified and a database row *does* contain an unknown name or ordinal value then a `SQLException` is thrown when the row is being read from the database. This is useful to handle backwards compatibility when handling out-of-date database values as well as forwards compatibility if old software is accessing up-to-date data or if you have to roll a release back.

`throwIfNull`

Boolean that tells ORMLite to throw an exception if it sees a null value in a database row and is trying to store it in a primitive field. By default it is false. If it is false and the database field is null, then the value of the primitive will be set to 0 (false, null, etc.). This can only be used on a primitive field.

`persisted`
: Set this to be false (default true) to not store this field in the database. This is useful if you want to have the annotation on all of your fields but turn off the writing of some of them to the database.

`format`
: This allows you to specify format information of a particular field. Right now only the Date fields support it when a default value is being converted or if you are using the `JAVA_DATE_STRING` type.

`unique`
: Adds a constraint to the field that it has to be unique across all rows in the table. This allows you to have a unique field in the table even though it is not the id field. For example, you might have an Account class which has a generated account-id but you also want the email address to be unique across all Accounts.

2.1.2 Using javax.persistence Annotations

Instead of using the ORMLite annotations (see Section 2.1.1 [Local Annotations], page 7), you can use the more standard annotations from the `javax.persistence` package. In place of the `@DatabaseTable` annotation, you can use the `javax.persistence` `@Entity` annotation. For example:

```
@Entity(name = "accounts")
public class Account {
...
```

The `@Entity` annotations can have an optional `name` argument which specifies the table name. If not specified, the class name all lowercase is used by default.

Instead of using the `@DatabaseField` annotation on each of the fields, you can use the `javax.persistence` annotations: `@Column`, `@Id`, `@GeneratedValue`, `@OneToOne`, and `@ManyToOne`. For example:

```
@Entity(name = "accounts")
public class Account {

    @Id
    private String name;

    @Column(nullable = false)
    private String password;
    ...
```

The following `javax.persistence` annotations and fields are supported:

`@Column`
: Specifies the field to be persisted to the database. You can also just specify the `@Id` annotation. The following annotation fields are supported, the rest are ignored.

`name`
: Used to specify the name of the associated database column. If not provided then the field name is taken.

`length`
: Specifies the length (or width) of the database field. Maybe only applicable for Strings and only supported by certain database types. Default for those that do is 255. Same as the `width` field in the `@DatabaseField` annotation.

`nullable`
: Set to true to have a field not be able to be inserted into the database with a null value. Same as the `canBeNull` field in the `@DatabaseField` annotation.

`unique`
: Adds a constraint to the field that it has to be unique across all rows in the table. Same as the `unique` field in the `@DatabaseField` annotation.

`@Id`
: Used to specify a field to be persisted to the database as a primary row-id. If you want to have the id be auto-generated, you will need to also specify the `@GeneratedValue` annotation.

`@GeneratedValue`
: Used to define an id field as having a auto-generated value. This is only used in addition to the `@Id` annotation. See the `generatedId` field in the `@DatabaseField` annotation for more details.

`@OneToOne or @ManyToOne`
: Fields with these annotations are assumed to be foreign fields. See Chapter 4 [Foreign Objects], page 25. ORMLite does *not* enforce the many or one relationship nor does it use any of the annotation fields. It just uses the existence of either of these annotations to indicate that it is a foreign object.

If the `@Column` annotation is used on a field that has a unknown type then it is assumed to be a Serializable field and the object should implement `java.io.Serializable`. See [serializable], page 13.

2.1.3 Adding a No-Argument-Constructor

After you have added the class and field annotations, you will also need to add a no-argument constructor with *at least* package visibility. When an object is returned from a query, ORMLite constructs the object using Java reflection and a constructor needs to be called.

```
Account() {
    // all persisted classes must define a no-arg constructor
    // with at least package visibility
```

```
}
```

So your final example Account class with annotations and constructor would look like:

```
@DatabaseTable(tableName = "accounts")
public class Account {

    @DatabaseField(id = true)
    private String name;

    @DatabaseField(canBeNull = false)
    private String password;
    ...

    Account() {
        // all persisted classes must define a no-arg constructor
        // with at least package visibility
    }
    ...
}
```

2.2 Persisted Data Types

The following Java types can be persisted to the database by ORMLite. Database specific code helps to translate between the SQL types and the database specific handling of those types. See Section 2.5 [Database Type], page 15.

`String (DataType.STRING)`
: Persisted as SQL type VARCHAR.

`boolean` or `Boolean (DataType.BOOLEAN` or `DataType.BOOLEAN_OBJ)`
: Persisted as SQL type BOOLEAN.

`java.util.Date (DataType.JAVA_DATE, DataType.JAVA_DATE_LONG, or JAVA_DATE_STRING)`
: Persisted as SQL type TIMESTAMP. *NOTE:* This is a different class from `java.sql.Date`. *NOTE:* Certain databases only provide seconds resolution so the milliseconds will be 0.

  You can also specify the `dataType` field to the `@DatabaseField` annotation as a `DataType.JAVA_DATE_LONG` in which case the milliseconds value of the `Date` will be stored as an LONG. Or you can use `DataType.JAVA_DATE_STRING` in which case the date will be stored as a string in `yyyy-MM-dd HH:mm:ss.SSSSSS` format. You can use the `format` field in `DatabaseField` to set the date to another format.

  NOTE: Because of reentrant issues with `SimpleDateFormat`, synchronization is done every time a JAVA_DATE_STRING date is converted to/from the database.

`byte` or `Byte (DataType.BYTE` or `DataType.BYTE_OBJ)`
: Persisted as SQL type TINYINT.

`short` or `Short` (`DataType.SHORT` or `DataType.SHORT_OBJ`)
: Persisted as SQL type SMALLINT.

`int` or `Integer` (`DataType.INTEGER` or `DataType.INTEGER_OBJ`)
: Persisted as SQL type INTEGER.

`long` or `Long` (`DataType.LONG` or `DataType.LONG_OBJ`)
: Persisted as SQL type BIGINT.

`float` or `Float` (`DataType.FLOAT` or `DataType.FLOAT_OBJ`)
: Persisted as SQL type FLOAT.

`double` or `Double` (`DataType.DOUBLE` or `DataType.DOUBLE_OBJ`)
: Persisted as SQL type DOUBLE.

`Serializable` (`DataType.SERIALIZABLE`)
: Persisted as SQL type VARBINARY. This is a special type that serializes an object as a sequence of bytes and then deserializes it on the way back. The field must be an object that implements the `java.io.Serializable` interface. Depending on the database type, there will be limits to the size of the object that can be stored. YMMV.

`enum` or `Enum` (`DataType.ENUM_STRING` or `DataType.ENUM_INTEGER`)
: Persisted by default as the enumerated value's string *name* as a VARCHAR type. You can also specify the `dataType` field to the `@DatabaseField` annotation as a `DataType.ENUM_INTEGER` in which case the `ordinal` of the enum value will be stored as an INTEGER. The name is the default (and recommended) because it allows you to add additional enums anywhere in the list without worrying about having to convert data later. If you insert (or remove) an enum from the list that is being stored as a number, then old data will be un-persisted incorrectly.

  You can also also specify an *unknownEnumName* name with the `@DatabaseField` annotation which will be used if an unknown value is found in the database. See [unknownEnumName], page 9.

NOTE: ORMLite also supports the concept of foreign objects where the id of another object is stored in the database. See Chapter 4 [Foreign Objects], page 25.

2.3 Setting Up the DAOs

Once you have annotated your classes, you will need to create the DAO class(es). The pattern that we recommend is to define an interface which extends the `Dao` interface and will be used in the code. The interface isn't required but it is a good pattern so your code is less tied to JDBC for persistence. Each DAO has two generic parameters: the class we are persisting with the DAO, and the class of the ID-column that will be used to identify a specific database row. If you class does not have an ID field, you can put `Object` or `Void` as the 2nd argument. For example, in the above `Account` class, the `"name"` field is the ID column (id = true) so the ID class is `String`. Example:

```
/** Account DAO which has a String id (Account.name) */
public interface AccountDao extends Dao<Account, String> {
```

```
        // empty wrapper, you can add additional DAO methods here
    }
```

Here's the example implementation of this interface.

```
/** JDBC implementation of the AccountDao interface. */
public class AccountDaoImpl extends BaseDaoImpl<Account, String>
  implements AccountDao {
    public AccountDaoImpl(ConnectionSource connectionSource) throws SQLException {
        super(connectionSource, Account.class);
    }
}
```

That's all you need to define your DAO classes. You are free to add more methods to your DAO interfaces and implementations if there are specific operations that are needed and not provided by the Dao base classes. More on how to use these DAOs later. See Section 2.8 [DAO Usage], page 17.

As mentioned above (see Section 1.3 [Starting DAO], page 4), you are not *required* to create a DAO class for every one of your persisted objects. You can use the `createDao` static method on the `BaseDaoImpl` class to create a DAO class without having to define one.

2.4 JDBC Connection Sources

NOTE: With regards to connection sources, Android users should see the Android specific documentation later in the manual. See Chapter 5 [Use With Android], page 27.

To use the database and the DAO objects, you will need to configure what JDBC calls a `DataSource` (see the `javax.sql.DataSource` class). The `DataSource` is a factory for connections to the physical SQL database. Since ORMLite also supports non-JDBC connections, we use a subset of the `DataSource` methods in the `ConnectionSource` interface so we can support non-JDBC database handlers. Here is a code example that creates a simple, single-connection source.

```
// single connection source example
ConnectionSource connectionSource =
  new JdbcConnectionSource("jdbc:h2:mem:account");
```

One of the more recent additions (10/2010) to the package is the `JdbcPooledConnectionSource`. It is a relatively simple implementation of a pooled connection source. As database connections are released, instead of being closed they are added to an internal list so they can be reallocated at a later time. New connections are created on demand only if there are no dormant connections that can be reused. `JdbcPooledConnectionSource` is also synchronized and can be used my multiple threads. It has settings for the maximum number of free connections before they are closed as well as a maximum age before a connection is closed.

```
// single connection source example
JdbcPooledConnectionSource connectionSource =
  new JdbcPooledConnectionSource("jdbc:h2:mem:account");
// only keep the connections open for 5 minutes
```

```
connectionSource.setMaxConnectionAgeMillis(5 * 60 * 1000);
```

There are many other, external data sources that can be used instead, including more robust and probably higher-performance pooled connection managers. You can instantiate your own directly and wrap it in the `DataSourceConnectionSource` class which delegates to it. Then you set that on the DAOs directly.

```
// basic Apache data source
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:h2:mem:account");
// we wrap it in the DataSourceConnectionSource
ConnectionSource connectionSource =
  new DataSourceConnectionSource(dataSource);
```

When you are done with your `ConnectionSource`, you will want to call a `close()` method to close any underlying connections. Something like the following pattern is recommended.

```
JdbcConnectionSource connectionSource =
    new JdbcPooledConnectionSource("jdbc:h2:mem:account");
try {
    // work with the data-source and DAOs
    ...
} finally {
    connectionSource.close();
}
```

Unfortunately, the `DataSource` interface does not have a close method so you will have to close the `DataSource` by hand – the `close()` method on the `DataSourceConnectionSource` does *nothing*.

2.5 Database Type

ORMLite uses an internal `DatabaseType` object which defines all of the per-database information necessary to support the various features on all of the different database types. The `JdbcConnectionSource` uses the database URL to pick the correct `DatabaseType`. If it picks an incorrect one then you may need to set the `DatabaseType` on the connection source *directly*. For example:

```
String databaseUrl = "jdbc:derby://dbserver1:1527/";
DatabaseType databaseType = new DerbyClientServerDatabaseType();
ConnectionSource connectionSource =
   new JdbcConnectionSource(databaseUrl, databaseType);
```

Android users do not need to worry about this because the `AndroidConnectionSource` always uses the `SqliteAndroidDatabaseType`. See Chapter 5 [Use With Android], page 27.

For more information about the database specific code in the DatabaseType see later in the manual. See Section 7.3 [Database Type Details], page 33.

2.6 Supported Databases

ORMLite supports the following database flavors. Some of them have some specific documentation that needs to be obeyed.

MySQL

Tables are created in MySQL with the InnoDB engine by default using `CREATE TABLE ... ENGINE=InnoDB`. If you want to use another engine, you can instantiate the `MysqlDatabaseType` directly and use the `setCreateTableSuffix()` method to use the default or another engine. Also, MySQL does some funky stuff with the last-modification time if a `Date` is defined as a `TIMESTAMP` so `DATETIME` was used instead.

Postgres

Microsoft SQL Server
H2

Derby

There are two drivers for Derby: one embedded and one client/server. Right now we will only detect and use the embedded driver. You will have to set the `DerbyClientServerDatabaseType` on your ConnectionSource directly. See Section 2.5 [Database Type], page 15.

HSQLDB

SQLite

There are multiple SQLite drivers out there. Make sure you use the Xerial one (`http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC`) and not the Zentus one (`http://www.zentus.com/sqlitejdbc/`) which does not support generated ids.

Android SQLite

Android's SQLite database is accessed through direct calls to the Android database API methods.

DB2

I do not have access to an DB2 database so I cannot run any tests to make sure that my support for it works well. Please contact me if you have an Oracle database that I can develop against.

Oracle

I do not have access to an Oracle database so I cannot run any tests to make sure that my support for it works well. Please contact me if you have an Oracle database that I can develop against.

Please contact the author if your database is not supported.

2.7 Tying It All Together

So you have annotated the objects to be persisted, added the no-arg constructor, defined your DAO classes, created your `DataSource`, and established your `DatabaseType`. You are ready to start persisting and querying your database objects. The following code ties it all together:

```
// h2 by default but change to match your database
String databaseUrl = "jdbc:h2:mem:account";
JdbcConnectionSource connectionSource =
  new JdbcConnectionSource(databaseUrl);

// instantiate the dao
AccountDaoImpl accountDao = new AccountDaoImpl(connectionSource);

// if you need to create the 'accounts' table make this call
TableUtils.createTable(connectionSource, Account.class);

// create an instance of Account
Account account = new Account("Jim Coakley");

// persist the account object to the database
// it should return 1 for the 1 row inserted
if (accountDao.create(account) != 1) {
  // error handling ...
}
// other code ...

// destroy the data source which should close underlying connections
connectionSource.destroy();
```

For more examples, see the `com.j256.ormlite.examples` package test classes in the Java -jdbc sources jar.

2.8 DAO Usage

The following database operations are easily accomplished by using the DAO classes:

create and persist an object to the database
: This inserts a new row to the database table associated with the object.

```
Account account = new Account();
account.name = "Jim Coakley";
// only 1 row should have been affected
if (accountDao.create(account) != 1) {
  // error handling ...
}
```

query for it's id column

If the object has an id field defined by the annotations, then we can lookup an object in the database using its id.

```
Account account = accountDao.queryForId(name);
if (account == null) {
  account not found handling ...
}
```

update the database row associated with the object

If you change fields in an object in memory, you must call update to persist those changes to the database. This also requires an id field.

```
account.password = "_secret";
// 1 row should be updated
if (accountDao.update(account) != 1) {
  // error handling ...
}
```

refreshing our object if the database has changed

If some other entity has changed a row the database corresponding to an object in memory, you will need to refresh that object to get the memory object up-to-date. This also requires an id field.

```
// 1 row should be found
if (accountDao.refresh(account) != 1) {
  // error handling ...
}
```

delete the account from the database

Removes the row that corresponds to the object from the database. Once the object has been deleted from the database, you can continue to use the object in memory but any update or refresh calls will fail. This also requires an id field.

```
// 1 row should be affected
if (accountDao.delete(account) != 1) {
  // error handling ...
}
```

iterate through all of the rows in a table:

The DAO is also an iterator so you can easily run through all of the rows in the database:

```
// page through all of the accounts in the database
for (Account account : accountDao) {
    System.out.println(account.getName());
}
```

NOTE: you must page through *all* items for the iterator to close the underlying SQL object. If you don't go all of the way, the garbage collector will close the SQL statement some time later which is considered bad form.

2.9 Identity Columns

Database rows are identified by a particular column which is defined as the *identity* column. This can either be supplied by the user or auto-generated by the database. Identity columns have unique values for every row in the table and they are required if you want to delete, refresh, or update a particular row using the DAO. To configure a field as an identity field, you should use one (and only one) of the following three settings from `@DatabaseField`: `id`, `generatedId`, or `generatedIdSequence`.

2.9.1 Fields With id

With our `Account` example class, the string `name` field has been marked with `id = true`. This means that the `name` is the identity column for the object. Each account stored in the database must have a unique value for the `name` field – you cannot have two rows with the name "John Smith".

```
public class Account {
    @DatabaseField(id = true)
    private String name;
    ...
}
```

When you use the DAO to lookup an account with a particular name, you will use the identity field to locate the `Account` object in the database:

```
Account account = accountDao.queryForId("John Smith");
if (account == null) {
    // the name "John Smith" does not match any rows
}
```

If you need to change the value of an object's id field, you must use the `Dao.updateId()` method which takes the current object still with its *old* id value and the new value. ORMLite has to first locate the object by its old id and then update it with the new id. See [updateId], page 35.

2.9.2 Fields With generatedId

You can configure a long or integer field to be a *generated* identity column. The id number column for each row will then be automatically generated by the database.

```
public class Order {
    @DatabaseField(generatedId = true)
    private int id;
    ...
}
```

When an `Order` object is passed to `create` and stored to the database, the generated identity value is returned by the database and set on the object by ORMLite. In the majority of database types, the generated value starts at 1 and increases by 1 every time a new row is inserted into the table.

```
// build our order object without and id
Order order = new Order("Jim Sanders", 12.34);
...
if (orderDao.create(order) != 1) {
   // error handling unless 1 row was inserted
}
System.out.println("Order id " + order.getId() + " was persisted to the database");
// query for the order with an id of 1372
order = orderDao.queryForId(1372);
if (order == null) {
    // none of the order rows have an id of 1372
}
```

In the above code example, an order is constructed with name and amount (for example). When it is passed to the DAO's `create` method, the id field has not been set. After it has been saved to the database, the generated-id will be set on the id field by ORMLite and will be available when `getId()` is called on the order after the `create` method returns.

NOTE: Depending on the database type, you may not be able to change the value of an auto-generated id field.

2.9.3 Fields With generatedIdSequence

Some databases use what's called a sequence number generator to provide the generated id value. If you use `generatedId = true` with those databases, a sequence name will be auto-generated by ORMLite. If, however, you need to set the name of the sequence to match existing schema, you can used the `generatedIdSequence` value which takes a string name for the sequence.

```
public class Order {
    @DatabaseField(generatedIdSequence = "order_id_seq")
    private int id;
    ...
}
```

In the above example, the `id` value is again automatically generated but using a sequence with the name `order_id_seq`. This will throw an exception if you are working with a database which does not support sequences.

NOTE: Depending on the database type, you may not be able to change the value of an auto-generated id field.

3 Custom Statement Builder

The DOAs have methods to query for an object that matches an id field (`queryForId`) as well as query for all objects (`queryForAll`) and iterating through all of the objects in a table (`iterator`). However, for more specified queries, there is the `queryBuilder()` method which returns a `QueryBuilder` object for the DAO with which you can construct custom queries to return a sub-set of the table.

3.1 Query Builder Basics

Here's how you use the query builder to construct custom queries. First, it is a good pattern to set the column names of the fields with Java constants so you can use them in queries. For example:

```
@DatabaseTable(tableName = "accounts")
public class Account {
    public static final String PASSWORD_FIELD_NAME = "password";

...
    @DatabaseField(canBeNull = false, columnName = PASSWORD_FIELD_NAME)
    private String password;
...
```

This allows us to construct queries using the password field name without having the renaming of a field in the future break our queries. This should be done *even* if the name of the field and the column name are the same.

```
// get our query builder from the DAO
QueryBuilder<Account, String> queryBuilder =
  accountDao.queryBuilder();
// the 'password' field must be equal to "qwerty"
queryBuilder.where().eq(Account.PASSWORD_FIELD_NAME, "qwerty");
// prepare our sql statement
PreparedQuery<Account, String> preparedQuery =
  queryBuilder.prepare();
// query for all accounts that have that password
List<Account> accountList = accountDao.query(preparedQuery);
```

You get a `QueryBuilder` object from the `Dao.queryBuilder` method, call methods on it to build your custom query, call `queryBuilder.prepare()` which returns a `PreparedQuery` object, and then pass the `PreparedQuery` to the DAO's `query` or `iterator` methods.

3.2 Building Queries

There are a couple of different ways that you can build queries. The `QueryBuilder` has been written for ease of use as well for power users. Simple queries can be done linearly:

```
QueryBuilder<Account, String> queryBuilder =
  accountDao.queryBuilder();
```

```
// get the WHERE object to build our query
Where where = queryBuilder.where();
// the name field must be equal to "foo"
where.eq(Account.NAME_FIELD_NAME, "foo");
// and
where.and();
// the password field must be equal to "_secret"
where.eq(Account.PASSWORD_FIELD_NAME, "_secret");
PreparedQuery<Account, String> preparedQuery =
  queryBuilder.prepare();
```

The SQL query that will be generated from the above example will be approximately:

```
SELECT * FROM account
  WHERE (name = 'foo' AND password = '_secret')
```

If you'd rather chain the methods onto one line (like StringBuilder), this can also be written as:

```
queryBuilder.where()
  .eq(Account.NAME_FIELD_NAME, "foo")
  .and()
  .eq(Account.PASSWORD_FIELD_NAME, "_secret");
```

If you'd rather use parenthesis to group the comparisons properly then you can call:

```
Where where = queryBuilder.where();
where.and(where.eq(Account.NAME_FIELD_NAME, "foo"),
          where.eq(Account.PASSWORD_FIELD_NAME, "_secret"));
```

All three of the above call formats produce the same SQL. For complex queries that mix ANDs and ORs, the last format may be necessary to get the grouping correct. For example, here's a complex query:

```
Where where = queryBuilder.where();
where.or(
  where.and(
    where.eq(Account.NAME_FIELD_NAME, "foo"),
    where.eq(Account.PASSWORD_FIELD_NAME, "_secret")),
  where.and(
    where.eq(Account.NAME_FIELD_NAME, "bar"),
    where.eq(Account.PASSWORD_FIELD_NAME, "qwerty")));
```

This produces the following approximate SQL:

```
SELECT * FROM account
  WHERE ((name = 'foo' AND password = '_secret')
         OR (name = 'bar' AND password = 'qwerty'))
```

The `QueryBuilder` also allows you to set what specific select columns you want returned, specify the 'ORDER BY' and 'GROUP BY' fields, and various other SQL features (LIKE, IN, >, >=, <, <=, <>, IS NULL, DISTINCT, ...). See the javadocs on`QueryBuilder` and `Where` classes for more information. A good SQL reference site can be found at http://www.w3schools.com/Sql/.

3.3 Building Statements

The DAO can also be used to construct custom `UPDATE` and `DELETE` statements. Update statements are used to change certain fields in rows from the table that match the `WHERE` pattern – or update *all* rows if no `where()`. Delete statements are used to delete rows from the table that match the `WHERE` pattern – or delete *all* rows if no `where()`.

For example, if you want to update the passwords for all of the Accounts in your table that are currently null to the string "none", then you might do something like the following:

```
UpdateBuilder<Account, String> updateBuilder =
  accountDao.updateBuilder();
// update the password to be "none"
updateBuilder.updateColumnValue("password", "none");
// only update the rows where password is null
updateBuilder.where().isNull(Account.PASSWORD_FIELD_NAME);
accountDao.update(updateBuilder.prepare());
```

With update, you can also specify the update value to be an expression:

```
// update hasDogs boolean to true if dogC > 0
updateBuilder.updateColumnExpression(
  "hasDogs", "dogC > 0");
```

To help you construct your expressions, you can use the UpdateBuilder's escape methods `escapeColumnName` and `escapeValue` can take a string or a `StringBuilder`. This will protect you if columns or values are reserved words.

If, instead, you wanted to delete the rows in the Accounts table whose password is currently null, then you might do something like the following:

```
DeleteBuilder<Account, String> deleteBuilder =
  accountDao.deleteBuilder();
// only delete the rows where password is null
deleteBuilder.where().isNull(Account.PASSWORD_FIELD_NAME);
accountDao.delete(deleteBuilder.prepare());
```

3.4 Using Select Arguments

The arguments that are used in WHERE operations can be specified directly as value arguments (as in the above examples) or as a `SelectArg` object. SelectArgs are used to set the value of an argument at a later time – they generate a SQL '?'. For example:

```
QueryBuilder<Account, String> queryBuilder =
  accountDao.queryBuilder();
Where where = queryBuilder.where();
SelectArg selectArg = new SelectArg();
// define our query as 'name = ?'
where.eq(Account.NAME_FIELD_NAME, selectArg);
// prepare it so it is ready for later query or iterator calls
PreparedQuery<Account, String> preparedQuery =
  queryBuilder.prepare();
```

```
// later we can set the select argument and issue the query
selectArg.setValue("foo");
List<Account> accounts = accountDao.query(preparedQuery);
// then we can set the select argument to another
// value and re-run the query
selectArg.setValue("bar");
accounts = accountDao.query(preparedQuery);
```

NOTE: `SelectArg` objects have protection against being used in more than one column name. You must instantiate a new object if you want to use a `SelectArg` with another column.

4 Foreign Object Fields

ORMLite supports the concept of "foreign" objects where one or more of the fields correspond to an object are persisted in another table in the same database. For example, if you had an `Order` objects in your database and each `Order` had a corresponding `Account` object, then the `Order` object would have foreign `Account` field. With foreign objects, *just* the id field from the `Account` is persisted to the `Order` table as the column `"account_id"`. For example, the `Order` class might look something like:

```
@DatabaseTable(tableName = "orders")
public class Order {

    @DatabaseField(generatedId = true)
    private int id;

    @DatabaseField(canBeNull = false, foreign = true)
    private Account account;
    ...
}
```

When the `Order` table was created, something like the following SQL would be generated:

```
CREATE TABLE `orders`
   (`id` INTEGER AUTO_INCREMENT , `account_id` INTEGER,
    PRIMARY KEY (`id`));
```

Notice that the name of the field is *not* `account` but is instead `account_id`. You will need to use this field name if you are querying for it. You can set the column name using the `columnName` field in the `DatabaseField` annotation. See [columnName], page 8.

When you query for an order, you will get an `Order` object with an account field object that *only* has its id field set – all of the fields in the foreign `Account` object will have default values (null, 0, false, etc.). If you want to use other fields in the `Account`, you must call `refresh` on the `accountDao` class to get the `Account` object filled in. For example:

```
if (accountDao.refresh(order.getAccount()) != 1) {
  // error handling ...
}
```

NOTE: Because we use refresh, foreign objects are therefor *required* to have an id field.

5 Using With Android

Because of the lack of support for JDBC in Android OS, ORMLite makes direct calls to the Android database APIs to access SQLite databases. You should make sure that you have downloaded and are depending on the `ormlite-android.jar` file and *not* the `ormlite-jdbc.jar` version. This part of the code is the newest and although tested and used in some projects, the proper patterns on *how* to use it have not solidified. Feedback on this would be most welcome.

After you have read the getting started section (see Chapter 1 [Getting Started], page 3), the following instructions should be followed to help you get ORMLite working under Android OS.

1. You will need to create your own database helper class which should probably extend the `OrmLiteSqliteOpenHelper` class. This class creates and upgrades the database when your app is installed and can also provide the DAO classes used by your other classes. Your helper class must implement the methods `onCreate(SQLiteDatabase sqliteDatabase, ConnectionSource connectionSource)` and `onUpgrade(SQLiteDatabase database, ConnectionSource connectionSource, int oldVersion, int newVersion)`. `onCreate` creates the database when your app is first installed while `onUpgrade` handles the upgrading of the database tables when you upgrade your app to a new version. There is a sample `DatabaseHelper` class as well as examples projects online: `http://ormlite.sourceforge.net/android/examples/`.
2. The helper can be kept open across all activities in your app with the same Sqlite database connection reused by all threads. If you open multiple connections to the same database, stale data and unexpected results may occur. We recommend using the `OpenHelperManager` to monitor the usage of the helper – it will create it on the first access, track each time a part of your code is using it, and then it will close the last time the helper is released.
3. The `OpenHelperManager` will by default look for the full class name of your own database helper class in the `open_helper_classname` value defined in `res/values/strings.xml`. You can instead set a `SqliteOpenHelperFactory` on the manager directly in a `static {}` block in your code.
4. Once you have defined your database helper and are managing it correctly, you will need to use it in your `Activity` classes. An easy way to use the `OpenHelperManager` is to extend `OrmLiteBaseActivity` for each of your activity classes – there is also `OrmLiteBaseService` and `OrmLiteBaseTabActivity`. These classes provide a `getHelper()` method to access the database helper whenever it is needed and will automatically release the helper in the `onDestroy()` method. There is a sample `HelloAndroid` activity class in the the sample section of the tests and on the web site along with a `SampleData` example class.
5. If you do not want to extend the base classes then you will need to duplicate their basic functionality of calling `OpenHelperManager.getHelper(Context context)` at the start of your code, save the helper and use it as much as you want, and then calling `OpenHelperManager.release()` when you are done with it.
6. The Android native SQLite database type is `SqliteAndroidDatabaseType` and is used by the base classes internally.

Please see the example code posted to the website for more information: `http://ormlite.sourceforge.net/android/examples/`. Again, feedback on this is welcome.

6 Upgrading Old Versions

6.1 Upgrade from Version 3.X to 4.0

Removed any outside usage of the `DatabaseType` since the `ConnectionSource` now provides it. Also added features to be able to prepare update and delete statements. To provide type safety, we've moved back to using `QueryBuilder` so we can have `UpdateBuilder` and `DeleteBuilder`. And instead of a `PreparedStmt` there is `PreparedQuery`, `PreparedUpdate`, and `PreparedDelete`. Here are the details:

- We have removed most of the cases where the user has to deal with the `DatabaseType`. All you need to set on the DAOs is the `ConnectionSource` which provides the database type internally. To create and drop the tables, also, you only need the `ConnectionSource`.
- Constructing a `BaseDaoImpl` now self-initializes if it is constructed with a `ConnectionSource`. This validates the class configurations meaning that it now throws a `SQLException`.
- Constructing a `JdbcConnectionSource` or `DataSourceConnectionSource` also now throw a `SQLException` since they also now self-initialize if they are constructed with the URL. This creates the internal database type and loads the driver class for it.
- Deprecated the `createJdbcConnectionSource` method in the `DatabaseTypeUtils` and turned the `loadDriver` method into a no-op. You now just instantiate the `JdbcConnectionSource` directly and there is no need for referencing the `DatabaseTypeUtils` anymore.
- `Dao.statementBuilder()` method changed (back) to `Dao.queryBuilder()`.
- `Dao.queryBuilder()` returns a `QueryBuilder` instead of a `StatementBuilder`.
- You now call `distinct()` and `limit()` on the `QueryBuilder`. Unfortunately, there are no deprecated methods for them on the `StatementBuilder`.
- You now call `selectColumns()` on the `QueryBuilder` instead of `columns()` since now we have columns also in the `UpdateBuilder`. Unfortunately, there are no deprecated methods for them on the `StatementBuilder`.
- You call `QueryBuilder.prepare()` instead of `StatementBuilder.prepareStatement()`. It returns a `PreparedQuery` instead of a `PreparedStmt`. You pass a `PreparedQuery` into the `Dao.query()` and `Dao.iterator()` methods instead of a `PreparedStmt`.
- We removed the `DatabaseTypeFactory` class since it was no longer needed for Spring configurations.
- Removed `BaseJdbcDao` since it had been deprecated in 3.X.

6.2 Upgrade from Version 2.X to 3.2

The 3.2 release involved a *very* large code reorganization and migration. The project was basically split into 3 pieces: core functionality, JDBC database handlers, and the new Android handler. With significant help from Kevin G, we abstracted all of the database calls into 3 interfaces: `ConnectionSource` (like a `DataSource`), `DatabaseConnection` (like a `Connection`) and `DatabaseResults` (like a `ResultSet`). Once we had the interfaces in place, we wrote delegation classes for JDBC and Android handlers. This means that as of 3.X we release 3 packages: ormlite-core (for developers), ormlite-jdbc (for people connecting to JDBC databases), and ormlite-android (for Android users). Both the JDBC and Android packages include all of the core code as well.

Along the way a number of specific changes were made to the methods and classes:

- Since we split off the JDBC, we renamed the `BaseJdbcDao` to be `BaseDaoImpl` in the core package. You will need to adjust any DAOs that you have.
- We are in the process of allowing custom delete and update methods so we took the major upgrade opportunity to rename the `QueryBuilder` object to be `StatementBuilder`.
- Because of the above, we also renamed `Dao.queryBuilder()` method to be `queryBuilder()`.
- Also renamed the `PreparedQuery` object to be `PreparedStmt`.
- One of the big changes for those of you using an external JDBC `DataSource` is that you no longer set it on the DAO directly. You need to wrap your `DataSource` in a `DataSourceDatabaseConnection` wrapper class which gets set on the DAO instead.

There were no on-disk changes unless you somehow managed to get ORMLite working previously on Android. Since we were using JDBC before to do the data marshalling and now are doing it by hand, some of the data representations may have changed. Sorry for the lack of detail here.

6.3 Upgrade from Version 2.3 to 2.4

A bug was fixed in 2.4 with how we were handling Derby and Hsqldb. Both of these databases seem to be capitalizing table and field names in certain situations which meant that customized queries of ORMLite generated tables were affected. In version 2.4, all tables and field names are capitalized in the SQL generated for Derby and Hsqldb databases. This means that if you have data in these databases from a pre 2.4 version, the 2.4 version will not be able to find the tables and fields without renaming to be uppercase.

7 Advanced Concepts

7.1 Spring Configuration

ORMLite contains some classes which make it easy to configure the various database classes using the Spring framework. For more information about the Spring Framework, see http://www.springsource.org/.

`TableCreator`
: Spring bean that auto-creates any tables that it finds DAOs for if the system property `ormlite.auto.create.tables` has been set to true. It will also auto-drop any tables that were auto-created if the property `ormlite.auto.drop.tables` has been set to true. This should be used carefully and probably only in tests.

Here's an example of a full Spring configuration.

```
<!-- URL used for database, probably should be in properties file -->
<bean id="databaseUrl" class="java.lang.String">
    <!-- we are using the in-memory H2 database in this example -->
    <constructor-arg index="0" value="jdbc:h2:mem:account" />
</bean>

<!-- datasource used by ORMLite to connect to the database -->
<bean id="connectionSource"
    class="com.j256.ormlite.jdbc.JdbcConnectionSource"
    init-method="initialize">
    <property name="url" ref="databaseUrl" />
    <!-- probably should use system properties for these too -->
    <property name="username" value="foo" />
    <property name="password" value="bar" />
</bean>

<!-- abstract dao that is common to all defined daos -->
<bean id="baseDao" abstract="true" init-method="initialize">
    <property name="connectionSource" ref="connectionSource" />
</bean>

<!-- our daos -->
<bean id="accountDao"
    class="com.j256.ormlite.examples.common.AccountDaoImpl"
    parent="baseDao" />
```

7.2 Class Configuration

The simplest mechanism for configuring a class to be persisted by ORMLite is to use the `@DatabaseTable` and `@DatabaseField` annotations. See Section 2.1.1 [Local Annotations],

page 7. However if you do not own the class you are persisting or there are permission problems with the class, you may want to configure the class using Java code instead.

To configure a class in code, you use the `DatabaseFieldConfig` and `DatabaseTableConfig` objects. The field config object holds all of the details that are in the `@DatabaseField` annotation as well as the name of the corresponding field in the object. The `DatabaseTableConfig` object holds the class and the corresponding list of `DatabaseFieldConfigs`. For example, to configure the `Account` object using Java code you'd do something like the following:

```
List<DatabaseFieldConfig> fieldConfigs =
    new ArrayList<DatabaseFieldConfig>();
fieldConfigs.add(new DatabaseFieldConfig("name", null, DataType.UNKNOWN,
    null, 0, false, false, true, null, false, null, false));
fieldConfigs.add(new DatabaseFieldConfig("password", null,
    DataType.UNKNOWN, null, 0, false, false, false, null, false, null,
    false));
DatabaseTableConfig<Account> accountTableConfig
    = new DatabaseTableConfig<Account>(Account.class, fieldConfigs);

AccountDaoImpl accountDao =
    new AccountDaoImpl(connectionSource, accountTableConfig);
```

See the Javadocs for the `DatabaseFieldConfig` class for the fields to pass to the constructor. You can also use the no-argument constructor and call the setters for each field. You use the setters as well when you are configuring a class using Spring wiring. Here is the above example in Spring:

```
<bean id="accountTableConfig"
  class="com.j256.ormlite.table.DatabaseTableConfig">
    <property name="dataClass"
        value="com.j256.ormlite.examples.common.Account" />
    <property name="tableName" value="account" />
    <property name="fieldConfigs">
        <list>
            <bean class="com.j256.ormlite.field.DatabaseFieldConfig">
                <property name="fieldName" value="name" />
                <property name="id" value="true" />
            </bean>
            <bean class="com.j256.ormlite.field.DatabaseFieldConfig">
                <property name="fieldName" value="password" />
                <property name="canBeNull" value="false" />
            </bean>
        </list>
    </property>
</bean>
```

7.3 Database Specific Code

ORMLite isolates the database-specific code in the `DatabaseType` classes found in `com.j256.ormlite.db`. Each of the supported databases has a class there which implements the code needed to handle the unique features of the database (H2DatabaseType, MySqlDatabaseType, etc.). If you want to help develop and test against other SQL databases, a externally available server that the author could connect to and test against would be appreciated. Please contact the author if your database is not supported or if you want to help.

The following methods are currently used by the system to isolate the database specific behavior in one place. See the javadocs for the `DatabaseType` class for more information.

`getDriverUrlPart`
: Return the part in the database URI which identifies the particular database. Usually the URI is in the form jdbc:XXX:... where XXX is the driver url part.

`getDriverClassName`
: Returns the class name of the driver that may or may not be in the classpath depending on what database is being used.

`loadDriver`
: Load the driver class associated with this database so it can wire itself into JDBC.

`appendColumnArg`
: Takes a field type and appends the SQL necessary to create the field. It may also generate arguments for the end of the table create statement or commands that must run before or after the table create.

`convertColumnName`
: Convert and return the column name for table and sequence creation. Often this is necessary to fix case issues.

`dropColumnArg`
: Takes a field type and adds all of the commands necessary to drop the column from the database.

`appendEscapedEntityName`
: Add a entity-name (table or column name) word to the SQL wrapped in the proper characters to escape it. This avoids problems with table, column, and sequence-names being reserved words.

`appendEscapedWord`
: Add the word to the string builder wrapped in the proper characters to escape it. This avoids problems with data values being reserved words.

`generateIdSequenceName`
: Return the name of an ID sequence based on the table-name and the field-type of the id. This is required by some database types when we have generated ids.

`getCommentLinePrefix`
: Return the prefix to put at the front of a SQL line to mark it as a comment.

`isIdSequenceNeeded`
: Return true if the database needs a sequence when you insert for generated IDs. Some databases handle generated ids internally.

`getFieldConverter`
: Return the field converter associated with a particular field type. This allows the database instance to convert a field as necessary before it goes to the database.

`isVarcharFieldWidthSupported`
: Return true if the database supports the width parameter on VARCHAR fields.

`isLimitSupported`
: Return true if the database supports the LIMIT sql command.

`isLimitAfterSelect`
: Return true if the LIMIT should be called after SELECT otherwise at the end of the WHERE (the default).

`appendLimitValue`
: Add the necessary SQL to limit the results to a certain number.

`appendSelectNextValFromSequence`
: Add the SQL necessary to get the next-value from a sequence. This is only necessary if isIdSequenceNeeded returns true.

`appendCreateTableSuffix`
: Append the SQL necessary to properly finish a CREATE TABLE line.

`isCreateTableReturnsZero`
: Returns true if a 'CREATE TABLE' statement should return 0. False if > 0.

`isEntityNamesMustBeUpCase`
: Returns true if table and field names should be made uppercase. This is an unfortunate "feature" of Derby and Hsqldb. See the Javadocs for the class for more information.

7.4 DAO Methods

The DAO classes provide the following methods that you can use to store your objects to your database. This list may be out of date. See the `Dao` interface class for the latest methods.

`queryForId(ID id)`
: Looks up the id in the database and retrieves an object associated with it.

`queryForFirst(PreparedQuery<T> preparedQuery)`
: Query for and return the first item in the object table which matches a prepared statement. This can be used to return the object that matches a single unique column. You should use queryForId if you want to query for the id column.

`queryForAll()`
Query for all of the items in the object table and return a list of them. For medium sized or large tables, this may load a lot of objects into memory so you should consider using the `iterator` method instead.

`queryForAllRaw(String query)`
Query for all of the items in the object table that match the SQL select query argument. This method allows you to do special queries that aren't supported otherwise. For medium sized or large tables, this may load a lot of objects into memory so you should consider using the `iteratorRaw` method instead.

`queryBuilder()`
Create and return a new `QueryBuilder` object which allows you to build a custom query. See Section 3.1 [QueryBuilder Basics], page 21.

`updateBuilder()`
Create and return a new `UpdateBuilder` object which allows you to build a custom update statement. See [updateBuilder], page 23.

`deleteBuilder()`
Create and return a new `DeleteBuilder` object which allows you to build a custom delete statement. See [deleteBuilder], page 23.

`query(PreparedQuery<T> preparedQuery)`
Query for the items in the object table which match a prepared statement. See Chapter 3 [Statement Builder], page 21. This returns a list of matching objects. For medium sized or large tables, this may load a lot of objects into memory so you should consider using the `iterator` method instead.

`create(T data)`
Create a new entry in the database from an object. Should return 1 indicating 1 row was inserted.

`update(T data)`
Save the fields from an object to the database. If you have made changes to an object, this is how you persist those changes to the database. You cannot use this method to update the id field – see updateId. This should return 1 since 1 row was updated.

`updateId(T data)`
Update an object in the database to change its id to a new id. The data *must* have its current id set and the new-id is passed in as an argument. After the id has been updated in the database, the id field of the data object will also be changed. This should return 1 since 1 row was updated.

`update(PreparedUpdate<T> preparedUpdate)`
Update objects that match a custom update statement.

`refresh(T data, ID newId)`
Does a query for the object's id and copies in each of the field values from the database to refresh the data parameter. Any local object changes to persisted fields will be overwritten. If the database has been updated this brings your local object up-to-date. This should return 1 since 1 row was retrieved.

`delete(T data)`
Delete an object from the database. This should return 1 since 1 row was removed.

`delete (Collection<T> datas)`
Delete a collection of objects from the database using an IN SQL clause. This returns the number of rows that were deleted.

`deleteIds(Collection<ID> ids)`
Delete the objects that match the collection of ids from the database using an IN SQL clause. This returns the number of rows that were deleted.

`delete(PreparedDelete<T> preparedDelete)`
Delete objects that match a custom delete statement.

`iterator`
This method satisfies the `Iterable` Java interface for the class and allows you to iterate through the objects in the table using SQL. This method allows you to do something like:

```
for (Account account : accountDao) { ... }
```

WARNING: See the `Dao` class for warnings about using this method.

`iterator(PreparedQuery<T> preparedQuery)`
Same is the `iterator` method but with a prepared statement parameter. See Chapter 3 [Statement Builder], page 21.

`iteratorRaw(String query)`
Same as the prepared statement iterator except it takes a raw SQL select statement argument. This is the iterator version of the `queryForAllRaw` method. Although you should use the `iterator` method for most queries, this method allows you to do special queries that aren't supported otherwise. Like the above iterator methods, you must call close on the returned `RawResults` object once you are done with it.

7.5 ORMLite Logging

ORMLite uses a log system which can plug into Apache commons logging, Log4j, Android Log, or its own internal log implementations. The logger code in `com.j256.ormlite.logger` first looks for the `org.apache.commons.logging.LogFactory` class in the classpath – if found it will use Apache commons logging. If that class is not found it then looks for `org.apache.log4j.Logger` and if found will use Log4j. Next it looks for `android.util.Log` and if found will use the Android internal logger. If none of these classes are available it will use an internal logger – see `LocalLog`. The logger code also provides simple `{}` argument expansion like `slf4j` which means that you can save on `toString()` calls and `StringBuilder` operations if the log level is not high enough. This allows me to do something like the following:

```
private static Logger logger =
  LoggerFactory.getLogger(QueryBuilder.class);
...
```

```
logger.debug("built statement {}", statement);
```

If you are using log4j (through Apache commons logging or directly), you can use something like the following as your log4j.properties file to see details about the SQL calls.

```
log4j.rootLogger=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# print the date in ISO 8601 format
log4j.appender.stdout.layout.ConversionPattern=%d{ISO8601} [%p] %c{1} %m%n

# be more verbose with our code
log4j.logger.com.j256.ormlite=DEBUG

# to enable logging of arguments to all of the SQL calls
# uncomment the following line
#log4j.logger.com.j256.ormlite.stmt.mapped.BaseMappedStatement=TRACE
```

Notice that you can uncomment the last line in the above log4j.properties file to log the arguments to the various SQL calls. This may expose passwords or other sensitive information in the database so probably should only be used during debugging and should not be the default.

7.6 External Dependencies

ORMLite does not have any direct dependencies. It has logging classes that depend on Apache commons-logging and Log4j but these classes will *not* be referenced unless they exist in the classpath.

If you want to get the ORMLite Junit tests to run, there are test dependencies on the following packages:

`javax.persistence`
: For testing the compatibility annotations `@Column` and the like.

`org.junit`
: We use Junit for our unit tasks.

`org.easymock.easymock`
: We use, and are in love with, EasyMock. `http://easymock.org/`. It allows us to mock out dependencies so we can concentrate on testing a particular class instead of the whole package.

`com.h2database`
: As a test database implementation, H2 is very fast and simple to use. Not as recommended as a production level database.

`org.apache.log4j`
: For logging to files using the `log4j.properties` config. In the log4j package, you can exclude the following dependencies: `com.sun.jmx.jmxri`,

`com.sun.jdmk.jmxtools`, `javax.activation.activation`, `javax.jms.jms`, `javax.mail.mail`.

8 Contributions

There are a couple of people that I'd like to thank who helped with the project.

Kevin Galligan
: Kevin was the impetus and the author of a good bit of the Android compatible code. He wrote the Android level support classes and did a ton of beta-testing of it. He's also provided all of the Android examples. Thanks much Kevin. See `http://www.kagii.com/`.

Jim Gloor
: Jim was one of the Java gurus at a previous company. Thanks much for his JDBC code samples that started this effort.

Nelson Erb
: Nelson is our self-appointed documentation and testing volunteer. He did a great job summarizing sections of this document so we could create a better 'Getting Started' section. He will also be working on tests.

Robert Adamsky
: I worked with Robert at a company where he laid out our entire DAO and hibernate class hierarchy. The `DAO` interface and the `BaseDaoImpl` where in some part modeled after his code. Thanks dude.

Thanks much to them all.

Index of Concepts

S

T

U

W

Table of Contents