ORM Lite Package

Version 2.3 June 2010 Copyright 2010 by Gray Watson.

Published by Gray Watson

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the chapter entitled "Copying" are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the chapter entitled "Copying" may be included in a translation approved by the author instead of in the original English.

ORMLite 1

ORMLite

Version 2.3 – June 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 natively MySQL, Postgres, Microsoft SQL Server, H2, Derby, HSQLDB, and Sqlite and can be extended to additional ones relatively easily. 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/doc-files/ormlite.html.

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

1 Getting Started

1.1 External Dependencies

ORMLite depends on the following external packages. You must make sure that these are listed in your Maven, Ant, or . . . configuration files.

org.apache.commons.logging

For logging, ORMLite has a loose wrapper around apache commons-logging. You can remove this dependency by editing com.j256.ormlite.logger.Logger. As of this writing, we use version 1.1.1 of commons logging. Previous or later versions should also work.

javax.persistence

To provide compatibility with other persistence packages, ORMLite depends on the JavaX persistence annotations. As of this writing, we use version 1.0 of the persistence package.

For those maven-ites, here is the section from the pom that you will need to mirror.

```
properties>
   <ormlite-version>2.3</ormlite-version>
    <persistence-api-version>1.0</persistence-api-version>
    <commons-logging-version>1.1.1//commons-logging-version>
</properties>
<dependencies>
    <dependency>
       <groupId>com.j256
       <artifactId>ormlite</artifactId>
       <version>${ormlite-version}</version>
   </dependency>
    <dependency>
       <groupId>javax.persistence</groupId>
       <artifactId>persistence-api</artifactId>
       <version>${persistence-api-version}</version>
    </dependency>
    <dependency>
       <groupId>commons-logging
       <artifactId>commons-logging</artifactId>
       <version>${commons-logging-version}</version>
   </dependency>
</dependencies>
```

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

org.junit

We use Junit for our unit tasks. Pretty simple/easy.

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 fast and easy 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: com.sun.jmx.jmxri, com.sun.jdmk.jmxtools, javax.activation.activation, javax.jms.jms, javax.mail.mail.

1.2 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 our 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 with at least package permissions.

1.2.1 Adding ORMLite Annotations to Your Classes

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 1.2.2 [Javax Persistence Annotations], page 7. Annotations are the easiest way to configure your classes but you can also configure the class using Java code or Spring XML. See Section 5.2 [Class Configuration], page 24.

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 <code>QDatabaseTable</code> annotation right above the <code>public class</code> 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.

Additionally, for each of the classes, you will need to add a <code>QDatabaseField</code> 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;
}
```

The @DatabaseField annotation can have the following fields:

columName

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.

jdbcType

The type of the field as the JdbcType 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 1.3 [Persisted Types], page 9.

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 or have no value. Default is true.

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.

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. 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.

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.

foreign

Boolean setting which identifies this field as corresponding to another class that is also stored in the database. Default is false. The field must 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 in it. See Chapter 4 [Foreign Objects], page 21.

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
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 row *does* contain an unknown name or ordinal value then a SQLException is thrown. 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 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. This can only be used on a primitive field.

1.2.2 Using javax.persistence Annotations Instead

Instead of using the ORMLite annotations (see Section 1.2.1 [Local Annotations], page 4), 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 <code>QDatabaseField</code> annotation on each of the fields, you can use the <code>javax.persistence</code> annotations: <code>QColumn</code>, <code>QId</code>, <code>QGeneratedValue</code>, <code>QOneToOne</code>, and <code>QManyToOne</code>. 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.

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 21. 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 Section 1.3 [Persisted Types], page 9.

1.2.3 Adding a No Argument Constructor to Your Class

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
}
```

1.3 Types That can be Persisted

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 5.3 [Database Type], page 25.

String

Persisted as SQL type VARCHAR.

Boolean or boolean

Persisted as SQL type BOOLEAN.

java.util.Date

Persisted as SQL type TIMESTAMP. *NOTE:* This is *not* java.sql.Date which is a different class.

Byte or byte

Persisted as SQL type TINYINT.

Short or short

Persisted as SQL type SMALLINT.

Integer or int

Persisted as SQL type INTEGER.

Long or long

Persisted as SQL type BIGINT.

Float or float

Persisted as SQL type FLOAT.

Double or double

Persisted as SQL type DOUBLE.

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 <code>java.io.Serializable</code> interface. Depending on the database type, there will be limits to the size of the object that can be stored. YMMV.

Enum or enum

Persisted by default as the enumerated value's string name as a VARCHAR type. You can also specify the jdbcType field to the @DatabaseField annotation as a JdbcType.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.

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 Section 1.2.1 [Local Annotations], page 4.

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 21.

1.4 Defining the DAOs

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 usually specialize in the handling a particular persisted class.

Once you have annotated your classes, you will need to create the DAO class(es). The pattern that we recommend is to define a DAO interface which extends the BaseDao 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 BaseDao<Account, String> {
    // empty wrapper, you can add additional DAO methods here
}
```

The implementation of this interface takes a DatabaseType object in its constructor which identifies the per-database flavor class. See Section 5.3 [Database Type], page 25. Example:

```
/** JDBC implementation of the AccountDao interface. */
public class AccountJdbcDao extends BaseJdbcDao<Account, String>
  implements AccountDao {
    public AccountJdbcDao(DatabaseType databaseType) {
        super(databaseType, 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 Chapter 2 [DAO Usage], page 13.

1.5 JDBC Data Sources

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. Here is a code example that creates a simple data source.

```
// single connection data source example
DataSource dataSource =
  DatabaseTypeUtils.createSimpleDataSource("jdbc:h2:mem:account");
```

There are many other data sources that can be used instead, including pooled connections which will create new connections on demand and re-use existing dormant connections. You can instantiate your own without the DatabaseTypeUtils and set it on the DAOs directly.

When you are done with your DataSource, you will want to call the close() or destroy() method to close any underlying java.sql.Connection objects. Something like the following pattern is recommended.

```
SimpleDataSource dataSource = null;
try {
    dataSource = new SimpleDataSource();
    dataSource.setURL("jdbc:h2:mem:account");
    dataSource.setUsername("billy");
    dataSource.setPassword("_secret");
    // work with the data-source and DAOs
    ...
} finally {
    if (dataSource != null) {
        dataSource.close();
    }
}
```

1.6 Database Connections

ORMLite works with a DatabaseType object which defines all of the per-database information necessary to support the various features on all of the different database types. See Section 5.3 [Database Type], page 25. To instantiate a DatabaseType, you can either use the DatabaseTypeUtils class or call your database class directly. You can pass in the database URL or pass in an already created DataSource object:

instantiate a DatabaseType class directly

```
DatabaseType databaseType = new H2DatabaseType();
```

After you instantiate your DatabaseType, you will need to call the loadDriver() method to make sure that the driver has wired itself into JDBC appropriately.

```
DatabaseType databaseType = new H2DatabaseType();
databaseType.loadDriver();
```

For more information about the database specific code in the DatabaseType. See Section 5.3 [Database Type], page 25.

1.7 Tying It All Together

So you have annotated the objects to persist, added the no-arg constructor, defined your DAO classes, established your DatabaseType, and created your DataSource. 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";
SingleConnectionDataSource dataSource =
  DatabaseTypeUtils.createSimpleDataSource(databaseUrl);
DatabaseType databaseType =
  DatabaseTypeUtils.createDatabaseType(dataSource);
databaseType.loadDriver();
// instantiate the dao
AccountJdbcDao accountDao = new AccountJdbcDao(databaseType);
accountDao.setDataSource(dataSource);
// _must_ make this call after the setters
accountDao.initialize();
// if you need to create the 'accounts' table make this call
TableUtils.createTable(databaseType, dataSource, Account.class);
// create an instance of Account
Account account = new Account();
account.name = "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 ...
}
// destroy the data source which should close underlying connections
dataSource.destroy();
```

NOTE: as you see in the above example, if you are instantiating a DAO class outside of Spring then you will need to call initialize() method to properly initialize the class.

For more examples, see the com.j256.ormlite.examples package test classes in the Java sources jar down in src/test/java.

2 Using the DAOs

2.1 Basic 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, 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 upto-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. 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.

2.2 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

Looks up the id in the database and retrieves an object associated with it.

queryForFirst

Query for and return the first item in the object table which matches a prepared query. 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, you should consider using the iterator method instead.

queryBuilder

Create and return a new QueryBuilder object which allows you to build a custom query. See Chapter 3 [Query Builder], page 17.

query

Query for the items in the object table which match a prepared query. See Chapter 3 [Query Builder], page 17. This returns a list of matching objects. For medium sized or large tables, you should consider using the iterator method instead.

create

Create a new entry in the database from an object. Should return 1 indicating 1 row was inserted.

update

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

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.

refresh

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

Delete an object from the database. This should return 1 since 1 row was removed.

delete (collection)

Delete a collection of objects from the database using an IN SQL clause. This returns the number of rows that were deleted.

deleteIds

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.

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 (prepared query)

Same is the iterator method but with a prepared query parameter. See Chapter 3 [Query Builder], page 17.

3 Custom Query 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). For more specified queries, there is also a 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 QueryBuilder from the DAO
QueryBuilder<Account, String> queryBuilder = accountDao.queryBuilder();
// set the WHERE to: the 'password' field must be equal to "qwerty"
queryBuilder.where().eq(Account.PASSWORD_FIELD_NAME, "qwerty");
PreparedQuery<Account, String> preparedQuery =
   queryBuilder.prepareQuery();
// query for all accounts that have that password
List<Account> accountList = accountDao.query(preparedQuery);
```

You get a QueryBuilder object from DAO, call methods on it to build your custom query, call queryBuilder.prepareQuery() which returns a PreparedQuery object, and then pass the PreparedQuery to the 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.prepareQuery();
  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 columns you want returned, specify the 'ORDER BY' and 'GROUP BY' fields, and various other SQL features (LIKE, IN, >, >=, <, <=, <>, IS NULL, ...). 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 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.prepareQuery();
// 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 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(canNull = 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'));
```

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 can use a refresh call to set all of the fields in the Account object. 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 Advanced Concepts

5.1 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/.

DatabaseTypeFactory

This factory class is used for Spring injections of the database types to the DAOs and other classes. Often, the databaseUrl parameter is provided by a system property.

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.

```
factory-method="getDatabaseType" />
<!-- used to get driver-class name out of database-type factory -->
<bean id="driverClassName" class="java.lang.String"</pre>
    factory-bean="databaseTypeFactory"
    factory-method="getDriverClassName" />
<!-- datasource used by ORMLite to connect to the database -->
<bean id="dataSource"</pre>
    class="com.j256.ormlite.support.SimpleDataSource"
    init-method="initialize">
    cproperty name="url" ref="databaseUrl" />
    <!-- probably should use system properties for these too -->
    cproperty name="username" value="foo" />
    cproperty name="password" value="bar" />
</bean>
<!-- abstract dao that is common to all defined daos -->
<bean id="baseDao" abstract="true" init-method="initialize">
    cproperty name="dataSource" ref="dataSource" />
    cproperty name="databaseType" ref="databaseType" />
</bean>
<!-- our daos -->
<bean id="accountDao"</pre>
    class="com.j256.ormlite.examples.common.AccountJdbcDao"
    parent="baseDao" />
```

5.2 Class Configuration

The simplest mechanism for configuring a class to be persisted by ORMLite is to use the @DatabaseTable and @DatabaseField annotations. See Section 1.2.1 [Local Annotations], page 4. 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, JdbcType.UNKNOWN,
    null, 0, false, false, true, null, false, null, false));
fieldConfigs.add(new DatabaseFieldConfig("password", null,
    JdbcType.UNKNOWN, null, 0, false, false, false, null, false, null,
    false));
```

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

5.3 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.

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 tableName and the fieldType 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.

5.4 ORMLite Logging

ORMLite uses a loose wrapper around Apache commons logging that provides simple {} argument expansion like slf4j. You can use something like the following as your log4j.properties file to see details about the SQL calls.

```
log4j.rootLogger=INFO, stdout
```

```
\label{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
```

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.

[#] uncomment the following line

[#]log4j.logger.com.j256.ormlite.stmt.mapped.BaseMappedStatement=TRACE

Index of Concepts 29

Index of Concepts

@	database access object
@Column	database connection
@DatabaseField 4	database driver load
@DatabaseTable	database sequences
@Entity 7	database specific code
@GeneratedValue	database type
©Id	DatabaseField annotation 4
@ManyToOne 7 @OneToOne 7	databases supported
@One roone	DatabaseTable annotation 4
	DatabaseTypeFactory
\mathbf{A}	DatabaseTypeUtils
annotations	Date9
arguments to queries	default value
author	delete multiple objects
auto create tables	delete object by id
auto drop tables	deleting an object
auto-generated id 5	destroy data source
	double
D	drop tables
В	drop tables
BaseDao interface	
boolean	E
building queries	Ľ
byte	Entity annotation
	Enum9
\mathbf{C}	enumerated name unknown
	enumerated types
can be null	examples of code
chain query methods	
close data source	
closing an iterator 14 code examples 1, 12	F
Column annotation	
column name	field access using getters and setters 6
comparisons	field type 5
complex query	field width
configuration with Spring	float
configure a class in code	foreign object refreshing
connection pooling	foreign objects 6, 21
constructor with no args 8	
create tables	
creating a database row	G
creating an object	. 1.1
custom query builder	generated id
	generated id sequence
D	Generated Value annotation
	get and set method usage 6
DAO	group by
dao interface	
dao methods	
DAO usage	H
data source	
data source, simple	hibernate

1	primitive null values 6
ibatis 1 Id annotation 7 id column 10 id field 5 in 18 initialize 12 int 9 introduction 1 is null 18 iterating through all rows 13 iterator 13, 15	Q 18 query arguments 18 query builder 14, 17 query for all 14 query for first 14 query for id 13, 14 query for objects 14
J	refreshing an object
java annotations	refreshing foreign objects
javax.persistence7jdbc dao implementation10jdbc type5	S saving an object
L length of field 5, 7 like 18 load database driver 11 log4j properties file 27 logging information 27 logging sql arguments 27 long 9	select arguments 18 sequences 5 Serializable 9 short 9 simple data source 10 spring 12 spring examples 23 spring framework 23 spring wire a class 25 sql? 18
M	sql argument logging
ManyToOne annotation	String
N	${f T}$
name of database column5no argument constructor8null value allowed5, 7null values and primitives6	throwIfNull
nun varues and primitives	\mathbf{U}
O object relational mapping	unknownEnumName6, 9update an object id14updating an object13, 14usage example11useGetSet6using get and set methods6using the DAOs13
Persist objects	
persist objects 9 persisted types 9 persisting an object 13 pooled connections 10 prepared query 17	W where

Table of Contents

1	Gett	ing Started 3
	1.1	External Dependencies
	1.2	Setting Up Your Classes
		 1.2.1 Adding ORMLite Annotations to Your Classes 4 1.2.2 Using javax.persistence Annotations Instead 6 1.2.3 Adding a No Argument Constructor to Your Class
	1.3	Types That can be Persisted 8
	1.4	Defining the DAOs
	1.5	JDBC Data Sources
	1.6	Database Connections
	1.7	Tying It All Together
2	Usin	g the DAOs
	2.1	Basic DAO Usage
	2.2	DAO Methods
3	Cust	com Query Builder
	3.1	Query Builder Basics
	3.2	Building Queries
	3.3	Using Select Arguments
4	Fore	ign Object Fields
5	Adv	anced Concepts
	5.1	Spring Configuration
	5.2	Class Configuration
	5.3	Database Specific Code
	5.4	ORMLite Logging
In	dex o	f Concepts