Chapter 34
Scala Style Database Access

34.1 Introduction

The last chapter introduced how Scala can use JDBC to access a database. However, there are a number of different Scala-based approaches being developed to explore more Scala-like ways of interacting with a database than JDBC. Although JDBC works very well it is far more Java like than Scala like and actually provides a very low level of abstraction (witness the many Object Relational Mapping, or ORM, tools within the Java world such as Hibernate).

The list of possible approaches includes

- SLICK (previously Scalaquery)
- Querulous
- Squeryl
- O/R Broker

We will look at each of these briefly in the remainder of this chapter.

34.2 Slick

Scala language-integrated connection kit (SLICK) provides database query and access facilities for Scala. It attempts to treat data stored in the database as if it was just another Scala collection. This means that using Slick can be (almost) as easy as working with standard Scala collections.

The aim of SLICK is that it abstracts (hides) the lower level JDBC or SQL calls. Thus an object of a specific type can extend the concept of a table (and indicate the name of the table in the underlying database). Methods on this object can then be used to add rows to the table, search the table, etc.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_34

The following simple code snippet illustrates these ideas:

```
object Coffees extends Table[(String, Int, Double)]("COFFEES") {
def name = column[String]("COF_NAME", O.PrimaryKey)
def supID = column[Int]("SUP_ID")
def price = column[Double]("PRICE")
}
```

The object Coffee relates to a table called COFFEES in a database. This table has three columns (COF_NAME, SUP_ID and PRICE). Based on these columns four methods are provided called name, supID and price which act as accessors. This object can now be used to insert new rows:

```
Coffees.insertAll(
   ("Colombian", 101, 7.99),
   ("Colombian_Decaf", 101, 8.99),
   ("French_Roast_Decaf", 49, 9.99)
)
```

And to query for values currently held by the table

```
val q = for {
   c <- Coffees if c.supID === 101
} yield (c.name, c.price)
```

34.3 Querulus

Querulus is a open-source Scala project from Twitter. Querulus still provides direct access to SQL but avoids a number of the basic frustrations of using JDBC. For example, the basic usage of Querulus is very simple. Having imported the core QueryEvaluator class from Querulus you can write:

```
val queryEvaluator = QueryEvaluator("host", "username", "password")
```

Once you have a queryEvaluator you can then use this to run a select statement:

```
val users = queryEvaluator.select("SELECT * FROM employee")
{ row =>  new User(row.getInt("id"),
row.getString("name")) }
```

or to insert or update values in the database:

```
queryEvaluator.execute
("INSERT INTO users VALUES (?, ?)", 1, "Jacques")
```

You can also group database operations into transactional units:

```
queryEvaluator.transaction { transaction =>
   transaction.select("SELECT ... FOR UPDATE", ...)
   transaction.execute(
         "INSERT INTO users VALUES (?, ?)", 1, "John")
   transaction.execute(
      "INSERT INTO users VALUES (?, ?)", 2, "Denise") }
```

34.4 Squeryl

Style-wise, Squeryl sits midway between SLICK, which hides SQL behind Scala comprehensions as much as possible, and Querulous which uses SQL strings directly.

Squeryl provides an SQL-like domain-specific language (DSL), which gives you type safety and aims to ensure that if a statement will compile then it will not fail at runtime.

To illustrate the core concepts let us look at another example. Here we have two classes Author and Book that map to information in tables in the database. For the class Book an annotation @Column is being used to map AUTHOR_ID to the field authorId.

```
import org.squeryl.PrimitiveTypeMode._


class Author(var id: Long,
             var firstName: String,
             var lastName: String)
class Book(var id: Long,
             var title: String,
             @Column("AUTHOR_ID") var authorId: Long,
             var coAuthorId: Option[Long]) {
    def this() = this(0,"",0,Some(0L))
}
```

The object Library maps these classes to the appropriate database tables. In this case the Author class is being mapped to the table Authors and the Book class is mapped to a table called Book (note if the names are the same they do not need to be repeated).

```
object Library extends Schema {
    //When the table name doesn't match the class name, it
is specified here :
     val authors = table[Authors]("AUTHORS")
     val books = table[Book]
}
```

The following provides a basic example of the use of Squeryl to insert some values into the books table. All interaction is via a session instance with a `using` block. The authors are added to the books table. A query is then run to retrieve all authors called hunt.

```
classOf["org.postgresql.Driver")]

val conn =
     java.sql.DriverManager.getConnection("jdbc:postgresql:/
     /localhost:5432/squeryl", "squeryl", "squeryl")

val session = Session.create(conn, new PostgreSqlAdapter)

//Squeryl database interaction is within a 'using' block :
import Library._

using(session) {
     books.insert(new Author(1, "Michel","Folco"))
     val a = from(authors)(a=> where(a.lastName === "Hunt") select(a))
}
```

34.5 O/R Broker

The final approach we will look at is called O/R Broker. It is not an Object Relational mapping tool (although the name might imply this). Instead it is a JDBC library for Scala that aims to hide most of the JDBC API and wrap it up into a more Scala-like interface.

Extractors are used to *extract* information from the database. Extractors are declarative, written in Scala. They can be reused in other queries that fit the expectation of the extractor. A very simple extractor is shown below:

```
object CustomerExtractor extends RowExtractor[Customer] {
 // Construct object from row.
 def extract(row: Row) = {
  val id = row.integer("ID").get
  val name = row.string("NAME").get
  new Customer(id, name)
 }
 }
```

This is then connected to a query by declaring a named Token:

```
 val SelectCustomer =
      Token('selectCustomer, CustomerExtractor)
```

This can then be executed via the broker:

```
val customer = broker.readOnly() { session =>
  session.selectOne(SelectCustomer, "custID"->1234)
 }
```

References

SLICK, from typesafe
http://slick.typesafe.com/

Squeryl
http://squeryl.org/

Querulous
https://github.com/twitter/querulous

O/R broker
http://code.google.com/p/orbroker/