Chapter 35
Slick: Functional Relational Mapping for Scala

35.1 Introduction

In this chapter, we will look at Slick. Slick (or in full Scala Language-integrated Connection Kit) is a library that provides access to relational databases from a Scala application. It is heavily based on the functional programming paradigm and as such is much more aligned with the Scala way or programming than the rather procedural approach of JDBC.

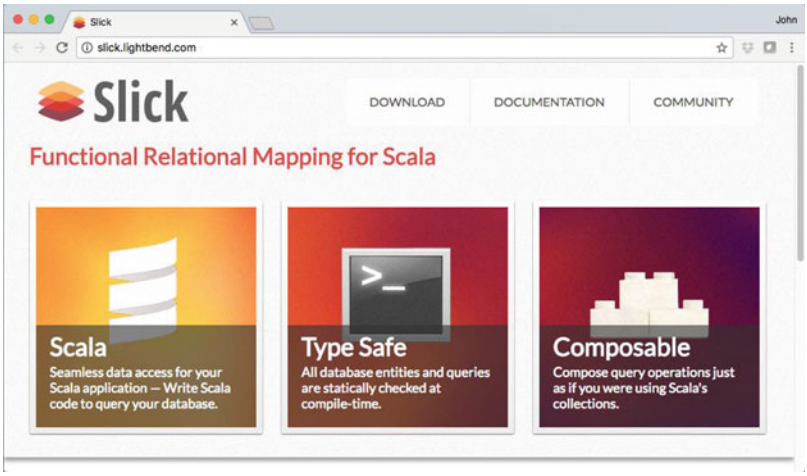
The approach is often referred to now as Functional Relational Mapping (or FRM for short). Slick's aim is to provide a solution to database access that allows regular operations over collections to be performed against a database (with of course a strong emphasis on type safety).

It should be noted that at the time of writing Slick is still a relatively young library and although the changes from one version to another have become less onerous (the changes from version 2.x to 3.x where quite extensive) backwards compatibility may still be an issue.

35.2 Obtaining Slick

Information on Slick can be found at www.slick.lightbend.com.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_35



Slick can be obtained in several ways from http://slick.lightbend.com/download. The easiest way is to follow the download link and download the slick <version>.jar file and then add it to IntelliJ as a library for your project.

However, in my case I have used Maven to set up a Scala project within IntelliJ to use Slick. The advantage of this is that the transitive dependencies (such as the libraries that Slick depends upon) are handled by Maven.

My Maven POM file has the following dependencies:

```
<dependency>
  <groupId>com.typesafe.slick</groupId>
  <artifactId>slick_2.12</artifactId>
  <version>3.2.1</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-nop</artifactId>
  <version>1.6.4</version>
</dependency>
<dependency>
  <groupId>com.typesafe.slick</groupId>
  <artifactId>slick-hikaricp_2.12</artifactId>
  <version>3.2.1</version>
</dependency>

<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-library</artifactId>
  <version>2.12.4</version>
</dependency>
```

Note that in this chapter we are also using MySQL as the database we will be connecting to. We therefore also need to add the MySQL libraries to the list of Maven dependencies:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>6.0.6</version>
</dependency>
```

We are now ready to get started.

35.3 Connecting to a Database

You can use Slick to create tables and populate data within a database (schema). To do this it is first necessary to create the database itself. This can be done in several ways, in our case I used MySQL Workbench and created the schema interactively. The schema/database is called company and can be created using

CREATE DATABASE 'company'

We can now set up the Slick database link. This can be done in several ways:

Database URL. A database URL is a JDBC style URL of the format jdbc:<protocol>:<protocol-specific-arguments>. This is supported by Database.forURL. The driver to use can also be specified using this method.

Via a Datasource. A DataSource object can be used to provide the database link. This is useful if an application framework is being used that configures the data source externally to your application. This is supported by Database.forDataSource.

Using a Config file. Another approach is to use a configuration file (called application.conf) that can be loaded from the classpath. This file can contain driver and data source information. This is supported via the Database.forConfig (config-name-string) method. For example, given the application.conf file:

```
mydb = {
  dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
  properties = {
    databaseName = "mydb"
    user = "myuser"
    password = "secret"
  }
  numThreads = 10
}
```

This file could be loaded using

val db = Database.forConfig(``mydb'')

In the following example we use the Database.forURL approach in which we specify the database URL and the database driver in a similar manner to that used with JDBC:

```
val db = Database.forURL(
  "jdbc:mysql://localhost:3306/company?" +
                "user=user&password=user123",
  driver="com.mysql.cj.jdbc.Driver")
```

Note that the Database object manages a connection pool used to connect to the database and a thread pool to execute operations. It is therefore very important that it is shut down/closed appropriately when it is no longer needed. This can be done using

```
db.close
```

35.4 Mapping Tables to Scala Types

In many cases the management of the tables within a database may occur outside of the Scala application. However, if not then Slick can automatically create the table structure for you.

The first thing required to create type-safe queries is a definition of a Table class that will represent the tables in your database. This class defines the structure of the table (its columns and the types of those columns) and how they map into the types within your application.

In the following example a case class User is mapped to a table USERS via a class Users (which extends the Table type):

```
package com.jjh.scala.db

import slick.jdbc.MySQLProfile.api._

case class User(id: Int, name: String)

// A Users table with columns: id, name
class Users(tag: Tag) extends Table[User](tag, "USERS") {

  // This is the primary key column:
  def id: Rep[Int] = column[Int]("ID", O.PrimaryKey)
  def name: Rep[String] = column[String]("NAME")

  // Every table needs a * projection with the same type as the table's type
parameter
  // the * projection (e.g. select * ...) auto-transforms the tupled
  // column values to / from a User
  def * = (id, name) <> (User.tupled, User.unapply)

}
```

The case class User is the domain model used within the Scala application to represent Users. In the relational database there is a table called USERs that contains an ID (which is the primary key of type Integer) and a column NAME which is a varchar.

The class Users maps the relational table to the case class User.

The Users class extends the Table class specifying the type to use with the Table (in this case User).

All columns are defined through the column method. Each column has a Scala type and a column name for the database (usually in upper case). The following primitive types are supported out of the box for JDBC-based databases in JdbcProfile (with certain limitations imposed by the individual database profiles):

- Numeric types: Byte, Short, Int, Long, BigDecimal, Float, Double
- LOB types: java.sql.Blob, java.sql.Clob, Array[Byte]
- Date types: java.sql.Date, java.sql.Time, java.sql.Timestamp
- Boolean
- String
- Unit
- java.util.UUID

Nullable columns are represented by Option[T] where T is one of the supported primitive types.

After the column name, you can add optional column options to a column definition. The applicable options are available through the table's O object. The following ones are defined for JdbcProfile:

- PrimaryKey: Mark the column as a (non-compound) primary key when creating the DDL statements.
- Default[T](defaultValue: T): Specify a default value for inserting data into the table without this column. This information is only used for creating DDL statements so that the database can fill in the missing information.
- SqlType(typeName: String): Use a non-standard database-specific type for the DDL statements (e.g. SqlType(`VARCHAR(20)') for a String column).
- AutoInc: Mark the column as an auto-incrementing key when creating the DDL statements. Unlike the other column options, this one also has a meaning outside of DDL creation: Many databases do not allow non-AutoInc columns to be returned when inserting data (often silently ignoring other columns), so Slick will check if the return column is properly marked as AutoInc where needed.

If you look back at the definition of the Users class you may note that there is an odd looking method defined at the end of the class:

```
def * = (id, name) <> (User.tupled, User.unapply)
```

Every table class must supply a '*' method containing a default projection. That is, this method defines how to convert a row returned from the database into the domain instances used within the application. Or to put it another way this method converts rows into Users in this case.

The above example uses a custom type for its * projection by adding a bidirectional mapping with the <> operator. This approach is optimised for case classes (with a simple apply method and an unapply method that wraps its result in an Option) but it can also be used with arbitrary mapping functions.

Interestingly, it is also possible to get Slick to auto-generate classes to handle the mapping from an existing database structure to a set of Scala classes. This is done using the click.codegen library (see online documentation for more information on how to do this https://github.com/slick/slick-codegen-example).

35.5 Table Query Interface

As well as the table to Scala type mapping class, it is also necessary to create a TableQuery object that will represent the actual database table.

This can be done very easily using the TableQuery[T] type. The simple TableQuery[T] is actually a macro which expands to a proper TableQuery instance that calls the table class's constructor (in our case that would be newTableQUery (new Users()).

This is can be used as follows:

```
// The query interface for the Users table
val users: TableQuery[Users] = TableQuery[Users]
```

It is also possible to create an object to allow additional functionality to be associated with a table:

```
object users extends TableQuery(new Users(_)) {
  val findByName = this.findBy(_.name)
}
```

35.6 Creating Tables

We can get Slick to create tables within our database (by generating the underlying Data Definition Language (DDL) statements). This can be done using the TableQuery's schema method. The object returned by this method supports actions such as create, drop and truncate.

For example, to create the users table we can use

```
// Create the USERS table
users.schema.create
```

For debugging purposes, you can get hold of the statements generated by these operations using:

```
schema.create.statements.foreach(println)
```

35.7 Inserting Data

Rows can be inserted into the database using the TableQuery object and the mappings supported by the Table type. In this case it means that we can add instances of the User case class to the TableQUery object users. The act of adding these instances to the TableQuery type results in a row being added to the table, for example

```
users += User(101, "John"),
```

This will add the user "John" with the Id 101 to the tables users, where users was defined as:

```
val users: TableQuery[Users] = TableQuery[Users]
```

By default, += gives you a count of the number of affected rows (which will usually be 1). When using the ++= batch insert operation, Slick makes use of the JDBC batch API. The underlying JDBC driver will decide how to transmit the batch (via SQL) to the database server.

35.8 Composing Database I/O Actions

All actions on a database in Slick are instances of a DBIOAction, parameterised by the result type it will produce when you execute it.

DBIOActions can be combined together using several different combinators accessible from the DBIO utility. The simplest of these is the DBIO.seq combinatory which takes a varargs list of actions to run in sequence, discarding their return value.

As an example, if we want to combine together a sequence of actions that will create a table and populate it with some initial data we can use the DBIO.seq combinatory as follows:

```
val setup = DBIO.seq(

 // Create the USERS table
 users.schema.create,

 // Insert some users
 users += User(101, "John"),
 users += User( 49, "Denise"),
 users += User(150, "Phoebe")

)
```

It is possible to pick up a return value as that via the andThen operator. This can be used to combine together a series of actions and keep the result of that last one, for example

```
val ins1: DBIO[Int] = users += User(101, `John')
val ins2: DBIO[Int] = users += User(49, `Denise'),

val a1: DBIO[Int] = ins1 andThen ins2
```

35.9 Example Setting Up a Database

The following listing brings all of the aspects presented earlier in this chapter together. It connects up to a company database and creates a new table USERS. It then adds several users to that table and then waits for the action to complete before terminating.

```
package com.jjh.scala.db

import slick.jdbc.MySQLProfile.api._

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration.Duration

import scala.concurrent.{Await, Future}

// The main application
object SetupDBApp extends App {

  val db = Database.forURL(
    "jdbc:mysql://localhost:3306/company" +
        "?user=user&password=user123",
    driver="com.mysql.cj.jdbc.Driver")

  try {
    println("starting")

    // The query interface for the Users table
    val users: TableQuery[Users] = TableQuery[Users]

    val setup = DBIO.seq(

      // Create the USERS table
      users.schema.create,

      // Insert some users
      users += User(101, "John"),
      users += User( 49, "Denise"),
      users += User(150, "Phoebe")

    )

    println("Run db setup")
    val f = db.run(setup)

    f.onComplete{ case s => println(s"Result: $s") }
    Await.result(f, Duration.Inf)

    println("Database setup complete")

  } catch {
    case e: Throwable => e.printStackTrace()
  } finally {
    db.close
  }
}
```

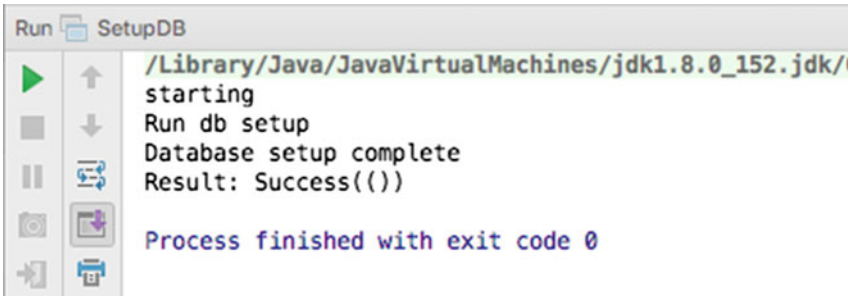
```
    println("Run db setup")
    val f = db.run(setup)

    f.onComplete{ case s => println(s"Result: $s") }
    Await.result(f, Duration.Inf)

    println("Database setup complete")

  } catch {
    case e: Throwable => e.printStackTrace()
  } finally {
    db.close
  }
}
```

The output generated when this application is run is illustrated below:



35.10 Querying for Data

This is where the power (and simplicity) of Slick really comes to the fore. Using Slick querying a database for data looks very much as it would like if a simple collection was being used as the source of the data.

Of course the source of the data is now a database and the query that underpins the retrieval of that data is still an SQL statement, but the programmers API hides all of this. The one aspect that is different is that the execution of the query is handled asynchronously and thus a Future is returned from the Database run method. This future will return a result at some point in the future (hence the name).

For example

```
Val q1 = for (u <- users) yield u
val a1 = q1.result
val f1: Future[Seq[User]] = db.run(a1)
f1.onComplete{ case s => println(s"F1 Result: $s") }
Await.result(f1, Duration.Inf)
```

The above code appears to use a for comprehension on the users object (of course this actually runs a SQL query and then applies the for comprehension to the

results. Strictly speaking (and working backwards) the db.run method takes a DBIOAction and returns a Future that will allow the results of the query to be accessed. Execution of the action starts in the background when the run method is called. Thus the call method is not blocked. This example therefore uses the Await. result method to wait until the result is returned from the Future. When the Future completes the onComplete method will be invoked which in this cases will print the result returned. The result is a collection of User objects.

Another approach is to stream results. In this case the actual collection type is ignored and elements are streamed directly from the result set through a Reactive Stream Publisher, which can be processed and consumed by an Akka Stream.

Execution of the DBIOAction does not start until a Subscriber is attached to the stream. If multiple Subscribers subscribe to the same Publisher, each one triggers an independent execution of the actions.

Stream elements are signalled as soon as they become available in the streaming part of the DBIOAction. The end of the stream is signalled only after the entire action has completed. For example, when streaming inside a transaction and all elements have been delivered successfully, the stream can still fail afterwards if the transaction cannot be committed. The Subscriber is notified of this failure.

The previous query can be rewritten to use streams as follows:

```
val q = for (u <- users) yield u
  val a = q.result
  val p: DatabasePublisher[User] = db.stream(a)
  p.foreach { s => println(s"Element: $s") }
```

It is also possible to sort and filter results. The Database.sortBy method can be used to sort the results of a query, for example

```
val q3 = coffees.sortBy(_.name.desc.nullsFirst)
```

In turn the database filter method can be used to filter a query, for example

```
val q2 = users.filter(_.id > 100)
val a2 = q2.result
val f2: Future[Seq[User]] = db.run(a2)
f2.onComplete{ case s => println(s"F2 Result: $s") }
Await.result(f2, Duration.Inf)
```

In this case the result is a collection of User instances where the id of the user is above 100.

Of course it is not necessary to always generate a collection of User objects, and if all we are interested in are the names of the users we could select those values from the query:

```
val q3 = users.filter(_.id > 100).map(_.name)
val a3 = q3.result
val f3: Future[Seq[String]] = db.run(a3)
f3.onComplete{ case s => println(s"F3 Result: $s") }
Await.result(f3, Duration.Inf)
```

Note how this is done using the map function chained to the result of the filter function. Of course in reality the SQL query generated from this combines both the filter and the map so that only the names are returned from the database.

It is also possible to use the filter function to select just a single row from the database:

```
val q4 = users.filter(_.id === 101)
  val a4 = q4.result
  val f4 = db.run(a4)
  f4.onComplete{ case s => println(s"F4 Result: $s") }
  Await.result(f4, Duration.Inf)
```

Note the use of the '===' comparison operator. In fact you also have to use =!= instead of != for inequality. This is necessary because these operators are already defined (with unsuitable types and semantics) on the base type Any, so they cannot be replaced by extension methods.

```
package com.jjh.scala.db

import scala.concurrent.{Await, Future}
import slick.jdbc.MySQLProfile.api._
import scala.concurrent.duration.Duration
import scala.concurrent.ExecutionContext.Implicits.global
import slick.basic.DatabasePublisher

// Demonstrates various ways of reading data
object QueryDbApp extends App {

  val db = Database.forURL(
  "jdbc:mysql://localhost:3306/company?" +
    "user=user&password=user123",
  driver="com.mysql.cj.jdbc.Driver")

  try {

  println("starting")
  // The query interface for the Users table
  val users = TableQuery[Users]

  val q1 = for (u <- users) yield u
  val a1 = q1.result
  val f1: Future[Seq[User]] = db.run(a1)
  f1.onComplete{ case s => println(s"F1 Result: $s") }
  Await.result(f1, Duration.Inf)

  println("-" * 25)

  val q2 = users.filter(_.id > 100)
  val a2 = q2.result
  val f2: Future[Seq[User]] = db.run(a2)
  f2.onComplete{ case s => println(s"F2 Result: $s") }
  Await.result(f2, Duration.Inf)
```

```
    println("-" * 25)

    val q3 = users.filter(_.id > 100).map(_.name)
    val a3 = q3.result
    val f3: Future[Seq[String]] = db.run(a3)
    f3.onComplete{ case s => println(s"F3 Result: $s") }
    Await.result(f3, Duration.Inf)

    println("-" * 25)

    val q4 = users.filter(_.id === 101)
    val a4 = q4.result
    val f4 = db.run(a4)
    f4.onComplete{ case s => println(s"F4 Result: $s") }
    Await.result(f4, Duration.Inf)

    println("-" * 25)

    val q = for (u <- users) yield u
    val a = q.result
    val p: DatabasePublisher[User] = db.stream(a)
    p.foreach { s => println(s"Element: $s") }

    Thread.sleep(1000)
    println("Done")

  } catch {
    case e: Throwable => e.printStackTrace()
  } finally {
    db.close
  }
}
```

35.11 Updating, Upserting and Deleting Data

Updates are performed by writing a query that selects the data to update and then replacing it with new data. The query must only return raw columns (no computed values) selected from a single table.

```
val q = for {c <- coffees if c.name === `Espresso'} yield c.price
val updateAction = q.update(10.49)
```

Deleting works very similarly to updating. You write a query which selects the rows to delete and then get an Action by calling the delete method on it:

```
val q = users.filter(_.Id === 15)
val action = q.delete
```

It is also possible to upsert—that is either insert or update (depending on whether the row already exists in the database or not):

```
val updated = users.insertOrUpdate(User(1, `Bob))
// returns: number of rows updated
```