Core Slick

Working with Relational Databases in Scala with Slick 2

Table of Contents

Chapter 1. Preface

Slick is a Scala library for working with databases: querying, inserting data, updating data, and representing a schema. Queries are written in Scala and type checked by the compiler. Slick aims to make working with a database similar to working with regular Scala collections.

This material is aimed at a Scala developer who has:

- taken the *Core Scala* course or equivalent;
- a good understanding of relation databases (rows, columns, joins, indexes) and SQL;
- access to a relational database (PostgreSQL is the example we use); and
- JDK 7 installed, along with an editor or IDE (Scala IDE for Eclipse, or IntelliJ).

It is useful to have experience using the SBT build tool.

Chapter 2. Orientation

(big picture example stuff)

(don’t worry about details like projects and tags, we’ll get to them later)

(followed by hands on getting there)

Chapter 3. Starting out with Slick

This section gets us working with Scala and Slick, creating a tables in a database, inserting rows, running simple queries.

3.1. Database Configuration

For this example we will use PostgreSQL 9, and a database called "core-slick":

Create a Database and Login from `psql`

```
CREATE DATABASE "core-slick" WITH ENCODING 'UTF8';
CREATE USER "core" WITH PASSWORD 'trustno1';
GRANT ALL ON DATABASE "core-slick" TO core;
```

Check you can login to this database:

```
$ psql -d core-slick core
```

Supported Databases

Slick supports PostgreSQL, MySQL, Derby, H2, SQLite, and Microsoft Access.

To work with DB2, SQL Server or Oracle you need a commercial license. These are the closed source *Slick Drivers* known as the *Slick Extensions*.

3.2. An SBT Project

To use Slick we create a regular Scala project and reference the Slick dependencies:

build.sbt

```
name := "core-slick-example"

version := "1.0"

scalaVersion := "2.10.3"

libraryDependencies += "com.typesafe.slick" %% "slick" % "2.0.1"

libraryDependencies += "ch.qos.logback" % "logback-classic" % "1.1.2"
libraryDependencies += "org.postgresql" % "postgresql" % "9.3-1101-jdbc41"
```

(To do: explain the dependencies)

When you’ve created *build.sbt* (or downloaded the example project), run SBT and the dependencies will be fetched.

IDEs

If you’re working with IntelliJ IDEA or the Eclipse Scala IDE, our *core-slick-example* project includes the plugins to generate the IDE project files:

```
sbt> eclipse
```

or

```
sbt> gen-idea
```

…and then open the project directory in your IDE. For Eclipse, this is *File → Import → Existing Project* menu.

3.3. Our First Table

schema1.scala

```
package underscoreio.schema

import scala.slick.driver.PostgresDriver.simple._

object Example1 extends App {

  class Planet(tag: Tag) extends Table[(Int,String,Double)](tag, "planet")
 {
    def id = column[Int]("id", O.PrimaryKey, O.AutoInc)
    def name = column[String]("name")
    def distance = column[Double]("distance_au")
    def * = (id, name, distance)
  }

  lazy val planets = TableQuery[Planet]
  Database.forURL("jdbc:postgresql:core-slick", user="core",
 password="trustno1", driver = "org.postgresql.Driver") withSession {
    implicit session =>
      planets.ddl.create
  }

}
```

Running this application will create the schema. You can run it from your IDE, or with `sbt "run-main underscoreio.schema.Example1"`.

If you example the schema, there should be no surprises:

```
core-slick=# \d
             List of relations
 Schema |     Name      |   Type   | Owner
--------+---------------+----------+-------
 public | planet        | table    | core
 public | planet_id_seq | sequence | core
(2 rows)

core-slick=# \d planet
                                   Table "public.planet"
   Column    |          Type          |                      Modifiers
-------------+------------------------
+------------------------------------------------------
 id          | integer                | not null default
 nextval('planet_id_seq'::regclass)
 name        | character varying(254) | not null
 distance_au | double precision       | not null
Indexes:
    "planet_pkey" PRIMARY KEY, btree (id)
```

(lots to discuss about the code)

- What is a `Tag`? "The Tag carries the information about the identity of the Table instance and how to create a new one with a different identity. Its implementation is hidden away in TableQuery.apply to prevent instantiation of Table objects outside of a TableQuery" and "The tag is a table alias. You can use the same table in a query twice by tagging it two different ways. I believe Slick assigns the tags for you."
- How does `Table[(Int,String)]` match up to `id` and `name` fields? - that's how Slick is going to represent rows. We can customize that to be something other than a tuple, a case class in particular.

- What is a projection (`*`) and why do I need to define it? It’s the default for queries and inserts. We will see how to convert this into more useful representation.
- What is a `TableQuery` ?
- What is a session?

Note that driver is specified. You might want to mix in something else (e.g., H2 for testing). See later.

Note we can talk about having longer column values later.

The `O` for PK or Auto means "Options".

3.3.1. Schema Creation

Our table, `planet` , was created with `table.dd.create` . That’s convenient for us, but Slick’s schema management is very simple. For example, if you run `create` twice, you’ll see:

```
org.postgresql.util.PSQLException: ERROR: relation "planet" already exists
```

That’s because `create` blindly issues SQL commands:

```
println(planets.ddl.createStatements.mkString)
```

…will output:

```
create table "planet" ("id"
 SERIAL NOT NULL PRIMARY KEY,"name" VARCHAR(254) NOT NULL)
```

(There’s a corresponding `dropStatements` that does the reverse).

To make our example easier to work with, we could query the database meta data and find out if our table already exists before we create it:

```
if (MTable.getTables(planets.baseTableRow.tableName).firstOption.isEmpty)
  planets.ddl.create
```

However, for our simple example we’ll end up dropping and creating the schema each time:

```
MTable.getTables(planets.baseTableRow.tableName).firstOption match {
   case None =>
     planets.ddl.create
   case Some(t) =>
     planets.ddl.drop
     planets.ddl.create
  }
```

We’ll look at other tools for managing schema migrations later.

3.4. Inserting Data

```
// Populate with some data:

planets += (100, "Earth",  1.0)

planets ++= Seq(
   (200, "Mercury",  0.4),
   (300, "Venus",    0.7),
   (400, "Mars" ,    1.5),
   (500, "Jupiter",  5.2),
   (600, "Saturn",   9.5),
   (700, "Uranus",  19.0),
   (800, "Neptune", 30.0)
)
```

Each `+=` or `++=` executes in its own transaction.

NB: result is a row count `Int` for a single insert, or `Option[Int]` for a batch insert. It’s optional because not all databases support returning a count for batches.

We’ve had to specify the id, name and distance, but this may be surprising because the ID is an auto incrementing field. What Slick does, when inserting this data, is ignore the ID:

```
core-slick=# select * from planet;
 id |   name    | distance_au
----+----------+-------------
  1 | Earth    |           1
  2 | Mercury |         0.4
  3 | Venus    |         0.7
  4 | Mars     |         1.5
  5 | Jupiter |         5.2
  6 | Saturn  |          9.5
  7 | Uranus  |           19
  8 | Neptune |           30
(8 rows)
```

This is, generally, what you want to happen, and applies only to auto incrementing fields. If the ID was not auto incrementing, the ID values we supplied (100,200 and so on) would have been used.

If you really want to include the ID column in the insert, use the `forceInsert` method.

3.5. A Simple Query

Let's fetch all the planets in the inner solar system:

```
val query = for {
  planet <- planets
  if planet.distance < 5.0
} yield planet.name

println("Inner planets: " + query.run)
```

This produces:

```
Inner planets: Vector(Earth, Mercury, Venus, Mars)
```

What did Slick do to produce those results? It ran this:

```
select s9."name" from "planet" s9 where s9."distance_au" < 5.0
```

Note that it did not fetch all the planets and filter them. There's something more interesting going on that that.

Logging What Slick is Doing

Slick uses a logging framework called SLFJ. You can configure this to capture information about the queries being run, and the log to different back ends. The "core-slick-example" project uses a logging back-end called *Logback*, which is configured in the file *src/main/resources/logback.xml*. In that file we enable statement logging by turning up the logging to debug level:

```
<logger name="scala.slick.jdbc.JdbcBackend.statement" level="DEBUG"/
>
```

When we next run a query, each statement will be recorded on standard output:

```
18:49:43.557 DEBUG s.slick.jdbc.JdbcBackend.statement -
 Preparing statement: drop table "planet"
18:49:43.564 DEBUG s.slick.jdbc.JdbcBackend.statement
 - Preparing statement: create table "planet" ("id"
 SERIAL NOT NULL PRIMARY KEY,"name" VARCHAR(254) NOT
 NULL,"distance_au" DOUBLE PRECISION NOT NULL)
```

You can enable a variety of events to be logged:

- `scala.slick.jdbc.JdbcBackend.statement` - which is for statement logging, as you've seen.
- `scala.slick.session` - for session information, such as connections being opened.
- `scala.slick` - for everything! This is usually too much.

3.6. Running Queries in the REPL

For experimenting with queries it's convenient to use the Scala REPL and create an implicit session to work with. In the "core-slick-example" SBT project, run the `console` command to enter the Scala REPL with the Slick dependencies loaded and ready to use:

```
> console
[info] Starting scala interpreter...
[info]

Session created, but you may want to also import a schema. For example:

    import underscoreio.schema.Example1._
 or import underscoreio.schema.Example5.Tables._

import scala.slick.driver.PostgresDriver.simple._
db: slick.driver.PostgresDriver.backend.DatabaseDef =
 scala.slick.jdbc.JdbcBackend$DatabaseFactoryDef$$anon$5@6dbc2f23
```

```
session: slick.driver.PostgresDriver.backend.Session =
 scala.slick.jdbc.JdbcBackend$BaseSession@5dbadb1d
Welcome to Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java
 1.7.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import underscoreio.schema.Example2._
import underscoreio.schema.Example2._

scala> planets.run
08:34:36.053 DEBUG s.slick.jdbc.JdbcBackend.statement - Preparing
 statement: select x2."id", x2."name", x2."distance_au" from "planet" x2
res1: Seq[(Int, String, Double)] = Vector((1,Earth,1.0), (2,Mercury,0.4),
 (3,Venus,0.7), (4,Mars,1.5), (5,Jupiter,5.2), (6,Saturn,9.5),
 (7,Uranus,19.0), (8,Neptune,30.0), (9,Earth,1.0))

scala> planets.firstOption
08:34:42.320 DEBUG s.slick.jdbc.JdbcBackend.statement - Preparing
 statement: select x2."id", x2."name", x2."distance_au" from "planet" x2
res2: Option[(Int, String, Double)] = Some((1,Earth,1.0))

scala>
```

3.7. Exercises

- What happens if you used 5 rather than 5.0 in the query?
- 1AU is roughly 150 million kilometers. Can you run query to return the distances in kilometers? Where is the conversion to kilometers performed? Is it in Scala or in the database?
- How would you count the number of planets? Hint: in the Scala collections the method `length` gives you the size of the collection.
- Select the planet with the name "Earth". You'll need to know that equals in Slick is represented by `===` (three equals signs). It's also useful to know that `=!=` is not equals.
- Using a for comprehension, select the planet with the id of 1. What happens if you try to find a planet with an id of 999?
- You know that for comprehensions are sugar for `map`, `flatMap`, and `filter`. Use `filter` to find the planet with an id of 1, and then the planet with an id of 999. Hint: `first` and `firstOption` are useful alternatives to `run`.

- The method `startsWith` tests to see if a string starts with a particular sequence of characters. For example `"Earth".startsWith("Ea")` is `true`. Find all the planets with a name that starts with "E". What query does the database run?
- Slick implements the method `like`. Find all the planets with an "a" in their name.
- Find all the planets with an "a" in their name that are more than 5 AU from the Sun.

3.8. Sorting

As you've seen, Slick can produce sensible queries from for comprehensions:

```
(for {
  p <- planets
  if p.name like "%a%"
  if p.distance > 5.0
 } yield p ).run
```

This equates to the query:

```
select
  s17."id", s17."name", s17."distance_au"
from
 "planet" s17
where
 (s17."name" like '%a%') and (s17."distance_au" > 5.0)
```

We can take a query and add a sort order to it:

```
val query = for { p <- planets if p.distance > 5.0} yield p
query.sortBy(row => row.distance.asc).run
```

(Or `desc` to go the other way).

This will run as:

```
select
  s22."id", s22."name", s22."distance_au"
from
  "planet" s22
where
  s22."distance_au" > 5.0
```

```
order by
  s22."distance_au"
```

…to produce:

```
Vector((5,Jupiter,5.2), (6,Saturn,9.5), (7,Uranus,19.0), (8,Neptune,30.0))
```

What's important here is that we are taking a query, using `sortBy` to create another query, before running it. Query composition is a topic we will return to later.

3.9. The Types Involved in a Query

3.10. Update & Delete

Queries are used for update and delete operations, replacing `run` with `update` or `delete`.

For example, we don't quite have the distance between the Sun and Uranus right:

```
val udist = planets.filter(_.name === "Uranus").map(_.distance)
udist.update(19.2)
```

WHen `update` is called, the database will receive:

```
update "planet" set "distance_au" = ? where "planet"."name" = 'Uranus'
```

The arguments to `update` must match the result of the query. In this example, we are just returning the distance, so we just modify the distance.

3.10.1. Exercises

- Modify both the distance and name of a planet. Hint: you can do this with one call to `update`.
- Delete Earth.
- Delete all the planets with a distance less than 5.0.
- Double the distance of all the planets. (You need to do this client-side, not in the database)

Chapter 4. Structuring the Schema

```
object Tables extends {
  val profile = scala.slick.driver.PostgresDriver
} with Tables

trait Tables {

  val profile: scala.slick.driver.JdbcProfile
  import profile.simple._

  class Planet(tag: Tag) extends Table[(Int,String,Double)](tag, "planet")
 {
    def id = column[Int]("id", O.PrimaryKey, O.AutoInc)
    def name = column[String]("name")
    def distance = column[Double]("distance_au")
    def * = (id, name, distance)
  }

  lazy val planets = TableQuery[Planet]
}

// Our application:

import Tables._

// session, queries, go here...
```

Chapter 5. Connecting, Transactions, Sessions

While we're restructuring, we'll move the `Database.forURL` code into a method:

```
object Tables extends {
    val profile = scala.slick.driver.PostgresDriver
} with Tables {
  val db = Database.forURL("jdbc:postgresql:core-slick",
                 user="core", password="trustno1",
                 driver = "org.postgresql.Driver")
}
```

This will allow us to run `db.withSession` anywhere we want to interact with the database.

You can think of a session as the connection to the database. You need it anytime you want to run a query, or lookup database metadata.

The session comes from a `Database` which you can create in a number of ways:

- `forDataSource` - when working with a `javax.sql.DataSource`
- `forName` - if you are using JNDI.
- `forURL` - which is what we've been using.

The `withSession` method ensures that the session is closed once the method returns. This means you don't have to worry about closing connections. It also means you must not "leak sessions" out of the method, for example by returning the session object (even inside a `Future`).

Inside a session, each interaction with the database happens in "auto commit" mode. If you want to manage transactions yourself, use the session object to create a transaction:

```
session.withTransaction {
  // Queries here as usual
}
```

The transaction will commit at the end of the block unless an exception is thrown, or you call `session.rollback` at any point.

5.1. Exercises

- Create a transaction to delete Earth, but rollback inside the transaction. Check the database still contains Earth.
- In the following code, will you see "Almost done…" printed?

```
session.withTransaction {
    planets.delete
    session.rollback()
    println("Almost done...")
}
```

Chapter 6. Using Case Classes

schema4.scala

```
object Tables extends {
  val profile = scala.slick.driver.PostgresDriver
} with Tables

trait Tables {

  val profile: scala.slick.driver.JdbcProfile
  import profile.simple._

  case class Planet(name: String, distance: Double, id: Long=0L)

  class PlanetTable(tag: Tag) extends Table[Planet](tag, "planet") {
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def name = column[String]("name")
    def distance = column[Double]("distance_au")
    def * = (name, distance, id) <> (Planet.tupled, Planet.unapply)
  }

  lazy val planets = TableQuery[PlanetTable]
}

// Our application:

import Tables._

// session, queries, go here...
```

Initialisation pattern.

naming: PlanetRow, Planets v. Planet, PlanetTable

6.1. Inserting data

```
planets += Planet("Earth", 1.0)

planets ++= Seq(
  Planet("Mercury",  0.4),
  Planet("Venus",    0.7),
  Planet("Mars" ,    1.5),
  Planet("Jupiter",  5.2),
  Planet("Saturn",    9.5),
  Planet("Uranus",   19.0),
  Planet("Neptune", 30.0)
)
```

6.2. Queries

Chapter 7. Table, Rows and Column Customisation

- NULL columms
- PK

7.1. Exercixces

- HList

Chapter 8. Joins

insert diagram here

schema5.scala

```
trait Tables {
  val profile: scala.slick.driver.JdbcProfile
  import profile.simple._

  case class Planet(name: String, distance: Double, id: Long=0L)

  class PlanetTable(tag: Tag) extends Table[Planet](tag, "planet") {
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def name = column[String]("name")
    def distance = column[Double]("distance_au")
    def * = (name, distance, id) <> (Planet.tupled, Planet.unapply)
  }

  lazy val planets = TableQuery[PlanetTable]

  case class Moon(name: String, planetId: Long, id: Long=0L)

  class MoonTable(tag: Tag) extends Table[Moon](tag, "moon") {
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def name = column[String]("name")
    def planetId = column[Long]("planet_id")

    def * = (name, planetId, id) <> (Moon.tupled, Moon.unapply)

    def planet = foreignKey("planet_fk", planetId, planets)(_.id)

  }

  lazy val moons = TableQuery[MoonTable]
}
```

Now that we have more tables, our automatic schema creation code becomes a little more complicated:

```
def exists[T <: Table[_]](table: TableQuery[T])(implicit session:
 Session) : Boolean =
  MTable.getTables(table.baseTableRow.tableName).firstOption.isDefined
```

```
def dropAndCreate(implicit session: Session) : Unit = {
  if (exists(moons)) moons.ddl.drop
  if (exists(planets)) planets.ddl.drop
  (planets.ddl ++ moons.ddl).create
}
```

Although `(planets.ddl ++ moons.ddl).drop` is smart enough to drop tables and constraints in the correct order, it will also try to drop tables that do not exists. This leads to a runtime error. To avoid that, we test and drop.

The resulting table creation SQL will be:

```
create table "planet" (
  "name" VARCHAR(254) NOT NULL,
  "distance_au" DOUBLE PRECISION NOT NULL,
  "id" SERIAL NOT NULL PRIMARY KEY
)

create table "moon" (
  "name" VARCHAR(254) NOT NULL,
  "planet_id" BIGINT NOT NULL,
  "id" SERIAL NOT NULL PRIMARY KEY
)

alter table "moon" add constraint "planet_fk" foreign key("planet_id")
```

Inserting data does not change, except that we need to query for the ID of the planets for the moon-to-planet relationship:

```
db.withSession {
  implicit session =>

    // Create the database table:
    dropAndCreate

    // Populate Planets:

    planets ++= Seq(
      Planet("Earth", 1.0)
      Planet("Mercury",  0.4),
      Planet("Venus",    0.7),
      Planet("Mars" ,    1.5),
      Planet("Jupiter",  5.2),
      Planet("Saturn",    9.5),
      Planet("Uranus",   19.0),
      Planet("Neptune", 30.0)
    )

    // We want to look up a planet by name to create the association
    def idOf(planetName: String) : Long =
      planets.filter(_.name === planetName).map(_.id).first

    val earthId = idOf("Earth")
    val marsId = idOf("Mars")

    moons ++= Seq(
      Moon("The Moon", earthId),
      Moon("Phobos", marsId),
      Moon("Deimos",  marsId)
    )
}
```

For the moons we execute two queries for the planet IDs, then three inserts. The resulting database is:

```
core-slick=# select * from moon;
   name    | planet_id | id
----------+-----------+----
 The Moon |         1 |  1
 Phobos   |         4 |  2
 Deimos   |         4 |  3
(3 rows)
```

8.1. Explicit Joins

```
val query = for {
  (planet, moon) <- moons innerJoin planets on (_.planetId === _.id)
} yield (planet.name, moon.name)
```

```
select x2."name", x3."name" from (select x4."name" as "name",
 x4."planet_id" as "planet_id", x4."id" as "id" from "moon" x4)
 x2 inner join (select x5."name" as "name", x5."distance_au" as
 "distance_au", x5."id" as "id" from "planet" x5) x3 on x2."planet_id" =
 x3."id"
```

8.2. Implicit Joins

```
val query = for {
  p <- planets
  m <- moons
  if m.planetId === p.id
} yield (p.name, m.name)
```

```
select x2."name", x3."name" from "planet" x2, "moon" x3 where
 x3."planet_id" = x2."id"
```

t1.join(t2).on(condition)

Chapter 9. Queries Compose

Reuse. Only runs when you say. Keep to a `Query` for as long as possible.

Chapter 10. Drop and Take

planets.drop(2).take(3)

Chapter 11. Unions

(q1 union q2).run without dups, or ++ for union all

Chapter 12. Calling Functions

```
   val dayOfWeek = SimpleFunction[Int]("day_of_week")

     val q1 = for {
       (dow, q) <- salesPerDay.map(s => (dayOfWeek2(s.day),
  s.count)).groupBy(_._1)
     } yield (dow, q.map(_._2).sum)
```

Chapter 13. Query Extensions

E.g., pagination or byName("Mars")

Chapter 14. Dynamic Queries

need to upper case everything??

```
implict.... dynamicSort(keys: String*) : Query[T,E] = {
 keys match {
   case nil = query
   case h :: t =>
    dynamicSortImpl(t).sortBy( table => )
    // split h on . to get asc desc
   h match {
    case name :: Nil =>  table.column[String](name).asc
    case _ => ???

 }
}
}
```

danger… access to user supplied input!!

```
dynamicSort("street.desc", "city.desc")
```

Chapter 15. Aggregations

counts, grouping and all that.

max, min, sum, avg

broupBy

Chapter 16. Virtual Columns and Server Side Casts

def x = whatever

`asColumnOf[Double]`

Chapter 17. More on Joins

17.1. Outer Joins

`leftJoin` - dealing with NULL values

map all columns to option types via `.?` (nullable column)

slick will do this for you one day.

17.2. Auto Join

https://skillsmatter.com/skillscasts/4577-patterns-for-slick-database-applications

15:23 in

table1.joinOn(table2) : Query[(T1,T2),(Ta,Tb)]

via implicit joinCondition for T1,T2

Chapter 18. Compiled Queries

Chapter 19. Custom Types

```
class SupplierId(val value: Int) extends AnyVal

case class Supplier(id: SupplierId, name: String,
 city: String)

implicit val supplierIdType = MappedColumnType.base
 [SupplierId, Int](_.value, new SupplierId(_))

class Suppliers(tag: Tag) extends
 Table[Supplier](tag, "SUPPLIERS") {
 def id = column[SupplierId]("SUP_ID", ...)
 ...
}
```

```
class SupplierId(val value: Int) extends MappedTo[Int]

case class Supplier(id: SupplierId, name: String,
 city: String)

class Suppliers(tag: Tag) extends
 Table[Supplier](tag, "SUPPLIERS") {
 def id = column[SupplierId]("SUP_ID", ...)
 ...
}
```

Chapter 20. Dates and Time

Joda! See https://mackler.org/LearningSlick2/

Chapter 21. Testing

https://groups.google.com/forum/#!topic/scalaquery/gDblUdKKrSY

Chapter 22. Terminology

Lifted Embedding

Chapter 23. Plain SQL

Chapter 24. Code Generation for Existing Databases

- basic usage
- customizing (snake case v. camel case StringExtensions)
- SourceCodeGenerator(model).code hook for adding more stuff. e.g., super.code + MORE STUFF. Nice autojoin example in https://skillsmatter.com/skillscasts/4577-patterns-for-slick-database-applications 31 mins in.

Chapter 25. Connection Pools

Chapter 26. Schema Migration

Chapter 27. Working with Multiple Databases

Chapter 28. Writing your own Driver

Chapter 29. Answers to Exercises

Chapter 30. Useful Links

- link to that nice mac postgresql app
- The Slick Mailing List[1] (the group is called "scalaquery" as that was the original name for the technology that we now call Slick).
- PostgreSQL manual[2].

[1] http://groups.google.com/group/scalaquery
[2] http://www.postgresql.org/docs/9.3/static/index.html