Messing Around with Typesafe Slick

Brian Clapper

March 19, 2015

Why am I qualified to talk about Slick?

- I've been using it, more or less daily, for a couple years now.
- I will not claim to have vast knowledge of Slick's inner workings. But, as someone who uses it a lot, I know a fair amount about it. (I guess that makes my knowledge... half-vast.)

Talk Outline

- Overview of Slick (`http://slick.typesafe.com`)
- Demonstration of Sample Application
- Some live coding

What is Slick?

What is Slick?

Slick:

- is a modern, database query and access library for Scala
- provides a collections-like view of database access
- allows you to construct queries in a type-safe fashion
- supports multiple backend databases
- allows you to drop down to SQL, if you really have to

What is Slick?

Slick:

- is a modern, database query and access library for Scala
- provides a collections-like view of database access
- allows you to construct queries in a type-safe fashion
- supports multiple backend databases
- allows you to drop down to SQL, if you really have to

Slick is not:

- a traditional, Hibernate-style ORM (*whew!*)
- particularly usable from Java

Some Simple Examples

Let's start out with a couple simple examples:

```
// Using Slick's query syntax
def allEmployees(maxSalary: Int): Seq[String] = {
  ( for (e <- Employees if e.salary <= maxSalary ) yield e
}

// Using SQL string interpolation
def allEmployees2(maxSalary: Int): Seq[String] = {
  sql"SELECT name FROM Employees WHERE e.salary <= $maxSala
}
```

Tables

A table is just a class.

```
class EmployeesTable(tag: Tag)
  extends Table[(String, Int, Option[String])](tag, "peopl

  def name   = column[String]("name", O.PrimaryKey)
  def salary = column[Int]("salary")
  def spouse = column[Option[String]]("spouse") // nullabl

  def * = (name, salary)
}
```

The *base query* is defined on the table:

```
val Employees = TableQuery[EmployeesTable]
```

It's Just a Collection

The previous `for` loop is, of course, just `map` and `filter`:

```
Employees.filter { _.salary <= maxSalary }.map { _.name }
```

It's Just a Collection

The previous `for` loop is, of course, just `map` and `filter`:

```
Employees.filter { _.salary <= maxSalary }.map { _.name }
```

And, you get type safety:

```
Employees.filter { _.salary <= "10000" } // won't compile
```

Queries are Composable

This query hasn't executed yet:

```
val q1 = Employees.filter { _.salary <= maxSalary }.map { _
```

Queries are Composable

This query hasn't executed yet:

```
val q1 = Employees.filter { _.salary <= maxSalary }.map { _
```

. . . so we can augment it:

```
val q2 = limitOpt.map { limit => q1.take(limit) }.getOrElse

q2.list
```

Slick Supports Various RDBMS Backends

Open Source

- Derby/JavaDB
- H2
- HSQLDB/HyperSQL
- Microsoft Access (yuck)
- MySQL
- PostgreSQL
- SQLite

Slick Supports Various RDBMS Backends

Closed Source

Supported via a special *slick-extensions* package available from the Typesafe repo.

- DB2
- Microsoft SQL Server
- Oracle

Lifted Embedding

This is the main Slick API.

- Means you are not working with standard Scala types.
- Instead, you're using types that are *lifted* into a `Rep` type constructor.

Lifted Embedding

A comparison with a regular collections example clarifies.

```
case class Employee(name: String, salary: Int)
val employees: List[Employee] = List(...) // normal collec
val l = employees.filter(_.salary > 100000).map(_.name)
//                              ^        ^            ^
//                             Int      Int        String

class EmployeesTable(tag: Tag)
  extends Table[(String, Int, Option[String])](tag, "employ
   // Our previous definition
}
val Employees = TableQuery[EmployeesTable]
val q = Employees.filter(_.salary > 100000).map(_.name) //
//                              ^        ^            ^
//                          Rep[Int]  Rep[Int]  Rep[String]
```

Plain types (and values, like 10000) are lifted into Rep, to allow

Tuples . . .

You can define your table with tuples, like this:

```
class EmployeesTable(tag: Tag)
  extends Table[(String, Int, Option[String])](tag, "emplo

  def name   = column[String]("name", O.PrimaryKey)
  def salary = column[Int]("salary")
  def spouse = column[Option[String]]("spouse") // nullabl

  def * = (name, salary)
}
```

. . . or Case Classes

. . . or with a case class, like this:

```
case class Employee(name: String, salary: Int, spouse: Opti

class EmployeesTable(tag: Tag) extends Table[Employee])(tag
  def name   = column[String]("name", O.PrimaryKey)
  def salary = column[Int]("salary")
  def spouse = column[Option[String]]("spouse")

  // Tell Slick how to pack and unpack the case class
  def * = (name, salary, spouse) <> (Employee.tupled, Emplo
}
```

Only 22 columns?

Both of the previous examples use tuples, which means tables are limited to 22 columns.

Only 22 columns?

Both of the previous examples use tuples, which means tables are limited to 22 columns.

You need *more* than 22 columns? What's *wrong* with you?

Only 22 columns?

Both of the previous examples use tuples, which means tables are limited to 22 columns.

You need *more* than 22 columns? What's *wrong* with you?

It's possible to define tables with an arbitrary number of columns, using Slick `Shape` types. Doing so is more advanced and beyond the scope of this talk. However, more info is here:

`http://slick.typesafe.com/doc/2.1.0/userdefined.html#polymorphic-types-e-g-custom-tuple-types-or-hlists`

ID Columns

Columns defined as `Option[Type]` are nullable. Slick also supports case classes with optional types that map onto non-nullable columns. This capability is *really* useful for so-called synthetic keys:

```
case class Employee(id:     Option[Int], // None if not sa
                    name:   String,
                    ssn:    String,
                    salary: Int)
class EmployeesTable(tag: Tag) extends Table[Employee](tag,
  def id     = column[Int]("id", O.PrimaryKey, O.AutoInc)
  def name   = column[String]("name")
  def ssn    = column[String]("ssn")
  def salary = column[Int]

  def * = (id.?, name, ssn, salary) <> (Employee.tupled, Em
//             ^
//          Makes it all compile.
}
```

Constraints

You can define indexes and foreign keys

```
case class Employee(id: Option[Int], name: String, salary:
case class Phone(id: Option[Int], employeeID: Int, number:

class EmployeesTable(tag: Tag) extends Table[Employee](tag,
  def id     = column[Int]("id", O.PrimaryKey, O.AutoInc)
  def name   = column[String]("name")
  def salary = column[Int]
  def *      = (id.?, name, ssn, salary) <> (Employee.tuple
}
class PhonesTable(tag: Tag) extends Table[Phone](tag, "phon
  def id         = column[Int]("id", O.PrimaryKey, O.AutoIn
  def employeeID = column[Int]("employee_id")
  def number     = column[String]("number")
  def *          = (id.?, employeeID, number) <> (Phone.tup
  def employee   = foreignKey("pn_fk_01", employeeID, Emplo
      .id,
```

Generating DDL

You can have Slick generate your DDL for you. That may or may not be useful to you. (I don't usually do that.)

```
val db = // we haven't talked about how to do this yet

val ddl = Employees.ddl ++ Phones.ddl

db withDynSession {
  ddl.drop
  ddl.create
}
```

Accessing your Database

To access your (JDBC) database, you use a Slick `Database` object, which can be created in a number of ways:

```
// JDBC URL
val db = Database.forURL("jdbc:sqlite:my.db", driver="org.s
// A javax.sql.DataSource
val db = Database.forDataSource(dataSource)
// A JNDI name
val db = Database.forName(someNameString)
```

Each Driver is its Own Import

To use Slick, you have to import the API for the driver you're using:

```
import scala.slick.driver.SQLiteDriver
```

That's kind of annoying: Do you really want dependencies on that driver littered throughout your code?

Each Driver is its Own Import

To use Slick, you have to import the API for the driver you're using:

```
import scala.slick.driver.SQLiteDriver
```

That's kind of annoying: Do you really want dependencies on that driver littered throughout your code?

No. No, you don't.

Getting Around That Annoyance

It's not difficult get fix that problem. Here's an example:

```
import scala.slick.driver.{MySQLDriver,PostgresDriver,SQLit
import scala.slick.jdbc.JdbcBackend.Database

class DAL(val profile: JdbcProfile, db: Database)

object Startup {
  def init(configuration: SomeConfigurationThingie) {
    val driver = cfg.getOrElse("db.driver", "org.sqlite.JDI
    val url    = cfg.getOrElse("db.url", "jdbc:sqlite:my.dl
    val user   = cfg.getOrElse("db.user", "")
    val pw     = cfg.getOrElse("db.password, "")
    val db     = Database.forURL(url, driver=driver, user=u

    val dal = driver match {
      case "org.postgresql.Driver" => new DAL(PostgresDrive
      case "org.mysql.jdbc.Driver" => new DAL(MySQLDriver.
```

And now, we're cool

With that code in place, we can do something like this:

```
class EmployeesDAO(dal: DAL) {
  import dal.profile.simple._ // Shhh... It's magic.
  import dal.db
  import org.example.thingie.db.tables.Employees // the ba

  def getAll(): Seq[Employee] = {
    db withSession { implicit session =>
      (for (e <- Employees) yield e).list
    }
  }
}
```

Joins

Using our previous table definitions, what if we want to get a list of all the phone numbers for a particular employee, given the employee's name (i.e., a SQL JOIN)?

Joins

Using our previous table definitions, what if we want to get a list of all the phone numbers for a particular employee, given the employee's name (i.e., a SQL JOIN)?

```
val name = // this came from somewhere...

val q = for { e <- Employees if e.name === name
              n <- Phones if n.employeeID === e.id }
        yield n
```

Note the use of ===. That's required. == won't work.

Other Query Capabilities

```
Employees.sortBy(_.name.desc.nullsFirst) // ... ORDER BY n
Employees.drop(10).take(5) // SELECT * FROM EMPLOYEES LIMI
Employees.filter(_.salary < 10000) union Employees.filter(_
Employees.map(_.salary).min // SELECT MIN(e.salary) FROM e
Employees.map(_.salary).sum // SELECT SUM(e.salary) FROM e
Employees.length // SELECT COUNT(1) FROM employees
```

There are others. See the Slick docs for details.

Deleting

```
Employees.delete // Oh, no! We nuked all of them!

(for (e <- Employees where e.name === "Joe Smith")).delete
```

Inserting

```
// If you don't need the ID back:

Employees += Employee(None, "Joe Smith", 990000)
Employees ++= Seq( Employee(None"Maria Sanchez", 200000),
                   Employee(None, "Freddie Guy", 55000) )

// If you want the ID back, this is the idiom

val e = Employee("Maria Sanchez", 200000)
val id = (Employees returning Employees.map(_.id)) += e
```

Updates

Updates are easy enough, though there's a coupling issue I could live without.

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.

```
def updateEmployee(toSave: Employee) = {
  db withSession {
    val q = for (e <- Employees if e.id === toSave.id)
            yield ((e.name, e.salary))
    q.update((toSave.name, toSave.salary))
  }
}
```

Queries can be Compiled

For instance:

```
val compiledPhoneQuery = Compiled{ (empID: Column[Int]) =>
  val q = for { p <- PhoneNumbers if p.employeeID === empII
  q.sorted(_.name)
}

...

compiledPhoneQuery(someEmployee.id.get).run
compiledPhoneQuery(someOtherEmployee.id.get).run
```

Seeing your Statements

You can use logging to see the statements being issued, but you can also get them manually.

```
Employees.filter(_.salary > 100000).map(_.name).selectState
Employees.filter(_.id === employeeID).deleteStatement
```

Transactions

You can use the Session object's withTransaction method to create a transaction when you need one.

It takes a block that's executed in a single transaction. Any thrown exception causes an automatic rollback, but you can force a rollback, as well.

```
db withSession { implicit session =>
  session withTransaction {
    // your queries go here

    if (holyCrapThisIsHorrible) {
      session.rollback // signals Slick to rollback later
    }
  }
} // <- rollback happens here, if an exception was thrown
  //    or session.rollback was called
```

Let's try it

Let's build a Slick application. Use Typesafe Activator (available at `http://scala-lang.org/download/` to create a minimal Scala application):

```
$ activator new slickness
<bunch of messages>
Choose from these featured templates or enter a template name:
  1) minimal-akka-java-seed
  2) minimal-akka-scala-seed
  3) minimal-java
  4) minimal-scala
  5) play-java
  6) play-scala
(hit tab to see a list of all templates)
> 4
```

Add Slick

In the resulting `slickness/build.sbt` file, add a dependency on Slick and SQLite:

```
libraryDependencies ++= Seq("com.typesafe.slick" %% "slick'
                            "org.xerial"          % "sqlite
```

Stepping Outside the Presentation

Stage Direction: Presenter puts on coder hat and fires up IDE. . .

Future Slick

Slick 3.0 is just around the corner. Let's look over here, to see what it boasts:

`http://slick.typesafe.com/news/2015/02/20/slick-3.0.0-RC1-released.html`

Speaking of questions

Are there any?