Implementing a many-to-many Relationship with Slick

Author: Hermann Hueck

Source code and Slides available at:

https://github.com/hermannhueck/MusicService/tree/master/Services/MusicService-Play-Scala-Slick-NoAuth

Who am I?

Hermann Hueck

Software Developer – Scala, Java, Akka, Play

https://www.xing.com/profile/Hermann_Hueck

Presentation Overview

- Short Intro to Slick
- Many-to-many Relationship with Slick – Example: A Music Service
- Q & A

Part 1: Intro to Slick

- What is Slick?
- What is FRM (Functional Relational Mapping)?
- How to execute a database action?
- Why is it reactive?
- Short Reminder of Scala Futures
- Simple Slick Example (Activator Template: hello-slick-3.1)
- Table Definitions without and with Case Class Mapping

What is Slick?

The Slick manual says:

Slick (“Scala Language-Integrated Connection Kit”) is Typesafe‘s Functional Relational Mapping (FRM) library for Scala that makes it easy to work with relational databases. It allows you to work with stored data almost ***as if you were using Scala collections*** while at the same time giving you full control over when a database access happens and which data is transferred. You can also ***use SQL directly***. Execution of database actions is done ***asynchronous***ly, making Slick a perfect fit for your reactive applications based on Play and Akka.

What is FRM?

- FRM = Functional Relational Mapping (opposed to ORM)
- Slick allows you to process persistent relational data stored in DB tables the same (functional) way as you do with in-memory data, i.e. Scala collections.
- Table Queries are monads.
- Table Queries are pre-optimized.
- Table Queries are type-safe.
- The Scala compiler complains, if you specify your table query incorrectly.

TableQuery s are Monads.

- ***filter()*** for the selection of data
- ***map()*** for the projection
- ***flatMap()*** to pass the output of your 1st DB operation as input to the 2nd DB operation.
- With the provision of these three monadical functions *TableQuery* s are monads.
- Hence you can also use the syntactic sugar of ***for-comprehensions***.
- There is more. E.g. the ***sortBy()*** function allows you to define the sort order of your query result.

How to execute a Slick DB Action?

- Define a TableQuery

```
val query: Query[…] = TableQuery(…)…
```

- For this TableQuery, define a database action which might be a query, insert, bulk insert, update or a delete action or even a DDL action.

```
val dbAction: DBIO[…] = query += record                                    // insert
val dbAction: DBIO[…] = query ++= Seq(row0, row1, …, rowN)       // bulk insert
val dbAction: DBIO[…] = query.update(valuesToUpdate)                  // update
val dbAction: DBIO[…] = query.delete                                                  // delete
val dbAction: DBIO[…] = query.result                                                  // insert
```

- Run the database action by calling *db.run(dbAction)*. *db.run* never returns the *Result*. You always get a *Future[Result]*.

```
val dbAction: DBIO[…] = TableQuery(…).result
val future: Future[…] = db.run(dbAction)
```

Why is Slick reactive?

- It is asynchronous and non-blocking.
- It provides its own configurable thread pool.
- If you run a database action you never get the *Result* directly. You always get a *Future[Result]*.

```
val dbAction: DBIOAction[Seq[String]] = TableQuery(…).result

val future: Future[Seq[String]] = db.run( dbAction )
```

- Slick supports Reactive Streams. Hence it can easily be used together with Akka-Streams (which is not subject of this talk).

```
val dbAction: StreamingDBIO[Seq[String]] = TableQuery(…).result

val publisher: DatabasePublisher[String] = db.stream( dbAction )

val source: Source = Source.fromPublisher( publisher )

// Now use the Source to construct a RunnableGraph. Then run the graph.
```

Short Reminder of Scala Futures

In Slick every database access returns a Future.

How can one async (DB) function process the result of another async (DB) function?

This scenario happens very often when querying and manipulating database records.

How to process the Result of an Async Function by another Async Function

Using ***Future.flatMap***:

```
def doAAsync(input: String): Future[A] = Future { val a = f(input); a }

def doBAsync(a: A): Future[B] = Future { val b = g(a); b }

val input = "some input"

val futureA: Future[A] = doAAsync(input)

val futureB: Future[B] = futureA flatMap { a => doBAsync(a) }

futureB.foreach { b => println(b) }
```

Async Function processing the Result of another Async Function

Using a **for-comprehension**:

```
def doAAsync(input: String): Future[A] = Future { val a = f(input); a }
def doBAsync(a: A): Future[B] = Future { val b = g(a); b }
val input = "some input"
val futureB: Future[B] = for {
    a <- doAAsync(input)
    b <- doBAsync(a)
} yield b
futureB.foreach { b => println(b) }
```

A Simple Slick Example

Activator Template: ***hello-slick-3.1***

Table Definition with a Tuple

Tuple

```
class Users(tag: Tag) extends Table[(String, Option[Int])](tag, "USERS") {


  // Auto Increment the id primary key column
  def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)


  // The name can't be null
  def name = column[String]("NAME", O.NotNull)


  // the * projection (e.g. select * ...)
  def * = (name, id.?)
}
```

Table Definition with Case Class Mapping

```scala
case class User(name: String, id: Option[Int] = None)


class Users(tag: Tag) extends Table[User](tag, "USERS") {

  // Auto Increment the id primary key column
  def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)


  // The name can't be null
  def name = column[String]("NAME", O.NotNull)


  // the * projection (e.g. select * ...) auto-transforms the tupled
  // column values to / from a User
  def * = (name, id.?) <> (User.tupled, User.unapply)
}
```

Map Tuple to User

Map User to Tuple

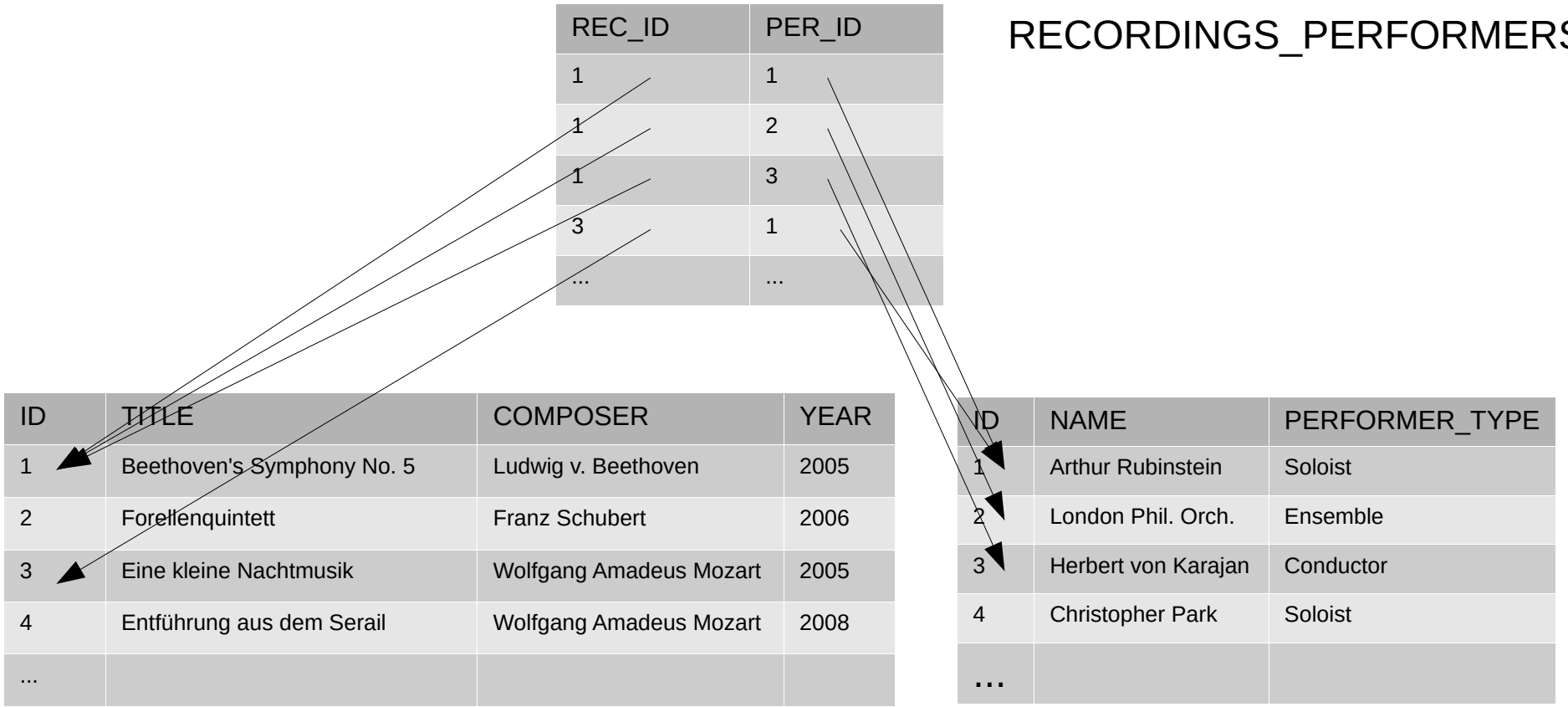
Part 2: Many-to-many with Slick

- The Demo App: *MusicService*
- Many-to-many Relationship (DB Schema)
- Web Application Demo
- Many-to-many in Slick Table Definitions
- Ensuring Referential Integrity
- Adding and Deleting Relationships
- Traversing the many-to-many Relationship in Queries

The Demo App: *MusicService*

- The MusicService manages Music ***Recording***s and ***Performer***s.
- A Recording is ***performedBy*** many (0 … n) Performers.
- A Performer is ***performingIn*** many (0 … n) Recordings.

Many-to-many in the DB Schema



REC_ID	PER_ID
1	1
1	2
1	3
3	1
...	...

RECORDINGS_PERFORMERS

ID	TITLE	COMPOSER	YEAR
1	Beethoven's Symphony No. 5	Ludwig v. Beethoven	2005
2	Forellenquintett	Franz Schubert	2006
3	Eine kleine Nachtmusik	Wolfgang Amadeus Mozart	2005
4	Entführung aus dem Serail	Wolfgang Amadeus Mozart	2008
...			

RECORDINGS

ID	NAME	PERFORMER_TYPE
1	Arthur Rubinstein	Soloist
2	London Phil. Orch.	Ensemble
3	Herbert von Karajan	Conductor
4	Christopher Park	Soloist
...		

PERFORMERS

Web Application Demo

MusicService is a Play Application with a rather primitive UI. This interface allows the user to …

- Create, delete, update Performers and Recordings
- Assign Performers to Recordings or Recordings to Performers and delete these Assignments
- Query / Search for Performers and Recordings
- Play Recordings

Many-to-many Relationship in the Slick Table Definitions

```
case class RecordingPerformer(recId: Long, perId: Long)


// Table 'RecordingsPerformers' mapped to case class 'RecordingPerformer' as join table to map
// the many-to-many relationship between Performers and Recordings
//
class RecordingsPerformers(tag: Tag)
      extends Table[RecordingPerformer](tag, "RECORDINGS_PERFORMERS") {

  def recId: Rep[Long] = column[Long]("REC_ID")
  def perId: Rep[Long] = column[Long]("PER_ID")

  def * = (recId, perId) <> (RecordingPerformer.tupled, RecordingPerformer.unapply)
  def pk = primaryKey("primaryKey", (recId, perId))

  def recFK = foreignKey("FK_RECORDINGS", recId, TableQuery[Recordings])(recording =>
    recording.id, onDelete=ForeignKeyAction.Cascade)

  def perFK = foreignKey("FK_PERFORMERS", perId, TableQuery[Performers])(performer =>
    performer.id)

  // onUpdate=ForeignKeyAction.Restrict is omitted as this is the default
}
```

Ensuring Referential Integrity

- Referential Integrity is guaranteed by the definition of a ***foreignKey()*** function in the referring table, which allows to navigate to the referred table.
- You can optionally specify an ***onDelete*** action and an ***onUpdate*** action, which has one of the following values:
  - *ForeignKeyAction.NoAction*
  - *ForeignKeyAction.Restrict*
  - *ForeignKeyAction.Cascade*
  - *ForeignKeyAction.SetNull*
  - *ForeignKeyAction.SetDefault*

Adding and Deleting Relationships

- Adding a concrete relationship == Adding an entry into the Mapping Table, if it doesn't already exist.
- Deleting a concrete relationship == Deleting an entry from the Mapping Table, if it exists.
- Updates in the Mapping Table do not make much sense. Hence I do not support them im my implementation.

Traversing many-to-many the Relationship in Queries

- The ***Query.join()*** function allows you to perform an inner join on tables.
- The ***Query.on()*** function allows you to perform an inner join on tables.
- Example:

```
val query = TableQuery[Performers] join TableQuery[RecordingsPerformers] on (_.id === _.perId)

val future: Future[Seq[(Performer, RecordingPerformer)]] = db.run { query.result }

// Now postprocess this future with filter, map, flatMap etc.

// Especially filter the result for a specific recording id.
```

Links

- Slick Website:
  http://slick.typesafe.com/doc/3.1.1/
- Slick Activator Template Website with Tutorial:
  https://www.typesafe.com/activator/template/hello-slick-3.1
- Slick Activator Template Source Code:
  https://github.com/typesafehub/activator-hello-slick#slick-3.1
- MusicService Source Code and Slides:
  https://github.com/hermannhueck/MusicService/tree/master/Services/MusicService-Play-Scala-Slick-NoAuth
- Authors XING Profile:
  https://www.xing.com/profile/Hermann_Hueck

Thank you for listening!

Q & A