Practical Slick:

tame your SQL with the power of Scala!

daniele gallingani

28/10/2015

WhoAmI

Now

Software Engineer @ buildo

Scala, DevOps, Stuff

Past

Mobile Security, Full Stack

@danigallingani


I THINK WE SHOULD BUILD AN SQL DATABASE.
UH-OH
S. Adams
E-mail: SCOTTADAMS@AOL.COM
DOES HE UNDERSTAND WHAT HE SAID OR IS IT SOMETHING HE SAW IN A TRADE MAGAZINE AD?
11/17
© 1995 United Feature Syndicate, Inc.(NYC)
WHAT COLOR DO YOU WANT THAT DATABASE?
I THINK MAUVE HAS THE MOST RAM.

Relational data management patterns?

ORM

object + relational = impedance mismatch

→ inheritance, transactions

→ performance issues, lack of fine grained control

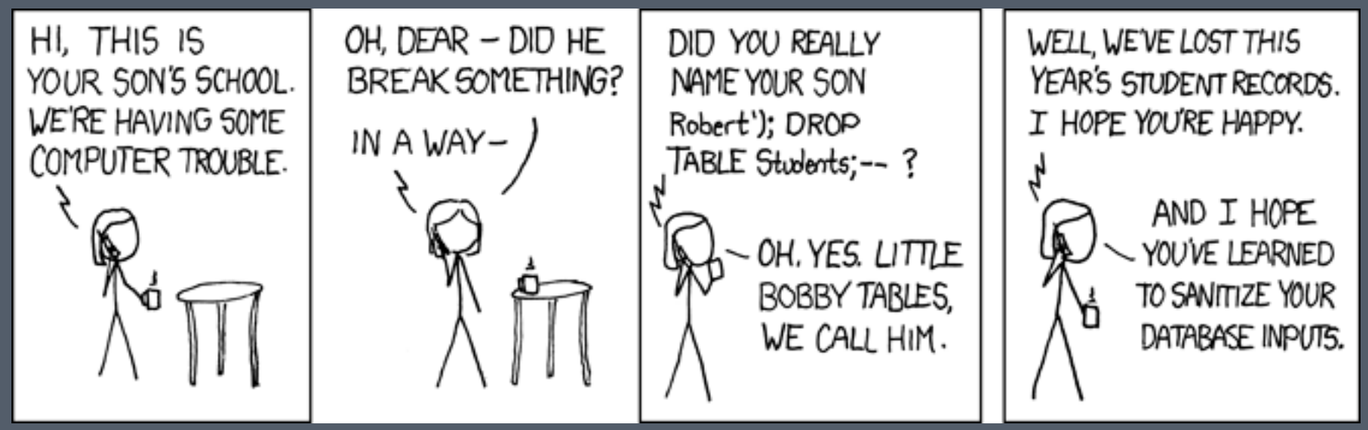
Object-Relational Mapping is the Vietnam of Computer Science

Relational data management patterns? (cont'd)

Raw SQL

→ verbose, poor composability, dialects

→ no static checks, no preparation enforced



A clever person solves a problem. A wise person avoids it.

— A. Einstein

What is Slick then?

not an ORM

not Raw SQL

Functional Relational Mapping

What is Slick then?

(from doc)

Slick is a modern **database query and access library** for Scala.

You can write your database queries in **Scala instead of SQL**, thus profiting from the **static checking**, **compile-time** safety and **compositionality** of Scala.

Cool, but… which DBs?

→ PostgreSQL

→ MySQL

→ H2

→ Hsqldb

→ Derby / JavaDB

→ SQLite

Cool, but... which (commercial) DBs?

→ Oracle *

→ DB2

→ SQL Server *

Outline

→ SQL <-> Scala mapping (DDL)

→ Querying

→ Data manupulation

→ Gotchas & Caveats

→ ~~CAP Theorem~~

SQL ↔ Scala mapping

Table definition

```
class Matches(tag: Tag) extends Table[Match](tag, "matches") {
  def _id = column[Int]("id", O.PrimaryKey, O.AutoInc)
  def _homeTeamId = column[String]("home_team_fk")
  def _awayTeamId = column[String]("away_team_fk")
  def matchDate = column[DateTime]("match_date")
  def score = column[Option[String]]("score") // nullable
}
```

as we were writing...

```
CREATE TABLE `matches` (`_id` INT NOT NULL AUTO_INCREMENT, ...
```

...feeling neat

Default projection

```
class Matches(tag: Tag) extends Table[Match](tag, "matches") {
  ...

  def * = (_id.?,
           _homeTeamId,
           _awayTeamId,
           matchDate,
           score) <> (Match.tupled, Match.unapply)
}

case class Match(
  _id: Option[Int],
  _homeTeamId: String,
  _awayTeamId: String,
  matchDate: DateTime,
  score: Option[String]
)
```

Tuples, HLists or Case Classes!

Default projection

```
class Matches(tag: Tag) extends Table[Match](tag, "matches") {
  ...

  def * = (_id.?,
           _homeTeamId,
           _awayTeamId,
           matchDate,
           score) <> (Match.tupled, Match.unapply)
}

case class Match(
  _id: Option[Int],
  _homeTeamId: String,
  _awayTeamId: String,
  matchDate: DateTime,
  score: Option[String]
)
```

`_id` VS `score` Options

Foreign keys

```
{ ...
  def homeTeam =
    foreignKey("match_home_team_fk", _homeTeamId, TableQuery[Teams])
      (_._id, onUpdate=ForeignKeyAction.Cascate, onDelete=ForeignKeyAction.NoAction)

  def awayTeam =
    foreignKey("match_away_team_fk", _awayTeamId, TableQuery[Teams])
      (_._id, onUpdate=ForeignKeyAction.Cascate, onDelete=ForeignKeyAction.NoAction)
... }
class Teams(tag: Tag) extends Table[Team](tag, "teams") {
  def _id = column[String]("id", O.PrimaryKey)
  def name = column[String]("name", O.NotNull)
  def * = (_id,name) <> (Team.tupled, Team.unapply)
}
```

No nested objects!

Indexes

Simple

```
def matchDateIndex = index("match_date_idx", matchDate)
```

Composite

```
def teamsIndex = index("home_away_idx", (_homeTeamId, awayTeamId))
```

unique ✓

asc, desc, full text ✗

```
class Matches(tag: Tag) extends Table[Match](tag, "matches") {
  def _id = column[Int]("id", O.PrimaryKey, O.AutoInc)
  def _homeTeamId = column[String]("home_team_fk")
  def _awayTeamId = column[String]("away_team_fk")
  def matchDate = column[DateTime]("match_date")
  def score = column[Option[String]]("score") // nullable

  def homeTeam =
    foreignKey("match_home_team_fk", _homeTeamId, TableQuery[Teams])(_._id)

  def awayTeam =
    foreignKey("match_away_team_fk", _awayTeamId, TableQuery[Teams])(_._id)

  def matchDateIndex = index("match_date", matchDate)

  def * = (_id.?,
           _homeTeamId,
           _awayTeamId,
           _teamTournamentId,
           matchDate,
           score) <> (Match.tupled, Match.unapply)

}

case class Match(_id: Option[Int], _homeTeamId: String, _awayTeamId: String, matchDate: DateTime, score: Option[String])
```

Custom types

```
// ADT for results
sealed trait MatchResult
object MatchResult {
  case object HomeWon extends MatchResult
  case object AwayWon extends MatchResult
  case object Draw extends MatchResult
  case object Pending extends MatchResult
}

def result = column[MatchResult]("result")

implicit def matchResultMapper =
 MappedColumnType.base[MatchResult, String](
    serialize,
    deserialize
  )
```

DB-Specific parameters

```
def runningCommentary = column[String](
    "running_commentary",
    O.DBType("TEXT"),
    O.NotNull,
    O.Default("Game not started yet")
)

def createdOn = column[DateTime](
  "created_on",
   O.DBType("timestamp default current_timestamp")
)
```

`String` default mapping is `VARCHAR(255)` in MySQL

More about DDL

```
val db =
  Database.forURL("jdbc:mysql:...")
val matches = new Matches()
val teams = new Teams()
val schema = matches.schema ++ teams.schema

db.run(Actions.seq(schema.drop, schema.create))

// schema.{create, drop}.statements to access statements
```

Schema code generation!

Querying

Simplest query ever

```
val allMatches: Future[Seq[Match]] =
  db.run(matches.result)
```

Futures!

`result` converts a `Query` into a `DBIOAction`

Basic operations

```
val matchIds: Future[Seq[Option[Int]]] = // projection
  db.run(matches.map(_._id).result)

val teamIdsOfACoolMatch: Future[Option[(String, String)]] =
  db.run(
    matches
      .filter(_._id `===` 42) // selection
      .map(m => (m._homeTeamId, m.awayTeamId)) // projection
      .result
      .headOption
  )
```

Inner Join

```
val homeTeamOfThatCoolMatchQuery: Query[] =
  for {
    (m, ht) <- matches join teams on (_._homeTeamId === _._id)
               if m._id === 42
  } yield ht

val homeTeamOfThatCoolMatchQuery2 =
  for {
    m <- matches if m._id === 42
    ht <- m.homeTeam
  } yield ht
```

Inner Join

```
val homeTeamOfThatCoolMatchQuery3 =
  for {
    m <- matches if m._id === 42
    ht <- teams if m._homeTeamId === ht._id
  } yield ht

val homeTeamOfThatCoolMatchQuery4 =
  matches.filter(_._id === 42).map(_.homeTeam)
```

joinLeft, joinRight, joinFull

Aggregations

```
val nextMatchDate =
  matches
    .filter(_.matchDate > DateTime.now)
    .map(_.matchDate)
    .min

val teamHomeMatchesCount =
  matches
    .filter(_._homeTeamId === "Italy")
    .length


val q = (for {
  m <- matches
  ht <- m.homeTeam if ht._id === "Italy"
} yield m).groupBy(_._awayTeamId)

q.map { case (awayTeamId, rest) =>
  (awayTeamId, rest.length)
}
```

Other Queries

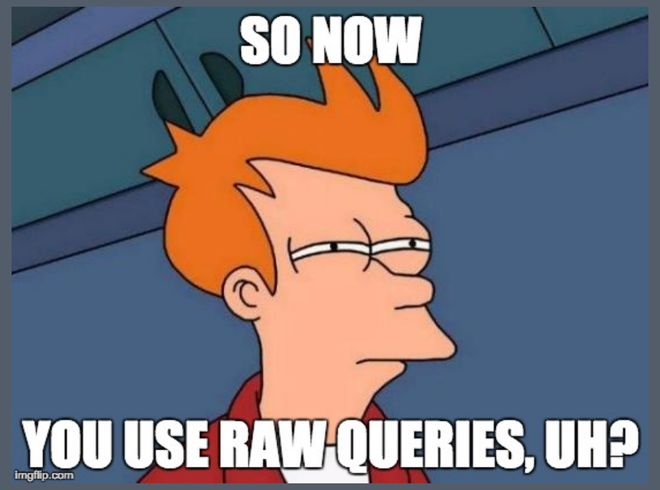
```
val matchesWithTeams: Future[Seq[(Match, Team, Team)]] =
  db.run(
    (for {
      m <- matches
      ht <- m.homeTeam
      at <- m.awayTeam
    } yield (m, ht, at)).sortBy(_._1.matchDate.desc)
  )
```

SQL String interpolation

```
def q(id: String) = sql"SELECT (_id, name) FROM `teams` where _id = $id".as[Team]

val italyQ = q("Italy")
```

Prepare your own!



Data Manipulation

Inserts

```
val teamsToInsert = Seq(
  Team("ACMILAN", "AC Milan"),
  Team("FCINTER", "Inter")
)
db.run(
  teams ++= teamsToInsert // returns number of inserted rows
)

val qReturningId =
  (teams returning teams.map(_._id)) += teamsToInsert(1)
```

Updates

```
db.run(
  teams
    .filter(_._id === "INTER")
    .map(_.name)
    .update("FC Inter")
)
// returns number of updated rows
```

Upserts are available at your own risk!

Deletes

```
db.run(
  matches
    .filter(m => m._id === 42 || m._id === 24)
    .delete
)
// returns number of deleted rows
```

Transactions

```
val q1 = teams.filter(_._id === "Italy").map(_._id).update("ITA")
val q2 = matches.filter(_._homeTeamId === "Italy").map(_._homeTeamId).update("ITA")
val q3 = matches.filter(_._awayTeamId === "Italy").map(_._awayTeamId).update("ITA")

val attempt = DB.seq(q1, q2, q3)

val better = DB.seq(q1, q2, q3).tranactionally
```

suppose `onUpdate NO ACTION`

I didn't come up with a better example

Gotchas & Caveats

User-Defined Functions

```
def dayOfWeek(c: Rep[Date]) =
  SimpleFunction.unary[Date, Int]("day_of_week")
```

Useful for DB-Specific missing mappings

Stuff

→ since slick 3.1 no more nested queries

→ since slick 3 you have an embedded Connection Pooler

→ debug query compilation: print statements

→ streaming APIs

→ Slick Test Kit

That's all, questions?

Thanks!