Supporting your data model with Slick

Jan Christopher Vogt
About me

- ~ 3 years software engineer in the EPFL Scala team
- Working on Slick together with Typesafe
- Recently started part time at Sport195 in NYC
- Previously freelance for web platforms
- Background in programming languages, web dev, databases, python, pragmatic functional programming, software quality, automated testing
Slick (vs. ORM)

- Functional-Relational Mapper
- natural fit (no impedance mismatch)
- declarative
- embraces relational
- stateless
- Slick is to ORM what Scala is to Java
Part 1
8 practical reasons for using Slick
1

Scala collection-like API
Scala collection-like API

```scala
for ( d <- Devices;
     if d.price > 1000.0 )
  yield d.acquisition

Devices
  .filter(_.price > 1000.0)
  .map(_.acquisition)
```

**Device**
- `id`: Long
- `price`: Double
- `acquisition`: Date
2
Predictable SQL structure
Predictable SQL structure

Devices

filter(_.price > 1000.0)
.map(_.acquisition)
.selectStatement

select x2."ACQUISITION"
from "DEVICE" x2
where x2."PRICE" > 1000.0
3
Type-safety
Compile-Time Safety

- Spelling mistake in column name?
- Wrong column type?
- Query doesn’t match expected result type?

scalac sees it all!
But: Error messages can be bad

Piotr Buda @piotrbuda
...and the 'Most Informative Stack Trace Award goes to...' evernote.com/shard/s28/sh/5... #slick #scala
12 hours ago
Enforce schema consistency

- Generate DDL from table classes
- Slick 2.x: Generate table classes and mapped classes from database
4
Small configuration using Scala code
class Devices(tag: Tag) extends Table[(Long, Double, Date)](tag,"DEVICES") {
  def id = column[Long] ("ID", 0.PrimaryKey)
  def price = column[Double]("PRICE")
  def acquisition = column[Date] ("ACQUISITION")
  def * = (id, price, acquisition)
}
def Devices = TableQuery[Devices]

can be auto-generated in Slick 2.x
import scala.slick.driver.H2Driver.simple._

val db = Database.forURL("jdbc:h2:mem:testdb", "org.h2.Driver")

db.withTransaction { implicit session =>
  // <- run queries here
}
5
Explicit control over execution and transfer
Execution control

```scala
val query = for {
  d <- Devices
  if d.price > 1000.0
} yield d.acquisition

db.withTransaction { implicit session =>
  val acquisitionDates = query.run

  // no unexpected behavior,
  // no loading strategy configuration,
  // just write code
```
6
Loosely-coupled, flexible mapping
Mapping to tuples

```scala
class Devices(tag: Tag)
extends Table[(Long, Double, Date)](tag,"DEVICES") {
  def id = column[Long] ("ID", O.PrimaryKey)
  def price = column[Double]("PRICE")
  def acquisition = column[Date] ("ACQUISITION")
  def * = (id, price, acquisition)
}
val Devices = TableQuery[Devices]
```
Mapping to HLists

class Devices(tag: Tag)
extends Table[Long :: Double :: Date :: HNil](tag,"DEVICES") {
  def id = column[Long] ("ID", O.PrimaryKey)
  def price = column[Double]("PRICE")
  def acquisition = column[Date] ("ACQUISITION")
  def * = id :: price :: acquisition :: HNil
}
val Devices = TableQuery[Devices]
Mapping to case classes

case class Device(id: Long,
    price: Double,
    acquisition: Date)

class Devices(tag: Tag)
extends Table[Device](tag,"DEVICES") {
  def id = column[Long] ("ID", O.PrimaryKey)
  def price = column[Double]("PRICE")
  def acquisition = column[Date] ("ACQUISITION")
  def * = (id, price, acquisition) <>
    (Device.tupled,Device.unapply)
}
val Devices = TableQuery[Devices]
Mapping to case classes

def construct : ((Long,Double,Date)) => CustomType
def extract: CustomType => Option[(Long,Double,Date)]

class Devices(tag: Tag)
  extends Table[CustomType](tag,"DEVICES") {
    def id = column[Long] ("ID", O.PrimaryKey)
    def price = column[Double]("PRICE")
    def acquisition = column[Date] ("ACQUISITION")
    def * = (id, price, acquisition) <>
               (construct,extract)
  }
val Devices = TableQuery[Devices]
7

First-class SQL support
Plain SQL support

```scala
import scala.slick.jdbc.{GetResult, StaticQuery}
import StaticQuery.interpolation

implicit val getDeviceResult =
  GetResult(r => Device(r.<<, r.<<, r.<<))

val price = 1000.0

val expensiveDevices: List[Device] =
  sql"select * from DEVICES where PRICE > $price"
    .as[Device].list
```
8

composable /
re-usabe queries
Composable queries

```scala
def deviceLocations (companies: Query[Companies,Company]) : Query[Column[String],String] = {
  companies.computers.devices.sites.map(_.location)
}

val apples = Companies.filter(_.name iLike "%apple%")
val locations : Seq[String] = {
  deviceLocations(apples)
    .filter(_.inAmerica: Column[String]=>Column[Boolean])
    .run
}
```
Composable queries

```scala
def deviceLocations (companies: Query[Companies,Company]): Query[Column[String],String] = {
  companies.computers.devices.sites.map(_.location)
}

val apples = Companies.filter(_.name iLike "%apple%")
val locations : Seq[String] = {
  deviceLocations(apples)
    .filter(_.inAmerica: Column[String]=>Column[Boolean])
    .run
}
```

Re-use queries

Freitag, 27. Juni 14
Composable queries

```scala
def deviceLocations
  (companies: Query[Companies, Company])
  : Query[Column[String], String] = {
    companies.computers.devices.sites.map(_.location)
  }

val apples = Companies.filter(_.name iLike "%apple%")
val locations = Seq[String] = {
  deviceLocations(apples)
  .filter(_.inAmerica: Column[String] => Column[Boolean])
  .run
}
```

Re-use joins
Composable queries

```scala
def deviceLocations (companies: Query[Companies,Company]) : Query[Column[String],String] = {
  companies.computers.devices.sites.map(_.location)
}

val apples = Companies.filter(_.name iLike "%apple%")
val locations : Seq[String] = {
  deviceLocations(apples)
    .filter(_.inAmerica: Column[String]=>Column[Boolean])
    .run
}
```
Composable queries

def deviceLocations
  (companies: Query[Companies,Company])
  : Query[Column[String],String] = {
    companies.computers.devices.sites.map(_.location)
  }

val apples = Companies.filter(_.name iLike "%apple%")
val locations : Seq[String] = {
  deviceLocations(apples)
  .filter(_.inAmerica: Column[String]=>Column[Boolean])
  .run
}

exactly one db roundtrip
Composable queries

```scala
def deviceLocations(companies: Query[Companies, Company]): Query[Column[String], String] = {
  companies.computers.devices.sites.map(_.location)
}

val apples = Companies.filter(_.name iLike "%apple%")
val locations: Seq[String] = {
  deviceLocations(apples)
    .filter(_.inAmerica: Column[String] => Column[Boolean])
    .run
}
```
let’s take a step back...
Part 2

Software data modeling
What are we doing?
We model a part of reality

... or fiction
The model is NOT in a single place of our code
It's all over the place
Examples

**db schema**

```sql
create table "COMPUTER" (  
  "ID" INTEGER PRIMARY KEY,  
  "NAME" VARCHAR NOT NULL,  
  "INTRODUCED" DATE,  
  "DISCONTINUED" DATE,  
  "COMPANY_ID" INTEGER
);
```

**Slick Table**

```scala
class Computers(tag: Tag) extends Table[Computer](tag, "COMPUTER") {  
  def * = (name, introduced, discontinued, companyId, id.?) <> ...  
  val name = column[String]("NAME")  
  val introduced = column[Option[java.sql.Date]]("INTRODUCED")  
  val discontinued = column[Option[java.sql.Date]]("DISCONTINUED")  
  val companyId = column[Option[Int]]("COMPANY_ID")  
  val id = column[Int]("ID", O.AutoInc, O.PrimaryKey)  
}
```

**Scala case class**

```scala
case class Computer(  
  name: String,  
  introduced: Option[java.sql.Date],  
  discontinued: Option[java.sql.Date],  
  companyId: Option[Int],  
  id: Option[Int] = None)
```

**Form**

```scala
Form(  
  mapping(  
    "name" -> nonEmptyText,  
    "introduced" -> optional(sqlDate("yyy-MM-dd")),  
    "discontinued" -> optional(sqlDate("yyy-MM-dd")),  
    "companyId" -> optional(number),  
    "id" -> optional(number)  
  )(Computer.apply)(Computer.unapply)
)
```

**Play form / html**

```scala
@inputText(computerForm("name"), _label -> "Computer name")
@inputText(computerForm("introduced"), _label -> "Introduced date")
@inputText(computerForm("discontinued"), _label -> "Discontinued date")
```
Why the repetition?
Why the repetition

- Language limitations
- Language / system borders
- Avoiding complicated types in abstractions
- Separation of concerns (e.g. Frontend / Backend)
Problems of repetition

- Bad out of the box experience
- Implementation effort
- Maintenance effort (refactoring, etc.)
- Inconsistencies!
- Repeated bugs
Let’s refactor
Data model driven software

- Slick Table
- Scala case class
- Play forms / html
- database schema
Wait... didn’t model driven fail?
Visual tool driven?

- Slick Table
- Scala case class
- Play forms / html
- database schema
Scala code driven?

- hand-written
- Scala case class + annotations

auto-generated

- Slick Table
- Scala case class
- Play forms / html
- database schema

needs migrations
Database schema driven?

managed by hand

auto-generated

Slick Table
Scala case class
Play forms / html

database schema
New in Slick 2
Slick code generation
Slick out-of-the-box codegen

`scala.slick.model.codegen.SourceCodeGenerator`

registered as a sourceGenerator or manually

your sbt project

Template: https://github.com/slick/slick-codegen-example
Slick out-of-the-box codegen

- generates all types for slick queries
- minimal customization may be required
- textual codegen (not Scala macros)
Slick out-of-the-box codegen

Model("Computers",
columns = Seq(
  Column("ID"),
  ...
),
)

Slick Model

database schema

database schema

Slick Table

Scala case class

Play forms / html

database schema

database schema

Template: https://github.com/slick/slick-codegen-example

Freitag, 27. Juni 14
Slick customized code generation
Generate whatever

- play forms
- DAO
- gui
- ...

Freitag, 27. Juni 14
Slick customized codegen

sbt multi-project build

codegen/CustomizedCodeGenerator.scala

codegen project

dependsOn

registered as a sourceGenerator or manually

main project

Template: https://github.com/slick/slick-codegen-customization-example

Freitag, 27. Juni 14
Textual codegen vs. Macros

- Macros are compiler-supported codegen
- Easier multi-stage expansions
- QuasiQuotes provide early syntax errors

However
- currently no preview of generated code
- some compiler api knowledge requires, e.g. names
// fetch data model
val model = db.withSession{ implicit session =>
  createModel(H2Driver.getTables.list, H2Driver)
}

// customize code generator
val codegen = new SourceCodeGenerator(model){
  override def code =
    "import foo.{MyCustomType, MyCustomTypeMapper}" + "\n" + super.code

  // override table generator
  override def Table = new Table(_){
    // disable entity class generation and mapping
    override def EntityType = new EntityType{
      override def classEnabled = false
    }
  }

  // override contained column generator
  override def Column = new Column(_){
    override def rawType =
      if(model.name == "SOME_SPECIAL_COLUMN_NAME") "MyCustomType"
      else super.rawType
  }
}
Slick SourceCodeGenerator

- allows very easy start
- simple customizations
- override methods like `def code`
Fully automatic Play CRUD demo app:

https://github.com/slick/play-slick-codegen
Demo app codegen features

- case classes
- Slick Tables
- Play form bindings / validations
- Play html view helpers / formatters / forms
- JavaScript form validation
- Many-to-one relationships in forms
All this, but at what price?

vanilla app
play-slick / computer-database
app/
hand-written: 1114 LOC

this demo app
slick / play-slick-codegen
app/
hand-written: 1148 LOC
generated: 228 LOC
slick-codegen/
hand-written: 204 LOC

total: 1352 LOC
Real world case study
Sport195

- www.sport195.com
- Sports social network - Athlete, Fan, Organization
- Sport data provider / content platform
- REST api using Scala/Slick/Play
- 107 tables, 1120 columns mapped using Slick, shared with RoRails app
- migrated from Slick 1 -> Slick 2 -> Slick 2 + codegen
hand-written -> codegen

- initial migration of code took ~3 weeks (107 tables)
  - wrong types (4 cases)
  - wrong nullability (109 cases in 66 tables)
  - wrong / missing column (few cases)
- after that new features for all tables 1-3 days
Generated features at S195

- case class-like classes (>22 cols)
- Slick Tables
- CRUD / with hooks
- typed associations
- polymorphic associations
- json serialization / deserialization
Sport195 codegen benefits

all model code for 107 tables, 1120 columns

**before codegen**

- Model-specific: 15127 LOC
- Abstractions: 781 LOC
- Scala macros: 309 LOC
- total: 16217 LOC

**using codegen**

- Model-specific: 10698 LOC
- Abstractions: 615 LOC
- Scala macros: 0 LOC
- Code generator: 399 LOC
- Code template: 301 LOC
- total: 12013 LOC
- generated: 37542 LOC

hand-written: 25% reduction

Freitag, 27. Juni 14
S195 codegen architecture

- Customized Slick codegen architecture

- Slick Table
- Scala case class
- Model
  - "computers"
  - columns = Seq( Column("ID"), ...
  - )
- ExtraMetaData
  - "computers"
  - customizeEntity=true
- hand-written extra meta data
- automatically obtained from DBMS
- jdbc metadata
- managed with migrations

code-template
- (SOC, syntax highlighting, etc.)
- generated code (never changed by hand)
- hand-written code
- interop via interfaces / inheritance
S195 additional meta data

complement your database schema as required

case class ExtraMetaData(
  table: String, // <- tie to db schema
  entityClassName: Option[String] = None,
  tableClassName: Option[String] = None,
  blacklistedColumns: Seq[String] = Seq(),
  overrideDefaultValues: Map[String, Default] = Map(), // literal or code
  mapColumnNames: Map[String, String] = Map(),
  tableParent: String = "RichTable",
  customizeEntityCompanion: Boolean = false,
  customizeTableBase: Boolean = false,
  associations: Option[Either[SimpleAssociation, PolyAssociation]] = None
)
Practical codegen tips
Never change generate code by hand
Never change generate code by hand

- keep codegen repeatable and evolvable
- change any of these instead of generated code:
  - code-generator
  - database schema
  - extra meta data
2
Codegen only if you have to
Initial cost of codegen

- more complex build
- more complex architecture for interop
If possible don’t codegen

- Keep it simple
- Generated code is often harder to maintain than hand-written (unless it is repetitive)
- Don’t codegen rare edge-cases, just write them by hand
- Abstract in Scala to support further abstractions
  - e.g. for Scala tuples, codegen breaks abstraction
When to codegen?

- as refactoring
- when forced to repeat at least once or twice
- usual suspects
  - entity members (case classes, slick tables, etc.)
  - tuple sizes (tables > 22)
  - type-system limitations (constructor inheritance)
3

Have excellent interop
hand-written <-> generated
interop
hand-written <-> generated

- Don’t capture all edge-cases. Allow customization!
- Many ways: inheritance, apis, type classes
- Care about it! Avoid stuff creeping into codegen
- Use extra meta data for customization indicators
S195 codegen interop: Athlete

generated code: interfaces

AthleteBase

AthleteCompanion Base

AthleteTableBase

hand-written code: customizations

AthleteCustomized

AthleteCompanion Customized

AthleteTableBase

generated code: tying the knot

class Athlete (constructor)

object Athlete def apply

class AthleteTable extends Table with ...
The generator is not just a tool. It’s part of your code.
Part of your code

- integral part of your code!
- be agile, evolve your generator alongside your code
- keep refactoring
- put both in version control together
Scale generator as needed

- start easy
  - override def code / use string interpolation
- advance: pull out code into separate template, e.g. twirl
  - separation of concerns
  - syntax highlighting (highlight template as Scala)
- transcend: say goodbye to Slick’s codegen class and use Slick’s model exclusively
5
Put generated sources or schema in version control
versioning generated code

- for very understandable diffs
- for checking white-space/docs changes
- allow compile without db
versioning meta data instead

- e.g. schema.sql file
- (atm: don’t use different db for codegen and prod, jdbc drivers are too different)
6
make generated code readable!
indention & scaladoc
Consider exposing your schema in your webservice
For backend/frontend teams

- expose the schema in your api for re-use
- e.g. /computer/schema
- or generate javascript that represents the schema
CodeGen summary

- Consider codegen to scrap your boiler plate
- It’s one way to do it. There are others.
- It works! Even for small projects. And it’s easy.
- Use it wisely.
- Enjoy productivity benefits :)
Thank you!

We are hiring at Sport195. Talk to me.

christopher.vogt@sport195.com

twitter: @cvogt

slick: http://slick.typesafe.com/