Leveraging your data model with Slick 2

code generation and other features

Jan Christopher Vogt

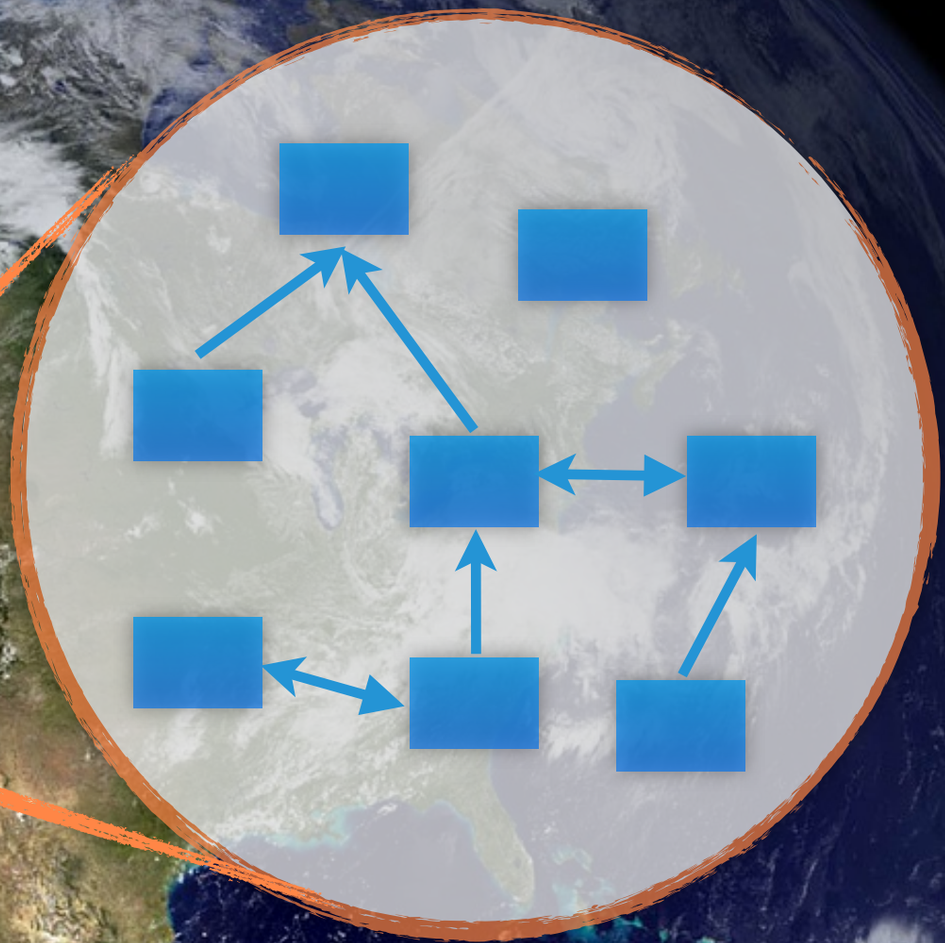
Before we get to Slick 2 features like codegen,

let’s take a step back...

Software data models

What are we doing?

We model a part of reality



... or fiction

Image source: http://pixabay.com

The model is NOT in a single place of our code

Slick

Play

Validation

DAO

API

It’s all over the place

Serialization

SQL

Scala

GUI

Examples

db schema

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

Slick Table

```
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

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

Play form / html

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

```
@inputText(computerForm("name"), '_label -> "Computer name")
@inputText(computerForm("introduced"), '_label -> "Introduced date")
@inputText(computerForm("discontinued"), '_label -> "Discontinued date")
```

Slick

Play

Validation

DAO

API

Why the repetition?

Serialization

SQL

Scala

GUI

Why the repetition

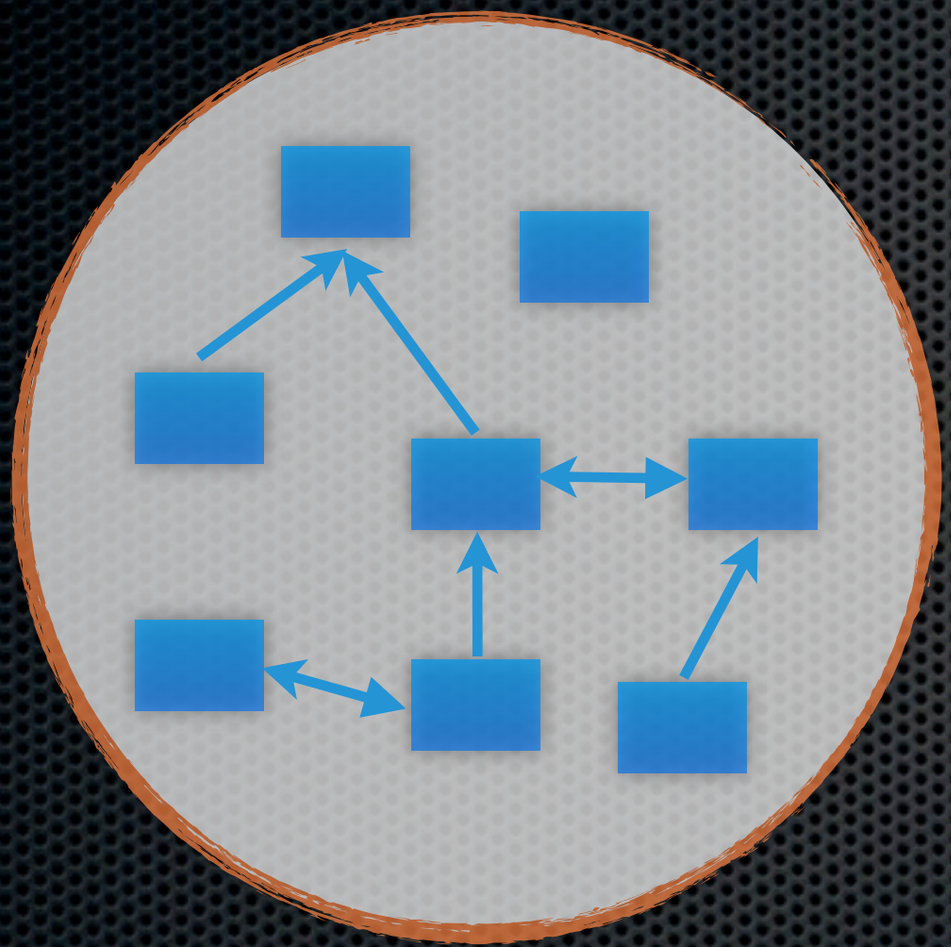
- Language limitations
- Avoiding complicated types in abstractions
- Separation of concerns (e.g. Frontend / Backend)

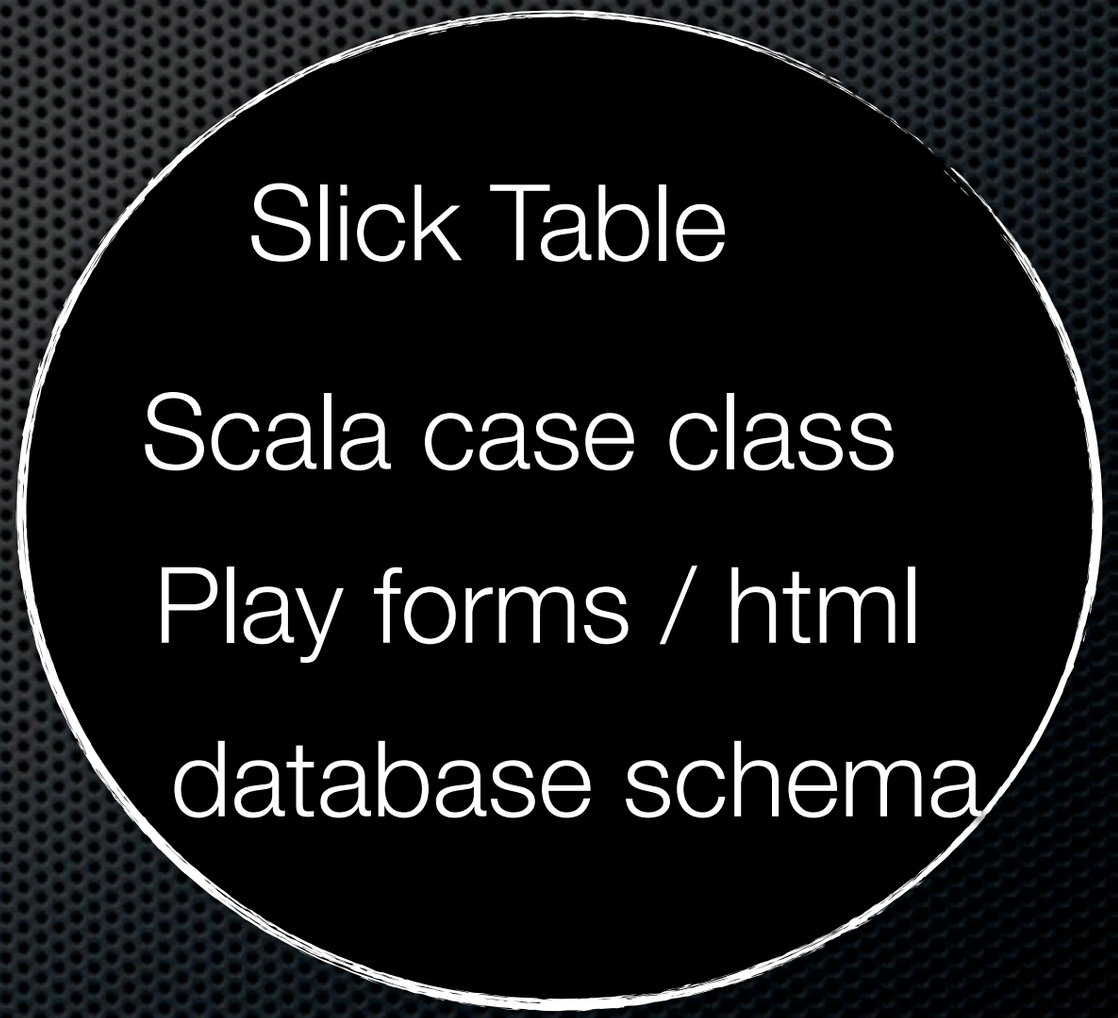
Problems of repetition

- Bad out of the box experience
- Implementation effort
- Maintenance effort (refactoring, etc.)
- Inconsistencies !
- Repeated bugs

Let’s refactor

Data model driven software





MDA

MDSE

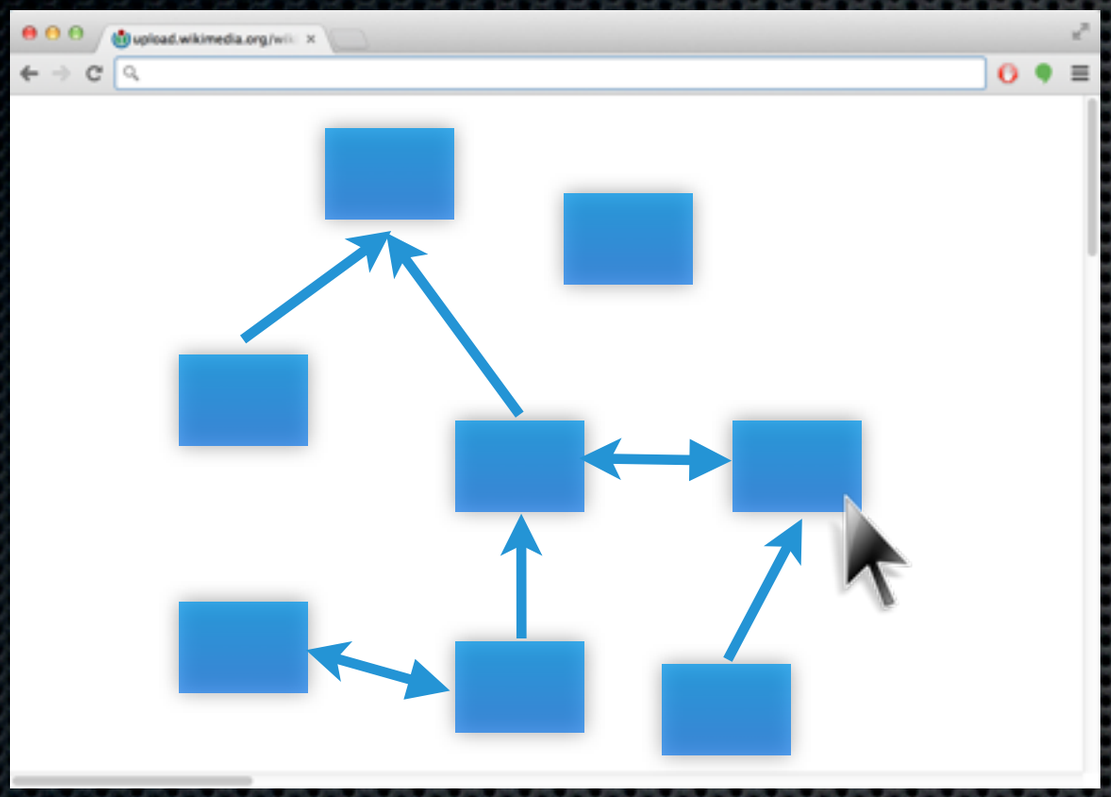
Wait... didn't model driven fail?

MDD

MDE

MDSD

Visual tool driven?



?

Slick Table

Scala case class

Play forms / html

database schema

Scala code driven?

hand-written

auto-generated

Scala case class
+
annotations

?

Slick Table

~~Scala case class~~

Play forms / html

database schema

needs migrations

Database schema driven?

managed by hand

auto-generated

database schema

?

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

- textual codegen (not Scala macros)
- generates all types for slick queries
- minimal customization may be required

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

Slick out-of-the-box codegen

database schema

jdbc meta data

```
Model( “Computers”,
  columns = Seq(
    Column(“ID”),
    ...
  )
)
```

Slick Model

Slick

code generator

Slick Table

Scala case class

~~Play forms / html~~

~~database schema~~

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

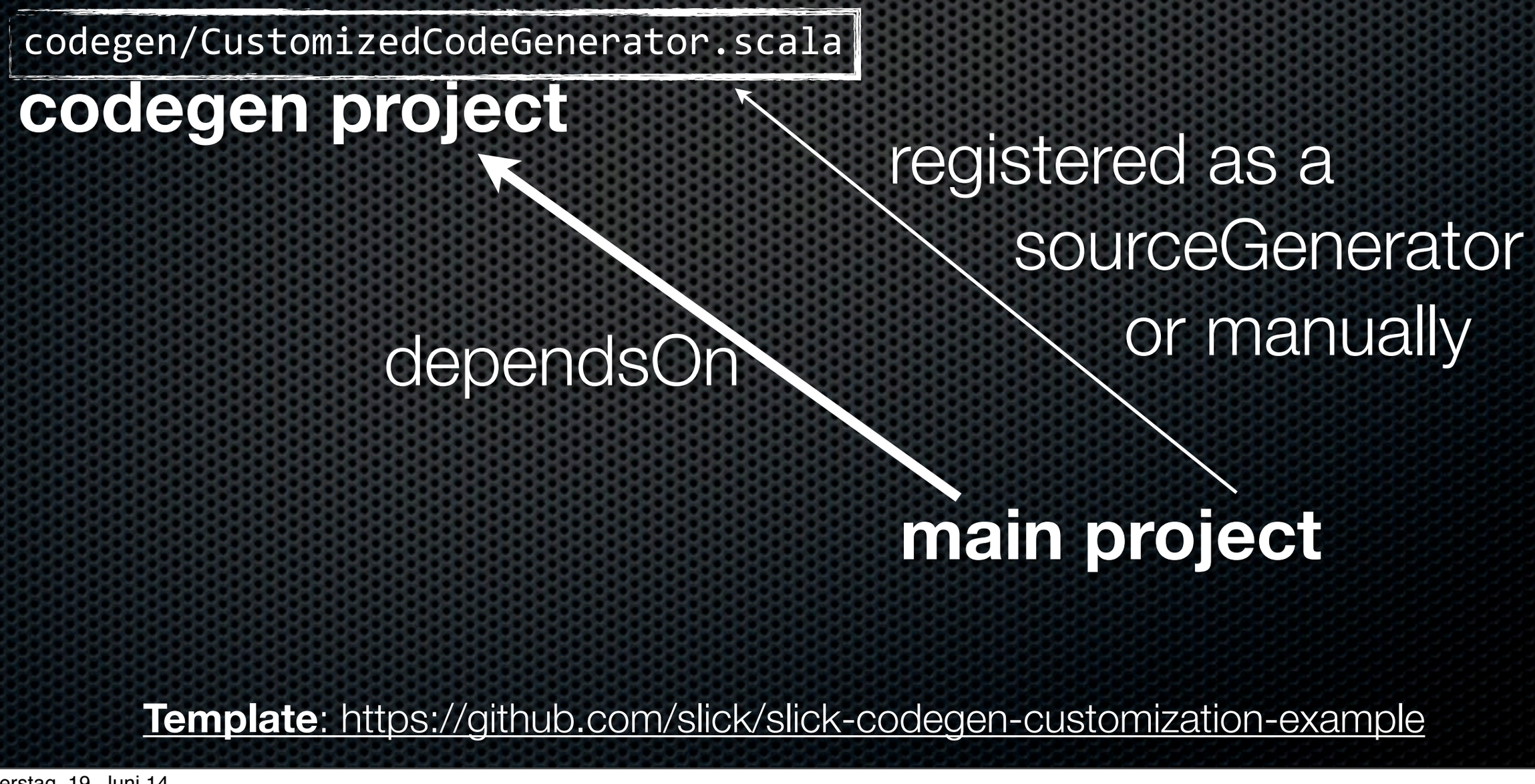
Slick customized code generation

Generate whatever

- play forms
- DAO
- gui
- ...

Slick customized codegen

sbt multi-project build



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

CustomizedSlickCodeGenerator.scala (from slick-codegen-customization-example)

```
// 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`

Let's generate a Play CRUD app all the way

demo time...

https://github.com/slick/play-slick-codegen

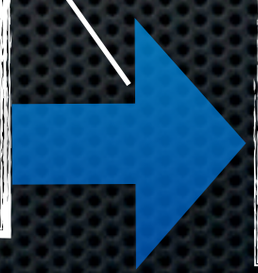
Generally interesting stuff

- Play html generic views
- unified edit/insert template and controller
- Slick CRUD
- pre-compiled Slick queries
- dynamic->static: Unified controllers serving all models
- multi-project, Slick codegen / Play sbt build

from dynamic to type-safe

- generic controller / play template are problematic
- use interfaces
- if no other way: codegen cases for all types
- .typed helper method (see Models.scala)

```
val modelsByName: Map[String,Model[_]] = ...
val m = modelsByName("computer")
val c = m.getById(1)
m.processEntity(c) // <- compile error
```



```
val modelsByName: Map[String,Model[_]] = ...
modelsByName("computer").typed{ m =>
  val c = m.getById(1)
  m.processEntity(c)
} // <- compiles fine
```

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

How did we do this?

- subclassed `scala.slick.model.codegen.`**`SourceCodeGenerator`**
- several override def code
- code templates via string interpolation

Look at generator

https://github.com/slick/play-slick-codegen

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**

Look at generated code

https://github.com/slick/play-slick-codegen

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

Sport195 Slick 1 -> Slick 2

- took 2-3 days

Slick 1

```
object Athletes
  extends Table[Athlete]("athlete"){
    def id = column(...)
    ...
    def * = ...

    def byId(id:Int)(implicit s: Session): Athlete
      = ...
}
```

Slick 2

```
class AthletesTable(tag: Tag)
  extends Table[Athlete]("athlete", tag){
    def id = column(...)
    ...
    def * = ...
}

object Athletes
  extends TableQuery(new AthletesTable(_)){
   def byId(id:Int)(implicit s: Session): Athlete
     = ...
}
```

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**

hand-written: **25% reduction**

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

S195 codegen architecture

managed with **migrations**

database schema

jdbc

meta data

automatically obtained from DBMS

```
Model( “computers”,
  columns = Seq(
    Column(“ID”),
    ...
  )
)
```

+

```
ExtraMetaData(
“computers”,
  customizeEntity=true
)
```

hand-written extra meta data

Customized Slick

codegen

generated code (never changed by hand)

Slick Table

Scala case class

~~SQL~~

...

interop via

interfaces / inheritance

hand-written code

```
case class $EntityName( ${columns...} )
```

code-template

(SOC, syntax highlighting, etc.)

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

1

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

- 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 ...

4

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

7

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

Other Slick 2 features

pre-compiled queries

```
object DAO[E]{
  /** caches compiled sql */
  private val byIdCompiled = Compiled{
    (id: Column[Int]) => query.filter(_.id === id)
  }

  def findById(id: Int)(implicit s: Session) = byIdCompiled(id).firstOption
  def update(id: Int, entity: E)(implicit s: Session) = byIdCompiled(id).update(entity)
  def delete(id: Int)(implicit s: Session)   = byIdCompiled(id).delete

  /** caches compiled sql */
  private lazy val insertInvoker = query.insertInvoker
  /** pre-compiled insert */
  def insert(entity: E)(implicit s: Session): Unit = insertInvoker.insert(entity)
}
```

because compiling Slick queries to SQL is slow

autoInc handling

Slick 1 autoInc projection

```
def * = ...
def autoInc = ... // <- same as * but excluding id
```

no more...
insert now ignores auto inc columns

Tuples and nested tuples

Slick 1

```
def * = id ~ name ~ ...
```

Slick 2

```
def * = (id, name, ...)
```

or nested

```
def * = ((id, name), ..., (createdAt, lastModifiedAt))
```

or nested an mapped

```
def basic = (id, name) <> ((BasicData.tupled _).apply, BasicData.unapply)
def metaData = (createdAt, lastModifiedAt) <> ((MetaData.tupled _).apply, MetaData.unapply)
def * = (basic, ..., metaData) <> ((Full.tupled _).apply, Full.unapply)
```

Various other changes

- query scheduling prototype
- lots of syntax/api/optimizer fixes and enhancements
- preparations for other backends
- threadLocalSession -> dynamicSession

http://slick.typesafe.com/doc/2.0.2/migration.html

Upcoming in Slick 2.1

- Scala 2.11 support
- insert-or-update
- precompiled take/drop queries
- more docs: from SQL to Slick, from ORM to Slick
- full outer join emulation
- improved codegen / model reverse engineering
- improved result set reading performance
- OSGi support

Codegen summary

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 to

- Slick community for bug reports and ideas!!
- Team@Sport195 for welcoming and pushing codegen
- Maxim@typesafe for JavaScript validations

Thank you!

We are hiring at Sport195. Talk to me.

christopher.vogt@sport195.com

twitter: @cvogt

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