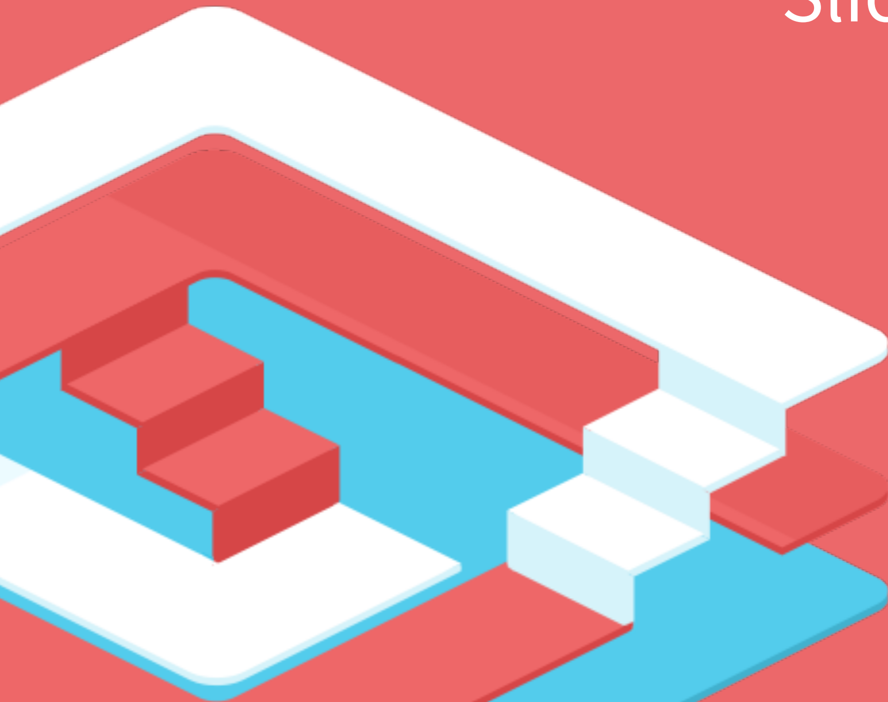
Patterns for Slick database applications

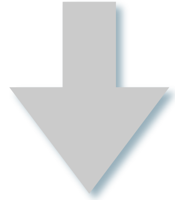
Jan Christopher Vogt, EPFL
Slick Team



#scalax

Recap: What is Slick?

```
(for ( c <- coffees;
       if c.sales > 999
) yield c.name).run
```



```
select "COF_NAME"
from   "COFFEES"
where  "SALES" > 999
```

Agenda

- Query composition and re-use
- Getting rid of boiler plate in Slick 2.0
  - outer joins / auto joins
  - auto increment
  - code generation
- Dynamic Slick queries

Query composition and re-use

For-expression desugaring in Scala

```
for ( c <- coffees;
      if c.sales > 999
) yield c.name
```

```
          coffees
            .withFilter(_.sales > 999)
            .map(_.name)
```

Types in Slick

```
class Coffees(tag: Tag) extends Table[C](tag,"COFFEES”) {
  def * = (name, supId, price, sales, total) <> ...
  val name = column[String]("COF_NAME", O.PrimaryKey)
  val supId = column[Int]("SUP_ID")
  val price = column[BigDecimal]("PRICE")
  val sales = column[Int]("SALES")
  val total = column[Int]("TOTAL")
}
lazy val coffees = TableQuery[Coffees]  <: Query[Coffees,C]

coffees.map(c => c.name)
(coffees: TableQuery[Coffees,_]).map(
                    (c:Coffees) => (c.name: Column[String])
): Query[Column[String],String]
```

Table extensions

```
class Coffees(tag: Tag) extends Table[C](tag,"COFFEES”) {
  …
  val price = column[BigDecimal]("PRICE")
  val sales = column[Int]("SALES")
  def revenue = price.asColumnOf[Double] *
                  sales.asColumnOf[Double]
}
```

```
coffees.map(c => c.revenue)
```

Query extensions

```
implicit class QueryExtensions[T,E]
                 ( val q: Query[T,E] ){
  def page(no: Int, pageSize: Int = 10)
      : Query[T,E]
      = q.drop( (no-1)*pageSize ).take(pageSize)
}

suppliers.page(5)
coffees.sortBy(_.name).page(5)
```

Query extensions by Table

```
implicit class CoffeesExtensions
                 ( val q: Query[Coffees,C] ){
  def byName( name: Column[String] )
      : Query[Coffees,C]
      = q.filter(_.name === name).sortBy(_.name)
}

coffees.byName("ColumbianDecaf").page(5)
```

Query extensions for joins

```
implicit class CoffeesExtensions2( val q: Query[Coffees,C] ){
  def withSuppliers
      (s: Query[Suppliers,S] = Tables.suppliers)
      : Query[(Coffees,Suppliers),(C,S)]
      = q.join(s).on(_.supId===_.id)
  def suppliers
      (s: Query[Suppliers, S] = Tables.suppliers)
      : Query[Suppliers, S]
      = q.withSuppliers(s).map(_._2)
}

coffees.withSuppliers() : Query[(Coffees,Suppliers),(C,S)
coffees.withSuppliers( suppliers.filter(_.city === "Henderson") )

// buyable coffees
coffeeShops.coffees().suppliers().withCoffees()
```

Query libraries by Interface

```
trait HasId{ _:Table[_] => def id: Column[Int] }
class Coffees(tag:Tag) extends Table[C](…)
  with HasId{ def id = column[Int](“ID”); … }
class Suppliers(tag:Tag) extends Table[C](…)
  with HasId{
def id = column[Int](“ID”) }

implicit class QueryExtensions[T <: HasId,C]
                              ( val q: Query[T,E] ){
  def byId(id: Int): Query[T,C]
      = q.byId( id )
  …
}

coffees.byId( 123 )
```

Query extensions by Interface

```
trait HasSuppliers{ def supId: Column[Int] }
class Coffees(…)      extends Table... with HasSuppliers {…}
class CofInventory(…) extends Table... with HasSuppliers {…}
implicit class HasSuppliersExtensions[T <: HasSupplier,E]
                      ( val q: Query[T,E] ){
  def bySupId(id: Column[Int]): Query[T,E]
                              = q.filter( _.supId === id )
  def withSuppliers
      (s: Query[Suppliers,S] = Tables.suppliers)
      : Query[(T,Suppliers),(E,S)]
      = q.join(s).on(_.supId===_.id)
  def suppliers ...
}

// available quantities of coffees
cofInventory.withSuppliers()
  .map{ case (i,s) =>
      i.quantity.asColumnOf[String] ++ " of " ++ i.cofName ++ " at " ++ s.name
  }
```

Query extensions summary

- **Mindshift required!**
  Think code, not monolithic query strings.
- **Stay completely lazy!**
  Keep Query[...]s as long as you can.
- **Re-use!**
  Write query functions or extensions methods for shorter, better and DRY code.

Getting rid of boilerplate

AUTO JOINS

Auto joins

```
implicit class QueryExtensions2[T,E]
               ( val q: Query[T,E] ){
  def autoJoin[T2,E2]
        ( q2:Query[T2,E2] )
        ( implicit condition: (T,T2) => Column[Boolean] )
        : Query[(T,T2),(E,E2)]
        = q.join(q2).on(condition)
}

implicit def joinCondition1
    = (c:Coffees,s:Suppliers) => c.supId === s.id

coffees.autoJoin( suppliers ) : Query[(Coffees,Suppliers),(C,S)]

coffees.autoJoin( suppliers ).map(_._2).autoJoin(cofInventory)
```

AUTO INCREMENTING INSERTS

Auto incrementing inserts

```
val supplier
 = Supplier( 0, "Arabian Coffees Inc.", ... )

// now ignores auto-increment column
suppliers.insert( supplier )

// includes auto-increment column
suppliers.forceInsert( supplier )
```

CODE GENERATION

Code generator for Slick code

```
// runner for default config

import scala.slick.meta.codegen.SourceCodeGenerator
SourceCodeGenerator.main(
  Array(
      "scala.slick.driver.H2Driver",
      "org.h2.Driver",
      "jdbc:h2:mem:test",
      "src/main/scala/", // base src folder
      "demo" // package
    )
)
```

Generated code

```
package demo
object Tables extends {
  val profile = scala.slick.driver.H2Driver
} with Tables

trait Tables {
  val profile: scala.slick.driver.JdbcProfile
  import profile.simple._
  case class CoffeeRow(name: String, supId: Int, ...)
  implicit def GetCoffees
        = GetResult{r => CoffeeRow.tupled((r.<<, ... )) }
  class Coffees(tag: Tag) extends Table[CoffeeRow](…){…}
  ...
```

OUTER JOINS

Outer join limitation in Slick

```scala
suppliers.leftJoin(coffees)
  .on(_.id === _.supId)
  .run // SlickException: Read NULL value ...
```

id	name	name	supId
1	Superior Coffee	NULL	NULL
2	Acme, Inc.	Colombian	2
2	Acme, Inc.	French_Roast	2

LEFT JOIN

id	name
1	Superior Coffee
2	Acme, Inc.

name	supId
Colombian	2
French_Roast	2

Outer join pattern

```
suppliers.leftJoin(coffees)
  .on(_.id === _.supId)
  .map{ case(s,c) => (s,(c.name.?,c.supId.?,…)) }
  .run
  .map{ case (s,c) =>
        (s,c._1.map(_ => Coffee(c._1.get,c._2.get,…))) }

// Generated outer join helper
suppliers.leftJoin(coffees)
  .on(_.id === _.supId)
  .map{ case(s,c) => (s,c.?) }
  .run
```

CUSTOMIZABLE CODE GENERATION

Using code generator as a library

```
val metaModel = db.withSession{ implicit session =>
  profile.metaModel // e.g. H2Driver.metaModel
}
import  scala.slick.meta.codegen.SourceCodeGenerator
val codegen = new SourceCodeGenerator(metaModel){
  // <- customize here
}
codegen.writeToFile(
  profile = "scala.slick.driver.H2Driver”,
  folder = "src/main/scala/",
  pkg = "demo",
  container = "Tables",
  fileName="Tables.scala" )
```

Adjust name mapping

```
import scala.slick.util.StringExtensions._
val codegen =
  new SourceCodeGenerator(metaModel){
    override def tableName
      = _.toLowerCase.toCamelCase
    override def entityName
       = tableName(_).dropRight(1)
  }
```

Generate auto-join conditions 1

```
class CustomizedCodeGenerator(metaModel: Model)
  extends SourceCodeGenerator(metaModel){
   override def code = {
    super.code + "\n\n" + s"""
/** implicit join conditions for auto joins */
object AutoJoins{
  ${indent(joins.mkString("\n"))}
}
    """.trim()
  }
  …
```

Generate auto-join conditions 2

```
 …
 val joins = tables.flatMap( _.foreignKeys.map{ foreignKey =>
   import foreignKey._
   val fkt = referencingTable.tableClassName
   val pkt = referencedTable.tableClassName
   val columns = referencingColumns.map(_.name) zip
                             referencedColumns.map(_.name)
   s"implicit def autojoin${name.capitalize} "+
    " = (left:${fkt},right:${pkt}) => " +
       columns.map{
         case (lcol,rcol) =>
           "left."+lcol + " === " + "right."+rcol
       }.mkString(" && ")
}
```

Other uses of Slick code generation

- Glue code (Play, etc.)
- n-n join code
- Migrating databases (warning: types change) (generate from MySQL, create in Postgres)
- Generate repetitive regarding data model (aka model driven software engineering)
- Generate DDL for external model

Use code generation wisely

- Don’t loose language-level abstraction
- Add your generator and data model to version control
- Complete but new and therefor experimental in Slick

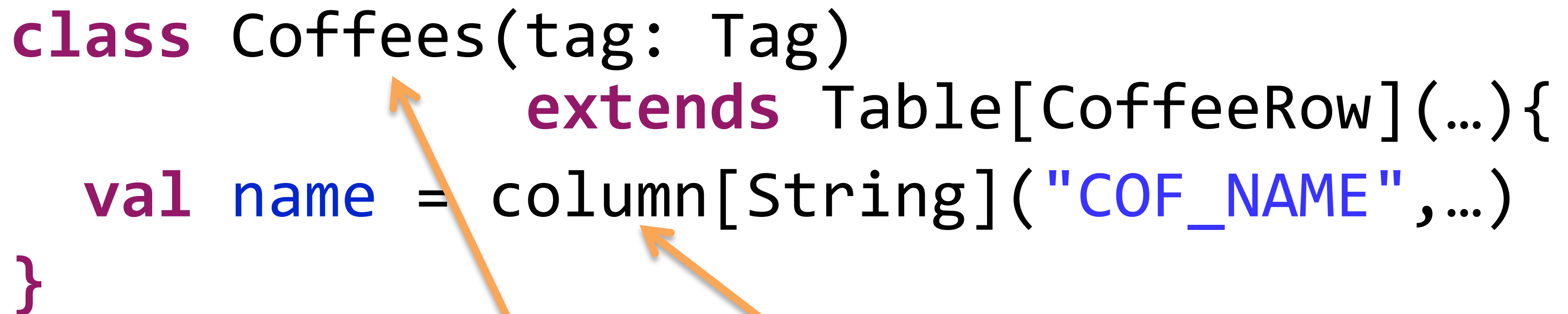
Dynamic Queries

Common use case for web apps

Dynamically decide

- displayed columns
- filter conditions
- sort columns / order

Dynamic column




```
coffees.map(c => c.name)
coffees.map(c =>
  c.column[String]("COF_NAME")
)
```

Be careful about security!

Example: sortDynamic

```
suppliers.sortDynamic("street.desc,city.desc")
```

sortDynamic 1

```
implicit class QueryExtensions3[E,T<: Table[E]]
                    ( val query: Query[T,E] ){
  def sortDynamic(sortString: String) : Query[T,E] = {
    // split string into useful pieces
    val sortKeys = sortString.split(',').toList.map(
                      _.split('.').map(_.toUpperCase).toList )
    sortDynamicImpl(sortKeys)
  }
  private def sortDynamicImpl(sortKeys: List[Seq[String]]) = ???
}
```

```
suppliers.sortDynamic("street.desc,city.desc")
```

sortDynamic 2

```
  ...
  private def sortDynamicImpl(sortKeys: List[Seq[String]]) : Query[T,E] = {
    sortKeys match {
      case key :: tail =>
        sortDynamicImpl( tail ).sortBy( table =>
          key match {
            case name :: Nil =>             table.column[String](name).asc
            case name :: "ASC" :: Nil =>  table.column[String](name).asc
            case name :: "DESC" :: Nil => table.column[String](name).desc
            case o => throw new Exception("invalid sorting key: "+o)
          }
        )
      case Nil => query
    }
  }
}

suppliers.sortDynamic("street.desc,city.desc")
```

Summary

- Query composition and re-use
- Getting rid of boiler plate in Slick 2.0
  - outer joins / auto joins
  - auto increment
  - code generation
- Dynamic Slick queries

Thank you

#scalax

slick.typesafe.com

@cvogt

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

https://github.com/cvogt/slick-presentation/tree/scala-exchange-2013

filterDynamic

```
coffees.filterDynamic("COF NAME like Decaf")
```