Introduction to Slick 2.1 and 2.2

Stefan Zeiger
Object-Relational Mapping
Object

Relational

Impedance Mismatch
## Concepts

<table>
<thead>
<tr>
<th>Object-Oriented</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>No Identity</td>
</tr>
<tr>
<td>State</td>
<td>Transactional State</td>
</tr>
<tr>
<td>Behavior</td>
<td>No Behavior</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>No Encapsulation</td>
</tr>
</tbody>
</table>
Laziness

Colombian
French_Roast
Espresso
Colombian_Decaf
French_Roast_Decaf

Espresso
Price: 9.99
Supplier: The High Ground

select NAME
from COFFEEES

select c.NAME, c.PRICE, s.NAME
from COFFEEES c
join SUPPLIERS s
on c.SUP_ID = s.SUP_ID
where c.NAME = ?
Laziness

<table>
<thead>
<tr>
<th>Coffee</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colombian</td>
<td>7.99</td>
</tr>
<tr>
<td>French_Roast</td>
<td>8.99</td>
</tr>
<tr>
<td>Espresso</td>
<td>9.99</td>
</tr>
<tr>
<td>Colombian_Decaf</td>
<td>8.99</td>
</tr>
<tr>
<td>French_Roast_Decaf</td>
<td>9.99</td>
</tr>
</tbody>
</table>

```scala
def getAllCoffees(): Seq[Coffee] = ...
def printLinks(s: Seq[Coffee]) {
  for(c <- s) println(c.name + c.price)
}
```
Laziness

def printDetails(c: Coffee) {
    println(c.name)
    println("Price: "+c.price)
    println("Supplier: "+c.supplier.name)
}

Colombian
French_Roast
Espresso
Colombian_Decaf
French_Roast_Decaf

Espresso
Price: 9.99
Supplier: The High Ground
# Level of Abstraction

<table>
<thead>
<tr>
<th>Data Organization</th>
<th>Object Oriented</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

Slick 2.1 and 2.2
Functional Relational Mapping
Relational Model

- Relation
- Attribute
- Tuple
- Relation Value
- Relation Variable

<table>
<thead>
<tr>
<th>COFFEE</th>
<th>NAME</th>
<th>PRICE</th>
<th>SUP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colombian</td>
<td>7.99</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>French_Roast</td>
<td>8.99</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Espresso</td>
<td>9.99</td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>
Relational Model

- **Relation**
- **Attribute**
- **Tuple**
- **Relation Value**
- **Relation Variable**

<table>
<thead>
<tr>
<th>COFFEES</th>
<th>NAME</th>
<th>PRICE</th>
<th>SUP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Colombian</td>
<td>7.99</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>French_</td>
<td>8.99</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Roast</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Espresso</td>
<td>9.99</td>
<td>150</td>
</tr>
</tbody>
</table>
Relational Model

- Relation
- **Attribute**
- Tuple
- Relation Value
- Relation Variable

<table>
<thead>
<tr>
<th>NAME</th>
<th>PRICE</th>
<th>SUP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colombian</td>
<td>7.99</td>
<td>101</td>
</tr>
<tr>
<td>French_Roast</td>
<td>8.99</td>
<td>49</td>
</tr>
<tr>
<td>Espresso</td>
<td>9.99</td>
<td>150</td>
</tr>
</tbody>
</table>
Relational Model

- Relation
- Attribute
- **Tuple**
- Relation Value
- Relation Variable

<table>
<thead>
<tr>
<th>COFFEEES</th>
<th>NAME</th>
<th>PRICE</th>
<th>SUP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Colombian</td>
<td>7.99</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>French_Roast</td>
<td>8.99</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Espresso</td>
<td>9.99</td>
<td>150</td>
</tr>
</tbody>
</table>
Relational Model

- Relation
- Attribute
- Tuple
- **Relation Value**
- Relation Variable

<table>
<thead>
<tr>
<th>COFFEE</th>
<th>NAME</th>
<th>PRICE</th>
<th>SUP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>COFFEE1</td>
<td>Colombian</td>
<td>7.99</td>
<td>101</td>
</tr>
<tr>
<td>COFFEE2</td>
<td>French_Roast</td>
<td>8.99</td>
<td>49</td>
</tr>
<tr>
<td>COFFEE3</td>
<td>Espresso</td>
<td>9.99</td>
<td>150</td>
</tr>
</tbody>
</table>
Relational Model

- Relation
- Attribute
- Tuple
- Relation Value
- Relation Variable

<table>
<thead>
<tr>
<th>COFFEEES</th>
<th>NAME : String</th>
<th>PRICE : Double</th>
<th>SUP_ID : Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colombian</td>
<td>7.99</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>French_Roast</td>
<td>8.99</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Espresso</td>
<td>9.99</td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>
Mapped to Scala

- Relation
- Attribute
- Tuple
- Relation Value
- Relation Variable

```scala
case class Coffee(
  name: String,
  supplierId: Int,
  price: Double
)

val coffees = Set(
  Coffee("Colombian", 101, 7.99),
  Coffee("French_Roast", 49, 8.99),
  Coffee("Espresso", 150, 9.99)
)
```
Mapped to Scala

- **Relation**
- **Attribute**
- **Tuple**
- **Relation Value**
- **Relation Variable**

```scala
case class Coffee(
  name: String,
  supplierId: Int,
  price: Double
)

val coffees = Set(
  Coffee("Colombian", 101, 7.99),
  Coffee("French_Roast", 49, 8.99),
  Coffee("Espresso", 150, 9.99)
)
```
Mapped to Scala

- Relation
- **Attribute**
- Tuple
- Relation Value
- Relation Variable

```scala
case class Coffee(
  name: String,
  supplierId: Int,
  price: Double
)

val coffees = Set(
  Coffee("Colombian", 101, 7.99),
  Coffee("French_Roast", 49, 8.99),
  Coffee("Espresso", 150, 9.99)
)
```
Mapped to Scala

- Relation
- Attribute
- **Tuple**
- Relation Value
- Relation Variable

```scala
case class Coffee(
  name: String,
  supplierId: Int,
  price: Double
)

val coffees = Set(
  Coffee("Colombian", 101, 7.99),
  Coffee("French_Roast", 49, 8.99),
  Coffee("Espresso", 150, 9.99)
)
```
Mapped to Scala

- Relation
- Attribute
- Tuple
- **Relation Value**
- Relation Variable

```scala
case class Coffee(
  name: String,
  supplierId: Int,
  price: Double
)

val coffees = Set(
  Coffee("Colombian", 101, 7.99),
  Coffee("French_Roast", 49, 8.99),
  Coffee("Espresso", 150, 9.99)
)
```
Mapped to Scala

- Relation
- Attribute
- Tuple
- Relation Value
- **Relation Variable**

```scala
case class Coffee(
  name: String,
  supplierId: Int,
  price: Double
)

val coffees = Set(
  Coffee("Colombian", 101, 7.99),
  Coffee("French_Roast", 49, 8.99),
  Coffee("Espresso", 150, 9.99)
)
```
Write Database Code in Scala

```scala
for { p <- persons } yield p.name
```

```sql
select p.NAME from PERSON p
```
(for {
  p <- persons.filter(_.age < 20) ++
      persons.filter(_.age >= 50)
  if p.name.startsWith("A")
} yield p).groupBy(_.age).map { case (age, ps) =>
  (age, ps.length)
}

select x2.x3, count(1) from (  
  select * from (  
    select x4."NAME" as x5, x4."AGE" as x3  
    from "PERSON" x4 where x4."AGE" < 20  
    union all select x6."NAME" as x5, x6."AGE" as x3  
    from "PERSON" x6 where x6."AGE" >= 50  
  ) x7 where x7.x5 like 'A%' escape '^'
) x2
  group by x2.x3
Functional Relational Mapping

- Embraces the relational model
- Prevents impedance mismatch

```scala
class Suppliers ... extends Table[(Int, String, String)](... "SUPPLIERS")

sup.filter(_.id < 2) ++ sup.filter(_.id > 5)
```
Functional Relational Mapping

- Embraces the relational model
- Prevents impedance mismatch
- Composable Queries

```scala
def f(id1: Int, id2: Int) = sup.filter(_.id < id1) ++ sup.filter(_.id > id2)
val q = f(2, 5).map(_.name)
```
Functional Relational Mapping

- Embraces the relational model
- Prevents impedance mismatch
- Composable Queries
- Explicit control over statement execution

```scala
val result = q.run
```
Functional

Relational
Functional

Relational
Slick
Slick

Scala Language Integrated Connection Kit

- Database query and access library for Scala
- Successor of ScalaQuery
- Developed at Typesafe and EPFL
- Open Source
Supported Databases

• **Slick**
  • PostgreSQL
  • MySQL
  • H2
  • Hsqldb
  • Derby / JavaDB
  • SQLite
  • Access

• **Slick Extensions**
  • Oracle
  • DB2
  • SQL Server

Closed source, with commercial support by Typesafe
Getting Started with Activator

http://typesafe.com/activator
Schema Definition
class Suppliers(tag: Tag) extends Table[(Int, String, String)](tag, "SUPPLIERS") {
    def id = column[Int]("SUP_ID",
        0.PrimaryKey, 0.AutoInc)
    def name = column[String]("NAME")
    def city = column[String]("CITY")
    def * = (id, name, city)
}

val suppliers = TableQuery[Suppliers]
case class Supplier(id: Int, name: String, city: String)

class Suppliers(tag: Tag) extends Table[Supplier](tag, "SUPPLIERS") {
  def id = column[Int]("SUP_ID",
                        0.PrimaryKey, 0.AutoInc)
  def name = column[String]("NAME")
  def city = column[String]("CITY")
  def * = (id, name, city) <>
           (Supplier.tupled, Supplier.unapply)
}

val suppliers = TableQuery[Suppliers]
class SupplierId(val value: Int) extends AnyVal

case class Supplier(id: SupplierId, name: String, city: String)

implicit val supplierIdType = MappedColumnType.base[
  SupplierId, Int](_.value, new SupplierId(_))

class Suppliers(tag: Tag) extends Table[Supplier](tag, "SUPPLIERS") {
  def id = column[SupplierId]("SUP_ID", ...)
  ...
}

Slick 2.1 and 2.2
Custom Column Types

class SupplierId(val value: Int) extends MappedTo[Int]

case class Supplier(id: SupplierId, name: String, city: String)

class Suppliers(tag: Tag) extends Table[Supplier](tag, "SUPPLIERS") {
  def id = column[SupplierId]("SUP_ID", ...)
  ... 
}

class Coffees(tag: Tag) extends Table[
    (String, SupplierId, Double)](tag, "COFFEES") {
  def name = column[String]("NAME", O.PrimaryKey)
  def supID = column[SupplierId]("SUP_ID")
  def price = column[Double]("PRICE")
  def * = (name, supID, price)
  def supplier =
    foreignKey("SUP_FK", supID, suppliers)(_.id)
}

val coffees = TableQuery[Coffees]
Code Generator

- Reverse-engineer an existing database schema
- Create table definitions and case classes
- Customizable
- Easy to embed in sbt build
Data Manipulation
import scala.slick.driver.H2Driver.simple._

val db = Database.forURL("jdbc:h2:mem:test1",
  driver = "org.h2.Driver")

db.withSession { implicit session =>
  // Use the session:
  val result = myQuery.run
}
Creating Tables and Inserting Data

```scala
val suppliers = new ArrayBuffer[Supplier]
val coffees = new ArrayBuffer[(String, SupplierId, Double)]

suppliers += Supplier(si1, "Acme, Inc.", "Groundsville")
suppliers += Supplier(si2, "Superior Coffee", "Mendocino")
suppliers += Supplier(si3, "The High Ground", "Meadows")

coffees += Seq(
  ("Colombian", si1, 7.99),
  ("French_Roast", si2, 8.99),
  ("Espresso", si3, 9.99),
  ("Colombian_Decaf", si1, 8.99),
  ("French_Roast_Decaf", si2, 9.99)
)
```
val ins = suppliers.map(s => (s.name, s.city))
  returning suppliers.map(_.id)

val si1 = ins += ("Acme, Inc.", "Groundsville")
val si2 = ins += ("Superior Coffee", "Mendocino")
val si3 = ins += ("The High Ground", "Meadows")

coffees ++= Seq(
  ("Colombian", si1, 7.99),
  ("French_Roast", si2, 8.99),
  ("Espresso", si3, 9.99),
  ("Colombian_Decaf", si1, 8.99),
  ("French_Roast_Decaf", si2, 9.99)
)
Querying
val q = for {
  c <- coffees
  if c.price < 9.0
  s <- c.supplier
} yield (c.name, s.name)

val result = q.run(session)
Nullable Columns

- We don't like `null` in Scala!
- ...but the database likes them

```scala
class Coffees(tag: Tag) extends Table[
  (String, Option[SupplierId], Double)](tag, "COFFEES") {
  def name = column[String]("NAME", O.PrimaryKey)
  def supID = column[Option[SupplierId]]("SUP_ID")
  def price = column[Double]("PRICE")
  def * = (name, supID, price)
  def supplier =
    foreignKey("SUP_FK", supID.?, suppliers)(_ . id)
}
```
Nullable Columns

- We don't like `null` in Scala!
- ...but the database likes them

```scala
coffees.map(_.price).max : Column[Option[Double]]
```
Plain SQL
def personsMatching(pattern: String)(conn: Connection) = {
  val st = conn.prepareStatement(
    "select id, name from person where name like ?"
  )
  try {
    st.setString(1, pattern)
    val rs = st.executeQuery()
    try {
      val b = new ListBuffer[(Int, String)]
      while (rs.next)
        b.append((rs.getInt(1), rs.getString(2)))
      b.toList
    }
    finally rs.close()
  }
  finally st.close()
}
def personsMatching(pattern: String)(implicit s: Session) =
  sql"select id, name from person where name like $pattern" 
  .as[(Int, String)].list
Slick 2.1
At 7th ScalaCamp @StefanZeiger will talk about new features of Slick 2.1. Join us at scalacamp.pl
Documentation

• New user manual chapters
  • Coming from ORM to Slick
  • Coming from SQL to Slick

• Activator Templates
  • Replacing *slick-examples* and other sample projects
  • Per major version

• More comprehensive API docs
Outer Join Emulation

- Full Outer Join ➔ Left Outer Join + Union All
- Right Outer Join ➔ Left Outer Join
- Left Outer Join ➔ Inner Join + Union All

```
select s2.s21, s3.s22
from (select s23."id" as s21 from "t" s23) s2
  full outer join (select s24."id" as s22 from "t" s24) s3
  on s2.s21 = s3.s22

```
Outer Join Emulation

```sql
select s21.s41, s21.s42
from (  
    select s27.s43 as s41, s27.s44 as s42 from (  
        select s2.s45 as s43, s3.s46 as s44
        from (select s53."id" as s45 from "t" s53) s2
        inner join (select s54."id" as s46 from "t" s54) s3
        on s2.s45 = s3.s46
    union all
    select s55."id" as s43, null as s44
    from "t" s55
    where not exists(select s57."id" from "t" s57 where s55."id" = s57."id")
    ) s27
    union all
    select null as s41, s59."id" as s42
    from "t" s59
    where not exists(select s61."id" from "t" s61 where s61."id" = s59."id")
) s21
```
Compiled Pagination Operators

- Overloaded for `ConstColumn`
- Values known at query execution time

```scala
Compiled { (d: ConstColumn[Long], t: ConstColumn[Long]) =>
  ids.sortBy(_.id).drop(d).take(t)
}
```

- `CompiledStatement`
  ```scala
  select s6."id" from (select s13."id" as "id" from "ids_compiled" s13 order by s13."id" limit ? offset ?) s6
  ```
Compiled Pagination Operators

- Overloaded for `ConstColumn`
- Values known at query execution time

- ParameterSwitch

  - `[<function1>](...) == 0]: CompiledStatement
    select s6."id" from (select s13."id" as "id" from "ids_compiled" s13 where 1=0 order by s13."id") s6

  - default: CompiledStatement
    select s6."id" from (select s13."id" as "id" from "ids_compiled" s13 order by s13."id" offset ? row fetch next ? row only) s6
Fast Path Result Converters

- Remove Boxing and Allocation Overhead

```scala
case class A(var a: Int, var b: Int, var c: Int)

class ARow ...
  extends Table ...
  {
    ...
    def proj = (i, io.get, io.getOrElse(-1))
  }

// Standard converters
val q1 = as.map(a => a.proj <> (A.tupled, A.unapply))

q1.foreach { a => ... }
```
Fast Path Result Converters

- Remove Boxing and Allocation Overhead

```scala
// Fast path
val q2 = as.map(a => a.proj <*> (A.tupled, A.unapply))
  fastPath(new FastPath(_) {
    val (a, b, c) =
      (next[Int], next[Int], next[Int])
    override def read(r: Reader) = new A(
      a.read(r), b.read(r), c.read(r))
  })
```
Fast Path Result Converters

- Remove Boxing and Allocation Overhead

```scala
// Allocation-free fast path
val sharedA = new A(0, 0, 0)
val q3 = as.map(a => a.proj <> (A.tupled, A.unapply))
  fastPath(new FastPath(_){
    val (a, b, c) =
      (next[Int], next[Int], next[Int])
    override def read(r: Reader) = {
      sharedA.a = a.read(r)
      sharedA.b = b.read(r)
      sharedA.c = c.read(r)
      sharedA
    }
  })
```
Insert or Update

- Longest standing feature request (issue #6) with most upvotes
- Uses native database support (UPSERT, MERGE) where possible
- Based on primary key comparison

```
// Slick 2.1 and 2.2

// Insert or Update

val ts = (1, "a")
val ts = ts insertOrUpdate (2, "b")
```
CaseClassShape

• Easily support monomorphic record types

case class B(a: Int, b: String)
case class LiftedB(a: Column[Int], b: Column[String])
implicit object BShape extends CaseClassShape(LiftedB.tupled, B.tupled)

class BRow(tag: Tag) extends Table[B](tag, "shape_b") {
  def id = column[Int]("id", O.PrimaryKey)
  def s = column[String]("s")
  def * = LiftedB(id, s)
}
val bs = TableQuery[BRow]

bs += B(1, "a")

val q3 = for {
  LiftedB(id, s) <- bs if id == 1
} yield LiftedB(id, s ++ s)
Collection Type Constructors

• Type constructor propagated through Query

• Used with Executor API (.run)

```scala
val xs = TableQuery[X]  // Query[X, ..., Seq]
x.s.run  // Seq[...]

val q = xs.to[Set]  // Query[X, ..., Set]
q.take(10).run  // Set[...]
```

• Long-term goal: Remove old Invoker API (.list, .first, .iterator, ...)

val xs = TableQuery[X]  // Query[X, ..., Seq]
x.s.run  // Seq[...]

val q = xs.to[Set]  // Query[X, ..., Set]
q.take(10).run  // Set[...]

• Long-term goal: Remove old Invoker API (.list, .first, .iterator, ...)

val xs = TableQuery[X]  // Query[X, ..., Seq]
x.s.run  // Seq[...]

val q = xs.to[Set]  // Query[X, ..., Set]
q.take(10).run  // Set[...]

Other

- Typesafe Config (*Database.forConfig*)
- OSGi Support
- More String methods in queries (e.g. *substring*)
- Pre-Compiled Inserts
- More flexible TestKit
- More stable and flexible schema reverse engineering and code generator
Slick 2.2
Reactive Slick

- Asynchronous execution
  - *Futures* for scalar / fully materialized results
  - *Reactive Streams* for streaming results

- Revamped API for synchronous execution

- Integrated connection pool support
  - Asynchronous execution on top of JDBC
  - Based on connection pool and automatically configured thread pool

- New API for composing database actions (*I/O monad*)
  - Prevent leaking / expired *Session* objects
  - Blocking-agnostic composition of actions
Nested and Multi-Column Options

YO DAWG

I heard you like monads so
I put an Option in your Query so
you can flatMap while you flatMap
Nested and Multi-Column Options

- Lift to Option: **Rep.Some**
- Generate lifted *None* value: **Rep.None**
- Extension methods: **fold, flatMap, map, flatten, filter, getOrElse, isEmpty, isDefined, nonEmpty**
- Not for column definitions
- No **get** method for non-primitive Options
Non-primitive Options are the correct representation of outer join results

case class Data(a: Int, b: String)
class Row(name: String)(tag: Tag)  
  extends Table[Data](tag, name) {
    def a = column[Int]("a")
    def b = column[String]("b")
    def * = (a, b) <> (Data.tupled, Data.unapply)
  }
val xs = TableQuery(new Row("xs")(__))
val ys = TableQuery(new Row("ys")(__))

val q1 = xs join ys on (_.b === _.b)
q1.run // Seq[(Data, Data)]
Outer Joins

- Non-primitive Options are the correct representation of outer join results

```scala
val q2 = xs leftJoin ys on (_.b === _.b)

q2.run // Seq[(Data, Data)]
```
Outer Joins

- Non-primitive Options are the correct representation of outer join results

```scala
val q2 =(xs leftJoin ys on (_.b === _.b) map {
  case (x, y) => (x, (y.a, y.b)).shaped.<>[
    case (Some(a), Some(b)) => Some(Data(a, b))
    case _ => None
  ],
  case _ => ???
})

q2.run  // Seq[(Data, Option[Data])]```

Slick 2.1 and 2.2
Outer Joins

- Non-primitive Options are the correct representation of outer join results

```scala
val q2 = xs leftJoin ys on (_.b === _.b) map {
  case (x, y) => (x, (y.a?.?, y.b?.?).shaped.<> [Option[Data]] ({
    case (Some(a), Some(b)) => Some(Data(a, b))
    case _ => None
  }), {
    case _ => ???
  })
}

q2.run // Seq[(Data, Option[Data])]`
```
Outer Joins

- Non-primitive Options are the correct representation of outer join results

```scala
val q3 = xs joinLeft ys on (_.b === _.b)
q3.run // Seq[(Data, Option[Data])]
```
Statically Checked Plain SQL

- Let the database server type-check Plain SQL queries when compiling your Scala code
- Automatically infer return types

```scala
def personsMatching(pattern: String)(implicit s: Session) =
  sql"select id, name from person where name like $pattern"
  .as[(Int, String)].list
```
Statically Checked Plain SQL

- Let the database server type-check Plain SQL queries when compiling your Scala code
- Automatically infer return types

```scala
def personsMatching(pattern: String)(implicit s: Session) =
  tsql"select id, name from person where name like $pattern"
    .list
```
Logging

- Result Set Summaries, Statements, Execution Times
- ANSI Colors and Unicode Symbols
- Configured via Typesafe Config (application.conf)
Logging

- Result Set Summaries, Statements, Execution Times
- ANSI Colors and Unicode Symbols
- Configured via Typesafe Config (application.conf)