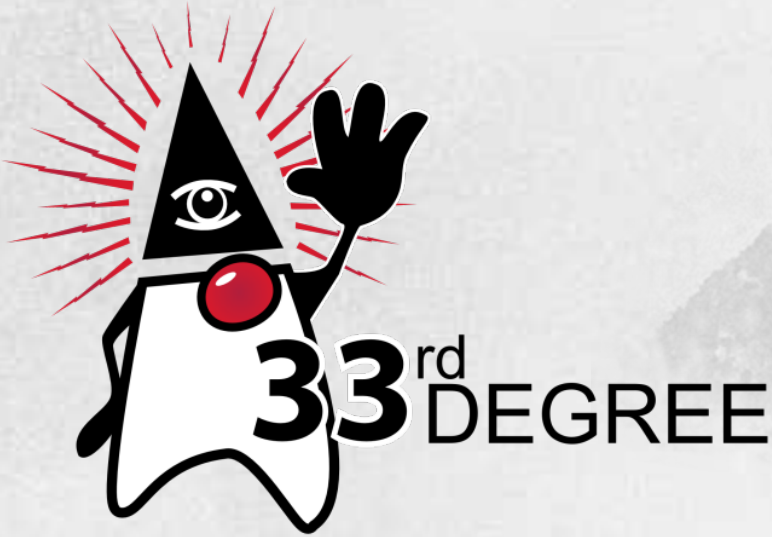
Platinum Sponsor

FUNCTIONAL RELATIONAL MAPPING WITH SLICK

Stefan Zeiger, Typesafe



Object Relational Mapping

Object

Relational

Object

Impedance Mismatch

Relational

Concepts

Object-Oriented	Relational
Identity	No Identity
State	Transactional State
Behavior	No Behavior
Encapsulation	No Encapsulation

Execution


```
select NAME
from COFFEES
```

```
select c.NAME, c.PRICE, s.NAME
from COFFEES c
join SUPPLIERS s
  on c.SUP_ID = s.SUP_ID
where c.NAME = ?
```

Execution

Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99
French_Roast_Decaf	9.99

```
def getAllCoffees(): Seq[Coffee] = …

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

Execution


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

Level of Abstraction

	Object Oriented	Relational
Data Organization	High	Low
Data Flow	Low	High

Functional Relational Mapping

Relational Model

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

COFFEES		
NAME : String	**PRICE** : Double	**SUP_ID** : Int
Colombian	7.99	101
French_ Roast	8.99	49
Espresso	9.99	150

Relational Model

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

COFFEES		
NAME : String	**PRICE** : Double	**SUP_ID** : Int
Colombian	7.99	101
French_ Roast	8.99	49
Espresso	9.99	150

Relational Model

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

COFFEES		
NAME : String	**PRICE** : Double	**SUP_ID** : Int
Colombian	7.99	101
French_ Roast	8.99	49
Espresso	9.99	150

Relational Model

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

COFFEES		
NAME : String	PRICE : Double	SUP_ID : Int
Colombian	7.99	101
French_ Roast	8.99	49
Espresso	9.99	150

Relational Model

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

COFFEES		
NAME : String	PRICE : Double	SUP_ID : Int
Colombian	7.99	101
French_ Roast	8.99	49
Espresso	9.99	150

Relational Model

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

COFFEES		
NAME : String	PRICE : Double	SUP_ID : Int
Colombian	7.99	101
French_ Roast	8.99	49
Espresso	9.99	150

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

```
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

```
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

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

```
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

```
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

```
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

```
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

Browse code

/ src / main / scala

CaseClassMapping.scala

HelloSlick.scala

InvokerMethods.scala

Tables.scala

src/main/scala/HelloSlick.scala

Delete Revert Save

```scala
import scala.slick.driver.H2Driver.simple._

// The main application
object HelloSlick extends App {

  // The query interface for the Suppliers table
  val suppliers: TableQuery[Suppliers] = TableQuery[Suppliers]

  // the query interface for the Coffees table
  val coffees: TableQuery[Coffees] = TableQuery[Coffees]

  // Create a connection (called a "session") to an in-memory H2
  val db = Database.forURL("jdbc:h2:mem:hello", driver = "org.h2.
  db.withSession { implicit session =>

    // Create the schema by combining the DDLs for the Suppliers
    // tables using the query interfaces
    (suppliers.ddl ++ coffees.ddl).create


    /* Create / Insert */

    // Insert some suppliers
    suppliers += (101, "Acme, Inc.", "99 Market Street", "Grounds
    suppliers += ( 49, "Superior Coffee", "1 Party Place", "Mendo
    suppliers += (150, "The High Ground", "100 Coffee Lane", "Mea
```

Tutorial

Run the App and the Tests

This template includes a simple Scala application, HelloSlick.scala, that does basic FRM operations with Slick. This application automatically runs when Activator is started and then re-runs after every successful compile. You can see the output in Run. Note: The example code in this app has intentionally verbose type information. In normal applications type inference is used more extensively but to assist with learning the type information was included.

The TablesSuite.scala file contains ScalaTest tests which do some basic integration tests. Check out the test results in Test.

http://typesafe.com/activator

Schema Definition

Table Definition

```
class Suppliers(tag: Tag) extends
    Table[(Int, String, String)](tag, "SUPPLIERS") {
  def id = column[Int]("SUP_ID",
                       O.PrimaryKey, O.AutoInc)
  def name = column[String]("NAME")
  def city = column[String]("CITY")
  def * = (id, name, city)
}

val suppliers = TableQuery[Suppliers]
```

Custom Row Types

```
case class Supplier(id: Int, name: String,
  city: String)

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

val suppliers = TableQuery[Suppliers]
```

Custom Column Types

```
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", ...)
  ...
}
```

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", ...)
  ...
}
```

Foreign Keys

```
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

- New in Slick 2.0
- Reverse-engineer an existing database schema
- Create table definitions and case classes
- Customizable
- Easy to embed in sbt build

Data Manipulation

Session Management

```
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

```
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)
)
```

Auto-Generated Keys

```
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

Queries

Query[(Column[String], Column[String]), (String, String)]

TableQuery[Coffees]

ColumnExtensionMethods.<

```
val q = for {
  c <- coffees if c.price < 9.0
  s <- c.supplier
} yield (c.name, s.name)
```

Coffees

Suppliers

ConstColumn(9.0)

(Column[String], Column[String])

Column[Double]

```
val result = q.run (session)
```

Seq[(String, String)]

Plain SQL

JDBC

```
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()
}
```

Slick: Plain SQL Queries

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

Compile-Time Checking of SQL

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

Expected in Slick 2.2

slick.typesafe.com

@StefanZeiger