FUNCTIONAL RELATIONAL MAPPING WITH SLICK

Stefan Zeiger, Typesafe
Object Relational Mapping
Object

Relational
Object

Impedance Mismatch

Relational
## 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>
Execution

Colombian  
French_Roast  
Espresso  
Colombian_Decaf  
French_Roast_Decaf

Espresso
Price: 9.99
Supplier: The High Ground

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 = ?
def getAllCoffees(): Seq[Coffee] = ...
def printLinks(s: Seq[Coffee]) {
  for(c <- s) println(c.name + c.price)
}
def printDetails(c: Coffee) {
    println(c.name)
    println("Price: "+c.price)
    println("Supplier: "+c.supplier.name)
}
# 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></td>
<td>Low</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Flow</th>
<th>Object Oriented</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td></td>
<td>High</td>
</tr>
</tbody>
</table>
Functional Relational Mapping
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>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_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>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>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>

Colombian: String
PRICE: Double
SUP_ID: Int
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_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>COFFEES</th>
<th>NAME</th>
<th>PRICE</th>
<th>SUP_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colombiam</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
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
Schema 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]
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]
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", ...)
    ...
}

Functional Relational Mapping with Slick
class Coffees(tag: Tag) extends Table[
  (String, SupplierId, Double)](tag, "COFFEES") {
  def name = column[String]("NAME", 0.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
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)
)
```
Auto-Generated Keys

```scala
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)

Seq[ (String, String) ]
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()
}
Slick: Plain SQL Queries

```scala
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

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

Expected in Slick 2.2