Typesafe

Slick

A Library for Functional Database Access

Brendan McAdams <brendan@typesafe.com>

Typesafe

Slick

Scala Language Integrated Connection Kit

Typesafe

Slick

An Elegant, Functional Database Library...

...For a More Civilized Age

Typesafe

Slick

Why are we here?

And why I'd like you to hang around for the next hour...

- Key Concepts
  - Functional Data Concepts: How We Work With Collections in Scala
  - Impedance Mismatch: Where Code and Data Fail To Meet
  - Mapping Scala's Functional Collections to Databases

Typesafe

Slick

What we aim to solve with Slick

Code + Data in Perfect Harmony

- Slick is a database query & access library for Scala
  - Write native Scala syntax
  - Let Scala generate your data access
  - No more SQL – without needing to go NoSQL*
- Slick 1.0 is out now, requires Scala 2.10+
- From the fine folks who brought you Scala...
  - Developed at Typesafe & EPFL

Typesafe

Slick

What Does Slick Support?

Focused around relational databases to begin with....

- PostgreSQL
- MySQL
- Microsoft SQL Server*
- SQLite
- H2
- Derby / JavaDB
- HyperSQL (HSQLDB)
- Microsoft Access

Closed Source Extensions
(For Typesafe Licensees)

- Oracle
- DB2

Coming 'Real Soon Now'
Evaluating NoSQL Support...

- MongoDB
- Riak
- Cassandra

Typesafe

Slick

Let's Talk About Functional Data Access

How do we manipulate Collections in Scala?

- My esteemed colleague (Stefan – @StefanZeiger) has previously done some demos around Coffee
- I decided to change that up
  - I wanted my own set of demos, in a different style
  - These days, I prefer tea to coffee
- So let's look at how we might track some data about Tea in Scala
- We are working entirely with "in-memory" collections of data

Typesafe

Slick

A Little Scala to Start The Day

Here's our basic structure

... Now let's put together some lists of Tea!

An Enumeration of limited values for "Color" of Tea

```
object TeaTypes extends Enumeration {
  type Color = Value
  val Black, White, Oolong, Green = Value
}
```

A container class for Tea vendors

```
case class Supplier(name: String, country: String, url: java.net.URI)
```

```
case class Tea(supplier: Supplier, name: String, kind: TeaTypes.Color,
               size: String, currency: Character, price: Double)
```

A container class for Tea types (from specific vendors)

Typesafe

Slick

Who Do We Buy Our Tea From?

A Few Places I've Encountered Around The World

... Now we know who's selling. What do they have to offer?

```
    // Suppliers
val Stash = Supplier("Stash Tea", "USA",
                     new URI("http://stashtea.com"))

val Mariage = Supplier("Mariage Frères", "France",
                       new URI("http://mariagefreres.com"))

val Postcard = Supplier("Postcard Teas", "England",
                        new URI("http://postcardteas.com"))

val TGTea = Supplier("TeaGschwndner", "Germany",
                     new URI("http://shop.tgtea.com"))

val SeattleTeaCup = Supplier("Seattle Teacup", "USA",
                             new URI("http://seattleteacup.com"))

val Palais = Supplier("Le Palais De Thés", "France",
                      new URI("http://us.palaisdethes.com/en_us"))
```

Typesafe

Slick

Mmm... Tea

A few random choices from our various vendors

```
   // Stash
val Darjeeling = Tea(Stash, "Darjeeling Estate Golden Tipped",
                     TeaTypes.Black, "100g", '$', 15.00)
val IrishBreakfast = Tea(Stash, "Irish Breakfast",
                         TeaTypes.Black, "100g", '$', 7.50)
val ChinaKeemun = Tea(Stash, "China Keemun",
                      TeaTypes.Black, “100g", '$', 7.50)
val MoroccanMint = Tea(Stash, "Moroccan Mint Green Tea",
                       TeaTypes.Green, "100g", '$', 7.50)

// Mariage Frères
val BeyondSkies = Tea(Mariage, "White Tea from beyond the Skies™",
                      TeaTypes.White, "100g", '€', 105.00)
val BlueHimalaya = Tea(Mariage, "Blue Himalaya™",
                       TeaTypes.Oolong, "100g", '€', 28)
val GoldenJamguri = Tea(Mariage, "Golden Jamguri SFTGFOP1",
                        TeaTypes.Black, "100g", '€', 60)
```

Typesafe

Slick

Mmm... Tea

A few random choices from our various vendors

So how would we collect and manipulate this in Scala?

```
// Postcard
val EarlGrey = Tea(Postcard, "Gianfranco's Earl Grey",
                   TeaTypes.Black, "50g", '£', 6.45)
val SuperGreen = Tea(Postcard, "Master Matsumoto's Supernatural Green",
                     TeaTypes.Green, "50g", '£', 11.95)

// Le Palais De Thés
val DarjeelingHilton = Tea(Palais, "2012 DARJEELING HILTON DJ1 S.F.T.P.G.F.O.P.1",
                           TeaTypes.Black, "100g", '$', 56.00)
```

Typesafe

Slick

Creating a Collection

Putting it all in one place

Scala's Type Inference Figures Out It's a Seq of 'Tea'

...I'm not going to go into the ins and outs of the various collection types we might have chosen. Seq is a good base.

```
val tea = Seq(Darjeeling, IrishBreakfast, ChinaKeemun,
              MoroccanMint, BeyondSkies, BlueHimalaya,
              GoldenJamguri, EarlGrey, SuperGreen,
              DarjeelingHilton)
```

Typesafe

Slick

Creating a Collection

Putting it all in one place

It would be the same if we explicitly declared the type

What exactly can we do with this sequence?

```
val tea: Seq[Tea] = Seq[Tea](Darjeeling, IrishBreakfast,
                             ChinaKeemun, MoroccanMint,
                             BeyondSkies, BlueHimalaya,
                             GoldenJamguri, EarlGrey, SuperGreen,
                             DarjeelingHilton)
```

Typesafe

Slick

Let's Talk About a Few Core Concepts

How Exactly Do We Work With Scala Collections?

- First and foremost
  - We are talking about **Function**al Programming
  - If you guessed that functions are somehow involved, congratulations!
- You might have heard these called '**Lambdas**' in some places, including the plans for their inclusion in Java 8
  - A **Lambda** usually refers to an "anonymous" function
  - As opposed to a **method**, which is named and concrete
    - (Scala can automatically "*lift*" methods into functions as needed)

Typesafe

Slick

Functions & Methods

We're Starting Towards that Elegant, Civilized Part I Hinted At

Takes one argument, of the type 'Tea'

Evaluates if the Tea's currency is '$'

```
  // this is a method - well defined, and we can call it normally
def costsDollars(t: Tea): Boolean = t.currency == '$'
```

Returns a Boolean

```
// this is a function - I just happen to have captured it
val costsEuros = (t: Tea) => t.currency == '€'
```

Typesafe

Slick

A Closer Look At A Function

A Core Unit of Work

The => (rocket) indicates a function (lamdba)

```
(t: Tea) => t.currency == '$'
```

The Left side declares the argument(s) to the function

The Right side type is inferred here (`Boolean`)

We can use this to define functions/methods that take functions... (Higher Order Functions)

```
def filter(p: (Tea) => Boolean): Seq[Tea]
```

...In fact, `filter` is one of the methods built-in to most Scala collections

Typesafe

Slick

Using Functions To Manipulate Sequences

Higher Order Functions in Action

```
def filter(p: (Tea) => Boolean): Seq[Tea]
```

predicate, evaluated against each entry for truth

returns a new Seq[Tea], with only 'true' items

A new Seq[Tea]

```
// pass a function we write inline
val inEuros = tea.filter(t => t.currency == '€')

// scala will also "lift" a method into a
// function when it needs to
val inDollars = tea.filter(costsDollars)
```

A new Seq[Tea]

Typesafe

Slick

Using Functions To Manipulate Sequences

Higher Order Functions in Action

```
def exists(p: (Tea) => Boolean): Boolean
```

do any elements matching the predicate exist in the Seq?

```
def groupBy[K](f: (Tea) => K): Map[K, Seq[Tea]]
```

K can be any type we want, and is used for the grouping

for each entry, "transform" it to **K**

We'll get back a Map with sub-sequences grouped by keys of **K**

Typesafe

Slick

Using Functions to Drive Transformations

Changing This to That...

```
def groupBy[K](f: (Tea) => K): Map[K, Seq[Tea]]
```

```
val typesOfTea = tea.groupBy(t => t.kind)
```

`Map[TeaTypes.Color, Seq[Tea]]`

`K = TeaTypes.Color`

```
// we can easily recast as a String
val typesOfTea = tea.groupBy(t => t.kind.toString)
```

`Map[String, Seq[Tea]]`

`K = String`

```
TeaEntries.tea.groupBy(t => t.kind.toString).keys
// Set[String](White, Oolong, Black, Green)
```

Typesafe

Slick

One Last Transformation Function

(At Least, That We Need for Today)

We'll get back a new Seq of **B**

```
def map[B](f: (Tea) => B): Seq[B]
```

```
val prices = tea.map(t => "%c%3.2f per %s".format(
                            t.currency, t.price, t.size)
```

Typesafe

Slick

Slick in Action

Let's create our tables...

```
    // Definition of the VENDORS table
  object Vendors extends Table[(Int, String, String, String)]("VENDORS") {
    def id = column[Int]("VENDOR_ID", O.PrimaryKey) // This is the primary key column
    def name = column[String]("VENDOR_NAME")
    def country = column[String]("VENDOR_COUNTRY")
    def url = column[String]("VENDOR_URL")
    // Every table needs a * projection with the same type as the table's type
parameter
    def * = id ~ name ~ country ~ url
  }


// Definition of the TEAS table
object Teas extends Table[(String, Int, String, String, Double, String)]("TEAS") {
  def name = column[String]("TEA_NAME", O.PrimaryKey)
  def vendorID = column[Int]("VENDOR_ID")
  def kind = column[String]("TEA_KIND")
  def currency = column[String]("PRICE_CURRENCY")
  def price = column[Double]("PRICE")
  def size = column[String]("PACKAGE_SIZE")
  def * = name ~ vendorID ~ kind ~ currency ~ price ~ size
  // A reified foreign key relation that can be navigated to create a join
  def vendor = foreignKey("VENDOR_ FK", vendorID, Vendors)(_.id)
}
```

Typesafe

Slick

Sessions & DDLs

```
    // Use the implicit threadLocalSession
import Database.threadLocalSession
// Connect to the database and execute the following block within a session
Database.forURL("jdbc:h2:mem:test1", driver = "org.h2.Driver") withSession {
  // The session is never named explicitly. It is bound to the current
  // thread as the threadLocalSession that we imported

  // Create the tables, including primary and foreign keys
  (Vendors.ddl ++ Teas.ddl).create

  // Insert some suppliers
  Vendors.insert(1, "Stash", "USA", "http://stashtea.com")
  Vendors.insert(2, "Mariage Frères", "France", "http://mariagefreres.com")
  Vendors.insert(3, "Postcard Teas", "England", "http://postcardteas.com")
  Vendors.insert(4, "Silk Road Teas", "USA", "http://silkroadteas.com")
  Vendors.insert(5, "TeaGschwndner", "Germany", "http://shop.tgtea.com")
  Vendors.insert(6, "Seattle Teacup", "USA", "http://seattleteacup.com")
  Vendors.insert(7, "Le Palais De Thés", "France", "http://us.palaisdethes.com/en_us")

  // Insert some tea (using JDBC's batch insert feature, if supported by the DB)
  Teas.insertAll(
    ("Darjeeling Estate Golden Tipped", 1, "Black", "$", 15.00, "100g"),
    ("Irish Breakfast", 1, "Black", "$", 7.50, "100g"),
    ("China Keemun", 1, "Black", "$", 7.50, "100g"),
    ("Moroccan Mint Green Tea", 1, "Green", "$", 7.50, "100g"),
    ("White Tea from beyond the Skies™", 2, "White", "€", 105.00, "100g"),
    ("Blue Himalaya™", 2, "Oolong", "€", 28.00, "100g"),
    ("Golden Jamguri SFTGFOP1", 2, "Black", "€", 60.00, "100g"),
    ("Gianfranco's Earl Grey", 3, "Black", "£", 6.45, "50g"),
    ("Master Matsumoto's Supernatural Green", 3, "Green", "£", 11.95, "50g"),
    ("2012 Darjeeling Hilton DJ1 SFTPGFOP1", 7, "Black", "$", 56.00, "100g")
  )
```

Typesafe

Slick

Basic Data Iteration

```
    // Iterate through all coffees and output them
    println("Teas:")
    Query(Teas) foreach { case (name, vendorID, kind, currency, price, size) =>
      println("  " + name + "\t" + vendorID + "\t" + kind + "\t" + currency +
price + "\t" + size)
    }
```

```
Teas:
  Darjeeling Estate Golden Tipped  1   Black   $15.0   100g
  Irish Breakfast   1   Black   $7.5    100g
  China Keemun  1   Black   $7.5    100g
  Moroccan Mint Green Tea  1   Green   $7.5    100g
  White Tea from beyond the Skies™ 2   White   €105.0  100g
  Blue Himalaya™    2   Oolong  €28.0   100g
  Golden Jamguri SFTGFOP1  2   Black   €60.0   100g
  Gianfranco's Earl Grey   3   Black   £6.45   50g
  Master Matsumoto's Supernatural Green    3   Green   £11.95  50g
  2012 Darjeeling Hilton DJ1 SFTPGFOP1 7   Black   $56.0   100g
```

Typesafe

Slick

Basic Data Iteration

```
// Why not let the database do the string conversion and concatenation?
println("Teas (concatenated by DB):")
val q1 = for(t <- Teas) // Teas lifted automatically to a Query
  yield ConstColumn("  ") ++ t.name ++ "\t" ++ t.vendorID.asColumnOf[String] ++
    "\t" ++ t.currency.asColumnOf[String] ++ t.price.asColumnOf[String] ++
    "\t" ++ t.size.asColumnOf[String]
// The first string constant needs to be lifted manually to a ConstColumn
// so that the proper ++ operator is found
q1 foreach println
```

```
Teas (concatenated by DB):
  Darjeeling Estate Golden Tipped  1   $15.0   100g
  Irish Breakfast   1   $7.5    100g
  China Keemun  1   $7.5    100g
  Moroccan Mint Green Tea  1   $7.5    100g
  White Tea from beyond the Skies™ 2   €105.0  100g
  Blue Himalaya™    2   €28.0   100g
  Golden Jamguri SFTGFOP1  2   €60.0   100g
  Gianfranco's Earl Grey   3   £6.45   50g
  Master Matsumoto's Supernatural Green    3   £11.95  50g
  2012 Darjeeling Hilton DJ1 SFTPGFOP1 7   $56.0   100g
```

Typesafe

Slick

Resolving Relationships

```
// Perform a join to retrieve tea names and supplier names for
// all the really good stuff (regardless of currency)
println("Manual join:")
val q2 = for {
  t <- Teas if t.price > 25.00
  v <- Vendors if v.id === t.vendorID
} yield (t.name, v.name)
for(r <- q2) println("  " + r._1 + " supplied by " + r._2)
```

```
Manual join:
  White Tea from beyond the Skies™ supplied by Mariage Frères
  Blue Himalaya™ supplied by Mariage Frères
  Golden Jamguri SFTGFOP1 supplied by Mariage Frères
  2012 Darjeeling Hilton DJ1 SFTPGFOP1 supplied by Le Palais De
Thés
```

Typesafe

Slick

Resolving Relationships

```
// Do the same thing using the navigable foreign key
println("Join by foreign key:")
val q3 = for {
  t <- Teas if t.price > 25.00
  v <- t.vendor
} yield (t.name, v.name)
// This time we read the result set into a List
val l3: List[(String, String)] = q3.list
for((r1, r2) <- l3) println("  " + r1 + " supplied by " + r2)
```

```
Join by foreign key:
  White Tea from beyond the Skies™ supplied by Mariage Frères
  Blue Himalaya™ supplied by Mariage Frères
  Golden Jamguri SFTGFOP1 supplied by Mariage Frères
  2012 Darjeeling Hilton DJ1 SFTPGFOP1 supplied by Le Palais De
Thés
```

```
// Check the SELECT statement for that query
println(q3.selectStatement)
```

```
select x2."TEA_NAME", x3."VENDOR_NAME" from "TEAS" x2, "VENDORS" x3
where (x2."PRICE" > 25.0) and (x3."VENDOR_ID" = x2."VENDOR_ID")
```

Typesafe

Slick

Other Computation

```
// Compute the number of coffees by each supplier
 println("Teas per supplier:")
 val q4 = (for {
   t <- Teas
   v <- t.vendor
 } yield (t, v)).groupBy(_._2.id).map {
   case (_, q) => (q.map(_._2.name).min.get, q.length)
 }
 // .get is needed because SLICK cannot enforce statically that
 // the supplier is always available (being a non-nullable foreign key),
// thus wrapping it in an Option
q4 foreach { case (name, count) =>
  println("  " + name + ": " + count)
}
```

```
Teas per supplier:
  Stash: 4
  Mariage Frères: 3
  Postcard Teas: 2
  Le Palais De Thés: 1
```

Typesafe

Slick

Typesafe

Slick

Databases and Code Have Come Into Conflict

The last few years have shown a move towards models that bring our code + data into harmony

- Developers have been moving to databases such as MongoDB
  - Not just for scalability, but for ease of data interaction
  - We want the way we work with our data to match closely with how our data is stored
- There's a tremendous impedance mismatch with SQL & Object Oriented Programming
  - Something as simple as a field projection doesn't match up well
  - We end up building new and ridiculous DSLs to provide querying from our OO world

Typesafe

Slick

Databases and Code Have Come Into Conflict

The last few years have shown a move towards models that bring our code + data into harmony

- The functional language we use to work with Scala collections
  - Maps cleanly to how we manipulate collections (with a few extra operators too)

```
case class Tea(supplier: Supplier, name: String,
               kind: TeaTypes.Color, size: String,
               currency: Character, price: Double)

val tea = Seq(Darjeeling, IrishBreakfast, ChinaKeemun,
              MoroccanMint, BeyondSkies, BlueHimalaya,
              GoldenJamguri, EarlGrey, SuperGreen,
              DarjeelingHilton)
```

A Relation

Attribute / Field

A Table

Typesafe

Slick

Why Not Compose Our Own SQL?

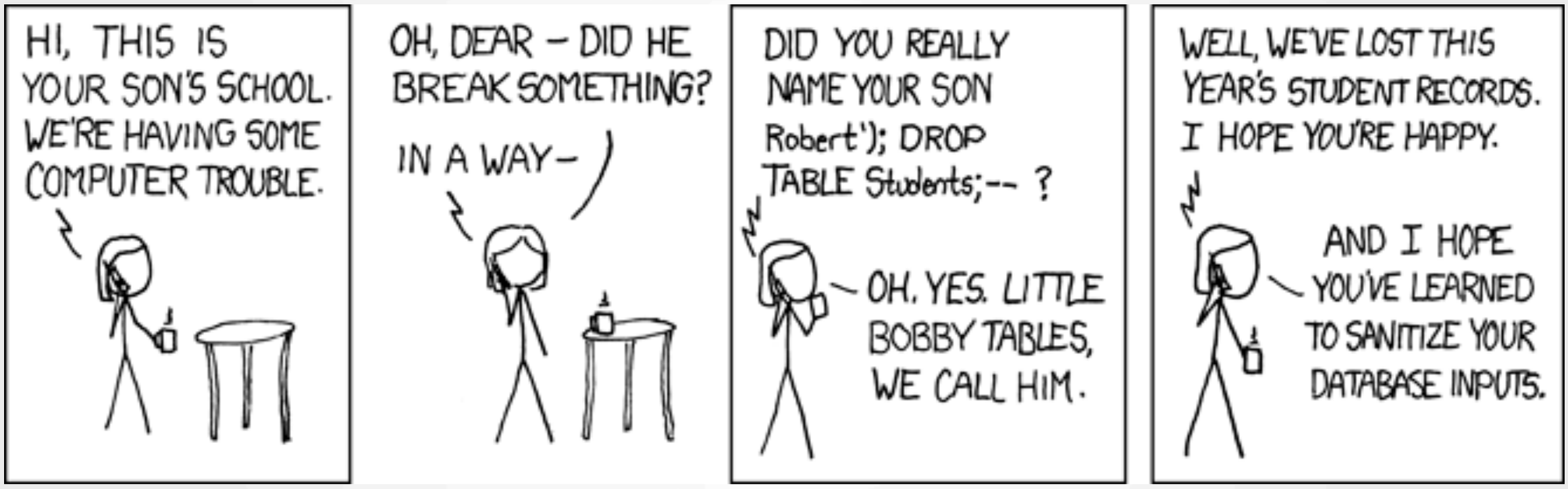
There's that word – compose – which explains much of it

- SQL doesn't compose
  - We can't easily add new manipulations onto our existing query
- Generating SQL via string manipulation is awkard
- Spelling mistakes & type errors aren't caught at compile-time
  - We lose our whole "compiled language" advantage when our SQL blows up at runtime
- SQL Injection – the bane of every developer

Typesafe

Slick

SQL Injection... fun!



http://xkcd.com/327/

Typesafe

Slick

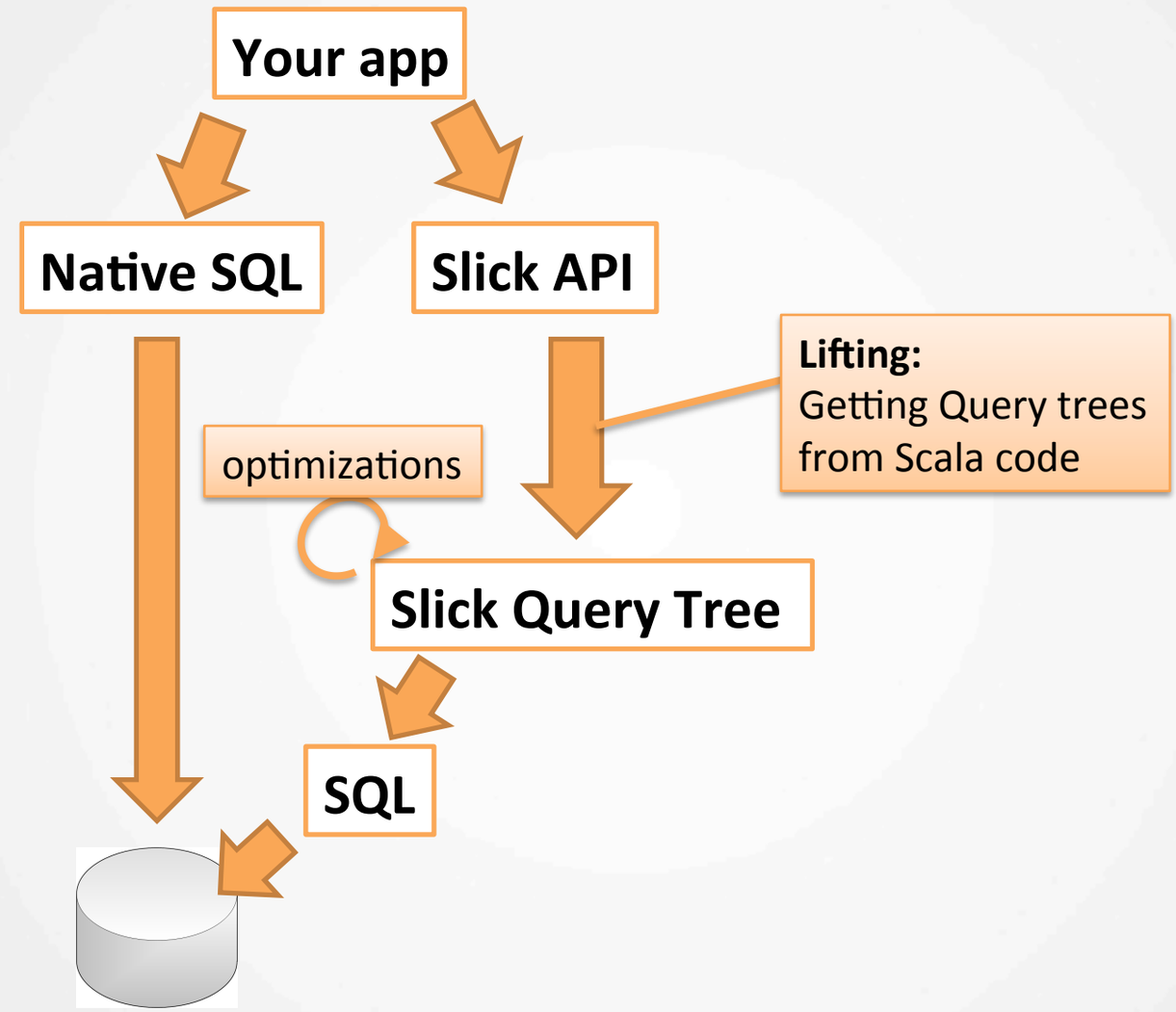
Types of Slick Interaction

- We've been looking so far at the "core" Slick API, known as "Lifted Embedding"
  - It's possible to do other things like invoke stored procedures and call raw SQL - that's a topic for another time
- Let's take a quick look at how the Lifted Embedding works, and a few other example queries

Typesafe

Slick

Under The Hood



Typesafe

Slick

How Query Lifting Works

```
for( p <- Persons if p.name === “Brendan”) yield p.name
```

Scala desugaring

Column[String]

String (implicitly to Column[String])

```
Persons.filter(p=>p.name === “Brendan").map(p=>p.name)
```

```
Projection(
  Filter(
    Table( Person ),
    Equals(
      ColumnRef( "p", "name" ),
      Constant( name )
    )
  ),
  ColumnRef( "p", "name" )
)
```

```
"select name
        from person
        where name = ‘Brendan’"
```

Typesafe

Slick

Sorting and Paging

```
Persons
 .sortBy(_.name)
 .drop(5).take(10)
```

Typesafe

Slick

Grouping and Aggregation

```
// Number of people per age

Persons
.groupBy(_.age)
.map( p =>( p._1, p._2.length ) )
```

Typesafe

Slick

First Entry

```
// person 3

Persons.filter(_.id === 3).first
```

Typesafe

Slick

Unions

```
Persons.filter(_.age < 18)
  unionAll
    Persons.filter(_.age > 65)
```

Typesafe

Slick

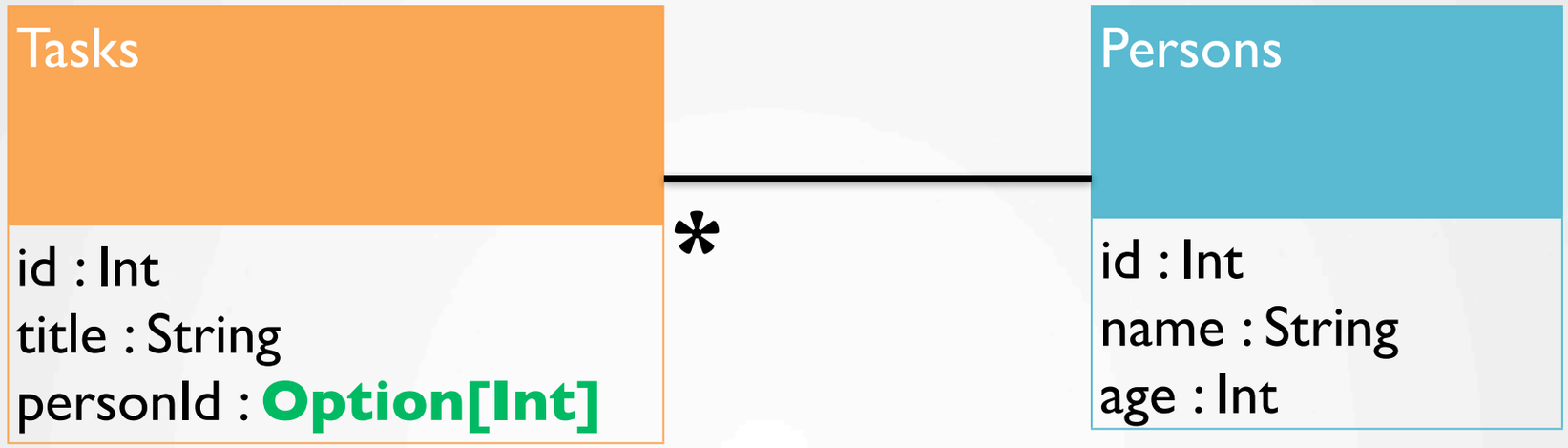
NULL Support

```
case class Person( ..., age : Option[Int] )

object Persons extends Table[Person]("person"){
  def age = column[Option[Int]]("id")
  ...
}

Persons.insertAll(
  Person( 1, „Chris“, Some(22) ),
  Person( 2, „Stefan“, None )
)
```

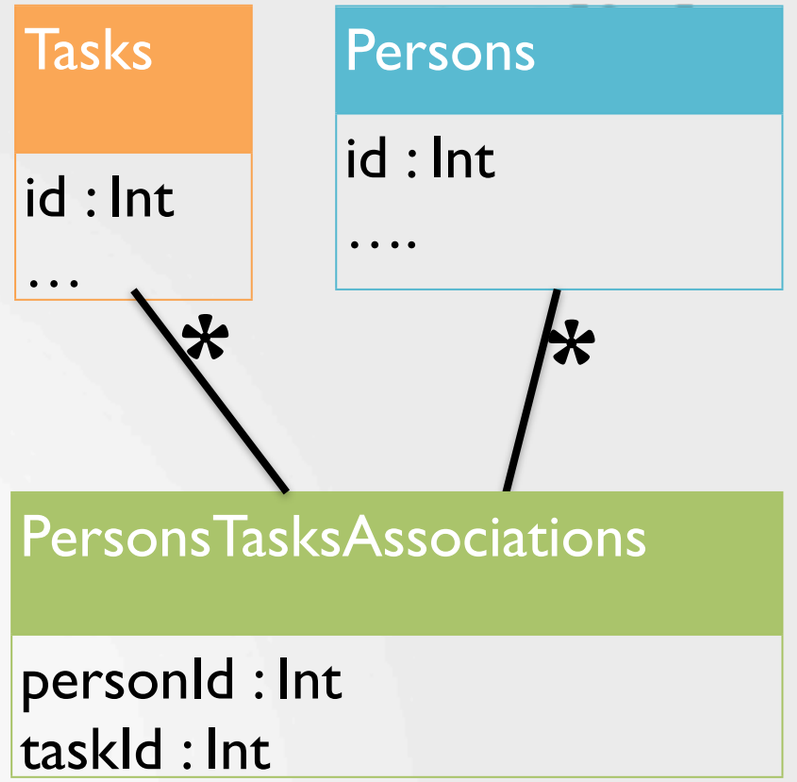
Outer Joins (left, right, full)




```
for (
  Join(p, t) <- Tasks outerJoin Persons
                      on (_.personId === _.id)
) yield p.title.? ~ t.name.?
```

Typesafe

Relationships




```
object Persons extends Table[Person]("person"){
  def id = column[Int]("id")
  ...
}
object Tasks extends Table[Task]("task"){
  def id = column[Int]("id")
  ...
  def assignees = for( pt <- PersonsTasksAssociations;
                       p <- pt.assignee; if pt.taskId === id ) yield p
}
object PersonsTasksAssociations extends Table[(Int,Int)]("person_task"){
  def personId = column[Int]("person_id")
  def taskId = column[Int]("task_id")
  def assignee = foreignKey( "person_fk", personId, Persons )(_.id)
  ...
}
```

Assignees of task 1:

```
for( t <- Tasks; ps <- t.assignees; if
        t.id === 1 ) yield ps
```

Column Operators

Common: .in(Query), .notIn(Query), .count, .countDistinct, .isNull, .isNotNull, .asColumnOf, .asColumnOfType

Comparison: === (.is), =!= (.isNot), <, <=, >, >=, .inSet, .inSetBind, .between, .ifNull

Numeric: +, -, *, /, %, .abs, .ceil, .floor, .sign, .toDegrees, .toRadians

Boolean: &&, ||, .unary_!

String: .length, .like, ++, .startsWith, .endsWith, .toUpperCase, .toLowerCase, .ltrim, .rtrim, .trim

Typesafe

Slick

Other Features

Far From Exhaustive

- auto-increment
- sub-queries
- CASE
- prepared statements
- custom data types
- foreach-iteration
- …

Typesafe

Slick

The Future Looks Bright, Too

We are working on a lot more

- New backend architectures
- Type Providers (using Type Macros)
- Distributed Querying (Joins against multiple datasources)
- NoSQL Support

Typesafe

Slick

Thanks!

brendan@typesafe.com
@rit

http://slick.typesafe.com

http://typesafe.com