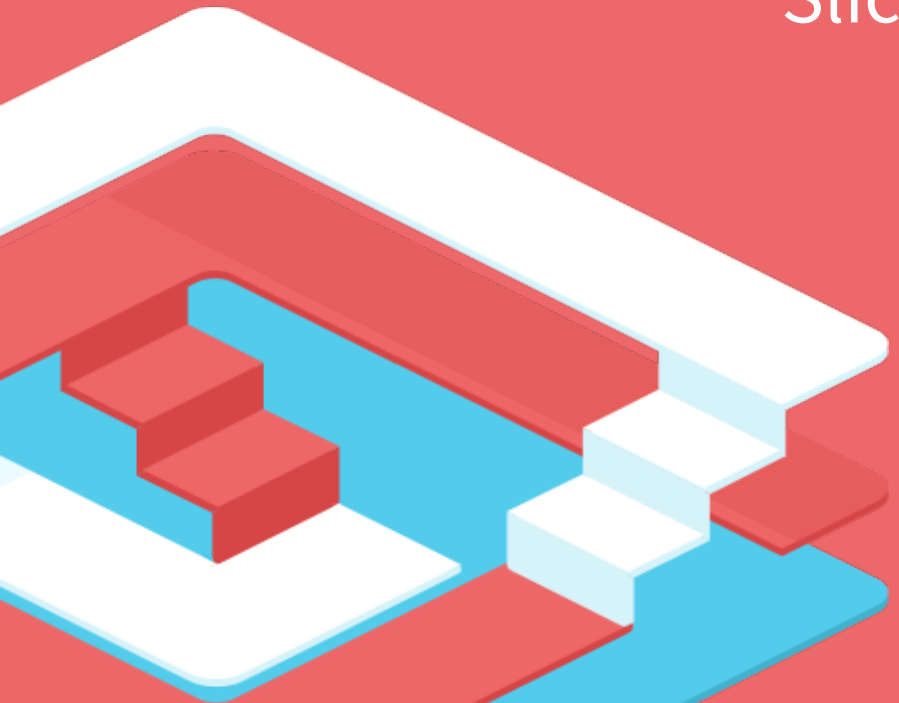
So Slick!
An introduction

Jan Christopher Vogt, EPFL
Slick Team



Scala User Group
Berlin Brandenburg

(vs. ORM)

- **Functional-Relational Mapper**
- natural fit (no impedance mismatch)
- declarative
- embraces relational
- stateless
- Slick is to ORM what Scala is to Java

8 Reasons for using Slick

1
Scala collection-like API

Scala collection-like API

```
for ( d <- Devices;
      if d.price > 1000.0
) yield d.acquisition
```

Device

id: Long
price: Double
acquisition: Date

```
Devices
  .filter(_.price > 1000.0)
  .map(_.acquisition)
```

2
Predictable SQL structure

Predictable SQL structure

```
Devices
   .filter(_.price > 1000.0)
   .map(_.acquisition)
   .selectStatement
```

```
select x2."ACQUISITION" from "DEVICE"
x2 where x2."PRICE" > 1000.0
```

3
Type-safety

Compile-Time Safety

- Spelling mistake in column name?
- Wrong column type?
- Query doesn’t match expected result type?

scalac sees it all!

Caution: Error messages can be bad

Piotr Buda @piotrbuda

...and the 'Most Informative Stack Trace Award goes to...' evernote.com/shard/s28/sh/5... #slick #scala

12 hours ago

```
overloaded method value <> with alternatives:
  [R(in method <>)(in method <>)(in method <>)(in method <>)(in method <>)(in
<>)(in method <>)(in method <>)(in method <>)(in method <>), g: R(in method <
String)])scala.slick.lifted.MappedProjection[R(in method <>)(in method <>)(in
  [R(in method <>)(in method <>)(in method <>)(in method <>)(in method <>)(in
com.upnext.wirespring.kernel.domain.Terminal.TerminalId)) => R(in method <>)(
<>)(in method <>)(in method <>)(in method <>)(in method <>)(in method <>)(in
com.upnext.wirespring.kernel.domain.Terminal.TerminalId)])scala.slick.lifted.
<>),(Option[com.upnext.wirespring.kernel.domain.Transaction.TransactionId], c
 cannot be applied to ((com.upnext.wirespring.kernel.domain.Transaction.Trans
com.upnext.wirespring.kernel.domain.Customer.CustomerId, Double, com.upnext.w
com.upnext.wirespring.kernel.domain.Transaction => Some[(com.upnext.wiresprin
com.upnext.wirespring.kernel.domain.Merchant.MerchantId, com.upnext.wiresprin
  def * = transactionId.? ~ terminalId <>(
```

Enforce schema consistency

- Generate DDL from table classes
- Slick 2.0: Generate table classes and mapped classes from database

4
Small configuration using Scala code

Table description

```
class Devices(tag: Tag)
      extends Table[(Long, Double, Date)](tag,"DEVICES") {
  def id          = column[Long]  ("ID", O.PrimaryKey)
  def price       = column[Double]("PRICE")
  def acquisition = column[Date]  ("ACQUISITION")
  def * = (id, price, acquisition)

}
def Devices = TableQuery[Devices]
```

can be auto-generated in Slick 2.0

Connect

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

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

db.withTransaction { implicit session =>

  // <- run queries here

}
```

5
Explicit control over execution and transfer

Execution control

```
val query = for {
    d <- Devices
      if d.price > 1000.0
  } yield d.acquisition
```

Device
id: Long
price: Double
acquisition: Date

```
db.withTransaction { implicit session =>

  val acquisitonDates = query.run(session)

}
```

no unexpected behavior,
no loading strategy configuration,
just write code

6
Loosely-coupled, flexible mapping

Table description

```
class Devices(tag: Tag)
      extends Table[(Long, Double, Date)](tag,"DEVICES") {
  def id          = column[Long]  ("ID", O.PrimaryKey)
  def price       = column[Double]("PRICE")
  def acquisition = column[Date]  ("ACQUISITION")
  def * = (id, price, acquisition)

}
val Devices = TableQuery[Devices]
```

Table description

```
class Devices(tag: Tag)
extends Table[Long :: Double :: Date :: HNil)](tag,"DEVICES") {
  def id          = column[Long]  ("ID", O.PrimaryKey)
  def price       = column[Double]("PRICE")
  def acquisition = column[Date]  ("ACQUISITION")
  def * = id :: price :: acquisition :: HNil

}
val Devices = TableQuery[Devices]
```

case class mapping

```
case class Device(id: Long,
   price: Double,
   acquisition: Date)

class Devices(tag: Tag)
      extends Table[Device](tag,"DEVICES") {
  def id          = column[Long]  ("ID", O.PrimaryKey)
  def price       = column[Double]("PRICE")
  def acquisition = column[Date]  ("ACQUISITION")

  def * = (id, price, acquisition) <>
            (Device.tupled,Device.unapply)
}
val Devices = TableQuery[Devices]
```

Custom mapping

```
def construct : ((Long,Double,Date)) => CustomType
def extract: CustomType => Option[(Long,Double,Date)]
```

```
class Devices(tag: Tag)
      extends Table[CustomType](tag,"DEVICES") {
  def id          = column[Long]  ("ID", O.PrimaryKey)
  def price       = column[Double]("PRICE")
  def acquisition = column[Date]  ("ACQUISITION")
  def * = (id, price, acquisition) <>
            (construct,extract)
}
val Devices = TableQuery[Devices]
```

7
Plain SQL support

Plain SQL support

```
import scala.slick.jdbc.{GetResult, StaticQuery}
import StaticQuery.interpolation

implicit val getDeviceResult =
  GetResult(r => Device(r.<<, r.<<, r.<<))

val price = 1000.0

val expensiveDevices: List[Device] =
  sql"select * from DEVICES where PRICE > $price"
    .as[Device].list
```

8
composable / re-usable queries

Composable, re-usable queries

```
def deviceLocations
 (companies: Query[Companies,Company])
 : Query[Column[String],String] = {
  companies.computers.devices.sites.map(_.location)
}

val apples = Companies.filter(_.name iLike "%apple%")
val locations : Seq[String] = {
  deviceLocations(apples)
    .filter(_.inAmerica: Column[String]=>Column[Boolean])
    .run
}
```

re-use joins

re-use queries

re-use user-defined operators

execute exactly one, precise query

Live Demo

Slick app design

Mental paradigm shift

Non-composable executor APIs (DAOs)

```
DevicesDAO
    .inPriceRange( 500.0, 2000.0 )
      : List[Device]
```

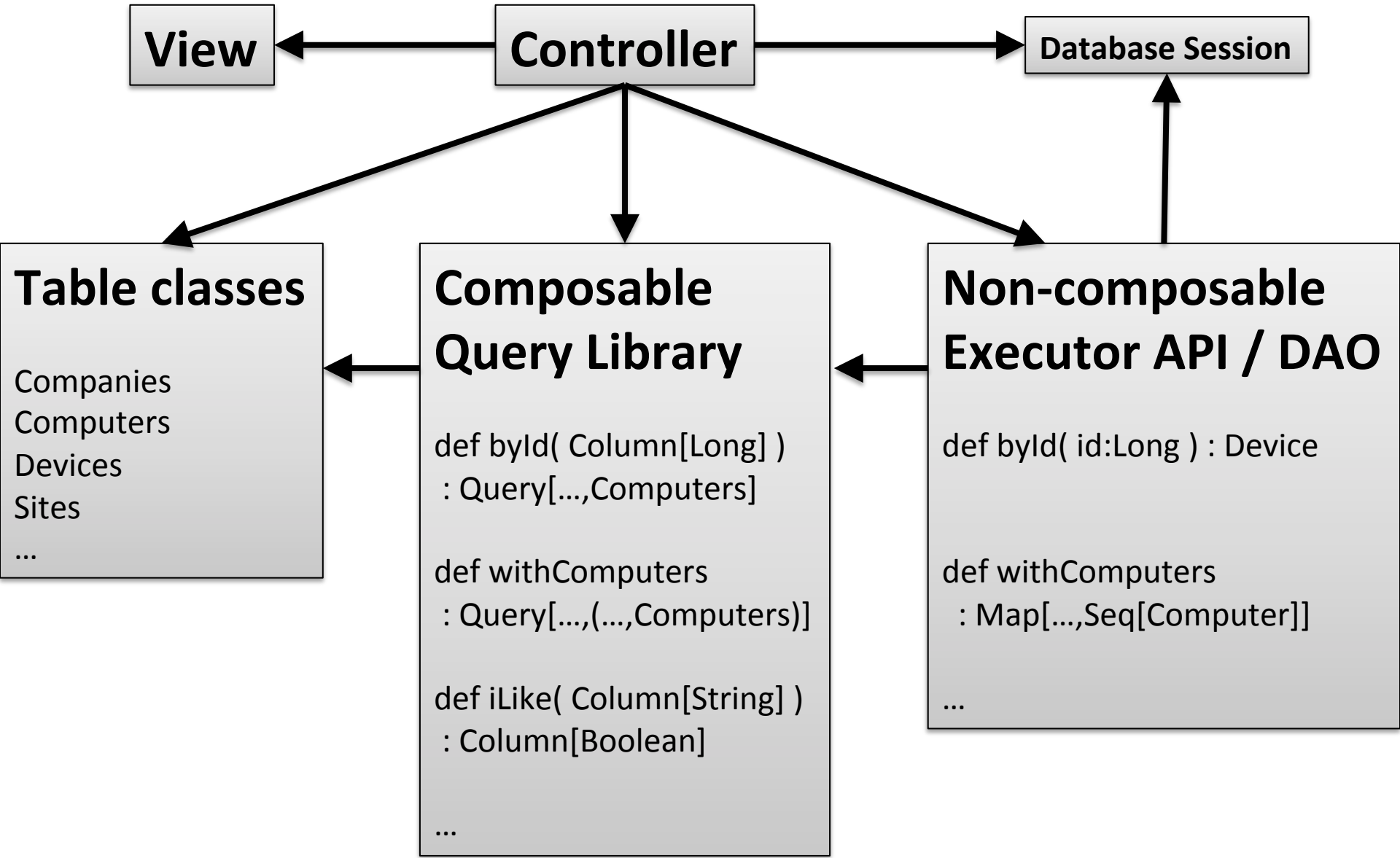
executes

Composable query libraries

```
devices
    .inPriceRange( 500.0, 2000.0 )
      : Query[_,Device]
```

composes

Suggested Slick app architecture



Relationships / Associations

- Via composable queries using foreign keys!

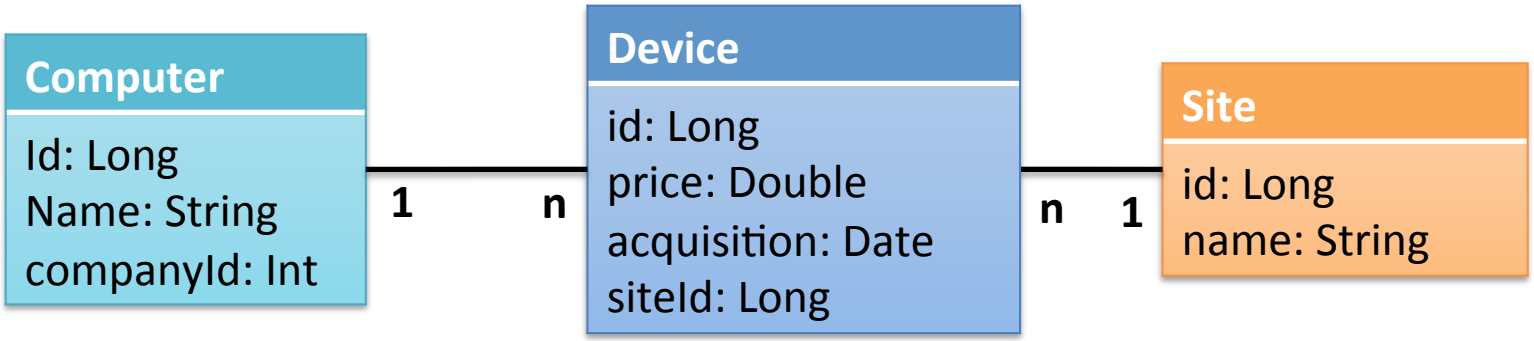
```
companies.withComputers
   : Query[…,(Company,Computer)]
```

- Not object references within query results
- Not executor APIs

Auto joins
(not in Slick, but easy to implement)

```
implicit def autojoin1 = joinCondition[Sites,Devices]
                                    (_.id === _.siteId)
implicit def autojoin2 = joinCondition[Devices,Computers]
                                  (_.computerId === _.id)


sites.autoJoin(devices).further(computers)
  : Query[_,(Site,Computer)]
sites.autoJoin(devices).autoJoinVia(computers)(_._2)
  : Query[_,((Site,Device),Computer)]
```



Other features

Other features

- inserts `+=` `++=`, updates `query.update(...)`
- user defined column types, e.g. type-safe ids
- user defined database functions
- …

Outlook

2.0 until end of 2013

- code-generation based type providers
- hlists and custom shapes (no 22-col limit, easy integration with shapeless, etc.)
- distributed queries (over multiple dbs)
- improved pre-compiled queries

Current experiments

- improved macro-based api (simpler types)
- macro-based type providers
- schema manipulation api
- migration/version management tool
- extended for-comprehensions (order, group)

Thanks to @amirsh @clhodapp @nafg

Thank you

Scala User Group
Berlin Brandenburg

slick.typesafe.com

@cvogt
@StefanZeiger

http://slick.typesafe.com/talks/
https://github.com/cvogt/slick-presentation/tree/2013/sug-berlin