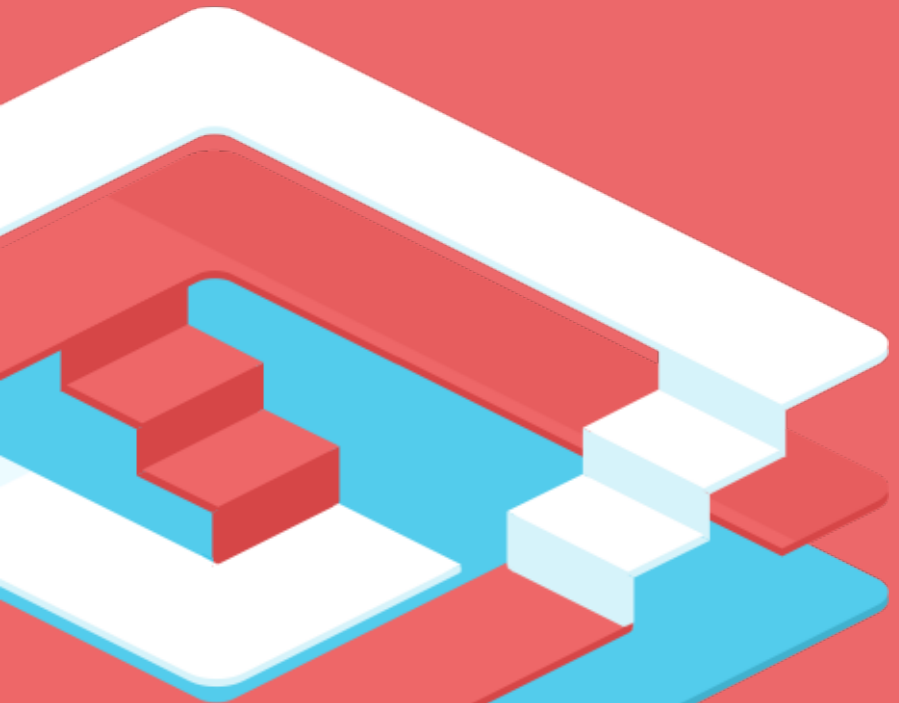
Slick vs ORM

Jan Christopher Vogt, EPFL
Stefan Zeiger, Typesafe



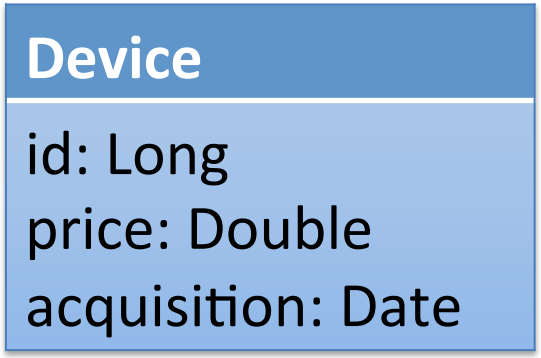
Object-Orientation + Relational

Functional + Relational

- Slick is a **Functional-Relational Mapper**
- embraces relational (not hidden)
- natural fit (no impedance mismatch)
- stateless (not stateful)
- Slick is to ORM what Scala is to Java

8 Reasons for using Slick

Sample App Data Model



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

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

2
Minimal configuration

Configuration

- Do mappings in Scala
- No XML, no "magic" behind the scenes
- Connect to a JDBC URL or DataSource
  - Use an external connection pool
- Wrap raw JDBC connections for use with DI containers that handle the transaction management

Connect

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

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

db.withSession { implicit s: Session =>
  …
}

db.withTransaction { implicit s: Session =>
  …
}
```

3
Loosely-coupled, flexible mapping

Keep Your Data Model Clean

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

```
class Devices extends Table[Device]("DEVICE") {
  def id = column[Long]("ID", O.PrimaryKey)
  def price = column[String]("PRICE")
  def acquisition = column[Date]("ACQUISITION")
  def * = id ~ price ~ acquisition <>
    (Device.apply _, Device.unapply _)
}
val Devices = new Devices
```

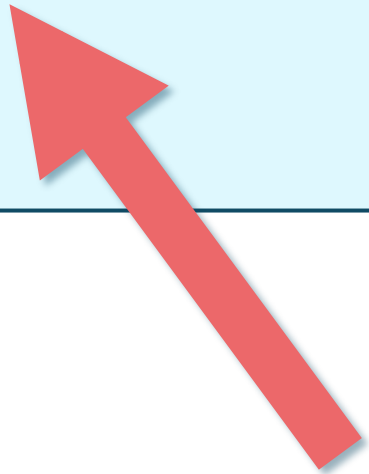
Keep Your Data Model Clean

...or omit it

```
class Devices extends Table[(Long, String, Date)]("DEVICE") {
  def id = column[Long]("ID", O.PrimaryKey)
  def price = column[String]("PRICE")
  def acquisition = column[Date]("ACQUISITION")
  def * = id ~ price ~ acquisition

}
val Devices = new Devices
```

Keep Your Data Model Clean

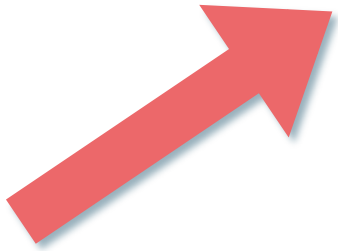
```
case class Device(id: Long,
  price: Double,
  acquisition: Date)
```



…but keep identity explicit

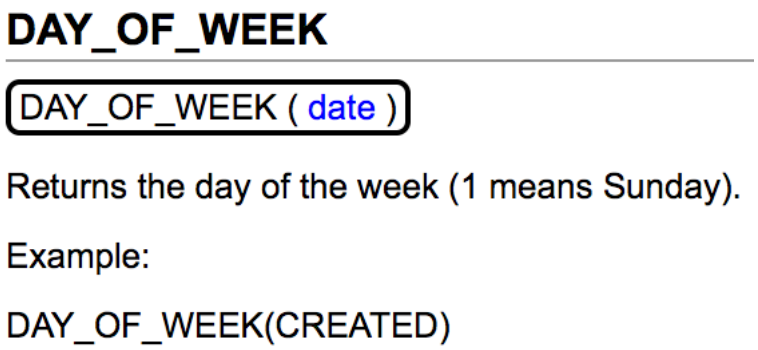
Custom Column Types

```
case class Device(id: DeviceId, …)
```



```
class DeviceId(val id: Long) extends AnyVal

implicit val deviceIdType = MappedTypeMapper.base
  [DeviceId, Long](_.id, new DeviceId(_))

class Devices extends Table[Device]("DEVICE") {
  def id = column[DeviceId]("ID", O.PrimaryKey)
  …
}
```

Custom Functions



DAY_OF_WEEK

DAY_OF_WEEK (date)

Returns the day of the week (1 means Sunday).

Example:

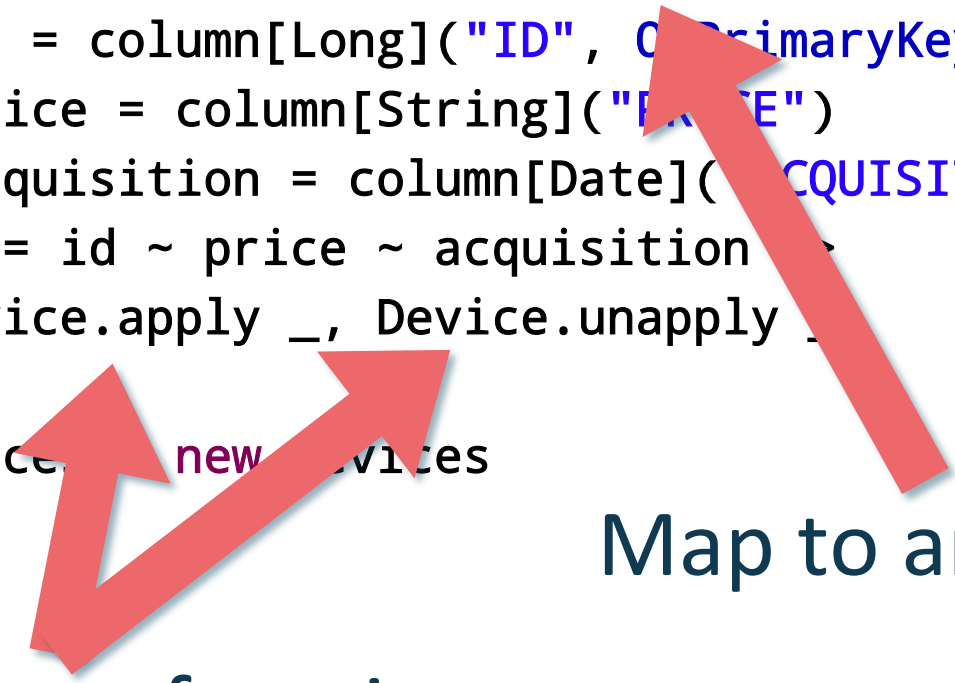
DAY_OF_WEEK(CREATED)

```
def dayOfWeek(c: Column[Date]) =
  SimpleFunction[Int]("DAY_OF_WEEK").apply(Seq(c))

val dows = Query(Devices).map { d =>
  (d.id, dayOfWeek(d.acquisition)) }.run
```

Work With Any DB Schema

- You define the schema:

```
class Devices extends Table[Device]("DEVICE") {
  def id = column[Long]("ID", O.PrimaryKey)
  def price = column[String]("PRICE")
  def acquisition = column[Date]("ACQUISITION")
  def * = id ~ price ~ acquisition <>
    (Device.apply _, Device.unapply _)
}
val Devices = new Devices
```



Map to anything

Use any function

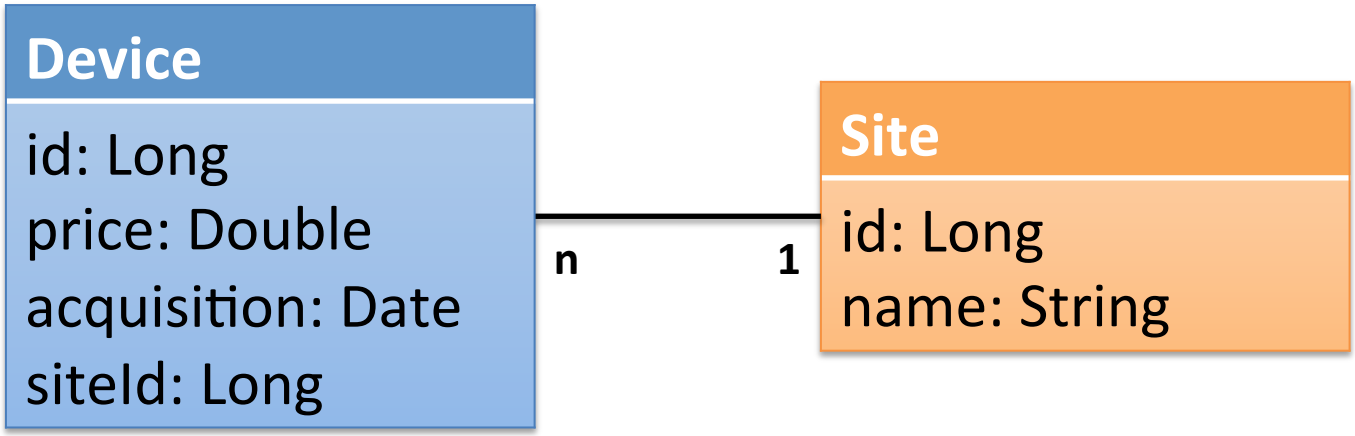
4
Explicit control over execution and transfer

Execution is always explicit

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

val results = query.run(session)
```

Device
id: Long
price: Double
acquisition: Date

Example Data Model

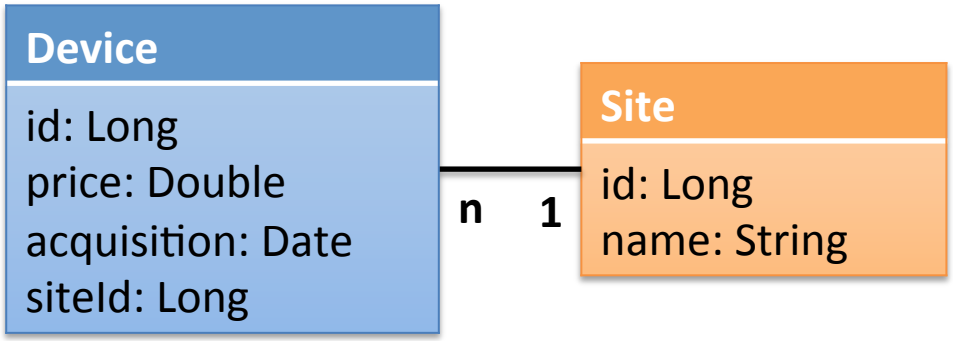


Transferred data

ORM

3 statements,
complete Device object loaded

```
val device  = Device.byId(123L) : Device
val site    = Site("New York")
device.site = site
ORM.save
```



Slick

2 statements,
nothing loaded,
device re-usable as query components

```
val device = Queries(Devices).byId(123L) : Query[Devices,Device]
val site   = Site(None,"New York")
val siteId = Sites.autoInc.insert( site )
device.map(_.siteId).update(siteId)
```

5
Predictable SQL structure

Predictable SQL structure

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

```
select x2."aquisition" from "DEVICE"
x2 where x2."price" > 1000.0
```

6
Plain SQL support

Plain SQL support

```
val price = 1000.0

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

Device

id: Long
price: Double
acquisition: Date

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

7

Type Safety

Enforce schema consistency

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

Compile-Time Safety

- Spelling mistake in column name?
- Wrong column type?
- Mapped to the wrong class?

scalac sees it all!

Compile-Time Safety

Piotr Buda @piotrbuda

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

12 hours ago

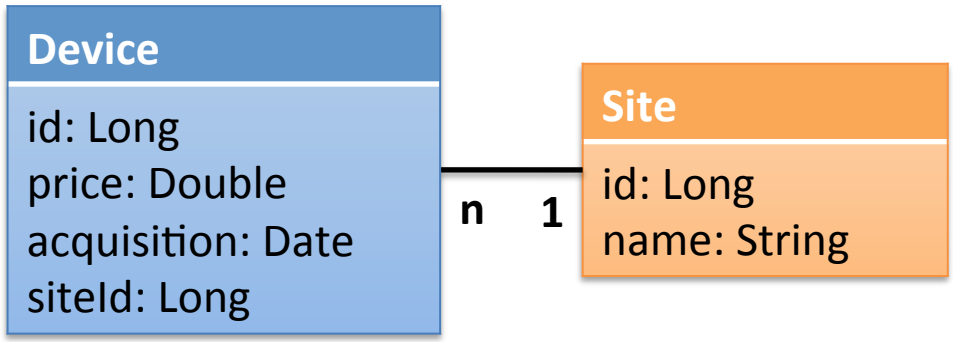
```
overloaded method value <> with alternatives:
  [R(in method <>)(in method <>)(in method <>)(in method <>)(in method <>)(in
<>)(in method <>)(in method <>)(in method <>)(in method <>), g: R(in method <
String)])scala.slick.lifted.MappedProjection[R(in method <>)(in method <>)(in
  [R(in method <>)(in 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 <>(
```

(before we get to #8)

Relationship Queries

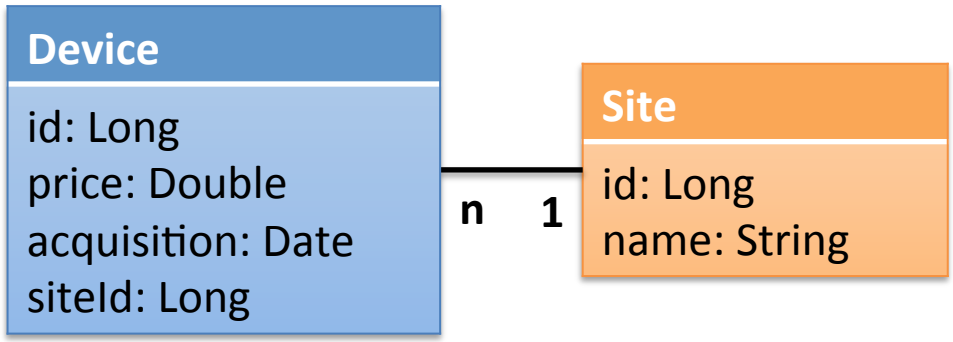
Relationships

- ORM
  - `device.getSite  : Site`
  - `site.getDevices : List[Device]`
- Slick
  - use relational queries with joins



Joins

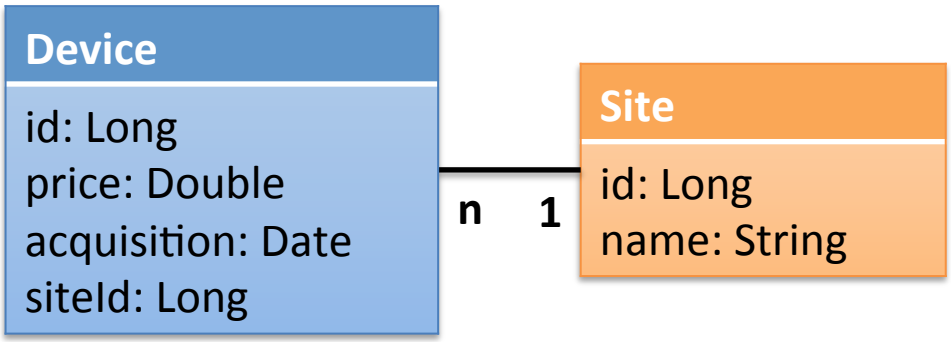
```
val sitesToDevices =
    (s:Sites,i:Devices) => s.id === i.siteId

val sites   = Query(Sites)  .filter(_.id === 1L)
val devices = Query(Devices).filter(_.price > 1000.0)

sites.join( devices ).on( sitesToDevices )
sites join devices on sitesToDevices
```



Auto joins (1-n)

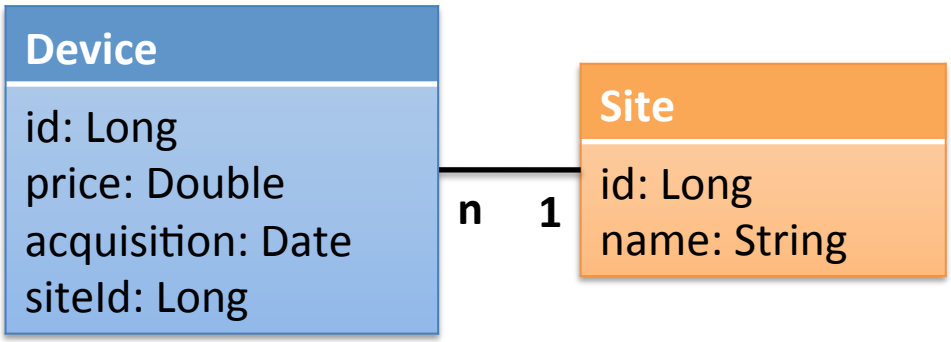
```
implicit def autojoin1 =
    joinCondition [Sites,Devices]
                                (_.id === _.siteId)
```

```
sites autoJoin devices
devices autoJoin sites
sites.autoJoin( devices, JoinType.Left )
```

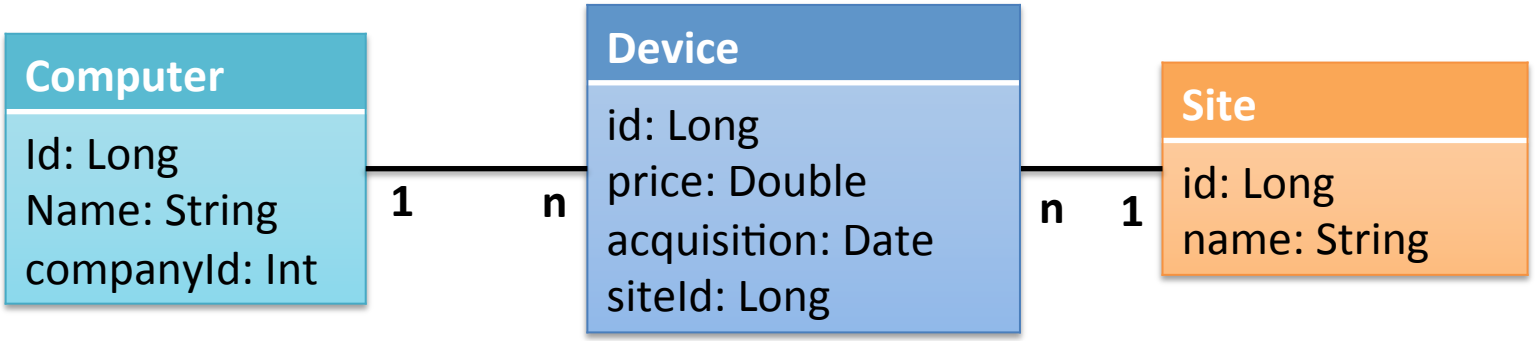


Join types

```
sites leftJoin  devices on sitesToDevices
sites rightJoin devices on sitesToDevices
sites outerJoin devices on sitesToDevices

sites.autoJoin( devices, JoinType.Left )
sites.autoJoin( devices, JoinType.Right )
sites.autoJoin( devices, JoinType.Outer )
```



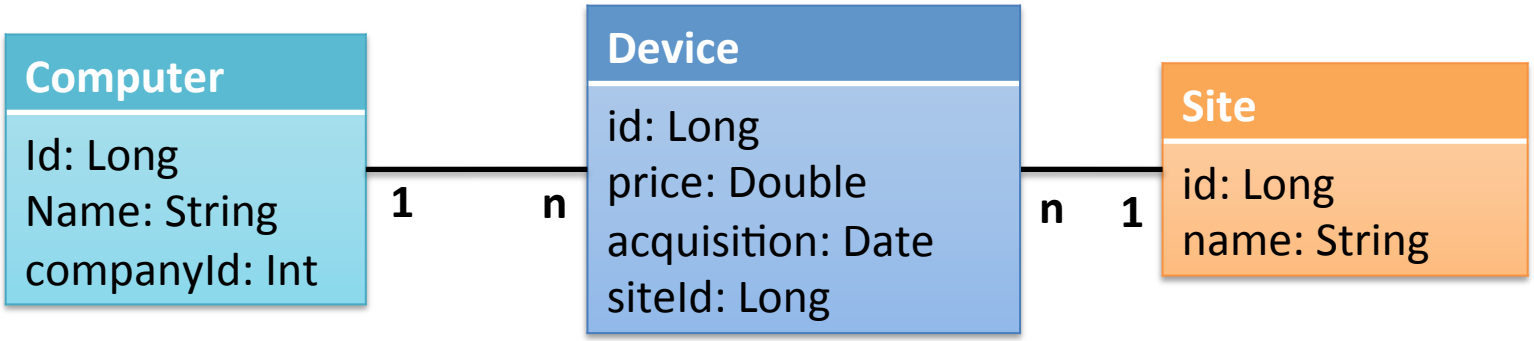
Example Data Model



Auto joins (n-n)

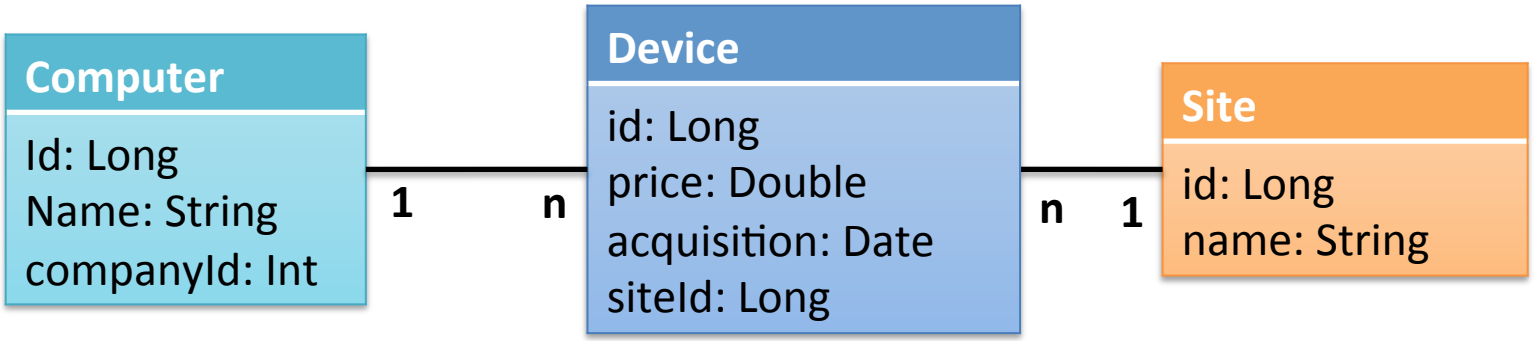
```
implicit def autojoin1 = joinCondition[Sites,Devices]
                                    (_.id === _.siteId)
implicit def autojoin2 = joinCondition[Devices,Computers]
                                  (_.computerId === _.id)
```

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



Complex Auto Joins (n-n, etc.)

```
implicit def autojoin4  =
  complexAutoJoin[Site,Computer,Sites,Computers]{
    case(sites,computers,joinType) =>
      val devices = Query(Devices)
      sites.autoJoin(devices,joinType).further(computers,joinType)
}

Sites.autoJoin(Computer) : Query[_,(Site,Computer)]
```



Modifying relationships

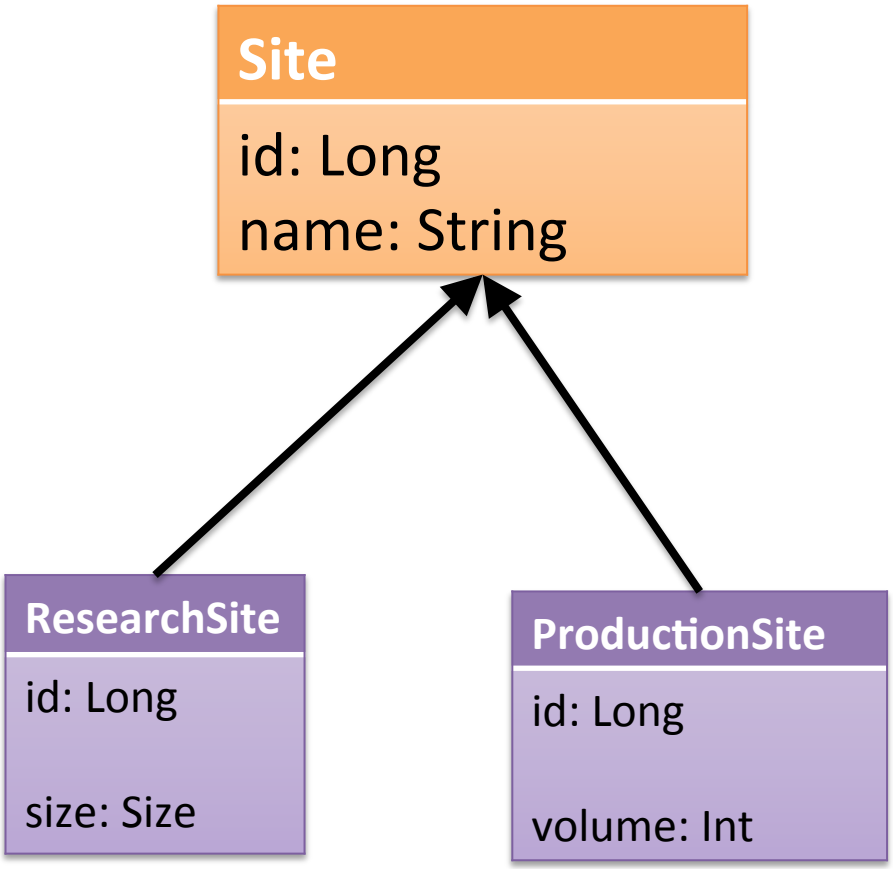
- query for what you want to modify
- insert / update that query

```
val device = Queries(Devices).byId(123L)
val site   = Site(None,"New York")
val siteId = Sites.autoInc.insert( site )
device.map(_.siteId).update(siteId)
```

Site
id: Long name: String

What about Inheritance?

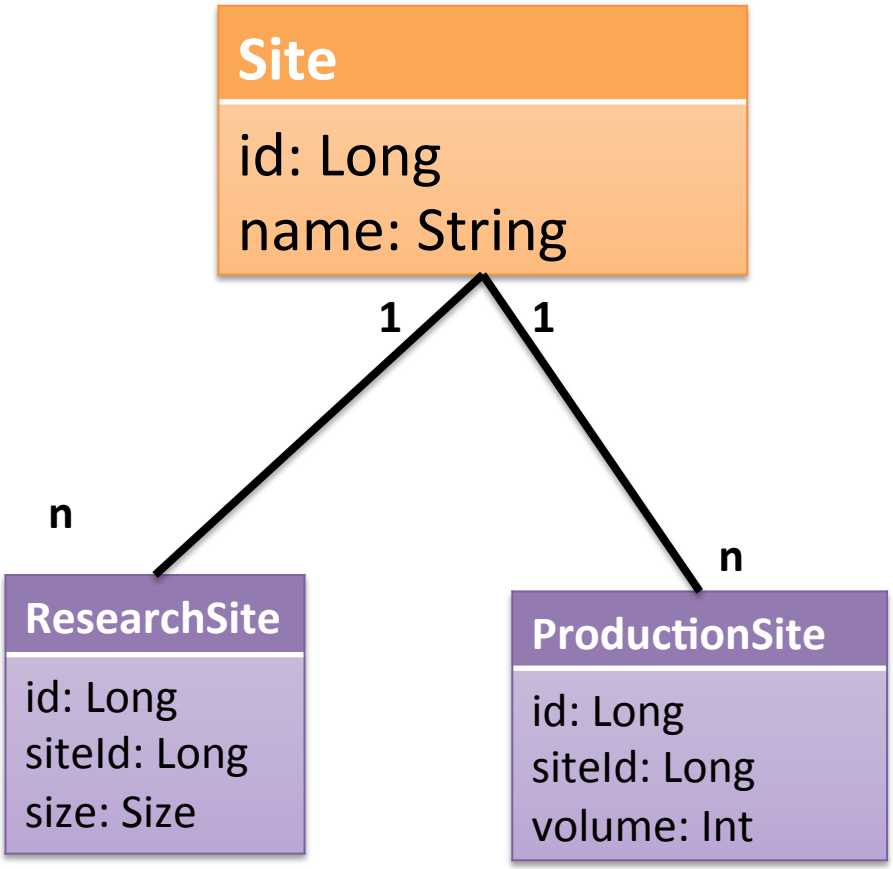
Example Data Model



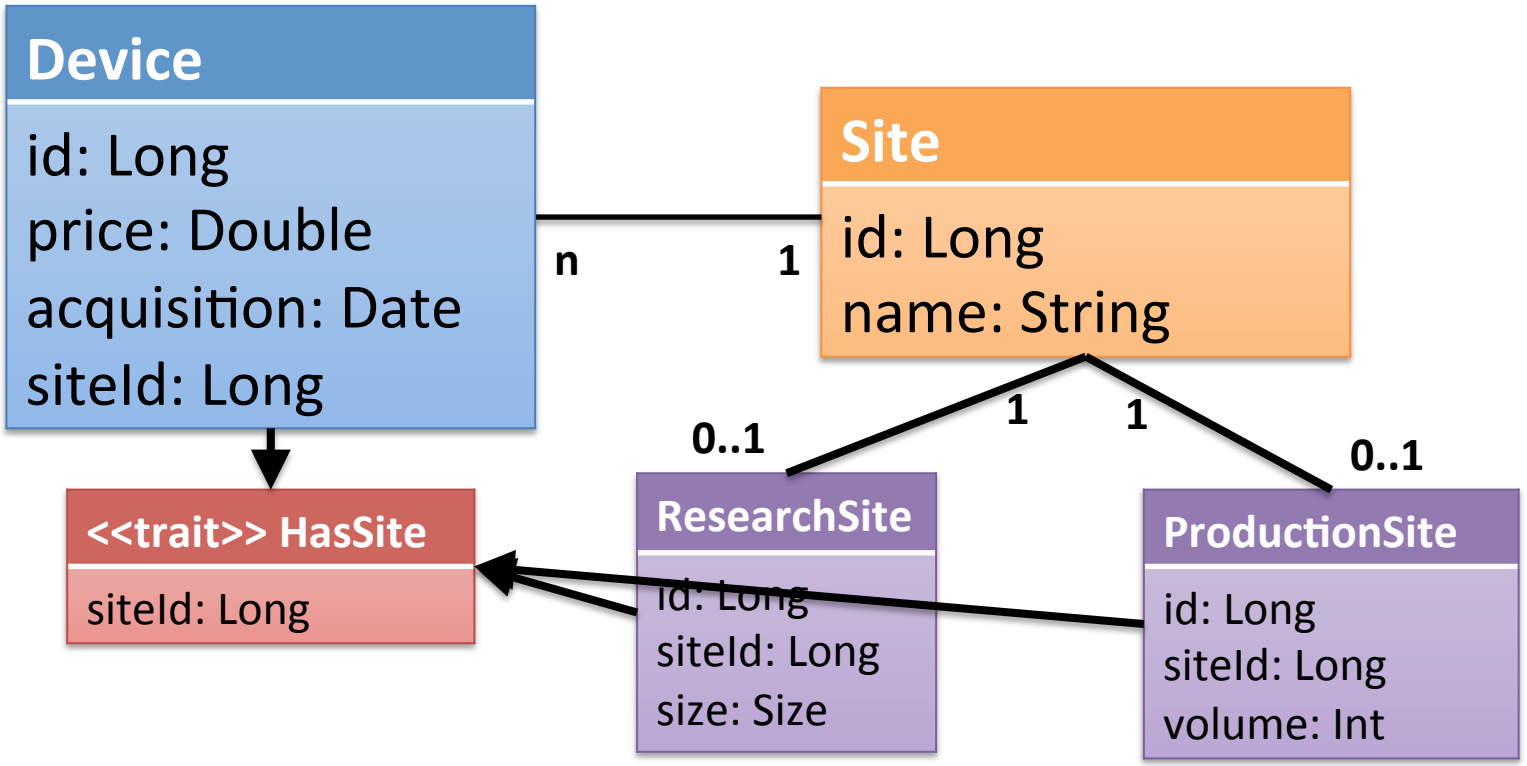
Inheritance

- Relational does not support inheritance
- Use relationships instead, you won’t loose (think “has role” instead of “is a”)

Example Data Model



Example Data Model



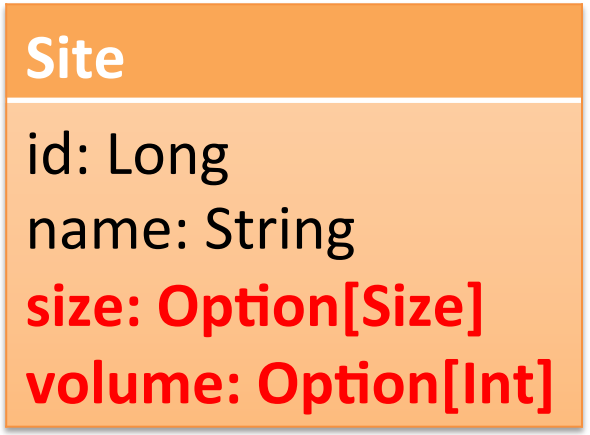

```
implicit def autojoin3
   = joinCondition[Sites,HasSite](_.id === _.siteId)
```

Joining all “sub-classes”

- a **re-usable query** that joins
  Sites with ResearchSites and ProductionSites

```
sites
 .autoJoin( researchSites, JoinType.Left )
 .autoJoinVia( productionSites,  JoinType.Left )(_._1)
 : Query[_,((Site,ResearchSite),ProductionSite)]
```

You can do inheritance




```
class Sites extends Table[Site]{
  ...
  def * = ...
  <> (
   {case (id,name,Some(size),_)   => ResearchSite(id,name,size)
    case (id,name,_,Some(volume)) => ProductionSite(id,name,volume) },
   {case ResearchSite(id,name,size)     => (id,name,Some(size),None)
    case ProductionSite(id,name,volume) => (id,name,None,Some(volume)) }
  )
}
```

8
composable / re-usable queries

Mental paradigm shift

The ORM Way: Executor APIs (DAOs)

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

The Slick Way: Query libraries

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

some stuff with criteria queries, but Slick for everything including joins, groupBy

Write query libraries

Row functions

```
class Sites extends Table[Site]{
  def name = column( ... )
  ...
  def nameLike( pattern:Column[String] ) : Column[Boolean]
        = name.toLowerCase like pattern.toLowerCase
}


Query(Sites).filter( _.nameLike(“EPFL”) )
```

Row functions

```
trait HasName{
  this:Table[_] =>
  def name = column( ... )
  ...
  def nameLike( pattern:Column[String] ) : Column[Boolean]
        = this.name.toLowerCase like pattern.toLowerCase
}

class Sites     extends Table[Site] with HasName
class Computers extends Table[Computer] with HasName
```

Query functions

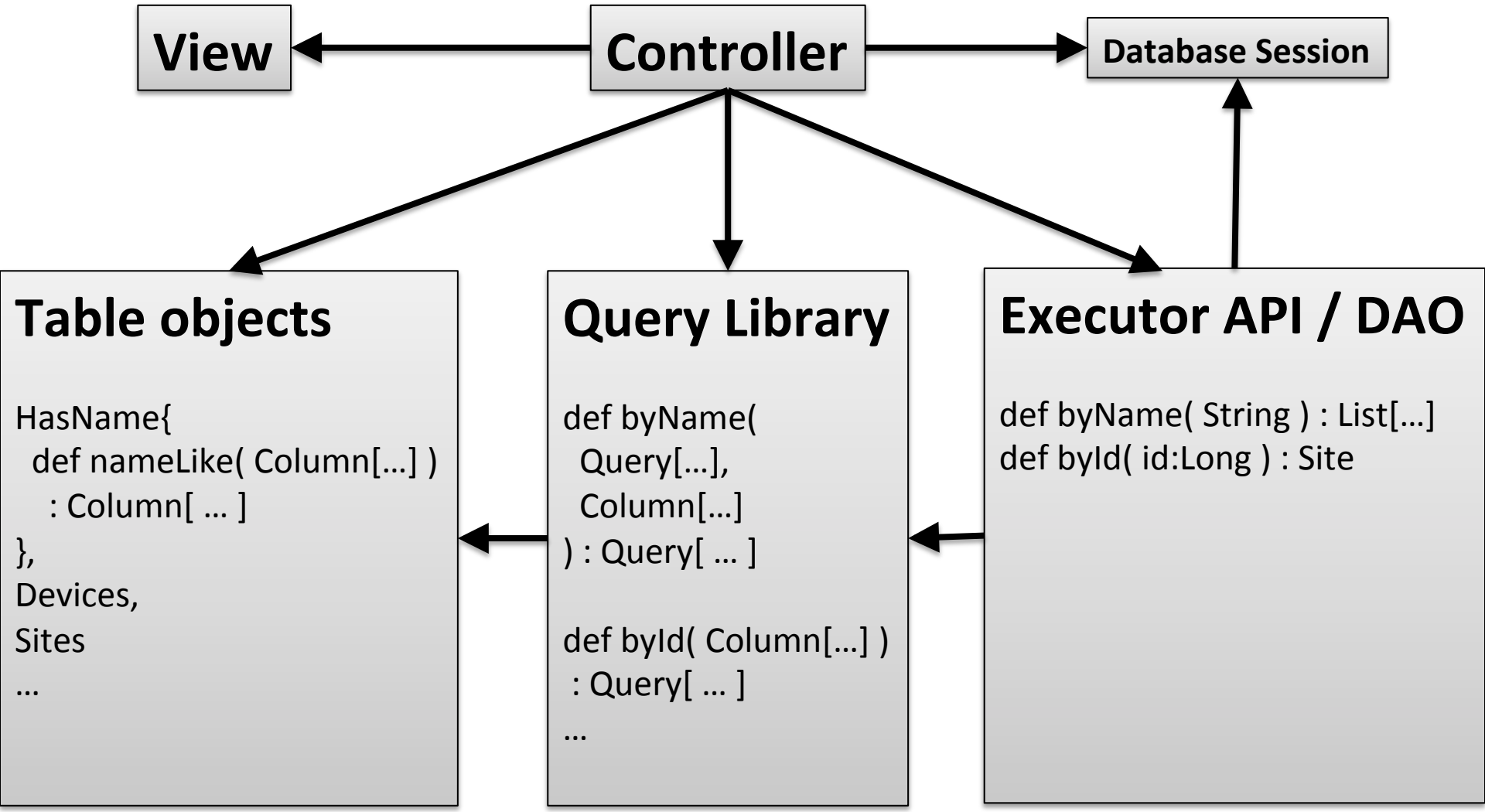
```
def byName[E,T <: Table[E] with HasName]
          ( q:Query[T,E], pattern:Column[String] )
          = q.filter( _nameLike.(pattern) )

byName( Query(Sites), “EPFL” )
```

Refinements

- method extensions (implicit conversions)
- Option support using Slick’s OptionMapper

Suggested slick app architecture

Suggested Slick app architecture

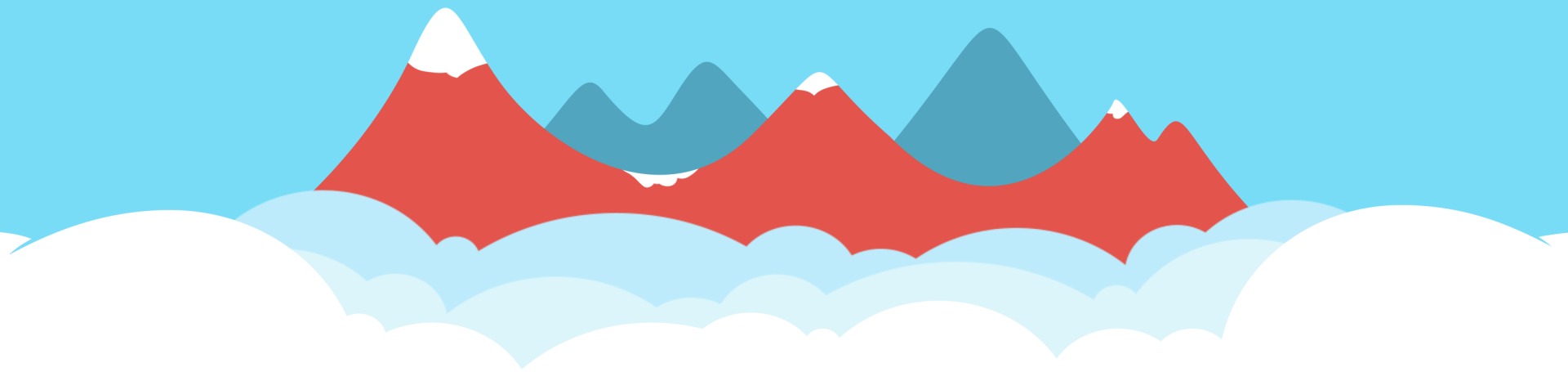


Outlook

Slick 2.0

- Coming Q3 / 2013
- Query scheduling
- Improved driver architecture
  - BasicProfile >: RelationalProfile >: SqlProfile >: JdbcProfile
- Generate Slick code from database schemas
  - Proper "type providers" need new Scala
- More to come

Scaladays JUNE 10TH-12TH



slick.typesafe.com

@cvogt
@StefanZeiger

http://slick.typesafe.com/talks/2013_scaladays/2013_scaladays.pdf
https://github.com/slick/play-slick/tree/scaladays2013

Extra slides

Join via foreign key

```
table Devices{
  ...
  def site = foreignKey("fk_site",Sites,_.siteId)(_.id)
}

for( d <- Devices; s <- d.site ) yield (i,s)
   : Query[_,(Device,Site)]
```