SLICK 3.1

For Scala Developer

DECEMBER 13, 2015

CONSOLIDIATE BY SCOTT HUANG
Personal Hobby

Contents Index

INTRODUCTION

What is Slick?

Slick ("Scala Language-Integrated Connection Kit") is Typesafe's Functional Relational Mapping (FRM) library for Scala that makes it easy to work with relational databases. It allows you to work with stored data almost as if you were using Scala collections while at the same time giving you full control over when a database access happens and which data is transferred. You can also use SQL directly. Execution of database actions is done asynchronously, making Slick a perfect fit for your reactive applications based on Play and Akka.

```
val limit = 10.0

// Your query could look like this:
( for( c <- coffees; if c.price < limit ) yield c.name ).result

// Equivalent SQL: select COF_NAME from COFFEES where PRICE < 10.0
```

When using Scala instead of raw SQL for your queries you benefit from compile-time safety and compositionality. Slick can generate queries for different back-end databases including your own, using its extensible query compiler.

Get started learning Slick in minutes using the Hello Slick template in Typesafe Activator. See here for an overview of the supported database systems for which Slick can generate code.

Functional Relational Mapping

Functional programmers have long suffered Object-Relational and Object-Math impedance mismatches when connecting to relational databases. Slick's new Functional Relational Mapping (FRM) paradigm allows mapping to be completed within Scala, with loose-coupling, minimal configuration requirements, and a number of other major advantages that abstract the complexities away from connecting with relational databases.

We don't try to fight the relational model, we embrace it through a functional paradigm. Instead of trying to bridge the gap between the object model and the database model, we've brought the database model into Scala so developers don't need to write SQL code.

```
class Coffees(tag: Tag) extends Table[(String, Double)](tag, "COFFEES") {
  def name = column[String]("COF_NAME", O.PrimaryKey)
```

```
  def price = column[Double]("PRICE")
  def * = (name, price)
}
val coffees = TableQuery[Coffees]
```

Slick integrates databases directly into Scala, allowing stored and remote data to be queried and processed in the same way as in-memory data, using ordinary Scala classes and collections.

```
// Query that only returns the "name" column
// Equivalent SQL: select NAME from COFFEES
coffees.map(_.name)

// Query that limits results by price < 10.0
// Equivalent SQL: select * from COFFEES where PRICE < 10.0
coffees.filter(_.price < 10.0)
```

This enables full control over when a database is accessed and which data is transferred. The language integrated query model in Slick's FRM is inspired by the LINQ project at Microsoft and leverages concepts tracing all the way back to the early work of Mnesia at Ericsson.

Some of the key benefits of Slick's FRM approach for functional programming include:

- Efficiency with Pre-Optimization

FRM is more efficient way to connect; unlike ORM it has the ability to pre-optimize its communication with the database - and with FRM you get this out of the box. The road to making an app faster is much shorter with FRM than ORM.

- No More Tedious Troubleshooting with Type Safety

FRM brings type safety to building database queries. Developers are more productive because the compiler finds errors automatically versus the typical tedious troubleshooting required of finding errors in untyped strings.

```
// The result of "select PRICE from COFFEES" is a Seq of Double
// because of the type safe column definitions
val coffeeNames: Future[Seq[Double]] = db.run(
  coffees.map(_.price).result
)

// Query builders are type safe:
coffees.filter(_.price < 10.0)
// Using a string in the filter would result in a compilation error
```

Misspelled the column name `price`? The compiler will tell you:

```
GettingStartedOverview.scala:89:  value  prices  is  not  a  member  of
com.typesafe.slick.docs.GettingStartedOverview.Coffees
```

```
        coffees.map(_.prices).result
                      ^
```

The same goes for type errors:

```
GettingStartedOverview.scala:89: type mismatch;
 found                                                                  :
slick.driver.H2Driver.StreamingDriverAction[Seq[String],String,slick.dbio.Effe
ct.Read]
    (which                                expands                                to)
slick.profile.FixedSqlStreamingAction[Seq[String],String,slick.dbio.Effect.Rea
d]
 required: slick.dbio.DBIOAction[Seq[Double],slick.dbio.NoStream,Nothing]
        coffees.map(_.name).result
                            ^
```

- A More Productive, Composable Model for Building Queries

FRM supports a composable model for building queries. It's a very natural model to compose pieces together to build a query, and then reuse pieces across your code base.

```
// Create a query for coffee names with a price less than 10, sorted by name
coffees.filter(_.price < 10.0).sortBy(_.name).map(_.name)
// The generated SQL is equivalent to:
// select name from COFFEES where PRICE < 10.0 order by NAME
```

Reactive Applications

Slick is easy to use in asynchronous, non-blocking application designs, and supports building applications according to the Reactive Manifesto. Unlike simple wrappers around traditional, blocking database APIs, Slick gives you:

- Clean separation of I/O and CPU-intensive code: Isolating I/O allows you to keep your main thread pool busy with CPU-intensive parts of the application while waiting for I/O in the background.
- Resilience under load: When a database cannot keep up with the load of your application, Slick will not create more and more threads (thus making the situation worse) or lock out all kinds of I/O. Back-pressure is controlled efficiently through a queue (of configurable size) for database I/O actions, allowing a certain number of requests to build up with very little resource usage and failing immediately once this limit has been reached.
- Reactive Streams for asynchronous streaming.
- Efficient utilization of database resources: Slick can be tuned easily and precisely for the parallelism (number of concurrent active jobs) and resource ussage (number of currently suspended database sessions) of your database server.

Plain SQL Support

The Scala-based query API for Slick allows you to write database queries like queries for Scala collections. Please see Getting Started for an introduction. Most of this user manual focuses on this API.
If you want to write your own SQL statements and still execute them asynchronously like a normal Slick queries, you can use the Plain SQL API:

```
val limit = 10.0

sql"select COF_NAME from COFFEES where PRICE < $limit".as[String]

// Automatically using a bind variable to be safe from SQL injection:
// select COF_NAME from COFFEES where PRICE < ?
```

License

Slick is released under a BSD-Style free and open source software license. See the chapter on the commercialSlick Extensions add-on package for details on licensing the Slick drivers for the big commercial database systems.

Next Steps

- If you are new to Slick, continue to Getting Started
- If you have used an older version of Slick, make sure to read the Upgrade Guides
- If you have used an ORM before, see Coming from ORM to Slick
- If you are familiar with SQL, see Coming from SQL to Slick

SUPPORTED DATABASES

- DB2 (via slick-extensions)
- Derby/JavaDB
- H2
- HSQLDB/HyperSQL
- Microsoft SQL Server (via slick-extensions)
- MySQL
- Oracle (via slick-extensions)
- PostgreSQL
- SQLite

Other SQL databases can be accessed right away with a reduced feature set. Writing a fully featured plugin for your own SQL-based backend can be achieved with a reasonable amount of work. Support for other backends (like NoSQL) is under development but not yet available.

The following capabilities are supported by the drivers. “Yes” means that a capability is fully supported. In other cases it may be partially supported or not at all. See the individual driver’s API documentation for details.

Driver Capabilities (core drivers only)						
Capability	Derby Driver	H2Driver	Hsqldb Driver	MySQL Driver	Postgres Driver	SQLite Driver
relational.other	Yes	Yes	Yes	Yes	Yes	Yes
relational.columnDefaults	Yes	Yes	Yes	Yes	Yes	Yes
relational.foreignKeyActions	Yes	Yes	Yes	Yes	Yes	Yes
relational.functionDatabase		Yes	Yes	Yes	Yes	
relational.functionUser	Yes	Yes	Yes	Yes	Yes	
relational.indexOf	Yes	Yes	Yes	Yes	Yes	Yes
relational.joinFull			Yes		Yes	
relational.joinLeft	Yes	Yes	Yes	Yes	Yes	Yes
relational.joinRight	Yes	Yes	Yes	Yes	Yes	
relational.likeEscape	Yes	Yes	Yes	Yes	Yes	Yes
relational.pagingDrop	Yes	Yes	Yes	Yes	Yes	Yes
relational.pagingNested		Yes	Yes	Yes	Yes	Yes
relational.pagingPreciseTake	Yes	Yes	Yes	Yes	Yes	Yes
relational.repeat		Yes	Yes	Yes	Yes	Yes
relational.replace		Yes	Yes	Yes	Yes	Yes
relational.reverse			Yes	Yes	Yes	Yes

Driver Capabilities (core drivers only)						
Capability	Derby Driver	H2Driver	Hsqldb Driver	MySQL Driver	Postgres Driver	SQLite Driver
relational.setByteArrayNull	Yes	Yes	Yes	Yes	Yes	Yes
relational.typeBigDecimal	Yes	Yes	Yes	Yes	Yes	
relational.typeBlob	Yes	Yes	Yes	Yes	Yes	
relational.typeLong	Yes	Yes	Yes	Yes	Yes	Yes
relational.zip		Yes	Yes	Yes	Yes	
sql.other	Yes	Yes	Yes	Yes	Yes	Yes
sql.sequence	Yes	Yes	Yes	Yes	Yes	
sql.sequenceCurr		Yes		Yes	Yes	Yes
sql.sequenceCycle			Yes	Yes	Yes	Yes
sql.sequenceLimited	Yes	Yes	Yes		Yes	Yes
sql.sequenceMax	Yes		Yes	Yes	Yes	Yes
sql.sequenceMin	Yes		Yes	Yes	Yes	Yes
jdbc.other	Yes	Yes	Yes	Yes	Yes	Yes
jdbc.booleanMetaData		Yes	Yes	Yes	Yes	
jdbc.createModel	Yes	Yes	Yes	Yes	Yes	Yes
jdbc.defaultValueMetaData	Yes	Yes	Yes	Yes	Yes	
jdbc.distinguishesIntTypes	Yes	Yes	Yes	Yes	Yes	
jdbc.forceInsert	Yes	Yes	Yes	Yes	Yes	Yes

Driver Capabilities (core drivers only)						
Capability	Derby Driver	H2Driver	Hsqldb Driver	MySQL Driver	Postgres Driver	SQLite Driver
jdbc.insertOrUpdate				Yes		
jdbc.mutable	Yes	Yes	Yes	Yes	Yes	
jdbc.nullableNoDefault	Yes	Yes	Yes			Yes
jdbc.returnInsertKey	Yes	Yes	Yes	Yes	Yes	Yes
jdbc.returnInsertOther			Yes		Yes	
jdbc.supportsByte		Yes	Yes	Yes		

GETTING STARTED

The easiest way to get started is with a working application in Typesafe Activator. The following templates are created by the Slick team, with updated versions being made for new Slick releases:

- To learn the basics of Slick, start with the Hello Slick template. It contains an extended version of the tutorial and code from this page.
- The Slick Plain SQL Queries template shows you how to do SQL queries with Slick.
- The Slick Multi-DB Patterns template shows you how to write Slick applications that can use different database systems and how to use custom database functions in Slick queries.
- The Slick TestKit Example template shows you how to use Slick TestKit to test your own Slick drivers.

There are more Slick templates created by the community, as well as versions of our own templates for other Slick releases. You can find all Slick templates on the Typesafe web site.

Adding Slick to Your Project

To include Slick in an existing project use the library published on Maven Central. For sbt projects add the following to your build definition – `build.sbt` or `project/Build.scala`:

```
libraryDependencies ++= Seq(
```

```
  "com.typesafe.slick" %% "slick" % "3.1.0",
  "org.slf4j" % "slf4j-nop" % "1.6.4"
)
```

For Maven projects add the following to your `<dependencies>` (make sure to use the correct Scala version prefix, `_2.10` or `_2.11`, to match your project's Scala version):

```
<dependency>
  <groupId>com.typesafe.slick</groupId>
  <artifactId>slick_2.10</artifactId>
  <version>3.1.0</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-nop</artifactId>
  <version>1.6.4</version>
</dependency>
```

Slick uses SLF4J for its own debug logging so you also need to add an SLF4J implementation. Here we are using `slf4j-nop` to disable logging. You have to replace this with a real logging framework like Logback if you want to see log output.
The Reactive Streams API is pulled in automatically as a transitive dependency.

If you want to use Slick's connection pool support for HikariCP, you need to add the `slick-hikaricp` module as a dependency in the same way as shown for `slick` above. It will automatically provide a compatible version of HikariCP as a transitive dependency.

Quick Introduction

Note

The rest of this chapter is based on the Hello Slick template. The prefered way of reading this introduction is in Activator, where you can edit and run the code directly while reading the tutorial.
To use Slick you first need to import the API for the database you will be using, like:

```
// Use H2Driver to connect to an H2 database
import slick.driver.H2Driver.api._
```

```
import scala.concurrent.ExecutionContext.Implicits.global
```

Since we are using H2 as our database system, we need to import features from Slick's `H2Driver`. A driver's `api` object contains all commonly needed imports from the driver and other parts of Slick such as database handling.

Slick's API is fully asynchronous and runs database call in a separate thread pool. For running user code in composition of `DBIOAction` and `Future` values, we import the global `ExecutionContext`. When using Slick as part of a larger application (e.g. with Play or Akka) the framework may provide a better alternative to this default `ExecutionContext`.

Database Configuration

In the body of the application we create a `Database` object which specifies how to connect to a database. In most cases you will want to configure database connections with Typesafe Config in your `application.conf`, which is also used by Play and Akka for their configuration:

```
h2mem1 = {
  url = "jdbc:h2:mem:test1"
  driver = org.h2.Driver
  connectionPool = disabled
  keepAliveConnection = true
}
```

For the purpose of this example we disable the connection pool (there is no point in using one for an embedded in-memory database) and request a keep-alive connection (which ensures that the database does not get dropped while we are using it). The database can be easily instantiated from the configuration like this:

```
val db = Database.forConfig("h2mem1")
try {
  // ...
} finally db.close
```

Note

A `Database` object usually manages a thread pool and a connection pool. You should always shut it down properly when it is no longer needed (unless the JVM process terminates anyway).

Schema

Before we can write Slick queries, we need to describe a database schema with `Table` row classes and `TableQuery` values for our tables. You can either use the code generator to automatically create them for your database schema or you can write them by hand:

```
// Definition of the SUPPLIERS table
```

```
class Suppliers(tag: Tag) extends Table[(Int, String, String, String, String,
String)](tag, "SUPPLIERS") {
  def id = column[Int]("SUP_ID", O.PrimaryKey) // This is the primary key column
  def name = column[String]("SUP_NAME")
  def street = column[String]("STREET")
  def city = column[String]("CITY")
  def state = column[String]("STATE")
  def zip = column[String]("ZIP")
  // Every table needs a * projection with the same type as the table's type
parameter
  def * = (id, name, street, city, state, zip)
}
val suppliers = TableQuery[Suppliers]

// Definition of the COFFEES table
class Coffees(tag: Tag) extends Table[(String, Int, Double, Int, Int)](tag,
"COFFEES") {
  def name = column[String]("COF_NAME", O.PrimaryKey)
  def supID = column[Int]("SUP_ID")
  def price = column[Double]("PRICE")
  def sales = column[Int]("SALES")
  def total = column[Int]("TOTAL")
  def * = (name, supID, price, sales, total)
  // A reified foreign key relation that can be navigated to create a join
  def supplier = foreignKey("SUP_FK", supID, suppliers)(_.id)
}
val coffees = TableQuery[Coffees]
```

All columns get a name (usually in camel case for Scala and upper case with underscores for SQL) and a Scala type (from which the SQL type can be derived automatically). The table object also needs a Scala name, SQL name and type. The type argument of the table must match the type of the special `*` projection. In simple cases this is a tuple of all columns but more complex mappings are possible.

The `foreignKey` definition in the `coffees` table ensures that the `supID` field can only contain values for which a corresponding `id` exists in the `suppliers` table, thus creating an *n to one* relationship: A `Coffees`row points to exactly one `Suppliers` row but any number of coffees can point to the same supplier. This constraint is enforced at the database level.

Populating the Database

The connection to the embedded H2 database engine provides us with an empty database. Before we can execute queries, we need to create the database schema (consisting of the `coffees` and `suppliers` tables) and insert some test data:

```
val setup = DBIO.seq(
  // Create the tables, including primary and foreign keys
  (suppliers.schema ++ coffees.schema).create,
```

```
  // Insert some suppliers
  suppliers += (101, "Acme, Inc.",       "99 Market Street", "Groundsville",
"CA", "95199"),
  suppliers += ( 49, "Superior Coffee", "1 Party Place",    "Mendocino",    "CA",
"95460"),
  suppliers += (150, "The High Ground", "100 Coffee Lane",  "Meadows",      "CA",
"93966"),
  // Equivalent SQL code:
  // insert into SUPPLIERS(SUP_ID, SUP_NAME, STREET, CITY, STATE, ZIP) values
(?,?,?,?,?,?)

  // Insert some coffees (using JDBC's batch insert feature, if supported by
the DB)
  coffees ++= Seq(
    ("Colombian",         101, 7.99, 0, 0),
    ("French_Roast",       49, 8.99, 0, 0),
    ("Espresso",          150, 9.99, 0, 0),
    ("Colombian_Decaf",   101, 8.99, 0, 0),
    ("French_Roast_Decaf", 49, 9.99, 0, 0)
  )
  // Equivalent SQL code:
  //  insert  into  COFFEES(COF_NAME,  SUP_ID,  PRICE,  SALES,  TOTAL)  values
(?,?,?,?,?)
)

val setupFuture = db.run(setup)
```

The `TableQuery`'s `ddl` method creates `DDL` (data definition language) objects with the database-specific code for creating and dropping tables and other database entities. Multiple `DDL` values can be combined with `++` to allow all entities to be created and dropped in the correct order, even when they have circular dependencies on each other. Inserting the tuples of data is done with the `+=` and `++=` methods, similar to how you add data to mutable Scala collections.

The `create`, `+=` and `++=` methods return an `Action` which can be executed on a database at a later time to produce a result. There are several different combinators for combining multiple Actions into sequences, yielding another Action. Here we use the simplest one, `Action.seq`, which can concatenate any number of Actions, discarding the return values (i.e. the resulting Action produces a result of type `Unit`). We then execute the setup Action asynchronously with `db.run`, yielding a `Future[Unit]`.

Note

Database connections and transactions are managed automatically by Slick. By default connections are acquired and released on demand and used in *auto-commit* mode. In this mode we have to populate the `suppliers` table first because the `coffees` data can only refer to valid supplier IDs. We could also use an explicit transaction bracket encompassing all these statements (`db.run(setup.transactionally)`). Then the order

would not matter because the constraints are only enforced at the end when the transaction is committed.

Querying

The simplest kind of query iterates over all the data in a table:

```
// Read all coffees and print them to the console
println("Coffees:")
db.run(coffees.result).map(_.foreach {
  case (name, supID, price, sales, total) =>
    println("  " + name + "\t" + supID + "\t" + price + "\t" + sales + "\t" +
total)
})
// Equivalent SQL code:
// select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES
```

This corresponds to a `SELECT * FROM COFFEES` in SQL (except that the `*` is the table's `*` projection we defined earlier and not whatever the database sees as `*`). The type of the values we get in the loop is, unsurprisingly, the type parameter of `Coffees`.

Let's add a *projection* to this basic query. This is written in Scala with the `map` method or a *for comprehension*:

```
// Why not let the database do the string conversion and concatenation?
val q1 = for(c <- coffees)
  yield LiteralColumn("  ") ++ c.name ++ "\t" ++ c.supID.asColumnOf[String] ++
    "\t" ++ c.price.asColumnOf[String] ++ "\t" ++ c.sales.asColumnOf[String] ++
    "\t" ++ c.total.asColumnOf[String]
// The first string constant needs to be lifted manually to a LiteralColumn
// so that the proper ++ operator is found

// Equivalent SQL code:
// select '  ' || COF_NAME || '\t' || SUP_ID || '\t' || PRICE || '\t' SALES ||
'\t' TOTAL from COFFEES

db.stream(q1.result).foreach(println)
```

The output will be the same: For each row of the table, all columns get converted to strings and concatenated into one tab-separated string. The difference is that all of this now happens inside the database engine, and only the resulting concatenated string is shipped to the client. Note that we avoid Scala's `+` operator (which is already heavily overloaded) in favor of `++` (commonly used for sequence concatenation). Also, there is no automatic conversion of other argument types to strings. This has to be done explicitly with the type conversion method `asColumnOf`.

This time we also use Reactive Streams to get a streaming result from the database and print the elements as they come in instead of materializing the whole result set upfront. Joining and filtering tables is done the same way as when working with Scala collections:

```
// Perform a join to retrieve coffee names and supplier names for
// all coffees costing less than $9.00
val q2 = for {
  c <- coffees if c.price < 9.0
  s <- suppliers if s.id === c.supID
} yield (c.name, s.name)
// Equivalent SQL code:
// select c.COF_NAME, s.SUP_NAME from COFFEES c, SUPPLIERS s where c.PRICE <
9.0 and s.SUP_ID = c.SUP_ID
```

Warning

Note the use of `===` instead of `==` for comparing two values for equality and `=!=` instead of `!=` for inequality. This is necessary because these operators are already defined (with unsuitable types and semantics) on the base type `Any`, so they cannot be replaced by extension methods. The other comparison operators are the same as in standard Scala code: `<`, `<=`, `>=`, `>`.

The generator expression `suppliers if s.id === c.supID` follows the relationship established by the foreign key `Coffees.supplier`. Instead of repeating the join condition here we can use the foreign key directly:

```
val q3 = for {
  c <- coffees if c.price < 9.0
  s <- c.supplier
} yield (c.name, s.name)
// Equivalent SQL code:
// select c.COF_NAME, s.SUP_NAME from COFFEES c, SUPPLIERS s where c.PRICE <
9.0 and s.SUP_ID = c.S
```

DATABASE CONFIGURATION

You can tell Slick how to connect to the JDBC database of your choice by creating a Database object, which encapsulates the information. There are several factory methods on *slick.jdbc.JdbcBackend.Database* that you can use depending on what connection data you have available.

Using Typesafe Config

The prefered way to configure database connections is through Typesafe Config in your application.conf, which is also used by Play and Akka for their configuration.

```
mydb = {
  dataSourceClass = "org.postgresql.ds.PGSimpleDataSource"
  properties = {
    databaseName = "mydb"
    user = "myuser"
    password = "secret"
  }
  numThreads = 10
```

```
}
```

Such a configuration can be loaded with *Database.forConfig* (see the API documentation of this method for details on the configuration parameters).

```
val db = Database.forConfig("mydb")
```

Using a JDBC URL

You can pass a JDBC URL to forURL. (see your database's JDBC driver's documentation for the correct URL syntax).

```
val db = Database.forURL("jdbc:h2:mem:test1;DB_CLOSE_DELAY=-1",
driver="org.h2.Driver")
```

Here we are connecting to a new, empty, in-memory H2 database called `test1` and keep it resident until the JVM ends (`DB_CLOSE_DELAY=-1`, which is H2 specific).

Using a Database URL

A Database URL, a platform independent URL in the form `vendor://user:password@host:port/db`, is often provided by platforms such as Heroku. You can use a Database URL in Typesafe Config as shown here:

```
databaseUrl {
  dataSourceClass = "slick.jdbc.DatabaseUrlDataSource"
  properties = {
    driver = "slick.driver.PostgresDriver$"
    url = "postgres://user:pass@host/dbname"
  }
}
```

By default, the data source will use the value of the `DATABASE_URL` environment variable. Thus you may omit the `url` property if the `DATABASE_URL` environment variable is set. You may also define a custom environment variable with standard Typesafe Config syntax, such as `${?MYSQL_DATABASE_URL}`.

Or you may pass a DatabaseUrlDataSource object to forDataSource.

```
val db = Database.forDataSource(dataSource: slick.jdbc.DatabaseUrlDataSource)
```

Using a DataSource

You can pass a DataSource object to forDataSource. If you got it from the connection pool of your application framework, this plugs the pool into Slick.

```
val db = Database.forDataSource(dataSource: javax.sql.DataSource)
```

Using a JNDI Name

If you are using JNDI you can pass a JNDI name to forName under which a DataSource object can be looked up.

```
val db = Database.forName(jndiName: String)
```

Database thread pool

Every Database contains an AsyncExecutor that manages the thread pool for asynchronous execution of Database I/O Actions. Its size is the main parameter to tune for the best performance of the Databaseobject. It should be set to the value that you would use for the size of the *connection pool* in a traditional, blocking application (see About Pool Sizing in the HikariCP documentation for further information). When using Database.forConfig, the thread pool is configured directly in the external configuration file together with the connection parameters. If you use any other factory method to get a Database, you can either use a default configuration or specify a custom AsyncExecutor:

```
val db = Database.forURL("jdbc:h2:mem:test1;DB_CLOSE_DELAY=-1",
driver="org.h2.Driver",
  executor = AsyncExecutor("test1", numThreads=10, queueSize=1000))
```

Connection pools

When using a connection pool (which is always recommended in production environments) the *minimum*size of the *connection pool* should also be set to at least the same size. The *maximum* size of the*connection pool* can be set much higher than in a blocking application. Any connections beyond the size of the *thread pool* will only be used when other connections are required to keep a database session open (e.g. while waiting for the result from an asynchronous computation in the middle of a transaction) but are not actively doing any work on the database.

Note that reasonable defaults for the connection pool sizes are calculated from the thread pool size when using Database.forConfig.

Slick uses *prepared* statements wherever possible but it does not cache them on its own. You should therefore enable prepared statement caching in the connection pool's configuration.

DatabaseConfig

On top of the configuration syntax for Database, there is another layer in the form of DatabaseConfig which allows you to configure a Slick driver plus a matching Database together. This makes it easy to abstract over different kinds of database systems by simply changing a configuration file.

Here is a typical DatabaseConfig with a Slick driver object in driver and the database configuration in db:

```
tsql {
  driver = "slick.driver.H2Driver$"
  db {
```

```
    connectionPool = disabled
    driver = "org.h2.Driver"
    url = "jdbc:h2:mem:tsql1;INIT=runscript from 'src/main/resources/create-schema.sql'"
  }
}
```

DATABASE I/O ACTIONS

Anything that you can execute on a database, whether it is a getting the result of a query (myQuery.result), creating a table (myTable.schema.create), inserting data (myTable += item) or something else, is an instance of DBIOAction, parameterized by the result type it will produce when you execute it.

Database I/O Actions can be combined with several different combinators (see the DBIOAction class andDBIO object for details), but they will always be executed strictly sequentially and (at least conceptually) in a single database session.

In most cases you will want to use the type aliases DBIO and StreamingDBIO for non-streaming and streaming Database I/O Actions. They omit the optional *effect types* supported by DBIOAction.

Executing Database I/O Actions

DBIOActions can be executed either with the goal of producing a fully materialized result or streaming data back from the database.

Materialized

You can use run to execute a DBIOAction on a Database and produce a materialized result. This can be, for example, a scalar query result (myTable.length.result), a collection-valued query result (myTable.to[Set].result), or any other action. Every DBIOAction supports this mode of execution.

Execution of the action starts when run is called, and the materialized result is returned as a Future which is completed asynchronously as soon as the result is available:

```
val q = for (c <- coffees) yield c.name
val a = q.result
val f: Future[Seq[String]] = db.run(a)

f.onSuccess { case s => println(s"Result: $s") }
```

Streaming

Collection-valued queries also support streaming results. In this case, the actual collection type is ignored and elements are streamed directly from the result set through a Reactive Streams Publisher, which can be processed and consumed by Akka Streams.

Execution of the DBIOAction does not start until a Subscriber is attached to the stream. Only a singleSubscriber is supported, and any further attempts to subscribe again will fail. Stream elements are signaled as soon as they become available in the streaming part of

the DBIOAction. The end of the stream is signaled only after the entire action has completed. For example, when streaming inside a transaction and all elements have been delivered successfully, the stream can still fail afterwards if the transaction cannot be committed.

```
val q = for (c <- coffees) yield c.name
val a = q.result
val p: DatabasePublisher[String] = db.stream(a)

// .foreach is a convenience method on DatabasePublisher.
// Use Akka Streams for more elaborate stream processing.
p.foreach { s => println(s"Element: $s") }
```

When streaming a JDBC result set, the next result page will be buffered in the background if the Subscriber is not ready to receive more data, but all elements are signaled synchronously and the result set is not advanced before synchronous processing is finished. This allows synchronous callbacks to low-level JDBC values like Blob which depend on the state of the result set. The convenience method mapResult is provided for this purpose:

```
val q = for (c <- coffees) yield c.image
val a = q.result
val p1: DatabasePublisher[Blob] = db.stream(a)
val p2: DatabasePublisher[Array[Byte]] = p1.mapResult { b =>
  b.getBytes(0, b.length().toInt)
}
```

Composing Database I/O Actions

DBIOActions describe sequences of individual actions to execute in strictly sequential order on one database session (at least conceptually), therefore the most commonly used combinators deal with sequencing. Since a DBIOAction eventually results in a Success or Failure, its combinators, just like the ones on Future, have to distinguish between successful and failed executions. Unless specifically noted, all combinators only apply to successful actions. Any failure will abort the sequence of execution and result in a failed Future or*Reactive Stream*.

Sequential Execution

The simplest combinator is DBIO.seq which takes a varargs list of actions to run in sequence, discarding their return value. If you need the return value, you can use andThen to combine two actions and keep the result of the second one. If you need both return values of two actions, there is the zip combinator. For getting all result values from a sequence of actions (of compatible types), use DBIO.sequence. All these combinators work with pre-existing DBIOActions which are composed eagerly.

If an action depends on a previous action in the sequence, you have to compute it on the fly with flatMap ormap. These two methods plus filter enable the use of *for comprehensions* for action sequencing. Since they take function arguments, they also require an implicit ExecutionContext on which to run the function. This way Slick ensures that no non-database code is run on the database thread pool.

Note

You should prefer the less flexible methods without an ExecutionContext where possible. The resulting actions can be executed more efficiently.

Similar to DBIO.sequence for upfront composition, there is DBIO.fold for working with sequences of actions and composing them based on the previous result.

Error Handling

You can use andFinally to perform a cleanup action, no matter whether the previous action succeeded or failed. This is similar to using try ... finally ... in imperative Scala code. A more flexible version ofandFinally is cleanUp. It lets you transform the failure and decide how to fail the resulting action if both the original one and the cleanup failed.

Note

For even more flexible error handling use asTry and failed. Unlike with andFinally and cleanUp the resulting actions cannot be used for streaming.

Primitives

You can convert a Future into an action with DBIO.from. This allows the result of the Future to be used in an action sequence. A pre-existing value or failure can be converted with DBIO.successful and DBIO.failed, respectively.

Debugging

The named combinator names an action. This name can be seen in debug logs if you enable theslick.backend.DatabaseComponent.action logger.

Transactions and Pinned Sessions

When executing a DBIOAction which is composed of several smaller actions, Slick acquires sessions from the connection pool and releases them again as needed so that a session is not kept in use unnecessarily while waiting for the result from a non-database computation (e.g. the function passed to flatMap that determines the next Action to run). All DBIOAction combinators which combine two database actions without any non-database computations in between (e.g. andThen or zip) can fuse these actions for more efficient execution, with the side-effect that the fused action runs inside a single session. You can use withPinnedSession to force the use of a single session, keeping the existing session open even when waiting for non-database computations.

There is a similar combinator called transactionally to force the use of a transaction. This guarantees that the entire DBIOAction that is executed will either succeed or fail atomically.

Warning

Failure is not guaranteed to be atomic *at the level of an individual* DBIOAction that is wrapped withtransactionally, so you should not apply error recovery combinators at that point. An actual database transaction is inly created and committed / rolled back for the outermost transactionally action.

```
val a = (for {
  ns <- coffees.filter(_.name.startsWith("ESPRESSO")).map(_.name).result
  _ <- DBIO.seq(ns.map(n => coffees.filter(_.name === n).delete): _*)
} yield ()).transactionally

val f: Future[Unit] = db.run(a)
```

JDBC Interoperability

In order to drop down to the JDBC level for functionality that is not available in Slick, you can use aSimpleDBIO action which is run on a database thread and gets access to the JDBC Connection:

```
val getAutoCommit = SimpleDBIO[Boolean](_.connection.getAutoCommit)
```

SCHEMAS

This chapter describes how to work with database schemas in Scala code, in particular how to write them manually, which is useful when you start writing an application without a pre-existing database. If you already have a schema in the database, you can also use the code generator to take this work off your hands.

Table Rows

In order to use the Scala API for type-safe queries, you need to define Table row classes for your database schema. These describe the structure of the tables:

```
class Coffees(tag: Tag) extends Table[(String, Int, Double, Int, Int)](tag,
"COFFEES") {
  def name = column[String]("COF_NAME", O.PrimaryKey)
  def supID = column[Int]("SUP_ID")
  def price = column[Double]("PRICE")
  def sales = column[Int]("SALES", O.Default(0))
  def total = column[Int]("TOTAL", O.Default(0))
  def * = (name, supID, price, sales, total)
}
```

All columns are defined through the column method. Each column has a Scala type and a column name for the database (usually in upper-case). The following primitive types are supported out of the box for JDBC-based databases in JdbcProfile (with certain limitations imposed by the individual database drivers):

- *Numeric types*: Byte, Short, Int, Long, BigDecimal, Float, Double
- *LOB types*: java.sql.Blob, java.sql.Clob, Array[Byte]

- *Date types*: java.sql.Date, java.sql.Time, java.sql.Timestamp
- Boolean
- String
- Unit
- java.util.UUID

Nullable columns are represented by `Option[T]` where `T` is one of the supported primitive types.

Note

Currently all operations on Option values use the database's null propagation semantics which may differ from Scala's Option semantics. In particular, `None === None` evaluates to `None`. This behaviour may change in a future major release of Slick.

After the column name, you can add optional column options to a `column` definition. The applicable options are available through the table's `O` object. The following ones are defined for `JdbcProfile`:

`PrimaryKey`

Mark the column as a (non-compound) primary key when creating the DDL statements.

`Default[T](defaultValue: T)`

Specify a default value for inserting data into the table without this column. This information is only used for creating DDL statements so that the database can fill in the missing information.

`DBType(dbType: String)`

Use a non-standard database-specific type for the DDL statements (e.g. `DBType("VARCHAR(20)")` for a `String` column).

`AutoInc`

Mark the column as an auto-incrementing key when creating the DDL statements. Unlike the other column options, this one also has a meaning outside of DDL creation: Many databases do not allow non-AutoInc columns to be returned when inserting data (often silently ignoring other columns), so Slick will check if the return column is properly marked as AutoInc where needed.

`NotNull`, `Nullable`

Explicitly mark the column as nullable or non-nullable when creating the DDL statements for the table. Nullability is otherwise determined from the type (Option or non-Option). There is usually no reason to specify these options.

Every table requires a `*` method containing a default projection. This describes what you get back when you return rows (in the form of a table row object) from a query. Slick's `*` projection does not have to match the one in the database. You can add new columns (e.g. with computed values) or omit some columns as you like. The non-lifted type corresponding to the `*` projection is given as a type parameter to `Table`. For simple, non-mapped tables, this will be a single column type or a tuple of column types.
If your database layout requires *schema names*, you can specify the schema name for a table in front of the table name, wrapped in `Some()`:

```
class Coffees(tag: Tag)
  extends Table[(String, Int, Double, Int, Int)](tag, Some("MYSCHEMA"),
"COFFEES") {
  //...
}
```

Table Query

Alongside the `Table` row class you also need a `TableQuery` value which represents the actual database table:

```
val coffees = TableQuery[Coffees]
```

The simple `TableQuery[T]` syntax is a macro which expands to a proper TableQuery instance that calls the table's constructor (`new TableQuery(new T(_))`).
You can also extend `TableQuery` to use it as a convenient namespace for additional functionality associated with the table:

```
object coffees extends TableQuery(new Coffees(_)) {
  val findByName = this.findBy(_.name)
}
```

Mapped Tables

It is possible to define a mapped table that uses a custom type for its `*` projection by adding a bi-directional mapping with the `<>` operator:

```
case class User(id: Option[Int], first: String, last: String)

class Users(tag: Tag) extends Table[User](tag, "users") {
  def id = column[Int]("id", O.PrimaryKey, O.AutoInc)
  def first = column[String]("first")
  def last = column[String]("last")
  def * = (id.?, first, last) <> (User.tupled, User.unapply)
}
val users = TableQuery[Users]
```

It is optimized for case classes (with a simple `apply` method and an `unapply` method that wraps its result in an `Option`) but it can also be used with arbitrary mapping functions. In these cases it can be useful to call `.shaped` on a tuple on the left-hand side in order to

get its type inferred properly. Otherwise you may have to add full type annotations to the mapping functions.
For case classes with hand-written companion objects, `.tupled` only works if you manually extend the correct Scala function type. Alternatively you can use `(User.apply _).tupled`. See SI-3664 and SI-4808.

Constraints

A foreign key constraint can be defined with a Table's foreignKey method. It first takes a name for the constraint, the referencing column(s) and the referenced table. The second argument list takes a function from the referenced table to its referenced column(s) as well as ForeignKeyAction for `onUpdate` and `onDelete`, which are optional and default to NoAction. When creating the DDL statements for the table, the foreign key definition is added to it.

```
class Suppliers(tag: Tag) extends Table[(Int, String, String, String, String,
String)](tag, "SUPPLIERS") {
  def id = column[Int]("SUP_ID", O.PrimaryKey)
  //...
}
val suppliers = TableQuery[Suppliers]

class Coffees(tag: Tag) extends Table[(String, Int, Double, Int, Int)](tag,
"COFFEES") {
  def supID = column[Int]("SUP_ID")
  //...
  def supplier = foreignKey("SUP_FK", supID, suppliers)(_.id,
onUpdate=ForeignKeyAction.Restrict, onDelete=ForeignKeyAction.Cascade)
  // compiles to SQL:
  //   alter table "COFFEES" add constraint "SUP_FK" foreign key("SUP_ID")
  //     references "SUPPLIERS"("SUP_ID")
  //     on update RESTRICT on delete CASCADE
}
val coffees = TableQuery[Coffees]
```

Independent of the actual constraint defined in the database, such a foreign key can be used to navigate to the referenced data with a *join*. For this purpose, it behaves the same as a manually defined utility method for finding the joined data:

```
def supplier = foreignKey("SUP_FK", supID, suppliers)(_.id,
onUpdate=ForeignKeyAction.Restrict, onDelete=ForeignKeyAction.Cascade)
def supplier2 = suppliers.filter(_.id === supID)
```

A primary key constraint can be defined in a similar fashion by adding a method that calls `primaryKey`. This is useful for defining compound primary keys (which cannot be done with the `O.PrimaryKey` column option):

```
class A(tag: Tag) extends Table[(Int, Int)](tag, "a") {
  def k1 = column[Int]("k1")
```

```
  def k2 = column[Int]("k2")
  def * = (k1, k2)
  def pk = primaryKey("pk_a", (k1, k2))
  // compiles to SQL:
  //   alter table "a" add constraint "pk_a" primary key("k1","k2")
}
```

Other indexes are defined in a similar way with the `index` method. They are non-unique by default unless you set the `unique` parameter:

```
class A(tag: Tag) extends Table[(Int, Int)](tag, "a") {
  def k1 = column[Int]("k1")
  def k2 = column[Int]("k2")
  def * = (k1, k2)
  def idx = index("idx_a", (k1, k2), unique = true)
  // compiles to SQL:
  //   create unique index "idx_a" on "a" ("k1","k2")
}
```

All constraints are discovered reflectively by searching for methods with the appropriate return types which are defined in the table. This behavior can be customized by overriding the `tableConstraints` method.

Data Definition Language

DDL statements for a table can be created with its `TableQuery`'s `schema` method. Multiple `DDL` objects can be concatenated with `++` to get a compound `DDL` object which can create and drop all entities in the correct order, even in the presence of cyclic dependencies between tables. The `create` and `drop` methods produce the Actions for executing the DDL statements:

```
val schema = coffees.schema ++ suppliers.schema
db.run(DBIO.seq(
  schema.create,
  //...
  schema.drop
))
```

You can use the the `statements` method to get the SQL code, like for most other SQL-based Actions. Schema Actions are currently the only Actions that can produce more than one statement.

```
schema.create.statements.foreach(println)
schema.drop.statements.foreach(println)
```

QUERIES

This chapter describes how to write type-safe queries for selecting, inserting, updating and deleting data with Slick's Scala-based query API. The API for building queries is a *lifted embedding*, which means that you are not working with standard Scala types but

with types that are *lifted* into a Rep type constructor. This becomes clearer when you compare the types of a simple Scala collections example

```
case class Coffee(name: String, price: Double)
val coffees: List[Coffee] = //...

val l = coffees.filter(_.price > 8.0).map(_.name)
//                     ^       ^         ^
//                     Double  Double    String
```

... with the types of similar code in Slick:

```
class Coffees(tag: Tag) extends Table[(String, Double)](tag, "COFFEES") {
  def name = column[String]("COF_NAME")
  def price = column[Double]("PRICE")
  def * = (name, price)
}
val coffees = TableQuery[Coffees]

val q = coffees.filter(_.price > 8.0).map(_.name)
//                     ^       ^         ^
//               Rep[Double]  Rep[Double]  Rep[String]
```

All plain types are lifted into `Rep`. The same is true for the table row type `Coffees` which is a subtype of`Rep[(String, Double)]`. Even the literal `8.0` is automatically lifted to a `Rep[Double]` by an implicit conversion because that is what the `>` operator on `Rep[Double]` expects for the right-hand side. This lifting is necessary because the lifted types allow us to generate a syntax tree that captures the query computations. Getting plain Scala functions and values would not give us enough information for translating those computations to SQL.

Expressions

Scalar (non-record, non-collection) values are represented by type `Rep[T]` for which an implicit`TypedType[T]` exists.

The operators and other methods which are commonly used in queries are added through implicit conversions defined in `ExtensionMethodConversions`. The actual methods can be found in the classes`AnyExtensionMethods`, `ColumnExtensionMethods`, `NumericColumnExtensionMethods`,`BooleanColumnExtensionMethods` and `StringColumnExtensionMethods` (cf. ExtensionMethods).

Warning

Most operators mimic the plain Scala equivalents, but you have to use `===` instead of `==` for comparing two values for equality and `=!=` instead of `!=` for inequality. This is necessary because these operators are already defined (with unsuitable types and semantics) on the base type `Any`, so they cannot be replaced by extension methods.

Collection values are represented by the `Query` class (a `Rep[Seq[T]]`) which contains many standard collection methods like `flatMap`, `filter`, `take` and `groupBy`. Due to the two different component types of a `Query` (lifted and plain, e.g. `Query[(Rep[Int], Rep[String]), (Int, String), Seq]`), the signatures for these methods are very complex but the semantics are essentially the same as for Scala collections.

Additional methods for queries of scalar values are added via an implicit conversion to `SingleColumnQueryExtensionMethods`.

Sorting and Filtering

There are various methods with sorting/filtering semantics (i.e. they take a `Query` and return a new `Query` of the same type), for example:

```
val q1 = coffees.filter(_.supID === 101)
// compiles to SQL (simplified):
//   select "COF_NAME", "SUP_ID", "PRICE", "SALES", "TOTAL"
//     from "COFFEES"
//     where "SUP_ID" = 101

val q2 = coffees.drop(10).take(5)
// compiles to SQL (simplified):
//   select "COF_NAME", "SUP_ID", "PRICE", "SALES", "TOTAL"
//     from "COFFEES"
//     limit 5 offset 10

val q3 = coffees.sortBy(_.name.desc.nullsFirst)
// compiles to SQL (simplified):
//   select "COF_NAME", "SUP_ID", "PRICE", "SALES", "TOTAL"
//     from "COFFEES"
//     order by "COF_NAME" desc nulls first

// building criteria using a "dynamic filter" e.g. from a webform.
val criteriaColombian = Option("Colombian")
val criteriaEspresso = Option("Espresso")
val criteriaRoast:Option[String] = None

val q4 = coffees.filter { coffee =>
  List(
      criteriaColombian.map(coffee.name === _),
      criteriaEspresso.map(coffee.name === _),
      criteriaRoast.map(coffee.name === _) // not a condition as `criteriaRoast`
evaluates to `None`
  ).collect({case   Some(criteria)      =>   criteria}).reduceLeftOption(_   ||
_).getOrElse(true: Rep[Boolean])
}
// compiles to SQL (simplified):
//   select "COF_NAME", "SUP_ID", "PRICE", "SALES", "TOTAL"
//     from "COFFEES"
```

```
//     where ("COF_NAME" = 'Colombian' or "COF_NAME" = 'Espresso')
```

Joining and Zipping

Joins are used to combine two different tables or queries into a single query. There are two different ways of writing joins: *Applicative* and *monadic*.

Applicative joins

Applicative joins are performed by calling a method that joins two queries into a single query of a tuple of the individual results. They have the same restrictions as joins in SQL, i.e. the right-hand side may not depend on the left-hand side. This is enforced naturally through Scala's scoping rules.

```
val crossJoin = for {
  (c, s) <- coffees join suppliers
} yield (c.name, s.name)
// compiles to SQL (simplified):
//   select x2."COF_NAME", x3."SUP_NAME" from "COFFEES" x2
//     inner join "SUPPLIERS" x3

val innerJoin = for {
  (c, s) <- coffees join suppliers on (_.supID === _.id)
} yield (c.name, s.name)
// compiles to SQL (simplified):
//   select x2."COF_NAME", x3."SUP_NAME" from "COFFEES" x2
//     inner join "SUPPLIERS" x3
//     on x2."SUP_ID" = x3."SUP_ID"

val leftOuterJoin = for {
  (c, s) <- coffees joinLeft suppliers on (_.supID === _.id)
} yield (c.name, s.map(_.name))
// compiles to SQL (simplified):
//   select x2."COF_NAME", x3."SUP_NAME" from "COFFEES" x2
//     left outer join "SUPPLIERS" x3
//     on x2."SUP_ID" = x3."SUP_ID"

val rightOuterJoin = for {
  (c, s) <- coffees joinRight suppliers on (_.supID === _.id)
} yield (c.map(_.name), s.name)
// compiles to SQL (simplified):
//   select x2."COF_NAME", x3."SUP_NAME" from "COFFEES" x2
//     right outer join "SUPPLIERS" x3
//     on x2."SUP_ID" = x3."SUP_ID"

val fullOuterJoin = for {
  (c, s) <- coffees joinFull suppliers on (_.supID === _.id)
} yield (c.map(_.name), s.map(_.name))
// compiles to SQL (simplified):
//   select x2."COF_NAME", x3."SUP_NAME" from "COFFEES" x2
//     full outer join "SUPPLIERS" x3
```

```
//     on x2."SUP_ID" = x3."SUP_ID"
```

Note the use of `map` in the `yield` clauses of the outer joins. Since these joins can introduce additional NULL values (on the right-hand side for a left outer join, on the left-hand sides for a right outer join, and on both sides for a full outer join), the respective sides of the join are wrapped in an `Option` (with `None` representing a row that was not matched).

Monadic joins

Monadic joins are created with `flatMap`. They are theoretically more powerful than applicative joins because the right-hand side may depend on the left-hand side. However, this is not possible in standard SQL, so Slick has to compile them down to applicative joins, which is possible in many useful cases but not in all of them (and there are cases where it is possible in theory but Slick cannot perform the required transformation yet). If a monadic join cannot be properly translated, it will fail at runtime. A *cross-join* is created with a `flatMap` operation on a `Query` (i.e. by introducing more than one generator in a for-comprehension):

```
val monadicCrossJoin = for {
  c <- coffees
  s <- suppliers
} yield (c.name, s.name)
// compiles to SQL:
//   select x2."COF_NAME", x3."SUP_NAME"
//     from "COFFEES" x2, "SUPPLIERS" x3
```

If you add a filter expression, it becomes an *inner join*:

```
val monadicInnerJoin = for {
  c <- coffees
  s <- suppliers if c.supID === s.id
} yield (c.name, s.name)
// compiles to SQL:
//   select x2."COF_NAME", x3."SUP_NAME"
//     from "COFFEES" x2, "SUPPLIERS" x3
//     where x2."SUP_ID" = x3."SUP_ID"
```

The semantics of these monadic joins are the same as when you are using `flatMap` on Scala collections.

Note

Slick currently generates *implicit* joins in SQL (`select ... from a, b where ...`) for monadic joins, and *explicit* joins (`select ... from a join b on ...`) for applicative joins. This is subject to change in future versions.

Zip joins

In addition to the usual applicative join operators supported by relational databases (which are based off a cross join or outer join), Slick also has *zip joins* which create a

pairwise join of two queries. The semantics are again the same as for Scala collections, using the `zip` and `zipWith` methods:

```
val zipJoinQuery = for {
  (c, s) <- coffees zip suppliers
} yield (c.name, s.name)

val zipWithJoin = for {
  res <- coffees.zipWith(suppliers, (c: Coffees, s: Suppliers) => (c.name,
s.name))
} yield res
```

A particular kind of zip join is provided by `zipWithIndex`. It zips a query result with an infinite sequence starting at 0. Such a sequence cannot be represented by an SQL database and Slick does not currently support it, either. The resulting zipped query, however, can be represented in SQL with the use of a *row number* function, so `zipWithIndex` is supported as a primitive operator:

```
val zipWithIndexJoin = for {
  (c, idx) <- coffees.zipWithIndex
} yield (c.name, idx)
```

Unions

Two queries can be concatenated with the `++` (or `unionAll`) and `union` operators if they have compatible types:

```
val q1 = coffees.filter(_.price < 8.0)
val q2 = coffees.filter(_.price > 9.0)

val unionQuery = q1 union q2
// compiles to SQL (simplified):
//   select x8."COF_NAME", x8."SUP_ID", x8."PRICE", x8."SALES", x8."TOTAL"
//     from "COFFEES" x8
//     where x8."PRICE" < 8.0
//   union select x9."COF_NAME", x9."SUP_ID", x9."PRICE", x9."SALES", x9."TOTAL"
//     from "COFFEES" x9
//     where x9."PRICE" > 9.0

val unionAllQuery = q1 ++ q2
// compiles to SQL (simplified):
//   select x8."COF_NAME", x8."SUP_ID", x8."PRICE", x8."SALES", x8."TOTAL"
//     from "COFFEES" x8
//     where x8."PRICE" < 8.0
//     union all select x9."COF_NAME", x9."SUP_ID", x9."PRICE", x9."SALES",
x9."TOTAL"
//     from "COFFEES" x9
//     where x9."PRICE" > 9.0
```

Unlike `union` which filters out duplicate values, `++` simply concatenates the results of the individual queries, which is usually more efficient.

Aggregation

The simplest form of aggregation consists of computing a primitive value from a Query that returns a single column, usually with a numeric type, e.g.:

```
val q = coffees.map(_.price)

val q1 = q.min
// compiles to SQL (simplified):
//   select min(x4."PRICE") from "COFFEES" x4

val q2 = q.max
// compiles to SQL (simplified):
//   select max(x4."PRICE") from "COFFEES" x4

val q3 = q.sum
// compiles to SQL (simplified):
//   select sum(x4."PRICE") from "COFFEES" x4

val q4 = q.avg
// compiles to SQL (simplified):
//   select avg(x4."PRICE") from "COFFEES" x4
```

Note that these aggregate queries return a scalar result, not a collection. Some aggregation functions are defined for arbitrary queries (of more than one column):

```
val q1 = coffees.length
// compiles to SQL (simplified):
//   select count(1) from "COFFEES"

val q2 = coffees.exists
// compiles to SQL (simplified):
//   select exists(select * from "COFFEES")
```

Grouping is done with the `groupBy` method. It has the same semantics as for Scala collections:

```
val q = (for {
  c <- coffees
  s <- c.supplier
} yield (c, s)).groupBy(_._1.supID)

val q2 = q.map { case (supID, css) =>
  (supID, css.length, css.map(_._1.price).avg)
}
// compiles to SQL:
//   select x2."SUP_ID", count(1), avg(x2."PRICE")
//     from "COFFEES" x2, "SUPPLIERS" x3
```

```
//     where x3."SUP_ID" = x2."SUP_ID"
//     group by x2."SUP_ID"
```

The intermediate query `q` contains nested values of type `Query`. These would turn into nested collections when executing the query, which is not supported at the moment. Therefore it is necessary to flatten the nested queries immediately by aggregating their values (or individual columns) as done in `q2`.

Querying

A Query can be converted into an Action by calling its `result` method. The Action can then be executeddirectly in a streaming or fully materialized way, or composed further with other Actions:

```
val q = coffees.map(_.price)
val action = q.result
val result: Future[Seq[Double]] = db.run(action)
val sql = action.statements.head
```

If you only want a single result value, you can call `head` or `headOption` on the `result` Action.

Deleting

Deleting works very similarly to querying. You write a query which selects the rows to delete and then get an Action by calling the `delete` method on it:

```
val q = coffees.filter(_.supID === 15)
val action = q.delete
val affectedRowsCount: Future[Int] = db.run(action)
val sql = action.statements.head
```

A query for deleting must only select from a single table. Any projection is ignored (it always deletes full rows).

Inserting

Inserts are done based on a projection of columns from a single table. When you use the table directly, the insert is performed against its `*` projection. Omitting some of a table's columns when inserting causes the database to use the default values specified in the table definition, or a type-specific default in case no explicit default was given. All methods for building insert Actions are defined inCountingInsertActionComposer and ReturningInsertActionComposer.

```
val insertActions = DBIO.seq(
  coffees += ("Colombian", 101, 7.99, 0, 0),

  coffees ++= Seq(
    ("French_Roast", 49, 8.99, 0, 0),
    ("Espresso",    150, 9.99, 0, 0)
```

```
  ),

  // "sales" and "total" will use the default value 0:
  coffees.map(c => (c.name, c.supID, c.price)) += ("Colombian_Decaf", 101, 8.99)
)

// Get the statement without having to specify a value to insert:
val sql = coffees.insertStatement

// compiles to SQL:
//   INSERT INTO "COFFEES" ("COF_NAME","SUP_ID","PRICE","SALES","TOTAL") VALUES
(?,?,?,?,?)
```

When you include an `AutoInc` column in an insert operation, it is silently ignored, so that the database can generate the proper value. In this case you usually want to get back the auto-generated primary key column. By default, `+=` gives you a count of the number of affected rows (which will usually be 1) and `++=` gives you an accumulated count in an `Option` (which can be `None` if the database system does not provide counts for all rows). This can be changed with the `returning` method where you specify the columns to be returned (as a single value or tuple from `+=` and a `Seq` of such values from `++=`):

```
val userId =
  (users returning users.map(_.id)) += User(None, "Stefan", "Zeiger")
```

Note

Many database systems only allow a single column to be returned which must be the table's auto-incrementing primary key. If you ask for other columns a `SlickException` is thrown at runtime (unless the database actually supports it).

You can follow the `returning` method with the `into` method to map the inserted values and the generated keys (specified in returning) to a desired value. Here is an example of using this feature to return an object with an updated id:

```
val userWithId =
  (users returning users.map(_.id)
        into ((user,id) => user.copy(id=Some(id)))
  ) += User(None, "Stefan", "Zeiger")
```

Instead of inserting data from the client side you can also insert data created by a `Query` or a scalar expression that is executed in the database server:

```
class Users2(tag: Tag) extends Table[(Int, String)](tag, "users2") {
  def id = column[Int]("id", O.PrimaryKey)
  def name = column[String]("name")
  def * = (id, name)
}
val users2 = TableQuery[Users2]

val actions = DBIO.seq(
```

```
  users2.schema.create,
  users2 forceInsertQuery (users.map { u => (u.id, u.first ++ " " ++ u.last) }),
  users2 forceInsertExpr (users.length + 1, "admin")
)
```

In these cases, `AutoInc` columns are *not* ignored.

Updating

Updates are performed by writing a query that selects the data to update and then replacing it with new data. The query must only return raw columns (no computed values) selected from a single table. The relevant methods for updating are defined in UpdateExtensionMethods.

```
val q = for { c <- coffees if c.name === "Espresso" } yield c.price
val updateAction = q.update(10.49)

// Get the statement without having to specify an updated value:
val sql = q.updateStatement

// compiles to SQL:
//   update "COFFEES" set "PRICE" = ? where "COFFEES"."COF_NAME" = 'Espresso'
```

There is currently no way to use scalar expressions or transformations of the existing data in the database for updates.

Upserting

Upserting is performed by supplying a row to be either inserted or updated. The object must contain the table's primary key, since the update portion needs to be able to find a uniquelly matching row.

```
val updated = users.insertOrUpdate(User(Some(1), "Admin", "Zeiger"))
// returns: number of rows updated

val updatedAdmin = (users returning users).insertOrUpdate(User(Some(1), "Slick
Admin", "Zeiger"))
// returns: None if updated, Some((Int, String)) if row inserted
```

Compiled Queries

Database queries typically depend on some parameters, e.g. an ID for which you want to retrieve a matching database row. You can write a regular Scala function to create a parameterized `Query` object each time you need to execute that query but this will incur the cost of recompiling the query in Slick (and possibly also on the database if you don't use bind variables for all parameters). It is more efficient to pre-compile such parameterized query functions:

```
def userNameByIDRange(min: Rep[Int], max: Rep[Int]) =
  for {
```

```
    u <- users if u.id >= min && u.id < max
  } yield u.first

val userNameByIDRangeCompiled = Compiled(userNameByIDRange _)

// The query will be compiled only once:
val namesAction1 = userNameByIDRangeCompiled(2, 5).result
val namesAction2 = userNameByIDRangeCompiled(1, 3).result
// Also works for .insert, .update and .delete
```

This works for all functions that take parameters consisting only of individual columns or or records of columns and return a `Query` object or a scalar query. See the API documentation for Compiled and its subclasses for details on composing compiled queries.

Be aware that `take` and `drop` take `ConstColumn[Long]` parameters. Unlike `Rep[Long]`, which could be substituted by another value computed by a query, a ConstColumn can only be literal value or a parameter of a compiled query. This is necessary because the actual value has to be known by the time the query is prepared for execution by Slick.

```
val userPaged = Compiled((d: ConstColumn[Long], t: ConstColumn[Long]) =>
users.drop(d).take(t))

val usersAction1 = userPaged(2, 1).result
val usersAction2 = userPaged(1, 3).result
```

You can use a compiled query for querying, inserting, updating and deleting data. For backwards-compatibility with Slick 1.0 you can still create a compiled query by calling `flatMap` on a Parameters object. In many cases this enables you to write a single *for comprehension* for a compiled query:

```
val userNameByID = for {
  id <- Parameters[Int]
  u <- users if u.id === id
} yield u.first

val nameAction = userNameByID(2).result.head

val userNameByIDRange = for {
  (min, max) <- Parameters[(Int, Int)]
  u <- users if u.id >= min && u.id < max
} yield u.first

val namesAction = userNameByIDRange(2, 5).result
```

SCHEMA CODE GENERATION

The Slick code generator is a convenient tool for working with an existing or evolving database schema. It can be run stand-alone or integrated into you sbt build for creating all code Slick needs to work.

Overview

By default, the code generator generates `Table` classes, corresponding `TableQuery` values, which can be used in a collection-like manner, as well as case classes for holding complete rows of values. For tables with more than 22 columns the generator automatically switches to Slick's experimental `HList` implementation for overcoming Scala's tuple size limit. (In Scala <= 2.10.3 use `HCons` instead of `::` as a type contructor due to performance issues during compilation, which are fixed in 2.10.4 and later.)
Parts of the generator are also explained in our talk at Scala eXchange 2013.

Standalone use

To include Slick's code generator use the published library. For sbt projects add following to your build definition - `build.sbt` or `project/Build.scala`:

```
libraryDependencies += "com.typesafe.slick" %% "slick-codegen" % "3.1.0"
```

For Maven projects add the following to your `<dependencies>`:

```
<dependency>
  <groupId>com.typesafe.slick</groupId>
  <artifactId>slick-codegen_2.10</artifactId>
  <version>3.1.0</version>
</dependency>
```

Slick's code generator comes with a default runner that can be used from the command line or from Java/Scala. You can simply execute

```
slick.codegen.SourceCodeGenerator.main(
  Array(slickDriver, jdbcDriver, url, outputFolder, pkg)
)
```

or

```
slick.codegen.SourceCodeGenerator.main(
  Array(slickDriver, jdbcDriver, url, outputFolder, pkg, user, password)
)
```

and provide the following values

- **slickDriver** Fully qualified name of Slick driver class, e.g. *"slick.driver.H2Driver"*
- **jdbcDriver** Fully qualified name of jdbc driver class, e.g. *"org.h2.Driver"*
- **url** jdbc url, e.g. *"jdbc:postgresql://localhost/test"*
- **outputFolder** Place where the package folder structure should be put
- **pkg** Scala package the generated code should be places in
- **user** database connection user name
- **password** database connection password

Integrated into sbt

The code generator can be run before every compilation or manually in sbt. An example project showing both can be found here.

Generated Code

By default, the code generator places a file `Tables.scala` in the given folder in a subfolder corresponding to the package. The file contains an `object Tables` from which the code can be imported for use right away. Make sure you use the same Slick driver. The file also contains a `trait Tables` which can be used in the cake pattern.

Warning

When using the generated code, be careful **not** to mix different database drivers accidentally. The default`object Tables` uses the driver used during code generation. Using it together with a different driver for queries will lead to runtime errors. The generated `trait Tables` can be used with a different driver, but be aware, that this is currently untested and not officially supported. It may or may not work in your case. We will officially support this at some point in the future.

Customization

The generator can be flexibly customized by overriding methods to programmatically generate any code based on the data model. This can be used for minor customizations as well as heavy, model driven code generation, e.g. for framework bindings in Play, other data-related, repetitive sections of applications, etc.

This example shows a customized code-generator and how to setup up a multi-project sbt build, which compiles and runs it before compiling the main sources.

The implementation of the code generator is structured into a small hierarchy of sub-generators responsible for different fragments of the complete output. The implementation of each sub-generator can be swapped out for a customized one by overriding the corresponding factory method. SourceCodeGenerator contains a factory method Table, which it uses to generate a sub-generator for each table. The sub-generator Table in turn contains sub-generators for Table classes, entity case classes, columns, key, indices, etc. Custom sub-generators can easily be added as well.

Within the sub-generators the relevant part of the Slick data model can be accessed to drive the code generation.

Please see the api documentation for info on all of the methods that can be overridden for customization.
Here is an example for customizing the generator:

```
import slick.codegen.SourceCodeGenerator
// fetch data model
val modelAction = H2Driver.createModel(Some(H2Driver.defaultTables)) // you can
filter specific tables here
val modelFuture = db.run(modelAction)
// customize code generator
val codegenFuture = modelFuture.map(model => new SourceCodeGenerator(model) {
  // override mapped table and class name
  override def entityName =
    dbTableName => dbTableName.dropRight(1).toLowerCase.toCamelCase
  override def tableName =
    dbTableName => dbTableName.toLowerCase.toCamelCase

  // add some custom import
  override def code = "import foo.{MyCustomType,MyCustomTypeMapper}" + "\n" +
super.code

  // override table generator
  override def Table = new Table(_){
    // disable entity class generation and mapping
    override def EntityType = new EntityType{
      override def classEnabled = false
    }

     // override contained column generator
    override def Column = new Column(_){
      // use the data model member of this column to change the Scala type,
      // e.g. to a custom enum or anything else
      override def rawType =
        if(model.name   ==   "SOME_SPECIAL_COLUMN_NAME")   "MyCustomType"   else
super.rawType
    }
  }
})
codegenFuture.onSuccess { case codegen =>
  codegen.writeToFile(

"slick.driver.H2Driver","some/folder/","some.packag","Tables","Tables.scala"
  )
}
```

USER-DEFINED FEATURES

This chapter describes how to use custom data types and database functions with Slick's Scala API.

Scalar Database Functions

If your database system supports a scalar function that is not available as a method in Slick you can define it as a SimpleFunction. There are predefined methods for creating unary, binary and ternary functions with fixed parameter and return types.

```
// H2 has a day_of_week() function which extracts the day of week from a timestamp
val dayOfWeek = SimpleFunction.unary[Date, Int]("day_of_week")

// Use the lifted function in a query to group by day of week
val q1 = for {
  (dow, q) <- salesPerDay.map(s => (dayOfWeek(s.day), s.count)).groupBy(_._1)
} yield (dow, q.map(_._2).sum)
```

If you need more flexibility regarding the types (e.g. for varargs, polymorphic functions, or to support Option and non-Option types in a single function), you can use `SimpleFunction.apply` to get an untyped instance and write your own wrapper function with the proper type-checking:

```
def dayOfWeek2(c: Rep[Date]) =
  SimpleFunction[Int]("day_of_week").apply(Seq(c))
```

SimpleBinaryOperator and SimpleLiteral work in a similar way. For even more flexibility (e.g. function-like expressions with unusual syntax), you can use SimpleExpression.

```
val current_date = SimpleLiteral[java.sql.Date]("CURRENT_DATE")
salesPerDay.map(_ => current_date)
```

Other Database Functions And Stored Procedures

For database functions that return complete tables or stored procedures please use Plain SQL Queries. Stored procedures that return multiple result sets are currently not supported.

Using Custom Scalar Types in Queries

If you need a custom column type you can implement ColumnType. The most common scenario is mapping an application-specific type to an already supported type in the database. This can be done much simpler by using MappedColumnType which takes care of all the boilerplate. It comes with the usual import from the driver.

```
// An algebraic data type for booleans
sealed trait Bool
case object True extends Bool
case object False extends Bool
```

```
// And a ColumnType that maps it to Int values 1 and 0
implicit val boolColumnType = MappedColumnType.base[Bool, Int](
  { b => if(b == True) 1 else 0 },    // map Bool to Int
  { i => if(i == 1) True else False } // map Int to Bool
)

// You can now use Bool like any built-in column type (in tables, queries, etc.)
```

You can also subclass MappedJdbcType for a bit more flexibility.

If you have a wrapper class (which can optionally be a case class and/or value class) for an underlying value of some supported type, you can make it extend MappedTo to get a macro-generated implicit `ColumnType` for free. Such wrapper classes are commonly used for type-safe table-specific primary key types:

```
// A custom ID type for a table
case class MyID(value: Long) extends MappedTo[Long]

// Use it directly for this table's ID -- No extra boilerplate needed
class MyTable(tag: Tag) extends Table[(MyID, String)](tag, "MY_TABLE") {
  def id = column[MyID]("ID")
  def data = column[String]("DATA")
  def * = (id, data)
}
```

Using Custom Record Types in Queries

Record types are data structures containing a statically known number of components with individually declared types. Out of the box, Slick supports Scala tuples (up to arity 22) and Slick's own HListimplementation. Record types can be nested and mixed arbitrarily.

In order to use custom record types (case classes, custom HLists, tuple-like types, etc.) in queries you need to tell Slick how to map them between queries and results. You can do that using a Shape extendingMappedScalaProductShape.

Polymorphic Types (e.g. Custom Tuple Types or HLists)

The distinguishing feature of a *polymorphic* record type is that it abstracts over its element types, so you can use the same record type for both, lifted and plain element types. You can add support for custom polymorphic record types using an appropriate implicit Shape.

Here is an example for a type `Pair`:

```
// A custom record class
case class Pair[A, B](a: A, b: B)

// A Shape implementation for Pair
final class PairShape[Level <: ShapeLevel, M <: Pair[_,_], U <: Pair[_,_] :
ClassTag, P <: Pair[_,_]](
  val shapes: Seq[Shape[_, _, _, _]])
```

```
extends MappedScalaProductShape[Level, Pair[_,_], M, U, P] {
  def buildValue(elems: IndexedSeq[Any]) = Pair(elems(0), elems(1))
  def copy(shapes: Seq[Shape[_ <: ShapeLevel, _, _, _]]) = new PairShape(shapes)
}

implicit def pairShape[Level <: ShapeLevel, M1, M2, U1, U2, P1, P2](
  implicit s1: Shape[_ <: Level, M1, U1, P1], s2: Shape[_ <: Level, M2, U2, P2]
) = new PairShape[Level, Pair[M1, M2], Pair[U1, U2], Pair[P1, P2]](Seq(s1, s2))
```

The implicit method `pairShape` in this example provides a Shape for a `Pair` of two element types whenever Shapes for the individual element types are available.

With these definitions in place, we can use the `Pair` record type in every location in Slick where a tuple or`HList` would be acceptable:

```
// Use it in a table definition
class A(tag: Tag) extends Table[Pair[Int, String]](tag, "shape_a") {
  def id = column[Int]("id", O.PrimaryKey)
  def s = column[String]("s")
  def * = Pair(id, s)
}
val as = TableQuery[A]

// Insert data with the custom shape
val insertAction = DBIO.seq(
  as += Pair(1, "a"),
  as += Pair(2, "c"),
  as += Pair(3, "b")
)

// Use it for returning data from a query
val q2 = as
  .map { case a => Pair(a.id, (a.s ++ a.s)) }
  .filter { case Pair(id, _) => id =!= 1 }
  .sortBy { case Pair(_, ss) => ss }
  .map { case Pair(id, ss) => Pair(id, Pair(42 , ss)) }
// returns: Vector(Pair(3,Pair(42,"bb")), Pair(2,Pair(42,"cc")))
```

Monomorphic Case Classes

Custom *case classes* are frequently used as monomorphic record types (i.e. record types where the element types are fixed). In order to use them in Slick, you need to define the case class for a record of plain values (as usual) plus an additional case class for a matching record of lifted values.

In order to provide a Shape for a custom case class, you can use CaseClassShape:

```
// two custom case class variants
case class LiftedB(a: Rep[Int], b: Rep[String])
case class B(a: Int, b: String)

// custom case class mapping
```

```
implicit object BShape extends CaseClassShape(LiftedB.tupled, B.tupled)

class BRow(tag: Tag) extends Table[B](tag, "shape_b") {
  def id = column[Int]("id", O.PrimaryKey)
  def s = column[String]("s")
  def * = LiftedB(id, s)
}
val bs = TableQuery[BRow]

val insertActions = DBIO.seq(
  bs += B(1, "a"),
  bs.map(b => (b.id, b.s)) += ((2, "c")),
  bs += B(3, "b")
)

val q3 = bs
  .map { case b => LiftedB(b.id, (b.s ++ b.s)) }
  .filter { case LiftedB(id, _) => id =!= 1 }
  .sortBy { case LiftedB(_, ss) => ss }

// returns: Vector(B(3,"bb"), B(2,"cc"))
```

Note that this mechanism can be used as an alternative to client-side mappings with the <> operator. It requires a bit more boilerplate but allows you to use the same field names in both, plain and lifted records.

Combining Mapped Types

In the following example we are combining a mapped case class and the mapped `Pair` type in another mapped case class.

```
// Combining multiple mapped types
case class LiftedC(p: Pair[Rep[Int],Rep[String]], b: LiftedB)
case class C(p: Pair[Int,String], b: B)

implicit object CShape extends CaseClassShape(LiftedC.tupled, C.tupled)

class CRow(tag: Tag) extends Table[C](tag, "shape_c") {
  def id = column[Int]("id")
  def s = column[String]("s")
  def projection = LiftedC(
    Pair(column("p1"),column("p2")), // (cols defined inline, type inferred)
    LiftedB(id,s)
  )
  def * = projection
}
val cs = TableQuery[CRow]

val insertActions2 = DBIO.seq(
  cs += C(Pair(7,"x"), B(1,"a")),
  cs += C(Pair(8,"y"), B(2,"c")),
  cs += C(Pair(9,"z"), B(3,"b"))
```

```
)

val q4 = cs
  .map { case c => LiftedC(c.projection.p, LiftedB(c.id,(c.s ++ c.s))) }
  .filter { case LiftedC(_, LiftedB(id,_)) => id =!= 1 }
  .sortBy { case LiftedC(Pair(_,p2), LiftedB(_,ss)) => ss++p2 }

.. куегктыЖ Мусещк(С(Зфшк(9бЭяЭ)бИ(ЗбЭииЭ))б С(Зфшк(8бЭнЭ)бИ(2бЭссЭ)))
```

PLAIN SQL QUERIES

Sometimes you may need to write your own SQL code for an operation which is not well supported at a higher level of abstraction. Instead of falling back to the low level of JDBC, you can use Slick's *Plain SQL*queries with a much nicer Scala-based API.

Note

The rest of this chapter is based on the Slick Plain SQL Queries template. The prefered way of reading this introduction is in Activator, where you can edit and run the code directly while reading the tutorial.

Scaffolding

The database connection is opened in the usual way. All *Plain SQL* queries result in DBIOActions that can be composed and run like any other action.

String Interpolation

Plain SQL queries in Slick are built via string interpolation using the `sql`, `sqlu` and `tsql` interpolators. They are available through the standard `api._` import from a Slick driver:

```
import slick.driver.H2Driver.api._
```

You can see the simplest use case in the following methods where the `sqlu` interpolator is used with a literal SQL string:

```
def createCoffees: DBIO[Int] =
  sqlu"""create table coffees(
    name varchar not null,
    sup_id int not null,
    price double not null,
    sales int not null,
    total int not null,
    foreign key(sup_id) references suppliers(id))"""

def createSuppliers: DBIO[Int] =
  sqlu"""create table suppliers(
    id int not null primary key,
    name varchar not null,
    street varchar not null,
    city varchar not null,
```

```
    state varchar not null,
    zip varchar not null)"""

def insertSuppliers: DBIO[Unit] = DBIO.seq(
  // Insert some suppliers
  sqlu"insert into suppliers values(101, 'Acme, Inc.', '99 Market Street',
'Groundsville', 'CA', '95199')",
  sqlu"insert into suppliers values(49, 'Superior Coffee', '1 Party Place',
'Mendocino', 'CA', '95460')",
  sqlu"insert into suppliers values(150, 'The High Ground', '100 Coffee Lane',
'Meadows', 'CA', '93966')"
)
```

The `sqlu` interpolator is used for DML statements which produce a row count instead of a result set. Therefore they are of type `DBIO[Int]`.

Any variable or expression injected into a query gets turned into a bind variable in the resulting query string. It is not inserted directly into a query string, so there is no danger of SQL injection attacks. You can see this used in here:

```
def insert(c: Coffee): DBIO[Int] =
  sqlu"insert into coffees values (${c.name}, ${c.supID}, ${c.price},
${c.sales}, ${c.total})"
```

The SQL statement produced by this method is always the same:

```
insert into coffees values (?, ?, ?, ?, ?)
```

Note the use of the DBIO.sequence combinator which is useful for this kind of code:

```
val inserts: Seq[DBIO[Int]] = Seq(
  Coffee("Colombian", 101, 7.99, 0, 0),
  Coffee("French_Roast", 49, 8.99, 0, 0),
  Coffee("Espresso", 150, 9.99, 0, 0),
  Coffee("Colombian_Decaf", 101, 8.99, 0, 0),
  Coffee("French_Roast_Decaf", 49, 9.99, 0, 0)
).map(insert)

val combined: DBIO[Seq[Int]] = DBIO.sequence(inserts)
combined.map(_.sum)
```

Unlike the simpler DBIO.seq combinator which runs a (varargs) sequence of database I/O actions in the given order and discards the return values, DBIO.sequence turns a `Seq[DBIO[T]]` into a `DBIO[Seq[T]]`, thus preserving the results of all individual actions. It is used here to sum up the affected row counts of all inserts.

Result Sets

The following code uses tbe `sql` interpolator which returns a result set produced by a statement. The interpolator by itself does not produce a `DBIO` value. It needs to be followed by a call to `.as` to define the row type:

```
sql"""select c.name, s.name
      from coffees c, suppliers s
      where c.price < $price and s.id = c.sup_id""".as[(String, String)]
```

This results in a `DBIO[Seq[(String, String)]]`. The call to `as` takes an implicit GetResult parameter which extracts data of the requested type from a result set. There are predefined `GetResult` implicits for the standard JDBC types, for Options of those (to represent nullable columns) and for tuples of types which have a `GetResult`. For non-standard return types you have to define your own converters:

```
// Case classes for our data
case class Supplier(id: Int, name: String, street: String, city: String, state:
String, zip: String)
case class Coffee(name: String, supID: Int, price: Double, sales: Int, total:
Int)

// Result set getters
implicit val getSupplierResult = GetResult(r => Supplier(r.nextInt,
r.nextString, r.nextString,
  r.nextString, r.nextString, r.nextString))
implicit val getCoffeeResult = GetResult(r => Coffee(r.<<, r.<<, r.<<, r.<<,
r.<<))
```

`GetResult[T]` is simply a wrapper for a function `PositionedResult => T`. The implicit val for `Supplier` uses the explicit `PositionedResult` methods `getInt` and `getString` to read the next `Int` or `String` value in the current row. The second one uses the shortcut method `<<` which returns a value of whatever type is expected at this place. (Of course you can only use it when the type is actually known like in this constructor call.

Splicing Literal Values

While most parameters should be inserted into SQL statements as bind variables, sometimes you need to splice literal values directly into the statement, for example to abstract over table names or to run dynamically generated SQL code. You can use `#$` instead of `$` in all interpolators for this purpose, as shown in the following piece of code:

```
val table = "coffees"
sql"select * from #$table where name = $name".as[Coffee].headOption
```

Type-Checked SQL Statements

The interpolators you have seen so far only construct a SQL statement at runtime. This provides a safe and easy way of building statements but they are still just embedded

strings. If you have a syntax error in a statement or the types don't match up between the database and your Scala code, this cannot be detected at compile-time. You can use the `tsql` interpolator instead of `sql` to get just that:

```
def getSuppliers(id: Int): DBIO[Seq[(Int, String, String, String, String, String)]] =
  tsql"select * from suppliers where id > $id"
```

Note that `tsql` directly produces a `DBIOAction` of the correct type without requiring a call to `.as`.

In order to give the compiler access to the database, you have to provide a configuration that can be resolved at compile-time. This is done with the StaticDatabaseConfig annotation:

```
@StaticDatabaseConfig("file:src/main/resources/application.conf#tsql")
```

In this case it points to the path "tsql" in a local `application.conf` file, which must contain an appropriate configiration for a StaticDatabaseConfig object, not just a `Database`.

Note

You can get `application.conf` resolved via the classpath (as usual) by omitting the path and only specifying a fragment in the URL, or you can use a `resource:` URL scheme for referencing an arbitrary classpath resouce, but in both cases, they have to be on the *compiler's* own classpath, not just the source path or the runtime classpath. Depending on the build tool this may not be possible, so it's usually better to use a relative `file:` URL.

You can also retrieve the statically configured DatabaseConfig at runtime:

```
val dc = DatabaseConfig.forAnnotation[JdbcProfile]
import dc.driver.api._
val db = dc.db
```

This gives you the Slick driver for the standard `api._` import and the `Database`. Note that it is not mandatory to use the same configuration. You can get a Slick driver and `Database` at runtime in any other way you like and only use the `StaticDatabaseConfig` for compile-time checking.

COMING FROM ORM TO SLICK

Introduction

Slick is not an object-relational mapper (ORM) like Hibernate or other JPA-based products. Slick is a data persistence solution like ORMs and naturally shares some concepts, but it also has significant differences. This chapter explains the differences in order to help you get the best out of Slick and avoid confusion for those familiar with

ORMs. We explain how Slick manages to avoid many of the problems often referred to as the object-relational impedance mismatch.
A good term to describe Slick is functional-relational mapper. Slick allows working with relational data much like with immutable collections and focuses on flexible query composition and strongly controlled side-effects. ORMs usually expose mutable object-graphs, use side-effects like read- and write-caches and hard-code support for anticipated use-cases like inheritance or relationships via association tables. Slick focuses on getting the best out of accessing a relational data store. ORMs focus on persisting an object-graph.

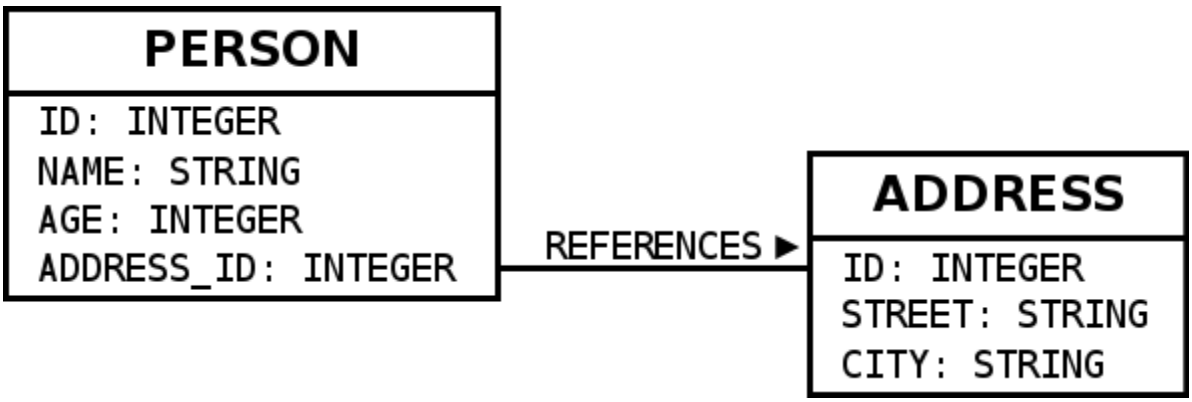
ORMs are a natural approach when using databases from object-oriented languages. They try to allow working with persisted object-graphs partly as if they were completely in memory. Objects can be modified, associations can be changed and the object graph can be traversed. In practice this is not exactly easy to achieve due to the so called object-relational impedance mismatch. It makes ORMs hard to implement and often complicated to use for more than simple cases and if performance matters. Slick in contrast does not expose an object-graph. It is inspired by SQL and the relational model and mostly just maps their concepts to the most closely corresponding, type-safe Scala features. Database queries are expressed using a restricted, immutable, purely-functional subset of Scala much like collections. Slick also offer first-class SQL support as an alternative.
In practice, ORMs often suffer from conceptual problems of what they try to achieve, from mere problems of the implementations and from mis-use, because of their complexity. In the following we look at many features of ORMs and what you would use with Slick instead. We'll first look at how to work with the object graph. We then look at a series of particular features and use cases and how to handle them with Slick.

Configuration

Some ORMs use extensive configuration files. Slick is configured using small amounts of Scala code. You have to provide information about how to connect to the database and write or auto-generate a database-schema description if you want Slick to type-check your queries. Everything else like relationship definitionsbeyond foreign keys are ordinary Scala code, which can use familiar abstraction methods for re-use.

Mapping configuration.

The later examples use the following database schema



mapped to Slick using the following code:

```
type Person = (Int,String,Int,Int)
class People(tag: Tag) extends Table[Person](tag, "PERSON") {
  def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
  def name = column[String]("NAME")
  def age = column[Int]("AGE")
  def addressId = column[Int]("ADDRESS_ID")
  def * = (id,name,age,addressId)
  def address = foreignKey("ADDRESS",addressId,addresses)(_.id)
}
lazy val people = TableQuery[People]

type Address = (Int,String,String)
class Addresses(tag: Tag) extends Table[Address](tag, "ADDRESS") {
  def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
  def street = column[String]("STREET")
  def city = column[String]("CITY")
  def * = (id,street,city)
}
lazy val addresses = TableQuery[Addresses]
```

Tables can alternatively be mapped to case classes. Similar code can be auto-generated or hand-written.

In ORMs you often provide your mapping specification in a configuration file. In Slick you provide it as Scala types like above, which are used to type-check Slick queries. A difference is that the Slick mapping is conceptually very simple. It only describes database tables and optionally maps rows to case classes or something else using arbitrary factories and extractors. It does contain information about foreign keys, but nothing else about relationships or other patterns. These are mapped using re-usable queries fragments instead.

Navigating the object graph

Using plain old method calls

This chapter could also be called strict vs. lazy or imperative vs. declarative. One common feature ORMs provide is using a persisted object graph just as if it was in-memory. And since it is not, artifacts like members or related objects are usually loaded ad-hoc only when they are needed. To make this happen, ORMs implement or intercept method calls, which look like they happen in-memory, but instead execute database queries as needed to return the desired results. Let's look at an example using a hypothetical ORM:

```
val people: Seq[Person] = PeopleFinder.getByIds(Seq(2,99,17,234))
val addresses: Seq[Address] = people.map(_.address)
```

How many database round trips does this require? In fact reasoning about this question for different code is one of the things you need to devote the most time to when learning the collections-like API of an ORM. What usually happens is, that the ORM would do an immediate database round trip for `getByIds` and return the resulting people. Then `map` would be a Scala List method and `.map(_.address)` accesses the `address` of each person. An ORM would witness the `address` accesses one-by-one not knowing upfront that they happen in a loop. This often leads to an additional database round trip for each person, which is not ideal (n+1 problem), because database round trips are expensive. To solve the problem, ORMs often provide means to work around this, by basically telling them about the future, so they can aggregate multiple upcoming round trips into fewer more efficient ones.

```
                          // tell the ORM to load all related addresses at once
val                    people:                    Seq[Person]                    =
PeopleFinder.getByIds(Seq(2,99,17,234)).prefetch(_.address)
val addresses: Seq[Address] = people.map(_.address)
```

Here the prefetch method instructs the hypothetical ORM to load all addresses immediately with the people, often in only one or two database round trips. The addresses are then stored in a cache many ORMs maintain. The later `.map(_.address)` call could then be fully served from the cache. Of course this is redundant as you basically need to provide the mapping to addresses twice and if you forget to prefetch you will have poor performance. How you specify the pre-fetching rules depends on the ORM, often using external configuration or inline like here.

Slick works differently. To do the same in Slick you would write the following. The type annotations are optional but shown here for clarity.

```
val     peopleQuery:     Query[People,Person,Seq]     =     people.filter(_.id
inSet(Set(2,99,17,234)))
```

```
val addressesQuery: Query[Addresses,Address,Seq] =
peopleQuery.flatMap(_.address)
```

As we can see it looks very much like collection operations but the values we get are of type `Query`. They do not store results, only a plan of the operations that are needed to create a SQL query that produces the results when needed. No database round trips happen at all in our example. To actually fetch results, we can have to compile the query to a database Action with `.result` and then `run` it on the Database.

```
val addressesAction: DBIO[Seq[Address]] = addressesQuery.result
val addresses: Future[Seq[Address]] = db.run(addressesAction)
```

A single query is executed and the results returned. This makes database round trips very explicit and easy to reason about. Achieving few database round trips is easy.

As you can see with Slick we do not navigate the object graph (i.e. results) directly. We navigate it by composing queries instead, which are just place-holder values for potential database round trip yet to happen. We can lazily compose queries until they describe exactly what we need and then use a single`Database.run` call for execution.
Navigating the object graph directly in an ORM is problematic as explained earlier. Slick gets away without that feature. ORMs often solve the problem by offering a declarative query language as an alternative, which is similar to how you work with Slick.

Query languages

ORMs often come with declarative query languages like JPA's JQL or Criteria API. Similar to SQL or Slick, they allow expressing queries yet to happen and make execution explicit.

String based embeddings

Quite commonly, these languages, for example HQL, but also SQL are embedded into programs as Strings. Here is an example for HQL.

```
val hql: String = "FROM Person p WHERE p.id in (:ids)"
val q: Query = session.createQuery(hql)
q.setParameterList("ids", Array(2,99,17,234))
```

Strings are a very simple way to embed an arbitrary language and in many programming languages the only way without changing the compiler, for example in Java. While simple, this kind of embedding has significant limitations.

One issue is that tools often have no knowledge about the embedded language and treat queries as ordinary Strings. The compilers or interpreters of the host languages do not detect syntactical mistakes upfront or if the query produces a different type of result than expected. Also IDEs often do not provide syntax highlighting, code completion, inline error hints, etc.

More importantly, re-use is very hard. You would need to compose Strings in order to re-use certain parts of queries. As an exercise, try to make the id filtering part of our above HQL example re-useable, so we can use it for table person as well as address. It is really cumbersome.

In Java and many other languages, strings are the only way to embed a concise query language. As we will see in the next sections, Scala is more flexible.

Method based APIs

Instead of getting the ultimate flexibility for the embedded language, an alternative approach is to go with the extensibility features of the host language and use those. Object-oriented languages like Java and Scala allow extensibility through the definition of APIs consisting of objects and methods. JPA's Criteria API use this concept and so does Slick. This allows the host language tools to partially understand the embedded language and provide better support for the features mentioned earlier. Here is an example using Criteria Queries.

```
val id = Property.forName("id")
val q = session.createCriteria(classOf[Person])
               .add( id in Array(2,99,17,234) )
```

A method based embedding makes queries compositional. Factoring out filtering by ids becomes easy:

```
def byIds(c: Criteria, ids: Array[Int]) = c.add( id in ids )

val c = byIds(
  session.createCriteria(classOf[Person]),
  Array(2,99,17,234)
)
```

Of course ids are a trivial example, but this becomes very useful for more complex queries.

Java APIs like JPA's Criteria API do not use Scala's operator overloading capabilities. This can lead to more cumbersome and less familiar code when expressing queries. Let's query for all people younger 5 or older than 65 for example.

```
val age = Property.forName("age")
val q = session.createCriteria(classOf[Person])
               .add(
                 Restrictions.disjunction
                   .add(age lt 5)
                   .add(age gt 65)
               )
```

With Scala's operator overloading we can do better and that's what Slick uses. Queries are very concise. The same query in Slick would look like this:

```
val q = people.filter(p => p.age < 5 || p.age > 65)
```

There are some limitations to Scala's overloading capabilities that affect Slick. In queries, one has to use `===`instead of `==`, `=!=` instead of `!=` and `++` for string concatenation instead of `+`. Also it is not possible to overload `if` expressions in Scala. Instead Slick comes with a small DSL for SQL case expressions.

As already mentioned, we are working with placeholder values, merely describing the query, not executing it. Here's the same expression again with added type annotations to allow us looking behind the scenes a bit:

```
val q = (people: Query[People, Person, Seq]).filter(
  (p: People) =>
    (
      ((p.age: Rep[Int]) < 5 || p.age > 65)
      : Rep[Boolean]
    )
)
```

`Query` marks collection-like query expressions, e.g. a whole table. `People` is the Slick Table subclass defined for table person. In this context it may be confusing that the value is used rather as a prototype for a row here. It has members of type `Rep` representing the individual columns. Expressions based on these columns result in other expressions of type `Rep`. Here we are using several `Rep[Int]` to compute a `Rep[Boolean]`, which we are using as the filter expression. Internally, Slick builds a tree from this, which represents the operations and is used to produce the corresponding SQL code. We often call this process of building up expression trees encapsulated in place-holder values as lifting expressions, which is why we also call this query interface the *lifted embedding* in Slick.

It is important to note that Scala allows to be very type-safe here. E.g. Slick supports a method `.substring`for `Rep[String]` but not for `Rep[Int]`. This is impossible in Java and Java APIs like Criteria Queries, but possible in Scala using type-parameter based method extensions via implicits. This allows tools like the Scala compiler and IDEs to understand the code much more precisely and offer better checking and support.

A nice property of a Slick-like query language is, that it can be used with Scala's comprehension syntax, which is just Scala-builtin syntactic sugar for collections operations. The above example can alternatively be written as

```
for( p <- people if p.age < 5 || p.age > 65 ) yield p
```

Scala's comprehension syntax looks much like SQL or ORM query languages. It however lacks syntactic support for some constructs like sorting and grouping, for which one has to use the method-based api, e.g.

```
( for( p <- people if p.age < 5 || p.age > 65 ) yield p ).sortBy(_.name)
```

Despite the syntactic limitations, the comprehension syntax is convenient when dealing with multiple inner joins.

It is important to note that the problems of method-based query apis like Criteria Queries described above are not a conceptual limitation of ORM query languages but merely an artifact of many ORMs being Java frameworks. In principle, a Scala ORMs could offer a query language just like Slick's and they should. Comfortably compositional queries allow for a high degree of code re-use. They seem to be Slick's favorite feature for many developers.

Macro-based embeddings

Scala macros allow other approaches for embedding queries. They can be used to check queries embedded as Strings at compile time. They can also be used to translate Scala code written without Query and Rep place holder types to SQL. Both approaches are being prototyped and evaluated for Slick but are not ready for prime-time yet. There are other database libraries out there that already use macros for their query language.

Query granularity

With ORMs it is not uncommon to treat objects or complete rows as the smallest granularity when loading data. This is not necessarily a limitation of the frameworks, but a habit of using them. With Slick it is very much encouraged to only fetch the data you actually need. While you can map rows to classes with Slick, it is often more efficient to not use that feature, but to restrict your query to the data you actually need in that moment. If you only need a person's name and age, just map to those and return them as a tuple.

```
people.map(p => (p.name, p.age))
```

This allows you to be very precise about what data is actually transferred.

Read caching

Slick doesn't cache query results. Working with Slick is like working with JDBC in this regard. Many ORMs come with read and write caches. Caches are side-effects. They can be hard to reason about. It can be tricky to manage cache consistency and lifetime.

```
PeopleFinder.getById(5)
```

This call may be served from the database or from a cache. It is not clear at the call site what the performance is. With Slick it is very clear that executing a query leads to a database round trip and that Slick doesn't interfere with member accesses on objects.

```
db.run(people.filter(_.id === 5).result)
```

Slick returns a consistent, immutable snapshot of a fraction of the database at that point in time. If you need consistency over multiple queries, use transactions.

Writes (and caching)

Writes in many ORMs require write caching to be performant.

```
val person = PeopleFinder.getById(5)
person.name = "C. Vogt"
person.age = 12345
session.save
```

Here our hypothetical ORM records changes to the object and the `.save` method syncs back changes into the database in a single round trip rather than one per member. In Slick you would do the following instead:

```
val personQuery = people.filter(_.id === 5)
personQuery.map(p => (p.name,p.age)).update("C. Vogt", 12345)
```

Slick embraces declarative transformations. Rather than modifying individual members of objects one after the other, you state all modifications at once and Slick creates a single database round trip from it without using a cache. New Slick users seem to be often confused by this syntax, but it is actually very neat. Slick unifies the syntax for queries, inserts, updates and deletes. Here `personQuery` is just a query. We could use it to fetch data. But instead, we can also use it to update the columns specified by the query. Or we can use it do delete the rows.

```
personQuery.delete // deletes person with id 5
```

For inserts, we insert into the query, that resembles the whole table and can select individual columns in the same way.

```
people.map(p => (p.name,p.age)) += ("S. Zeiger", 54321)
```

Relationships

ORMs usually provide built-in, hard-coded support for 1-to-many and many-to-many relationships. They can be set up centrally in the configuration. In SQL on the other hand you would specify them using joins in every single query. You have a lot of flexibility what you join and how. With Slick you get the best of both worlds. Slick queries are as flexible as SQL, but also compositional. You can store fragements like join conditions in

central places and use language-level abstraction. Relationships of any sort are just one thing you can naturally abstract over like in any Scala code. There is no need for Slick to hard-code support for certain use cases. You can easily implement arbitrary use cases yourself, e.g. the common 1-n or n-n relationships or even relationships spanning over multiple tables, relationships with additional discriminators, polymorphic relationships, etc.

Here is an example for person and addresses.

```
implicit class PersonExtensions[C[_]](q: Query[People, Person, C]) {
  // specify mapping of relationship to address
  def withAddress = q.join(addresses).on(_.addressId === _.id)
}

val chrisQuery = people.filter(_.id === 2)
val stefanQuery = people.filter(_.id === 3)

val chrisWithAddress: Future[(Person, Address)] =
  db.run(chrisQuery.withAddress.result.head)
val stefanWithAddress: Future[(Person, Address)] =
  db.run(stefanQuery.withAddress.result.head)
```

A common question for new Slick users is how they can follow a relationships on a result. In an ORM you could do something like this:

```
val chris: Person = PeopleFinder.getById(2)
val address: Address = chris.address
```

As explained earlier, Slick does not allow navigating the object-graph as if data was in memory, because of the problem that comes with it. Instead of navigating relationships on results you write new queries instead.

```
val chrisQuery: Query[People,Person,Seq] = people.filter(_.id === 2)
val           addressQuery:           Query[Addresses,Address,Seq]           =
chrisQuery.withAddress.map(_._2)
val address = db.run(addressQuery.result.head)
```

If you leave out the optional type annotation and some intermediate vals it is very clean. And it is very clear where database round trips happen.

A variant of this question Slick new comers often ask is how they can do something like this in Slick:

```
case class Address( … )
case class Person( …, address: Address )
```

The problem is that this hard-codes that a Person requires an Address. It can not be loaded without it. This does't fit to Slick's philosophy of giving you fine-grained control

over what you load exactly. With Slick it is advised to map one table to a tuple or case class without them having object references to related objects. Instead you can write a function that joins two tables and returns them as a tuple or association case class instance, providing an association externally, not strongly tied one of the classes.

```
val tupledJoin: Query[(People,Addresses),(Person,Address), Seq]
      = people join addresses on (_.addressId === _.id)

case class PersonWithAddress(person: Person, address: Address)
val                          caseClassJoinResults                          =
db.run(tupledJoin.result).map(_.map(PersonWithAddress.tupled))
```

An alternative approach is giving your classes Option-typed members referring to related objects, where None means that the related objects have not been loaded yet. However this is less type-safe then using a tuple or case class, because it cannot be statically checked, if the related object is loaded.

Modifying relationships

When manipulating relationships with ORMs you usually work on mutable collections of associated objects and inserts or remove related objects. Changes are written to the database immediately or recorded in a write cache and commited later. To avoid stateful caches and mutability, Slick handles relationship manipulations just like SQL - using foreign keys. Changing relationships means updating foreign key fields to new ids, just like updating any other field. As a bonus this allows establishing and removing associations with objects that have not been loaded into memory. Having their ids is sufficient.

Inheritance

Slick does not persist arbitrary object-graphs. It rather exposes the relational data model nicely integrated into Scala. As the relational schema doesn't contain inheritance so doesn't Slick. This can be unfamiliar at first. Usually inheritance can be simply replaced by relationalships thinking along the lines of roles. Instead of foo is a bar think foo has role bar. As Slick allows query composition and abstraction, inheritance-like query-snippets can be easily implemented and put into functions for re-use. Slick doesn't provide any out of the box but allows you to flexibly come up with the ones that match your problem and use them in your queries.

Code-generation

Many of the concepts described above can be abstracted over using Scala code to avoid repetition. There cases however, where you reach the limits of Scala's type system's abstraction capabilities. Code generation offers a solution to this. Slick comes with a very flexible and fully customizable code generator, which can be used to avoid

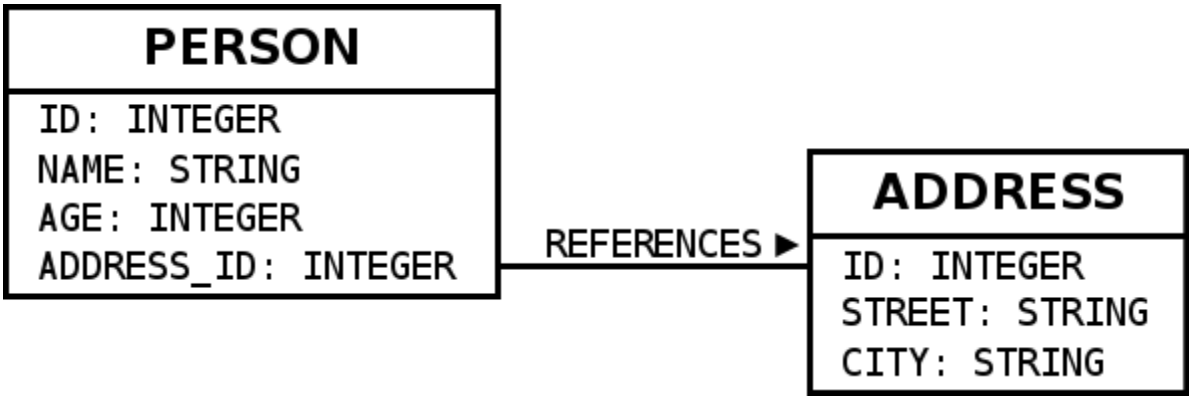
repetition in these cases. The code generator operates on the meta data of the database. Combine it with your own extra meta data if needed and use it to generate Slick types, relationship accessors, association classes, etc. For more info see our Scala Days 2014 talk athttp://slick.typesafe.com/docs/ .

COMING FROM SQL TO SLICK

Coming from JDBC/SQL to Slick is pretty straight forward in many ways. Slick can be considered as a drop-in replacement with a nicer API for handling connections, fetching results and using a query language, which is integrated more nicely into Scala than writing queries as Strings. The main obstacle for developers coming from SQL to Slick seems to be the semantic differences of seemingly similar operations between SQL and Scala's collections API which Slick's API imitates. The following sections give a quick overview over the differences. They start with conceptual differences and then list examples of many SQL operators and their Slick equivalents. For a more detailed explanations of Slick's API please refer to chapter queries and the equivalent methods in the the Scala collections API.

Schema

The later examples use the following database schema



mapped to Slick using the following code:

```
type Person = (Int,String,Int,Int)
class People(tag: Tag) extends Table[Person](tag, "PERSON") {
  def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
  def name = column[String]("NAME")
  def age = column[Int]("AGE")
  def addressId = column[Int]("ADDRESS_ID")
  def * = (id,name,age,addressId)
  def address = foreignKey("ADDRESS",addressId,addresses)(_.id)
}
lazy val people = TableQuery[People]

type Address = (Int,String,String)
```

```
class Addresses(tag: Tag) extends Table[Address](tag, "ADDRESS") {
  def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
  def street = column[String]("STREET")
  def city = column[String]("CITY")
  def * = (id,street,city)
}
lazy val addresses = TableQuery[Addresses]
```

Tables can alternatively be mapped to case classes. Similar code can be auto-generated or hand-written.

Queries in comparison

JDBC Query

A JDBC query with error handling could look like this:

```
import java.sql._

Class.forName("org.h2.Driver")
val conn = DriverManager.getConnection("jdbc:h2:mem:test1")
val people = new scala.collection.mutable.MutableList[(Int,String,Int)]()
try{
  val stmt = conn.createStatement()
  try{

    val rs = stmt.executeQuery("select ID, NAME, AGE from PERSON")
    try{
      while(rs.next()){
        people += ((rs.getInt(1), rs.getString(2), rs.getInt(3)))
      }
    }finally{
      rs.close()
    }

  }finally{
    stmt.close()
  }
}finally{
  conn.close()
}
```

Slick gives us two choices how to write queries. One is SQL strings just like JDBC. The other are type-safe, composable queries.

Slick Plain SQL queries

This is useful if you either want to continue writing queries in SQL or if you need a feature not (yet) supported by Slick otherwise. Executing the same query using Slick Plain SQL, which has built-in error-handling and resource management optimized for asynchronous execution, looks like this:

```
import slick.driver.H2Driver.api._

val db = Database.forConfig("h2mem1")

val action = sql"select ID, NAME, AGE from PERSON".as[(Int,String,Int)]
db.run(action)
```

`.list` returns a list of results. `.first` a single result. `.foreach` can be used to iterate over the results without ever materializing all results at once.

Slick type-safe, composable queries

Slick's key feature are type-safe, composable queries. Slick comes with a Scala-to-SQL compiler, which allows a (purely functional) sub-set of the Scala language to be compiled to SQL queries. Also available are a subset of the standard library and some extensions, e.g. for joins. The familiarity allows Scala developers to instantly write many queries against all supported relational databases with little learning required and without knowing SQL or remembering the particular dialect. Such Slick queries are composable, which means that you can write and re-use fragments and functions to avoid repetitive code like join conditions in a much more practical way than concatenating SQL strings. The fact that such queries are type-safe not only catches many mistakes early at compile time, but also eliminates the risk of SQL injection vulnerabilities.

The same query written as a type-safe Slick query looks like this:

```
import slick.driver.H2Driver.api._

val db = Database.forConfig("h2mem1")

val query = people.map(p => (p.id,p.name,p.age))
db.run(query.result)
```

`.run` automatically returns a Seq for collection-like queries and a single value for scalar queries. `.list`,`.first` and `.foreach` are also available.

A key benefit compared to SQL strings is, that you can easily transform the query by calling more methods on it. E.g. `query.filter(_.age > 18)` returns transformed query which further restricts the results. This allows to build libraries of queries, which re-use each other become much more maintainable. You can abstract over join conditions, pagination, filters, etc.

It is important to note that Slick needs the type-information to type-check these queries. This type information closely corresponds to the database schema and is provided to Slick in the form of Table sub classes and TableQuery values shown above.

Main obstacle: Semantic API differences

Some methods of the Scala collections work a bit differently than their SQL counter parts. This seems to be one of the main causes of confusion for people newly coming from SQL to Slick. Especially groupBy seems to be tricky.
The best approach to write queries using Slick's type-safe api is thinking in terms of Scala collections. What would the code be if you had a Seq of tuples or case classes instead of a Slick TableQuery object. Use that exact code. If needed adapt it with workarounds where a Scala library feature is currently not supported by Slick or if Slick is slightly different. Some operations are more strongly typed in Slick than in Scala for example. Arithmetic operation in different types require explicit casts using `.asColumnOf[T]`. Also Slick uses 3-valued logic for Option inference.

Scala-to-SQL compilation during runtime

Slick runs a Scala-to-SQL compiler to implement its type-safe query feature. The compiler runs at Scala run-time and it does take its time which can even go up to second or longer for complex queries. It can be very useful to run the compiler only once per defined query and upfront, e.g. at app startup instead of each execution over and over. Compiled queries allow you to cache the generated SQL for re-use.

Limitations

When you use Slick extensively you will run into cases, where Slick's type-safe query language does not support a query operator or JDBC feature you may desire to use or produces non-optimal SQL code. There are several ways to deal with that.

Missing query operators

Slick is extensible to some degree, which means you can add some kinds of missing operators yourself.

Definition in terms of others

If the operator you desire is expressible using existing Slick operations you can simply write a Scala function or implicit class that implements the operator as a method in terms of existing operators. Here we implement `squared` using multiplication.

```
implicit class MyStringColumnExtensions(i: Rep[Int]){
  def squared = i * i
}

// usage:
people.map(p => p.age.squared)
```

Definition using a database function

If you need a fundamental operator, which is not supported out-of-the-box you can add it yourself if it operates on scalar values. For example Slick currently does not have a `power` method out of the box. Here we are mapping it to a database function.

```
val power = SimpleFunction.binary[Int,Int,Int]("POWER")

// usage:
people.map(p => power(p.age,2))
```

More information can be found in the chapter about Scalar database functions.

You can however not add operators operating on queries using database functions. The Slick Scala-to-SQL compiler requires knowledge about the structure of the query in order to compile it to the most simple SQL query it can produce. It currently couldn't handle custom query operators in that context. (There are some ideas how this restriction can be somewhat lifted in the future, but it needs more investigation). An example for such operator is a MySQL index hint, which is not supported by Slick's type-safe api and it cannot be added by users. If you require such an operator you have to write your whole query using Plain SQL. If the operator does not change the return type of the query you could alternatively use the workaround described in the following section.

Non-optimal SQL code

Slick generates SQL code and tries to make it as simple as possible. The algorithm doing that is not perfect and under continuous improvement. There are cases where the generated queries are more complicated than someone would write them by hand. This can lead to bad performance for certain queries with some optimizers and DBMS. For example, Slick occasionally generates unnecessary sub-queries. In MySQL <= 5.5 this easily leads to unnecessary table scans or indices not being used. The Slick team is working towards generating code better factored to what the query optimizers can currently optimize, but that doesn't help you now. To work around it you have to write the more optimal SQL code by hand. You can either run it as a Slick Plain SQL query or you can use a hack, which allows you to simply swap out the SQL code Slick uses for a type-safe query.

```
people.map(p => (p.id,p.name,p.age))
      .result
      //           inject           hand-written           SQL,           see
https://gist.github.com/cvogt/d9049c63fc395654c4b4
      .overrideSql("SELECT id, name, age FROM Person")
```

SQL vs. Slick examples

This section shows an overview over the most important types of SQL queries and a corresponding type-safe Slick query.

SELECT *

SQL

```
sql"select * from PERSON".as[Person]
```

Slick

The Slick equivalent of `SELECT *` is the `result` of the plain TableQuery:

```
people.result
```

SELECT

SQL

```
sql"""
  select AGE, concat(concat(concat(NAME,' ('),ID),')')
  from PERSON
""".as[(Int,String)]
```

Slick

Scala's equivalent for `SELECT` is `map`. Columns can be referenced similarly and functions operating on columns can be accessed using their Scala eqivalents (but allowing only `++` for String concatenation, not `+`).

```
people.map(p => (p.age, p.name ++ " (" ++ p.id.asColumnOf[String] ++ ")")).result
```

WHERE

SQL

```
sql"select * from PERSON where AGE >= 18 AND NAME = 'C. Vogt'".as[Person]
```

Slick

Scala's equivalent for `WHERE` is `filter`. Make sure to use `===` instead of `==` for comparison.

```
people.filter(p => p.age >= 18 && p.name === "C. Vogt").result
```

ORDER BY

SQL

```
sql"select * from PERSON order by AGE asc, NAME".as[Person]
```

Slick

Scala's equivalent for `ORDER BY` is `sortBy`. Provide a tuple to sort by multiple columns. Slick's `.asc` and `.desc` methods allow to affect the ordering. Be aware that a single `ORDER BY` with multiple columns is not equivalent to multiple `.sortBy` calls but to a single `.sortBy` call passing a tuple.

```
people.sortBy(p => (p.age.asc, p.name)).result
```

Aggregations (max, etc.)

SQL

```
sql"select max(AGE) from PERSON".as[Option[Int]].head
```

Slick

Aggregations are collection methods in Scala. In SQL they are called on a column, but in Slick they are called on a collection-like value e.g. a complete query, which people

coming from SQL easily trip over. They return a scalar value, which can be run individually. Aggregation methods such as `max` that can return `NULL` return Options in Slick.

```
people.map(_.age).max.result
```

GROUP BY

People coming from SQL often seem to have trouble understanding Scala's and Slick's `groupBy`, because of the different signatures involved. SQL's `GROUP BY` can be seen as an operation that turns all columns that weren't part of the grouping key into collections of all the elements in a group. SQL requires the use of it's aggregation operations like `avg` to compute single values out of these collections.

SQL

```
sql"""
  select ADDRESS_ID, AVG(AGE)
  from PERSON
  group by ADDRESS_ID
""".as[(Int,Option[Int])]
```

Slick

Scala's groupBy returns a Map of grouping keys to Lists of the rows for each group. There is no automatic conversion of individual columns into collections. This has to be done explicitly in Scala, by mapping from the group to the desired column, which then allows SQL-like aggregation.

```
people.groupBy(p => p.addressId)
       .map{ case (addressId, group) => (addressId, group.map(_.age).avg) }
       .result
```

SQL requires to aggregate grouped values. We require the same in Slick for now. This means a `groupBy` call must be followed by a `map` call or will fail with an Exception. This makes Slick's grouping syntax a bit more complicated than SQL's.

HAVING

SQL

```
sql"""
  select ADDRESS_ID
  from PERSON
  group by ADDRESS_ID
  having avg(AGE) > 50
""".as[Int]
```

Slick

Slick does not have different methods for `WHERE` and `HAVING`. For achieving semantics equivalent to `HAVING`, just use `filter` after `groupBy` and the following `map`.

```
people.groupBy(p => p.addressId)
```

```
      .map{ case (addressId, group) => (addressId, group.map(_.age).avg) }
      .filter{ case (addressId, avgAge) => avgAge > 50 }
      .map(_._1)
      .result
```

Implicit inner joins

SQL

```
sql"""
  select P.NAME, A.CITY
  from PERSON P, ADDRESS A
  where P.ADDRESS_ID = a.id
""".as[(String,String)]
```

Slick

Slick generates SQL using implicit joins for `flatMap` and `map` or the corresponding for-expression syntax.

```
people.flatMap(p =>
  addresses.filter(a => p.addressId === a.id)
           .map(a => (p.name, a.city))
).result

// or equivalent for-expression:
(for(p <- people;
     a <- addresses if p.addressId === a.id
 ) yield (p.name, a.city)
).result
```

Explicit inner joins

SQL

```
sql"""
  select P.NAME, A.CITY
  from PERSON P
  join ADDRESS A on P.ADDRESS_ID = a.id
""".as[(String,String)]
```

Slick

Slick offers a small DSL for explicit joins.

```
(people join addresses on (_.addressId === _.id))
  .map{ case (p, a) => (p.name, a.city) }.result
```

Outer joins (left/right/full)

SQL

```
sql"""
  select P.NAME,A.CITY
  from ADDRESS A
  left join PERSON P on P.ADDRESS_ID = a.id
```

```
""".as[(Option[String],String)]
```

Slick

Outer joins are done using Slick's explicit join DSL. Be aware that in case of an outer join SQL changes the type of outer joined, non-nullable columns into nullable columns. In order to represent this in a clean way even in the presence of mapped types, Slick lifts the whole side of the join into an `Option`. This goes a bit further than the SQL semantics because it allows you to distinguish a row which was not matched in the join from a row that was matched but already contained nothign but NULL values.

```
(addresses joinLeft people on (_.id === _.addressId))
  .map{ case (a, p) => (p.map(_.name), a.city) }.result
```

Subquery

SQL

```
sql"""
  select *
  from PERSON P
  where P.ID in (select ID
                 from ADDRESS
                 where CITY = 'New York City')
""".as[Person]
```

Slick

Slick queries are composable. Subqueries can be simply composed, where the types work out, just like any other Scala code.

```
val address_ids = addresses.filter(_.city === "New York City").map(_.id)
people.filter(_.id in address_ids).result // <- run as one query
```

The method `.in` expects a sub query. For an in-memory Scala collection, the method `.inSet` can be used instead.

Scalar value subquery / custom function

SQL

```
sql"""
  select * from PERSON P,
                    (select rand() * MAX(ID) as ID from PERSON) RAND_ID
  where P.ID >= RAND_ID.ID
  order by P.ID asc
  limit 1
""".as[Person].head
```

Slick

This code shows a subquery computing a single value in combination with a user-defined database function.

```
val rand = SimpleFunction.nullary[Double]("RAND")
```

```
val rndId = (people.map(_.id).max.asColumnOf[Double] * rand).asColumnOf[Int]

people.filter(_.id >= rndId)
       .sortBy(_.id)
       .result.head
```

insert

SQL

```
sqlu"""
  insert into PERSON (NAME, AGE, ADDRESS_ID) values ('M Odersky', 12345, 1)
"""
```

Slick

Inserts can be a bit surprising at first, when coming from SQL, because unlike SQL, Slick re-uses the same syntax that is used for querying to select which columns should be inserted into. So basically, you first write a query and instead of creating an Action that gets the result of this query, you call `+=` on with value to be inserted, which gives you an Action that performs the insert. `++=` allows insertion of a Seq of rows at once. Columns that are auto-incremented are automatically ignored, so inserting into them has no effect. Using`forceInsert` allows actual insertion into auto-incremented columns.

```
people.map(p => (p.name, p.age, p.addressId)) += ("M Odersky",12345,1)
```

update

SQL

```
sqlu"""
  update PERSON set NAME='M. Odersky', AGE=54321 where NAME='M Odersky'
"""
```

Slick

Just like inserts, updates are based on queries that select and filter what should be updated and instead of running the query and fetching the data `.update` is used to replace it.

```
people.filter(_.name === "M Odersky")
       .map(p => (p.name,p.age))
       .update(("M. Odersky",54321))
```

delete

SQL

```
sqlu"""
  delete PERSON where NAME='M. Odersky'
"""
```

Slick

Just like inserts, deletes are based on queries that filter what should be deleted. Instead of getting the query result of the query, `.delete` is used to obtain an Action that deletes the selected rows.

```
people.filter(p => p.name === "M. Odersky")
       .delete
```

CASE

SQL

```
sql"""
  select
    case
      when ADDRESS_ID = 1 then 'A'
      when ADDRESS_ID = 2 then 'B'
    end
  from PERSON P
""".as[Option[String]]
```

Slick

Slick uses a small DSL to allow `CASE` like case distinctions.

```
people.map(p =>
  Case
    If(p.addressId === 1) Then "A"
    If(p.addressId === 2) Then "B"
).result
```

UPGRADE GUIDES

Compatibility Policy

Slick requires Scala 2.10 or 2.11. (For Scala 2.9 please use ScalaQuery, the predecessor of Slick).

Slick version numbers consist of an epoch, a major and minor version, and possibly a qualifier (for milestone, RC and SNAPSHOT versions).

For release versions (i.e. versions without a qualifier), backward binary compatibility is guaranteed between releases with the same epoch and major version (e.g. you could use 2.1.2 as a drop-in relacement for 2.1.0 but not for 2.0.0). Slick Extensions requires at least the same minor version of Slick (e.g. Slick Extensions 2.1.2 can be used with Slick 2.1.2 but not with Slick 2.1.1). Binary compatibility is not preserved for *slick-codegen*, which is generally used at compile-time.

We do not guarantee source compatibility but we try to preserve it within the same major release. Upgrading to a new major release may require some changes to your sources. We generally deprecate old features and keep them around for a full major release cycle

(i.e. features which become deprecated in 2.1.0 will not be removed before 2.2.0) but this is not possible for all kinds of changes.

Release candidates have the same compatibility guarantees as the final versions to which they lead. There are *no compatibility guarantees* whatsoever for milestones and snapshots.

Upgrade from 3.0 to 3.1

This section describes the changes that are needed when upgrading from Slick 3.0 to 3.1. If you are currently using an older version of Slick, please see the older Slick Manuals for details on other changes that may be required.

Deprecations

Most deprecated features from 3.0, including the old `Invoker` and `Executor` APIs and the package aliases for `scala.slick` were removed.

HikariCP

The HikariCP support for Slick was factored out into its own module with a non-optional dependency on HikariCP itself. This makes it easier to use the correct version of HikariCP (which does not have a well-defined binary compatibility policy) with Slick. See the section on dependencies for more information.

Due to packaging constraints imposed by OSGi, HikariCPJdbcDataSource was moved from package `slick.jdbc` to `slick.jdbc.hikaricp`.

Counting Option columns

Counting collection-valued queries with `.length` now ignores nullability of the columns, i.e. it is equivalent to `COUNT(*)` in SQL, no matter what is being counted. The previous approach of picking a random column led to inconsistent results. This is particularly relevant when you try to count one side of an outer join. Up to Slick 3.0 the goal (although not achieved in all cases due to a design problem) was not to include non-matching rows in the total (equivalent to counting only the discriminator column). This does not make sense anymore for the new outer join operators (introduced in 3.0) with correct `Option` types. The new semantics are identical to those of Scala collections.

There is a new operator `.countDefined` for counting only the defined / matching (i.e. non-NULL in SQL) rows. To avoid any ambiguities in the definition, it is only available for collection-valued queries of a single column with an `Option` type.

Default String type on MySQL

Slick 3.0 changed the default string type for MySQL to *TEXT*, which is not allowed for primary keys and columns with default values. In these cases we now fall back to the old *VARCHAR(254)* type which was used up to Slick 2.1. Like in 3.0 you can change this default by setting the application.conf key *slick.driver.MySQL.defaultStringType*.

Default String type on SQL Server

SQLServerDriver (part of Slick Extensions) used *VARCHAR(MAX)* as the default type for strings without a size limit in 3.0. This type cannot be used for primary keys, so we now use *VARCHAR(254)* instead if a column has the *PrimaryKey* option set. Like for MySQL (see previous paragraph), the default type can be overridden by setting *slick.driver.SQLServer.defaultStringType* in application.conf.

SLICK EXTENSIONS

Slick Extensions, a closed-source package with commercial support provided by Typesafe, Inc contains Slick drivers for:

- Oracle (`com.typesafe.slick.driver.oracle.OracleDriver`)
- IBM DB2 (`com.typesafe.slick.driver.db2.DB2Driver`)
- Microsoft SQL Server (`com.typesafe.slick.driver.ms.SQLServerDriver`)

Note

You may use it for development and testing purposes under the terms and conditions of the Typesafe Subscription Agreement (PDF). Production use requires a Typesafe Subscription.

If you are using sbt, you can add *slick-extensions* and the Typesafe repository (which contains the required artifacts) to your build definition like this:

```
libraryDependencies += "com.typesafe.slick" %% "slick-extensions" %
"3.1.0"


resolvers          +=          "Typesafe          Releases"          at
"http://repo.typesafe.com/typesafe/maven-releases/"
```

SLICK TESTKIT

Note

This chapter is based on the Slick TestKit Example template. The prefered way of reading this introduction is in Activator, where you can edit and run the code directly while reading the tutorial.

When you write your own database driver for Slick, you need a way to run all the standard unit tests on it (in addition to any custom tests you may want to add) to ensure that it works correctly and does not claim to support any capabilities which are not actually implemented. For this purpose the Slick unit tests have been factored out into a separate Slick TestKit project.

To get started, you can clone the Slick TestKit Example template which contains a copy of Slick's standard PostgreSQL driver and all the infrastructure required to build and test it.

Scaffolding

Its `build.sbt` file is straight-forward. Apart from the usual name and version settings, it adds the dependencies for Slick, the TestKit, junit-interface, Logback and the PostgreSQL JDBC driver, and it sets some options for the test runs:

```
libraryDependencies ++= Seq(
  "com.typesafe.slick" %% "slick" % "3.1.0",
  "com.typesafe.slick" %% "slick-testkit" % "3.1.0" % "test",
  "com.novocode" % "junit-interface" % "0.10" % "test",
  "ch.qos.logback" % "logback-classic" % "0.9.28" % "test",
  "postgresql" % "postgresql" % "9.1-901.jdbc4" % "test"
)

testOptions += Tests.Argument(TestFrameworks.JUnit, "-q", "-v", "-s", "-a")

parallelExecution in Test := false

logBuffered := false
```

There is a copy of Slick's logback configuration in `src/test/resources/logback-test.xml` but you can swap out the logging framework if you prefer a different one.

Driver

The actual driver implementation can be found under `src/main/scala`.

Test Harness

In order to run the TestKit tests, you need to add a class that extends `DriverTest`, plus an implementation of `TestDB` which tells the TestKit how to connect to a test database, get a list of tables, clean up between tests, etc.

In the case of the PostgreSQL test harness (in `src/test/slick/driver/test/MyPostgresTest.scala`) most of the default implementations can be used out of the box. Only `localTables` and `getLocalSequences` require custom implementations. We also

modify the driver's `capabilities` to indicate that our driver does not support the JDBC `getFunctions` call:

```
@RunWith(classOf[Testkit])
class MyPostgresTest extends DriverTest(MyPostgresTest.tdb)

object MyPostgresTest {
  def tdb = new ExternalJdbcTestDB("mypostgres") {
    val driver = MyPostgresDriver
    override def localTables(implicit ec: ExecutionContext):
DBIO[Vector[String]] =
      ResultSetAction[(String,String,String,
String)](_.conn.getMetaData().getTables("", "public", null, null)).map { ts =>
        ts.filter(_._4.toUpperCase == "TABLE").map(_._3).sorted
      }
    override def getLocalSequences(implicit session: profile.Backend#Session) =
{
      val tables = ResultSetInvoker[(String,String,String,
String)](_.conn.getMetaData().getTables("", "public", null, null))
      tables.buildColl[List].filter(_._4.toUpperCase ==
"SEQUENCE").map(_._3).sorted
    }
    override def capabilities = super.capabilities -
TestDB.capabilities.jdbcMetaGetFunctions
  }
}
```

The name of a configuration prefix, in this case `mypostgres`, is passed to `ExternalJdbcTestDB`:

```
def tdb =
  new ExternalJdbcTestDB("mypostgres") ...
```

Database Configuration

Since the PostgreSQL test harness is based on `ExternalJdbcTestDB`, it needs to be configured in `test-dbs/testkit.conf`:

```
mypostgres.enabled = true
mypostgres.user = myuser
mypostgres.password = secret
```

There are several other configuration options that need to be set for an `ExternalJdbcTestDB`. These are defined with suitable defaults in `testkit-reference.conf` so that `testkit.conf` can be kept very simple in most cases.

Testing

Running `sbt test` discovers `MyPostgresTest` and runs it with TestKit's JUnit runner. This in turn causes the database to be set up through the test harness and all tests which are

applicable for the driver (as determined by the `capabilities` setting in the test harness) to be run.