Essential Slick

Richard Dallaway and Jonathan Ferguson

underscore

Essential Slick

Published by Underscore Consulting LLP, Brighton, UK.

Copies of this, and related topics, can be found at http://underscore.io/training.

Team discounts, when available, may also be found at that address.

Contact the author regarding this text at: hello@underscore.io.

Contents

Preface

Slick is a Scala library for working with relational databases. That means it allows you to model a schema, run queries, insert data, and update data.

You write queries in Scala and they are type checked by the compiler. This makes working with a database like working with regular Scala collections.

We've seen that developers using Slick for the first time often need help getting the most from it. For example, key concepts that need to be known include:

- *queries*: which compose using combinators such as `map`, `flatMap`, and `filter`;
- *actions*: the things you can run against a database, which themselves compose; and
- *futures*: which are the result of actions, and also support a set of combinators.

We've produced *Essential Slick* as a guide for those who want to get started using Slick. This material is aimed at beginner-to-intermediate Scala developers. You need:

- a working knowledge of Scala (we recommend Essential Scala or an equivalent book);
- experience with relational databases (familiarity with concepts such as rows, columns, joins, indexes, SQL); and
- an installed JDK 8 or better, along with a programmer's text editor or IDE (Scala IDE for Eclipse or IntelliJ are both good choices).

The material presented focuses on Slick version 3. Examples use H2 as the relational database.

How to Contact Us

You can provide feedback on this text via:

- our Gitter channel; or
- email to hello@underscore.io using the subject line of "Essential Slick".

The Underscore Newsletter contains announcements regarding this and other publications from Underscore.

You can follow us on Twitter as @underscoreio.

Acknowledgements

Many thanks to Renato Cavalcanti, Dave Gurnell, Kevin Meredith, Joseph Ottinger, Yann Simon and the team at Underscore for their invaluable contributions and proof reading.

Conventions Used in This Book

This book contains a lot of technical information and program code. We use the following typographical conventions to reduce ambiguity and highlight important concepts:

Typographical Conventions

New terms and phrases are introduced in *italics*. After their initial introduction they are written in normal roman font.

Terms from program code, filenames, and file contents, are written in `monospace font`. Note that we do not distinguish between singular and plural forms. For example, might write `String` or `Strings` to refer to the `java.util.String` class or objects of that type.

References to external resources are written as hyperlinks. References to API documentation are written using a combination of hyperlinks and monospace font, for example: `scala.Option`.

Source Code

Source code blocks are written as follows. Syntax is highlighted appropriately where applicable:

```
object MyApp extends App {
  println("Hello world!") // Print a fine message to the user!
}
```

Some lines of program code are too wide to fit on the page. In these cases we use a *continuation character* (curly arrow) to indicate that longer code should all be written on one line. For example, the following code:

```
println("This code should all be written ↩
  on one line.")
```

should actually be written as follows:

```
println("This code should all be written on one line.")
```

Callout Boxes

We use three types of *callout box* to highlight particular content:

> **Tip**
>
> Tip callouts indicate handy summaries, recipes, or best practices.

> **Advanced**
>
> Advanced callouts provide additional information on corner cases or underlying mechanisms. Feel free to skip these on your first read-through—come back to them later for extra information.

Warning

Warning callouts indicate common pitfalls and gotchas. Make sure you read these to avoid problems, and come back to them if you're having trouble getting your code to run.

Chapter 1

Basics

1.1 Orientation

Slick is a Scala library for accessing relational databases using an interface similar to the Scala collections library. You can treat queries like collections, transforming and combining them with methods like `map`, `flatMap`, and `filter` before sending them to the database to fetch results. This is how we'll be working with Slick for the majority of this text.

Standard Slick queries are written in plain Scala. These are *type safe* expressions that benefit from compile time error checking. They also *compose*, allowing us to build complex queries from simple fragments before running them against the database. If writing queries in Scala isn't your style, you'll be pleased to know that Slick also supports *plain SQL queries* that allow you to write SQL.

In addition to querying, Slick helps you with all the usual trappings of relational database, including connecting to a database, creating a schema, setting up transactions, and so on. You can even drop down below Slick to deal with JDBC (Java Database Connectivity) directly, if that's something you're familiar with and find you need.

This book provides a compact, no-nonsense guide to everything you need to know to use Slick in a commercial setting:

- Chapter 1 provides an abbreviated overview of the library as a whole, demonstrating the fundamentals of data modelling, connecting to the database, and running queries.
- Chapter 2 covers basic select queries, introducing Slick's query language and delving into some of the details of type inference and type checking.
- Chapter 3 covers queries for inserting, updating, and deleting data.
- Chapter 4 discusses data modelling, including defining custom column and table types.
- Chapter 5 looks at actions and how you combine multiple actions together.
- Chapter 6 explores advanced select queries, including joins and aggregates.
- Chapter 7 provides a brief overview of *Plain SQL* queries—a useful tool when you need fine control over the SQL sent to your database.

> **Tip**
>
> **Slick isn't an ORM**
>
> If you're familiar with other database libraries such as Hibernate or Active Record, you might expect Slick to be an *Object-Relational Mapping (ORM)* tool. It is not, and it's best not to think of Slick in this way.
>
> ORMs attempt to map object oriented data models onto relational database backends. By contrast, Slick provides a more database-like set of tools such as queries, rows and columns. We're not going to argue

> the pros and cons of ORMs here, but if this is an area that interests you, take a look at the Coming from ORM to Slick article in the Slick manual.
>
> If you aren't familiar with ORMs, congratulations. You already have one less thing to worry about!

1.2 Running the Examples and Exercises

The aim of this first chapter is to provide a high-level overview of the core concepts involved in Slick, and get you up and running with a simple end-to-end example. You can grab this example now by cloning the Git repo of exercises for this book:

```
bash$ git clone git@github.com:underscoreio/essential-slick-code.git
Cloning into 'essential-slick-code'...

bash$ cd essential-slick-code

bash$ ls -1
README.md
chapter-01
chapter-02
chapter-03
chapter-04
chapter-05
chapter-06
chapter-07
```

Each chapter of the book is associated with a separate sbt project that provides a combination of examples and exercises. We've bundled everything you need to run sbt in the directory for each chapter.

We'll be using a running example of a chat application similar to *Slack*, *Gitter*, or *IRC*. The app will grow and evolve as we proceed through the book. By the end it will have users, messages, and rooms, all modelled using tables, relationships, and queries.

For now, we will start with a simple conversation between two famous celebrities. Change to the `chapter-01` directory now, use the `sbt.sh` script to start sbt, and compile and run the example to see what happens:

```
bash$ cd chapter-01

bash$ ./sbt.sh
# sbt log messages...

> compile
# More sbt log messages...

> run
Creating database table

Inserting test data

Selecting all messages:
Message("Dave","Hello, HAL. Do you read me, HAL?",1)
Message("HAL","Affirmative, Dave. I read you.",2)
Message("Dave","Open the pod bay doors, HAL.",3)
```

```
Message("HAL","I'm sorry, Dave. I'm afraid I can't do that.",4)

Selecting only messages from HAL:
Message("HAL","Affirmative, Dave. I read you.",2)
Message("HAL","I'm sorry, Dave. I'm afraid I can't do that.",4)
```

If you get output similar to the above, congratulations! You're all set up and ready to run with the examples and exercises throughout the rest of this book. If you encounter any errors, let us know on our Gitter channel and we'll do what we can to help out.

> **Tip**
>
> **New to sbt?**
>
> The first time you run sbt, it will download a lot of library dependencies from the Internet and cache them on your hard drive. This means two things:
>
> - you need a working Internet connection to get started; and
> - the first `compile` command you issue could take a while to complete.
>
> If you haven't used sbt before, you may find the sbt Tutorial useful.

1.3 Working Interactively in the sbt Console

To get you up to speed quickly, we've created an `exec` method and imported the base requirements to run examples from the console. You can see this by starting `sbt` and then running the `console` command. Which will give output similar to:

```
> console
[info] Compiling 1 Scala source to /Users/jonoabroad/developer/books/ ↩
essential-slick-code/chapter-01/target/scala-2.11/classes...
[info] Starting scala interpreter...
[info]
import slick.driver.H2Driver.api._
import Example._
import scala.concurrent.duration._
import scala.concurrent.Await
import scala.concurrent.ExecutionContext.Implicits.global
db: slick.driver.H2Driver.backend.Database =
      slick.jdbc.JdbcBackend$DatabaseDef@75028b56                    ↩
exec: [T](program: slick.driver.H2Driver.api.DBIO[T])T
res0: Option[Int] = Some(4)
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM,   ↩
Java 1.8.0_25).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

This means we can focus on Slick, rather than the boilerplate. There is a complete explanation of `exec` later in the chapter. For now, a small example showing its usuage and output:

```
scala> exec(messages.result)
//res1: Seq[Example.MessageTable#TableElementType] =
//Vector(Message(Dave,Hello, HAL. Do you read me, HAL?,1),
//       Message(HAL,Affirmative, Dave. I read you.,2),
//       Message(Dave,Open the pod bay doors, HAL.,3),
//       Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,4))
```

Note: We reference the `messages` table and due to the import in the previous code snippet there was no reason to qualify it with as `Example.messages`.

1.4 Example: A Sequel Odyssey

The test application we saw above creates an in-memory database using H2, creates a single table, populates it with test data, and then runs some example queries. The rest of this section will walk you through the code and provide an overview of things to come. We'll reproduce the essential parts of the code in the text, but you can follow along in the codebase for the exercises as well.

> **Advanced**
>
> **Choice of Database**
>
> All of the examples in this book use the H2 database. H2 is written in Java and runs in-process beside our application code. We've picked H2 because it allows us to forego any system administration and skip to writing Scala.
>
> You might prefer to use *MySQL*, *PostgreSQL*, or some other database—and you can. In Appendix A we point you at the changes you'll need to make to work with other databases. However, we recommend sticking with H2 for at least this first chapter so you can build confidence using Slick without running into database-specific complications.

1.4.1 Library Dependencies

Before diving into Scala code, let's look at the sbt configuration. You'll find this in `build.sbt` in the example:

```
name := "essential-slick-chapter-01"

version := "1.0.0"

scalaVersion := "2.11.6"

libraryDependencies ++= Seq(
  "com.typesafe.slick" %% "slick"          % "3.1.0",
  "com.h2database"      % "h2"             % "1.4.185",
  "ch.qos.logback"      % "logback-classic" % "1.1.2"
)
```

This file declares the minimum library dependencies for a Slick project:

- Slick itself;
- the H2 database; and

- a logging library.

If we were using a separate database like MySQL or PostgreSQL, we would substitute the H2 dependency for the JDBC driver for that database.

1.4.2 Importing Library Code

Database management systems are not created equal. Different systems support different data types, different dialects of SQL, and different querying capabilities. To model these capabilities in a way that can be checked at compile time, Slick provides most of its API via a database-specific *driver*. For example, we access most of the Slick API for H2 via the following `import`:

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

Slick makes heavy use of implicit conversions and extension methods, so we generally need to include this import anywhere where we're working with queries or the database. Chapter 5 looks how you can keep a specific database driver out of your code until necessary.

1.4.3 Defining our Schema

Our first job is to tell Slick what tables we have in our database and how to map them onto Scala values and types. The most common representation of data in Scala is a case class, so we start by defining a `Message` class representing a row in our single example table:

```
final case class Message(
  sender:  String,
  content: String,
  id:      Long = 0L)
```

We also define a helper method to create a few test `Messages` for demonstration purposes:

```
def freshTestData = Seq(
  Message("Dave", "Hello, HAL. Do you read me, HAL?"),
  Message("HAL",  "Affirmative, Dave. I read you."),
  Message("Dave", "Open the pod bay doors, HAL."),
  Message("HAL",  "I'm sorry, Dave. I'm afraid I can't do that.")
)
```

Next we define a `Table` object, which corresponds to our database table and tells Slick how to map back and forth between database data and instances of our case class:

```
final class MessageTable(tag: Tag)
      extends Table[Message](tag, "message") {

  def id      = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def sender  = column[String]("sender")
  def content = column[String]("content")

  def * = (sender, content, id) <>
          (Message.tupled, Message.unapply)
}
```

`MessageTable` defines three `columns`: `id`, `sender`, and `content`. It defines the names and types of these columns, and any constraints on them at the database level. For example, `id` is a column of `Long` values, which is also an auto-incrementing primary key.

The `*` method provides a *default projection* that maps between columns in the table and instances of our case class. Slick's `<>` method defines a two-way mapping between three columns and the three fields in `Message`, via the standard `tupled` and `unapply` methods generated as part of the case class. We'll cover projections and default projections in detail in Chapter 5. For now, all we need to know is that this line allows us to query the database and get back `Messages` instead of tuples of `(String, String, Long)`.

The `tag` on the first line is an implementation detail that allows Slick to manage multiple uses of the table in a single query. Think of it like a table alias in SQL. We don't need to provide tags in our user code—Slick takes case of them automatically.

1.4.4 Example Queries

Slick allows us to define and compose queries in advance of running them against the database. We start by defining a `TableQuery` object that represents a simple `SELECT *` style query on our message table:

```
val messages = TableQuery[MessageTable]
```

Note that we're not *running* this query at the moment—we're simply defining it as a means to build other queries. For example, we can create a `SELECT * WHERE` style query using a combinator called `filter`:

```
val halSays = messages.filter(_.sender === "HAL")
```

Again, we haven't run this query yet—we've simply defined it as a useful building block for yet more queries. This demonstrates an important part of Slick's query language—it is made from *composable* building blocks that permit a lot of valuable code re-use.

> **Tip**
>
> **Lifted Embedding**
>
> If you're a fan of terminology, know that what we have discussed so far is called the *lifted embedding* approach in Slick:
>
> - define data types to store row data (case classes, tuples, or other types);
> - define `Table` objects representing mappings between our data types and the database;
> - define `TableQueries` and combinators to build useful queries before we run them against the database.
>
> Lifted embedding is the standard way to work with Slick. We will discuss the other approach, called *Plain SQL querying*, in Chapter 7.

1.4.5 Configuring the Database

We've written all of the code so far without connecting to the database. Now it's time to open a connection and run some SQL. We start by defining a `Database` object, which acts as a factory for managing connections and transactions:

```
val db = Database.forConfig("chapter01")
```

The parameter to `Database.forConfig` determines which configuration to use from the `application.conf` file. This file is found in `src/main/resources`. It looks like this:

```
chapter01 = {
  driver = "org.h2.Driver"
  url    = "jdbc:h2:mem:chapter01"
  keepAliveConnection = true
  connectionPool = disabled
}
```

This syntax comes from the Typesafe Config library, which is also used by Akka and the Play framework.

The parameters we're providing are intended to configure the underlying JDBC layer. The `driver` parameter is the fully qualified class name of the JDBC driver for our chosen DBMS.

The `url` parameter is the standard JDBC connection URL, and in this case we're creating an in-memory database called `"chapter01"`.

By default the H2 in-memory database is deleted when the last connection is closed. As we will be running multiple connections in our examples, we enable `keepAliveConnection` to keep the data around until our program completes.

Slick manages database connections and transactions using auto-commit. We'll look at transactions in Chapter 4. ,

> **Tip**
>
> **JDBC**
>
> If you don't have a background working with Java, you may not have heard of Java Database Connectivity (JDBC). It's a specification for accessing databases in a vendor neutral way. That is, it aims to be independent of the specific database you are connecting to.
>
> The specification is mirrored by a library implemented for each database you want to connect to. This library is called the *JDBC driver*.
>
> JDBC works with *connection strings*, which are URLs like the one above that tell the driver where your database is and how to connect to it (e.g. by providing login credentials).

1.4.6 Creating the Schema

Now that we have a database configured as `db`, we can use it.

Let's start with a `CREATE` statement for `MessageTable`, which we build using methods of our `TableQuery` object, `messages`. The Slick method `schema` gets the schema description. We can see what that would be via the `createStatements` method:

```
messages.schema.createStatements.mkString
// res0: String =
//  create table "message" (
//    "sender" VARCHAR NOT NULL,
//    "content" VARCHAR NOT NULL,
//    "id" BIGINT GENERATED BY DEFAULT
```

```
//     AS IDENTITY(START WITH 1)
//     NOT NULL PRIMARY KEY
// )
```

But we've not sent this to the database yet. We've just printed the statement, to check it is what we think it should be.

In Slick, what we run against the database is an *action*. This is how we create an action for the `messages` schema:

```
val action: DBIO[Unit] = messages.schema.create
```

The result of this `messages.schema.create` expression is a `DBIO[Unit]`. This is an object representing a DB action that, when run, completes with a result of type `Unit`. Anything we run against a database is a `DBIO[T]` (or a `DBIOAction`, more generally). This includes queries, updates, schema alterations, and so on.

> **Tip**
>
> **DBIO and DBIOAction**
>
> In this book we will talk about actions as having the type `DBIO[T]`.
>
> This is a simplification. The more general type is `DBIOAction`, and specifically for this example, it is a `DBIOAction[Unit, NoStream, Effect.Schema]`. The details of all of this we will get to later in the book.
>
> But `DBIO[T]` is a type alias supplied by Slick, and is perfectly fine to use.

Let's run this action:

```
import scala.concurrent.Future

val future: Future[Unit] = db.run(action)
```

The result of `run` is a `Future[T]`, where `T` is the type of result returned by the database. Creating a schema is a side-effecting operation so the result type is `Future[Unit]`. This matches the type `DBIO[Unit]` of the action we started with.

`Future`s are asynchronous. That's to say, they are place holders for values that will eventually appear. We say that a future *completes* at some point. In production code, futures allow us to chain together computations without blocking to wait for a result. However, in simple examples like this we can simply block until our action completes:

```
import scala.concurrent.Await
import scala.concurrent.duration._

val result = Await.result(future, 2 seconds)
```

1.4.7 Inserting Data

Once our table is set up, we need to insert some test data. This is also an action:

```
val insert: DBIO[Option[Int]] = messages ++= freshTestData
```

The ++= method of `message` accepts a sequence of `Message` objects and translates them to a bulk `INSERT` query (recall that `freshTestData` is just a regular Scala `Seq[Message]`). We run the `insert` via `db.run`, and when the future completes our table is populated with data:

```
val result: Future[Option[Int]] = db.run(insert)
```

The result of an insert operation is the number of rows inserted. The `freshTestData` contains four messages, so in this case the result is `Some(4)`. The result is optional because the underlying Java APIs do not guarantee a count of rows for batch inserts—some databases simply return None. We discuss single and batch inserts and updates further in Chapter 3.

1.4.8 Selecting Data

Now our database has a few rows in it, we can start selecting data. We do this by taking a query, such as `messages` or `halSays`, and turning it into an action via the `result` method:

```
val messagesAction: DBIO[Seq[Message]] = messages.result

val messagesFuture: Future[Seq[Message]] = db.run(messagesAction)

val messagesResults = Await.result(messagesFuture, 2 seconds)
// messagesResults: Seq[Example.Message] = Vector(
//   Message(Dave,Hello, HAL. Do you read me, HAL?,1),
//   Message(HAL,Affirmative, Dave. I read you.,2),
//   Message(Dave,Open the pod bay doors, HAL.,3),
//   Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,4))
```

We can see the SQL issued to H2 using the `statements` method on the action:

```
messages.result.statements.mkString
// res2: String = select x2."sender", x2."content", x2."id" from "message" x2
```

> **Tip**
>
> **The exec Helper Method**
>
> In our applications we should avoid blocking on `Futures` whenever possible. However, in the examples in this book we'll be making heavy use of `Await.result`. We will introduce a simple helper method called `exec` to reduce typing and make the examples easier to read:
>
> ```
> def exec[T](action: DBIO[T]): T =
>   Await.result(db.run(action), 2 seconds)
> ```
>
> All `exec` does is run the supplied action and wait for the result. For example, to run a select query we can write:
>
> ```
> exec(messages.result)
> ```
>
> Use of `Await.result` is strongly discouraged in production code. Many web frameworks provide direct

> means of working with `Futures` without blocking. In these cases, the best approach is simply to transform the `Future` query result to a `Future` of an HTTP response and send that to the client.

If we want to retrieve a subset of the messages in our table, we simply run a modified version of our query. For example, calling `filter` on `messages` creates a modified query with a `WHERE` expression that retrieves the expected subset of results:

```
messages.filter(_.sender === "HAL").result.statements.mkString
// res3: String = select x2."sender", x2."content", x2."id"
//                from "message" x2
//                where x2."sender" = 'HAL'
```

To run this query, we convert it to an action using `result`, run it against the database with `db.run`, and await the final result with `exec`:

```
exec(messages.filter(_.sender === "HAL").result)
// res4: Seq[Example.MessageTable#TableElementType] = Vector(
//   Message(HAL,Affirmative, Dave. I read you.,2),
//   Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,4))
```

We actually generated this query earlier and stored it in the variable `halSays`. We can get exactly the same results from the database by running this stored query instead:

```
exec(halSays.result)
// res5: Seq[Example.MessageTable#TableElementType] = Vector(
//   Message(HAL,Affirmative, Dave. I read you.,2),
//   Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,4))
```

Notice that we created our original `halSays` before connecting to the database. This demonstrates perfectly the notion of composing a query from small parts and running it later on. We can even stack modifiers to create queries with multiple additional clauses. For example, we can `map` over the query to retrieve a subset of the data, modifying the `SELECT` clause in the SQL and the return type of the `result`:

```
halSays.map(_.id).result.statements
// res6:List[String] = List(
//        select x2."id" from "message" x2 where x2."sender" = 'HAL'
//      )

exec(halSays.map(_.id).result)
// res7: Seq[Int] = Vector(2, 4)
```

1.4.9 Combining Queries with For Comprehensions

`Query` is a *monad*. It implements the methods `map`, `flatMap`, `filter`, and `withFilter`, making it compatible with Scala for comprehensions. For example, you will often see Slick queries written in this style:

```
val halSays2 = for {
  message <- messages if message.sender === "HAL"
} yield message
```

Remember that for comprehensions are simply aliases for chains of method calls. All we are doing here is building a query with a WHERE clause on it. We don't touch the database until we execute the query:

```
exec(halSays2.result)
// res8: Seq[Message] = ...
```

1.4.10 Actions Combine

Like `Query`, `DBIOAction` is also a monad. It implements the same methods described above, and shares the same compatibility with for comprehensions.

We can combine the actions to create the schema, insert the data, and query results into one action. We can do this before we have a database connection, and we run the action like any other:

```
val actions: DBIO[Seq[Message]] = (
  messages.schema.create       andThen
  (messages ++= freshTestData) andThen
  halSays.result
)
```

And if you want to get funky, >> is another name for andThen:

```
val actions = (
  messages.schema.create       >>
  (messages ++= freshTestData) >>
  halSays.result
)
```

One important reason for composing queries and actions is to wrap them inside a transaction. In Chapter 4 we'll see this, and also that actions can be composed with for comprehensions, just like queries.

Warning

Queries, Actions, Futures... Oh My!

The difference between queries, actions, and futures is the biggest point of confusion for newcomers to Slick 3. The three types share many properties: they all have methods like `map`, `flatMap`, and `filter`, they are all compatible with for comprehensions, and they all flow seamlessly into one another through methods in the Slick API. However, their semantics are quite different:

- `Query` is used to build SQL for a single query. Calls to `map` and `filter` modify clauses to the SQL, but only one query is created.
- `DBIOAction` is used to build sequences of SQL queries. Calls to `map` and `filter` chain queries together and transform their results once they are retrieved in the database. `DBIOAction` is also used to delineate transactions.
- `Future` is used to transform the asynchronous result of running a `DBIOAction`. Transformations on `Futures` happen after we have finished speaking to the database.

In many cases (for example select queries) we create a `Query` first and convert it to a `DBIOAction` using the `result` method. In other cases (for example insert queries), the Slick API gives us a `DBIOAction`

immediately, bypassing `Query`. In all cases, we *run* a `DBIOAction` using `db.run(...)`, turning it into a `Future` of the result.

We recommend taking the time to thoroughly understand `Query`, `DBIOAction`, and `Future`. Learn how they are used, how they are similar, how they differ, what their type parameters represent, and how they flow into one another. This is perhaps the single biggest step you can take towards demystifying Slick 3.

1.5 Take Home Points

In this chapter we've seen a broad overview of the main aspects of Slick, including defining a schema, connecting to the database, and issuing queries to retrieve data.

We typically model data from the database as case classes and tuples that map to rows from a table. We define the mappings between these types and the database using `Table` classes such as `MessageTable`.

We define queries by creating `TableQuery` objects such as `messages` and transforming them with combinators such as `map` and `filter`. These transformations look like transformations on collections, but they are used to build SQL code rather than manipulate the results returned.

We execute a query by creating an action object via its `result` method. Actions are used to build sequences of related queries and wrap them in transactions.

Finally, we run the action against the database by passing it to the `run` method of the database object. We are given back a `Future` of the result. When the future completes, the result is available.

The query language is the one of the richest and most significant parts of Slick. We will spend the entire next chapter discussing the various queries and transformations available.

1.6 Exercise: Bring Your Own Data

Let's get some experience with Slick by running queries against the example database. Start sbt using `sbt.sh` and type `console` to enter the interactive Scala console. We've configured sbt to run the example application before giving you control, so you should start off with the test database set up and ready to go:

```
bash$ ./sbt.sh
# sbt logging...

> console
# More sbt logging...
# Application runs...

scala>
```

Start by inserting an extra line of dialog into the database. This line hit the cutting room floor late in the development of the film 2001, but we're happy to reinstate it here:

```
Message("Dave","What if I say 'Pretty please'?")
```

You'll need to insert the row using the += method on `messages`. Alternatively you could put the message in a `Seq` and use ++=. We've included some common pitfalls in the solution in case you get stuck.

See the solution

Now retrieve the new dialog by selecting all messages sent by Dave. You'll need to build the appropriate query using `messages.filter`, and create the action to be run by using its `result` method. Don't forget to run the query by using the `exec` helper method we provided.

Again, we've included some common pitfalls in the solution.

See the solution

Chapter 2

Selecting Data

The last chapter provided a shallow end-to-end overview of Slick. We saw how to model data, create queries, convert them to actions, and run those actions against a database. In the next two chapters we will look in more detail at the various types of query we can perform in Slick.

This chapter covers *selecting* data using Slick's rich type-safe Scala reflection of SQL. Chapter 3 covers *modifying* data by inserting, updating, and deleting records.

Select queries are our main means of retrieving data. In this chapter we'll limit ourselves to simple select queries that operate on a single table. In Chapter 6 we'll look at more complex queries involving joins, aggregates, and grouping clauses.

2.1 Select All The Rows!

The simplest select query is the `TableQuery` generated from a `Table`. In the following example, `messages` is a `TableQuery` for `MessageTable`:

```
final class MessageTable(tag: Tag)
    extends Table[Message](tag, "message") {

  def id      = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def sender  = column[String]("sender")
  def content = column[String]("content")

  def * = (sender, content, id) <>
    (Message.tupled, Message.unapply)
}
// defined class MessageTable

lazy val messages = TableQuery[MessageTable]
// messages: slick.lifted.TableQuery[Example.MessageTable] = <lazy>
```

The type of `messages` is `TableQuery[MessageTable]`, which is a subtype of a more general `Query` type that Slick uses to represent select, update, and delete queries. We'll discuss these types in the next section.

We can see the SQL of the select query by calling `result.statements`:

```
messages.result.statements
// res12: Iterable[String] =
//  List(select x2."sender", x2."content", x2."id" from "message" x2)
```

Our `TableQuery` is the equivalent of the SQL `select * from message`.

> **Advanced**
>
> **Query Extension Methods**
>
> Like many of the methods discussed below, the `result` method is actually an extension method applied to `Query` via an implicit conversion. You'll need to have everything from `H2Driver.api` in scope for this to work:
>
> ```
> import slick.driver.H2Driver.api._
> ```

2.2 Filtering Results: The *filter* Method

We can create a query for a subset of rows using the `filter` method:

```
messages.filter(_.sender === "HAL")
// res13: slick.lifted.Query[
//   MessageTable,
//   MessageTable#TableElementType,
//   Seq
// ] = Rep(Filter)
```

The parameter to `filter` is a function from an instance of `MessageTable` to a value of type `Rep[Boolean]` representing a `WHERE` clause for our query:

```
messages.filter(_.sender === "HAL").result.statements
// res14: Iterable[String] =
// List(select x2."sender", x2."content", x2."id"
//        from "message" x2 where x2."sender" = 'HAL')
```

Slick uses the `Rep` type to represent expressions over columns as well as individual columns. A `Rep[Boolean]` can either be a `Boolean`-valued column in a table, or a `Boolean` expression involving multiple columns. Slick can automatically promote a value of type `A` to a constant `Rep[A]`, and provides a suite of methods for building expressions as we shall see below.

2.3 The Query and TableQuery Types

The types in our `filter` expression deserve some deeper explanation. Slick represents all queries using a trait `Query[M, U, C]` that has three type parameters:

- `M` is called the *mixed* type. This is the function parameter type we see when calling methods like `map` and `filter`.
- `U` is called the *unpacked* type. This is the type we collect in our results.

- C is called the *collection* type. This is the type of collection we accumulate results into.

In the examples above, `messages` is of a subtype of `Query` called `TableQuery`. Here's a simplified version of the definition in the Slick codebase:

```
trait TableQuery[T <: Table[_]] extends Query[T, T#TableElementType, Seq] {
  // ...
}
```

A `TableQuery` is actually a `Query` that uses a `Table` (e.g. `MessageTable`) as its mixed type and the table's element type (the type parameter in the constructor, e.g. `Message`) as its unpacked type. In other words, the function we provide to `messages.filter` is actually passed a parameter of type `MessageTable`:

```
messages.filter { messageTable: MessageTable =>
  messageTable.sender === "HAL"
}
```

This makes sense: `messageTable.sender` is one of the columns we defined in `MessageTable` above, and `messageTable.sender === "HAL"` creates a Scala value representing the SQL expression `message.sender = 'HAL'`.

This is the process that allows Slick to type-check our queries. `Query`s have access to the type of the `Table` used to create them, allowing us to directly reference the columns on the `Table` when we're using combinators like `map` and `filter`. Every column knows its own data type, so Slick can ensure we only compare columns of compatible types. If we try to compare `sender` to an `Int`, for example, we get a type error:

```
messages.filter(_.sender === 123)
// <console>:16: error: Cannot perform option-mapped operation
//       with type: (String, Int) => R
//   for base type: (String, String) => Boolean
//               messages.filter(_.sender === 123)
//                                        ^
```

> **Tip**
>
> **Constant Queries**
>
> How can we perform useful queries such as `select 1` in Slick, and why would we want to?
>
> We can use the `Query` companion object!
>
> ```
> Query(1)
> ```
>
> will produce
>
> ```
> select 1
> ```
>
> The `apply` method of the `Query` object allows us to lift a scalar value to a `Query`.
>
> A simple query such as `select 1` can be used to confirm we have database connectivity. A useful thing to do as an application is starting up and as a heartbeat system check that will consume minimal resources.
>
> We'll see another example of using a `from`-less query in Chapter 3.

2.4 Transforming Results

2.4.1 The *map* Method

Sometimes we don't want to select all of the columns in a `Table`. We can use the `map` method on a `Query` to select specific columns for inclusion in the results. This changes both the mixed type and the unpacked type of the query:

```
messages.map(_.content)
// res1: slick.lifted.Query[
//   slick.lifted.Rep[String],
//   String,
//   Seq
// ] = slick.lifted.Query
```

Because the unpacked type (second type parameter) has changed to `String`, we now have a query that selects `Strings` when run. If we run the query we see that only the `content` of each message is retrieved:

```
val query = messages.map(_.content)
// query: slick.lifted.Query[slick.lifted.Rep[String],String,Seq] =
//   Rep(Bind)exec(messages.map(_.content).result)

exec(query.result)
// res15: Seq[String] = Vector(
//   Hello, HAL. Do you read me, HAL?,
//   Affirmative, Dave. I read you.,
//   Open the pod bay doors, HAL.,
//   I'm sorry, Dave. I'm afraid I can't do that.,
//   What if I say 'Pretty please'?)
```

Tip

exec

Just as we did in Chapter 1, we're using a simple helper method to run queries in the REPL:

```
def exec[T](action: DBIO[T]): T =
  Await.result(db.run(action), 2 seconds)
```

This is included in the example source code for this chapter, in the `main.scala` file. You can run these examples in the REPL to follow along with the text:

```
val query = messages.map(_.content)
// query: slick.lifted.Query[
//  slick.lifted.Rep[String],
//  String,
//  Seq] = Rep(Bind)

exec(query.result)
// res1: Seq[String] =
//  Vector(Hello, HAL. Do you read me, HAL?, ...etc
```

Also notice that the generated SQL has changed. Slick isn't cheating: it is actually telling the database to restrict the results to that column in the SQL:

```
messages.map(_.content).result.statements
// res16: Iterable[String] = List(select x2."content" from "message" x2)
```

Finally, notice that the mixed type (first type parameter) of our new query has changed to `Rep[String]`. This means we are only passed the `content` column when we `filter` or `map` over this query:

```
val seekBeauty = messages.
  map(_.content).
  filter{content:Rep[String] => content like "%Pretty%" }

// seekBeauty: slick.lifted.Query[
//   slick.lifted.Rep[String],
//   String,
//   Seq
// ] = Rep(Filter)

exec(seekBeauty.result)
// res17: Seq[String] = Vector(What if I say 'Pretty please'?)
```

This change of mixed type can complicate query composition with `map`. We recommend calling `map` only as the final step in a sequence of transformations on a query, after all other operations have been applied.

It is worth noting that we can `map` to anything that Slick can pass to the database as part of a `select` clause. This includes individual Reps and `Tuples` of the above. For example, we can use `map` to select the `id` and `content` columns of messages:

```
messages.map(t => (t.id, t.content))
// res18: slick.lifted.Query[
//    (slick.lifted.Rep[Long], slick.lifted.Rep[String]),
//    (Long, String),
//     Seq
// ] = Rep(Bind)
```

The mixed and unpacked types change accordingly, and the SQL is modified as we might expect:

```
messages.map(t => (t.id, t.content)).result.statements
// res19: Iterable[String] = List(
//    select x2."id", x2."content" from "message" x2
// )
```

We can even map sets of columns to Scala data structures using the projection operator, <>. Don't worry about this too much now—we'll cover <> in detail in Chapter 5:

```
case class TextOnly(id: Long, content: String)

val contentQuery = messages.
  map(t => (t.id, t.content) <> (TextOnly.tupled, TextOnly.unapply))
// contentQuery: Query[
//   MappedProjection[Test,(Long, String)],
//   Test,
//   Seq] = ...

exec(contentQuery.result)
// res5: Seq[TextOnly] = Vector(
//   TextOnly(1,Hello, HAL. Do you read me, HAL?),
//   TextOnly(2,Affirmative, Dave. I read you.),
//   TextOnly(3,Open the pod bay doors, HAL.),
//   ...)
```

We can also select column expressions as well as single columns:

```
messages.map(t => t.id * 1000L).result.statements
// res20: Iterable[String] = List(select x2."id" * 1000 from "message" x2)
```

This all means that `map` is a powerful combinator for controling the `SELECT` part of your query.

> **Tip**
>
> **Query's *flatMap* Method**
>
> `Query` also has a `flatMap` method with similar monadic semantics to that of `Option` or `Future`. `flatMap` is mostly used for joins, so we'll cover it in Chapter 6.

2.4.2 *exists*

Sometimes we are less interested in the contents of a queries result than if results exist at all. For this we have `exists`, which will return `true` if the result set is not empty and false otherwise.

Let's look at quick example to show how we can use an existing query with the `exists` keyword:

```
val containsBay = for {
  m <- messages
  if m.content like "%bay%"
} yield m

val bayMentioned: DBIO[Boolean] =
  containsBay.exists.result
```

The `containsBay` query returns all messages that mention "bay". We can then use this query in the `bayMentioned` expression to determine what to execute.

The above will generate SQL which looks similar to this:

```
select exists(
  select x2."sender", x2."content", x2."id"
  from "message" x2
  where x2."content" like '%bay%'
)
```

We will see a more useful example in Chapter 3.

2.5 Converting Queries to Actions

Before running a query, we need to convert it to an *action*. We typically do this by calling the `result` method on the query. Actions represent sequences of queries. We start with actions representing single queries and compose them to form multi-action sequences.

Actions have the type signature `DBIOAction[R, S, E]`. The three type parameters are:

- `R` is the type of data we expect to get back from the database (`Message`, `Person`, etc);
- `S` indicates whether the results are streamed (`Streaming[T]`) or not (`NoStream`); and
- `E` is the effect type and will be inferred.

In many cases we can simplify the representation of an action to just `DBIO[T]`, which is an alias for `DBIOAction[T, NoStream, Effect.All]`.

> **Tip**
>
> **Effects**
>
> Effects are not part of Essential Slick, and we'll be working in terms of `DBIO[T]` for most of this text.
>
> However, broadly speaking, an `Effect` is a way to annotate an action. For example, you can write a method that will only accept queries marked as `Read` or `Write`, or a combination such as `Read with Transactional`.
>
> The effects defined in Slick under the `Effect` object are:
>
> - `Read` for queries that read from the database.
> - `Write` for queries that have a write effect on the database.
> - `Schema` for schema effects.
> - `Transactional` for transaction effects.
> - `All` for all of the above.
>
> Slick will infer the effect for your queries. For example, `messages.result` will be:
>
> ```
> DBIOAction[Seq[String], NoStream, Effect.Read]
> ```
>
> In the next chapter we will look at inserts and updates. The inferred effect for an update in this case is: `DBIOAction[Int, NoStream, Effect.Write]`.
>
> You can also add your own `Effect` types by extending the existing types.

2.6 Executing Actions

To execute an action, we pass it to one of two methods on our `db` object:

- `db.run(...)` runs the action and returns all the results in a single collection. These are known as a *materalized* result.
- `db.stream(...)` runs the action and returns its results in a `Stream`, allowing us to process large datasets incrementally without consuming large amounts of memory.

In this book we will deal exclusively with materialized queries. `db.run(...)` returns a `Future` of the final result of our action. We need to have an `ExecutionContext` in scope when we make the call:

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

val futureMessages = db.run(halSays.result)
// futureMessages: Future[Seq[Message]] = ...
```

> **Tip**
>
> **Streaming**
>
> In this book we will deal exclusively with materialized queries. Let's take a quick look at streams now, so we are aware of the alternative.
>
> Calling `db.stream` returns a `DatabasePublisher` object instead of a `Future`. This exposes three methods to interact with the stream:
>
> - `subscribe` which allows integration with Akka;
> - `mapResult` which creates a new `Publisher` that maps the supplied function on the result set from the original publisher; and
> - `foreach`, to perform a side-effect with the results.
>
> Streaming results can be used to feed reactive streams, or Akka streams or actors. Alternatively, we can do something simple like use `foreach` to `println` our results:
>
> ```
> db.stream(messages.result).foreach(println)
> // res1: scala.concurrent.Future[Unit] =
> //   scala.concurrent.impl.Promise$DefaultPromise@52a01c1e
>
> // Output:
> // Message(Dave,Hello, HAL. Do you read me, HAL?,1)
> // Message(HAL,Affirmative, Dave. I read you.,2)
> // Message(Dave,Open the pod bay doors, HAL.,3)
> // Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,4)
> // Message(Dave,What if I say 'Pretty please'?,5)
> ```
>
> If you want to explore this area, start with the Slick documentation on streaming.

2.7 Column Expressions

Methods like `filter` and `map` require us to build expressions based on columns in our tables. The `Rep` type is used to represent expressions as well as individual columns. Slick provides a variety of extension methods on `Rep` for building expressions.

We will cover the most common methods below. You can find a complete list in ExtensionMethods.scala in the Slick codebase.

2.7.1 Equality and Inequality Methods

The `===` and `=!=` methods operate on any type of `Rep` and produce a `Rep[Boolean]`. Here are some examples:

```
messages.filter(_.sender === "Dave").result.statements
//res21: Iterable[String] =
// List(select x2."sender", x2."content", x2."id"
//      from "message" x2 where x2."sender" = 'Dave')

messages.filter(_.sender =!= "Dave").result.statements
// res22: Iterable[String] =
//  List(select x2."sender", x2."content", x2."id"
//      from "message" x2
//      where not (x2."sender" = 'Dave'))
```

The `<`, `>`, `<=`, and `>=` methods can operate on any type of `Rep` (not just numeric columns):

```
messages.filter(_.sender < "HAL").result.statements
// res23: Iterable[String] =
//  List(select x2."sender", x2."content", x2."id"
//       from "message" x2 where x2."sender" < 'HAL')


messages.filter(m => m.sender >= m.content).result.statements
// res24: Iterable[String] =
//  List(select x2."sender", x2."content", x2."id"
//       from "message" x2
//       where x2."sender" >= x2."content"
```

Table 2.1: `Rep` comparison methods. Operand and result types should be interpreted as parameters to `Rep[_]`.

Scala Code	Operand Types	Result Type	SQL Equivalent
`col1 === col2`	`A` or `Option[A]`	`Boolean`	`col1 = col2`
`col1 =!= col2`	`A` or `Option[A]`	`Boolean`	`col1 <> col2`
`col1 < col2`	`A` or `Option[A]`	`Boolean`	`col1 < col2`
`col1 > col2`	`A` or `Option[A]`	`Boolean`	`col1 > col2`
`col1 <= col2`	`A` or `Option[A]`	`Boolean`	`col1 <= col2`
`col1 >= col2`	`A` or `Option[A]`	`Boolean`	`col1 >= col2`

2.7.2 String Methods

Slick provides the ++ method for string concatenation (SQL's || operator):

```
messages.map(m => m.sender ++ "> " ++ m.content).result.statements
// res25: Iterable[String] =
//  List(select (x2."sender"||'> ')||x2."content"
//       from "message" x2)
```

and the `like` method for SQL's classic string pattern matching:

```
messages.filter(_.content like "%Pretty%").result.statements
// res26:  Iterable[String] = List(... where x2."content" like '%Pretty%')
```

Slick also provides methods such as `startsWith`, `length`, `toUpperCase`, `trim`, and so on. These are implemented differently in different DBMSs—the examples below are purely for illustration:

Table 2.2: String column methods. Operand and result types should be interpreted as parameters to `Rep[_]`.

Scala Code	Operand Column Types	Result Type	SQL Equivalent
`col1.length`	`String` or `Option[String]`	`Int`	`char_length(col1)`
`col1 ++ col2`	`String` or `Option[String]`	`String`	`col1 || col2`
`col1 like col2`	`String` or `Option[String]`	`Boolean`	`col1 like col2`
`col1 startsWith col2`	`String` or `Option[String]`	`Boolean`	`col1 like (col2 || '%')`
`col1 endsWith col2`	`String` or `Option[String]`	`Boolean`	`col1 like ('%' || col2)`
`col1.toUpperCase`	`String` or `Option[String]`	`String`	`upper(col1)`
`col1.toLowerCase`	`String` or `Option[String]`	`String`	`lower(col1)`
`col1.trim`	`String` or `Option[String]`	`String`	`trim(col1)`
`col1.ltrim`	`String` or `Option[String]`	`String`	`ltrim(col1)`
`col1.rtrim`	`String` or `Option[String]`	`String`	`rtrim(col1)`

2.7.3 Numeric Methods

Slick provides a comprehensive set of methods that operate on Reps with numeric values: `Int`s, `Long`s, `Double`s, `Float`s, `Short`s, `Byte`s, and `BigDecimal`s.

Table 2.3: Numeric column methods. Operand and result types should be interpreted as parameters to `Rep[_]`.

Scala Code	Operand Column Types	Result Type	SQL Equivalent
`col1 + col2`	`A` or `Option[A]`	`A`	`col1 + col2`
`col1 - col2`	`A` or `Option[A]`	`A`	`col1 - col2`
`col1 * col2`	`A` or `Option[A]`	`A`	`col1 * col2`
`col1 / col2`	`A` or `Option[A]`	`A`	`col1 / col2`
`col1 % col2`	`A` or `Option[A]`	`A`	`mod(col1, col2)`
`col1.abs`	`A` or `Option[A]`	`A`	`abs(col1)`

Scala Code	Operand Column Types	Result Type	SQL Equivalent
`col1.ceil`	`A` or `Option[A]`	`A`	`ceil(col1)`
`col1.floor`	`A` or `Option[A]`	`A`	`floor(col1)`
`col1.round`	`A` or `Option[A]`	`A`	`round(col1, 0)`

2.7.4 Boolean Methods

Slick also provides a set of methods that operate on boolean Reps:

Table 2.4: Boolean column methods. Operand and result types should be interpreted as parameters to `Rep[_]`.

Scala Code	Operand Column Types	Result Type	SQL Equivalent
`col1 && col2`	`Boolean` or `Option[Boolean]`	`Boolean`	`col1 and col2`
`col1 || col2`	`Boolean` or `Option[Boolean]`	`Boolean`	`col1 or col2`
`!col1`	`Boolean` or `Option[Boolean]`	`Boolean`	`not col1`

2.7.5 Option Methods and Type Equivalence

Slick models nullable columns in SQL as Reps with `Option` types. We'll discuss this in some depth in Chapter 5. However, as a preview, know that if we have a nullable column in our database, we declare it as optional in our `Table`:

```
final class PersonTable(tag: Tag) /* ... */ {
  // ...
  def nickname = column[Option[String]]("nickname")
  // ...
}
```

When it comes to querying on optional values, Slick is pretty smart about type equivalence.

What do we mean by type equivalence? Slick type-checks our column expressions to make sure the operands are of compatible types. For example, we can compare `String`s for equality but we can't compare a `String` and an `Int`:

```
messages.filter(_.id === "foo")
//<console>:14: error: Cannot perform option-mapped operation
//      with type: (Long, String) => R
//  for base type: (Long, Long) => Boolean
//              messages.filter(_.id === "foo")
//                                   ^
```

Interestingly, Slick is very finickity about numeric types. For example, comparing an `Int` to a `Long` is considered a type error:

```
messages.filter(_.id === 123)
// <console>:14: error: Cannot perform option-mapped operation
//       with type: (Long, Int) => R
```

```
//   for base type: (Long, Long) => Boolean
//               messages.filter(_.id === 123)
//                                  ^
```

On the flip side of the coin, Slick is clever about the equivalence of optional and non-optional columns. As long as the operands are some combination of the types `A` and `Option[A]` (for the same value of `A`), the query will normally compile:

```
messages.filter(_.id === Option(123L)).result.statements
// res27: Iterable[String] =
//  List(select x2."sender", x2."content", x2."id"
//       from "message" x2 where x2."id" = 123)
```

However, any optional arguments must be strictly of type `Option`, not `Some` or `None`:

```
messages.filter(_.id === Some(123L)).result.statements
// <console>:14: error: type mismatch;
//  found   : Some[Long]
//  required: slick.lifted.Rep[?]
//               messages.filter(_.id === Some(123L)).result.statements
//                                           ^
```

If you find yourself in this situation, remember you can always provide a type ascription to the value:

```
messages.filter(_.id === (Some(123L): Option[Long]) )
```

2.8 Controlling Queries: Sort, Take, and Drop

There are a trio of functions used to control the order and number of results returned from a query. This is great for pagination of a result set, but the methods listed in the table below can be used independently.

Table 2.5: Methods for ordering, skipping, and limiting the results of a query.

Scala Code	SQL Equivalent
`sortBy`	`ORDER BY`
`take`	`LIMIT`
`drop`	`OFFSET`

We'll look at each in turn, starting with an example of `sortBy`. Say we want want messages in order of the sender's name:

```
exec(messages.sortBy(_.sender).result)
// res28: Seq[Example.MessageTable#TableElementType] =
//  Vector(Message(Dave,Hello, HAL. Do you read me, HAL?,1),
//  Message(Dave,Open the pod bay doors, HAL.,3),
//  Message(HAL,Affirmative, Dave. I read you.,2),
//  Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,4))
```

To sort by multiple columns, return a tuple of columns:

```
messages.sortBy(m => (m.sender, m.content)).result.statements
// res29: Iterable[String] =
//  List(select x2."sender", x2."content", x2."id"
//       from "message" x2
//       order by x2."sender", x2."content")
```

Now we know how to sort results, perhaps we want to show only the first five rows:

```
messages.sortBy(_.sender).take(5)
```

If we are presenting information in pages, we'd need a way to show the next page (rows 6 to 10):

```
messages.sortBy(_.sender).drop(5).take(5)
```

This is equivalent to:

```
select "sender", "content", "id"
from "message"
order by "sender"
limit 5 offset 5
```

> **Tip**
>
> **Sorting on Null columns**
>
> We had a brief introduction to nullable columns earlier in the chapter when we looked at Option Methods and Type Equivalence. Slick offers three modifiers which can be used in conjunction with `desc` and `asc` when sorting on nullable columns: `nullFirst`, `nullsDefault` and `nullsLast`. These do what you expect, by including nulls at the beginning or end of the result set. The `nullsDefault` behaviour will use the SQL engines preference.
>
> An example of sorting a nullable column:
>
> ```
> users.sortBy { _.name.nullsFirst }.result.statements.foreach { println }
> ```
>
> The generated SQL would be:
>
> ```
> select x2."name", x2."email", x2."id"
> from "user" x2
> order by x2."name" nulls first
> ```
>
> We cover nullable columns in Chapter 5 and include an example of sorting on nullable columns in example project the code is in *nulls.scala* in the folder *chapter-05*.

2.9 Take Home Points

Starting with a `TableQuery` we can construct a wide range of queries with `filter` and `map`. As we compose these queries, the types of the `Query` follow along to give type-safety throughout our application.

The expressions we use in queries are defined in extension methods, and include `===`, `=!=`, `like`, `&&` and so on, depending on the type of the `Rep`. Comparisons to `Option` types are made easy for us as Slick will compare `Rep[T]` and `Rep[Option[T]]` automatically.

We've seen that `map` acts like a SQL `select`, and `filter` is like a `WHERE`. We'll see the Slick representation of `GROUP` and `JOIN` in Chapter 6.

We introduced some new terminology:

- *unpacked* type, which is the regular Scala types we work with, such as `String`; and
- *mixed* type, which is Slick's column representation, such as `Rep[String]`.

We run queries by converting them to actions using the `result` method. We run the actions against a database using `db.run`.

The database action type constructor `DBIOAction` takes three arguments that represent the result, streaming mode, and effect. `DBIO[R]` simplifies this to just the result type.

2.10 Exercises

If you've not already done so, try out the above code. In the example project the code is in *main.scala* in the folder *chapter-02*.

Once you've done that, work through the exercises below. An easy way to try things out is to use *triggered execution* with SBT:

```
$ cd example-02
$ ./sbt.sh
> ~run
```

That `~run` will monitor the project for changes, and when a change is seen, the *main.scala* program will be compiled and run. This means you can edit *main.scala* and then look in your terminal window to see the output.

2.10.1 Count the Messages

How would you count the number of messages? Hint: in the Scala collections the method `length` gives you the size of the collection.

See the solution

2.10.2 Selecting a Message

Using a for comprehension, select the message with the id of 1. What happens if you try to find a message with an id of 999?

Hint: our IDs are `Long`s. Adding `L` after a number in Scala, such as `99L`, makes it a long.

See the solution

2.10.3 One Liners

Re-write the query from the last exercise to not use a for comprehension. Which style do you prefer? Why?

See the solution

2.10.4 Checking the SQL

Calling the `result.statements` methods on a query will give you the SQL to be executed. Apply that to the last exercise. What query is reported? What does this tell you about the way `filter` has been mapped to SQL?

See the solution

2.10.5 Is HAL Real?

Find if there are any messages by HAL in the database, but only return a boolean value from the database.

See the solution

2.10.6 Selecting Columns

So far we have been returning `Message` classes or counts. Select all the messages in the database, but return just their contents. Hint: think of messages as a collection and what you would do to a collection to just get back a single field of a case class.

Check what SQL would be executed for this query.

See the solution

2.10.7 First Result

The methods `head` and `headOption` are useful methods on a `result`. Find the first message that HAL sent. What happens if you use `head` to find a message from "Alice" (note that Alice has sent no messages).

See the solution

2.10.8 Then the Rest

In the previous exercise you returned the first message HAL sent. This time find the next five messages HAL sent. What messages are returned?

What if we'd asked for HAL's tenth through to twentieth message?

See the solution

2.10.9 The Start of Something

The method `startsWith` on a `String` tests to see if the string starts with a particular sequence of characters. Slick also implements this for string columns. Find the message that starts with "Open". How is that query implemented in SQL?

See the solution

2.10.10 Liking

Slick implements the method `like`. Find all the messages with "do" in their content. Can you make this case insensitive?

See the solution

2.10.11 Client-Side or Server-Side?

What does this do and why?

```
exec(messages.map(_.content + "!").result)
```

See the solution

Chapter 3

Creating and Modifying Data

In the last chapter we saw how to retrieve data from the database using select queries. In this chapter we will look at modifying stored data using insert, update, and delete queries.

SQL veterans will know that update and delete queries share many similarities with select queries. The same is true in Slick, where we use the `Query` monad and combinators to build the different kinds of query. Ensure you are familiar with the content of Chapter 2 before proceeding.

This chapter also introduces the important concept of *action combinators*. These combinators enable us to combine actions into a single action. The result is an action that can be made up of multiple updates, selects, deletes, or other actions.

3.1 Inserting Rows

As we saw in Chapter 1, adding new data looks like an append operation on a mutable collection. We can use the += method to insert a single row into a table, and ++= to insert multiple rows. We'll discuss both of these operations below.

3.1.1 Inserting Single Rows

To insert a single row into a table we use the += method. Note that, unlike the select queries we've seen, this creates a `DBIOAction` immediately without an intermediate `Query`:

```
val action =
  messages += Message("HAL", "No. Seriously, Dave, I can't let you in.")

exec(action)
// res1: Int = 1
```

The result of the action is the number of rows inserted. However, it is often useful to return something else, such as the primary key generated for the new row. We can get this information using a method called `returning`. Before we get to that, we first need to understand where the primary key comes from.

3.1.2 Primary Key Allocation

When inserting data, we need to tell the database whether or not to allocate primary keys for the new rows. It is common practice to declare auto-incrementing primary keys, allowing the database to allocate values automatically if we don't manually specify them in the SQL.

Slick allows us to allocate auto-incrementing primary keys via an option on the column definition. Recall the definition of `MessageTable` from Chapter 1, which looked like this:

```
final class MessageTable(tag: Tag)
      extends Table[Message](tag, "message") {

  def id      = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def sender  = column[String]("sender")
  def content = column[String]("content")

  def * = (sender, content, id) <>
          (Message.tupled, Message.unapply)
}
```

The `O.AutoInc` option specifies that the `id` column is auto-incrementing, and tells Slick to ignore the column in the SQL generated by +=:

```
action.statements.head
// res2: String =
//   insert into "message" ("sender","content")
//   values (?,?)
```

As a convenience, in our example code we put the `id` field at the end of the case class and gave it a default value of `0L`, allowing us to skip the field when creating new objects of type `Message`:

```
final case class Message(
  sender:  String,
  content: String,
  id:      Long = 0L
)

Message("Dave", "You're off my Christmas card list.")
```

There is nothing special about our default value of `0L`—it's simply a default value for the field. It is the `O.AutoInc` option that determines the behaviour of +=.

Sometimes we want to override the database's default auto-incrementing behaviour and specify our own primary key. Slick provides a `forceInsert` method that does just this:

```
val forceInsertAction = messages forceInsert Message(
   "HAL",
   "I'm a computer, what would I do with a Christmas card anyway?",
   1000L)
```

Notice that the SQL generated for this action includes a manually specified ID, and that running the action results in a record with the ID being inserted:

```
forceInsertAction.statements.head
// res4: String =
//       insert into "message" ("sender","content","id")  values (?,?,?)

exec(forceInsertAction)
```

```
exec(messages.filter(_.id === 1000L).result)
// res4: Seq[Example.MessageTable#TableElementType] =
//   Vector(Message(
//     HAL,
//     I'm a computer, what would I do with a Christmas card anyway?,
//     1000))
```

3.1.3 Retrieving Primary Keys on Insert

When the database allocates primary keys for us sit's often the case that we want get the key back as after an insert. Slick supports this via the `returning` method:

```
val insert: DBIO[Long] =
  messages returning messages.map(_.id) += Message("Dave", "Point taken.")

exec(insert)
// res5: Long = 1001
```

The argument to `messages returning` is a `Query` over the same table, which is why `messages.map(_.id)` makes sense here. The query specifies what data we'd like the database to return once the insert has finished.

We can demonstrate that the return value is a primary key by looking up the record we just inserted:

```
exec(messages.filter(_.id === 1001L).result.headOption)
// res6: Option[Example.Message] =
//   Some(Message(Dave,Point taken.,1001))
```

For convenience, we can save a few keystrokes and define an insert query that always returns the primary key:

```
lazy val messagesReturningId = messages returning messages.map(_.id)
// messagesReturningId: slick.driver.H2Driver.ReturningInsertActionComposer[
//    Example.MessageTable#TableElementType,
//    Long
//  ] = <lazy>

exec(messagesReturningId += Message("HAL", "I don't know. I guess we wait."))
// res8: Long = 1002
```

Using `messagesReturningId` will return us the `id`, rather than the count of the number of rows inserted.

3.1.4 Retrieving Rows on Insert

Some databases allow us to retrieve the complete inserted record, not just the primary key. For example, we could ask for the whole `Message` back:

```
exec(messages returning messages +=
 Message("Dave", "So... what do we do now?" ))

// res7: Message = ...
```

Not all databases provide complete support for the `returning` method. H2 only allows us to retrieve the primary key from an insert.

If we tried this with H2, we get a runtime error:

```
exec(messages returning messages +=
      Message("Dave", "So... what do we do now?" ))
// slick.SlickException:
//   This DBMS allows only a single AutoInc column ↩
//     to be returned from an INSERT
//   at ...
```

This is a shame, but getting the primary key is often all we need.

> **Tip**
>
> **Driver Capabilities**
>
> The Slick manual contains a comprehensive table of the capabilities for each database driver. The ability to return complete records from an insert query is referenced as the `jdbc.returnInsertOther` capability.
>
> The API documentation for each driver also lists the capabilities that the driver *doesn't* have. For an example, the top of the H2 Driver Scaladoc page points out several of its shortcomings.

If want to get a complete populated `Message` back from a database without `jdbc.returnInsertOther` support, we retrieve the primary key and manually add it to the inserted record. Slick simplifies this with another method, `into`:

```
val messagesReturningRow =
  messages returning messages.map(_.id) into { (message, id) =>
    message.copy(id = id)
  }
// messagesReturningRow: slick.driver.H2Driver.IntoInsertActionComposer[
//   Example.MessageTable#TableElementType,
//   Example.Message
// ] = ...

val insert: DBIO[Message] =
  messagesReturningRow += Message("Dave", "You're such a jerk.")

exec(insert)
// res9: messagesReturningRow.SingleInsertResult =
//   Message(Dave,You're such a jerk.,1004)
```

The `into` method allows us to specify a function to combine the record and the new primary key. It's perfect for emulating the `jdbc.returnInsertOther` capability, although we can use it for any post-processing we care to imagine on the inserted data.

3.1.5 Inserting Specific Columns

If our database table contains a lot of columns with default values, it is sometimes useful to specify a subset of columns in our insert queries. We can do this by `mapping` over a query before calling `insert`:

```
messages.map(_.sender).insertStatement
// res10: String =
//   insert into "message" ("sender")
//   values (?)
```

The parameter type of the += method is matched to the *unpacked* type of the query:

```
messages.map(_.sender)
// res11: slick.lifted.Query[
//  slick.lifted.Rep[String],
//  String,
//  Seq] = Rep(Bind)
```

... so we execute this query by passing it a `String` for the `sender`:

```
exec(messages.map(_.sender) += "HAL")
// org.h2.jdbc.JdbcSQLException:
//   NULL not allowed for column "content"; SQL statement:
// insert into "message" ("sender")  values (?) [23502-185]
//   at ...
```

The query fails at runtime because the `content` column is non-nullable in our schema. No matter. We'll cover nullable columns when discussing schemas in Chapter 5.

3.1.6 Inserting Multiple Rows

Suppose we want to insert several `Messages` at the same time. We could just use += to insert each one in turn. However, this would result in a separate query being issued to the database for each record, which could be slow for large numbers of inserts.

As an alternative, Slick supports *batch inserts*, where all the inserts are sent to the database in one go. We've seen this already in the first chapter:

```
val testMessages = Seq(
  Message("Dave", "Hello, HAL. Do you read me, HAL?"),
  Message("HAL",  "Affirmative, Dave. I read you."),
  Message("Dave", "Open the pod bay doors, HAL."),
  Message("HAL",  "I'm sorry, Dave. I'm afraid I can't do that.")
)
// testMessages: Seq[Message] = ...

exec(messages ++= testMessages)
// res11: Option[Int] = Some(4)
```

This code prepares one SQL statement and uses it for each row in the `Seq`. In principle Slick could optimize this insert further using database-specific featues. This can result in a significant boost in performance when inserting many records.

As we saw earlier this chapter, the default return value of a single insert is the number of rows inserted. The multi-row insert above is also returning the number of rows, except this time the type is `Option[Int]`. The

reason for this is that the JDBC specification permits the underlying database driver to indicate that the number of rows inserted is unknown.

Slick also provides a batch version of `messages returning...`, including the `into` method. We can use the `messagesReturningRow` query we defined last section and write:

```
exec(messagesReturningRow ++= testMessages)
// res12: messagesReturningRow.MultiInsertResult = List(
//   Message(Dave,Hello, HAL. Do you read me, HAL?,13),
//   ...)
```

3.1.7 More Control over Inserts

At this point we've inserted fixed data into the database. Sometimes you need more flexibility, including inserting data based on another query. Slick supports this via `forceInsertQuery`.

The argument to `forceInsertQuery` is a query. So the form is:

```
insertExpression.forceInsertQuery(selectExpression)
```

Our `selectExpression` can be pretty much anything, but it needs to match the columns required by our `insertExpression`.

As an example, our query could check to see if a particular row of data already exists, and insert it if it doesn't. That is, an "insert if doesn't exist" function.

Let's say we only want the director to be able to say "Cut!" once. The SQL would end up like this:

```
insert into "messages" ("sender", "content")
  select 'Stanley', 'Cut!'
where
  not exists(
    select
      "id", "sender", "content"
    from
      "messages" where "name" = 'Stanley'
                 and   "content" = 'Cut!')
```

That looks quite involved, but we can build it up gradually.

The tricky part of this is the `select 'Stanley', 'Cut!'` part, as there is no `FROM` clause there. We saw an example of how to create that in Chapter 2, with `Query.apply`. For this situation it would be:

```
val data = Query(("Stanley", "Cut!"))

// data: slick.lifted.Query[
//  (slick.lifted.ConstColumn[String], slick.lifted.ConstColumn[String]),
//  (String, String),
//  Seq] = Rep(Pure $@1413606951)
```

`data` is a constant query that returns a fixed value—a tuple of two columns. It's the equivalent of running `SELECT 'Stanley', 'Cut!';` against the database, which is one part of the query we need.

We also need to be able to test to see if the data already exists. That's straightforward:

```
val exists =
  messages.
   filter(m => m.sender === "Stanley" && m.content === "Cut!").
   exists

// exists: slick.lifted.Rep[Boolean] = Rep(Apply Function exists)
```

We want to use the `data` when the row *doesn't* exist, so combine the `data` and `exists` with `filterNot` rather than `filter`:

```
val selectExpression = data.filterNot(_ => exists)

// selectExpression: slick.lifted.Query[
//  (slick.lifted.ConstColumn[String], slick.lifted.ConstColumn[String]),
//  (String, String),Seq] = Rep(Filter)
```

Finally, we need to apply this query with `forceInsertQuery`. But remember the column types for the insert and select need to match up. So we `map` on `messages` to make sure that's the case:

```
val action =
  messages
    .map(m => m.sender -> m.content)
    .forceInsertQuery(selectExpression)

exec(action)
// res13: Int = 1

exec(action)
//  res14: Int = 0
```

The first time we run the query, the message is inserted. The second time, no rows are affected.

In summary, `forceInsertQuery` provides a way to build-up more complicated inserts. If you find situations beyond the power of this method, you can always make use of Plain SQL inserts, described in Chapter 7.

3.2 Deleting Rows

Slick lets us delete rows using the same `Query` objects we saw in Chapter 2. That is, we specify which rows to delete using the `filter` method, and then call `delete`:

```
val removeHal: DBIO[Int] =
  messages.filter(_.sender === "HAL").delete

exec(removeHal)
// res1: Int = 4
```

The return value is the number of rows affected.

The SQL generated for the action can be seen by calling `delete.statements`:

```
messages.filter(_.sender === "HAL").delete.statements
// res2: Iterable[String] = List(
//   delete from "message"
//   where "message"."sender" = 'HAL')
```

Note that it is an error to use `delete` in combination with `map`. We can only call `delete` on a `TableQuery`:

```
messages.map(_.content).delete
// <console>:14: error: value delete is not a member of ↩
//   slick.lifted.Query[slick.lifted.Column[String],String,Seq]
//          messages.map(_.content).delete
//                                  ^
```

3.3 Updating Rows

So far we've only looked at inserting new data and deleting existing data. But what if we want to update existing data without deleting it first? Slick lets us create SQL `UPDATE` actions via the kinds of `Query` values we've been using for selecting and deleting rows.

3.3.1 Updating a Single Field

In the `Messages` we've created so far we've referred to the computer from *2001: A Space Odyssey* as "`HAL`", but the correct name is "`HAL 9000`". Let's fix that:

```
val updateQuery =
  messages.filter(_.sender === "HAL").map(_.sender)
//updateQuery: slick.lifted.Query[
//  slick.lifted.Rep[String],
//  String,Seq] = Rep(Bind)

exec(updateQuery.update("HAL 9000"))
// res1: Int = 2
```

We can retrieve the SQL for this query by calling `updateStatment` instead of `update`:

```
messages.filter(_.sender === "HAL").
  map(_.sender).updateStatement
// res2: String =
//   update "message"
//   set "sender" = ?
//   where "message"."sender" = 'HAL'
```

Let's break down the code in the Scala expression. By building our update query from the `messages` `TableQuery`, we specify that we want to update records in the `message` table in the database:

```
val messagesByHal = messages.filter(_.sender === "HAL")
// messagesByHal: scala.slick.lifted.Query[
//   Example.MessageTable,
//   Example.MessageTable#TableElementType,
//   Seq
// ] = Rep(Filter)
```

We only want to update the `sender` column, so we use `map` to reduce the query to just that column:

```
val halSenderCol  = messagesByHal.map(_.sender)
// halSenderCol: slick.lifted.Query[
//   slick.lifted.Rep[String],
//   String,
//   Seq
// ] = Rep(Bind)
```

Finally we call the `update` method, which takes a parameter of the *unpacked* type (in this case `String`), runs the query, and returns the number of affected rows:

```
val action: DBIO[Int] = halSenderCol.update("HAL 9000")

val rowsAffected = exec(action)
// rowsAffected: Int = 4
```

3.3.2 Updating Multiple Fields

We can update more than one field at the same time by mapping the query down to a tuple of the columns we care about...

```
val query = messages.
    filter(_.id === 4L).
    map(message => (message.sender, message.content))
// query: slick.lifted.Query[
//  (slick.lifted.Rep[String], slick.lifted.Rep[String]),
//  (String, String),
//  Seq] = Rep(Bind)
```

...and then supplying the tuple values we want to used in the update:

```
val action: DBIO[Int] =
  query.update(("HAL 9000", "Sure, Dave. Come right in."))

exec(action)
// res3: Int = 1

exec(messages.filter(_.sender === "HAL 9000").result)
// res4: Seq[Example.MessageTable#TableElementType] = Vector(
//   Message(HAL 9000,Affirmative, Dave. I read you.,2),
//   Message(HAL 9000,Sure, Dave. Come right in.,4))
```

Again, we can see the SQL we're running using the `updateStatement` method. The returned SQL contains two `?` placeholders, one for each field as expected:

```
messages.
  filter(_.id === 4L).
  map(message => (message.sender, message.content)).
  updateStatement
```

```
// res5: String =
//   update "message"
//   set "sender" = ?, "content" = ?
//   where "message"."id" = 4
```

We can even use the projection operator, <>, to manipulate the type of the parameter to `update`:

```
case class NameText(name: String, text: String)

messages.
  filter(_.id === 4L).
  map(m => (m.sender, m.content) <>
    (NameText.tupled, NameText.unapply)).
  update(NameText("Dave", "Now I totally don't trust you."))
```

We discuss <> in Chapter 5.

3.3.3 Updating with a Computed Value

Let's now turn to more interesting updates. How about converting every message to be all capitals? Or adding an exclamation mark to the end of each message? Both of these queries involve expressing the desired result in terms of the current value in the database. In SQL we might write something like:

```
update "message" set "content" = CONCAT("content", '!')
```

This is not currently supported by `update` in Slick, but there are ways to achieve the same result. One such way is to use Plain SQL queries, which we cover in Chapter 7. Another is to perform a *client-side update* by defining a Scala function to capture the change to each row:

```
def exclaim(msg: Message): Message =
  msg.copy(content = msg.content + "!")
// exclaim: (msg: Example.Message)Example.Message
```

We can update rows by selecting the relevant data from the database, applying this function, and writing the results back individually. Note that approach can be quite inefficient for large datasets—it takes `N + 1` queries to apply an update to `N` results.

You may be tempted to write something like this:

```
def modify(msg: Message): DBIO[Int] =
  messages.filter(_.id === msg.id).update(exclaim(msg))

// Don't do it this way:
for {
  msg <- exec(messages.result)
} yield exec(modify(msg))
```

This will have the desired effect, but at some cost. What we have done there is use our own `exec` method which will wait for results. We use it to fetch all rows, and then we use it on each row to modify the row. That's a lot of waiting. There is also no support for transactions as we `db.run` each action separately.

A better approach is to turn our logic into a single `DBIO` action using *action combinators*. This, together with transactions, is the topic of the next chapter.

However, for this particular example, we recommend using Plain SQL (Chapter 7) instead of client-side updates.

3.4 Take Home Points

For modifying the rows in the database we have seen that:

- inserts are via a += or ++= call on a table;
- updates are via an `update` call on a query, but are somewhat limited when you need to update using the existing row value; and
- deletes are via a `delete` call to a query.

Auto-incrementing values are inserted by Slick, unless forced. The auto-incremented values can be returned from the insert by using `returning`.

Databases have different capabilities. The limitations of each driver is listed in the driver's Scala Doc page.

3.5 Exercises

The code for this chapter is in the GitHub repository in the *chapter-03* folder. As with chapter 1 and 2, you can use the `run` command in SBT to execute the code against a H2 database.

> **Tip**
>
> **Where Did My Data Go?**
>
> Several of the exercises in this chapter require you to delete or update content from the database. If you get a different solution to provided, try ensuring the content of the database is correct by running:
>
> ```
> exec(populate)
> ```
>
> This will drop, create and populate the `messages` table with known values.

3.5.1 Methodical Inserts

Create a method to insert a message and return it with the `id` field populated.

See the solution

3.5.2 Get to the Specifics

In Inserting Specific Columns we looked at only inserting the sender column:

```
exec(messages.map(_.sender) += "HAL")
```

This failed as we didn't meet the requirements of the `message` table schema. For this to succeed we need to inlcude `content` as well as `sender`.

Rewrite the above query to include the `content` column.

See the solution

3.5.3 Bulk All the Inserts

Insert the conversation below between Alice and Bob, returning the messages populated with `ids`.

```
val conversation = List(
  Message("Bob",  "Hi Alice"),
  Message("Alice","Hi Bob"),
  Message("Bob",  "Are you sure this is secure?"),
  Message("Alice","Totally, why do you ask?"),
  Message("Bob",  "Oh, nothing, just wondering."),
  Message("Alice","Ten was too many messages"),
  Message("Bob",  "I could do with a sleep"),
  Message("Alice","Let's just to to the point"),
  Message("Bob",  "Okay okay, no need to be tetchy."),
  Message("Alice","Humph!"))
```

See the solution

3.5.4 No Apologies

Write a query to delete messages that contain “sorry”.

See the solution

3.5.5 Update Using a For Comprehension

Rewrite the update statement below to use a for comprehension.

```
val rowsAffected = messages.
  filter(_.sender === "HAL").
  map(msg => (msg.sender, msg.content)).
  update("HAL 9000", "Rebooting, please wait...")
```

Which style do you prefer?

See the solution

3.5.6 Selective Memory

Delete HALs first two messages. This is a more difficult exercise.

Hint: First write a query to select the two messages. Then see if you can find a way to use it as a subquery.

See the solution

Chapter 4

Combining Actions

At some point you'll find yourself writing a piece of code made up of multiple actions. You might need a simple sequence of actions to run one after another; or you might need something more sophisticated where one action depends on the results of another.

In Slick you use *action combinators* to turn a number of actions into a single action. You can then run this combined action just like any single action. You might also run these combined actions in a *transaction*.

This chapter focuses on these combinators. Some, such as `map`, `fold`, and `zip`, will be familiar from the Scala collections library. Others, such as `sequence` and `asTry` may be less familiar. We will give examples of how to use many of them in this chapter.

This is a key concept in Slick. Make sure you spend time getting comfortable with combining actions.

4.1 Combinators Summary

The temptation with multiple actions might be to run each action, use the result, and run another action. This will require you to deal with multiple `Future`s. We recommend you avoid that whenever you can.

Instead, focus on the actions and how they combine together, not on the messy details of running them. Slick provides a set of combinators to make this possible.

Before getting into the detail, take a look at the two tables below. They list out the key methods available on an action, and also the combinators available on `DBIO`.

Table 4.1: Combinators on action instances of `DBIOAction`, specifically a `DBIO[T]`. Types simplified.

Method	Arguments	Result Type	Notes
`map`	`T => R`	`DBIO[R]`	Execution context required
`flatMap`	`T => DBIO[R]`	`DBIO[R]`	*ditto*
`filter`	`T => Boolean`	`DBIO[T]`	*ditto*
`named`	`String`	`DBIO[T]`	
`zip`	`DBIO[R]`	`DBIO[(T,R)]`	
`asTry`		`DBIO[Try[T]]`	
`andThen` or `>>`	`DBIO[R]`	`DBIO[Unit]`	Example in Chapter 1.
`andFinally`	`DBIO[_]`	`DBIO[T]`	
`cleanUp`	`Option[Throwable]=>DBIO[_]`	`DBIO[T]`	Execution context required

Method	Arguments	Result Type	Notes
`failed`		`DBIO[Throwable]`	

Table 4.2: Combinators on `DBIO` object, with types simplified.

Method	Arguments	Result Type	Notes
`sequence`	`TraversableOnce[DBIO[T]]`	`DBIO[TraversableOnce[T]]`	
`seq`	`DBIO[_]*`	`DBIO[Unit]`	Combines actions, ignores results
`from`	`Future[T]`	`DBIO[T]`	
`successful`	`V`	`DBIO[V]`	
`failed`	`Throwable`	`DBIO[Nothing]`	
`fold`	`(Seq[DBIO[T]], T)` `(T,T)=>T`	`DBIO[T]`	Execution context required

4.2 Combinators in Detail

Advanced

Combined Actions Are Not Automatically Transactions

By default, when you combine actions together you do not get a single transaction. At the end of this chapter we'll see that it's very easy to run combined actions in a transaction with:

```
db.run(actions.transactionally)
```

4.2.1 `andThen` (or `>>`)

The simplest way to run one action after another is perhaps `andThen`. The combined actions are both run, but only the result of the second is returned:

```
val reset: DBIO[Int] =
  messages.delete andThen messages.size.result

exec(reset)
// res1: Int = 0
```

The result of the first query is ignored, so we cannot use it. Later we will see how `flatMap` allows us to use the result to make choices about which action to run next.

4.2.2 `DBIO.seq`

If you have a bunch of actions you want to run, you can use `DBIO.seq` to combine them:

```
val reset: DBIO[Unit] =
  DBIO.seq(messages.delete, messages.size.result)
```

This is rather like combining the actions with andThen, but even the last value is discarded.

4.2.3 map

Mapping over an action is a way to set up a transformation of a value from the database. The transformation will run on the result of the action when it is returned by the database.

As an example, we can create an action to return the content of a message, but reverse the text:

```
val text: DBIO[Option[String]] =
  messages.map(_.content).result.headOption

val backwards: DBIO[Option[String]] =
  text.map( optionalContent => optionalContent.map(_.reverse) )

exec(backwards)
// res1: Option[String] =
//  Option[String] = Some(?LAH ,em daer uoy oD .LAH ,olleH)
```

Here we have created an action called backwards that, when run, ensures a function is applied to the result of the text action. In this case the function is to apply reverse to an optional String.

Note that we have made three uses of map in this example:

- an Option map to apply reverse to our Option[String] result;
- a map on a query to select just the content column; and
- map on our action so that the result will be transform when the action is run.

Combinators everywhere!

This example transformed an Option[String] to another Option[String]. As you may expect if map changes the type of a value, the type of DBIO changes too:

```
text.map(os => os.map(_.length))
// res2: slick.dbio.DBIOAction[
//   Option[Int],
//   slick.dbio.NoStream,
//   slick.dbio.Effect.All
// ]
```

> **Tip**
>
> **Execution Context Required**
>
> Some methods require an execution context and some don't. For example, map does, but andThen does not. What gives?
>
> The reason is that map allows you to call arbitrary code when joining the actions together. Slick cannot allow that code to be run on its own execution context, because it has no way to know if you are going to tie up Slicks threads for a long time.
>
> In contrast, methods such as andThen which combine actions without custom code can be run on Slick's

> own execution context. Therefore, you do not need an execution context available for andThen.
>
> You'll know if you need an execution context, because the compiler will tell you:
>
> ```
> error: Cannot find an implicit ExecutionContext. You might pass
>   an (implicit ec: ExecutionContext) parameter to your method
>   or import scala.concurrent.ExecutionContext.Implicits.global.
> ```
>
> The Slick manual discusses this in the section on Database I/O Actions.

4.2.4 `DBIO.successful` and `DBIO.failed`

When combining actions you will sometimes need to create an action that represents a simple value. Slick provides `DBIO.successful` for that purpose:

```
val v: DBIO[Int] = DBIO.successful(100)
// v: slick.dbio.DBIO[Int] = SuccessAction(100)
```

We'll see an example of this when we discuss `flatMap`.

And for failures, the value is a `Throwable`:

```
val v: DBIO[Nothing] =
  DBIO.failed(new RuntimeException("pod bay door unexpectedly locked"))
// v: slick.dbio.DBIO[Nothing] = FailureAction(
//  java.lang.RuntimeException: pod bay door unexpectedly locked)
```

This has a particular role to play inside transactions, which we cover later in this chapter.

> **Tip**
>
> **Error: value successful is not a member of object slick.dbio.DBIO**
>
> Due to a bug in Scala you may experience something like the above error when using `DBIO` methods on the REPL with Slick 3.0. This is resolved in Slick 3.1.
>
> If you do encounter it, and have to stay with Slick 3.0, you can carry on by writing your code in a `.scala` source file and running it from SBT.

4.2.5 `flatMap`

Ahh, `flatMap`. Wonderful `flatMap`. This method gives us the power to sequence actions and decide what we want to do at each step.

The signature of `flatMap` should feel similar to the `flatMap` you see elsewhere:

```
// Simplified:
def flatMap[S](f: R => DBIO[S])(implicit e: ExecutionContext): DBIO[S]
```

That is, we give `flatMap` a function that depends on the value from an action, and evaluates to another action.

As an example, let's write a method to remove all the crew's messages, and post a message saying how many messages were removed. This will involve an `INSERT` and a `DELETE`, both of which we're familiar with:

```
val delete: DBIO[Int] =
  messages.delete

def insert(count: Int) =
  messages += Message("NOBODY", s"I removed ${count} messages")
```

The first thing `flatMap` allows us to do is run these actions in order:

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

val resetMessagesAction: DBIO[Int] =
  delete.flatMap{ count => insert(count) }

exec(resetMessagesAction)
// res1: Int = 1
```

This single action produces the two SQL expressions you'd expect:

```
delete from "message";
insert into "message" ("sender","content")
  values ('NOBODY', 'I removed 4 messages');
```

Beyond sequencing, `flatMap` also gives us control over which actions are run. To illustrate this we will change `resetMessagesAction` to not insert a message if no messages were removed in the first step:

```
val resetMessagesAction: DBIO[Int] =
  delete.flatMap {
    case 0 => DBIO.successful(0)
    case n => insert(n)
  }
```

We've decided a result of `0` is right if no message was inserted. But the point here is that `flatMap` gives us arbitrary control over how actions can be combined.

Occasionally the compiler will complain about a `flatMap` and need your help to figuring out the types. Recall that `DBIO[T]` is an alias for `DBIOAction[T,S,E]`, encoding streaming and effects. When mixing effects, such as inserts and selects, you may need to explicitly specify the type parameters to apply to the resulting action:

```
query.flatMap[Int, NoStream, Effect.All] { result => ... }
```

...but in many cases the compiler will figure these out for you.

> **Tip**
>
> **Do it in the database if you can**
>
> Combining actions to sequence queries is a powerful feature of Slick. However, you may be able to reduce multiple queries into a single database query. If you can do that, you're probably better off doing it.
>
> As an example, you could implement "insert if not exists" like this:

```
// Not the best way:
def insertIfNotExists(m: Message): DBIO[Int] = {
  val alreadyExists =
    messages.filter(_.content === m.content).result.headOption
  alreadyExists.flatMap {
    case Some(m) => DBIO.successful(0)
    case None    => messages += m
  }
}
```

...but as we saw earlier in "More Control over Inserts" you can achieve the same effect with a single SQL statement.

One query can often (but doesn't always) perform better than a sequence of queries. You mileage may vary.

4.2.6 `DBIO.sequence`

Despite the similarity in name to `DBIO.seq`, `DBIO.sequence` has a different purpose. It takes a sequence of `DBIO`s and gives back a `DBIO` of a sequence. That's a bit of a mouthful, but an example may help.

At the end of the last chapter we attempted to update rows based on their current value. Here we'll say we want to reverse the text of every message. We start with this:

```
def reverse(msg: Message): DBIO[Int] =
  messages.filter(_.id === msg.id).
  map(_.content).
  update(msg.content.reverse)
```

That's a straightforward method that returns an update action for one message. We can apply it to every message....

```
// Don't do this
val updates: DBIO[Seq[DBIO[Int]]] =
  messages.result.
  map(msgs => msgs.map(reverse))
```

...which will give us an action that returns actions! Note the crazy type signature.

You can find yourself in this awkward situation when you're trying to do something like a join, but not quite. The puzzle is how to run this kind of a beast.

This is where `DBIO.sequence` saves us. Rather than produce many actions via `msgs.map(reverse)` we use `DBIO.sequence` to return a single action:

```
val updates: DBIO[Seq[Int]] =
  messages.result.
  flatMap(msgs => DBIO.sequence(msgs.map(reverse)))
```

The difference is:

- we've wrapped the `Seq[DBIO]` with `DBIO.sequence` to give a single `DBIO[Seq[Int]]`; and

- we use `flatMap` to combine the sequence with the original query.

The end result is a sane type which we can run like any other action.

Of course this one action turns into many SQL statements:

```
select "sender", "content", "id" from "message"
update "message" set "content" = ? where "message"."id" = 1
update "message" set "content" = ? where "message"."id" = 2
update "message" set "content" = ? where "message"."id" = 3
update "message" set "content" = ? where "message"."id" = 4
```

4.2.7 `DBIO.fold`

Recall that many Scala collections support `fold` as a way to combine values:

```
List(3,5,7).fold(1) { (a,b) => a * b }
// res1: Int = 105
```

You can do the same kind of thing in Slick: when you need to run a sequence of actions, and reduce the results down to a value, you use `fold`.

As an example, suppose we have a number of reports to run. We want to summarize all these reports to a single number.

```
val report1: DBIO[Int] = ...
val report2: DBIO[Int] = ...

val reports: List[DBIO[Int]] =
  report1 :: report2 :: Nil
```

We can `fold` those `reports` with a function.

But we also need to consider our starting position:

```
val default: Int = 0
```

Finally we can produce an action to summarize the reports:

```
val summary: DBIO[Int] =
  DBIO.fold(reports, default) {
    (total, report) => total + report
}
```

`DBIO.fold` is a way to combine actions, such that the results are combined by a function you supply. As with other combinators, your function isn't run until we execute the action itself. In this case all our reports are run, and the sum of the values reported.

4.2.8 `zip`

We've seen how `DBIO.seq` combines actions and ignores the results. We've also seen that `andThen` combines actions and keeps one result. If you want to keep both results, `zip` is the combinator for you:

```
val countAndHal: DBIO[(Int, Seq[Message])] =
  messages.size.result zip messages.filter(_.sender === "HAL").result

exec(countAndHall)
// res1: (Int, Seq[Example.Message]) =
//  (4,
//   Vector(
//    Message(HAL,Affirmative, Dave. I read you.,8),
//    Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,10)
//   )
// )
```

The action returns a tuple representing the results of both queries.

4.2.9 `andFinally` and `cleanUp`

The two methods `cleanUp` and `andFinally` act a little like Scala's `catch` and `finally`.

`cleanUp` runs after an action completes, and has access to any error information as an `Option[Throwable]`:

```
// Let's record problems we encounter:
def log(err: Throwable): DBIO[Int] =
  messages += Message("SYSTEM", err.getMessage)

// Pretend this is important work which might fail:
val work =
  DBIO.failed(new RuntimeException("Boom!"))

val action =
  work.cleanUp {
    case Some(err) => log(err)
    case None      => DBIO.successful(0)
  }

exec(action)
// java.lang.RuntimeException: Boom!
//  ... 45 elided

exec(messages.filter(_.sender === "SYSTEM").result)
// res1: Seq[Example.MessageTable#TableElementType] =
//  Vector(Message(SYSTEM,Boom!,11))
```

Notice the result is still the original exception, but `cleanUp` has produced a side-effect for us.

Both `cleanUp` and `andFinally` run after an action, regardless of whether it succeeds or fails. `cleanUp` runs in response to a previous failed action; `andFinally` runs all the time, regardless of success or failure, and has no access to the `Option[Throwable]` that `cleanUp` sees.

4.2.10 `asTry`

Calling `asTry` on an action changes the action's type from a `DBIO[T]` to a `DBIO[Try[T]]`. This means you can work in terms of Scala's `Success[T]` and `Failure` instead of exceptions.

Suppose we had an action that might throw an exception:

```
val work =
  DBIO.failed(new RuntimeException("Boom!"))
```

We can place this inside `Try` by combining the action with `asTry`:

```
exec(work.asTry)
// res1: scala.util.Try[Nothing] =
//  Failure(java.lang.RuntimeException: Boom!)
```

And successful actions will evauluate to a `Success[T]`:

```
exec(messages.size.result.asTry)
// res2: scala.util.Try[Int] =
//  Success(4)
```

4.3 Logging Queries and Results

With actions combined together, it's useful to see the queries that are being executed.

We've seen how to retrieve the SQL of a query using `insertStatement` and similar methods on a query, or the `statements` method on an action. These are useful for experimenting with Slick, but sometimes we want to see all the queries *when Slick executes them*. We can do that by configuring logging.

Slick uses a logging interface called SLF4J. We can configure this to capture information about the queries being run. The `build.sbt` files in the exercises use an SLF4J-compatible logging back-end called Logback, which is configured in the file *src/main/resources/logback.xml*. In that file we can enable statement logging by turning up the logging to debug level:

```
<logger name="slick.jdbc.JdbcBackend.statement" level="DEBUG"/>
```

This causes Slick to log every query, even modifications to the schema:

```
DEBUG slick.jdbc.JdbcBackend.statement - Preparing statement: ↩
  delete from "message" where "message"."sender" = 'HAL'
```

We can change the level of various loggers, as shown in the table below:

Table 4.3: Slick loggers and their effects.

Logger	Effect
`slick.jdbc.JdbcBackend.statement`	Logs SQL sent to the database as described above.
`slick.jdbc.StatementInvoker.result`	Logs the first few results of each query.
`slick.session`	Logs session events such as opening/closing connections.
`slick`	Logs everything! Equivalent to changing all of the above.

The `StatementInvoker.result` logger, in particular, is pretty cute:

```
SI.result - /--------+----------------------+----\
SI.result - | sender | content              | id |
SI.result - +--------+----------------------+----+
SI.result - | HAL    | Affirmative, Dave... | 2  |
SI.result - | HAL    | I'm sorry, Dave. ... | 4  |
SI.result - \--------+----------------------+----/
```

4.4 Transactions

So far each of the changes we've made to the database have run independently of the others. That is, each insert, update, or delete query we run can succeed or fail independently of the rest.

We often want to tie sets of modifications together in a *transaction* so that they either *all* succeed or *all* fail. We can do this in Slick using the `transactionally` method.

As an example, let's re-write the script. We want to make sure the script changes all complete or nothing changes:

```
def updateContent(id: Long) =
  messages.filter(_.id === id).map(_.content)

exec {
  (updateContent(2L).update("Wanna come in?") andThen
   updateContent(3L).update("Pretty please!") andThen
   updateContent(4L).update("Opening now.") ).transactionally
}

exec(messages.result)
// res1: Seq[Example.MessageTable#TableElementType] = Vector(
//   Message(Dave,Hello, HAL. Do you read me, HAL?,1),
//   Message(HAL,Wanna come in?,2),
//   Message(Dave,Pretty please!,3),
//   Message(HAL,Opening now.,4))
```

The changes we make in the `transactionally` block are temporary until the block completes, at which point they are *committed* and become permanent.

To manually force a rollback you need to call `DBIO.failed` with an appropriate exception.

```
val willRollback = (
  (messages += Message("HAL",  "Daisy, Daisy..."))                   >>
  (messages += Message("Dave", "Please, anything but your singing")) >>
  DBIO.failed(new Exception("agggh my ears"))                        >>
  (messages += Message("HAL", "Give me your answer do"))
  ).transactionally

exec(willRollback.asTry)
// scala.util.Try[Int] =
//  Failure(java.lang.Exception: agggh my ears)
```

The result of running `willRollback` is that the database won't have changed. Inside of transactional block you would see the inserts until `DBIO.failed` is called.

If we removed the `.transactionally` that is wrapping our combined actions, the first two inserts would succeed, even though the combined action failed.

4.5 Take Home Points

Inserts, selects, deletes and other forms of Database Action can be combined using `flatMap` and other combinators. This is a powerful way to sequence actions, and make actions depend on the results of other actions.

Combining actions avoid having to deal with awaiting results or having to sequence `Futures` yourself.

We saw that the SQL statements executed and the result returned from the database can be monitored by configuring the logging system.

Finally, we saw that actions that are combined together can also be run inside a transaction.

4.6 Exercises

4.6.1 And Then what?

In Chapter 1 we create a schema and populate the database as separate actions. Use your newly found knowledge to combine them.

This exercise expects to start with an empty database. If you're already in the REPL and the database exists, you'll need to drop the table first:

```
val drop:     DBIO[Unit]        = messages.schema.drop
val create:   DBIO[Unit]        = messages.schema.create
val populate: DBIO[Option[Int]] = messages ++= testData

exec(drop)
exec(create)
exec(populate)
```

See the solution

4.6.2 First!

Create a method that will insert a message, but if it is the first message in the database, automatically insert the message "First!" before it.

Your method signature should be:

```
def insert(m: Message): DBIO[Int]
```

Use your knowledge of the `flatMap` action combinator to achieve this.

See the solution

4.6.3 There Can be Only One

Implement `onlyOne`, a method that guarantees that an action will return only one result. If the action returns anything other than one result, the method should fail with an exception.

Below is the method signature and two test cases:

```
def onlyOne[T](xs:DBIO[Seq[T]]):DBIO[T] = ???
```

In the example there is only one message that contains the word "Sorry", so we expect `onlyOne` to return that row:

```
val happy = messages.filter(_.content like "%sorry%").result

exec(onlyOne(happy))
//res25: Example.MessageTable#TableElementType =
// Message(HAL, I'm sorry, Dave. I'm afraid I can't do that., 4)
```

However, there are two messages containing the word "I". In this case `onlyOne` will fail:

```
val boom  = messages.filter(_.content like "%I%").result
exec(onlyOne(boom))
//java.lang.RuntimeException: Expected 1 result, not 2
//  ...
```

Hints: The signature of `onlyOne` is telling us we will take an action that produces a `Seq[T]` and return an action that produces a `T`. That tells us we need an action combinator here. That fact that the method may fail means we want to use `DBIO.successful` and `DBIO.failed` in there somewhere.

See the solution

4.6.4 Let's be Reasonable

Some *fool* is throwing exceptions in our code, destroying our ability to reason about it. Implement `exactlyOne` which wraps `onlyOne` encoding the possibility of failure using types rather than exceptions.

Then rerun the test cases.

See the solution

4.6.5 Filtering

There is a `DBIO filter` method, but it produces a runtime exception if the filter predicate is false. It's like `Future`'s `filter` method in that respect. We've not found a situation where we need it.

However, we can create our own kind of filter. It can take some alternative action when the filter predicate fails.

The signature could be:

```
def myFilter[T]
  (action: DBIO[T])
  (p: T => Boolean)
  (alternative: => T) = ???
```

If you're not comfortable with the `[T]` type parameter, or the by name parameter on `alternative`, just use `Int` instead:

```
def myFilter
  (action: DBIO[Int])
  (p: Int => Boolean)
  (alternative: Int) = ???
```

Go ahead and implement `myFilter`.

We have an example usage from the ship's marketing department. They are happy to report the number of chat messages, but only if that number is at least 100:

```
val marketingCount = exec(
  myFilter(messages.size.result)( _ > 100)(100)
)
```

See the solution

4.6.6 Unfolding

This is a challenging exercise.

We saw that `fold` can take a number of actions and reduce them using a function you supply. Now imagine the opposite: unfolding an initial value into a sequence of values via a function. In this exercise we want you to write an `unfold` method that will do just that.

Why would you need to do something like this? One example would be when you have a tree structure represented in a database and need to search it. You can follow a link between rows, possibly recording what you find as you follow those links.

As an example, let's pretend the crew's ship is a set of rooms, one connected to just one other:

```
final case class Room(name: String, connectsTo: String)

final class FloorPlan(tag: Tag) extends Table[Room](tag, "floorplan") {
  def name       = column[String]("name")
  def connectsTo = column[String]("next")
  def * = (name, next) <> (Room.tupled, Room.unapply)
}

lazy val floorplan = TableQuery[FloorPlan]

exec {
  (floorplan.schema.create) >>
  (floorplan += Room("Outside",     "Podbay Door")) >>
  (floorplan += Room("Podbay Door", "Podbay"))      >>
  (floorplan += Room("Podbay",      "Galley"))      >>
  (floorplan += Room("Galley",      "Computer"))    >>
  (floorplan += Room("Computer",    "Engine Room"))
}
```

For any given room it's easy to find the next room. For example:

```
SELECT
  "connectsTo"
FROM
  "foorplan"
WHERE
  "name" = 'Podbay'

-- Returns 'Galley'
```

Write a method `unfold` that will take any room name as a starting point, and a query to find the next room, and will follow all the connections until there are no more connecting rooms.

The signature of `unfold` *could* be:

```
def unfold(
  z: String,
  f: String => DBIO[Option[String]]
  ): DBIO[Seq[String]]
```

... where `z` is the starting ("zero") room, and `f` will lookup the connecting room.

If `unfold` is given `"Podbay"` as a starting point it should return an action which, when run, will produce: `Seq("Podbay", "Galley", "Computer", "Engine Room")`.

See the solution

Chapter 5

Data Modelling

We can do the basics of connecting to a database, running queries, and changing data. We turn now to richer models of data and how our application hangs together.

In this chapter we will:

- understand how to structure an application;
- look at alternatives to modelling rows as case classes;
- store richer data types in columns; and
- expand on our knowledge of modelling tables to introduce optional values and foreign keys.

To do this, we'll expand the chat application schema to support more than just messages.

5.1 Application Structure

So far, all of our examples have been written in a single Scala file. This approach obviously doesn't scale to larger application codebases. In this section we'll explain how to split up application code into modules.

Until now we've also been exclusively using Slick's H2 driver. When writing real applications we often need to be able to switch drivers in different circumstances. For example, we may use PostgreSQL in production and H2 in our unit tests.

An example of this pattern can be found in the example project, folder *chapter-05*, file *structure.scala*.

5.1.1 Abstracting over Databases

Let's look at how we can write code that works with multiple different database drivers. When we previously wrote:

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

We now have to write an import that works with a variety of drivers. Fortunately, Slick provides a common supertype for the drivers for the most popular databases—a trait called `JdbcProfile`:

```
import slick.driver.JdbcProfile
```

> **Tip**
>
> *Drivers and Profiles*
>
> Slick uses the words "driver" and "profile" interchangeably. We'll start referring to Slick drivers as "profiles" here to distinguish them from the JDBC drivers that sit lower in the code.

We can't import directly from `JdbcProfile` because it isn't a concrete object. Instead, we have to *inject a dependency* of type `JdbcProfile` into our application and import from that. The basic pattern we'll use is as follows:

- isolate our database code into a trait (or a few traits);
- declare the Slick profile as an abstract `val` and import from that; and
- extend our database trait to make the profile concrete.

Here's the simplest form of this pattern:

```
trait DatabaseModule {
  // Declare an abstract profile:
  val profile: JdbcProfile

  // Import the Slick API from the profile:
  import profile.api._

  // Write our database code here...
}

object Main extends App {
  // Instantiate the database module, assigning a concrete profile:
  val databaseLayer = new DatabaseModule {
    val profile = slick.driver.H2Driver
  }
}
```

In this pattern, we declare our profile using an abstract `val`. Surprisingly, this is enough to allow us to write `import profile.api._`. The compiler knows that the `val` is *going to be* an immutable `JdbcProfile` even if we haven't yet said which one. When we instantiate the `DatabaseModule` we bind `profile` to our profile of choice.

5.1.2 Scaling to Larger Codebases

As our applications get bigger, we need to split our code up into multiple files to keep it manageable. We can do this by extending the pattern above to a family of traits:

```
trait Profile {
  val profile: JdbcProfile
}

trait DatabaseModule1 { self: Profile =>
  import profile.api._
```

```
  // Write database code here
}

trait DatabaseModule2 { self: Profile =>
  import profile.api._

  // Write more database code here
}

// Mix the modules together:
class DatabaseLayer(val profile: JdbcProfile)
  extends Profile
  with DatabaseModule1
  with DatabaseModule2

// Instantiate the modules and inject a profile:
object Main extends App {
  val databaseLayer = new DatabaseLayer(slick.driver.H2Driver)
}
```

Here we factor out our `profile` dependency into its own `Profile` trait. Each module of database code specifies `Profile` as a self-type, meaning it can only be extended by a class that also extends `Profile`. This allows us to share the `profile` across our family of modules.

To work with a different database, we simply inject a different profile when we instantiate the database code:

```
val anotherDatabaseLayer = new DatabaseLayer(slick.driver.PostgresDriver)
```

This basic pattern is reasonable way of structuring your application.

5.2 Representations for Rows

In previous chapters we modelled rows as case classes. Although this is the most common usage pattern, and the one we recommend, there are several representation options available, including tuples, case classes, and `HLists`. Let's investigate these by looking in more detail at how Slick relates columns in our database to fields in our classes.

5.2.1 Projections, `ProvenShapes`, and `<>`

When we declare a table in Slick, we are required to implement a $*$ method that specifies a "default projection":

```
final class MyTable(tag: Tag) extends Table[(String, Int)](tag, "mytable") {
  def column1 = column[String]("column1")
  def column2 = column[Int]("column2")
  def * = (column1, column2)
}
```

Projections provide mappings between database columns and Scala values. In the code above, the definition of $*$ is mapping `column1` and `column2` from the database to the `(String, Int)` tuples defined in the `extends Table` clause.

If we look at the definition of `*` in the `Table` class, we see something confusing:

```
abstract class Table[T] {
  def * : ProvenShape[T]
}
```

The type of `*` is actually something called a `ProvenShape`, not a tuple of columns as we specified in our example. There is clearly some cleverness here—Slick is using implicit conversions to build a `ProvenShape` object from the columns we provided.

The internal workings of `ProvenShape` are certainly beyond the scope of this book. Suffice to say that Slick can use any Scala type as a projection provided it can generate a compatible `ProvenShape`. If we look at the rules for `ProvenShape` generation, we will get an idea about what data types we can map. Here are the three most common use cases:

1. Single `column` definitions produce shapes that map the column contents to a value of the column's type parameter. For example, a column of `Rep[String]` maps a value of type `String`:

   ```
   final class MyTable(tag: Tag) extends Table[String](tag, "mytable") {
     def column1 = column[String]("column1")
     def * = column1
   }
   ```

2. Tuples of database columns map tuples of their type parameters. For example, `(Rep[String], Rep[Int])` is mapped to `(String, Int)`:

   ```
   final class MyTable(tag: Tag)
       extends Table[(String, Int)](tag, "mytable") {
     def column1 = column[String]("column1")
     def column2 = column[Int]("column2")
     def * = (column1, column2)
   }
   ```

3. If we have a `ProvenShape[A]`, we can convert it to a `ProvenShape[B]` using the "projection operator" `<>`. We supply functions to convert each way between `A` and `B` and Slick builds the resulting shape:

   ```
   final class UserTable(tag: Tag) extends Table[User](tag, "user") {
     def id   = column[Long]("id", O.PrimaryKey, O.AutoInc)
     def name = column[String]("name")
     def * = (name, id) <> (User.tupled, User.unapply)
   }
   ```

The projection operator `<>` is the secret ingredient that allows us to map a wide variety of types. As long as we can convert a tuple of columns to and from some type `B`, we can store instances of `B` in a database.

The two arguments to `<>` are:

- a function from `A => B`, which converts from the existing shape's unpacked row-level encoding `(String, Long)` to our preferred representation (`User`);
- a function from `B => Option[A]`, which converts the other way.

We can supply these functions by hand if we want:

```
def intoUser(pair: (String, Long)): User =
  User(pair._1, pair._2)

def fromUser(user: User): Option[(String, Long)] =
  Some((user.name, user.id))
```

and write:

```
def * = (name, id) <> (intoUser, fromUser)
```

In the `User` example, the case class supplies these functions via `User.tupled` and `User.unapply`, so we don't need to build them ourselves. However it is useful to remember that we can provide our own functions for more elaborate packaging and unpackaging of rows. We will see this in one of the exercises in this section.

5.2.2 Tuples versus Case Classes

We've seen how Slick is able to map case classes and tuples of values. But which should we use? In one sense there is little difference between case classes and tuples—both represent fixed sets of values. However, case classes differ from tuples in two important respects:

1. Case classes have field names, which improves code readability:

```
val user = User("Dave", 0L)
user.name // case class field access

val tuple = ("Dave", 0L)
tuple._1 // tuple field access
```

2. Case classes have types that distinguish them from other case classes with the same field types:

```
val user = User("Dave", 0L)
val dog  = Dog("Lassie", 0L)

user == dog // false -- different types
```

As a general rule, we recommend using case classes to represent database rows for these reasons.

> **Tip**
>
> **Expose Only What You Need**
>
> We can hide information by excluding it from our row definition. The default projection controls what is returned, in what order, and is driven by our row definition.
>
> For example, we don't need to map everything in a table with legacy columns that aren't being used.

5.2.3 Heterogeneous Lists

We've seen how Slick can map database tables to tuples and case classes. Scala veterans identify a key weakness in this approach—tuples and case classes don't scale beyond 22 fields[1].

Many of us have heard horror stories of legacy tables in enterprise databases that have tens or hundreds of columns. We can't map everything in these tables using the tuplebased approach described above. Fortunately, Slick provides an `HList` implementation to support tables with very large numbers of columns.

To motivate this, let's consider a poorly-designed legacy table for storing product attributes:

```
final class AttrTable(tag: Tag) extends Table[Attr](tag, "attrs") {
  def id        = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def productId = column[Long]("product_id")
  def name1     = column[String]("name1")
  def value1    = column[Int]("value1")
  def name2     = column[String]("name2")
  def value2    = column[Int]("value2")
  def name3     = column[String]("name3")
  def value3    = column[Int]("value3")
  def name4     = column[String]("name4")
  def value4    = column[Int]("value4")
  def name5     = column[String]("name5")
  def value5    = column[Int]("value5")
  def name6     = column[String]("name6")
  def value6    = column[Int]("value6")
  def name7     = column[String]("name7")
  def value7    = column[Int]("value7")
  def name8     = column[String]("name8")
  def value8    = column[Int]("value8")
  def name9     = column[String]("name9")
  def value9    = column[Int]("value9")
  def name10    = column[String]("name10")
  def value10   = column[Int]("value10")
  def name11    = column[String]("name11")
  def value11   = column[Int]("value11")
  def name12    = column[String]("name12")
  def value12   = column[Int]("value12")

  def * = ??? // we'll fill this in below
}
```

Hopefully you don't have a table like this at your organization, but accidents do happen.

This table has 26 columns—too many to model using tuples and <>. Fortunately, Slick provides an alternative mapping representation that scales to arbitrary numbers of columns. This new representation is called a *heterogeneous list* or `HList`.

An `HList` is a sort of hybrid of a list and a tuple. It has an arbitrary length like a `List`, but each element can be a different type like a tuple. Here are some examples:

[1] Scala 2.11 introduced the ability to define case classes with more than 22 fields, but tuple and function arities are still limited to 22.

```
import slick.collection.heterogeneous.{ HList, HCons, HNil }

val emptyHList: HNil =
  HNil

val shortHList: Int :: HNil =
  123 :: HNil

val longerHList: Int :: String :: Boolean :: HNil =
  123 :: "abc" :: true :: HNil
```

HLists are constructed recursively like Lists, allowing us to model arbitrarily large collections of values:

- an empty HList is represented by the singleton object HNil;
- longer HLists are formed by prepending values using the :: operator, which creates a new list *of a new type*.

Notice the the types and values of each HList mirror each other: the longerHList comprises values of types Int, String, and Boolean, and its type comprises the types Int, String, and Boolean as well. Because the element types are preserved, we can write code that takes each precise type into account.

Slick is able to produce ProvenShapes to map HLists of columns to HLists of their values. For example, the shape for a Rep[Int] :: Rep[String] :: HNil maps values of type Int :: String :: HNil.

We can use an HList to map the large table in our example above. Here's what the default projection looks like:

```
import slick.collection.heterogeneous.{ HList, HCons, HNil }
import slick.collection.heterogeneous.syntax._

type AttrHList =
  Long :: Long ::
  Int :: String :: Int :: String :: Int :: String ::
  Int :: String :: Int :: String :: Int :: String ::
  Int :: String :: Int :: String :: Int :: String ::
  Int :: String :: Int :: String :: Int :: String ::
  HNil

final class AttrTable(tag: Tag) extends Table[AttrHList](tag, "attrs") {
  // Column definitions omitted

  def * = id :: productId ::
          name1 :: value1 :: name2 :: value2 :: name3 :: value3 ::
          name4 :: value4 :: name5 :: value5 :: name6 :: value6 ::
          name7 :: value7 :: name8 :: value8 :: name9 :: value9 ::
          name10 :: value10 :: name11 :: value11 :: name12 :: value12 ::
          HNil
}

val AttrTable = TableQuery[AttrTable]
```

Writing HList types and values is cumbersome and error prone, so we've introduced a type alias for AttrHList to avoid as much typing as we can.

Working with this table involves inserting, updating, selecting, and modifying instances of `AttrHList`. For example:

```
AttrTable += 0L :: productId ::
  "name1" :: 1 :: "name2" :: 2 :: "name3" :: 3 ::
  "name4" :: 4 :: "name5" :: 5 :: "name6" :: 6 ::
  "name7" :: 7 :: "name8" :: 8 :: "name9" :: 9 ::
  "name10" :: 10 :: "name11" :: 11 :: "name12" :: 12 ::
  HNil

val myAttrs: AttrHList =
  exec(AttrTable.find(_.productId === productId).result.head)
```

We can extract values from our query results `HList` using pattern matching or a variety of type-preserving methods defined on `HList`, including `head`, `apply`, `drop`, and `fold`:

```
// Extracting values using pattern matching...
myAttrs match {
  case id :: pId :: n1 :: v1 :: n2 :: v2 :: _ =>
    // The types of each member are preserved:
    //  - id and pId are Longs
    //  - n1 and n2 are Strings
    //  - v1 and v2 are Ints
}

// Extracting values using methods...
val id: Long = myAttrs.head
val productId: Long = myAttrs.tail.head
val name1: String = myAttrs(2)
val value1: String = myAttrs(3)
// And so on...
```

In practice we'll want to map instances of `AttrHList` to a regular class to make them easier to work with. Fortunately Slick's <> operator works with `HList` shapes as well as tuple shapes. We have to produce our own mapping functions in place of `apply` and `unapply`, but otherwise this approach is the same as we've seen for tuples:

```
case class Attrs(id: Long, productId: Long,
  name1: String, value1: Int, name2: String, value2: Int, /* etc */)

object Attrs {
  type AttrHList = Long :: Long ::
    String :: Int :: String :: Int :: /* etc */ :: HNil

  def hlistApply(hlist: AttrHList): Attrs = hlist match {
    case id :: pId :: n1 :: v1 :: n2 :: v2 :: /* etc */ :: HNil =>
      Attrs(id, pId, n1, v1, n2, v2, /* etc */)
  }

  def hlistUnapply(a: Attrs): Option[AttrHList] =
    Some(a.id :: a.productId ::
      a.name1 :: a.value1 :: a.name2 :: a.value2 :: /* etc */ :: HNil)
```

```
}

final class AttrTable(tag: Tag) extends Table[Attrs](tag, "attributes") {
  def id        = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def productId = column[Long]("product_id")
  def name1     = column[String]("name1")
  def value1    = column[Int]("value1")
  /* etc */

  def * = (
    id :: productId ::
    name1 :: value1 :: name2 :: value2 :: name3 :: value3 ::
    name4 :: value4 :: name5 :: value5 :: name6 :: value6 ::
    name7 :: value7 :: name8 :: value8 :: name9 :: value9 ::
    name10 :: value10 :: name11 :: value11 :: name12 :: value12 ::
    HNil
  ) <> (Attrs.hlistApply, Attrs.hlistUnapply)
}
```

Now our table is defined on a plain Scala class, we can query and modify the data using regular data objects as normal:

```
AttrTable += Attrs(0L, productId, "n1", 1, "n2", 2, /* etc */)

val myAttrs: Attrs =
  exec(AttrTable.find(_.productId === productId).result.head)
```

As you can see, typing all of the code to define `HList` mappings by hand is error prone and likely to induce stress. There are two ways to improve on this:

- The first is to know that Slick can *generate* this code for us from an existing database. If our main use for `HLists` is to map legacy database tables, code generation is the way to go.
- Second, we can improve the readability of our `HLists` by using *value classes* to replace more vanilla column types like `String` and `Int`. This can increase verbosity but significantly reduces errors. We'll see this in the section on value classes, later in this chapter.

> **Tip**
>
> **Code Generation**
>
> Sometimes your code is the definitive description of the schema; other times it's the database itself. The latter is the case when working with legacy databases, or database where the schema is managed independently of your Slick application.
>
> When the database is considered the source truth in your organisation, the Slick code generator is an important tool. It allows you to connect to a database, generate the table definitions, and customize the code produced.
>
> Prefer it to manually reverse engineering a schema by hand.

5.2.4 Exercises

5.2.4.1 Turning a Row into Many Case Classes

Our `HList` example mapped a table with many columns. It's not the only way to deal with lots of columns.

Use custom functions with <> and map `UserTable` into a tree of case classes. To do this you will need to define the schema, define a `User`, insert data, and query the data.

To make this easier, we're just going to map six of the columns. Here are the case classes to use:

```
case class EmailContact(name: String, email: String)
case class Address(street: String, city: String, country: String)
case class User(contact: EmailContact, address: Address, id: Long = 0L)
```

You'll find a definition of `UserTable` that you can copy and paste in the example code in the file *chapter-05/src/main/scala/nested_case_class.scala*.

See the solution

5.3 Table and Column Representation

Now we know how rows can be represented and mapped, let's look in more detail at the representation of the table and the columns it comprises. In particular we'll explore nullable columns, foreign keys, more about primary keys, composite keys, and options you can apply a table.

5.3.1 Nullable Columns

Columns defined in SQL are nullable by default. That is, they can contain `NULL` as a value. Slick makes columns non-nullable by default—if you want a nullable column you model it naturally in Scala as an `Option[T]`.

Let's create a variant of `User` with an optional email address:

```
case class User(name: String, email: Option[String] = None, id: Long = 0L)

class UserTable(tag: Tag) extends Table[User](tag, "user") {
  def id    = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def name  = column[String]("name")
  def email = column[Option[String]]("email")

  def * = (name, email, id) <> (User.tupled, User.unapply)
}

lazy val users = TableQuery[UserTable]
lazy val insertUser = users returning users.map(_.id)
```

We can insert users with or without an email address:

```
users += User("Dave", Some("dave@example.org"))
users += User("HAL")
```

and retrieve them again with a select query:

```
val myUsers = exec(users.result)
// myUsers: Seq[User] = List(
//   User(Dave,Some(dave@example.org),1),
//   User(HAL,None,2))
```

So far, so ordinary. What might be a surprise is how you go about selecting all rows that have no email address:

```
// Don't do this
val none: Option[String] = None
val myUsers = exec(users.filter(_.email === none).result)
// myUsers: Seq[User] = Nil
```

Interestingly, despite the fact that we have one row in the database no email address, this query produces no results.

Veterans of database administration will be familiar with this interesting quirk of SQL: expressions involving `null` themselves evaluate to `null`. For example, the SQL expression `'Dave' = 'HAL'` evaluates to `false`, whereas the expression `'Dave' = null` evaluates to `null`.

Our Slick query above amounts to:

```
SELECT * FROM "user" WHERE "email" = NULL
```

The SQL expression `"email" = null` evaluates to `null` for any value of `"email"`. SQL's `null` is a falsey value, so this query never returns a value.

To resolve this issue, SQL provides two operators: `IS NULL` and `IS NOT NULL`, which are provided in Slick by the methods `isEmpty` and `isDefined` defined on any `Rep[Option[A]]`:

Table 5.1: Optional column methods. Operand and result types should be interpreted as parameters to `Rep[_]`. The `?` method is described in the next section.

Scala Code	Operand Column Types	Result Type	SQL Equivalent
`col.?`	`A`	`Option[A]`	`col`
`col.isEmpty`	`Option[A]`	`Boolean`	`col is null`
`col.isDefined`	`Option[A]`	`Boolean`	`col is not null`

We can fix our query by replacing our equality check with `isEmpty`:

```
val myUsers = exec(users.filter(_.email.isEmpty).result)
// myUsers: Seq[User] = List(User(HAL,None,2))
```

which translates to the following SQL:

```
SELECT * FROM "user" WHERE "email" IS NULL
```

5.3.2 Primary Keys

We had our first introduction to primary keys in Chapter 1, where we started setting up `id` fields using the `O.PrimaryKey` and `O.AutoInc` column options:

```
def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
```

These options do two things:

- they modify the SQL generated for DDL statements;
- `O.AutoEnc` removes the corresponding column from the SQL generated for `INSERT` statements, allowing the database to insert an auto-incrementing value.

In Chapter 1 we combined `O.AutoInc` with a case class that has a default ID of `0L`, knowing that Slick will skip the value in insert statements:

```
case class User(name: String, id: Long = 0L)
```

While the authors like the simplicity of this style, some developers prefer to wrap primary key values in `Options`:

```
case class User(name: String, id: Option[Long] = None)
```

In this model we use `None` as the primary key of an unsaved record and `Some` as the primary key of a saved record. This approach has advantages and disadvantages:

- on the positive side it's easier to identify unsaved records;
- on the negative side it's harder to get the value of a primary key for use in a query.

Let's look at the changes we need to make to our `UserTable` to make this work:

```
case class User(id: Option[Long], name: String)

class UserTable(tag: Tag) extends Table[User](tag, "user") {
  def id    = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def name  = column[String]("name")

  def * = (id.?, name, email) <> (User.tupled, User.unapply)
}
```

The key thing to notice here is that we *don't* want the primary key to be optional in the database. We're using `None` to represent an *unsaved* value—the database assigns a primary key for us on insert, so we can never retrieve a `None` via a database query.

We need to map our non-nullable column to an optional value. This is handled by the `?` method in the default projection, which converts a `Rep[A]` to a `Rep[Option[A]]`.

5.3.3 Compound Primary Keys

There is a second way to declare a column as a primary key:

```
def id = column[Long]("id", O.AutoInc)
def pk = primaryKey("pk_id", id)
```

This separate step doesn't make much of a difference in this case. It separates the column definition from the key constraint, meaning the schema will include:

```
ALTER TABLE "user" ADD CONSTRAINT "pk_id" PRIMARY KEY("id")
```

> **Tip**
>
> **H2 Issue**
>
> As it happens, this specific example doesn't currently work with H2 and Slick.
>
> The `O.AutoInc` marks the column as an H2 "IDENTIY" column which is, implicitly, a primary key as far as H2 is concerned.

The `primaryKey` method is more useful for defining *compound* primary keys that involve two or more columns.

Let's look at this by adding the ability for people to chat in rooms. First we need a table for storing rooms, which is straightforward:

```
// Regular table definition for a chat room:
case class Room(title: String, id: Long = 0L)

class RoomTable(tag: Tag) extends Table[Room](tag, "room") {
 def id    = column[Long]("id", O.PrimaryKey, O.AutoInc)
 def title = column[String]("title")
 def * = (title, id) <> (Room.tupled, Room.unapply)
}

lazy val rooms = TableQuery[RoomTable]
lazy val insertRoom = rooms returning rooms.map(_.id)
```

Next we need a table that relates users to rooms. We'll call this the *occupant* table. Rather than give this table an auto-generated primary key, we'll make it a compound of the user and room IDs:

```
case class Occupant(roomId: Long, userId: Long)

class OccupantTable(tag: Tag) extends Table[Occupant](tag, "occupant") {
  def roomId = column[Long]("room")
  def userId = column[Long]("user")

  def pk = primaryKey("room_user_pk", (roomId, userId))

  def * = (roomId, userId) <> (Occupant.tupled, Occupant.unapply)
}

lazy val occupants = TableQuery[OccupantTable]
```

We can define composite primary keys using tuples or `HLists` of columns (Slick generates a `ProvenShape` and inspects it to find the list of columns involved). The SQL generated for the `occupant` table is:

```
CREATE TABLE "occupant" (
  "room" BIGINT NOT NULL,
  "user" BIGINT NOT NULL
)

ALTER TABLE "occupant"
ADD CONSTRAINT "room_user_pk" PRIMARY KEY("room", "user")
```

Using the `occupant` table is no different from any other table:

```
val daveId: Long = insertUser += User(None, "Dave", Some("dave@example.org"))
val airLockId: Long = insertRoom += Room("Air Lock")

// Put Dave in the Room:
occupants += Occupant(airLockId, daveId)
```

Of course, if we try to put Dave in the Air Lock twice, the database will complain about duplicate primary keys.

5.3.4 Indices

We can use indices to increase the efficiency of database queries at the cost of higher disk usage. Creating and using indices is the highest form of database sorcery, different for every database application, and well beyond the scope of this book. However, the syntax for defining an index in Slick is simple:

```
def nameIndex = index("name_idx", name, unique=true)
```

The corresponding DDL statement produced from a called to `schema` will be:

```
CREATE UNIQUE INDEX "name_idx" ON "user" ("name")
```

We can create compound indices on multiple columns just like we can with primary keys:

```
def nameIndex = index("sample_idx", (column1, column2), unique=true)
```

In this case the corresponding DDL statement will be:

```
CREATE UNIQUE INDEX "sample_idx" ON "mytable" ("column1", "column2")
```

5.3.5 Foreign Keys

Foreign keys are declared in a similar manner to compound primary keys.

The method `foreignKey` takes four required parameters:

- a name;
- the column, or columns, that make up the foreign key;
- the `TableQuery` that the foreign key belongs to; and
- a function on the supplied `TableQuery[T]` taking the supplied column(s) as parameters and returning an instance of T.

We'll step through this by using foreign keys to connect a `message` to a `user`. We do this by changing the definition of `message` to reference the `id` of its sender instead of their name:

```
case class Message(
  senderId: Long,
  content:  String,
  id:       Long = 0L)

class MessageTable(tag: Tag) extends Table[Message](tag, "message") {
  def id       = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def senderId = column[Long]("sender")
  def content  = column[String]("content")

  def * = (senderId, content, id) <> (Message.tupled, Message.unapply)

  def sender = foreignKey("sender_fk", senderId, users)(_.id)
}
```

The column for the sender is now a `Long` instead of a `String`. We have also defined a method, `sender`, providing the foreign key linking the `senderId` to a `user` `id`.

The `foreignKey` gives us two things. First, it adds a constraint to the DDL statement generated by Slick:

```
ALTER TABLE "message" ADD CONSTRAINT "sender_fk"
  FOREIGN KEY("sender") REFERENCES "user"("id")
  ON UPDATE NO ACTION
  ON DELETE NO ACTION
```

Tip

On Update and On Delete

A foreign key makes certain guarantees about the data. In the case we've looked at there must be a `sender` in the `user` table to successfully insert a new `message`.

So what happens if something changes with the `user` row? There are a number of *referential actions* that could be triggered. The default is for nothing to happen, but you can change that.

Let's look at an example. Suppose we delete a user, and we want all the messages associated with that user to be removed. We could do that in our application, but it's something the database can provide for us:

```
def sender =
  foreignKey("sender_fk", senderId, users) ↩
  (_.id, onDelete=ForeignKeyAction.Cascade)
```

Providing Slick's `schema` command has been run for the table, or the SQL `ON DELETE CASCADE` action has been manually applied to the database, the following action will remove HAL from the `users` table, and all of the messages that HAL sent:

```
users.filter(_.name === "HAL").delete
```

Slick supports `onUpdate` and `onDelete` for the five actions:

Action	Description
`NoAction`	The default.

`Cascade`	A change in the referenced table triggers a change in the referencing table. In our example, deleting a user will cause their messages to be deleted.
`Restrict`	Changes are restricted, triggered a constraint violation exception. In our example, you would not be allowed to delete a user who had posted a message.
`SetNull`	The column referencing the updated value will be set to NULL.
`SetDefault`	The default value for the referencing column will be used. Default values are discussion in Table and Column Modifiers, later in this chapter.

Second, the foreign key gives us a query that we can use in a join. We've dedicated the next chapter to looking at joins in detail, but here's a simple join to illustrate the use case:

```
val q = for {
  msg <- messages
  usr <- msg.sender
} yield (usr.name, msg.content)
```

This is equivalent to the query:

```
SELECT u."name", m."content"
FROM "message" m, "user" u
WHERE "id" = m."sender"
```

and produces the following results:

```
Vector(
 (Dave,Hello, HAL. Do you read me, HAL?),
 (HAL,Affirmative, Dave. I read you.),
 (Dave,Open the pod bay doors, HAL.),
 (HAL,I'm sorry, Dave. I'm afraid I can't do that.))
```

Tip

Save Your Sanity With Laziness

Defining foreign keys places constraints on the order in which we have to define our database tables. In the example above, the foreign key from `MessageTable` to `UserTable` requires us to place the latter definition above the former in our Scala code.

Ordering constraints make complex schemas difficult to write. Fortunately, we can work around them using `defs` and `lazy vals`. In the example below, the `sender` foreign key is defined above the `users` table that it references. However, because `sender` is a `def` and `users` is a `lazy val`, the code runs fine without any of the `NullPointerExceptions` we would otherwise receive at startup.

```
class MessageTable(tag: Tag) extends Table[Message](tag, "message") {
  def id       = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def senderId = column[Long]("sender")
  def content  = column[String]("content")

  def * = (senderId, content, id) <> (Message.tupled, Message.unapply)

  def sender = foreignKey(
    "sender_fk",
    senderId,
    users
  )(_.id, onDelete=ForeignKeyAction.Cascade)
}

lazy val users      = TableQuery[UserTable]
lazy val messages   = TableQuery[MessageTable]
lazy val insertUser = users returning users.map(_.id)
```

5.3.6 Column Options

We'll round off this section by looking at modifiers for columns and tables. These allow us to tweak the default values, sizes, and data types for columns at the SQL level.

We have already seen two examples of column options, namely `O.PrimaryKey` and `O.AutoInc`. Column options are defined in `ColumnOption`, and as you have seen are accessed via `O`.

The following example introduces three new options: `O.Length`, `O.DBType`, and `O.Default`.

```
case class User(
  name:   String,
  avatar: Option[Array[Byte]] = None,
  id:     Long = 0L)

class UserTable(tag: Tag) extends Table[User](tag, "user") {
  def id     = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def name   = column[String]("name",
                 O.Length(64, true), O.Default("Anonymous Coward"))
  def avatar = column[Option[Array[Byte]]]("avatar", O.DBType("BINARY(2048)"))

  def * = (name, avatar, id) <> (User.tupled, User.unapply)
}
```

In this example we've done three things:

1. We've used `O.Length` to give the `name` column a maximum length. This modifies the type of the column in the DDL statement. The parameters to `O.Length` are an `Int` specifying the maximum length, and a `Boolean` indicating whether the length is variable. Setting the `Boolean` to `true` sets the SQL column type to `VARCHAR`; setting it to `false` sets the type to `CHAR`.

2. We've used `O.Default` to give the `name` column a default value. This adds a `DEFAULT` clause to the column definition in the DDL statement.

3. We've used `O.DBType` to control the exact type used by the database. The values allowed here depend on the database we're using.

5.3.7 Exercises

5.3.7.1 Filtering Optional Columns

Sometimes you want to look at all the users in the database, and sometimes you want to only see rows matching a particular value.

Working with the optional email address for a user, write a method that will take an optional value, and list rows matching that value.

The method signature is:

```
def filterByEmail(email: Option[String]) = ???
```

Assume we only have two user records: one with an email address of "dave@example.org", and one with no email address.

We want `filterByEmail(Some("dave@example.org"))` to produce one row, and `filterByEmail(None)` to produce two rows.

Tip: it's OK to use multiple queries.

See the solution

5.3.7.2 Inside the Option

Build on the last exercise to match rows that start with the supplied optional value. Recall that `Rep[String]` defines `startsWith`.

So this time even `filterByEmail(Some("dave@")).run` will produce one row.

See the solution

5.3.7.3 Matching or Undecided

Not everyone has an email address, so perhaps when filtering it would be safer to only exclude rows that don't match our filter criteria. That is, keep `NULL` addresses in the results.

Add Elena to the database...

```
insert += User("Elena", Some("elena@example.org"))
```

...and modify `filterByEmail` so when we search for `Some("elena@example.org")` we only exclude Dave, as he definitely doesn't match that address.

This time you can do this in one query.

See the solution

5.3.7.4 Enforcement

What happens if you try adding a message for a user ID of `3000`?

For example:

```
messages += Message(UserPK(3000L), "Hello HAL!", DateTime.now)
```

Note that there is no user in our example with an ID of 3000. If you are using an editor rather than the REPL the file to open is `chapter-05/src/main/scala/value_classes.scala`.

See the solution

5.3.7.5 Model This

We're now charging for our chat service. Outstanding payments will be stored in a table called `bill`. The default change is $12.00, and bills are recorded against a user. A user should only have one or zero entries in this table. Make sure it is impossible for a user to be deleted while they have a bill to pay.

Go ahead and model this.

Hint: Remember to include your new table when creating the schema:

```
(messages.schema ++ users.schema ++ bills.schema).create
```

Additionally, provide queries to give the full details of users:

- who do have an outstanding bill; and
- who have no outstanding bills.

Hint: Slick provides `in` for SQL's `WHERE x IN (SELECT ...)` expressions.

See the solution

5.4 Custom Column Mappings

We want to work with types that have meaning to our application. This means converting data from the simple types the database uses to something more developer-friendly.

We've already seen Slick's ability to map tuples and `HLists` of columns to case classes. However, so far the fields of our case classes have been restricted to simple types such as `Int` and `String`,

Slick also lets us control how individual columns are mapped to Scala types. For example, perhaps we'd like to use Joda Time's `DateTime` class for anything date and time related. Slick doesn't provide native support for Joda Time, but it's painless for us to implement it via Slick's `ColumnType` type class:

```
import java.sql.Timestamp
import org.joda.time.DateTime
import org.joda.time.DateTimeZone.UTC

object CustomColumnTypes {
  implicit val jodaDateTimeType =
    MappedColumnType.base[DateTime, Timestamp](
      dt => new Timestamp(dt.getMillis),
      ts => new DateTime(ts.getTime, UTC)
    )
}
```

What we're providing here is two functions:

- one that takes a `DateTime` and converts it to a database-friendly `java.sql.Timestamp`; and
- one that does the reverse, taking a `Timestamp` and converting it to a `DataTime`.

Once we have declared this custom column type, we are free to create columns containing `DateTimes`:

```
case class Message(
  senderId:  Long,
  content:   String,
  timestamp: DateTime,
  id:        Long = 0L)

class MessageTable(tag: Tag) extends Table[Message](tag, "message") {
  import CustomColumnTypes._

  def id        = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def senderId  = column[Long]("sender")
  def content   = column[String]("content")
  def timestamp = column[DateTime]("timestamp")

  def * = (senderId, content, timestamp, id) <>
    (Message.tupled, Message.unapply)
}

lazy val messages      = TableQuery[MessageTable]
lazy val insertMessage = messages returning messages.map(_.id)
```

Our modified definition of `MessageTable` allows us to work directly with `Messages` containing `DateTime` timestamps, without having to do cumbersome type conversions by hand:

```
// Insert a Message containing a DateTime:
val messageId = exec(insertMessage += Message(
  daveId,
  "Open the pod bay doors, HAL.",
  DateTime.now))

// Query Messages containing DateTimes:
val message = exec(messages.find(_.id === messageId).result.head)
// message: Message = Message(
//   1L,
//   "Open the pod bay doors, HAL.",
//   1968-05-10T08:59:00.000Z,
//   2001L)
```

This model of working with semantic types is immediately appealing to Scala developers. We strongly encourage you to use `ColumnTypes` in your applications, to help reduce bugs and let Slick take care of cumbersome type conversions.

5.4.1 Value Classes

We are currently using `Longs` to model primary keys. Although this is a good choice at a database level, it's not great for our application code. The problem is we can make silly mistakes:

```
// This code will fail:
for {
  message <- messages.head
  rubbish <- users.filter(_.senderId === message.id)
} yield rubbish
```

Do you see the problem here? We've incorrectly used the `id` field of the `message` to search for its sender, instead of the `senderId` field as would be correct.

This code compiles, runs, and may even find a user if there happens to be one with the same ID as our message. However, it is clear that the code is incorrect.

We can prevent these kinds of problems using types.The essential approach is to model primary keys using value classes:

```
case class MessagePK(value: Long) extends AnyVal
case class UserPK(value: Long) extends AnyVal
```

A value class is a compile-time wrapper around a value. At run time, the wrapper goes away, leaving no allocation or performance overhead[2] in our running code.

To use a value class we need to provide Slick with `ColumnTypes` to use these types with our tables. This is the same process we used for Joda Time `DateTimes`:

```
implicit val messagePKColumnType =
  MappedColumnType.base[MessagePK, Long](_.value, MessagePK(_))

implicit val userPKColumnType =
   MappedColumnType.base[UserPK, Long](_.value, UserPK(_))
```

Defining all these type class instances can be time consuming, especially if we're defining one for every table in our schema. Fortunately, Slick provides a short-hand called `MappedTo` to take care of this for us:

```
case class MessagePK(value: Long) extends AnyVal with MappedTo[Long]
case class UserPK(value: Long) extends AnyVal with MappedTo[Long]
```

When we use `MappedTo` we don't need to define a separate `ColumnType`. `MappedTo` works with any class that:

- has a method called `value` that returns the underlying database value;
- has a single-parameter constructor to create the Scala value from the database value.

Value classes are a great fit for the `MappedTo` pattern.

Let's redefine our tables to use our custom primary key types:

```
case class User(name: String, id: UserPK = UserPK(0L))

class UserTable(tag: Tag) extends Table[User](tag, "user") {
  def id   = column[UserPK]("id", O.PrimaryKey, O.AutoInc)
```

[2] It's not totally cost free: there are situations where a value will need allocation, such as when passed to a polymorphic method.

```
  def name = column[String]("name")
  def * = (name, id) <> (User.tupled, User.unapply)
}

case class Message(
  senderId:  UserPK,
  content:   String,
  id: MessagePK = MessagePK(0L))

class MessageTable(tag: Tag) extends Table[Message](tag, "message") {
  def id        = column[MessagePK]("id", O.PrimaryKey, O.AutoInc)
  def senderId  = column[UserPK]("sender")
  def content   = column[String]("content")
  def * = (senderId, content, id) <>
    (Message.tupled, Message.unapply)
  def sender = foreignKey("sender_fk", senderId, users) ↩
    (_.id, onDelete=ForeignKeyAction.Cascade)
}
```

Notice how we're able to be explicit: the `User.id` and `Message.senderId` are `UserPK`s and the `Message.id` is a `MessagePK`. Now, if we try our buggy query again, the compiler catches the problem:

```
for {
  message <- messages.head
  rubbish <- users.filter(_.senderId === message.id)
} yield rubbish

// Cannot perform option-mapped operation
//      with type: (PKs.UserPK, PKs.MessagePK) => R
//  for base type: (PKs.UserPK, PKs.UserPK) => Boolean
// [error] rubbish <- users.filter(_.senderId === message.id)
```

Values classes are a low-cost way to make code safer and more legible. The amount of code required is small, however for a large database it can still be an overhead. We can either use code generation to overcome this, or generalise our primary key type by making it generic:

```
final case class PK[A](value: Long) extends AnyVal with MappedTo[Long]

case class User(
  name:  String,
  id:    PK[UserTable])

class UserTable(tag: Tag) extends Table[User](tag, "user") {
  def id    = column[PK[UserTable]]("id", O.AutoInc, O.PrimaryKey)
  def name  = column[String]("name")

  def * = (name, id) <> (User.tupled, User.unapply)
}

lazy val users = TableQuery[UserTable]
```

With this approach we achieve type safety without the boiler plate of many primary key type definitions. Depending on the nature of your application, this may be convenient for you.

The general point is that we can use the whole of the Scala type system to represent primary keys, foreign keys, rows, and columns from our database. This is enormously valuable and should not be overlooked.

5.4.2 Modelling Sum Types

We've used case classes extensively for modelling data. Using the language of *algebraic data types*, case classes are "product types" (created from conjunctions of their field types). The other common form of algebraic data type is known as a *sum type*, formed from a *disjunction* of other types. We'll look at modelling these now.

As an example let's add a flag to our `Message` class to model messages as important, offensive, or spam. The natural way to do this is establish a sealed trait and a set of case objects:

```
sealed trait Flag
case object Important extends Flag
case object Offensive extends Flag
case object Spam extends Flag

case class Message(
  senderId:  UserPK,
  content:   String,
  flag:      Option[Flag] = None,
  id:        MessagePK = MessagePK(0L))
```

There are a number of ways we could represent the flags in the database. For the sake of the argument, let's use characters: `!`, X, and `$`. We need a new custom `ColumnType` to manage the mapping:

```
implicit val flagType =
  MappedColumnType.base[Flag, Char](
    flag => flag match {
      case Important => '!'
      case Offensive => 'X'
      case Spam      => '$'
    },
    code => code match {
      case '!' => Important
      case 'X' => Offensive
      case '$' => Spam
    })
```

This is similar to the enumeration pattern from the last set of exercises. In this case, however, the compiler can ensure we've covered all the cases. If we add a new flag (`OffTopic` perhaps), the compiler will issue warnings until we add it to our `Flag => Char` function. We can turn these compiler warnings into errors by enabling the Scala compiler's `-Xfatal-warnings` option, preventing us shipping the application until we've covered all bases.

Using `Flag` is the same as any other custom type:

```
class MessageTable(tag: Tag) extends Table[Message](tag, "message") {
  def id       = column[MessagePK]("id", O.PrimaryKey, O.AutoInc)
  def senderId = column[UserPK]("sender")
  def content  = column[String]("content")
  def flag     = column[Option[Flag]]("flag")
```

```
  def * = (senderId, content, flag, id) <> ↩
    (Message.tupled, Message.unapply)

  def sender = foreignKey("sender_fk", senderId, users) ↩
    (_.id, onDelete=ForeignKeyAction.Cascade)
}

lazy val messages = TableQuery[MessageTable]
```

We can insert a message with a flag easily:

```
messages +=
  Message(halId, "Just kidding. LOL.", Some(Important))
```

We can also query for messages with a particular flag. However, we need to give the compiler a little help with the types:

```
messages.filter(_.flag === (Important : Flag))
```

The *type annotation* here is annoying. We can work around it easily in two ways:

First, we can define a "smart constructor" method for each flag that returns it pre-cast as a `Flag`:

```
object Flag {
  val important: Flag = Important
  val offensive: Flag = Offensive
  val spam: Flag = Spam
}

messages.filter(_.flag === Flag.important).result
```

Second, we can define some custom syntax to build our filter expressions:

```
implicit class MessageQueryOps(message: MessageTable) {
  def isImportant = message.filter === (Important : Flag)
  def isOffensive = message.filter === (Offensive : Flag)
  def isOffTopic  = message.filter === (OffTopic  : Flag)
}

messages.filter(_.isImportant).result
```

5.4.3 Exercises

5.4.3.1 Mapping Enumerations

We can use the same trick that we've seen for `DateTime` and value classes to map enumerations.

Here's a Scala Enumeration for a user's role:

```
object UserRole extends Enumeration {
  type UserRole = Value
  val Owner   = Value("O")
  val Regular = Value("R")
}
```

Modify the `user` table to include a `UserRole`. In the database store the role as a single character.

See the solution

5.4.3.2 Alternative Enumerations

Modify your solution to the previous exercise to store the value in the database as an integer.

Oh, and by the way, this is a legacy system. If we see an unrecognized user role value, just default it to a `UserRole.Regular`.

See the solution

5.4.3.3 Custom Boolean

Messages can be high priority or low priority. The database value for high priority messages will be: `y`, `Y`, `+`, or `high`. For low priority messages the value will be: `n`, `N`, `–`, `lo`, or `low`.

Go ahead and model this with a sum type.

See the solution

5.5 Take Home Points

In this Chapter we covered a lot of Slick's features for defining database schemas. We went into detail about defining tables and columns, mapping them to convenient Scala types, adding primary keys, foreign keys, and indices, and customising Slick's DDL SQL. We also discussed writing generic code that works with multiple database back-ends, and how to structure the database layer of your application using traits and self-types.

The most important points are:

- We can separate the specific profile for our database (H2, Postgres, etc...) from our schema using *dependency injection*. We assemble a database layer from a number of traits, leaving the profile as an abstract field that can be implemented ("injected") at runtime.
- We can represent rows in a variety of ways: tuples, `HLists`, and arbitrary classes and case classes via the <> method.
- We can represent individual values in columns using arbitrary Scala data types by providing `ColumnTypes` to manage the mappings. We've seen numerous examples supporting typed primary keys such as `UserPK`, sealed traits such as `Flag`, and third party classes such as `DateTime`.
- Nullable values are typically represented as `Options` in Scala. We can either define columns to store `Options` directly, or use the `?` method to map non-nullable columns to `Optional` ones.
- We can define simple primary keys using `O.PrimaryKey` and compound keys using the `primaryKey` method.

- We can define `foreignKeys`, which gives us a simple way of linking tables in a join. More on this next chapter.

Slick's philosophy is to keep models simple. We model rows as flat case classes, ignoring joins with other tables. While this may seem inflexible at first, it more than pays for itself in terms of simplicity and transparency. Database queries are explicit and type-safe, and return values of convenient types.

In the next chapter we will build on the foundations of primary and foreign keys and look at writing more complex queries involving joins and aggregate functions.

Chapter 6

Joins and Aggregates

Wrangling data with joins and aggregates can be painful. In this chapter we'll try to ease that pain by exploring:

- different styles of join (monadic and applicative);
- different ways to join (inner, outer and zip); and
- aggregate functions and grouping.

6.1 Two Kinds of Join

There are two styles of join in Slick. One, called *applicative*, is based on an explicit `join` method. It's a lot like the SQL `JOIN ... ON` syntax.

The second style of join, *monadic*, makes use of `flatMap` as a way to join tables.

These two styles of join are not mutually exclusive. We can mix and match them in our queries. It's often convenient to create an applicative join and use it in a monadic join.

6.2 Monadic Joins

We have seen an example of monadic joins in the previous chapter:

```
val q = for {
  msg <- messages
  usr <- msg.sender
} yield (usr.name, msg.content)
```

Notice how we are using `msg.sender` which is defined as a foreign key:

```
class MessageTable(tag: Tag) extends Table[Message](tag, "message") {
  def id       = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def senderId = column[Long]("sender")
  def content  = column[String]("content")

  def * = (senderId, content, id) <> (Message.tupled, Message.unapply)

  def sender = foreignKey("sender_fk", senderId, users)(_.id)
}
```

We can express the same query without using a for comprehension:

```
val q =
  messages flatMap { msg =>
    msg.sender.map { usr =>
      (usr.name, msg.content)
    }
  }
```

Either way, when we run the query Slick generates something like the following SQL:

```
select
  u."name", m."content"
from
  "message" m, "user" u
where
  u."id" = m."sender"
```

That's the monadic style of query, using foreign key relationships.

> **Tip**
>
> **Run the Code**
>
> You'll find the example queries for this section in the file `joins.sql` over at the associated GitHub repository.
>
> From the `chapter-06` folder start SBT and at the SBT > prompt run:
>
> ```
> runMain JoinsExample
> ```

Even if we don't have a foreign key, we can use the same style and control the join ourselves:

```
val q = for {
  msg <- messages
  usr <- users if usr.id === msg.senderId
} yield (usr.name, msg.content)
```

Note how this time we're using `msg.senderId`, not the foreign key `sender`. This produces the same query when we joined using `sender`.

You'll see plenty of examples of this style of join. They look straight-forward to read, and are natural to write. The cost is that Slick has to translate the monadic expression down to something that SQL is capable of running.

6.3 Applicative Joins

An applicative join is where we explicitly write the join in code. In SQL this is via the `JOIN` and `ON` keywords, which are mirrored in Slick with the following methods:

- `join` — an inner join,
- `joinLeft` — a left outer join,

- `joinRight` — a right outer join,
- `joinFull` — a full outer join.

We will work through examples of each of these methods. But as a quick taste of the syntax, here's how we can join the `messages` table to the `users` on the `senderId`:

```
val q: Query[(MessageTable, UserTable), (Message, User), Seq] =
  messages join users on (_.senderId === _.id)
```

As you can see, this code produces be a query of `(MessageTable, UserTable)`. If we want to, we can be more explicit about the values used in the `on` part:

```
val q: Query[(MessageTable, UserTable), (Message, User), Seq] =
  messages join users on
    ( (m: MessageTable, u: UserTable) =>  m.senderId === u.id )
```

We can also write the join condition using pattern matching:

```
val q: Query[(MessageTable, UserTable), (Message, User), Seq] =
  messages join users on { case (m, u) =>  m.senderId === u.id }
```

Joins like this form queries that we convert to actions the usual way:

```
val action: DBIO[Seq[(Message, User)]] =
  q.result
```

In the rest of this section we'll work through a variety of more involved joins. You may find it useful to refer to figure 6.1, which sketches the schema we're using in this chapter.

6.3.1 Inner Join

An inner join selects data from multiple tables, where the rows in each table match up in some way. Typically the matching up is by comparing primary keys. If there are rows that don't match up, they won't appear in the join results.

We'll look at an example of an inner join in Slick with a chat example. Let's lookup messages that have a sender in the user table, and a room in the rooms table:

```
val usersAndRooms =
  messages.
  join(users).on(_.senderId === _.id).
  join(rooms).on{ case ((msg,user), room) => msg.roomId === room.id }

// usersAndRooms: slick.lifted.Query[
//  ((MessageTable, UserTable), RoomTable),
//  (
//    (MessageTable#TableElementType, UserTable#TableElementType),
//     RoomTable#TableElementType
    ),
//  Seq
// ] = Rep(Join Inner)
```

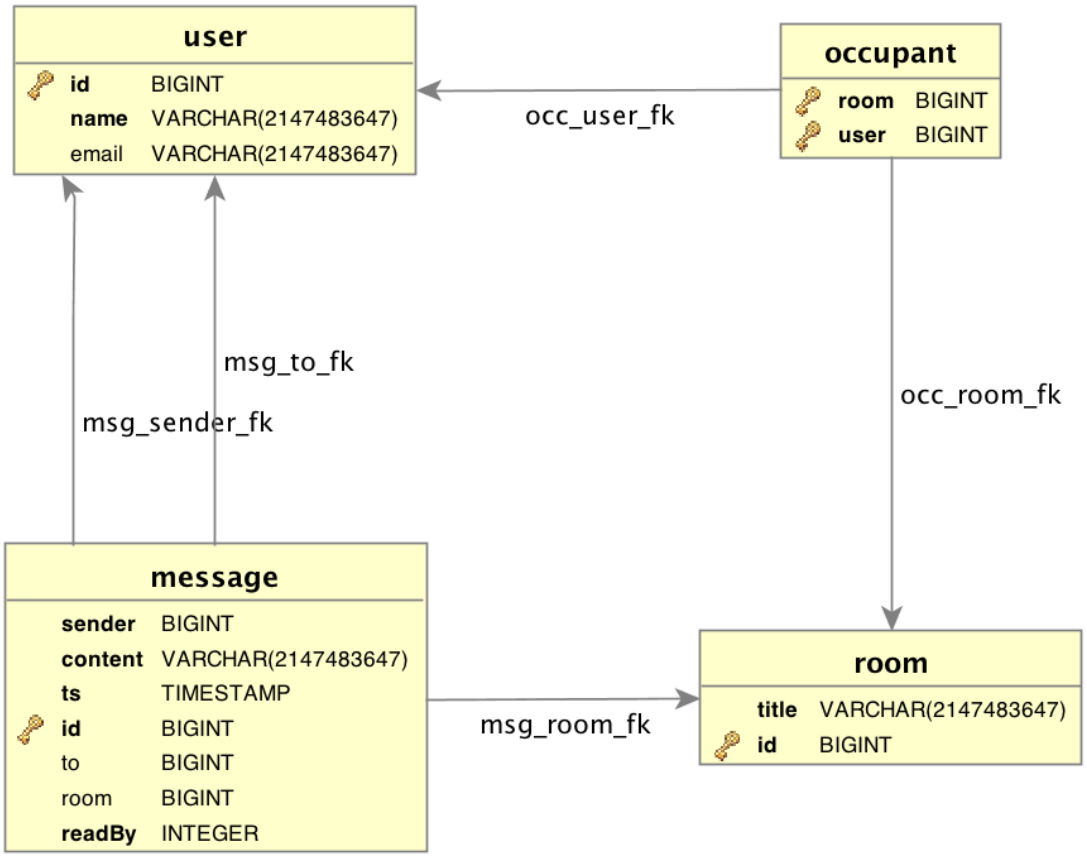

Figure 6.1: The database schema for this chapter. Find this code in the *chat-schema.scala* file of the example project on GitHub. A *message* can have a *sender*, which is a join to the *user* table. Also, a *message* can be in a *room*, which is a join to the *room* table. Finally, a *user* can be in a *room*, which is a join between *user* and *room* via the *occupant* table.

We're joining `messages` to `users`, and `messages` to `rooms`. We need two `joins`—if you are joining *n* tables you'll need *n-1* join expressions.

Notice that we're supplying a binary function to first call to `on` and a pattern matching function on our second call. Because each join results in a query of a tuple, successive joins result in nested tuples. Pattern matching is our preferred syntax for unpacking these tuples because it explicitly clarifies the structure of the query. However, you may see this more concisely expressed as a binary function:

```
val usersAndRooms =
  messages.
  join(users).on(_.senderId  === _.id).
  join(rooms).on(_._1.roomId === _.id)
```

6.3.1.1 Mapping Joins

We can turn this query into an action as it stands:

```
val action: DBIO[Seq[((Message, User), Room)]] =
  usersAndRooms.result
```

...but our results will contain nested tuples. That's OK, if that's what you want. But typically we want to `map` over the query to flatten the results and select the columns we want:

```
val usersAndRooms =
  messages.
  join(users).on(_.senderId  === _.id).
  join(rooms).on { case ((msg,user), room) => msg.roomId === room.id }.
  map { case ((msg, user), room) => (msg.content, user.name, room.title) }

val action: DBIO[Seq[(String, String, String)]] =
  usersAndRooms.result

exec(action)
// res1: Seq[(String, String, String)] =
//   Vector(
//     (Hello, HAL. Do you read me, HAL?, Dave, Air Lock),
//     (Affirmative, Dave. I read you.,   HAL,  Air Lock)
//     ...)
```

6.3.1.2 Filter with Joins

As joins are just queries, we can transform them using the combinators we learned in previous chapters. We've already seen an example of the `map` combinator. Another example would be the `filter` method.

As an example, we can use our `usersAndRooms` query and modify it to focus on a particular room. Perhaps we want to use our join for the Air Lock room:

```
// The query we've already seen...
val usersAndRooms =
  messages.
  join(users).on(_.senderId === _.id).
```

```
  join(rooms).on { case ((msg,user), room) => msg.roomId === room.id }

// ...modified to focus on one room:
val airLockMsgs =
  usersAndRooms.
  filter { case (_, room) => room.title === "Air Lock" }
```

As with other queries, the filter become a `WHERE` clause in SQL. Something like this:

```
SELECT
  "message"."content", "user"."name", "room"."title"
FROM
  "message"
  INNER JOIN "user" ON "message"."sender" = "user"."id"
  INNER JOIN "room" ON "message"."room"   = "room"."id"
WHERE
  "room"."title" = 'Air Lock';
```

6.3.2 Left Join

A left join (a.k.a. left outer join), adds an extra twist. Now we are selecting *all* the records from a table, and matching records from another table *if they exist*. If we find no matching record on the left, we will end up with `NULL` values in our results.

For an example from our chat schema, observe that messages can optionally be sent privately to another user via the `toId` column:

```
// Abbreviated table:
class MessageTable(tag: Tag) extends Table[Message](tag, "message") {
  def id       = column[Id[MessageTable]]("id", O.PrimaryKey, O.AutoInc)
  def senderId = column[Id[UserTable]]("sender")
  def content  = column[String]("content")
  def toId     = column[Option[Id[UserTable]]]("to")
  // ... etc
}
```

Let's suppose we want a list of all the messages and who they were sent to. Visually the left outer join is as shown below:

That is, we are going to select all the data from the messages table, plus data from the user table for those users that have been sent messages.

The join would be:

```
val left = messages.joinLeft(users).on(_.toId === _.id)
```

This query, `left`, is going to fetch messages and look up their corresponding recipients from the user table. Some messages may have been sent to the whole room rather than a specific user, in which case the `toId` column and the corresponding user fields will be `NULL`.

Slick will lift that possibly null value into something more comfortable: an `Option`. The full type of `left` is:

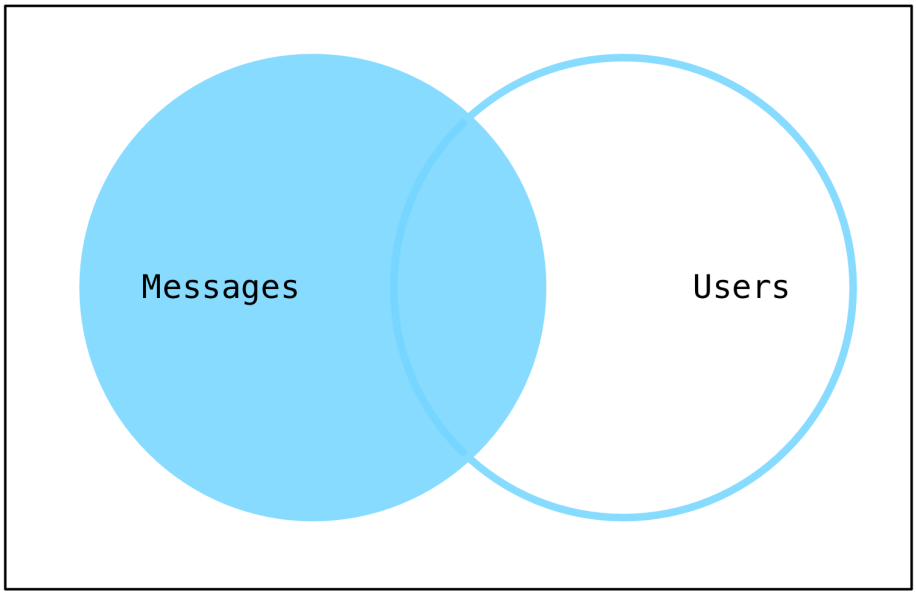

Figure 6.2: A visualization of the left outer join example. Selecting messages and associated recipients (users). For similar diagrams, see A Visual Explanation of SQL Joins, *Coding Horror*, 11 Oct 2007.

```
Query[
  (MessageTable, Rep[Option[UserTable]]),
  (MessageTable#TableElementType, Option[User]),
  Seq]
```

The results of this query are of type `(Message, Option[User])`—Slick has made the `User` side optional for us automatically.

If we want to just pick out the message content and the recipient name, we can `map` over the query:

```
val left =
  messages.
  .joinLeft(users).on(_.toId === _.id)
  .map { case (msg, user) => (msg.content, user.map(_.name)) }
```

Because the `user` element is optional, we naturally extract the `name` element using `Option.map`: `user.map(_.name)`.

The type of this query then becomes:

```
Query[
  (Rep[String], Rep[Option[String]]),
  (String, Option[String]),
  Seq]
```

The types `String` and `Option[String]` correspond to the sender name and the recipient name.

The sample data in *joins.sql* in the *chapter06* folder contains just two private messages (between Frank and Dave). The rest are public. So our query results are:

```
exec(left.result).foreach(println)

// (Hello, HAL. Do you read me, HAL?,             None)
// (Affirmative, Dave. I read you.,               None)
// (Open the pod bay doors, HAL.,                 None)
// (I'm sorry, Dave. I'm afraid I can't do that., None)
// (Well, whaddya think?,                         None)
// (I'm not sure, what do you think?,             None)
// (Are you thinking what I'm thinking?,          Some(Dave))
// (Maybe,                                        Some(Frank))
```

6.3.3 Right Join

In the previous section, we saw that a left join selects all the records from the left hand side of the join, with possibly NULL values from the right.

Right joins (or right outer joins) reverse the situation, selecting all records from the right side of the join, with possibly NULL values from the left.

We can demonstrate this by reversing our left join example. We'll ask for all users together with any private messages have they received. We'll use for comprehension syntax this time for variety:

```
val right = for {
  (msg, user) <- messages joinRight (users) on (_.toId === _.id)
} yield (user.name, msg.map(_.content))
```

From the results this time we can see that just Dave and Frank have seen private messages:

```
exec(right.result).foreach(println)
// (Dave,  Some(Are you thinking what I'm thinking?))
// (HAL,   None)
// (Elena, None)
// (Frank, Some(Maybe))
```

6.3.4 Full Outer Join

Full outer joins mean either side can be NULL.

From our schema an example would be the title of all rooms and messages in those rooms. Either side could be NULL because messages don't have to be in rooms, and rooms don't have to have any messages.

```
val outer = for {
  (room, msg) <- rooms joinFull messages on (_.id === _.roomId)
} yield (room.map(_.title), msg.map(_.content))
```

The type of this query has options on either side:

```
Query[
  (Rep[Option[String]], Rep[Option[String]]),
  (Option[String], Option[String]),
  Seq]
```

We can see this by running the query against the `chapter-06` example data in `chat_schema.scala`:

```
exec(outer.result).foreach(println)

// (Some(Air Lock),Some(Hello, HAL. Do you read me, HAL?))
// (Some(Air Lock),Some(Affirmative, Dave. I read you.))
// (Some(Air Lock),Some(Open the pod bay doors, HAL.))
// (Some(Air Lock),Some(I'm sorry, Dave. I'm afraid I can't do that.))
// (Some(Pod),Some(Well, whaddya think?))
// (Some(Pod),Some(I'm not sure, what do you think?))
// (Some(Pod),Some(Are you thinking what I'm thinking?))
// (Some(Pod),Some(Maybe))
// (Some(Crew Quarters),None)
// (None,Some(I am a HAL 9000 computer.))
// (None,Some(I became operational at the H.A.L. plant in Urbana,
//   Illinois on the 12th of January 1992.))
```

As you can see from the results, some rooms have many messages, the Crew Quarters has no messages, and HAL isn't in a room when he gives his final monologue.

> **Tip**
>
> At the time of writing H2 does not support full outer joins. Whereas earlier versions of Slick would throw a runtime exception, Slick 3 compiles the query into to something that will run, emulating a full outer join.

6.3.5 Cross Joins

In the examples above, whenever we've used `join` we've also used an `on` to constrain the join. This is optional.

If we omit the `on` condition for any `join`, `joinLeft`, or `joinRight`, we end up with a *cross join*.

Cross joins include every row from the left table with every row from the right table. If we have 10 rows in the first table and 5 in the second, the cross join produces 50 rows.

An example:

```
val cross = messages joinLeft users
// cross: slick.lifted.BaseJoinQuery[
//   MessageTable, Rep[Option[UserTable]],
//   Message, Option[User],
//   Seq,
//   MessageTable,
//   UserTable] = Rep(Join LeftOption)
```

6.4 Zip Joins

Zip joins are equivalent to `zip` on a Scala collection. Recall that the `zip` in the collections library operates on two lists and returns a list of pairs:

```
val xs = List(1, 2, 3)

xs zip xs.drop(1)
// List[(Int, Int)] = List((1,2), (2,3))
```

Slick provides the equivalent `zip` method for queries, plus two variations. Let's say we want to pair up adjacent messages into what we'll call a "conversation":

```
// Select message content, ordered by the date the messages were sent
val msgs = messages.sortBy(_.ts.asc).map(_.content)

// Pair up adjacent messages:
val conversations = msgs zip msgs.drop(1)

exec(conversations.result).foreach(println)
```

This will turn into an inner join, producing output like:

```
(Hello, HAL. Do you read me, HAL?, Affirmative, Dave. I read you.),
(Affirmative, Dave. I read you.  , Open the pod bay doors, HAL.),
(Open the pod bay doors, HAL.    , I'm sorry, Dave. ↩
                                   I'm afraid I can't do that.)
```

A second variation, `zipWith`, lets us provide a mapping function along with the join. We can provide a function to upper-case the first part of a conversation, and lower-case the second part:

```
def combiner(c1: Rep[String], c2: Rep[String]) =
  (c1.toUpperCase, c2.toLowerCase)

val query = msgs.zipWith(msgs.drop(1), combiner)
```

The final variant is `zipWithIndex`, which is as per the Scala collections method of the same name. Let's number each message:

```
val query = messages.map(_.content).zipWithIndex

val action: DBIO[Seq[(String, Long)]] =
  query.result
```

For H2 the SQL `ROWNUM()` function is used to generate a number. The data from this query will start:

```
exec(action)
// (Hello, HAL. Do you read me, HAL?, 0),
// (Affirmative, Dave. I read you.,   1),
// (Open the pod bay doors, HAL.,     2),
// ...
```

Not all databases support zip joins. Check for the `relational.zip` capability in the `capabilities` field of your chosen database profile:

```
slick.driver.H2Driver.capabilities
  .map(_.toString)
  .contains("relational.zip")
// true -- H2 supports zip et al

slick.driver.SQLiteDriver.capabilities
  .map(_.toString)
  .contains("relational.zip")
// false -- SQLite does not support zip et al
```

6.5 Joins Summary

In this chapter we've seen examples of the two different styles of join: applicative and monadic. We've also mixed and matched these styles.

We've seen how to construct the arguments to `on` methods, either with a binary join condition or by deconstructing a tuple with pattern matching.

Each join step produces a tuple. Using pattern matching in `map` and `filter` allows us to clearly name each part of the tuple, especially when the tuple is deeply nested.

We've also explored inner and outer joins, zip joins, and cross joins. We saw that each type of join is a query, making it compatible with combinators such as `map` and `filter` from earlier chapters.

6.6 Seen Any Scary Queries?

If you've been following along and running the example joins, you may have noticed large and unusual queries being generated. Or you may not have. Since Slick 3.1, the SQL generated by Slick has improved greatly.

However, you may find the SQL generated a little strange or involved. If Slick generates verbose queries are they are going to be slow? Yes, sometimes they will be.

Here's the key concept: the SQL generated by Slick is fed to the database optimizer. That optimizer has far better knowledge about your database, indexes, query paths, than anything else. It will optimize the SQL from Slick into something that works well.

Unfortunately, some optimizers don't manage this very well. Postgres does a good job. MySQL is, at the time of writing, pretty bad at this. The trick here is to watch for slow queries using Slick's performance logging, and use your database's `EXPLAIN` command to examine and debug the query plan.

Optimisations can often be achieved by rewriting monadic joins in applicative style and judiciously adding indices to the columns involved in joins. However, a full discussion of query optimisation is out of the scope of this book. See your database's documentation for more information.

If all else fails, we can rewrite queries for ultimate control using Slick's *Plain SQL* feature. We will look at this in Chapter 7.

6.7 Aggregation

Aggregate functions are all about computing a single value from some set of rows. A simple example is `count`. This section looks at aggregation, and also at grouping rows, and computing values on those groups.

6.7.1 Functions

Slick provides a few aggregate functions, as listed in the table below.

Table 6.1: A Selection of Aggregate Functions

Method	SQL
`length`	`COUNT(1)`
`countDistinct`	`COUNT(DISTINCT column)`
`min`	`MIN(column)`

Method	SQL
`max`	`MAX(column)`
`sum`	`SUM(column)`
`avg`	`AVG(column)` — mean of the column values

Using them causes no great surprises, as shown in the following examples:

```
val numRows: DBIO[Int] = messages.length.result

val numDifferentSenders: DBIO[Int] =
  messages.map(_.senderId).countDistinct.result

val firstSent: DBIO[Option[DateTime]] =
  messages.map(_.ts).min.result
```

While `length` and `countDistinct` return an `Int`, the other functions return an `Option`. This is because there may be no rows returned by the query, meaning the is no minimum, maximum and so on.

6.7.2 Grouping

Aggregate functions are often used with column grouping. For example, how many messages has each user sent? That's a grouping (by user) of a aggregate (count).

6.7.2.1 `groupBy`

Slick provides `groupBy` which will group rows by some expression. Here's an example:

```
val msgPerUser =
  messages.groupBy(_.senderId).
  map { case (senderId, msgs) => senderId -> msgs.length }.
  result
```

A `groupBy` must be followed by a `map`. The input to the `map` will be the grouping key (`senderId`) and a query for the group.

In the sample code for this chapter we're using a primary key of...

```
case class PK[A](value: Long) extends AnyVal with MappedTo[Long]
```

...to keep our keys usefully typed. So the type of `msgPerUser` query is:

```
DBIO[Seq[(PK[UserTable], Int)]]
```

When we run the query, it'll work, but it will be in terms of a user's primary key:

```
exec(msgPerUser)
// res1: Seq[(ChatSchema.PK[schema.UserTable], Int)] =
//  Vector((PK(1),4), (PK(2),4), (PK(4),2))
```

6.7.2.2 Groups and Joins

It'd be nicer to see the user's name. We can do that using our join skills:

```
val msgsPerUser =
   messages.join(users).on(_.senderId === _.id).
   groupBy { case (msg, user)   => user.name }.
   map     { case (name, group) => name -> group.length }.
   result
```

The results would be:

```
Vector((Frank,2), (HAL,4), (Dave,4))
```

So what's happened here? What `groupBy` has given us is a way to place rows into groups according to some function we supply. In this example the function is to group rows based on the user's name. It doesn't have to be a `String`, it could be any type in the table.

When it comes to mapping, we now have the key to the group (the user's name in our case), and the corresponding group rows *as a query*.

Because we've joined messages and users, our group is a query of those two tables. In this example we don't care what the query is because we're just counting the number of rows. But sometimes we will need to know more about the query.

6.7.2.3 More Complicated Grouping

Let's look at a more involved example by collecting some statistics about our messages. We want to find, for each user, how many messages they sent, and the date of their first message. We want a result something like this:

```
Vector(
  (Frank, 2, Some(2001-02-16T20:55:00.000Z)),
  (HAL,   4, Some(2001-02-17T10:22:52.000Z)),
  (Dave,  4, Some(2001-02-16T20:55:04.000Z)))
```

We have all the aggregate functions we need to do this:

```
val stats =
   messages.join(users).on(_.senderId === _.id).
   groupBy { case (msg, user) => user.name }.
   map     {
    case (name, group) =>
      (name, group.length, group.map{ case (msg, user) => msg.ts}.min)
   }
```

We've now started to create a bit of a monster query. We can simplify this, but before doing so, it may help to clarify that this query is equivalent to the following SQL:

```
select
  user.name, count(1), min(message.ts)
from
  message inner join user on message.sender = user.id
group by
  user.name
```

Convince yourself the Slick and SQL queries are equivalent, by comparing:

- the `map` expression in the Slick query to the `SELECT` clause in the SQL;
- the `join` to the SQL `INNER JOIN`; and
- the `groupBy` to the SQL `GROUP` expression.

If you do that you'll see the Slick expression makes sense. But when seeing these kinds of queries in code it may help to simplify by introducing intermediate functions with meaningful names.

There are a few ways to go at simplifying this, but the lowest hanging fruit is that `min` expression inside the `map`. The issue here is that the `group` pattern is a `Query` of `(MessageTable, UserTable)` as that's our join. That leads to us having to split it further to access the message's timestamp field.

Let's pull that part out as a method:

```
import scala.language.higherKinds

def timestampOf[S[_]]
  (group: Query[(MessageTable,UserTable), (Message,User), S]) =
    group.map { case (msg, user) => msg.ts }
```

What we've done here is introduced a method to work on the group query, using the knowledge of the `Query` type introduced in The Query and TableQuery Types section of Chapter 2.

The query (`group`) is parameterized by the join, the unpacked values, and the container for the results. By container we mean something like `Seq[T]`. We don't really care what our results go into, but we do care we're working with messages and users.

With this little piece of domain specific language in place, the query becomes:

```
val nicerStats =
   messages.join(users).on(_.senderId === _.id).
   groupBy { case (msg, user)   => user.name }.
   map     { case (name, group) => (name, group.length, timestampOf(group).min) }
```

We think these small changes make code more maintainable and, quite frankly, less scary. It may be marginal in this case, but real world queries can become large. Your team mileage may vary, but if you see Slick queries that are hard to understand, try pulling the query apart into named methods.

> **Tip**
>
> **Group By True**
>
> There's a `groupBy { _ => true}` trick you can use where you want to select more than one aggregate from a query.
>
> As an example, have a go at translating this SQL into a Slick query:

```
select min(ts), max(ts) from message where content like '%read%'
```

It's pretty easy to get either `min` or `max`:

```
messages.filter(_.content like "%read%").map(_.ts).min
```

But you want both `min` and `max` in one query. This is where `groupBy { _ => true}` comes into play:

```
messages.
 filter(_.content like "%read%").
 groupBy(_ => true).
 map {
  case (_, msgs) => (msgs.map(_.ts).min, msgs.map(_.ts).max)
}
```

The effect here is to group all rows into the same group! This allows us to reuse the `msgs` query, and obtain the result we want.

6.7.2.4 Grouping by Multiple Columns

The result of `groupBy` doesn't need to be a single value: it can be a tuple. This gives us access to grouping by multiple columns.

We can look at the number of messages per user per room. Something like this:

```
Vector(
  (Air Lock, HAL,   2),
  (Air Lock, Dave,  2),
  (Pod,      Dave,  2),
  (Pod,      Frank, 2) )
```

That is, we need to group by room and then by user, and finally count the number of rows in each group:

```
val msgsPerRoomPerUser =
   rooms.
   join(messages).on(_.id === _.roomId).
   join(users).on{ case ((room,msg), user) => user.id === msg.senderId }.
   groupBy { case ((room,msg), user)   => (room.title, user.name) }.
   map     { case ((room,user), group) => (room, user, group.length) }.
   sortBy  { case (room, user, group)  => room }
```

Hopefully you're now in a position where you can unpick this:

- We join on messages, room and user to be able to display the room title and user name.
- The value passed into the `groupBy` will be determined by the join.
- The result of the `groupBy` is the columns for the grouping, which is a tuple of the room title and the user's name.
- We select (`map`) just the columns we want: room, user and the number of rows.
- For fun we've thrown in a `sortBy` to get the results in room order.

6.8 Take Home Points

Slick supports `join`, `joinLeft`, `joinRight`, `joinOuter` and a `zip` join. You can map and filter over these queries as you would other queries with Slick. Using pattern matching on the query tuples can be more readable than accessing tuples via `._1`, `._2` and so on.

Aggregation methods, such as `length` and `sum`, produce a value from a set of rows.

Rows can be grouped based on an expression supplied to `groupBy`. The result of a grouping expression is a group key and a query defining the group. Use `map`, `filter`, `sortBy` as you would with any query in Slick.

The SQL produced by Slick might not be the SQL you would write. Slick expects the database query engine to perform optimisation. If you find slow queries, take a look at *Plain SQL*, discussed in the next chapter.

6.9 Exercises

Because these exercises are all about multiple tables, take a moment to remind yourself of the schema. You'll find this in the example code, `chatper-06`, in the source file `chat_schema.scala`.

6.9.1 Name of the Sender

Each message is sent by someone. That is, the `messages.senderId` will have a matching row via `users.id`.

Please...

- Write a monadic join to return all `Message` rows and the associated `User` record for each of them.
- Change your answer to just return the content of a message and the name of the sender.
- Modify the query to return the results in name order.
- Re-write the query as an applicative join.

These exercises will get your fingers familiar with writing joins.

See the solution

6.9.2 Messages of the Sender

Write a method to fetch all the message sent by a particular user. The signature is:

```
def findByName(name: String): Query[Rep[Message], Message, Seq] = ???
```

See the solution

6.9.3 Having Many Messages

Modify the `msgsPerUser` query...

```
val msgsPerUser =
   messages.join(users).on(_.senderId === _.id).
   groupBy { case (msg, user)  => user.name }.
   map     { case (name, group) => name -> group.length }
```

...to return the counts for just those users with more than 2 messages.

See the solution

6.9.4 Collecting Results

A join on messages and senders will produce a row for every message. Each row will be a tuple of the user and message:

```
users.join(messages).on(_.id === _.senderId)
// res1: slick.lifted.Query[
//  (UserTable, MessageTable),
//  (UserTable#TableElementType, MessageTable#TableElementType),
//  Seq] = Rep(Join Inner)
```

Sometimes you'll really want something like a `Map[User, Seq[Message]]`.

There's no built-in way to do that in Slick, but you can do it in Scala using the collections `groupBy` method.

```
Seq(
  ("HAL"  -> "Hello"),
  ("Dave" -> "How are you?"),
  ("HAL"  -> "I have terrible pain in all the diodes")
  ).groupBy{ case (name, message) => name }
// res2: Map[String,Seq[(String, String)]] = Map(
//  HAL  -> List((HAL,Hello), (HAL,I have terrible pain in all the diodes)),
//  Dave -> List((Dave,How are you?))
// )
```

We can go further and reduce this to:

```
res2.mapValues { values =>
  values.map{ case (name, msg) => msg }
}
// res3: Map[String,Seq[String]] = Map(
//  HAL -> List(Hello, I have terrible pain in all the diodes),
//  Dave -> List(How are you?)
// )
```

Go ahead and write a method to encapsulate this:

```
def userMessages: DBIO[Map[User, Seq[Message]]] = ???
```

See the solution

Chapter 7

Plain SQL

Slick supports Plain SQL queries in addition to the lifted embedded style we've seen up to this point. Plain queries don't compose as nicely as lifted, or offer quite the same type safely. But they enable you to execute essentially arbitrary SQL when you need to. If you're unhappy with a particular query produced by Slick, dropping into Plain SQL is the way to go.

In this section we will see that:

- the interpolators `sql` (for select) and `sqlu` (for updates) are used to create Plain SQL queries;
- values can be safely substituted into queries using a `${expresson}` syntax;
- custom types can be used in Plain SQL, as long as there is a converter in scope; and
- the `tsql` interpolator can be used to check the syntax and types of a query via a database at compile time.

7.1 Selects

Let's start with a simple example of returning a list of room IDs.

```
val action = sql""" select "id" from "room" """.as[Long]

Await.result(db.run(action), 2 seconds)
// Vector(1, 2, 3)
```

Running a Plain SQL query looks similar to other queries we've seen in this book: just call `db.run` as usual.

The big difference is with the construction of the query. We supply both the SQL we want to run and specify the expected result type using `as[T]`.

The `as[T]` method is pretty flexible. Let's get back the room ID and room title:

```
val roomInfo = sql""" select "id", "title" from "room" """.as[(Long,String)]

// When executed will produce:
// Vector((1,Air Lock), (2,Pod), (3,Brain Room))
```

Notice we specified a tuple of `(Long, String)` as the result type. This matches the columns in our SQL SELECT statement.

Using `as[T]` we can build up arbitrary result types. Later we'll see how we can use our own application case classes too.

One of the most useful features of the SQL interpolators is being able to reference Scala values in a query:

```
val roomName = "Pod"
val podRoomAction = sql"""
  select
    "id", "title"
  from
    "room"
  where
    "title" = $roomName """.as[(Long,String)].headOption

// When run will produce:
// Some((2,Pod))
```

Notice how `$roomName` is used to reference a Scala value `roomName`. This value is incorporated safely into the query. That is, you don't have to worry about SQL injection attacks when you use the SQL interpolators in this way.

Advanced

The Danger of Strings

The SQL interpolators are essential for situations where you need full control over the SQL to be run. Be aware there there is some loss compile-time of safety. For example:

```
val t = 42
sql""" select "id" from "room" where "title" = $t """.as[Long].headOption
// JdbcSQLException: Data conversion error converting "Air Lock"
```

That example compiles without error, but fails at runtime as the type of the `title` column is a `String` and we've provided an `Int`. The equivalent query using the lifted embedded style would have caught the problem at compile time.

The `tsql` interpolator, described later in this chapter, helps here by connecting to a database at compile time to check the query and types.

Another place you can become unstuck is with the `#$` style of substitution. This is called *splicing*, and is used when you *don't* want SQL escaping to apply. For example, perhaps the name of the table you want to use may change:

```
val table = "message"
val action = sql""" select "id" from "#$table" """.as[Long]
```

In this situation we do not want the value of `table` to be treated as a `String`. If you did, you'd end up with the invalid query: `select "id" from "'message'"` (notice the double quotes and single quotes around the table name, which is not valid SQL).

However, this means you can produce dangerous SQL with splicing. The golden rule is to never use `#$` with input supplied by a user.

To be sure you remember it, say it again with us: never use `#$` with input supplied by a user.

7.1.1 Select with Custom Types

Out of the box Slick knows how to convert many data types to and from SQL data types. The examples we've seen so far include turning a Scala `String` into a SQL string, and a SQL BIGINT to a Scala `Long`.

These conversions are available to `as[T]`. If we want to work with a type that Slick doesn't know about, we need to provide a conversion. That's the role of the `GetResult` type class.

As an example, we can fetch the timestamp on our messages using JodaTime's `DateTime`:

```
sql""" select "ts" from "message" """.as[DateTime]
```

For this to compile we need to provide an instance of `GetResult[DateTime]`:

```
import slick.jdbc.GetResult

implicit val GetDateTime =
  GetResult[DateTime](r => new DateTime(r.nextTimestamp(), DateTimeZone.UTC))
```

`GetResult` is wrapping up a function from `r` (a `PositionedResult`) to `DateTime`. The `PositionedResult` provides access to the database value (via `nextTimestamp`, `nextLong`, `nextBigDecimal` and so on). We use the value from `nextTimestamp` to feed into the constructor for `DateTime`.

The name of this value doesn't matter. What's important is the type, `GetResult[DateTime]`, and that it is marked as implicit. This allows the compiler to lookup our conversion function when we mention a `DateTime`.

If we try to construct a query without a `GetResult[DateTime]` instance in scope, the compiler will complain:

```
could not find implicit value for parameter rconv:
  slick.jdbc.GetResult[DateTime]
```

7.1.2 Case Classes

As you've probably guessed, returning a case class from a Plain SQL query means providing a `GetResult` for the case class. Let's work through an example for the messages table.

> **Tip**
>
> **Run the Code**
>
> You'll find the example queries for this section in the file `select.sql` inside the `chapter-07` folder. This is all in the example code base on GitHub.

Recall that a message contains: an ID, some content, the sender ID, a timestamp, an optional room ID, and an optional recipient for private messages. We'll model this as we did in Chapter 5, by wrapping the `Long` primary keys in the type `Id[Table]`.

This gives us:

```
case class Message(
  senderId: Id[UserTable],
  content:  String,
  ts:       DateTime,
  roomId:   Option[Id[RoomTable]] = None,
  toId:     Option[Id[UserTable]] = None,
  id:       Id[MessageTable]      = Id(0L) )
```

To provide a `GetResult[Message]` we need all the types inside the `Message` to have `GetResult` instances. We've already tackled `DateTime`. That leaves `Id[MessageTable]`, `Id[UserTable]`, `Option[Id[UserTable]`, and `Option[Id[RoomTable]`.

Dealing with the two non-option IDs is straightforward:

```
implicit val GetUserId    = GetResult(r => Id[UserTable](r.nextLong))
implicit val GetMessageId = GetResult(r => Id[MessageTable](r.nextLong))
```

For the optional ones we need to use `nextLongOption` and then `map` to the right type:

```
implicit val GetOptUserId = GetResult(r =>
                                  r.nextLongOption.map(i => Id[UserTable](i)))
implicit val GetOptRoomId = GetResult(r =>
                                  r.nextLongOption.map(i => Id[RoomTable](i)))
```

With all the individual columns mapped we can pull them into a `GetResult` for `Message`. There are two helper methods which make it easier to construct these instances:

- `<<` for calling the appropriate *nextXXX* method; and
- `<<?` when the value is optional.

We can use them like this:

```
implicit val GetMessage = GetResult(r =>
   Message(senderId  = r.<<,
           content   = r.<<,
           ts        = r.<<,
           id        = r.<<,
           roomId    = r.<<?,
           toId      = r.<<?) )
```

This works because we've provided implicits for the components of the case class. As the types of the fields are known, `<<` and `<<?` simply expect the implicit `GetResult[T]` for each type.

Now we can select into `Message` values:

```
val action: DBIO[Seq[Message]] =
  sql""" select * from "message" """.as[Message]
```

In all likelihood you'll prefer the lifted embedded style over Plain SQL in this specific example. But if you do find yourself using Plain SQL, for performance reasons perhaps, it's useful to know how to convert database values up into meaningful domain types.

> **Advanced**
>
> **`SELECT *`**
>
> We sometimes use `SELECT *` in this chapter to fit our code examples onto the page. You should avoid this in your code base as it leads to brittle code.
>
> An example: if, outside of Slick, a table is modified to add a column, the results from the query will unexpectedly change. You code may not longer be able to map results.

7.2 Updates

Back in Chapter 3 we saw how to modify rows with the `update` method. We noted that batch updates were challenging when we wanted to use the row's current value. The example we used was appending an exclamation mark to a message's content:

```
UPDATE "message" SET "content" = CONCAT("content", '!')
```

Plain SQL updates will allow us to do this. The interpolator is `sqlu`:

```
val action =
  sqlu"""UPDATE "message" SET "content" = CONCAT("content", '!')"""
```

The `action` we have constructed, just like other actions, is not run until we evaluate it via `db.run`. But when it is run, it will append the exclamation mark to each row value, which is what we couldn't do as efficiently with the lifted embedded style.

Just like the `sql` interpolator, we also have access to `$` for binding to variables:

```
val char = "!"
val query =
  sqlu"""UPDATE "message" SET "content" = CONCAT("content", $char)"""
```

This gives us two benefits: the compiler will point out typos in variables names, but also the input is sanitized against SQL injection attacks.

7.2.1 Updating with Custom Types

Working with basic types like `String` and `Int` is fine, but sometimes you want to update using a richer type. We saw the `GetResult` type class for mapping select results, and for updates this is mirrored with the `SetParameter` type class.

What happens if you want to set a parameter of a type not automatically handled by Slick? You need to provide an instance of `SetParameter` for the type.

For example, JodaTime's `DateTime` is not known to Slick by default. We can teach Slick how to set `DataTime` parameters like this:

```
import slick.jdbc.SetParameter

implicit val SetDateTime = SetParameter[DateTime](
  (dt, pp) => pp.setTimestamp(new Timestamp(dt.getMillis))
 )
```

The value `pp` is a `PositionedParameters`. This is an implementation detail of Slick, wrapping a SQL statement and a placeholder for a value. Effectively we're saying how to treat a `DateTime` regardless of where it appears in the update statement.

In addition to a `Timestamp` (via `setTimestamp`), you can set: `Boolean`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `BigDecimal`, `Array[Byte]`, `Blob`, `Clob`, `Date`, `Time`, as well as `Object` and `null`. There are *setXXX* methods on `PositionedParameters` for `Option` types, too.

There's further symmetry with `GetResuts` in that we could have used >> in our `SetParameter`:

```
(dt, pp) => pp >> new Timestamp(dt.getMillis)
```

With this in place we can construct Plain SQL updates using `DateTime` instances:

```
val now =
  sqlu"""UPDATE "message" SET "ts" = ${DateTime.now}"""
```

Without the `SetParameter[DateTime]` instance the compiler would tell you:

```
could not find implicit SetParameter[DateTime]
```

7.3 Typed Checked Plain SQL

We've mentioned the risks of Plain SQL, which can be summarized as not discovering a problem with your query until runtime. The `tsql` interpolator removes some of this risk, but at the cost of requiring a connection to a database at compile time.

7.3.1 Compile Time Database Connections

To get started with `tsql` we provide a database configuration information on a class:

```
import slick.backend.StaticDatabaseConfig

@StaticDatabaseConfig("file:src/main/resources/application.conf#tsql")
object PlainExample extends App {
  ...
}
```

The `@StaticDatabaseConfig` syntax is called an *annotation*. This particular `StaticDatabaseConfig` annotation is telling Slick to use the connection called "tsql" in our configuration file. That entry will look like this:

```
tsql = {
  driver = "slick.driver.H2Driver$"
  db {
    connectionPool = disabled
    url = "jdbc:h2:mem:chapter06; ↩
      INIT=runscript from 'src/main/resources/integration-schema.sql'"
    driver = "org.h2.Driver"
    keepAliveConnection = false
  }
}
```

Note the `$` in the driver class name is not a typo. The class name is being passed to Java's `Class.forName`, but of course Java doesn't have a singleton as such. The Slick configuration does the right thing to load `$MODULE` when it sees `$`. This interoperability with Java is described in Chapter 29 of *Programming in Scala*.

You won't have seen this when we introduced the database configuration in Chapter 1. That's because this `tsql` configuration has a different format, and combines the Slick driver (`slicker.driver.H2Driver$`) and the JDBC driver (`org.h2.Drvier`) in one entry.

A consequence of supplying a `@StaticDatabaseConfig` is that you can define one databases configuration for your application and a different one for the compiler to use. That is, perhaps you are running an application, or test suite, against an in-memory database, but validating the queries at compile time against a full-populated production-like integration database.

In the example above, and the accompanying example code, we use an in-memory database to make Slick easy to get started with. However, an in-memory database is empty by default, and that would be no use for checking queries against. To work around that we provide an `INIT` script to populate the in-memory database.

7.3.2 Type Checked Plain SQL

With the `@StaticDatabaseConfig` in place we can use `tsql`:

```
val action: DBIO[Seq[String]] =
  tsql"""select "content" from "message""""
```

You can run that query as you would `sql` or `sqlu` query. You can also use custom types via `SetParameter` type class. However, `GetResult` type classes are not supported for `tsql`.

To make this interesting, let's get the query wrong and see what happens:

```
val action: DBIO[Seq[String]] =
  tsql"""select "content", "id" from "message""""
```

Do you see what's wrong? If not, don't worry because the compiler will find the problem:

```
type mismatch;
[error]  found    : SqlStreamingAction[
                                        Vector[(String, Int)],
                                        (String, Int),Effect
                                      ]
[error]  required : DBIO[Seq[String]]
[error]     (which expands to) DBIOAction[Seq[String],NoStream,Effect.All]
```

The compiler wants a `String` for each row, because that's what we've declared the result to be. However it is found, via the database, that the query will return `(String,Int)` rows.

If we had omitted the type declaration, the action would have the inferred type of `DBIO[Seq[(String,Int)]]`. So if you want to catch these kinds of mismatches, it's good practice to declare the type you expect when using `tsql`.

Let's see other kinds of errors the compiler will find.

How about if the SQL is just wrong:

```
val action: DBIO[Seq[String]] =
  tsql"""select "content" from "message" where"""
```

This is incomplete SQL, and the compiler tells us:

```
exception during macro expansion: ERROR: syntax error at end of input
[error]   Position: 38
[error]     tsql"""select "content" from "message" WHERE"""
[error]     ^
```

And if we get a column name wrong...

```
val action: DBIO[Seq[String]] =
  tsql"""select "text" from "message" where"""
```

...that's also a compile error too:

```
Exception during macro expansion: ERROR: column "text" does not exist
[error]   Position: 8
[error]     tsql"""select "text" from "message""""
[error]     ^
```

Of course, in addition to selecting rows, you can insert:

```
val greeting = "Hello"
val action: DBIO[Seq[Int]] =
  tsql"""insert into "message" ("content") values ($greeting)"""
```

Note that at run time, when we execute the query, a new row will be inserted. At compile time, Slick uses a facility in JDBC to compile the query and retrieve the meta data without having to run the query. In other words, at compile time the database is not mutated.

7.4 Take Home Points

Plain SQL allows you a way out of any limitations you find with Slick's lifted embedded style of querying.

Two main string interpolators for SQL are provided: `sql` and `sqlu`:

- Values can be safely substituted into Plain SQL queries using `${expression}`.
- Custom types can be used with the interpolators providing an implicit `GetResult` (select) or `SetParameter` (update) is in scope for the type.
- Raw values can be spliced into a query with `$#`. Use this with care: end-user supplied information should never be spliced into a query.

The `tsql` interpolator will check Plain SQL queries against a database at compile time. The database connection is used to validate the query syntax, and also discover the types of the columns being selected. To make best use of this, always declare the type of the query you expect from `tsql`.

7.5 Exercises

The examples for this section are in the `chapter-07` folder, in the source files `selects.scala`, `updates.scala`, and `tsql.scala`. Familiarise yourself with the schema and example data from `chat_schema.scala`.

7.5.1 Plain Selects

Let's get warmed up some some simple exercises.

Write the following four queries as Plain SQL queries:

- Count the number of rows in the message table.
- Select the content from the messages table.
- Select the length of each message ("content") in the messages table.
- Select the content and length of each message.

Tips:

- Remember that you need to use double quotes around table and column names in the SQL.
- We gave the database tables names which are singular: `message`, `user`, etc.

See the solution

7.5.2 Conversion

Convert the following lifted embedded query to a Plain SQL query.

```
val whoSaidThat =
  messages.join(users).on(_.senderId === _.id).
  filter{ case (message,user) =>
    message.content === "Open the pod bay doors, HAL."}.
  map{ case (message,user) => user.name }

exec(whoSaidThat.result)
// res1: Seq[String] = Vector(Dave)
```

Tips:

- If you're not familiar with SQL syntax, you can peak at the `whoSaidThat.result.statements`.
- Remember that strings in SQL are wrapped in single quotes, not double quotes.
- In the database the `senderId` is in a column called `sender`.

See the solution

7.5.3 Substitution

Complete the implementation of this method using a Plain SQL query:

```
def whoSaid(content: String): DBIO[Seq[String]] =
  ???

exec(whoSaid("Open the pod bay doors, HAL."))
// res1: Seq[String] = Vector(Dave)
```

This should be a small change to your solution to the last exercise.

See the solution

7.5.4 First and Last

This H2 query returns the alphabetically first and last messages:

```
exec(sql"""
      select min("content"), max("content")
      from "message" """.
      as[(String,String)])
// res1: Vector[(String, String)] = Vector(
//   (Affirmative, Dave. I read you., Well, whaddya think?)
// )
```

In this exercise we want you to write a `GetResult` type class instance so that the result of the query is one of these:

```
case class FirstAndLast(first: String, last: String)
```

The steps are:

1. Remember to `import slick.jdbc.GetResult`.
2. Provide an implicit value for `GetResult[FirstAndLast]`
3. Make the query use `as[FirstAndLast]`

See the solution

7.5.5 Plain Change

We can use Plain SQL to modify the database. That means inserting rows, updating rows, deleting rows, and also modifying the schema.

Go ahead and create a new table, using Plain SQL, to store the crew's jukebox playlist. Just store a song title. Insert a row into the table.

See the solution

7.5.6 Robert Tables

We're building a web site that allows searching for users by their email address:

```
def lookup(email: String) =
  sql"""select "id" from "user" where "user"."email" = '#${email}'"""

// Example use:
exec(lookup("dave@example.org").as[Long].headOption)
// res1: Option[Long] = Some(1)
```

What the problem with this code?

See the solution

Appendix A

Using Different Database Products

As mentioned during the introduction, H2 is used throughout the book for examples. However Slick also supports PostgreSQL, MySQL, Derby, SQLite, Oracle, and Microsoft Access. To work with DB2, SQL Server or Oracle you need a commercial license. These are the closed source *Slick Drivers* known as the *Slick Extensions*.

For MS-SQL and Oracle users, there is an open source Slick driver in development. You can find out more about this from the FreeSlick GitHub page.

A.1 Changes

If you want to use a different database for the exercises in the book, you will need to make changes detailed below.

In summary you will need to ensure that:

- you have installed the database (details beyond the scope of this book);
- a database is available with the correct name;
- the `build.sbt` file has the correct dependency;
- the correct JDBC driver is referenced in the code; and
- the correct Slick driver is used.

Each chapter uses its own database—so these steps will need to be applied for each chapter.

We've given detailed instructions for two populate databases below.

A.2 PostgreSQL

If it is not currently installed, it can be downloaded from the PostgreSQL website.

A.2.1 Create a Database

Create a database named `chapter-01` with user `essential`. This will be used for all examples and can be created with the following:

```
CREATE DATABASE "chapter-01" WITH ENCODING 'UTF8';
CREATE USER "essential" WITH PASSWORD 'trustno1';
GRANT ALL ON DATABASE "chapter-01" TO essential;
```

Confirm the database has been created and can be accessed:

```
$ psql -d chapter-01 essential
```

A.2.2 Update `build.sbt` Dependencies

Replace

```
"com.h2database" % "h2" % "1.4.185"
```

with

```
"org.postgresql" % "postgresql" % "9.3-1100-jdbc41"
```

If you are already in SBT, type `reload` to load this changed build file. If you are using an IDE, don't forget to regenerate any IDE project files.

A.2.3 Update JDBC References

Replace `application.conf` parameters with:

```
chapter01 = {
  connectionPool      = disabled
  url                 = jdbc:postgresql:chapter-01
  driver              = org.postgresql.Driver
  keepAliveConnection = true
  users               = essential
  password            = trustno1
}
```

A.2.4 Update Slick Driver

Change the import from `slick.driver.H2Driver.api._` to `slick.driver.PostgresDriver.api._`.

A.3 MySQL

If it is not currently installed, it can be downloaded from the MySQL website.

A.3.1 Create a Database

Create a database named `chapter-01` with user `essential`. This will be used for all examples and can be created with the following:

```
CREATE USER 'essential'@'localhost' IDENTIFIED BY 'trustno1';
CREATE DATABASE `chapter-01` CHARACTER SET utf8 COLLATE utf8_bin;
GRANT ALL ON `chapter-01`.* TO 'essential'@'localhost';
FLUSH PRIVILEGES;
```

Confirm the database has been created and can be accessed:

```
$ mysql -u essential chapter-01 -p
```

A.3.2 Update `build.sbt` Dependencies

Replace

```
"com.h2database" % "h2" % "1.4.185"
```

with

```
"mysql" % "mysql-connector-java" % "5.1.34"
```

If you are already in SBT, type `reload` to load this changed build file. If you are using an IDE, don't forget to regenerate any IDE project files.

A.3.3 Update JDBC References

Replace `Database.forURL` parameters with:

```
chapter01 = {
  connectionPool      = jdbc:mysql://localhost:3306/chapter-01     ↵
                                &useUnicode=true             ↵
                                &amp;characterEncoding=UTF-8 ↵
                                &amp;autoReconnect=true
  url                 = jdbc:postgresql:chapter-01
  driver              = com.mysql.jdbc.Driver
  keepAliveConnection = true
  users               = essential
  password            = trustno1
}
```

A.3.4 Update Slick Driver

Change the import from `slick.driver.H2Driver.api._` to `slick.driver.MySQLDriver.api._`.

Appendix B

Play Framework Integration

You'll probably want to provide a way for people to interact with your lovely schema at some point—possibly via the internet! The Play Framework is a popular tool for developing just such an interface. A plugin called Play Slick makes doing so relatively painless. The Play Slick plugin integrates Slick with Play's lifecycle, ensuring all resources are freed when an application is stopped.

B.1 Overview

To use the plugin we need to make three changes:

- sbt configuration,
- application configuration, and
- code.

Don't panic, these are all small changes.

> **Tip**
>
> **Run the Code**
>
> By now, you know the drill. You'll find the code for this section in the *play-slick-example* folder over at the associated GitHub repository.
>
> From there start SBT and at the SBT > prompt run:
>
> ```
> ~run
> ```
>
> You can then navigate to `http://localhost:9000/`. The tilde symbol means any changes you make will be automagically recompiled and you can refresh your browser and see the effect.

B.2 sbt Configuration

We need to tell sbt about Play and the Play Slick plugin.

B.2.1 Play

Play has its own sbt plugin which adds the following capabilities Play console, dependencies and functionality such as live reloading to sbt. To include this add the following to `plugins.sbt` in the `project` directory:

```
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.4.3")
```

We also need to enable the plugin for a given project in `build.sbt`:

```
lazy val root = (project in file(".")).enablePlugins(PlayScala)
```

If you interested in learning Play Underscore has an excellent book—Essential Play.

B.2.2 Play Slick Plugin

Now, on to the bits we care about—The Play Slick plugin! This is a library dependency added to our `build.sbt`:

```
  "com.typesafe.play" %% "play-slick" % "1.1.0"
```

> **Tip**
>
> **Plugin Version**
>
> The project provides a handy table to determine which version of the plugin is appropriate for your project. For Essential Slick, we are using the latest available version of the plugin `1.1.0`.

B.3 Application Configuration

Play Slick expects Slick datasources to be located under `slick.dbs` with the database to labeled `default`. Leaving `application.conf` as follows:

```
slick {
  dbs {
    default {
      driver = "slick.driver.H2Driver$",
      db {
        driver   = "org.h2.Driver"
        url      = "jdbc:h2:mem:play"
      }
    }
  }
}
```

It is worth noting we needed to declare **both** the JDBC and Slick drivers. If you want to use a label other than `default` for a database, the Database configuration outlines how to do this.

B.3.1 Code

Let's review our schema implementation:

```
trait Profile {
  val profile: slick.driver.JdbcProfile
}

trait Tables {
  this: Profile =>

  import profile.api._

  ...
}
case class Schema(val profile: JdbcProfile) extends Tables with Profile
```

We have defined a trait to hold our Slick profile and mixed this into our `Tables` definitions. Giving us access to the contents of the profile. We bring the `Profile` and `Tables` traits together in our `Schema` case class which provides a concrete implementation of the profile. This is sometimes known as the cake pattern—less a bakery of doom and more a boulangerie of tastiness, at least in our case.

Play Slick has its own self type for our `Tables` trait called `HasDatabaseConfig`:

```
trait Tables {
  this: HasDatabaseConfig[JdbcProfile] =>

  import driver.api._
```

Note: The import exposing the profile functionality has also changed.

`HasDatabaseConfig` is doing a little more than our `Profile` case class. It also provides access to our database via the method `db` as well as the Slick profile via `driver`.

Our next change, is a little larger, but is mostly content shuffling. The recipe for our cake has changed a little, our `Schema` is no longer a case class, but a case object. This is because we are no longer passing in a Profile, but rather mixing in the `HasDatabaseConfig` trait. We are also providing a way to get a concrete implementation of `DatabaseConfig`, using `DatabaseConfigProvider`.

```
case object Schema extends HasDatabaseConfig[JdbcProfile] with Tables {
  //We use DatabaseConfigProvider to retrieve a database config
  protected val dbConfig =
    DatabaseConfigProvider.get[JdbcProfile](Play.current)

  //and import the appropriate driver API
  import driver.api._
```

What is happening in the above snippet? We have provided `HasDatabaseConfig` with a profile, by overriding the value `dbConfig`. `DatabaseConfigProvider` is used to retrieve a DatabaseConfig based on `application.conf`.

B.3.2 DatabaseConfigProvider

`Play` provides access to Play's global features, `current` is the current `Application` instance. This meta information about the currently running application, including configuration `application.conf`. `Play.current`

is an implicit parameter and doesn't need to be supplied. It does however help us grok where our database configuration is from.

Finally, we need to provide access to runnable versions of our database queries in our `Schema` case object.

```
    def populate    = db.run(schemaPopulate)
    def msgs        = db.run(namedMessages.result)
```

With this all in place, we can start using Slick in our Play application!

Note

Our `Profile` trait and the `Tables` self type are no longer needed and can be removed.

B.3.3 Calling Slick

Let's populate the schema as we have done throughout the book with the standard conversation.

```
object Global extends GlobalSettings {

  //When the application starts up, populate the schema.
  override def onStart(app: Application) =
    Await.result(Schema.populate, Duration.Inf)

}
```

And, provide a way for people to view the messages:

```
object Application extends Controller {

  //Display index page
  def index = Action { Ok(views.html.index()) }


  //Don't construct your JSON via tuples, use case classes and an encoder.
  def messages = Action.async  {
    Schema.msgs.map{s =>
      Ok(
        JsArray(s.map(t =>Json.obj("sender" -> t._1, "content" -> t._2)))
        )
    }
  }

}
```

Appendix C

Solutions to Exercises

C.1 Basics

C.1.1 Solution to: Bring Your Own Data

Here's the solution:

```
exec(messages += Message("Dave","What if I say 'Pretty please'?"))
// res5: Int = 1
```

The return value indicates that 1 row was inserted. Because we're using an auto-incrementing primary key, Slick ignores the `id` field for our `Message` and asks the database to allocate an `id` for the new row. It is possible to get the insert query to return the new `id` instead of the row count, as we shall see next chapter.

Here are some things that might go wrong:

If you don't pass the action created by += to `db` to be run, you'll get back the `Action` object instead.

```
messages += Message("Dave","What if I say 'Pretty please'?")
//res6: slick.profile.FixedSqlAction[
//                  Int,
//                  slick.dbio.NoStream,slick.dbio.Effect.Write
//                  ] =
// slick.driver.JdbcActionComponent$InsertActionComposerImpl
//                                          $$anon$8@7e0e6d1e
```

If you don't wait for the future to complete, you'll see just the future itself:

```
db.run(messages += Message("Dave","What if I say 'Pretty please'?"))
// res7: scala.concurrent.Future[Int] =
// scala.concurrent.impl.Promise$DefaultPromise@652a41e8
```

Return to the exercise

C.1.2 Solution to: Bring Your Own Data Part 2

Here's the code:

```
exec(messages.filter(_.sender === "Dave").result)

// res0: Seq[Example.MessageTable#TableElementType] = Vector(
//   Message(Dave,Hello, HAL. Do you read me, HAL?,1),
//   Message(Dave,Open the pod bay doors, HAL.,3),
//   Message(Dave,What if I say 'Pretty please'?,5))
```

Here are some things that might go wrong:

Note that the parameter to `filter` is built using a triple-equals operator, ===, not a regular ==. If you use == you'll get an interesting compile error:

```
exec(messages.filter(_.sender == "Dave").result)

//<console>:18: error: inferred type arguments [Boolean] do not conform to
//                     method filter's
//  type parameter bounds [T <: slick.lifted.Rep[_]]
//              exec(messages.filter(_.sender == "Dave").result)
//                                   ^
//<console>:18: error: type mismatch;
// found   : Example.MessageTable => Boolean
// required: Example.MessageTable => T
//              exec(messages.filter(_.sender == "Dave").result)
//                                                ^
//<console>:18: error: Type T cannot be a query condition
//  (only Boolean, Rep[Boolean] and Rep[Option[Boolean]] are allowed
//              exec(messages.filter(_.sender == "Dave").result)
//                                      ^
```

The trick here is to notice that we're not actually trying to compare `_.sender` and `"Dave"`. A regular equality expression evaluates to a `Boolean`, whereas === builds an SQL expression of type `Rep[Boolean]` (Slick uses the `Rep` type to represent expressions over `Columns` as well as `Columns` themselves.). The error message is baffling when you first see it but makes sense once you understand what's going on.

Finally, if you forget to call `result`, you'll end up with a compilation error as `exec` and the call it is wrapping `db.run` both expect actions:

```
exec(messages.filter(_.sender === "Dave"))
<console>:18: error: type mismatch;
 found   : slick.lifted.Query[Example.MessageTable,
                              Example.MessageTable#TableElementType,
                              Seq]
    (which expands to)  slick.lifted.Query[Example.MessageTable,
                                           Example.Message,
                                           Seq]
 required: slick.driver.H2Driver.api.DBIO[?]
    (which expands to)  slick.dbio.DBIOAction[?,
                                              slick.dbio.NoStream,
                                              slick.dbio.Effect.All]
              exec(messages.filter(_.sender === "Dave"))
                                   ^
```

`Query` types tend to be verbose, which can be distracting from the actual cause of the problem (which is that we're not expecting a `Query` object at all). We will discuss `Query` types in more detail next chapter.

Return to the exercise

C.2 Selecting Data

C.2.1 Solution to: Count the Messages

```
val results = exec(halSays.length.result)
```

You could also use `size`, which is an alias for `length`.

Return to the exercise

C.2.2 Solution to: Selecting a Message

```
val query = for {
  message <- messages if message.id === 1L
} yield message

val results = exec(query.result)
```

Asking for 999, when there is no row with that ID, will give back an empty collection.

Return to the exercise

C.2.3 Solution to: One Liners

```
val results = exec(messages.filter(_.id === 1L).result)
```

Return to the exercise

C.2.4 Solution to: Checking the SQL

The code you need to run is:

```
val sql = messages.filter(_.id === 1L).result.statements
println(sql)
```

The result will be something like:

```
select
  x2."id", x2."sender", x2."content", x2."ts"
from
  "message" x2
where
  x2."id" = 1
```

From this we see how `filter` corresponds to a SQL `where` clause.

Return to the exercise

C.2.5 Solution to: Is HAL Real?

That's right, we want to know if HAL `exists`:

```
val query = messages.filter(_.sender === "HAL").exists


println(s"The query is:  ${query.result.statements}")
println(s"The result is: ${exec(query.result)}")
```

The query will return `true` as we do have records from HAL, and Slick will generate the following SQL:

```
select exists(
  select "sender", "content", "id"
  from "message"
  where "sender" = 'HAL'
)
```

Return to the exercise

C.2.6 Solution to: Selecting Columns

```
val query = messages.map(_.content)
println(s"The query is:  ${query.result.statements}")
println(s"The result is: ${exec(query.result)}")
```

You could have also said:

```
val query = for { message <- messages } yield message.content
```

The query will just return the `content` column from the database:

```
select x2."content" from "message" x2
```

Return to the exercise

C.2.7 Solution to: First Result

```
val msg1 = messages.filter(_.sender === "HAL").map(_.content).result.head
```

You should get an action that produces "Affirmative, Dave. I read you."

For Alice, head will throw a run-time exception as we are trying to return the head of an empty collection. Using `headOption` will prevent the exception.

Return to the exercise

C.2.8 Solution to: Then the Rest

It's pagination's friends `drop` and `take` to the rescue:

```
val msgs = messages.filter(_.sender === "HAL").drop(1).take(5).result
```

HAL has only three messages in total. Therefore our result set should contain two messages:

```
Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,4)
Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,6)
```

And asking for any more messages will result in an empty collection.

```
val msgs = exec(
            messages.
              filter(_.sender === "HAL").
              drop(10).
              take(10).
              result
          )
// msgs: Seq[Example.MessageTable#TableElementType] = Vector()
```

Return to the exercise

C.2.9 Solution to: The Start of Something

```
messages.filter(_.content startsWith "Open")
```

The query is implemented in terms of `LIKE`:

```
select
  x2."id", x2."sender", x2."content", x2."ts"
from
  "message" x2
where
  x2."content" like 'Open%' escape '^'
```

Return to the exercise

C.2.10 Solution to: Liking

The query is:

```
messages.filter(_.content.toLowerCase like "%do%")
```

The SQL will turn out as:

```
select
  x2."id", x2."sender", x2."content", x2."ts"
from
  "message" x2
where
  lower(x2."content") like '%do%'
```

There are three results: "*Do* you read me", "Open the pod bay *do*ors", and "I'm afraid I can't *do* that".

Return to the exercise

C.2.11 Solution to: Client-Side or Server-Side?

The query Slick generates looks something like this:

```
select '(message Ref @421681221).content!' from "message" x2
```

That is a select expression for a strange constant string.

The `_.content + "!"` expression converts `content` to a string and appends the exclamation point. What is `content`? It's a `Rep[String]`, not a `String` of the content. The end result is that we're seeing something of the internal workings of Slick.

This is an unfortunate effect of Scala allowing automatic conversion to a `String`. If you are interested in disabling this Scala behaviour, tools like WartRemover can help.

It is possible to do this mapping in the database with Slick. We just need to remember to work in terms of `Rep[T]` classes:

```
messages.map(m => m.content ++ LiteralColumn("!"))
```

Here `LiteralColumn[T]` is type of `Rep[T]` for holding a constant value to be inserted into the SQL. The ++ method is one of the extension methods defined for any `Rep[String]`.

Using ++ will produce the desired query:

```
select "content"||'!' from "message"
```

You can also write:

```
messages.map(m => m.content ++ "!")
```

... as "!" will be lifted to a 'Rep[String]"

This exercise highlights that inside of a `map` or `filter` you are working in terms of `Rep[T]`. You should become familiar with the operations available to you. The tables we've included in this chapter should help with that.

Return to the exercise

C.3 Creating and Modifying Data

C.3.1 Solution to: Methodical Inserts

If you tried

```
exec(
  messages returning messages +=
    Message("Dave", "So... what do we do now?"))
```

You will have seen the exception :

```
// slick.SlickException:
//   This DBMS allows only a single AutoInc column ↩
//     to be returned from an INSERT
//   at ...
```

Recall from Retrieving Rows on Insert that H2 doesn't support returning more than the `id` field.

We need to use `into`:

```
val messagesReturningRow =
  messages returning messages.map(_.id) into { (message, id) =>
    message.copy(id = id)
  }
// messagesReturningRow: slick.driver.H2Driver.IntoInsertActionComposer[
// Example.MessageTable#TableElementType,Example.Message] =
// slick.driver.
// JdbcActionComponent$ReturningInsertActionComposerImpl@6cfcdefc


val insert:Message => DBIO[Message] = m => messagesReturningRow += m
// insert: Example.Message =>
// slick.driver.H2Driver.api.DBIO[Example.Message]
// = <function1>

exec(insert(Message("Dave", "So... what do we do now?")) )
// res4: messagesReturningRow.SingleInsertResult =
//                Message(Dave,So... what do we do now?,6)
```

Return to the exercise

C.3.2 Solution to: Get to the Specifics

The requirements of the `messages` table is `sender` and `content` can not be null. Given this, we can correct our query:

```
exec(messages.map( m => (m.sender,m.content)) += (("HAL","Hellllo Dave")))
```

Return to the exercise

C.3.3 Solution to: Bulk All the Inserts

It is `messagesReturningRow` to the rescue once again:

```
val messagesReturningRow =
  messages returning messages.map(_.id) into { (message, id) =>
    message.copy(id = id)
  }
// messagesReturningRow: slick.driver.H2Driver.IntoInsertActionComposer[
//   Example.MessageTable#TableElementType,
//   Example.Message
// ] = ...

exec(messagesReturningRow ++= conversation)
//res16: messagesReturningRow.MultiInsertResult = Vector(
// Message(Bob,Hi Alice,28),
// Message(Alice,Hi Bob,29),
// Message(Bob,Are you sure this is secure?,30),
// Message(Alice,Totally, why do you ask?,31
// Message(Bob,Oh, nothing, just wondering.,32),
// Message(Alice,Ten was too many messages,33),
// Message(Bob,I could do with a sleep,34),
// Message(Alice,Let's just to to the point ,35),
// Message(Bob,Okay okay, no need to be tetchy.,36),
// Message(Alice,Humph!,37))
```

Return to the exercise

C.3.4 Solution to: No Apologies

```
messages.filter(_.content like "%sorry%").delete
```

Return to the exercise

C.3.5 Solution to: Update Using a For Comprehension

```
val query = for {
  message <- messages
  if message.sender === "HAL"
} yield (message.sender, message.content)

val rowsAffected = query.update("HAL 9000", "Rebooting, please wait...")
```

Return to the exercise

C.3.6 Solution to: Selective Memory

```
val selectiveMemory =
  messages.filter{
   _.id in messages.
      filter { _.sender === "HAL" }.
      sortBy { _.id asc }.
      map    {_.id}.
      take(2)
  }.delete

exec(selectiveMemory)
//res1: Int = 2
```

Return to the exercise

C.4 Combining Actions

C.4.1 Solution to: And Then what?

```
exec( drop andThen create andThen populate)
```

Return to the exercise

C.4.2 Solution to: First!

There are two elements to this problem:

1. being able to use the result of a count, which is what `flatMap` gives us; and
2. combining two inserts via `andThen`.

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

def insert(m: Message): DBIO[Int] =
    messages.size.result.flatMap {
      case 0 =>
        (messages += Message(m.sender, "First!")) andThen (messages += m)
      case n =>
        messages += m
    }

// Throw away all the messages:
exec(messages.delete)
// res1: Int = 3

// Try out the method:
exec {
  insert(Message("Me", "Hello?"))
}
```

```
// res2: Int = 1

// What's in the database?
exec(messages.result).foreach(println)
// Message(Me,First!,7)
// Message(Me,Hello?,8)
```

Return to the exercise

C.4.3 Solution to: There Can be Only One

You may not have seen `+:` before: it is `cons` for Seq.

```
  def onlyOne[T](action:DBIO[Seq[T]]):DBIO[T] = action.flatMap{ xs =>
    xs match {
      case x +: Nil =>
        DBIO.successful(x)
      case ys       =>
        DBIO.failed(
          new RuntimeException(s"Expected 1 result, not ${ys.length}")
        )
    }
  }

exec(onlyOne(boom))
//java.lang.RuntimeException: Expected 1 result, not 2
//  ...

exec(onlyOne(happy))
// Message(HAL, I'm sorry, Dave. I'm afraid I can't do that., 4)
```

Return to the exercise

C.4.4 Solution to: Let's be Reasonable

There are several ways we could have implemented this, the simplest is using `asTry`

```
def exactlyOne[T](action:DBIO[Seq[T]]):DBIO[Try[T]] = onlyOne(action).asTry


exec(exactlyOne(happy))
// res26: scala.util.Try[Example.MessageTable#TableElementType] =
//   Success(Message(HAL,I'm sorry, Dave. I'm afraid I can't do that.,4))


exec(exactlyOne(boom))
// res27: scala.util.Try[Example.MessageTable#TableElementType] =
//   Failure(java.lang.RuntimeException: Expected 1 result, not 2)
```

Return to the exercise

C.4.5 Solution to: Filtering

This is a fairly simple example of using `map`:

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

def myFilter[T]
  (action: DBIO[T])
  (p: T => Boolean)
  (alternative: => T) =
    action.map {
      case t if p(t) => t
      case _ => alternative
    }
```

Return to the exercise

C.4.6 Solution to: Unfolding

The trick here is to recognize that:

1. this is a recursive problem, so we need to define a stopping condition;
2. we need `flatMap` to pass a value long; and
3. we need to accumulate results from each step.

The solution below is generalized with T rather than having a hard-coded `String` type.

```
def unfold[T]
  (z: T, acc: Seq[T] = Seq.empty)
  (f: T => DBIO[Option[T]]): DBIO[Seq[T]] =
  f(z).flatMap {
    case None    => DBIO.successful(acc :+ z)
    case Some(t) => unfold(t, acc :+ z)(f)
  }

val path: DBIO[Seq[String]] =
  unfold("Podbay") {
     roomName => floorplan
          .filter(_.name === roomName)
          .map(_.connectsTo).result.headOption
   }

println( exec(path) )
// List(Podbay, Galley, Computer, Engine Room)
```

Return to the exercise

C.5 Data Modelling

C.5.1 Solution to: Turning a Row into Many Case Classes

A suitable projection is:

```
def pack(row: (String, String, String, String, String, Long)): User =
  User(
    EmailContact(row._1, row._2),
    Address(row._3, row._4, row._5),
    row._6
  )

def unpack(user: User): Option[(String, String, String, String, ↩
                                                    String, Long)] =
  Some((user.contact.name, user.contact.email,
        user.address.street, user.address.city, user.address.country,
        user.id))

def * = (name, email, street, city, country, id) <> (pack, unpack)
```

We can insert and query as normal:

```
users += User(
  EmailContact("Dr. Dave Bowman", "dave@example.org"),
  Address("123 Some Street", "Any Town", "USA")
 )
```

Executing `exec(users.result)` will produce:

```
Vector(
  User(
    EmailContact(Dr. Dave Bowman,dave@example.org),
    Address(123 Some Street,Any Town,USA),
    1
  )
)
```

You can continue to select just some fields. For example `users.map(_.email).result` will produce:

```
Vector(dave@example.org)
```

However, notice that if you used `users.schema.create`, only the columns defined in the default projection were created in the H2 database.

Return to the exercise

C.5.2 Solution to: Filtering Optional Columns

We can decide on the query to run in the two cases from inside our application:

```
def filterByEmail(email: Option[String]) =
  if (email.isEmpty) users
  else users.filter(_.email === email)
```

You don't always have to do everything at the SQL level.

Return to the exercise

C.5.3 Solution to: Inside the Option

As the `email` value is optional we can't simply pass it to `startsWith`.

```
def filterByEmail(email: Option[String]) =
  email.map(e =>
    users.filter(_.email startsWith e)
  ) getOrElse users
```

Return to the exercise

C.5.4 Solution to: Matching or Undecided

This problem we can represent in SQL, so we can do it with one query:

```
def filterByEmail(email: Option[String]) =
  users.filter(u => u.email.isEmpty || u.email === email)
```

Return to the exercise

C.5.5 Solution to: Enforcement

We get a runtime exception as we have violated referential integrity. There is no row in the `user` table with a primary id of `3000`.

Return to the exercise

C.5.6 Solution to: Model This

There are a few ways to model this table regarding constraints and defaults. Here's one way, where the default is on the database, and the unique primary key is simply the user's `id`:

```
case class Bill(userId: Long, amount: BigDecimal)

class BillTable(tag: Tag) extends Table[Bill](tag, "bill") {
  def userId = column[Long]("user", O.PrimaryKey)
  def amount = column[BigDecimal]("dollars", O.Default(12.00))
  def * = (userId, amount) <> (Bill.tupled, Bill.unapply)
  def user = foreignKey("fk_bill_user", userId, users)  ↩
                      (_.id, onDelete=ForeignKeyAction.Restrict)
}

lazy val bills = TableQuery[BillTable]
```

Exercise the code as follows:

```
bills += Bill(daveId, 12.00)
println(exec(bills.result)

// Unique index or primary key violation:
// exec(bills += Bill(daveId, 24.00))

// Referential integrity constraint violation: "fk_bill_user:
// exec(users.filter(_.name === "Dave").delete)

// Who has a bill?
val has = for {
  b <- bills
  u <- b.user
} yield u

// Who doesn't have a bill?
val hasNot = for {
  u <- users
  if !(u.id in bills.map(_.userId))
} yield u
```

Return to the exercise

C.5.7 Solution to: Mapping Enumerations

The first step is to supply an implicit to and from the database values:

```
object UserRole extends Enumeration {
  type UserRole = Value
  val Owner   = Value("O")
  val Regular = Value("R")
}

import UserRole._
implicit val userRoleMapper =
  MappedColumnType.base[UserRole, String](_.toString, UserRole.withName(_))
```

Then we can use the `UserRole` in the table definition:

```
case class User(name: String,
                userRole: UserRole = Regular,
                id: UserPK = UserPK(0L))

class UserTable(tag: Tag) extends Table[User](tag, "user") {
  def id   = column[UserPK]("id", O.PrimaryKey, O.AutoInc)
  def name = column[String]("name")
  def role = column[UserRole]("role", O.Length(1,false))

  def * = (name, role, id) <> (User.tupled, User.unapply)
}
```

Return to the exercise

C.5.8 Solution to: Alternative Enumerations

The only change to make is to the mapper, to go from a `UserRole` and `String`, to a `UserRole` and `Int`:

```
implicit val userRoleMapper =
  MappedColumnType.base[UserRole, Int](
  _.id,
  v => UserRole.values.find(_.id == v) getOrElse Regular)
```

Return to the exercise

C.5.9 Solution to: Custom Boolean

This is similar to the `Flag` example above, except we need to handle multiple values from the database.

```
sealed trait Priority
case object HighPriority extends Priority
case object LowPriority  extends Priority

implicit val priorityType =
  MappedColumnType.base[Priority, String](
    flag => flag match {
      case HighPriority => "y"
      case LowPriority  => "n"
    },
    str => str match {
      case "Y" | "y" | "+" | "high"         => HighPriority
      case "N" | "n" | "-" | "lo"   | "low" => LowPriority
  })
```

Return to the exercise

C.6 Joins and Aggregates

C.6.1 Solution to: Name of the Sender

```
val ex1 = for {
  m <- messages
  u <- users
  if u.id === m.senderId
} yield (m, u)

val ex2 = for {
  m <- messages
  u <- users
  if u.id === m.senderId
} yield (m.content, u.name)
```

```
val ex3 = ex2.sortBy{ case (content, name) => name }

val ex4 =
  messages.
   join(users).on(_.senderId === _.id).
   map    { case (msg, usr)     => (msg.content, usr.name) }.
   sortBy { case (content,name) => name }
```

Return to the exercise

C.6.2 Solution to: Messages of the Sender

```
def findByName(name: String): Query[Rep[Message], Message, Seq] = for {
  u <- users    if u.name === name
  m <- messages if m.senderId === u.id
} yield m
```

...or...

```
def findByName(name: String): Query[Rep[Message], Message, Seq] =
  users.filter(_.name === name).
  join(messages).on(_.id === _.senderId).
  map{ case (user, msg) => msg }
```

Return to the exercise

C.6.3 Solution to: Having Many Messages

SQL distinguishes between `WHERE` and `HAVING`. In Slick you just use `filter`:

```
val msgsPerUser =
   messages.join(users).on(_.senderId === _.id).
   groupBy { case (msg, user)  => user.name }.
   map     { case (name, group) => name -> group.length }.
   filter  { case (name, count) => count > 2 }
```

Running this on the data in *aggregates.scala* produces:

```
Vector((Frank,2), (HAL,4), (Dave,4))
```

Running it in the REPL, which has less data set up by default, produces:

```
Vector((HAL,4), (Dave,4))
```

Return to the exercise

C.6.4 Solution to: Collecting Results

You need all the code in the question and also what you know about action combinators:

```
def userMessages: DBIO[Map[User,Seq[Message]]] =
  users.join(messages).on(_.id === _.senderId).result.
  map { rows =>
    rows.groupBy{ case (user, message) => user }.
    mapValues(values => values.map{ case (name, msg) => msg })
  }
```

Return to the exercise

C.7 Plain SQL

C.7.1 Solution to: Plain Selects

The SQL statements are relatively simple. You need to take care to make the `as[T]` align to the result of the query.

```
exec(sql""" select count(*) from "message" """.as[Int])
// res1: Vector[Int] = Vector(8)

exec(sql""" select "content" from "message" """.as[String])
// res2: Vector[String] = Vector(
//   Hello, HAL. Do you read me, HAL?,
//   Affirmative, Dave. I read you.,
//   Open the pod bay doors, HAL.,
//   I'm sorry, Dave. I'm afraid I can't do that.,
//   Well, whaddya think?,
//   I'm not sure, what do you think?,
//   Are you thinking what I'm thinking?,
//   Maybe)

exec(sql""" select length("content") from "message" """.as[String])
// res3: Vector[String] = Vector(32, 30, 28, 44, 20, 32, 35, 5)

exec(sql""" select "content", length("content") from "message" """.as[(String,Int)])
// res4: Vector[(String, Int)] = Vector(
//  (Hello, HAL. Do you read me, HAL?,32),
//  (Affirmative, Dave. I read you.,30),
//  (Open the pod bay doors, HAL.,28),
// ...
```

Return to the exercise

C.7.2 Solution to: Conversion

There are various ways to implement this query in SQL. Here's one of them...

```
val whoSaidThat = sql"""
  select
    "name" from "user" u
  join
    "message" m on u."id" = m."sender"
  where
    m."content" = 'Open the pod bay doors, HAL.'
  """.as[String]

exec(whoSaidThat)
// res1: Seq[String] = Vector(Dave)
```

Return to the exercise

C.7.3 Solution to: Substitution

The solution just requires the use of a $ substitution:

```
def whoSaid(content: String): DBIO[Seq[String]] =
  sql"""
    select
      "name" from "user" u
    join
      "message" m on u."id" = m."sender"
    where
      m."content" = $content
    """.as[String]

exec(whoSaid("Open the pod bay doors, HAL."))
// res1: Seq[String] = Vector(Dave)

exec(whoSaid("Affirmative, Dave. I read you."))
//res2: Seq[String] = Vector(HAL)
```

Return to the exercise

C.7.4 Solution to: First and Last

```
import slick.jdbc.GetResult
// import slick.jdbc.GetResult

case class FirstAndLast(first: String, last: String)
// defined class FirstAndLast

implicit val GetFirstAndLast =
  GetResult[FirstAndLast](r => FirstAndLast(r.nextString, r.nextString))
// GetFirstAndLast: slick.jdbc.GetResult[FirstAndLast] = <function1>
```

```
val query =  sql""" select min("content"), max("content")
                    from "message" """.as[FirstAndLast]
// query: slick.profile.SqlStreamingAction[
//  Vector[FirstAndLast],FirstAndLast,slick.dbio.Effect
// ] = slick.jdbc.SQLActionBuilder$$anon$1@1fb692f9


exec(query)
// res1: Vector[FirstAndLast] = Vector(
//   FirstAndLast(Affirmative, Dave. I read you.,Well, whaddya think?)
// )
```

Return to the exercise

C.7.5 Solution to: Plain Change

For modifications we use `sqlu`, not `sql`:

```
exec(sqlu""" create table "jukebox" ("title" text) """)
// res1: Int = 0

exec(sqlu""" insert into "jukebox"("title")
             values ('Bicycle Built for Two') """)
// res2: Int = 1

exec(sql""" select "title" from "jukebox" """.as[String])
// res3: Vector[String] = Vector(Bicycle Built for Two)
```

Return to the exercise

C.7.6 Solution to: Robert Tables

If you are familiar with xkcd's Little Bobby Tables, the title of the exercise has probably tipped you off: #$ does not escape input.

This means a user could use a carefully crafted email address to do evil:

```
lookup("""';DROP TABLE "user";--- """).as[Long]
```

This "email address" turns into two queries:

```
SELECT * FROM "user" WHERE "user"."email" = '';
```

and

```
DROP TABLE "user";
```

Trying to access the users table after this will produce:

```
exec(users.result)
// org.h2.jdbc.JdbcSQLException: Table "user" not found; SQL statement:
// select "id", "name", "email" from "user" [42102-185]
//  at org.h2.message.DbException.getJdbcSQLException(DbException.java:345)
// ...
```

Yes, the table was dropped by the query.

Never use #$ with user supplied input.

Return to the exercise