Essential Slick

Richard Dallaway and Jonathan Ferguson

First Edition for Slick 2.1, July 2015

underscore

Essential Slick

First Edition for Slick 2.1, July 2015

Copyright 2015 Richard Dallaway and Jonathan Ferguson.

Published by Underscore Consulting LLP, Brighton, UK.

Contents

Preface **9**

How to Contact Us 9

Conventions Used in This Book 9

Typographical Conventions 10

Source Code 10

Callout Boxes 10

1 Basics **11**

1.1 Orientation 11

1.2 Running the Examples and Exercises 12

1.3 Example: A Sequel Odyssey 13

1.3.1 Library Dependencies 13

1.3.2 Importing Library Code 14

1.3.3 Defining our Schema 14

1.3.4 Example Queries 15

1.3.5 Connecting to the Database 16

1.3.6 Inserting Data 16

1.3.7 Selecting Data 17

1.3.8 For Comprehensions 18

1.4 Take Home Points 18

1.5 Exercise: Bring Your Own Data 19

2 Selecting Data **21**

2.1 Select All The Rows! 21

2.2 Filtering Results: The *filter* Method 22

2.3 The Query and TableQuery Types 22

2.4 Transforming Results: The *map* Method 23

2.5 Query Invokers 25

2.6 Column Expressions 26

2.6.1 Equality and Inequality Methods 26

2.6.2 String Methods 27
2.6.3 Numeric Methods 27
2.6.4 Boolean Methods 28
2.6.5 Option Methods and Type Equivalence 28
2.7 Controlling Queries: Sort, Take, and Drop 29
2.8 Take Home Points 30
2.9 Exercises 30
2.9.1 Count the Messages 31
2.9.2 Selecting a Message 31
2.9.3 One Liners 31
2.9.4 Selecting Columns 31
2.9.5 First Result 31
2.9.6 The Start of Something 31
2.9.7 Liking 32
2.9.8 Client-Side or Server-Side? 32

3 Creating and Modifying Data 33

3.1 Inserting Data 33
3.1.1 Inserting Single Rows 33
3.1.2 Primary Key Allocation 33
3.1.3 Inserting Specific Columns 34
3.1.4 Retrieving Primary Keys on Insert 35
3.1.5 Inserting Multiple Rows 36
3.2 Updating Rows 37
3.2.1 Updating a Single Field 37
3.2.2 Updating Multiple Fields 38
3.2.3 Updating with a Computed Value 38
3.3 Deleting Rows 39
3.4 Transactions 39
3.5 Logging Queries and Results 41
3.6 Take Home Points 41
3.7 Exercises 42
3.7.1 Insert New Messages Only 42
3.7.2 Rollback 42
3.7.3 Update Using a For Comprehension 42
3.7.4 Delete All The Messages 43

4 Data Modelling **45**
4.1 Application Structure . . . 45
4.1.1 Pattern Outline . . . 45
4.1.2 Working with a Profile . . . 46
4.1.3 Additional Considerations . . . 47
4.2 Representations for Rows . . . 47
4.2.1 Case Classes and <> . . . 47
4.2.2 Tuples . . . 48
4.2.3 Heterogeneous Lists . . . 49
4.2.4 Exercises . . . 51
4.3 Table and Column Representation . . . 51
4.3.1 Nullable Columns . . . 51
4.3.2 Primary Keys . . . 53
4.3.3 Compound Primary Keys . . . 54
4.3.4 Foreign Keys . . . 55
4.3.5 Table and Column Modifiers . . . 57
4.3.6 Exercises . . . 58
4.4 Custom Column Mappings . . . 60
4.5 Value Classes . . . 60
4.5.1 Exercises . . . 63
4.6 Sum Types . . . 64
4.6.1 Exercises . . . 65
4.7 Take Home Points . . . 65

5 Joins and Aggregates **67**
5.1 Implicit Joins . . . 67
5.2 Explicit Joins . . . 68
5.2.1 Inner Join . . . 70
5.2.2 Left Join . . . 70
5.2.3 Right Join . . . 72
5.2.4 Full Outer Join . . . 72
5.3 Zip Joins . . . 72
5.4 Joins Summary . . . 74
5.5 Seen Any Scary Queries? . . . 74
5.6 Aggregation . . . 75
5.6.1 Functions . . . 75
5.6.2 Grouping . . . 75

5.7 Take Home Points 78
5.8 Exercises 79
5.8.1 `HAVING` Many Messages 79
5.8.2 User Rooms 79

6 Plain SQL 81
6.1 Selects 81
6.1.1 Constructing Queries 82
6.1.2 Select with Custom Types 83
6.1.3 Case Classes 83
6.2 Updates 85
6.2.1 Composing Updates 85
6.2.2 Updating with Custom Types 86
6.3 Take Home Points 87
6.4 Exercises 87
6.4.1 Robert Tables 87
6.4.2 String Interpolation Mistake 87
6.4.3 Unsafe Composition 87

A Using Different Database Products 89
A.1 Changes 89
A.2 PostgreSQL 89
A.2.1 Create a Database 89
A.2.2 Update `build.sbt` Dependencies 90
A.2.3 Update JDBC References 90
A.2.4 Update Slick Driver 90
A.3 MySQL 90
A.3.1 Create a Database 90
A.3.2 Update `build.sbt` Dependencies 91
A.3.3 Update JDBC References 91
A.3.4 Update Slick Driver 91

B Solutions to Exercises 93
B.1 Basics 93
B.1.1 Solution to: Bring Your Own Data 93
B.1.2 Solution to: Bring Your Own Data Part 2 93
B.2 Selecting Data 95
B.2.1 Solution to: Count the Messages 95

B.2.2 Solution to: Selecting a Message . . . 95
B.2.3 Solution to: One Liners . . . 95
B.2.4 Solution to: Checking the SQL . . . 96
B.2.5 Solution to: Selecting Columns . . . 96
B.2.6 Solution to: First Result . . . 96
B.2.7 Solution to: The Start of Something . . . 96
B.2.8 Solution to: Liking . . . 97
B.2.9 Solution to: Client-Side or Server-Side? . . . 97
B.3 Creating and Modifying Data . . . 98
B.3.1 Solution to: Insert New Messages Only . . . 98
B.3.2 Solution to: Rollback . . . 98
B.3.3 Solution to: Update Using a For Comprehension . . . 98
B.3.4 Solution to: Delete All The Messages . . . 98
B.4 Data Modelling . . . 99
B.4.1 Solution to: Turning Many Rows into Case Classes . . . 99
B.4.2 Solution to: Filtering Optional Columns . . . 100
B.4.3 Solution to: Inside the Option . . . 100
B.4.4 Solution to: Matching or Undecided . . . 100
B.4.5 Solution to: Enforcement . . . 100
B.4.6 Solution to: Model This . . . 100
B.4.7 Solution to: Mapping Enumerations . . . 101
B.4.8 Solution to: Alternative Enumerations . . . 102
B.4.9 Solution to: Custom Boolean . . . 102
B.5 Joins and Aggregates . . . 103
B.5.1 Solution to: Many Messages . . . 103
B.5.2 Solution to: User Rooms . . . 103
B.6 Plain SQL . . . 103
B.6.1 Solution to: Robert Tables . . . 103
B.6.2 Solution to: String Interpolation Mistake . . . 104
B.6.3 Solution to: Unsafe Composition . . . 104

Preface

Essential Slick is aimed at beginner-to-intermediate Scala developers who want to get started using Slick.

Slick is a Scala library for working with databases: querying, inserting data, updating data, and representing a schema. Queries are written in Scala and type checked by the compiler. Slick aims to make working with a database similar to working with regular Scala collections.

This material is aimed at a Scala developer who has:

- 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 7, 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 2.1.0. Examples use H2 as the relational database.

Many thanks to Dave Gurnell, and the team at Underscore for their invaluable contributions and proof reading.

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.

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 look more like the prepared statements you may be used to from JDBC.

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 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 explores advanced select queries, including joins and aggregates.
- Chapter 6 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$ git checkout 2.1

bash$ cd essential-slick-code

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

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, *Slack*, *Flowdock*, or an *IRC* application. 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 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 along-side our application code. We've picked H2 because it allows us to forego any system administration and skip to writing Scala code.

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.

1.3.1 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"
```

```
scalaVersion := "2.11.6"

libraryDependencies ++= Seq(
  "com.typesafe.slick" %% "slick"           % "2.1.0",
  "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. We may also bring in a connection pooling library such as C3P0 or DBCP. Slick is based on JDBC under the hood, so many of the same low-level configuration options exist.

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

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 4 looks at working with different drivers.

1.3.3 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 4. For now, all you 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` 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.

1.3.4 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.

Slick provides other querying models, but lifted embedding is the standard, non-experimental, way to work with Slick. We will discuss another type of approach, called *Plain SQL querying*, in Chapter 6.

1.3.5 Connecting to 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 opening connections and starting transactions:

```
def db = Database.forURL(
  url    = "jdbc:h2:mem:chat-database;DB_CLOSE_DELAY=-1",
  driver = "org.h2.Driver")
```

The `Database.forURL` method is part of Slick, but the parameters we're providing are intended to configure the underlying JDBC layer. The `url` parameter is the standard JDBC connection URL, and the `driver` parameter is the fully qualified class name of the JDBC driver for our chosen DBMS. In this case we're creating an in-memory database called `"chat-database"` and configuring H2 to keep the data around indefinitely when no connections are open. H2-specific JDBC URLs are discussed in detail in the H2 documentation.

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).

We can use the `db` object to open a `Session` with our database, which wraps a JDBC-level `Connection` and provides a context in which we can execute a sequence of queries.

```
db.withSession { implicit session =>
  // Run queries, profit!
}
```

The `session` object provides methods for starting, committing, and rolling back transactions (see Chapter 3), and is passed an implicit parameter to methods that actually run queries against the database.

1.3.6 Inserting Data

Having opened a session, we can start sending SQL to the database. We start by issuing a `CREATE` statement for `MessageTable`, which we build using methods of our `TableQuery` object, `messages`:

```
messages.ddl.create
```

"DDL" in this case stands for *Data Definition Language*—the standard part of SQL used to create and modify the database schema. The Scala code above issues the following SQL to H2:

```
messages.ddl.createStatements.toList
// res0: List[String] = List("""
//   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
//   )
// """)
```

Once our table is set up, we need to insert some test data:

```
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]`). Our table is now populated with data.

1.3.7 Selecting Data

Now our database is populated, we can start running queries to select it. We do this by invoking one of a number of "invoker" methods on a query object. For example, the `run` method executes the query and returns a Seq of results:

```
messages.run
// 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))
```

We can see the SQL issued to H2 using the `selectStatement` method on the query:

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

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 an extra `WHERE` statement in the SQL that retrieves the expected subset of results:

```
scala> messages.filter(_.sender === "HAL").selectStatement
// res3: String = select x2."sender", x2."content", x2."id" ↩
//                from "message" x2 ↩
//                where x2."sender" = 'HAL'
```

```
scala> messages.filter(_.sender === "HAL").run
// 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))
```

If you remember, 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:

```
scala> halSays.run
// 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))
```

The observant among you will remember that we created `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 `run`:

```
halSays.map(_.id).selectStatement
// res6: String = select x2."id" ↩
//                from "message" x2 ↩
//                where (x2."sender" = 'HAL')

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

1.3.8 For Comprehensions

Queries implement methods called `map`, `flatMap`, `filter`, and `withFilter`, making them compatible with Scala for comprehensions. 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:

```
halSays2.run
// res8: Seq[Message] = ...
```

1.4 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 the operate on the parameters of the query rather than the results returned. We execute a query by opening a session with the database and calling an *invoker* method such as run.

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.

1.5 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 connect to the database using db.withSession and 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 connect to the database again using db.withSession, build the appropriate query using messages.filter, and execute it using its run method. 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, connect to a database, and run those queries. 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 5 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`:

```scala
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: scala.slick.lifted.TableQuery[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 the `selectStatement` method:

```
messages.selectStatement
// res11: String = 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 "query invoker" methods discussed below, the `selectStatement` method is actually an extension method applied to `Query` via an implicit conversion. You'll need to have everything from `H2Driver.simple` in scope for this to work:

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

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")
// res14: scala.slick.lifted.Query[
//   MessageTable,
//   MessageTable#TableElementType,
//   Seq
// ] = scala.slick.lifted.WrappingQuery@1b4b6544
```

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

```
messages.filter(_.sender === "HAL").selectStatement
// res15: String = select ... where x2."sender" = 'HAL'
```

Slick uses the `Column` type to represent expressions over columns as well as individual columns. A `Column[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 `Column[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. Queries have access to the type of the Table used to create them, which allows 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)
//                                        ^
```

2.4 Transforming Results: The *map* Method

Sometimes we don't want to select all of the data 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: scala.slick.lifted.Query[
//   scala.slick.lifted.Column[String],
//   String,
//   Seq
// ] = scala.slick.lifted.WrappingQuery@407beadd
```

Because the unpacked type 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:

```
messages.map(_.content).run
// res2: 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'?)
```

Also notice that the generated SQL has changed. The revised query isn't just selecting a single column from the query results—it is actually telling the database to restrict the results to that column in the SQL:

```
messages.map(_.sender).selectStatement
// res3: String = select x2."content" from "message" x2
```

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

```
val seekBeauty = messages.
  map(_.content).
  filter(content: Column[String] => content like "%Pretty%")
// seekBeauty: scala.slick.lifted.Query[
//   scala.slick.lifted.Column[String],
//   String,
//   Seq
// ] = scala.slick.lifted.WrappingQuery@6cc2be89

seekBeauty.run
// res4: 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 `Columns` and `Tables`, as well as `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))
// res5: scala.slick.lifted.Query[
//   (Column[Long], Column[String]),
//   (Long, String),
//   Seq
// ] = scala.slick.lifted.WrappingQuery@2a1117d3
```

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

```
messages.map(t => (t.id, t.content)).selectStatement
// res6: String = select x2."id", x2."content" ...
```

We can also select column expressions as well as single `Columns`:

```
messages.map(t => t.id * 1000L).selectStatement
// res7: String = select x2."id" * 1000 ...
```

2.5 Query Invokers

Once we've built a query, we can run it by establishing a session with the database and using one of several *query invoker* methods. We've seen one invoker—the `run` method—already. Slick has several invoker methods, each of which is added to `Query` as an extension method, and each of which accepts an implicit `Session` parameter that determines which database to use.

If we want to return a sequence of the results of a query, we can use the `run` or `list` invokers. `list` always returns a `List` of the query's unpacked type; `run` returns the query's collection type:

```
messages.run
// res0: Seq[Example.MessageTable#TableElementType] = Vector(
//   Message(Dave,Hello, HAL. Do you read me, HAL?,1),
//   ...)

messages.list
// res1: List[Example.MessageTable#TableElementType] = List(
//   Message(Dave,Hello, HAL. Do you read me, HAL?,1),
//   ...)
```

If we only want to retrieve a single item from the results, we an use the `firstOption` invoker. Slick retrieves the first row and discards the rest of the results:

```
messages.firstOption
// res2: Option[Example.MessageTable#TableElementType] =
//   Some(Message(Dave,Hello, HAL. Do you read me, HAL?,1))

messages.filter(_.sender === "Nobody").firstOption
// res3: Option[Example.MessageTable#TableElementType] =
//   None
```

If we want to retrieve large numbers of records, we can use the `iterator` invoker to return an `Iterator` of results. We can extract results from the iterator one-at-a-time without consuming large amounts of memory:

```
messages.iterator.foreach(println)
// Message(Dave,Hello, HAL. Do you read me, HAL?,1)
// ...
```

Note that the `Iterator` can only retrieve results while the session is open:

```
db.withSession { implicit session =>
  messages.iterator
}.foreach(println)
// org.h2.jdbc.JdbcSQLException: ↩
//   The object is already closed [90007-185]
//   at ...
```

Finally, we can use the `execute` invoker to run a query and discard all of the results. This will come in useful in the next chapter when we cover insert, update, and delete queries.

Table 2.1: Common query invoker methods. Return types are specified for a query of type `Query[M, U, C]`.

Method	Return Type	Description
`run`	`C[U]`	Return a collection of results. The collection type is determined by the
`list`	`List[U]`	Run the query, return a `List` of results. Ignore the query's collection type.
`iterator`	`Iterator[U]`	Run the query, return an `Iterator` of results. Results must be retrieved from the iterator before the session is closed.
`firstOption`	`Option[U]`	Return the first result wrapped in an `Option`; return `None` if there are no results.
`execute`	`Unit`	Run the query, ignore the result. Useful for updating the database—see Chapter 3.

2.6 Column Expressions

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

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

2.6.1 Equality and Inequality Methods

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

```
messages.filter(_.sender === "Dave").selectStatement
// res3: String = select ... where x2."sender" = 'Dave'

messages.filter(_.sender =!= "Dave").selectStatement
// res4: String = select ... where not (x2."sender" = 'Dave')
```

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

```
messages.filter(_.sender < "HAL").selectStatement
// res7: String = select ... where x2."sender" < 'HAL'

messages.filter(m => m.sender >= m.content).selectStatement
// res8: String = select ... where x2."sender" >= x2."content"
```

Table 2.2: Column comparison methods. Operand and result types should be interpreted as parameters to `Column[_]`.

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`

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`

2.6.2 String Methods

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

```scala
messages.filter(m => m.sender ++ "> " + m.content).selectStatement
// res9: String = select x2."sender" || '> ' || x2."content" ...
```

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

```scala
messages.filter(_.content like "%Pretty%").selectStatement
// res10: String = ... 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 2.3: String column methods. Operand and result types should be interpreted as parameters to `Column[_]`.

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)`

2.6.3 Numeric Methods

Slick provides a comprehensive set of methods that operate on `Columns` with numeric values: `Ints`, `Longs`, `Doubles`, `Floats`, `Shorts`, `Bytes`, and `BigDecimals`.

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

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)`
`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)`

2.6.4 Boolean Methods

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

Table 2.5: Boolean column methods. Operand and result types should be interpreted as parameters to `Column[_]`.

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`

2.6.5 Option Methods and Type Equivalence

Slick models nullable columns in SQL as `Columns` with `Option` types. We'll discuss this in some depth in Chapter 4. However, as a preview, know that if we have a nullable column in our database, we declare it as optional in our `Table`:

```scala
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 `Strings` for equality but we can't compare a `String` and an `Int`:

```scala
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").selectStatement
//                                    ^
```

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).selectStatement
//                                      ^
```

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)).selectStatement
// res16: String = select ... where x2."id" = 123
```

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

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

2.7 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 2.6: 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 term, starting with an example of `sortBy`:

```
messages.sortBy(_.sender).run
// res17: 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)).selectStatement
// res18: String =
//  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
```

2.8 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 `Column`. Comparisons to `Option` types are made easy for us as Slick will compare `Column[T]` and `Column[Option[T]]` automatically.

Finally, 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 `Column[String]`.

2.9 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.

2.9.1 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

2.9.2 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

2.9.3 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

2.9.3.1 Checking the SQL

Calling the `selectStatement` method 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

2.9.4 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

2.9.5 First Result

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

See the solution

2.9.6 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

2.9.7 Liking

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

See the solution

2.9.8 Client-Side or Server-Side?

What does this do and why?

```scala
messages.map(_.content + "!").list
```

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 modifying stored data using insert, update, and delete queries.

SQL veterans will know that update and delete queries, in particular, 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.

3.1 Inserting Data

As we saw in Chapter 1, adding new data to a table looks like a destructive 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, which is an alias for `insert`:

```
messages += Message("HAL", "No. Seriously, Dave, I can't let you in.")
// res2: Int = 1

messages insert Message("Dave", "You're off my Christmas card list.")
// res3: Int = 1
```

In each case the return value 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, or the entire row as a case class. As we will see below, we can get this information using a new method called `returning`.

3.1.2 Primary Key Allocation

If we recall the definition of `Message`, we put the `id` field at the end of the case class and gave it a default value of `0L`:

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

Giving the `id` parameter a default value allows us to omit it when creating a new object. Placing the `id` at the end of the constructor allows us to omit it without having to pass the remaining arguments using keyword parameters:

```
Message("HAL",
        "I'm a computer, Dave, what would I do with a Christmas card anyway?")
```

There's nothing special about our default value of `0L`—it's not a magic value meaning "this record has no `id`". In our running example the `id` field of `Message` is mapped to an auto-incrementing primary key (using the `O.AutoInc` option), causing Slick to ignore the value of the field when generating an insert query and allows the database to step in an generate the value for us. We can see the SQL we're executing using the `insert-Statement` method:

```
messages.insertStatement
// res4: String =
//   insert into "message" ("sender","content")
//   values (?,?)
```

Slick provides a `forceInsert` method that allows us to specify a primary key on insert, ignoring the database's suggestion:

```
messages forceInsert Message("Dave", "Point taken.", 1000)
// res5: Int = 1

messages.filter(_.id === 1000L).run
// res6: Seq[Example.MessageTable#TableElementType] =
//   Vector(Message(Dave,Point taken.,1000))
```

3.1.3 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
// res7: String =
//   insert into "message" ("sender")
//   values (?)
```

The parameter type of the += method is matched to the *unpacked* type of the query, so we execute this query by passing it a `String` for the `sender`:

```
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 4.

3.1.4 Retrieving Primary Keys on Insert

Let's modify the insert to give us back the primary key generated:

```
(messages returning messages.map(_.id)) +=
  Message("Dave", "So... what do we do now?")
// res5: Long = 7
```

The argument to `messages returning` is a `Query`, which is why `messages.map(_.id)` makes sense here. We can show that the return value is a primary key by looking up the record we just inserted:

```
messages.filter(_.id === 7L).firstOption
// res6: Option[Example.MessageTable#TableElementType] =
//   Some(Message(Dave,So... what do we do now?,7))
```

H2 only allows us to retrieve the primary key from an insert. Some databases allow us to retrieve the complete inserted record. For example, we could ask for the whole `Message` back:

```
(messages returning messages) +=
  Message("HAL", "I don't know. I guess we wait.")
// res7: Message = ...
```

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

```
(messages returning messages) +=
  Message("HAL", "I don't know. I guess we wait.")
// scala.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. Typing `messages returning messages.map(_.id)` isn't exactly convenient, but we can easily define a query specifically for inserts:

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

messagesInsert += Message("Dave", "You're such a jerk.")
// res8: Long = 8
```

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 we do want to get a populated Message back from an insert for any database, we can do it by retrieving the primary key and manually adding it to the inserted record. Slick simplifies this with another method, into:

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

messagesInsertWithId += Message("Dave", "You're such a jerk.")
// res8: messagesInsertWithId.SingleInsertResult =
//   Message(Dave,You're such a jerk.,8)
```

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 Multiple Rows

Suppose we want to insert several Messages into the database. 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] = ...

messages ++= testMessages
// res9: Option[Int] = Some(4)
```

This code prepares one SQL statement and uses it for each row in the Seq. 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 messagesInsertWithId query we defined last section and write:

```
messagesInsertWithId ++= testMessages
// res9: messagesInsertWithId.MultiInsertResult = List(
//   Message(Dave,Hello, HAL. Do you read me, HAL?,13),
//   ...)
```

3.2 Updating Rows

So far we've only looked at inserting new data into the database, but what if we want to update records that are already in the database? Slick lets us create SQL `UPDATE` queries using the same `Query` objects we saw in Chapter 2.

3.2.1 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:

```
messages.filter(_.sender === "HAL").
  map(_.sender).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` `Table-Query`, 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
// ] = scala.slick.lifted.WrappingQuery@537c3243
```

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: scala.slick.lifted.Query[
//   scala.slick.lifted.Column[String],
//   String,
//   Seq
// ] = scala.slick.lifted.WrappingQuery@509f9e50
```

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 rowsAffected = halSenderCol.update("HAL 9000")
// rowsAffected: Int = 4
```

3.2.2 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:

```
messages.
  filter(_.id === 4L).
  map(message => (message.sender, message.content)).
  update("HAL 9000", "Sure, Dave. Come right in.")
// res3: Int = 1

messages.filter(_.sender === "HAL 9000").run
// 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
```

3.2.3 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 6. 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: Message => Message = <function1>
```

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:

```
messages.iterator.foreach { message =>
  messages.filter(_.id === message.id).update(exclaim(message))
}
```

We recommend plain SQL queries over this approach if you can use them. See Chapter 6 for details.

3.3 Deleting Rows

Deleting rows is very similar to updating them. We specify which rows to delete using the `filter` method and call `delete`:

```
messages.filter(_.sender === "HAL").delete
// res6: Int = 2
```

As usual, the return value is the number of rows affected, and as usual, Slick provides a method that allows us to view the generated SQL:

```
messages.filter(_.sender === "HAL").deleteStatement
// res7: String =
//   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 ↩
//   scala.slick.lifted.Query[scala.slick.lifted.Column[String],String,Seq]
//                 messages.map(_.content).delete
//                                         ^
```

3.4 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 `session.withTransaction` method:

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

db.withSession { implicit session =>
  session.withTransaction {
    updateContent(2L).update("Wanna come in?")
    updateContent(3L).update("Pretty please!")
    updateContent(4L).update("Opening now.")
  }

  messages.run
}
// 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 `withTransaction` block are temporary until the block completes, at which point they are *committed* and become permanent. We can alternatively *roll back* the transaction mid-stream by calling `session.rollback`, which causes all changes to be reverted:

```
db.withSession { implicit session =>
  session.withTransaction {
    updateContent(2L).update("Wanna come in?")
    updateContent(3L).update("Pretty please!")
    updateContent(4L).update("Opening now.")
    session.rollback
  }

  messages.run
}
// 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 that the rollback doesn't happen until the `withTransaction` block ends. If we run queries *within* the block, before the rollback actually occurs, they will still see the modified state:

```
db.withSession { implicit session =>
  session.withTransaction {
    session.rollback
    updateContent(2L).update("Wanna come in?")
    updateContent(3L).update("Pretty please!")
    updateContent(4L).update("Opening now.")
    messages.run
  }
}
// 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))
```

3.5 Logging Queries and Results

We've seen how to retrieve the SQL of a query using the `selectStatement`, `insertStatement`, `updateS–tatement`, and `deleteStatement` queries. These are useful for exprimenting with Slick, but sometimes we want to see all the queries, fully populated with parameter data, *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 SBT builds 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="scala.slick.jdbc.JdbcBackend.statement" level="DEBUG"/>
```

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

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

We can modify the level of various loggers, as shown in table 3.1.

Table 3.1: Slick loggers and their effects.

Logger	Effect
`scala.slick.jdbc.JdbcBackend.statement`	Logs SQL sent to the database as described above.
`scala.slick.jdbc.StatementInvoker.result`	Logs the results of each query.
`scala.slick.session`	Logs session events such as opening/closing connections.
`scala.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              | ts                   | id |
SI.result - +--------+----------------------+----------------------+----+
SI.result - | HAL    | Affirmative, Dave... | 2001-02-17 10:22:... | 2  |
SI.result - | HAL    | I'm sorry, Dave. ... | 2001-02-17 10:22:... | 4  |
SI.result - \--------+----------------------+----------------------+----/
```

3.6 Take Home Points

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

- inserts are via an `insert` (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.

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

3.7 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.

3.7.1 Insert New Messages Only

Messages sent to our application might fail, and might be resent to us. Write a method that will insert a message for someone, but only if the message content hasn't already been stored. We want the `id` of the message as a result.

The signature of the method is:

```scala
def insertOnce(sender: String, message: String): Long = ???
```

See the solution

3.7.2 Rollback

Assuming you already have an `implicit session`, what is the state of the database after this code is run?

```scala
session.withTransaction {
  messages.delete
  session.rollback()
  messages.delete
  println("Surprised?")
}
```

Is "Surprised?" printed?

See the solution

3.7.3 Update Using a For Comprehension

Rewrite the update statement below to use a for comprehension.

```scala
val rowsAffected = messages.
                    filter(_.sender === "HAL").
                    map(msg => (msg.sender, msg.ts)).
                    update("HAL 9000", DateTime.now)
```

Which style do you prefer?

See the solution

3.7.4 Delete All The Messages

How would you delete all messages?

See the solution

Chapter 4

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;
- expand on our knowledge of modelling tables to introduce optional values and foreign keys; and
- use custom types to avoid working with just low-level database values.

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

4.1 Application Structure

Our examples so far have been stand-alone application. That's not how you'd work with Slick for a more substantial project. We'll explain how to split up an application in this section.

We've also been importing the H2 driver. We need a driver of course, but it's useful to delay picking the driver until the code needs to be run. This will allow us to switch driver, which can be useful for testing. For example, you might use H2 for unit testing, but PostgresSQL for production.

4.1.1 Pattern Outline

The basic pattern we'll use is as follows:

- Isolate our schema into a trait (or a few traits) in which the Slick *profile* is abstract. We will often refer to this trait as "the tables".

- Create an instance of our tables using a specific profile.

- Finally, configure a `Database` to obtain a `Session` to work with the table instance.

Profile is a new term for us. When we have previously written…

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

…that gave us an H2-specific JDBC driver. That's a `JdbcProfile`, which in turn is a `RelationalProfile` provided by Slick. It means that Slick could, in principle, be used with non-JDBC-based, or indeed non-relational, databases. In other words, *profile* is an abstraction above a specific driver.

4.1.2 Working with a Profile

Re-working the example from previous chapters, we have the schema in a trait:

```scala
trait Profile {
  // Place holder for a specific profile
  val profile: scala.slick.driver.JdbcProfile
}

trait Tables {
  // Self-type indicating that our tables must be mixed in with a Profile
  this: Profile =>

  // Whatever that Profile is, we import it as normal:
  import profile.simple._

  // Row and table definitions here as normal
}
```

We currently have a small schema and can get away with putting all the table definitions into a single trait. However, there's nothing to stop us from splitting the schema into, say `UserTables` and `MessageTables`, and so on. They can all be brought together with `extends` and `with`:

```scala
// Bring all the components together:
class Schema(val profile: JdbcProfile) extends Tables with Profile

object Main extends App {

  // A specific schema with a particular driver:
  val schema = new Schema(scala.slick.driver.H2Driver)

  // Use the schema:
  import schema._, profile.simple._

  def db = Database.forURL("jdbc:h2:mem:chapter03", driver="org.h2.Driver")

  db.withSession {
    // Work with the database as normal here
  }
}
```

To work with a different database, create a different `Schema` instance and supply a different driver. The rest of the code does not need to change.

4.1.3 Additional Considerations

There is a potential down-side of packaging everything into a single `Schema` and performing `import schema._`. All your case classes, and table queries, custom methods, implicits, and other values are imported into your current namespace.

If you recognise this as a problem, it's time to split your code more finely and take care over importing just what you need.

4.2 Representations for Rows

In previous chapters we modelled rows as case classes. That's a great choice, and the one we recommend, but you should be aware that Slick is more flexible that that.

There are in fact three common representations used: tuples, case classes, and an experimental `HList` implementation.

4.2.1 Case Classes and <>

To explore these different representations we'll start with comparing tuples and case classes. For a little bit of variety, let's define a `user` table so we no longer have to store names in the `message` table.

A user will have an ID and a name. The row representation will be:

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

The schema is:

```scala
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)
}
```

None of this should be a surprise, as it is essentially what we have seen in the first chapter. What we'll do now is look a little bit deeper into how rows are mapped into case classes.

`Table[T]` class requires the ∗ method to be defined. This *projection* is of type `ProvenShape[T]`. A "shape" is a description of how the data in the row is to be structured. Is it to be a case class? A tuple? A combination of these? Something else? Slick provides implicit conversions from various data types into a "shape", allowing it to be sure at compile time that what you have asked for in the projection matches the schema defined.

To explain this, let's work through an example.

If we had simply tried to define the projection as a tuple…

```scala
def * = (name, id)
```

…the compiler would tell us:

```
Type mismatch
 found: (scala.slick.lifted.Column[String], scala.slick.lifted.Column[Long])
 required: scala.slick.lifted.ProvenShape[chapter03.User]
```

This is good. We've defined the table as a `Table[User]` so we want `User` values, and the compiler has spotted that we've not defined a default projection to supply `User`s.

How do we resolve this? The answer here is to give Slick the rules it needs to prove it can convert from the `Column` values into the shape we want, which is a case class. This is the role of the mapping function, <>.

The two arguments to <> are:

- a function from `U => R`, which converts from our unpacked row-level encoding into our preferred representation; and
- a function from `R => Option[U]`, which is going the other way.

We can supply these functions by hand if we want:

```scala
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:

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

Case classes already supply these functions via `User.tupled` and `User.unapply`, so there's no point doing this. However it is useful to know, and comes in handy for more elaborate packaging and unpackaging of rows. We will see this in one of the exercises in this section.

4.2.2 Tuples

You've seen how Slick is able to map between a tuple of columns into case classes. However you can use tuples directly if you want, because Slick already knows how to convert from a `Column[T]` into a T for a variety of Ts.

Let's return to the compile error we had above:

```
Type mismatch
 found: (scala.slick.lifted.Column[String], scala.slick.lifted.Column[Long])
 required: scala.slick.lifted.ProvenShape[chapter03.User]
```

We fixed this by supplying a mapping to our case class. We could have fixed this error by redefining the table in terms of a tuple:

```scala
type TupleUser = (String, Long)

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

As you can see there is little difference between the case class and the tuple implementations.

Unless you have a special reason for using tuples, or perhaps just a table with a single value, you'll probably use case classes for the advantages they give:

- we have a simple type to pass around (`User` vs. `(String, Long)`); and
- the fields have names, which improves readability.

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 it is driven by our row definition. You don't need to project all the rows, for example, when working with a table with legacy columns that aren't being used.

4.2.3 Heterogeneous Lists

Slick's **experimental** `HList` implementation is useful if you need to support tables with more than 22 columns, such as a legacy database.

To motivate this, let's suppose our user table contains lots of columns for all sorts of information about a user:

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

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 age          = column[Int]("age")
  def gender       = column[Char]("gender")
  def height       = column[Float]("height_m")
  def weight       = column[Float]("weight_kg")
  def shoeSize     = column[Int]("shoe_size")
  def email        = column[String]("email_address")
  def phone        = column[String]("phone_number")
  def accepted     = column[Boolean]("terms")
  def sendNews     = column[Boolean]("newsletter")
  def street       = column[String]("street")
  def city         = column[String]("city")
  def country      = column[String]("country")
  def faveColor    = column[String]("fave_color")
  def faveFood     = column[String]("fave_food")
  def faveDrink    = column[String]("fave_drink")
  def faveTvShow   = column[String]("fave_show")
  def faveMovie    = column[String]("fave_movie")
  def faveSong     = column[String]("fave_song")
  def lastPurchase = column[String]("sku")
  def lastRating   = column[Int]("service_rating")
  def tellFriends  = column[Boolean]("recommend")
  def petName      = column[String]("pet")
  def partnerName  = column[String]("partner")

  def * = name :: age :: gender :: height :: weight :: shoeSize ::
        email :: phone :: accepted :: sendNews ::
        street :: city :: country ::
        faveColor :: faveFood :: faveDrink ::
        faveTvShow :: faveMovie :: faveSong ::
        lastPurchase :: lastRating :: tellFriends ::
        petName :: partnerName :: id :: HNil
}
```

I hope you don't have a table that looks like this, but it does happen.

You could try to model this with a case class. Scala 2.11 supports case classes with more than 22 arguments, but it does not implement the `unapply` method we'd want to use for mapping. Instead, in this situation, we're using a *hetrogenous list*.

An HList has a mix of the properties of a list and a tuple. It has an arbitrary length, just as a list, but unlike a list, each element can be a different type, like a tuple. As you can see from the ∗ definition, an `Hlist` is a kind of shape that Slick knows about.

This `HList` projection needs to match with the definition of `User` in `Table[User]`. For that, we list the types in a type alias:

```
type User =
  String :: Int :: Char :: Float :: Float :: Int ::
  String :: String :: Boolean :: Boolean ::
  String :: String :: String :: String :: String :: String ::
  String :: String :: String ::
  String :: Int :: Boolean ::
  String :: String  :: Long :: HNil
```

Typing this in by hand is error prone and likely to drive you crazy. There are two ways to improve on this:

- The first is to know that Slick can generate this code for you from an existing database. We expect you'd be needing `HLists` for legacy database structures, and in that case code generate is the way to go.
- Second, you can improve the readability of `User` by *value clases* to replace `String` with a more meaningful type. 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 the truth, 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.

Once you have an `HList`-based schema, you work with it in much the same way as you would other data representations. To create an instance of an `HList` we use the cons operator and `HNil`:

```
users +=
  "Dr. Dave Bowman" :: 43 :: 'M' :: 1.7f :: 74.2f :: 11 ::
  "dave@example.org" :: "+1555740122" :: true :: true ::
  "123 Some Street" :: "Any Town" :: "USA" ::
  "Black" :: "Ice Cream" :: "Coffee" :: "Sky at Night" :: "Silent Running" ::
  "Bicycle made for Two" :: "Acme Space Helmet" :: 10 :: true ::
  "HAL" :: "Betty" :: 0L :: HNil
```

A query will produce an `HList` based `User` instance. To pull out fields you can use `head`, `apply`, `drop`, `fold`, and the appropriate types from the `Hlist` will be preserved:

```scala
val dave = users.first

val name: String = dave.head
val age: Int = dave.apply(1)
```

However, accessing the `HList` by index is dangerous. If you run off the end of the list with `dave(99)`, you'll get a run-time exception.

The `HList` representation probably won't be the one you choose to use; but you need to know it's there for you when dealing with nasty schemas.

Warning

Extra Dependencies

Some parts of `HList` depend on Scala reflection. Modify your *build.sbt* to include:

```scala
"org.scala-lang" % "scala-reflect" % scalaVersion.value
```

4.2.4 Exercises

4.2.4.1 Turning Many Rows into 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:

```scala
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 folder *chapter-04*.

See the solution

4.3 Table and Column Representation

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

4.3.1 Nullable Columns

Columns defined in SQL are nullable by default. That is, they can contain `NULL` as a value. Slick makes columns not nullable by default, and if you want a nullable column you model it naturally in Scala as an `Option[T]`.

Let's modify `User` to have an optional email address:

```scala
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:

```scala
val daveId: Long = insertUser += User("Dave", Some("dave@example.org"))
val halId:  Long = insertUser += User("HAL")
```

Selecting those rows out produces:

```
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:

```scala
// Don't do this
val none: Option[String] = None
users.filter(_.email === none).list
```

We have one row in the database without an email address, but the query will produce 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 `true`, whereas the expression `'Dave' = null` evaluates to `null`.

The Slick query amounts to:

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

Null comparison is a classic source of errors for inexperienced SQL developers. No value is actually equal to `null`—the equality check evaluates to `null`. 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 `Column[Option[A]]`:

Table 4.1: Optional column methods. Operand and result types should be interpreted as parameters to `Column[_]`.

Scala Code	Operand Column Types	Result Type	SQL Equivalent
`col1.?`	`A`	`A`	`col1`
`col1.isEmpty`	`Option[A]`	`Boolean`	`col1 is null`
`col1.isDefined`	`Option[A]`	`Boolean`	`col1 is not null`

We fix our query with isEmpty:

```
users.filter(_.email.isEmpty).list
// result: List(User(HAL,None,2))
```

That rounds off what you need to know to model optional columns with Slick. However, we will meet the subject again when dealing with joins in the next chapter, and in a moment when looking at a variation of primary keys.

4.3.2 Primary Keys

We've defined primary keys by using the class O which provides column options:

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

We combine this with a class class that has a default ID, knowing that Slick won't insert this value because the field is marked as auto incrementing:

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

That's the style that suits us in this book, but you should be aware of alternatives. You may find our `id: Long = 0L` default a bit arbitrary. Perhaps you'd prefer to model the primary key as an `Option[Long]`. It will be None until it is assigned, and then `Some[Long]`.

We can do that, but we need to know how to convert our non-null primary key into an optional value. This is handled by the ? method on the column:

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

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

The change is small:

- the row class, `User`, defines id as `Option[Long]`; and
- the projection has to convert the database non-null id into an `Option`, via ?.

To see why we need this, imagine what would happen if we omitted the call to `id.?`. The projection from (Long, `String`, `Option[String]`) would not match the `Table[User]` definition, which requires `Option[Long]` in the first position. In fact, Slick would report:

```
No matching Shape found. Slick does not know how to map the given types.
```

Given what we know about Slick so far, this is a pretty helpful message.

4.3.3 Compound Primary Keys

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

```scala
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 DDL will emit:

```sql
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.

Where `primaryKey` is more useful is when you have a compound key. This is a key which is based on the value of two or more columns.

We'll look at this by adding the ability for people to chat in rooms.

The room definition is straight-forward:

```scala
// 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)
```

Tip

Benefit of Case Classes

Now we have `room` and `user` the benefit of case classes over tuples becomes apparent. As tuples, both tables would have the same type signature: `(String,Long)`. It would get error prone passing around tuples like that.

To say who is in which room, we will add a table called `occupant`. And rather than have an auto-generated primary key for `occupant`, we'll make it a compound of the user and the room:

```scala
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]
```

The SQL generated for the `occupant` table is:

```sql
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:

```scala
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 you try to put Dave in the Air Lock twice, the database will complain that the key has been violated.

4.3.4 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 will step through this by using foreign keys to connect a `message` to a `user`. To do this we change the definition of `message` to reference an `id` of a `user`:

```
case class Message(senderId: Long,
                   content: String,
                   ts: DateTime,
                   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 ts       = column[DateTime]("ts")

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

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

The change here is that the column for the message sender is now a `Long`, when previously we just had a `String`. We have also defined a method, `sender`, which is the foreign key linking the `senderId` to a `user` `id`.

The `foreignKey` gives us two things.

First, in the DDL, if you use it, the appropriate constraint is added:

```
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 Slicks DDL command has been run for the table, or the SQL `ON DELETE CASCADE` action has been manually applied to the database, the following 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.

The second thing we get is a query which we can use in a join. We've dedicated the next chapter to looking at joins in detail, but to illustrate the foreign key usage, here's a simple join:

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

This is equivalent the the query:

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

...and will produce:

```
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.))
```

Notice that we modelled the `Message` row using a `Long sender`, rather than a `User`:

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

That's the design approach to take with Slick. The row model should reflect the row, not the row plus a bunch of other data from different tables. To pull data from across tables, use a query.

4.3.5 Table and Column Modifiers

We'll round off this section by looking at modifiers for columns and tables. These allow you to change default values or sizes for columns, and add indexes to a table.

We have already seen examples of these, namely `O.PrimaryKey` and `O.AutoInc`. Column options are defined in `ColumnOption`, and as you have seen are accessed via O.

We'll look at `Length`, DBTYPE, and `Default` now:

```
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)
}
```

We have modified name to fix the maximum length of the column, and give a default value.

`O.Default` gives a default value for rows being inserted. Remember it's the DDL commands from `users.ddl` that instruct the database to provide this default.

`O.Length` takes one parameter you'd expect, one one you might not expect:

- `Int` - maximum length of the column; and
- `Boolean` - `true` to use VARCHAR, `false` for a SQL CHAR.

You may or may not care if a `String` is represented as a VARCHAR or CHAR. If you're storing strings that are the same length, it can be more efficient to use CHAR. But check with the documentation for the relational database you're using.

On the `avatar` column we've used `O.DBType` to control the exact type used by the database. Again, the values you use here will depend on the database product in use.

Finally, we can add an index to the table:

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

The corresponding DDL statement will be:

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

4.3.6 Exercises

4.3.6.1 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:

```scala
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")).run` to produce one row, and `filterByEmail(None).run` to produce two rows.

See the solution

4.3.6.2 Inside the Option

Build on the last exercise to match rows that start with the supplied optional value. Recall that `Column[String]` defines `startsWith`.

So this time even `filterByEmail(Some("dave@")).run` will produce one row.

See the solution

4.3.6.3 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.

Add Elena to the database...

```scala
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.

See the solution

4.3.6.4 Enforcement

What happens if you try adding a message for a user ID of `3000`?

For example:

```scala
messages += Message(3000L, "Hello HAL!")
```

Note that there is no user in our example with an ID of 3000.

See the solution

4.3.6.5 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.ddl ++ users.ddl ++ bills.dll).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

4.4 Custom Column Mappings

We want to work with types that have meaning to our application. This means moving data from the simple types the database uses into something else. We've already seen one aspect of this where the column values for `id`, `sender`, `content`, and `ts` fields are mapped into a row representation of `Message`.

At a level down from that, we can also control how our types are converted into column values. For example, we'd like to use JodaTime's `DateTime` class for anything data and time related. Support for this is not built-in to Slick, but it's painless to map custom types to the database.

The mapping for JodaTime's `DateTime` is:

```
import java.sql.Timestamp
import org.joda.time.DateTime
import org.joda.time.DateTimeZone.UTC

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 turns it into a database-friendly value, namely a `java.sql.Timestamp`; and
- another that does the reverse, taking a database value and turning it into a `DataTime`.

Using the Slick `MappedColumnType.base` call enables this machinery, which is marked as `implicit` so the Scala compiler can invoke it when we mention a `DateTime`.

This is something we will emphasis and encourage you to use in your applications: work with meaningful types in your code, and let Slick take care of the mechanics of how those types are turned into database values.

4.5 Value Classes

In modelling rows we are using `Long`s as primary keys. Although that's a good choice for the database, it's not a great choice in our application. The problem with it is that we can make some silly mistakes:

```
// Users:
val halId:  Long = insertUser += User("HAL")
val daveId: Long = insertUser += User("Dave")

// Buggy lookup of a sender
val id = messages.filter(_.senderId === halId).map(_.id).first

val rubbish = messages.filter(_.senderId === id)
```

Do you see the problem here? We've looked up a *message* `id`, and then used it to search for a *user* (via `senderId`) with that `id`. It compiles, it runs, and produces nonsense. We can prevent these kinds of problems using Scala's type system.

Before showing how, here's another downside of using `Long` as a primary key. You may find yourself writing small helper methods such as:

```
def lookupByUserId(id: Long) = users.filter(_.id === id)
```

It would be much clearer to document this method using the types, rather than the method name:

```
def lookup(id: UserPK) = users.filter(_.id === id)
```

We can do that, and have the compiler help us matching up primary keys, by using value classes. A value class is a compile-time wrapper around a value. At run time, the wrapper goes away, leaving no allocation or performance overhead in our running code.

We define value classes like this:

```
object PKs {
  case class MessagePK(value: Long) extends AnyVal
  case class UserPK(value: Long) extends AnyVal
}
```

To be able to use them in our tables, we need to provide Slick with the conversion rules. This is just the same as we've previously added for JodaTime:

```
import PKs._
implicit val messagePKMapper = MappedColumnType.base[MessagePK, Long]  ↩
                                              (_.value, MessagePK(_))
implicit val userPKMapper    = MappedColumnType.base[UserPK, Long]  ↩
                                              (_.value, UserPK(_))
```

Recall that `MappedColumnType.base` is how we define the functions to convert between our classes (`MessagePK`, `UserPK`) and the database values (`Long`).

We *can* do that, but for such a mechanical piece of code, Slick provides a macro to take care of this for us. We only need to write…

```
object PKs {
  import scala.slick.lifted.MappedTo
  case class MessagePK(value: Long) extends AnyVal with MappedTo[Long]
  case class UserPK(value: Long) extends AnyVal with MappedTo[Long]
}
```

...and the `MappedTo` macro takes care of creating the `MappedColumnType.base` implicits for us.

With our value classes and implicits in place, we can now use them to give us type checking on our primary and therefore foreign keys:

```scala
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)
  def name = column[String]("name")
  def * = (name, id) <> (User.tupled, User.unapply)
}

case class Message(senderId: UserPK,
                   content: String,
                   ts: DateTime,
                   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 ts       = column[DateTime]("ts")
  def * = (senderId, content, ts, 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` is a `UserPK` and the message sender is also a `UserPK`.

Now, if we try our buggy query again, the compiler catches the problem:

```
Cannot perform option-mapped operation
     with type: (PKs.UserPK, PKs.MessagePK) => R
 for base type: (PKs.UserPK, PKs.UserPK) => Boolean
[error] val rubbish = messages.filter(_.senderId === id)
                                                 ^
```

The compiler is telling us it wanted to compare `UserPK` to another `UserPK`, but found a `UserPK` and a `MessagePK`.

Values classes are a reasonable way to make your code safer and more legible. The amount of code you need to write is small, however for a large database it can become dull writing many such methods. In that case, consider either generating the source code rather than writing it or by generalising our definition of a primary key, so we only need to define it once.

Tip

An `Id[T]` Class

Rather than providing a value class definition for each table...

```scala
final case class MessagePK(value: Long) extends AnyVal with MappedTo[Long]
```

...we can supply the table as a type parameter:

```scala
final case class PK[A](value: Long) extends AnyVal with MappedTo[Long]
```

We can then define primary keys in terms of PK[Table]:

```scala
case class User(id: Option[PK[UserTable]],
                name: String,
                email: Option[String] = None)

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 email = column[Option[String]]("email")

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

lazy val users = TableQuery[UserTable]
```

We now get type safety without having to define the boiler plate of individual primary key case classes per table. Depending on the nature of your application, that might be convenient for you.

4.5.1 Exercises

4.5.1.1 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:

```scala
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

4.5.1.2 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

4.6 Sum Types

We've used case classes extensively for modelling data. These are known as *product types*, which form one half of *algebraic data types* (ADTs). The other half is known as *sum types*, which we will look at now.

As an example we will add a flag to messages. Perhaps an administrator of the chat will be able to mark 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
```

How we store them in the database depends on the mapping. Maybe we want to store them as characters: `!`, X, and $:

```
implicit val flagType =
  MappedColumnType.base[Flag, Char](
    flag => flag match {
      case Important => '!'
      case Offensive => 'X'
      case Spam      => '$'
    },
    ch => ch match {
      case '!' => Important
      case 'X' => Offensive
      case '$' => Spam
    })
```

This is similar to the enumeration pattern from the last set of exercises. There is a difference, though, in that sealed traits can be checked by the compiler to ensure we have covered all the cases. That is, if we add a new flag (`OffTopic` perhaps), but forget to add it to our `Flag => Char` function, the compiler will warn us that we have missed a case. (By enabling the Scala compiler's `-Xfatal-warnings` option, these warnings will become errors, preventing your program from compiling).

Using `Flag` is the same as any other custom type:

```
case class Message(
  senderId: UserPK,
  content:  String,
  ts:       DateTime,
  flag:     Option[Flag] = None,
  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 flag     = column[Option[Flag]]("flag")
  def ts       = column[DateTime]("ts")

  def * = (senderId, content, ts, flag, id) <>  ↩
                                      (Message.tupled, Message.unapply)
  def sender =
    foreignKey("sender_fk", senderId, users)  ↩
                                (_.id, onDelete=ForeignKeyAction.Cascade)
}

lazy val messages = TableQuery[MessageTable]
```

And we can add a message with a flag set:

```
messages +=
  Message(halId, "Just kidding. LOL.", start plusSeconds 20, Some(Important))
```

When we execute a query, we can work in terms of our meaningful type. However, we need to give the compiler a little help:

```
messages.filter(_.flag === (Important : Flag)).run
```

Notice the _type ascription added to the `Important` value. If you find yourself writing that kind of query often, be aware that extension methods allow you to package code like this into `messages.isImportant` or similar.

4.6.1 Exercises

4.6.1.1 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

4.7 Take Home Points

Separate the specific profile (H2, Postgres...) from your schema definition if you need to be portable across databases. In this chapter we looked at a class called `Schema` that pulled together a profile with table definitions, which could then be imported into an application.

Rows can be represented in a variety of ways: case classes, tuples, and HLists, for example. You have control over how columns are mapped to a row representation, using <>.

Nullable columns are represented as `Option[T]` values, and the `?` operator lifts a non-null value into an optional value.

Foreign keys define a constraint, and allow you to link tables in a join.

Slick makes it relatively easy to abstract away from raw database types, such as `Long`, to meaningful types such as `UserPK`. This removes a class of errors in your application, where you could have passed the wrong `Long` key value around.

Slick's philosophy is to keep models simple. Model rows as rows, and don't try to include values from different tables.

Chapter 5

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 (implicit and explicit);
- different ways to join (inner, outer and zip); and
- aggregate functions and grouping.

5.1 Implicit Joins

The SQL standards recognises two styles of join: implicit and explicit. Implicit joins have been deprecated, but they're common enough to deserve a brief investigation.

We have seen an example of implicit joins in the last chapter:

```scala
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:

```scala
def sender = foreignKey("msg_sender_fk", senderId, users)(_.id)
```

Slick generates something like the following SQL:

```sql
select
  u."name", m."content"
from
  "message" m, "user" u
where
  u."id" = m."sender"
```

That's the implicit style of query, using foreign key relations.

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-05* folder, start SBT and at the SBT > prompt run:

```
runMain chapter05.JoinsExample
```

We can also rewrite the query to control the table relationships 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`.

5.2 Explicit Joins

An explicit join is where the join type is, unsurprisingly, explicitly defined. In SQL this is via the `JOIN` and `ON` keyword, which is mirrored in Slick as the `join` and `on` methods.

Slick offers the following methods to join two or more tables:

- `innerJoin` or `join` — an inner join
- `leftJoin` — a left outer join
- `rightJoin` — a right outer join
- `outerJoin` — a full outer join.

The above are convenience methods to `join` with an explicit `JoinType` parameter (which defaults to `innnerJoin`). As you can see, Slicks's explicit join syntax gives you more options for how to join tables.

As a quick taste of the syntax, we can join the `messages` table to the `users` on the `senderId`:

```
val q = messages innerJoin users on (_.senderId === _.id)
```

This will be a query of `(MessageTable, UserTable)`. If we wanted to, we could be more explicit about the values used in the on part:

```
val q = messages innerJoin users on ( (m: MessageTable, u: UserTable) =>  ↩
                                                            m.senderId === u.id)
```

...but it reads well without this.

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 5.1, which sketches the schema we're using in this chapter.

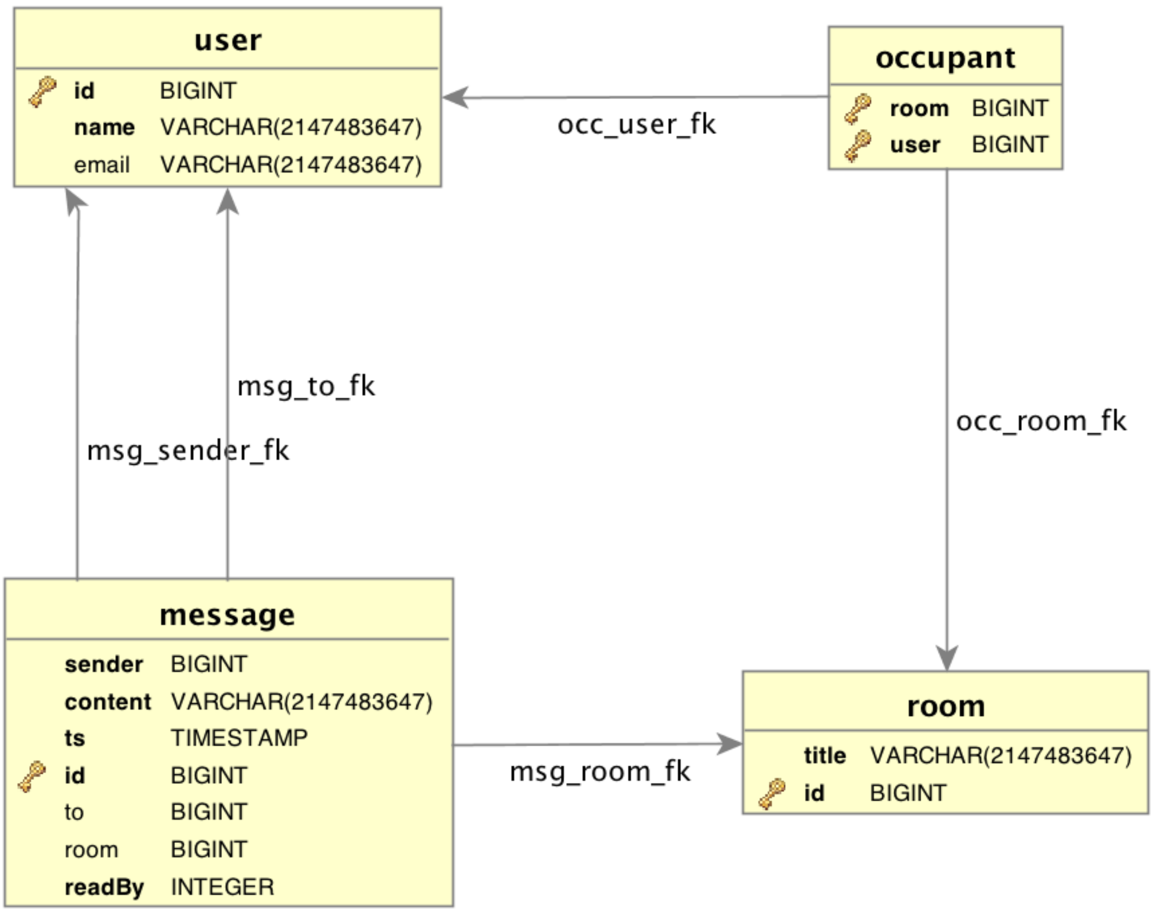

Figure 5.1: The database schema for this chapter. Find this code in the *chat-schema.scala* file of the example project on GitHub.

5.2.1 Inner Join

An inner join is where we select records from multiple tables, where those records exist (in some sense) in all tables. We'll lok at a chat example where we expect messages that have a sender in the user table, and a room in the rooms table:

```
val inner =
  messages.
  innerJoin(users).on(_.senderId === _.id).
  innerJoin(rooms).on{ case ((msg,user), room) => msg.roomId === room.id}

val query = for {
  ((msgs, usrs), rms) <- inner
  if usrs.id === daveId && rms.id === airLockId
} yield (msgs.content, usrs.name, rms.title)

val results = query.run
```

You might prefer to inline `inner` within the `query`. That's fine, but we've separated the parts out here to discuss them. And as queries in Slick compose, this works out nicely.

Let's start with the `inner` part. We're joining `messages` to `users`, and `messages` to `rooms`. We need two `join` (if you are joining *n* tables you'll need *n-1* join expressions). Notice that the second on method call is given a tuple of `(MessageTable,UserTable)` and `RoomTable`.

We're using a pattern match to make this explicit, and that's the style we prefer. However, you may see this more concisely expressed as:

```
val inner =
  messages.
  innerJoin(users).on(_.senderId  === _.id).
  innerJoin(rooms).on(_._1.roomId === _.id)
```

Either way, when it comes to the `query` itself we're using pattern matching again to unpick the results of `inner`, and adding additional guard conditions (which will be a `WHERE` clause in SQL).

Finally, we mapping to the columns we want: content, user name, and room title.

5.2.2 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*, and if not we will have `NULL` values in the query result.

For an example of from our chat schema, observe that messages can optionally be sent privately to another user. So let's say we want a list of all the messages and who they were sent to. Visually the left outer join is as shown in figure 5.2.

To implement this type of query we need to be aware of what columns are being returned, and if they can be `NULL` or not:

```
val left = messages.
  leftJoin(users).on(_.toId === _.id).
  map { case (m, u) => (m.content, u.name.?) }

left.run.foreach(result => println(result))
```

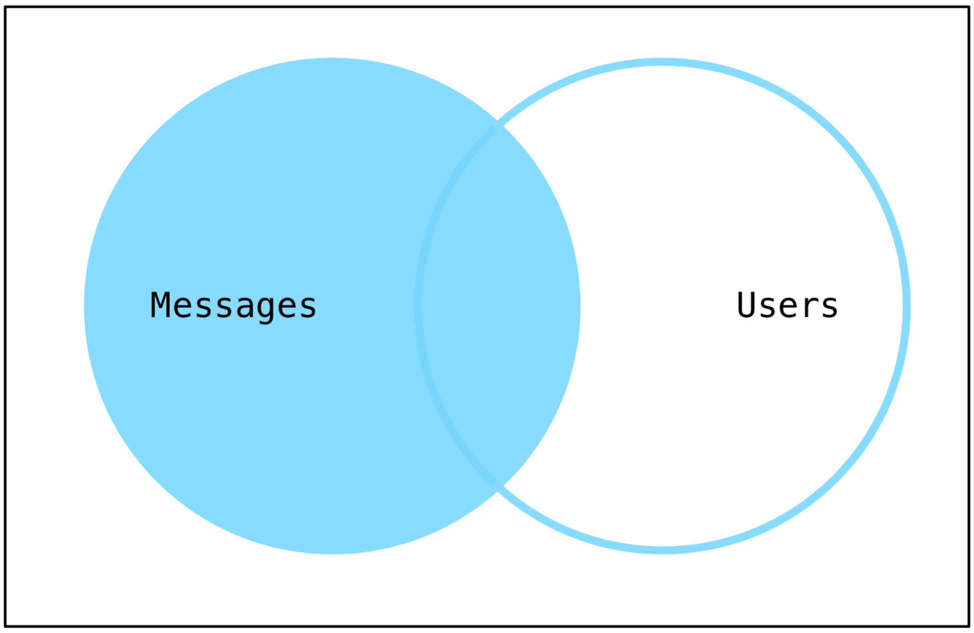

Figure 5.2: A visualization of the left outer join example. Selecting messages and associated recipients (users). For similar diagrams, see A Visual Explanation of SQL Joins, *Coding Horror*, 11 Oct 2007.

We're producing a list of messages and the name of user they were sent to (if any). Note the `u.name.?` expression is required to turn the potentially null result from the query into an `Option` value. You need to deal with this on a column-by-column basis.

The sample data we have in *joins.sql* in the *chapter05* folder contains just two private messages (between Frank and Dave). The rest are public. So our query results are:

```
(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))
```

Tip

NULLs in Joins

If you're thinking that detecting null values and adding `.?` to a column is a bit of a pain, you'd be right. The good news is that the situation will be much better for Slick 3. But in the meantime, you need to handle it for each column yourself.

There's a way to tell if you've got it wrong. Take a look at this query, which is trying to list users and the rooms they are occupying:

```scala
val outer = for {
  (usrs, occ) <- users leftJoin occupants on (_.id === _.userId)
} yield usrs.name -> occ.roomId
```

Do you see what's wrong here? If we run this, we'll get a `SlickException` with the following message:

`Read NULL value (null) for ResultSet column Path s2._2.`

This is due to users not having to be in a room. And in our test data, one user has not been assigned to any room.

It's a limitation in Slick 2.x. even for non nullable columns.

The fix is to manually add `.?` to the selected column:

```
...
} yield usrs.name -> occ.roomId.?
```

5.2.3 Right Join

In the left join we selected all the records from one side of the join, with possibly `NULL` values from the other tables. The right join (or right outer join) reverses this.

We can switch the example for left join and ask for all users, what private messages have they received:

```
val right = for {
  (msg, user) <- messages.rightJoin(users).on(_.toId === _.id)
} yield (user.name, msg.content.?)

right.run.foreach(result => println(result))
```

From the results this time we can see that just Dave and Frank have seen private messages:

```
(Dave,  Some(Are you thinking what I'm thinking?))
(HAL,   None)
(Elena, None)
(Frank, Some(Maybe))
```

5.2.4 Full Outer Join

At the time of writing H2 does not support full outer joins. But if you want to try it out with a different database, a simple example would be:

```
val outer = for {
  (room, msg) <- rooms outerJoin messages on (_.id === _.roomId)
} yield room.title.? -> msg.content.?
```

That would be title of all room 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.

5.3 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:

```scala
scala> val xs = List(1,2,3)

scala> xs zip xs.drop(1)
/// List[(Int, Int)] = List((1,2), (2,3))
```

Slick provides the equivalent for queries, plus two variations. Let's say we want to pair up adjacent messages into what we'll call a "conversation":

```scala
// Select messages, ordered by the date the messages were sent
val msgs = messages.sortBy(_.ts asc)

// Pair up adjacent messages:
val conversations = msgs zip msgs.drop(1)

// Select out just the contents of the first and second messages:
val results: List[(String,String)] =
  conversations.map { case (fst, snd) => fst.content -> snd.content }.list
```

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`, allows you to give a mapping function along with the join. We could have written the above as:

```scala
def combiner(fst: MessageTable, snd: MessageTable) =
  fst.content -> snd.content

val results = msgs.zipWith(msgs.drop(1), combiner).list
```

The final variant is `zipWithIndex`, which is as per the Scala collections method of the same name. Let's number each message:

```scala
messages.zipWithIndex.map {
  case (msg, index) => index -> msg.content
}.list
```

The data from this query will start:

```
(0,Hello, HAL. Do you read me, HAL?),
(1,Affirmative, Dave. I read you.),
(2,Open the pod bay doors, HAL.),
...
```

5.4 Joins Summary

We've seen examples of the different kinds of join. You can also mix join types. If you want to left join on a couple of tables, and then right join on something else, go ahead because Slick supports that.

We can also see different ways to construct the arguments to our on methods, either deconstructing the tuple with pattern matching, or by referencing the tuple position with an underscore method, e.g. _1. We would recommend using a case statement as it easier to read than walking the tuple.

The examples above show a join and each time we've used an on to constrain the join. This is optional. If you omit the on call, you end up with an implicit cross join (every row from the left table with every row from the right table). For example:

```
(messages leftJoin users).run.foreach(println)
```

Finally, we have shown examples of building queries using either for comprehension or maps and filters. You get to pick which style you prefer.

5.5 Seen Any Scary Queries?

If you've been following along and running the example joins, you might have noticed large and unusual queries being generated.

An example is looking up the user's name and message content for each message:

```
users.
  join(messages).
  on(_.id === _.senderId).
  map{ case (u,m) => u.name -> m.content }
```

The query we'd write by hand for this is:

```
select
  "user".name, "message".content
from
  "user" inner join "message" on "user".id = "message".sender
```

Slick actually produces:

```
select
  x2.x3, x4.x5
from
  (select x6."name" as x3, x6."id" as x7 from "user" x6) x2
  inner join
  (select x8."content" as x5, x8."sender" as x9 from "message" x8) x4 on  ↩
                                                         x2.x7 = x4.x9
```

That's not so bad, but it is a little strange. For more involved queries they can look much worse. If Slick generates such 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. You know the lesson here: measure, use your database tools to EXPLAIN the query plan, and adjust queries as necessary. The ultimate adjustment of a query is to re-write it using *Plain SQL*. We will introduce Plain SQL in Chapter 6.

5.6 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.

5.6.1 Functions

Slick provides a few aggregate functions, as listed in the table below.

Table 5.1: A Selection of Aggregate Functions

Method	SQL
`length`	`COUNT(1)`
`countDistinct`	`COUNT(DISTINCT column)`
`min`	`MIN(column)`
`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:

```scala
val numRows: Int = messages.length.run

val numSenders: Int = messages.map(_.senderId).countDistinct.run

val firstSent: Option[DateTime] = messages.map(_.ts).min.run
```

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.

5.6.2 Grouping

You may find you use the aggregate functions with column grouping. For example, how many messages has each user sent? Slick provides `groupBy` which will group rows by some expression. Here's an example:

```scala
val msgPerUser =
  messages.groupBy(_.senderId).
  map { case (senderId, msgs) => senderId -> msgs.length }.
  run
```

That'll work, but it will be in terms of a user's primary key. 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 }.
   run
```

The results would be:

```
Vector((Frank,2), (HAL,2), (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.

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,   2, 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:

```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:

```scala
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 Vector[T] (from run-ing the query) or List[T] (if we list the query). 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:

```scala
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:

```sql
select min(ts), max(ts) from message where content like '%read%'
```

It's pretty easy to get either min or max:

```scala
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:

```scala
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` collection, and obtain the result we want.

5.6.2.1 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:

```scala
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 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.

5.7 Take Home Points

Slick supports `innerJoin`, `leftJoin`, `rightJoin`, `outerJoin` and a `zipJoin`. 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.

5.8 Exercises

5.8.1 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

5.8.2 User Rooms

In this chapter we saw this query:

```
val outer = for {
  (usrs, occ) <- users leftJoin occupants on (_.id === _.userId)
} yield usrs.name -> occ.roomId
```

It causes us a problem because not every user occupies a room.

Can you find another way to express this that doesn't cause this problem?

See the solution

Chapter 6

Plain SQL

Slick supports plain SQL queries as well as the lifted embedded style we've seen up to this point. Plain queries don't compose as nicely as lifted, or offer 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` and `sqlu` (for updates) are used to create plain SQL queries;
- values can be safely substituted into queries using a `${expresson}` syntax;
- you can build up a query from `Strings` and values using `+` and `+?`; and
- custom types can be used in plain SQL, as long as there is a converter in scope.

6.1 Selects

Let's start with a simple example of returning a list of room IDs.

```scala
import scala.slick.jdbc.StaticQuery.interpolation

val query = sql""" select "id" from "room" """.as[Long]
val result = query.list

println(results)
// List(1, 2, 3)
```

We need to import `interpolation` to enable the use of the `sql` interpolator.

Once we've done that, running a plain SQL looks similar to other queries we've seen in this book: just call `list` (or `first` etc). You need an implicit `session` in scope, as you do for all queries.

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:

```scala
val roomInfoQ = sql""" select "id", "title" from "room" """.as[(Long,String)]
val roomInfo = roomInfoQ.list
println(roomInfo)
// List((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 t = "Pod"
sql""" select "id", "title" from "room" where "title" = $t """. ↩
                                              as[(Long,String)].firstOption
// Some((2,Pod))
```

Notice how `$t` is used to reference a Scala value `t`. 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].firstOption
// JdbcSQLException: Data conversion error converting "Air Lock"; ↩
//                                                    SQL statement:
// [error]  select "id" from "room" where "title" = ?
```

That example compiles without error, but fails at runtime as the type of the `title` column is a `String` and we've provided an `Integer`. The equivalent query using the lifted embedded style would have caught the problem at compile time.

Another place you can unstuck is with the `#$` style of substitution. This 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 query = sql""" select "id" from "#$table" """.as[Long]
```

In this situation we do not want the value of `table` to be treated as a String. That would give you the invalid query: `select "id" from "'message'"`. However, using this construct means you can produce dangerous SQL. The golden rule is to never use `#$` with input supplied by a user.

6.1.1 Constructing Queries

In addition to using `$` to reference Scala values in queries, you can build up queries incrementally.

The queries produced by both and `sql` and `sqlu` (which we see later) are `StaticQuerys`. As the word "static" suggests, these kinds of queries do not compose, other than via a form of string concatenation.

The operations available to you are:

- `+` to append a string to the query, giving a new query; and

- +? to add a value, and correctly escape the value for use in SQL.

As an example, we can find all IDs for messages…

```
val query = sql"""SELECT "id" from "message"""".as[Long]
```

…and then create a new query based on this to filter by message content:

```
val pattern   = "%Dave%"
val sensitive = query + """ WHERE "content" NOT LIKE """ +? pattern
```

The result of this is a new `StaticQuery` which we can execute.

6.1.2 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]` and `+?`. 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]`:

```
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]`. 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:
  scala.slick.jdbc.GetResult[DateTime]
```

6.1.3 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* over at the associated GitHub repository.

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 4, by wrapping the Long primary keys in the type `Id[Table]`.

This gives us:

```scala
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 straight-forward:

```scala
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:

```scala
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:

```scala
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:

```scala
val result: List[Message] =
  sql""" select * from "message" """.as[Message].list
```

In all likelihood you'll prefer `messages.list` 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.

6.2 Updates

Back in Chapter 4 we saw how to modify rows with the `update` method. We noted that batch updates where 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:

```sql
UPDATE "message" SET "content" = CONCAT("content", '!')
```

Plain SQL updates will allow us to do this. The interpolator is `sqlu`:

```scala
import scala.slick.jdbc.StaticQuery.interpolation

val query =
  sqlu"""UPDATE "message" SET "content" = CONCAT("content", '!')"""

val numRowsModified = query.first
```

The `query` we have constructed, just like other queries, is not run until we evaluate it in the context of a session.

We also have access to $ for binding to variables, just as we did for `sql`:

```scala
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.

6.2.1 Composing Updates

All the techniques described for selects applies for composing plain SQL updates.

As an example, we can start with an unconditional update…

```
val query = sqlu"""UPDATE "message" SET "content" = CONCAT("content", $char)"""
```

…and then create an alternative query using the + method defined on `StaticQuery`:

```
val pattern = "%!"
val sensitive =  query + """ WHERE "content" NOT LIKE """ +? pattern
```

The resulting query would append an `!` only to rows that don't already end with that character.

6.2.2 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:

```
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:

```
sqlu"""UPDATE message SET "ts" = """ +? DateTime.now
```

Without the `SetParameter[DateTime]` instance the compiler would tell you:

```
could not find implicit SetParameter[DateTime]
```

Advanced

Compile Warnings

The code we've written in this chapter produces the following warning:

```
Adaptation of argument list by inserting () has been deprecated:
  this is unlikely to be what you want.
```

This is a limitation of the Slick 2.1 implementation, and is being resoled for Slick 3.0. For now, you'll have to live with the warning.

6.3 Take Home Points

Plain SQL allows you a way out of any limitations you find with Slick's lifted embedded style of querying. Two 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.

The tools for composing these kinds of queries is limited. Use +, +?, and $#, but do so with care. End-user supplied information should always be escaped before being used in a query.

6.4 Exercises

The examples for this section are in the *chatper-06* folder, in the source files *selects.scala* and *updates.scala*.

6.4.1 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:
lookup("dave@example.org").as[Long].firstOption
```

What the problem with this code?

See the solution

6.4.2 String Interpolation Mistake

When we constructed our `sensitive` query, we used `+?` to include a `String` in our query.

It looks as if we could have used regular string interpolation instead:

```
val sensitive = query + s""" WHERE "content" NOT LIKE $pattern"""
```

Why didn't we do that?

See the solution

6.4.3 Unsafe Composition

Here's a utility method that takes any string, and return a query to append the string to all messages.

```
def append(s: String) =
  sqlu"""UPDATE "message" SET "content" = CONCAT("content", $s)"""
```

Using, but not modifying, the method, restrict the update to messages from “HAL”.

Would it be possible to construct invalid SQL?

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 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:

- 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:

```sql
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 `Database.forURL` parameters with:

```
"jdbc:postgresql:chapter-01", user="essential", password="trustno1",  ↩
                                                driver="org.postgresql.Driver"
```

A.2.4 Update Slick Driver

Change the import from:

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

to

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

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:

```
"jdbc:mysql://localhost:3306/chapter-01&useUnicode=true&amp;       ↩
 characterEncoding=UTF-8&amp;autoReconnect=true",user="essential", ↩
 password="trustno1",driver="com.mysql.jdbc.Driver"
```

A.3.4 Update Slick Driver

Change the import from

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

to

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

Appendix B

Solutions to Exercises

B.1 Basics

B.1.1 Solution to: Bring Your Own Data

Here's the solution:

```scala
db.withSession { implicit session =>
  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:

When using `db.withSession`, be sure to mark the `session` parameter as `implicit`. If you don't do this you'll get an error message saying the compiler can't find an implicit `Session` parameter for the += method:

```scala
db.withSession { session =>
  messages += Message("Dave","What if I say 'Pretty please'?")
}
// <console>:15: error: could not find implicit value ↩
//   for parameter session: scala.slick.jdbc.JdbcBackend#SessionDef
//                 messages += Message("Dave","What if I say 'Pretty please'?")
//                          ^
```

Return to the exercise

B.1.2 Solution to: Bring Your Own Data Part 2

Here's the code:

```
db.withSession { implicit session =>
  messages.filter(_.sender === "Dave").run
}
// 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:

Again, if we omit the `implicit` keyword, we'll get an error message about a missing implicit parameter, this time on the `run` method:

```
db.withSession { session =>
  messages.filter(_.sender === "Dave").run
}
// <console>:15: error: could not find implicit value ↩
//   for parameter session: scala.slick.jdbc.JdbcBackend#SessionDef
//                 messages.filter(_.sender === "Dave").run
//                                                      ^
```

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:

```
db.withSession { implicit session =>
  messages.filter(_.sender == "Dave").run
}
// <console>:15: error: inferred type arguments [Boolean] ↩
//   do not conform to method filter's type parameter bounds ↩
//   [T <: scala.slick.lifted.Column[_]]
//                 messages.filter(_.sender == "Dave").run
//                          ^
// <console>:15: error: type mismatch;
//  found   : Example.MessageTable => Boolean
//  required: Example.MessageTable => T
//                 messages.filter(_.sender == "Dave").run
//                                         ^
// <console>:15: error: Type T cannot be a query condition ↩
//   (only Boolean, Column[Boolean] and Column[Option[Boolean]] are allowed
//                 messages.filter(_.sender == "Dave").run
//                                ^
```

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 `Column[Boolean]`[1]. 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 `run`, you'll end up returning the query object itself rather than the result of executing it:

[1]Slick uses the `Column` type to represent expressions over `Columns` as well as `Columns` themselves.

```
db.withSession { implicit session =>
  messages.filter(_.sender === "Dave")
}
// res1: scala.slick.lifted.Query[ ↩
//         Example.MessageTable, ↩
//         Example.MessageTable#TableElementType,
//         Seq
//       ] = ↩
//   scala.slick.lifted.WrappingQuery@ead3be9
```

`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

B.2 Selecting Data

B.2.1 Solution to: Count the Messages

```
val results = halSays.length.run
```

You could also use `size`, which is an alias for `length`.

Return to the exercise

B.2.2 Solution to: Selecting a Message

```
val query = for {
  message <- messages if message.id === 1L
} yield message

val results = query.run
```

Asking for 999, when there is no row with that ID, will give back an empty collection.

Return to the exercise

B.2.3 Solution to: One Liners

```
val results = messages.filter(_.id === 1L).run
```

Return to the exercise

B.2.4 Solution to: Checking the SQL

The code you need to run is:

```
val sql = messages.filter(_.id === 1L).selectStatement
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

B.2.5 Solution to: Selecting Columns

```
val query = messages.map(_.content)
println(s"The query is:  ${query.selectStatement}")
println(s"The result is: ${query.run}")
```

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

B.2.6 Solution to: First Result

```
val msg1 = messages.filter(_.sender === "HAL").map(_.content).first
println(msg1)
```

You should get "Affirmative, Dave. I read you."

For Alice, `first` will throw a run-time exception. Use `firstOption` instead.

Return to the exercise

B.2.7 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

B.2.8 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 *doors*", and "I'm afraid I can't *do* that".

Return to the exercise

B.2.9 Solution to: Client-Side or Server-Side?

The query Slick generates looks something like this:

```
select '(message Path @1413221682).content!' from "message"
```

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 `Column[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 `Column[T]` classes:

```
messages.map(m => m.content ++ LiteralColumn("!")).run
```

Here `LiteralColumn[T]` is type of `Column[T]` for holding a constant value to be inserted into the SQL. The ++ method is one of the extension methods defined for any `Column[String]`.

This will produce the desired result:

```
select "content"||'!' from "message"
```

Return to the exercise

B.3 Creating and Modifying Data

B.3.1 Solution to: Insert New Messages Only

```
def insertOnce(sender: String, text: String) ↩
              (implicit session: Session): Long = {
  val query =
    messages.filter(m => m.content === text &&  ↩
                         m.sender === sender).map(_.id)

  query.firstOption getOrElse {
    (messages returning messages.map(_.id)) +=  ↩
              Message(sender, text, DateTime.now)
  }
}
```

Return to the exercise

B.3.2 Solution to: Rollback

The call to `rollback` only impacts Slick calls.

This means the two calls to `delete` will have no effect: the database will have the same message records it had before this block of code was run.

It also means the message "Surprised?" will be printed.

Return to the exercise

B.3.3 Solution to: Update Using a For Comprehension

```
val query = for {
  message <- messages
  if message.sender === "HAL"
} yield (message.sender, message.ts)

val rowsAffected = query.update("HAL 9000", DateTime.now)
```

Return to the exercise

B.3.4 Solution to: Delete All The Messages

```
val deleted = messages.delete
```

Return to the exercise

B.4 Data Modelling

B.4.1 Solution to: Turning Many Rows into 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 `users.run` 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).run` will produce:

```
Vector(dave@example.org)
```

However, notice that if you used `users.ddl.create`, only the columns defined in the default projection were created in the H2 database.

Return to the exercise

B.4.2 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

B.4.3 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

B.4.4 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

B.4.5 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

B.4.6 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(bills.list)

// Unique index or primary key violation:
//bills += Bill(daveId, 24.00)

// Referential integrity constraint violation: "fk_bill_user:
//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

B.4.7 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

B.4.8 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

B.4.9 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

B.5 Joins and Aggregates

B.5.1 Solution to: 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((Dave,4))
```

Return to the exercise

B.5.2 Solution to: User Rooms

A right join between users and occupants can help us here. For a row to exist in the occupant table it must have a room:

```
val usersRooms = for {
  (usrs,occ) <- users rightJoin occupants on (_.id === _.userId)
} yield usrs.name -> occ.roomId
```

Return to the exercise

B.6 Plain SQL

B.6.1 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].list
```

This "email address" turns into two queries:

```
SELECT * FROM "user" WHERE "user"."email" = '';
```

and

```sql
DROP TABLE "user";
```

Trying to access the users table after this will produce:

```
org.h2.jdbc.JdbcSQLException: Table "user" not found
```

Yes, the table was dropped by the query.

Return to the exercise

B.6.2 Solution to: String Interpolation Mistake

The standard Scala string interpolator doesn't have any knowledge of SQL. It doesn't know that `Strings` need to be quoted in single quotes, for example.

In contrast, Slick's `sql` and `sqlu` interpolators do understand SQL and do the correct embedding of values. When working with regular `Strings`, as we were, you must use `+?` to ensure values are correctly encoded for SQL.

Return to the exercise

B.6.3 Solution to: Unsafe Composition

```scala
def append(s: String) =
  sqlu"""UPDATE "message" SET "content" = CONCAT("content", $s)"""

val halOnly = append("!") + """ WHERE "sender" = 'HAL' """
```

It is very easy to get this query wrong and only find out at run-time. Notice, for example, we had to include a space before "WHERE" and use the correct single quoting around "HAL".

Return to the exercise