Preview: Essential Slick

Richard Dallaway and Jonathan Ferguson

Early Access, May 2015

underscore

Preview: Essential Slick

Early Access, May 2015

Published by Underscore Consulting LLP, Brighton, UK.

Contents

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.

Notes on the Early Access Edition

This book is in early access status. The content is complete but may contain typos and errata.

As a early access customer you will receive a free copy of the final text when it is released, plus free lifetime updates thereafter.

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$ 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:

```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)
}
```

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:

```scala
lazy 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:

```scala
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:

```scala
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.

```scala
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 rRecall 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:

```sql
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?

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

See the solution

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