Build iOS Database Apps with Swift and SQLite

Kevin Languedoc

Apress®

Build iOS Database Apps with Swift and SQLite

Kevin Languedoc

Apress®

Build iOS Database Apps with Swift and SQLite

Kevin Languedoc
Montreal
Canada

ISBN-13 (pbk): 978-1-4842-2231-7 ISBN-13 (electronic): 978-1-4842-2232-4
DOI 10.1007/978-1-4842-2232-4

Library of Congress Control Number: 2016958726

Managing Director: Welmoed Spahr
Lead Editor: Aaron Black
Technical Reviewer: Massimo Nardone
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Jessica Vakili
Copy Editor: April Rondeau
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

I dedicate this book to my wife, Louisa, and my children, Patrick and Megan.

Contents at a Glance

About the Author xv

About the Technical Reviewer xvii

Acknowledgments xix

Preface xxi

■ Chapter 1: Creating the Objective-C Wrapper 1

■ Chapter 2: Creating SQLite Databases 11

■ Chapter 3: Creating Databases During Runtime 25

■ Chapter 4: Altering Databases and Other Features 45

■ Chapter 5: Inserting Records 63

■ Chapter 6: Selecting Records 89

■ Chapter 7: Updating Records 113

■ Chapter 8: Deleting Records 131

■ Chapter 9: Searching for Records in SQLite 145

■ Chapter 10: Working with Multiple Databases 161

■ Chapter 11: Backing Up SQLite Databases 175

■ Chapter 12: Analyzing SQLite Databases 187

Index 197

Contents

About the Author xv

About the Technical Reviewer xvii

Acknowledgments xix

Preface xxi

Chapter 1: Creating the Objective-C Wrapper 1

Getting Started 1

Creating the Swift iOS Project 2

Create the Db Mgr Project Structure 3

Adding the SQLite 3 Library 3

Creating the Bridge 5

Creating the Bridge Header File 5

Configuring the Swift Compiler 7

Creating the Swift Wrapper Functions 7

Add the DbMgrDAO Class 8

Create the init() func 9

Creating the SQLite Execute Function 10

Summary 10

Chapter 2: Creating SQLite Databases 11

Creating Databases and Adding them to the Project 11

Launching SQLite Manager 12

The SQLite Manager Menu 12

Create the Database 13

Add Table and Columns 15

Add an Index 18
Add a View 19
Add a Trigger 21
Create an iOS Project 22
Add Database to the Project 23
Summary 24
Chapter 3: Creating Databases During Runtime 25
Building the DB Mgr App 25
The Application UI 25
Creating the Data Model 27
Creating the Controllers 30
The DbMgrDAO Controller 31
The MasterViewController 36
The DetailViewController 39
Building the Winery Database 41
Create the Winery.sqlite File 42
Summary 44
Chapter 4: Altering Databases and Other Features 45
Modifying Tables 46
Renaming a Table 46
Adding Columns 47
Re-indexing a Table 48
Modifying Views 48
Modifying Indexes 48
Modifying Triggers 49
Adding and Altering Collation Sequences 50
Binary 51
NoCase 51
Rtrim 51
sqlite3_create_collation 51

The SQLite DELETE Statement 52
Deleting Tables 52
Deleting Views 52
Deleting Indices 53
Deleting Triggers 53
Deleting Collation Sequences 53
SQLite Functions 53
The JSON Extension 53
Creating Functions using Swift 55
Using Functions in a SQLite Database using Swift 58
Pragma Statements 59
Foreign_key_check 59
Foreign_key_list 60
Integrity_check 60
Automatic_index 60
Busy_timeout 60
Shrink_memory 60
Auto-vacuum 60
Corrupting a SQLite Database 61
SQLite Limits 61
Summary 62
Chapter 5: Inserting Records 63
The Data-Binding Functions 63
The SQLite INSERT function 64
Insert or Replace 66
Insert or Rollback 67
Insert or Ignore 67
Insert or Abort 67
Insert or Fail 67
Inserting Blobs 68

Creating the Winery App 69
Create the Project 69
Add the Bridge 69
Creating the UI View for Inserting 72
Creating the Data Model 75
Add the Wineries Database 76
Add the Wine Type 76
Add the Wineries Type 77
Add the Database Schema 77
Creating the Controllers 78
Add the WineryDAO Class 78
The init() function init() 78
The buildSchema Function 79
The createOrOpenDatabase Function 80
The insertWineRecord Function 80
The insertWineryRecord Function 81
The FirstViewController 81
Add Photo Capture Functionality 82
The SecondViewController 83
Running the App 84
Inserting Records 84
Summary 87
Chapter 6: Selecting Records 89
Column Data Types 89
The SELECT Statement 90
Selecting Data 90
Using a Dynamic WHERE Clause 92
Perform a SELECT using a Sub-Query 93
Perform a SELECT using Joins 93
Select and Display Images 94

Select and Playback Audio Records 95
Select and Display Video Records 96
Adding SELECT Functionality to the Winery App 96
Add the SelectWineries UIPicker 96
The viewDidLoad Function 96
The UIPickerView Functions 97
The selectWineriesList Function 98
The selectWineList Function 99
The selectWineryByName Function 99
Modifying the UI for Displaying Records 100
Adding the UITableViewControllers 100
Adding the Navigation Controllers 102
Connect the TableViewControllers and TableViewCellController 104
Adding the IBOutlets: WineList Controller 106
Add the Business Logic 107
Running the App 109
Summary 112
Chapter 7: Updating Records 113
SQLite Update Statement 113
UPDATE Using a Where Clause 115
UPDATE Using a Sub-query 115
Updating Records Using a Join 116
UPDATE Using a Sub-Query in FROM Clause 116
Update On Conflict 116
A Sample SQLite UPDATE Operation in Swift 117
Adding the UPDATE Functionality to the Winery App 118
Modifying the WineryDAO Controller 118
Modifying the UI for Updates 120
Running the App 124
Updating Records 125
Summary 130

Chapter 8: Deleting Records 131
The DELETE Statement in SQLite 131
Using the WHERE Clause 131
Restrictions and Triggers 132
DELETE Limits 132
A Swift SQLite Delete Example 132
Adding the Delete Functionality to the Winery App 134
Modifying the WineryDAO Class 135
Modifying the ViewControllers 136
Modifying the TableViewControllers 136
Modifying the UI for Delete 137
Modifying the UI 137
Running the App 138
Summary 144
Chapter 9: Searching for Records in SQLite 145
The Search App 145
Create the SQLite Database 146
Create the iOS/SQLite Project 148
Running the App 158
Summary 159
Chapter 10: Working with Multiple Databases 161
The ATTACH Statement 161
The DETACH Statement 162
Multi-Database Limits 162
Performing Joins 163
Attach and Detach in Swift 163
The Project 164
The ViewController 164
Building the UI 169

Running the App 171
Summary 173

Chapter 11: Backing Up SQLite Databases 175

Overview of the SQLite Backup Methods 175
Make a Backup Copy 175
Back Up In-Memory SQLite Databases 176
Back Up On-Disk SQLite Databases 177
The Backup App 177
The ViewController 181
Building the UI 183
Summary 185

Chapter 12: Analyzing SQLite Databases 187

The Analyze Statement 187
The sqldiff Tool 188
The sqlite3_analyzer tool 189
Summary 196

Index 197

About the Author

Kevin Languedoc has been a database and software developer for more than 20 years, having worked with many of the leading databases systems on desktop, client/server, mainframe, web/cloud, IoT, and mobile. He brings his expertise in database development to the iOS (iPhone and iPad) platform and SQLite. He graduated from University of Quebec at Montreal with a degree in computer science and from McGill University with a degree in marketing management. He has developed over 50 iPhone and iPad apps at the corporate level for clients of all sizes. He has worked in finance, banking services and insurance, and retail and distribution, as well as in energy, transportation, and pharmaceuticals on both sides of the fence. He brings a wealth of practical experience to this book, along with extensive code samples and working apps.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, web/mobile development, cloud, and IT architecture. His true IT passions are security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in computing science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years.

Technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, and so on.

He currently works as chief information security officer (CISO) for Cargotec Oyj.

He has worked as a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (in the PKI, SIP, SAML, and Proxy areas).

Massimo has reviewed more than 40 IT books for different publishing companies, and he is the coauthor of *Pro Android Games* (Apress, 2015).

This book is dedicated to Antti Jalonen and his family, who are always there when I need them.

Acknowledgments

A book is never written in isolation; it is a team effort. Even though the author's name appears on the front cover, it requires the support and effort of many people who work behind the scenes to make sure the project is successful. First and foremost, I wish to thank my wife, Louisa, and my kids, Patrick and Megan, for giving me the support, the space, and the understanding to write this book. At Apress, I offer my sincere thanks to Steve Anglin for offering me this opportunity and to Jessica Vakili, who was always patient and helpful and made sure the project moved along smoothly. My thanks also go to Aaron Black for his support in organizing this project and bringing together this great team, which also includes Jim Markham, Massimo Nardone, and Eric Levasseur, who all provided fantastic insight to smooth out the rough edges and make the book shine. To Dhaneesh Kumar, for preparing the front and back covers, thank you. To everyone else who has worked on this project at Apress, I wish to say thank you for helping me with my project.

All my sincere thanks,
—Kevin

Preface

I've been working as a software developer for over 20 years. The central theme of my IT career has been database development, so I wanted to bring this experience to this book. I have worked with many of the popular database systems over the years, like Oracle, MSSQL, and DB2, not to mention oldies like Borland Dbase4, Lotus Approach, Superbase95, and, of course, Lotus Domino. However, I am uniquely impressed with SQLite. The versatility of this technology is incredible. You can take a SQLite database that was created on a Windows PC, for instance, and run it unchanged on an OSX, Linux, Android, JavaScript, and, of course, on iOS.

Did you know that SQLite was once the basis for the Web DB specification in an early version of W3C's HTML5 specification before being dropped from the final release candidate in 2010? The technology is in widespread use across most computing platforms, including mobile and embedded systems. Apple uses SQLite on both OSX and iOS for various applications. For data-driven iOS applications, SQLite is the logical choice. There are other choices, like InterBase or BerkeleyDB, if you want to use a relational database, or you can go the NoSQL route and choose Couchbase or Realm, among others. However, SQLite offers a solid footing and strong industry support, which I find is essential if you are planning a production-grade data-driven iOS app.

With the introduction of Swift, Apple has provided millions of iOS developers with a modern, well-designed programming language in tune with Python, Ruby, Scala, Groovy, and many others. The language is fun to work with. It is also much less verbose than Objective-C, and I have had the opportunity to work on an iOS project with this language and found Swift to be a great, lightweight language to work with.

Since the first edition of this book was completely written in Objective-C, with one chapter on a beta version of Swift 1, I felt it was time to re-write the book using Swift 3 and iOS 10. I wanted to explore the opportunity this new language offers to build data-driven apps using SQLite as a backend. I also wanted to show you how easy it is to create bridges in Swift to interface with SQLite's C API—or any C/C++ API, for that matter—to harness many of the mature libraries that have been developed and have stood the test of time, as well as new ones, through Swift.

In keeping with the original design of the first edition, *Build iOS Database Apps using Swift and SQLite* walks you through the steps of creating a bridge, developing SQLite databases, and performing the standard CRUD operations that characterize a database's core functionality of storing and retrieving data. I also wanted to explore extending SQLite with custom functions and attaching them using Swift, and to look at the multi-database and backup APIs.

The code for the apps in the book was written in Swift 3. I hope you find the book useful and practical.

CHAPTER 1

Creating the Objective-C Wrapper

Creating a wrapper for a C library is a very easy task with Swift, as the language was designed for interoperability with Objective-C and C, as well as for C++ language. Although this chapter focuses on using the SQLite C API with Swift, you could easily do the reverse and call Swift from these languages. This chapter shows how to create an Objective-C wrapper around the SQLite C API, which can then be interfaced through Swift. This method can be replicated with other C libraries.

In this chapter, I'm going to show you how to do the following:

- Create a Swift project
- Add the SQLite 3 C API library to the project
- Create a bridge header file to interface with the C API
- Configure the Swift compiler
- Create a DAO class to handle the execution for queries

Getting Started

Before getting to the bridge, I think it is important to mention the tools that are needed for this book and also for iOS application development in general. All the iOS code for this book is produced through Xcode 8.0 at the time of this writing. I am also using Swift 3.

If you are planning on developing Swift iOS apps for this book or in a production environment, your only choice is Xcode on OSX. Some will argue that Swift can compile and run on Linux also. While this is true, Xcode cannot, and you need Xcode to develop iOS, TVos and OSX apps. You will also need an Apple Developer account. For the development in this book, you can get by with the free edition, but to provision and deploy you will need an Apple Developer license, which you can get though Apple Developer website.

I will also use SQLite Manager in Firefox to create SQLite databases in chapters 2, 5, and 10; otherwise, all development, including creating SQLite databases, will be done in Xcode.

Xcode can be installed for free through the OSX App Store, which is accessible uniquely on a MAC OSX. While you can purchase a copy of OSX through an Apple store, and it can be installed on a virtual machine, the OSX App Store will only work on a real OSX machine.

Installing Xcode is very easy. Once you select it in the App Store, it is installed automatically on your OSX machine. When you install Xcode, the SDK (for Swift/Objective-C) is also installed. You can install more tools through Xcode. Under the Xcode menu, select Preferences and then Components.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2232-4_1) contains supplementary material, which is available to authorized users.

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_1

The SQLite library is already included in the iOS SDK, and this is the version that I will be using throughout this book. You can download and install SQLite from the SQLite site, which has extra APIs such as the JSON extension, but this is out of the scope of this book. You can also download and use Swift SQLite frameworks that have been created as wrappers over the SQLite C API. However, I won't be using them in this book, as I find the C API very easy to work with on its own.

Other than Xcode and an Apple OSX computer, you only need Firefox SQLite Manager, which can be downloaded and installed from Mozilla. The SQLite Manager is installed from the Addon marketplace in Firefox.

Let's get started.

Creating the Swift iOS Project

To work with SQLite 3 in iOS Swift applications, you need a bridge header file to interface with the SQLite C API. In fact, this interface is applicable to all interfaces between Swift and Objective-C, C, or C++.

Open Xcode and create a new project from the launchpad. For this project, we are building a SQLite Database Manager app that will allow a developer to connect to a SQLite database. The app will also provide functionality to add or modify tables, views, indexes, and triggers. Finally, one will also be able to insert, update, and select data from the database.

To build this app, select the Master-Detail iOS template from the list of templates under the iOS Application heading. Name the iPad app SQLite Mgr and create it. Figure 1-1 shows an example of the Master-Detail template under the iOS Application heading. After selecting the template, move to the next page to fill in the app details. This template provides a great classic layout for building iPad apps and takes advantage of the extra screen real estate. The template has a view on the left-hand side that acts as an index by default, although you could replace the `UITableView` with another view. The view on the right-hand side provides an expansive window to display input forms or other app details, including web pages and tables or other view controllers.

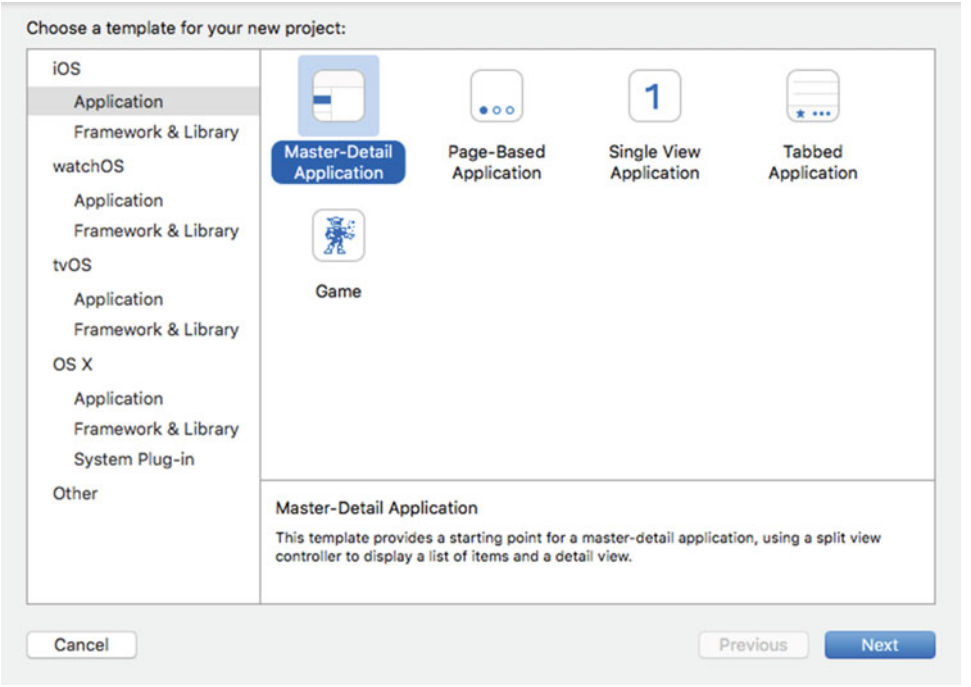

Figure 1-1. *Master-Detail iOS Application template*

For this reference app, we will use some of the default design elements that are included with the template, namely the `UITableView` in the `MasterViewController`. For the `DetailViewController`, we will develop a new UI layout.

Figure 1-2 illustrates the information needed to name and set up the basic app structure. The Organization Name is used throughout the project and documentation, such as for the copyright notice. The Organization Identifier is used to create part of the Bundle ID for the App Store and to identify the app in iTunes Connect, for instance.

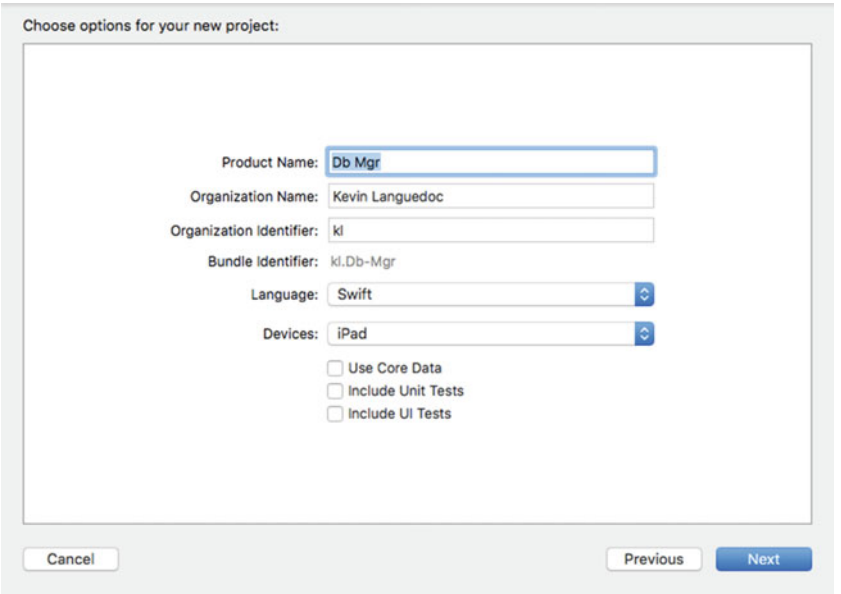

Figure 1-2. *Enter the app name and other required information*

Create the Db Mgr Project Structure

The project structure of a Swift app is identical to that of an Objective-C structure in Xcode. As a matter of preference, I like to organize my app objects into logical groups. So, we will create groups for "views," "models," and "controllers," as well groups for "libs," "bridge," and "utils." The first three are to group the files related to the MVC design pattern. The last two will group the SQLite 3 library and some helper classes that we will use for the app that don't fit in the other groups.

To add files to the project, right click on the group heading and use the context menu to select the Add New File command, which will insert the new file that we will create under that header. You can also add a new file from the menu in Xcode and select from the dialog box for naming the file the group where you want to file to be created.

Adding the SQLite 3 Library

Add the SQLite 3 library to the project as you would normally do through the Linked Libraries and Frameworks. Select the project root in the navigator and scroll down to the Linked Libraries and Frameworks from the General sheet of the Project Properties page, which appears in the main window.

Figure 1-3 provides an example of the Search dialog box used to add libraries to an Xcode project. Click on the “+” sign to open the Search window and type "Sqlite3." You will get two results; one of the selections is the library and the second is a link to the library. Select the libsqlite3.tbd file and click the Add button to add the library to the project.

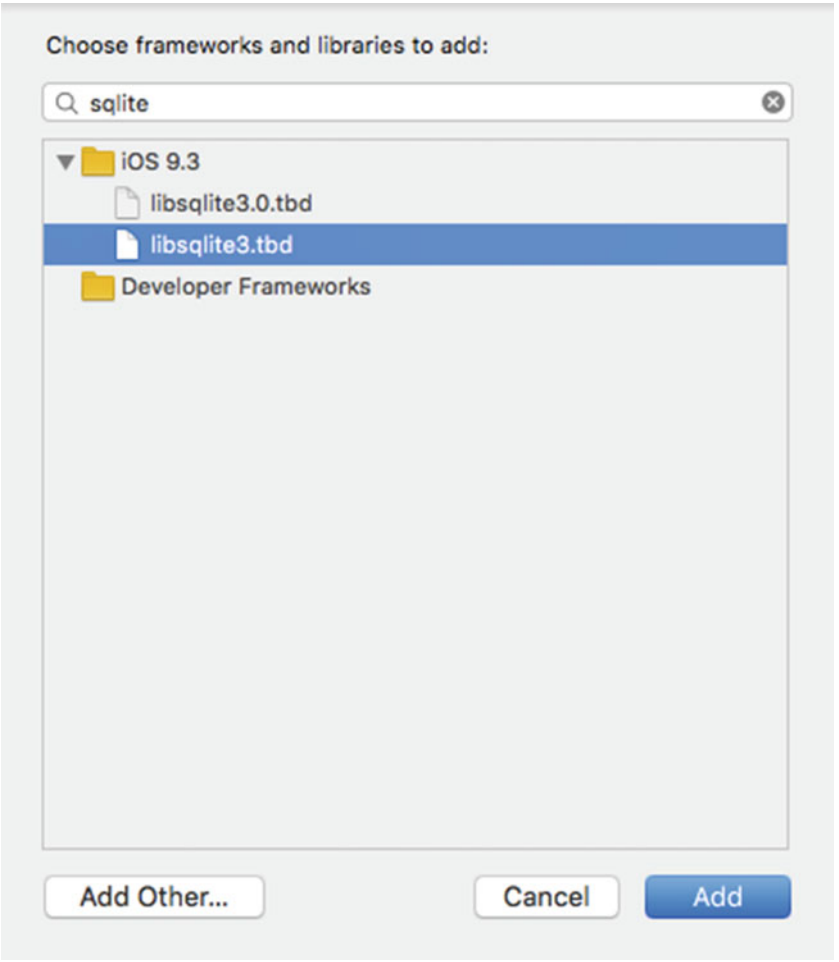

Figure 1-3. *Add the SQLite library to the project*

In the project explorer (navigator), drag the SQLite3 library into the "libs" group. In the next sections, I will show you how to create the bridge file and configure the Swift compiler.

Creating the Bridge

The bridge is an interface file with which to configure access to the C, C++, or Objective-C libraries and to configure the Swift compiler to look for and use said libraries when compiling the code. Since the SQLite library is written in C, we will need a bridge file in order to access the C API from Swift.

Creating the Bridge Header File

From the File menu in Xcode, select File > New File. In the template selection page, select the Header File template and click the Next button to move to the next page, where we will be able to name the file (Figure 1-4).

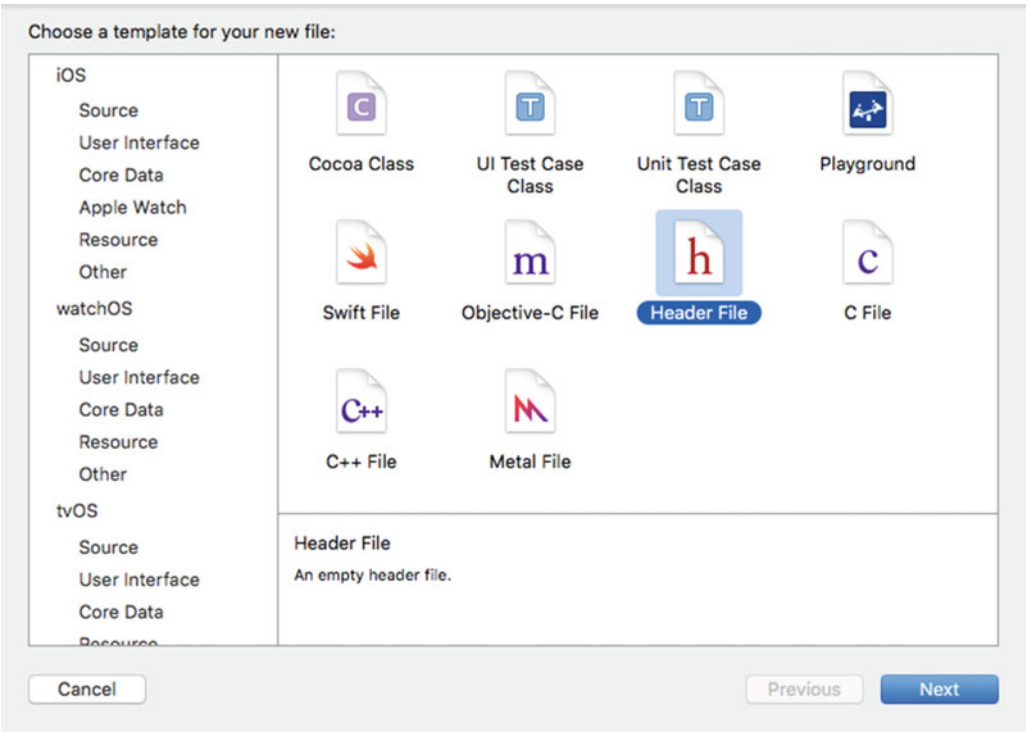

Figure 1-4. *Select the C Header File template*

For this example, I will name it SQLite3Bridge.h, but you are free to name it as you wish; Xcode will automatically append the header extension (.h) to the file name. If you created the project groupings as indicated earlier, select the "bridge" group and click on the Create button to create the file and add it to the project (Figure 1-5). Once the header file is added to the project, it will open in the Xcode editor.

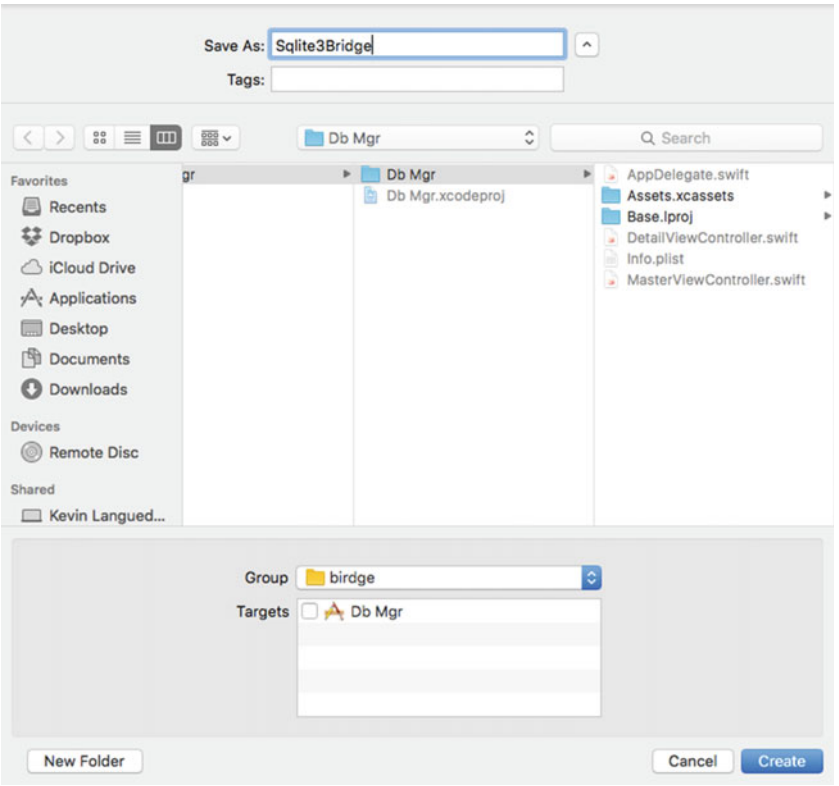

Figure 1-5. *Select the group and file location*

In the C header file code that follows, you'll notice that the template has added some directives and constants for us to define the header file. To create the SQLite3 bridge, I will add a reference to the SQLite3 library by adding an `import` statement along with the SQLite3 header file name, as in the code snippet. Save the file.

```
//
//  SQlite3Bridge.h
//  Db Mgr
//
//  Created by Kevin Languedoc on 2016-05-20.
//  Copyright © 2016 Kevin Languedoc. All rights reserved.
//

#ifndef SQlite3Bridge_h
#define SQlite3Bridge_h

#endif /* SQlite3Bridge_h */

// Add this code to import the sqlite3 header. The code above is supplied by the template
#import <sqlite3.h>
```

Configuring the Swift Compiler

The next step is to let the Swift compiler know about the bridge file, and that is all that is needed to create the bridge.

Open the Build Settings page by selecting the project root in the Navigator pane. In the field, type “swift” to locate the Swift Compiler - Code Generation compiler section. In the Objective-C Bridge Header field, add the sub-directory, which is the project name followed by the name of the bridge header file (Figure 1-6).

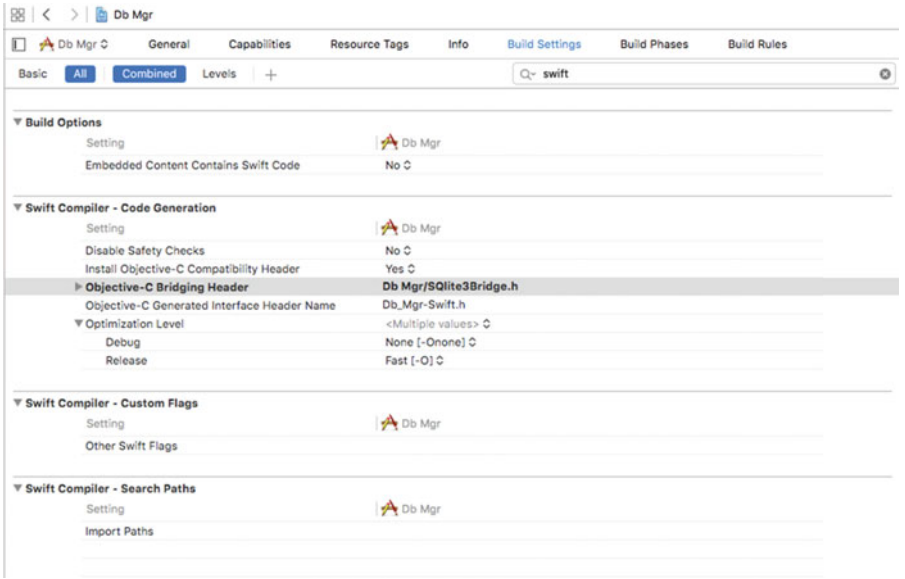

Figure 1-6. *Swift compiler setting on the Build Settings page*

Creating the Swift Wrapper Functions

With the bridge in place, we can move on to the final step in this chapter, which is create the basis of the DAO class, `DbMgrDAO`. This class is a sub-class of the `NSObject` class. You can create this class using one of two templates. You can create it using the Cocoa Touch template, which will add the proper class signature, but you will need to replace the `import` statement from `UIKit` to `Foundation;` otherwise, you will get some errors. However, using the template, you will have the opportunity to select the sub-class and language. The other way is to use the Swift template, which is what I will use, as the template will create a bare-bones file.

Using the Swift template (Figure 1-7), we will need to add the class definition, as the template only adds the `import Foundation` statement after we name the file and add it to the project under the "controllers" group. The class will be titled `DbMgrDAO`.

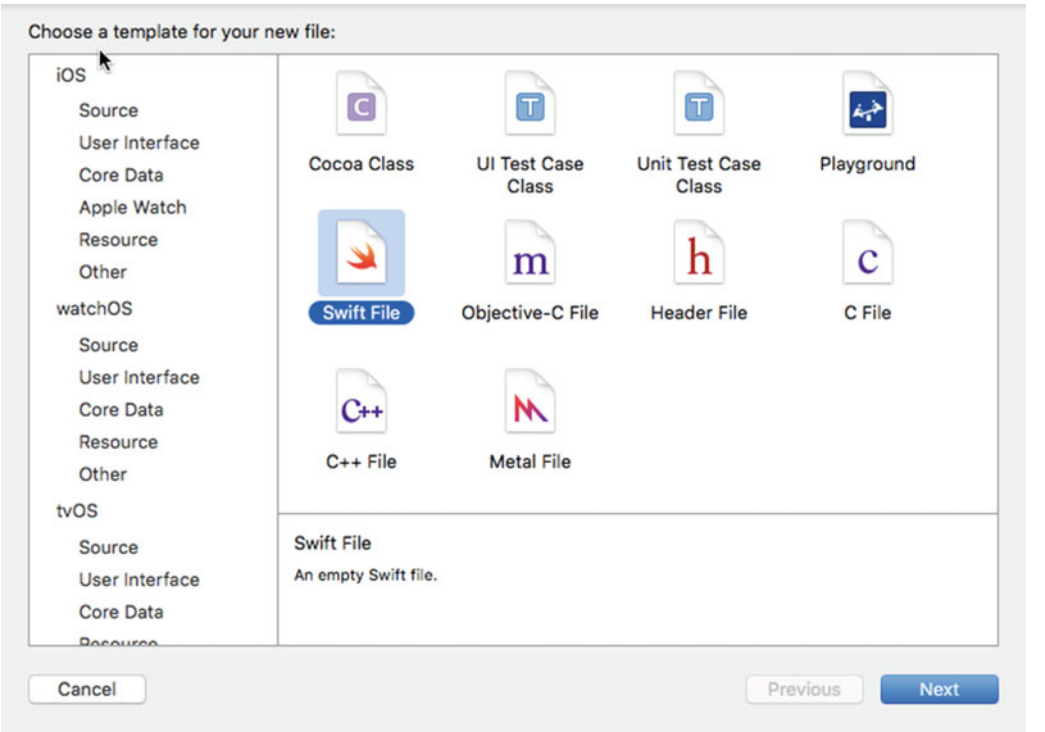

Figure 1-7. *Select the Swift template*

Add the DbMgrDAO Class

The `DbMgrDAO` class will provide most of the interaction between the two view controllers and the SQLite database—or databases, if you opt to create more than one with the app.

From the File menu in Xcode, select the Swift File template under the iOS heading. Move to the next page using the Next button. On the next page, name the file DbMgrDAO (Xcode will add the Swift extension if you don't) and click the Create button (Figure 1-8). Again, if you added groups to the project structure you will be able to select one here. I will select the "controllers" group before creating the file.

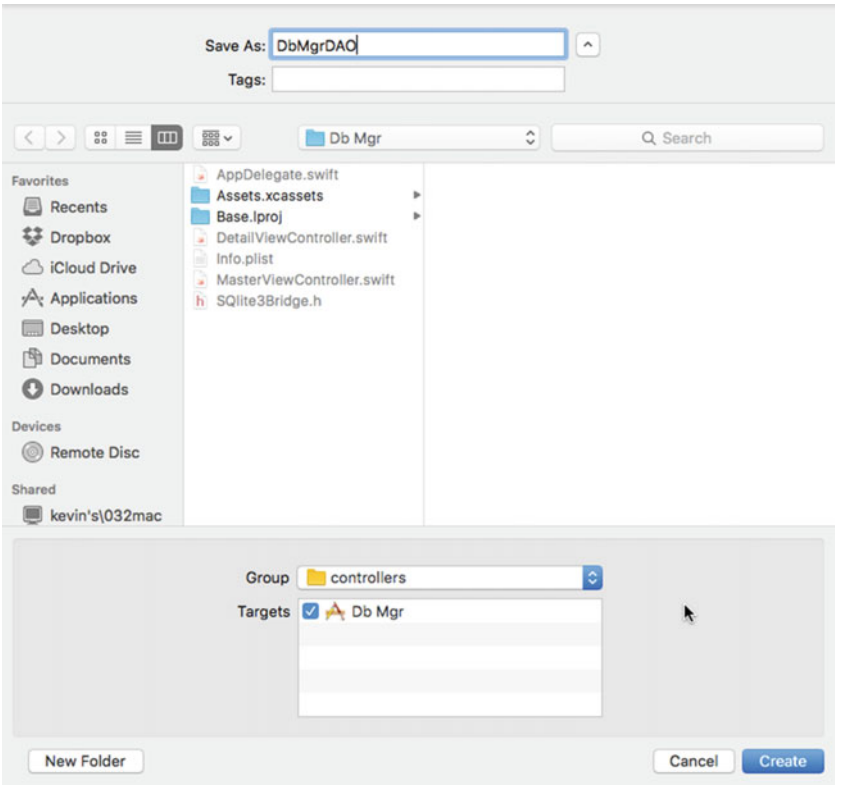

Figure 1-8. *Add the DbMgrDAO file to the project*

```
//
//  DbMgrDAO.swift
//  Db Mgr
//
//  Created by Kevin Languedoc on 2016-05-24.
//  Copyright © 2016 Kevin Languedoc. All rights reserved.
//

import Foundation

public class DbMgrDAO:NSObject{

}
```

Create the init() func

To finish up I will add two basic functions. The first is the `init` function. This function is called to initialize a `DbMgrDAO` object and can be used to set up the object, such as establishing a SQLite database connection and opening a database, as I will show later in the next chapter.

```
import Foundation

public class DbMgrDAO:NSObject{

    override init() {
        //code here
    }
}
```

Creating the SQLite Execute Function

Lastly, I will add the `executeQuery` function that will be used to execute queries like inserts, updates, and deletes. For `select` operations that return records, I will create another function in a later chapter, as I won't need it in this example app.

```
//
//  DbMgrDAO.swift
//  Db Mgr
//
//  Created by Kevin Languedoc on 2016-05-24.
//  Copyright © 2016 Kevin Languedoc. All rights reserved.
//

import Foundation

public class DbMgrDAO:NSObject{

    override init() {
        //code here
    }

    func executeQuery(query:String?){

    }

}
```

The function signature includes one string data–type parameter for the query. That's it for now.

Summary

This chapter started off by showing you the tools that are needed for iOS app development in general and for SQLite specifically. Then, we focused in on creating the Objective-C bridge that allows Swift to converse with the SQLite C API. This is standard procedure from Apple for Swift development using C/C++ APIs.

This bridge is the cornerstone for every app we will build and explore in the following chapters, and it is the primary piece of glue required in order to work with SQLite using Swift.

The next chapter will show you how to develop a SQLite database using Firefox and the SQLite Manager add-on. Once the database is created, we will develop the Db Mgr app, add the database to the project, and copy the database into the Document directory, because this is the primary writeable directory in an iOS app sandbox (filesystem).

CHAPTER 2

Creating SQLite Databases

SQLite is very flexible in regards to creating databases. You can include a file extension, or not, as all you really need is to pass a file name and path to the `sqlite3_open` function and the database will be created and opened—although it won't contain any structure. Unlike other relational databases like MSSQL, SQLite databases are self-contained and portable. A SQLite database file can run without any changes on all supported platforms, including, of course, iOS. SQLite databases are not designed to run a server.

In this chapter, I will continue to build on the previous chapter by showing how to create a SQLite database using a database tool like SQLite Manager in Firefox and adding it to the project. In the next chapter, I will provide you with the knowledge to create databases during runtime, including adding the necessary structure.

You can also create databases using the command line, like Terminal (OSX, Linux) or Windows Command.

The following topics are covered in this chapter:

1. Creating and adding a SQLite database to a project
2. Adding tables and columns
3. Adding views
4. Adding a trigger
5. Adding an index

Note While you can use SQLite Manager in Firefox on all supported platforms to develop the SQLite database, you will need to export the database file to OSX for inclusion in your iOS/Swift project. Even though Swift and SQLite are supported on Linux, you will need Xcode to provision your app and deploy it to iTunesConnect. The focus of this chapter, and indeed this book, is to show you how to develop iOS apps (iPhone/iPad) using Swift 3 and SQLite databases. I assume you are using Xcode 8 on OSX El Capitan, along with the SQLite 3 included in the iOS SDK and in SQLite Manager.

Creating Databases and Adding them to the Project

In this first part of this chapter, I will focus on developing a SQLite database using the SQLite Manager add-on in Firefox, and then I will import the finished database into an iOS Swift iPad app for later use. There are a few open source and commercial database editors for SQLite. I am using SQLite Manager in Firefox, as it is free (on all supported platforms) and lightweight. It is a Firefox add-on and can be installed in Firefox through the Add-on interface.

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_2

The second part of the chapter will concentrate on creating databases though a running iOS app. I will expand upon the iPad app I am creating in this book to create my own database-manager app. This will involve creating a SQLite database in the Db Mgr app, which I will build in the next chapter.

Launching SQLite Manager

Once the SQLite Manager is installed via the Add-on gallery, you will be able to launch it from the Tools menu in Firefox. Figure 2-1 shows the SQLite Manager menu item in the Tools menu in Firefox.

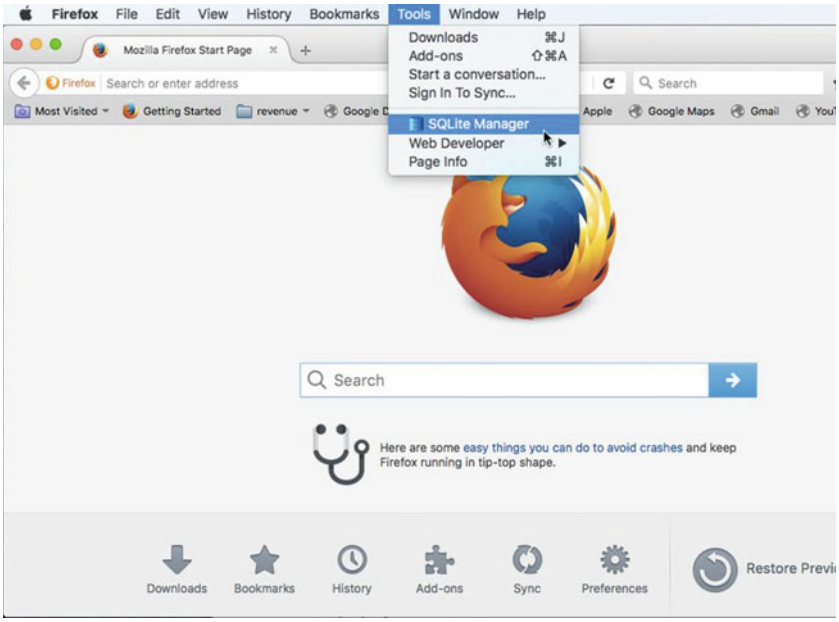

Figure 2-1. *SQLite Manager menu in Firefox*

The SQLite Manager Menu

The SQLite Manager has a quick-access menu of the main activities you can perform on a SQLite database, in addition to various menu options. Under each menu option you can create, drop, rename, and modify the listed database element. Under the Database menu, in addition to creating, modifying, indexing, re-indexing, attaching, and detaching a database, you can also compact a database and create in-memory databases. The app also offers options to analyze a database.

Figure 2-2 provides a visual of the SQLite Manager Toolbar. You can choose to create new databases, tables, and views. You can also open an existing database.

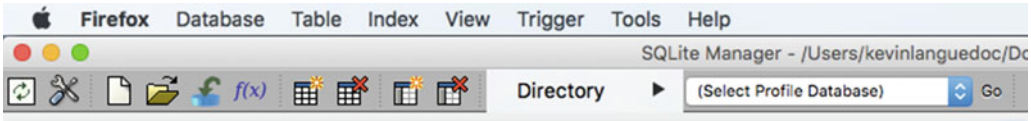

Figure 2-2. *SQLite Manager toolbar*

For the purposes of this example, I will create a database and then add tables, views, a trigger, and indexes. Once the database is created, I will add it to the Db Mgr project.

The Directory selector lets you change the current directory from the Profile directory, which is the directory used by Firefox to store the various databases it uses, to use the directory where our databases will be located.

Create the Database

You can create the database using the new document icon in the menu bar or by choosing the New Database option under the Database menu item. I will name the database Chapter 2 for this example. By default, the .sqlite extension is used; however, you can change this behavior in the settings (the criss-cross screwdriver/wrench icon). Figure 2-3 shows the dialog box that lets you provide the SQLite database name.

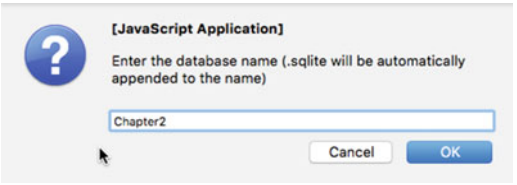

Figure 2-3. *Name the SQLite database*

SQLite Manager offers you many options to tweak and analyze a database. Switch to the Db Settings tab, for instance, after the database is open, and you can set a panoply of different settings to fine tune the database (Figure 2-4). I will accept the defaults for this example, but I invite you to explore these many features of SQLite Manager and the different settings available through the SQLite API.

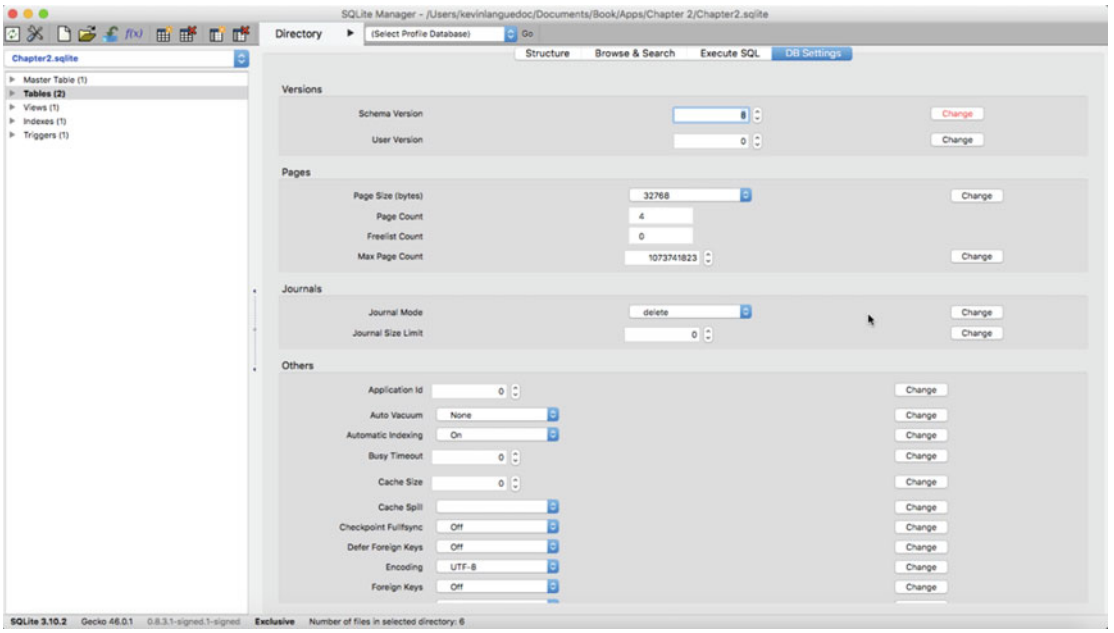

Figure 2-4. *SQLite Manager Settings tab*

I would like to touch on a couple of interesting and useful concepts, which are the `sqlite_master` table located under the Master Table node, the `sqlite_sequence` table, and the Main database versus the Temp database or other databases.

Sqlite_master

The `sqlite_master` table contains all the queries that are used to create the database schema; for instance, when you add or modify a table, view, or trigger. The table also contains the names and types of each database schema element. Later in the chapter, we will make use of this table to populate the data source for the `TableView` in the `MasterViewController` in the next chapter. The schema for the table looks like the code snippet that follows. SQLite creates this table for you when the first table is added to the database. This is a representation only:

```
CREATE TABLE sqlite_master (
  type TEXT,
  name TEXT,
  tbl_name TEXT,
  rootpage INTEGER,
  sql TEXT
);
```

You can also browse the table by selecting the Browse & Search tab with the `sqlite_master` table selected. You can also browse the information in the table or any other table by performing a `SELECT` query on it, as in the code snippet following Figure 2-5.

```
SELECT * FROM sqlite_master

or

SELECT name, tbl_name FROM sqlite_master
```

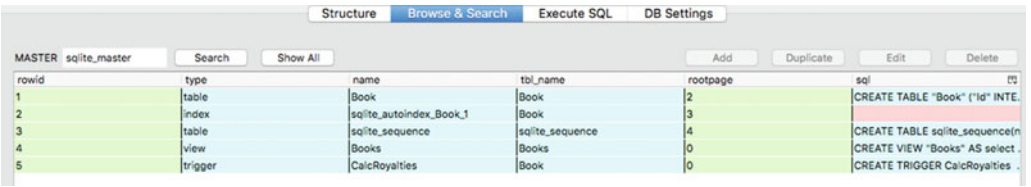

Figure 2-5. *Sqlite_master Browse & Search window*

The sqlite_sequence

The `sqlite_sequence` table maintains the largest `ROWINDEX` of a given table. It is used in conjunction with the `AUTOINCREMENT` property on a column. When I create the tables in the database, I will use `AUTOINCREMENT` as the primary key. If the table is empty, then the largest `ROWID` will be 1 and so on as records are added to the table.

The last feature I want to touch on is the Main database. In SQLite you can attach many databases to the same connection or database file. When you create a new database file or connection, SQLite automatically adds a database to the file and calls it Main. You have the option to create additional databases in the same connection and you name these databases separately and attach them together using the `ATTACH` command. Likewise, you can remove these auxiliary databases using the `DETACH` command.

Figure 2-6 shows a screenshot the Directory menu item. The Directory menu is another useful way of locating SQLite databases and their home directory. You can locate the database file on disk by choosing the "Default User Directory" option. This will open a Finder window if you are on OS X or a File Explorer window if you are using Windows.

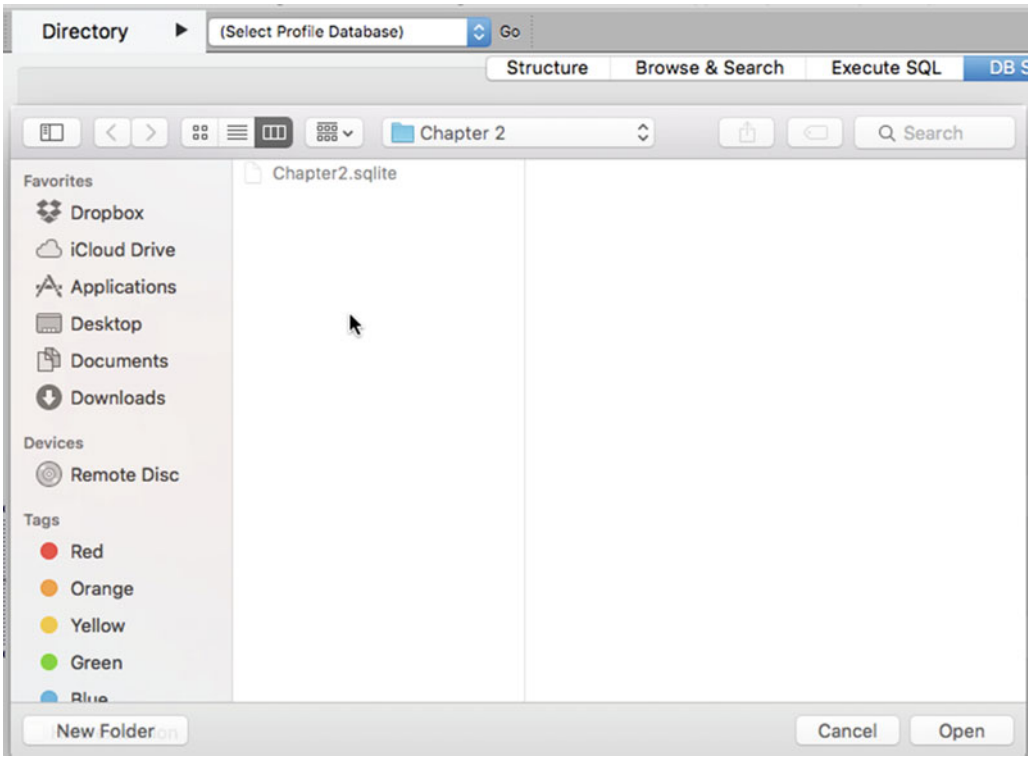

Figure 2-6. *SQLite Manager Directory menu*

With the basic database in place, I will move to the next step and create a table and columns.

Add Table and Columns

Creating tables and columns in SQLite is quite easy. You can opt to use the new table icon or select the Create Table option from the Table menu. I have provided a screenshot of the Create Table interface in SQLite Manager (Figure 2-7). When you enter the table definition, it only captures information to build a `Create Table` query.

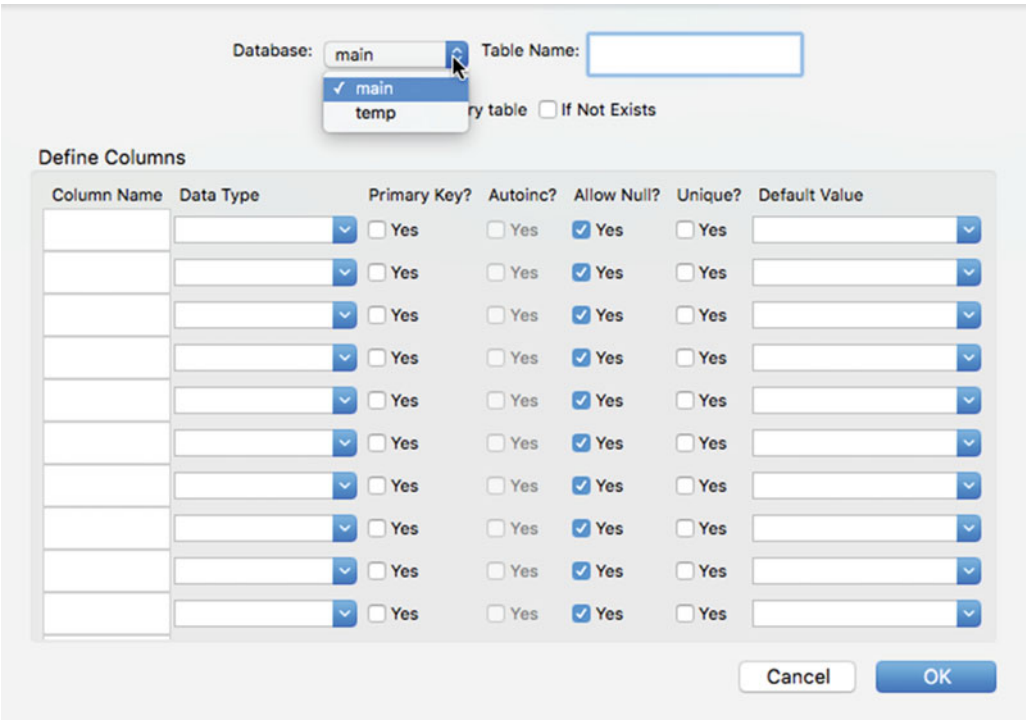

Figure 2-7. *SQLite Manager's Create Table interface*

You'll notice the dropdown menu for the database selection near the top of the interface. You have a choice between main, which is already selected, and temp. If you attach other databases, they would appear here as well.

If you have experience developing database applications, you will find many of these fields self-explanatory. You have a field for the table name. You can choose to make the table temporary using the "Temporary table" option. You should always select the "if not exists" option so you don't accidentally overwrite the database table and its contents when running the app. This option is added to the query, and the database engine checks to see if the table already exists before creating it.

Figure 2-7 illustrates how you can also define your table columns. You can provide a column name, a data type—the dropdown includes the data types supported by SQL language—and whether the column will act as the primary key. You can have the column auto increment by enabling the "Autoinc" option. Note that this option is only active if you opt to enable the primary key on the column. Of course, you can choose to not allow nulls by selecting the "Yes" option. This is especially important for primary key fields. The "Unique" option ensures that no duplicates are introduced through an `INSERT` or `UPDATE`. Finally, you can set a default value depending on the selected data type.

Once the table definition is complete and you click on the OK button, a confirmation will appear displaying the exact query that will be performed to create the table. This is a great way to learn the proper syntax to use to prepare the SQL queries supported by SQLite.

For my sample database, I will create a table to store some basic information on books, like the book title, author, pages, selling price, royalties, and publisher. Figure 2-8 provides the basic details of the table. The screenshot immediately following the table structure provides the details of the actual query string being used to create the table in the database (Figure 2-9). You could easily do a copy + paste in another iOS application if you weren't sure of the proper syntax or if you were lazy like I am sometimes and didn't feel like writing the code.

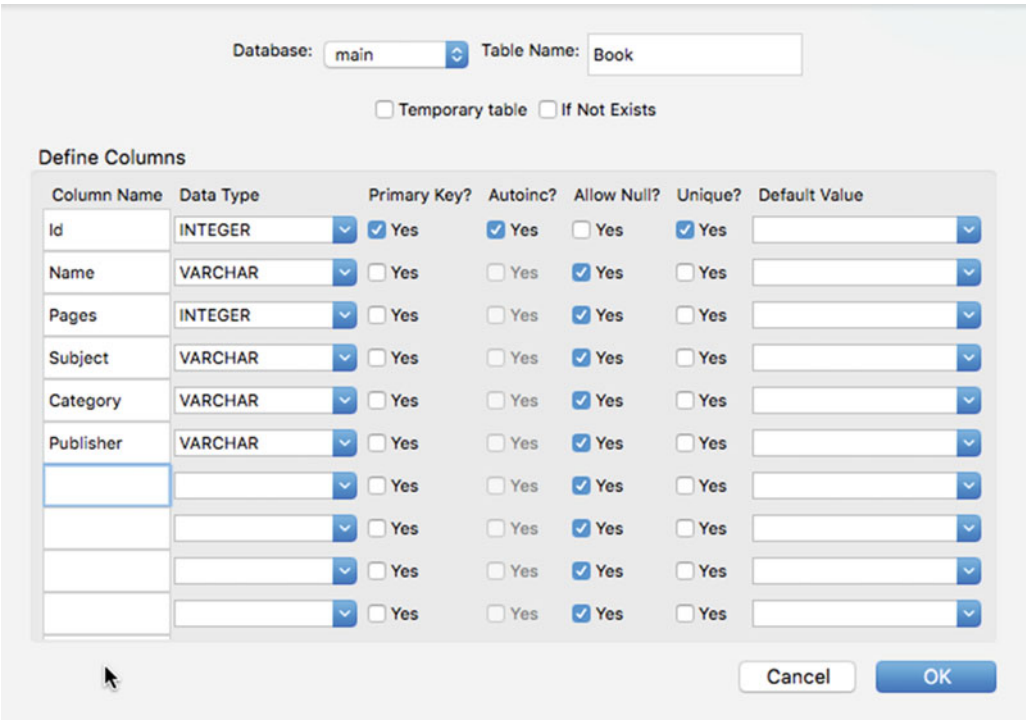

Figure 2-8. *Create Table definition*

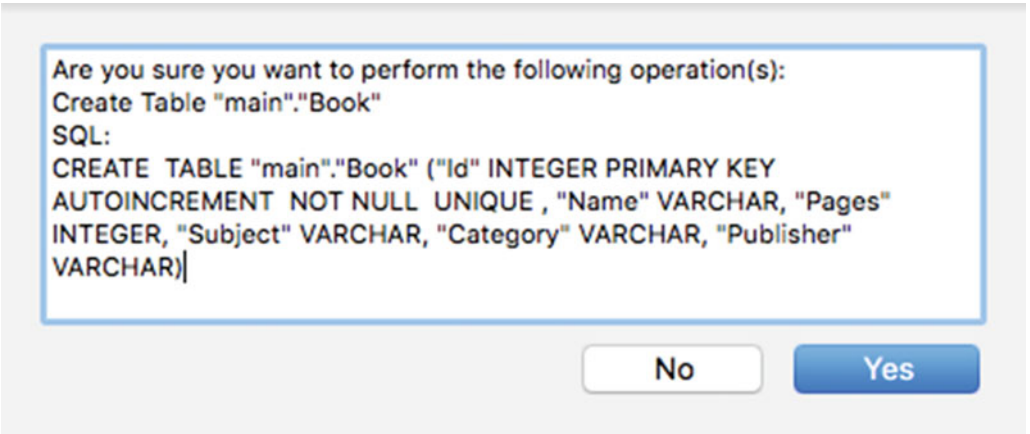

Figure 2-9. *Create Table query*

In SQLite Manager, you can browse the table structure by selecting the table name and expanding the node in the left pane, then selecting the Structure tab along the top in the pane on the right-hand side (Figure 2-10). In addition to browsing the table structure, you can also perform some operations, like dropping the table or re-indexing the table.

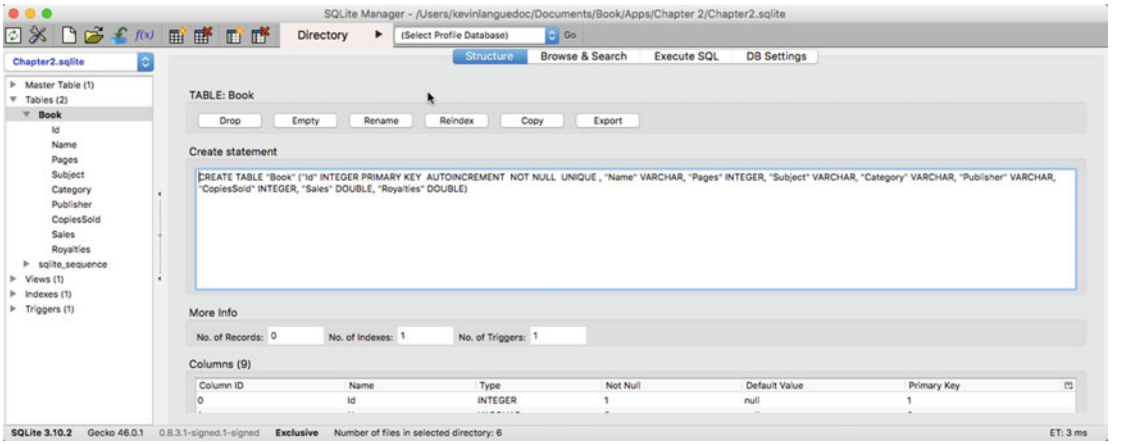

Figure 2-10. *Browse database structure*

Notice in the screenshot that an index was automatically added under the Indexes node. This index got created as a result of my specifying a primary key in my Book table. SQLite also added a `sqlite_sequence` table, which I mentioned before, to maintain a handle on the latest `ROWID`. Although SQLite created the index for us, I'll show you the syntax of the SQLite query needed to create your own.

Add an Index

In a SQLite database, as in all other relational databases, tables have indexes to help them locate records quickly. Creating indexes in SQLite is very straightforward, especially with SQLite Manager. Figure 2-11 shows the `Chapter.sqlite` schema in SQLite Manager with one index, `sqlite_autoindex_book_1`. This index is automatically created when a column is configured with the autoincrement property when creating a table.

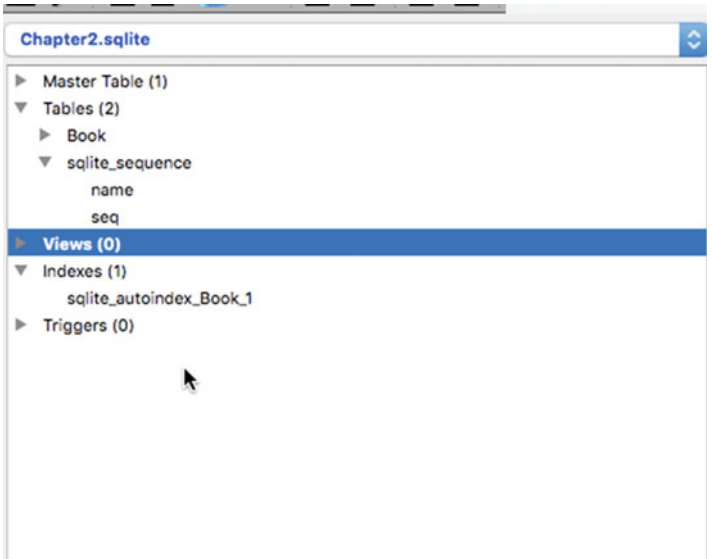

Figure 2-11. *Indexes in SQLite Manager*

To create an index, select the Create Index command from the Index menu in SQLite Manager. The interface window provides the necessary fields to build the `Index` query in SQLite. You need to provide the name of the target table. In Figure 2-12, the Book table is preselected, since it is the only one in the database other than the `sqlite_sequence` table. In the Define Index Columns section, which displays the available columns of the selected table, you can define which columns are needed for the index. Click OK to generate the query shown in Figure 2-13.

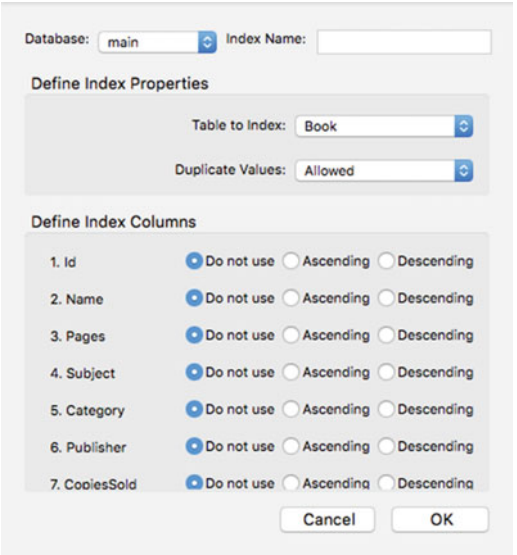

Figure 2-12. *Create Index interface in SQLite Manager*

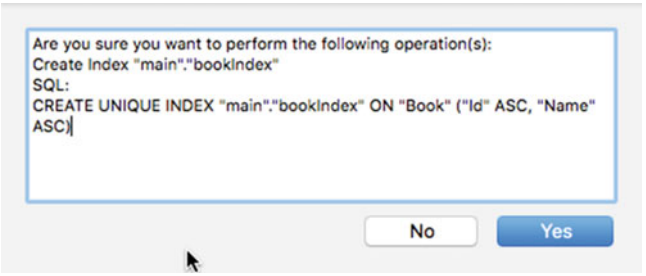

Figure 2-13. *Create Index query*

With the index now in place for the table, I will proceed with adding a view and a trigger before adding the finished database to an iOS project.

Add a View

A view provides a listing based on a `SELECT` statement run against the contents of a table. The view may present all the data in a table or a subset of data depending on your needs. A view can provide better performance when accessing large datasets if you are limiting yourself to a subset of the entire table's contents.

The View menu in SQLite Manager provides several operations that you can perform on a view in addition to creating a view. For instance, you can opt to modify a view, drop a view, or rename a view. For now, select the Create View command from the View menu. However, in the case of modifying or altering a view, in reality you would be performing a drop view/create view operation. You can only alter, in the true sense of the word, a table in SQLite. Figure 2-14 illustrates the Create View interface. In the following screenshot, you can see where to add the view name and the target database. You can also indicate if it is temporary or that SQLite should check to see if the view already exists. The Select Statement field lets you define the subset of records to view through the view.

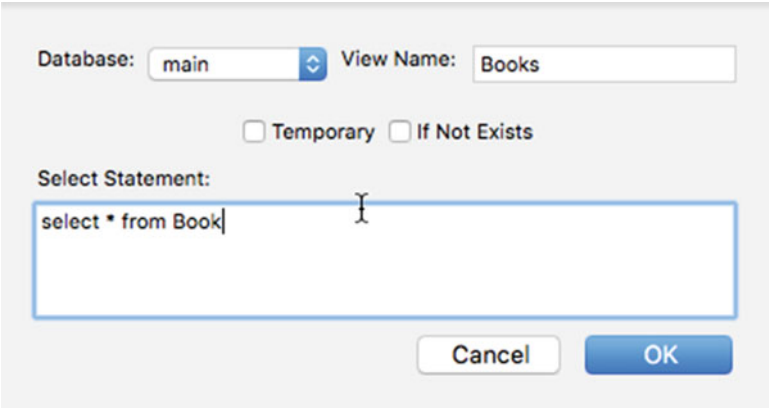

Figure 2-14. *Create View query interface*

Figure 2-15 provides a confirmation of the query that will be used to generate the view. For this example, I am selecting all the records, but I could have easily created a subset of content by modifying the `SELECT` query accordingly. Click the OK button, and the SQLite database engine will provide the finished `CREATE VIEW SQL` query string to create the view.

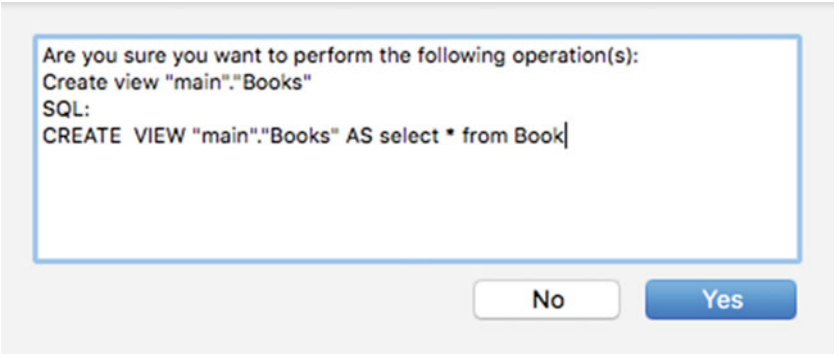

Figure 2-15. *Create View query confirmation*

Views are a handy tool for accessing subsets of records in a table or tables, since you could add a `JOIN` or even an inline `SELECT` statement to build a more complex `SELECT` statement.

Add a Trigger

The trigger is the final database element I will demonstrate before adding the database to the iOS project. Triggers act on data in a table after a record is inserted, updated, or deleted. These database programs are useful for performing a given operation or generating logs in case you need to maintain an audit trail.

Triggers can be created from the Trigger menu in SQLite Manager or from the context menu by right-clicking on the Trigger node, like all the other database schema elements, as shown in Figure 2-16.

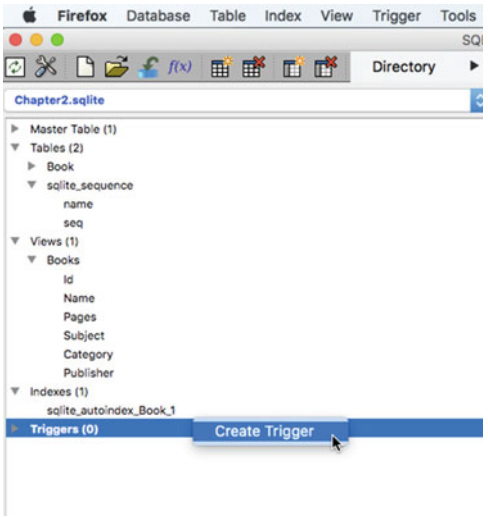

Figure 2-16. *Create Trigger context menu*

The Trigger interface window provides the necessary fields to define the trigger (Figure 2-17). You need to specify a name and the target table or view. You also need to indicate if the trigger is launched before, after, or instead of the creation of a record, as well as which database event the trigger is the result of, like `UPDATE`, `INSERT`, or `DELETE`.

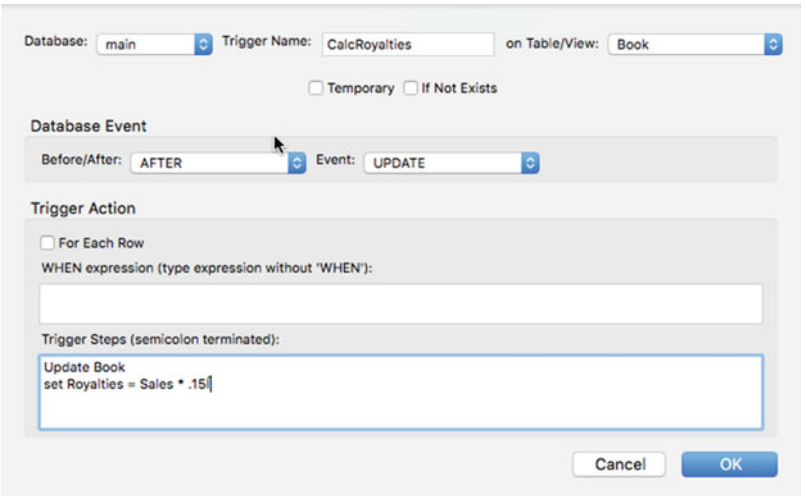

Figure 2-17. *Create Trigger interface*

Next, you need to write the query to be performed on the records. In this example, I want the trigger to calculate the royalties an author will receive on book sales. I have a provided the code snippet for the example trigger `CalcRoyalties` (Figure 2-18).

```
CREATE TRIGGER CalcRoyalties  AFTER  INSERT ON Book
FOR EACH ROW
WHEN (Sales) >= 1
BEGIN
 update Book set Royalties = Sales * .15;

END
```

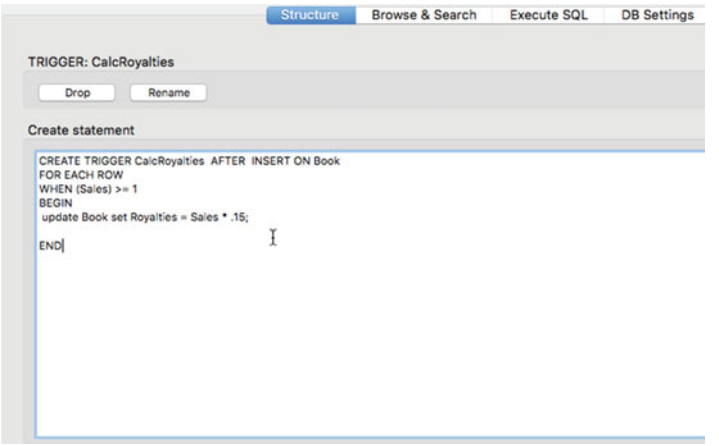

Figure 2-18. *Create Trigger query*

Once the trigger query is to your liking, you can add the trigger to the database by clicking the OK button. As usual, SQLite Manager will confirm the `CREATE TRIGGER` action before actually adding it to the database.

Create an iOS Project

To complete this section of the chapter on developing SQLite databases using a database tool, I will create an iOS project. When you add a database to an iOS project, it is inserted in the Resources directory. The Resources directory is also the root directory of the project. This directory is read-only, and you cannot change these file permissions.

If you try to write data to the database while in this location, you won't receive any errors, but your `INSERT` or `UPDATE` queries will fail. To make your database writable, you need to move it to the Document directory. I will show you how to do this in the following sections.

From within Xcode, create a new iOS project. For this chapter, I am creating a database editor, so I will need a Master-Detail template. I am naming the project Db Mgr as an abbreviation for Database Manager. The language is Swift, of course, and the target device is iPad. Accept the other defaults, but make sure that "Core Data" is unchecked.

■ **Note** Did you know that Apple uses SQLite behind the scenes for Core Data?

Add Database to the Project

Adding the database to the project is simple. From the File menu in Xcode, select the Add Files to "Db Mgr ..." command. A Finder window will open requesting the file location of the database. If you don't remember where you saved the database file, you can go back to SQLite Manager and use Directory➤Select Default Directory to identify the location of the directory where the SQLite database file is located.

Select the file and click the Add button. By default, the file is added to the selected directory or group, if you selected a group. You can drag and drop the database file anywhere in the Explorer. However, as I mentioned earlier, the SQLite database file is read-only in this location. You need to copy or move the file to the Documents directory to ensure that it's writable.

For this example, I will add code to the AppDelegate application's `didFinishLaunchWithOptions` method to copy the database file to the Documents directory. The code is provided here:

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[_ NSObject: AnyObject]?) -> Bool {

//....Code remove for brevity   ....
        var srcPath:URL
        var destPath:URL
        let dirManager = FileManager.default()
        let projectBundle = Bundle.main()
        do {
            let resourcePath = projectBundle.pathForResource("Chapter2", ofType: "sqlite")
            let documentURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)

            srcPath = URL(fileURLWithPath: resourcePath!)
            destPath = documentURL.appendingPathComponent("Chapter2.sqlite")

            if !dirManager.fileExists(atPath:destPath.path!) {
               try dirManager.copyItem(at: srcPath, to: destPath)
            }

        } catch let err as NSError {
            print("Error: \(err.domain)")
        }

    }
```

This code will run every time the app is started and will check to see if the file is located in the Documents directory. If it's not, then it is copied. Ideally, the code should check to see if the file in the bundle is newer than the version in the Documents directory and update accordingly.

There are different ways you could handle this, like having a button in the UI that provides the syntax to select the file and copy it to the Documents directory. Or, you could have the same code in the `viewDidLoad` method of the main `ViewController` that is loaded after the app is started.

Summary

This completes this chapter on creating SQLite databases and adding them to an iOS project. The next chapter focuses on creating SQLite databases during runtime. I will continue to add functionality to the Db Mgr iPad app.

CHAPTER 3

Creating Databases During Runtime

One of the great features of SQLite is its ability to create databases while the app is running. You can quickly add or modify databases as needed.

In the following sections I will add a data model, a UI for various databases commands and to display the database schema, and the view controllers to handle the operations. The app will be able to create the database and add tables and columns as well as indexes, views, and triggers.

Building the DB Mgr App

I will continue to build on the same iPad project from the previous chapter in order to provide the functionality needed to be able to create databases while the app is loaded into memory. I won't win any design awards with the app, but it will serve its purpose of reinforcing some of the key features of SQLite on how to build databases.

At a high level, the app was conceived to use the `detailViewController` as the main interface. The `MasterViewController` will retain and display the databases in the app along with schema elements. The `DetailViewController` will include buttons to create and open databases and to create tables, views, indexes, and triggers. The `ViewController` will also provide an action to execute queries.

A data-access class will handle the communication between the data model, which includes the SQLite database and a custom type as, and the ViewControllers.

The Application UI

In this section, I'll add `UIButtons` and connect them to the assigned view controller. Then, I'll add the `detailSQLiteQueryField`.

Adding the Buttons

Adding buttons and connecting them to the assigned view controller is an easy task with the Interface Builder, or IB for short. To add the buttons, open the main storyboard in the app (`Main.storyboard`) and locate the detail scene.

From the Component palette on the lower right side, drag and drop `UIButtons` and align them along the top. Figure 3-1 shows the app with the buttons and the `detailSQLiteQueryField`. The buttons use .gif images. I have added all these under the Assets group in the project. If you need to add more or change some, you can do so by using the Add Files context menu or the File menu in Xcode.

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_3

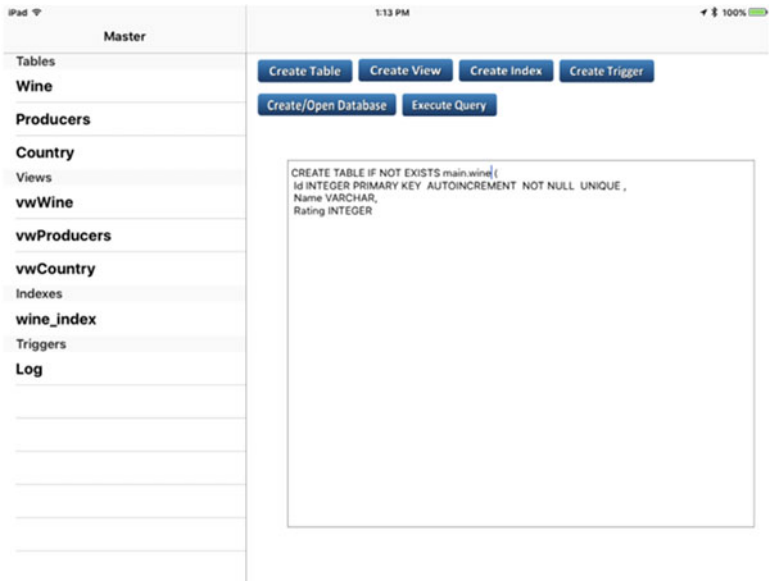

Figure 3-1. *Db Mgr user interface*

To configure the image for the button, do the following:

- Select the button in the detail scene.
- From the Attributes inspector, locate the image field.
- Select an image using the file-selector button.

Once all the buttons are added and configured, we will need to create `@IBActions` for the buttons in the `DetailViewController`. To create the connections, open the Assistant Editor from the open `Main.storyboard` file and drag and drop a connection from the `UIButton` to the open `DetailViewController`. When you release the mouse button, a popup appears (Figure 3-2) through which you can do the following:

- Add the `IBAction` connection name.
- Select the connection type.
- Click on Connect to create the `IBAction` function and create a connection (dark dot).
- Repeat this process with all the buttons, naming them as follows:
    - `createTableButton`
    - `createDbButton`
    - `createViewButton`
    - `createIndexButton`
    - `createTrigger`
    - `executeQueryButton`

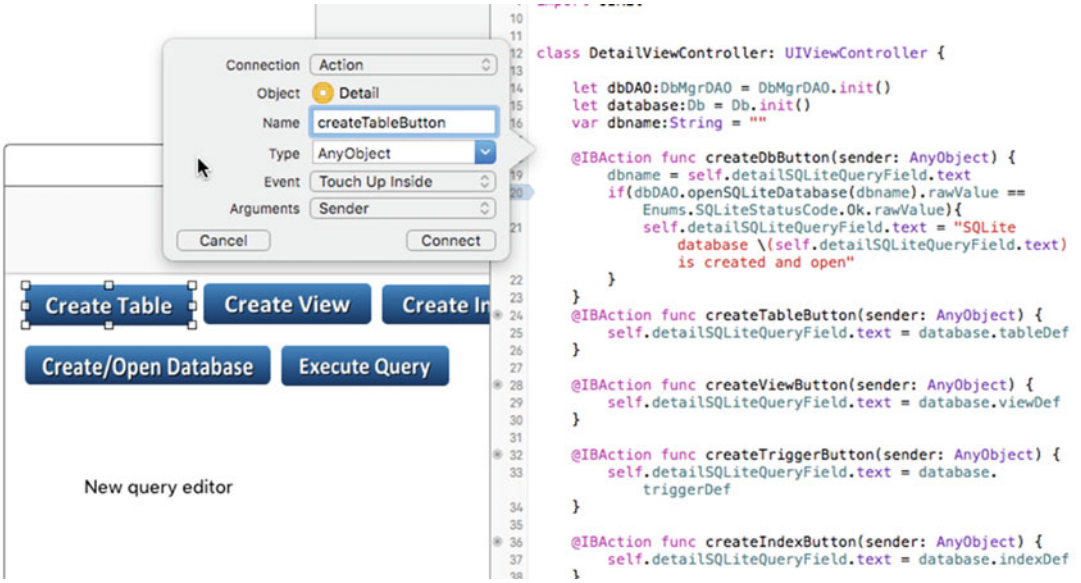

Figure 3-2. *Create IBAction button*

We will add the processing logic later when we discuss the `DetailViewController`.

Adding the detailSQLiteQueryField

The `detailSQLiteQueryField` is a `UITextArea` component that is also available from the Component palette in Xcode. It is used to edit the database structure SQL queries and to display the SQL queries that were used to create existing database schemas.

You can also create a SQLite database by entering the name of the database with the .sqlite extension and clicking on the Create/Open Database button. Query confirmation messages and error messages are also displayed in this field.

In order to interact with the `DetailViewController` and the `DbMgrDAO` controller, we will need to add an `IBOutlet` connection to the controller like we did with the buttons.

With the `detailSQLiteQueryField` selected in the storyboard, do the following:

- Open the Assistant Editor.
- Drag and drop a connection.
- Name the outlet `detailSQLiteQueryField`.
- Make sure the Connection type is `IBOutlet` and click Connect.

Creating the Data Model

The model contains the `Db` Swift class, which is encapsulated in the `Db.swift` file. The class is designed with the same fields as in the `sqlite_master` table, namely type, name, and sql. There are also constants defined to hold the query templates that will be modified to architect the database. Figure 3-3 provides a snapshot of the `Db` class.

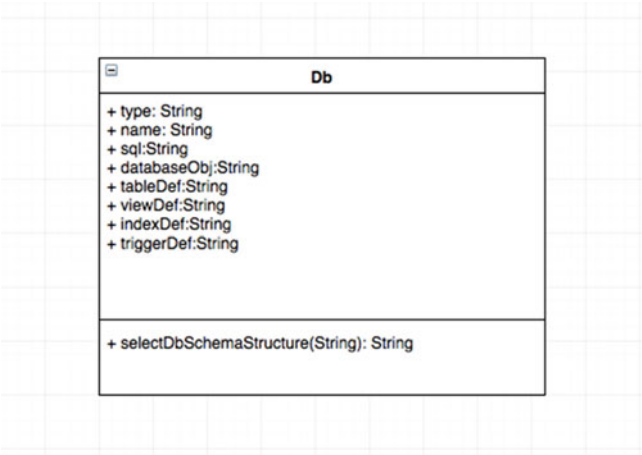

Figure 3-3. *Db data model*

The templates could be contained in a SQLite database, which would improve the design of the app, but I added them to constants to keep the concept of the app simple. I will use the templates later to create a wine database that I will use in the next few chapters to show you how to create CRUD queries.

These templates are basic examples of how to write queries to create a database schema. For instance, you could create a table using an `INSERT` statement rather than a `CREATE TABLE` statement. You could also use `CREATE TABLE` along with a `SELECT` statement to create a table based on a `SELECT` query definition. Tables can also be temp, or temporary, as with other SQL systems.

After creating the `Db.swift` file using the Swift file template, add the class definition, starting with the public access identifier, followed by the class keyword and class name. As you can see from the code listing that follows, the `Db` class resembles the data model, and you can see the assignment for the Create Table template, `tableDef`. There are also SQL templates for views, indexes, and triggers, all assigned to their respective constants.

```
import Foundation

public class Db{
    // These properties will populate array for MasterViewController
    var type:String = ""
    var name:String = ""
    var sql:String = ""
    var databaseObj:String = ""

    // QUERY TEMPLATES
    let tableDef:String = "CREATE TABLE IF NOT EXISTS main.tablename ( \n " +
                    "Id INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL  UNIQUE ,\n " +
                    "colVarchar VARCHAR, \n " +
                    "colInt INTEGER, \n " +
                    "colDouble DOUBLE, \n " +
                    "colBool BOOL, \n " +
                    "colFloat FLOAT, \n " +
                    "colReal REAL, \n " +
                    "colChar CHAR, \n " +
                    "colBlob BLOB, \n " +
```

```
                "colDateTime DATETIME, \n " +
                "colNumeric NUMERIC, \n " +
          "colRealStrict REAL check(typeof('colRealStrict') = 'real'), \n " +
          "colIntStrict INTEGER check(typeof('colIntStrict') = 'integer'), \n " +
                "colTextStrict TEXT check(typeof('colTextStrict') = 'text') \n "

let viewDef:String="CREATE VIEW IF NOT EXISTS “ +
                    “ viewname AS SELECT * FROM main.tablename " +
                    " or CREATE VIEW viewname AS SELECT columns FROM main.tablename
                    where column equals value" +
   " or CREATE temp (or temporary) VIEW viewname AS SELECT columns FROM “ +
           “ main.tablename "

let indexDef:String = "CREATE UNIQUE INDEX IF NOT EXISTS main.indexname " +
                    " ON TABLE tablename (Column defintion) WHERE where clause"

let triggerDef:String="CREATE TRIGGER triggername  AFTER INSERT ON main.table" +
                        "FOR EACH ROW " +
                        "WHEN (columnnmae) some condition " +
                        "BEGIN " +
                        " update Book set Royalties = Sales * .15; " +
                        "END"
```

The next part of the Db class includes the methods. The selectDbSchemaStructure method returns the schema of either a specific element or all elements in a database. The selectDbSchemaListByType method returns a list of element names based on a type that is specified though the typeName String parameter. This method is used to build the lists of arrays for the data source of the UITableView in the MasterViewController. The queries are straightforward, SQL-compatible query strings. These will be used later in the executeQuery method.

```
init(){

 }

 func selectDbSchemaStructure(_ objectName:String)->String{
     var def:String = ""
     if(!objectName.isEmpty){
         def = "SELECT type, name, tbl_name, “ +
             “sql FROM main.sqlite_master WHERE name='\(objectName)';"
     }else{
         def = "SELECT type, name, tbl_name, sql FROM main.sqlite_master ;"
     }

     return def
 }

 func selectDbSchemaListByType(_ typeName:String)->String{
     var def:String = ""
     if(!typeName.isEmpty){
       def = "SELECT name FROM main.sqlite_master WHERE type='\(typeName)';"
     }
```

```
        return def
    }

}
```

The Enums class is a small helper class used to generate the SQLite return codes that the various functions (SQLite and Swift alike) will use as an enum. The complete code for the class is provided here:

```
import Foundation
public class Enums{
    enum SQLiteStatusCode : Int32 {
        case ok = 0
        case error = 1
        case internalLogicError = 2
        case accessPermissionDenied = 3
        case abort = 4
        case busy = 5
        case noMemory = 7
        case readOnly = 8
        case interrupt = 9
        case iOError = 10
        case corrupt = 11
        case notFound = 12
        case full = 13
        case cantOpen = 14
        case protocol = 15
        case empty = 16
        case schema = 17
        case tooBig = 18
        case constraint = 19
        case mismatch = 20
        case misuse = 21
        case noLFS = 22
        case authDeniedUTH = 23
        case format = 24
        case range = 25
        case notADatabase = 26
        case row = 100
        case done = 101
    }
```

Creating the Controllers

In this section, I'll be creating three controllers: DbMgrDAO (the main workhorse of the app), MasterView (retrieves and displays a list of databases and its corresponding schemas), and DetailView (the working area of the app). Let's have a look now.

The DbMgrDAO Controller

The Swift class handles the database operations of creating the database, tables, views, indexes, and triggers as needed. The class also fetches records from the open SQLite database that will be displayed in the MasterViewController. Query execution operations are also handled by the class.

DbMgrDAO is a subclass of the standard NSObject, which is a basic class in the Foundation framework. To create the subclass, right click on the controller group and select New File from the context menu. From the iOS Source heading of the available file templates, select the Swift file template. You could create the class by using the CocoaTouch template, which would provide an Input page on which to select the subclass from a list of available classes. However, using this method, the template would add the Import UIKit framework at the top of the class file, while the NSObject class is located in the Foundation framework, so you would need to change that; otherwise, you would receive complaints.

By using the Swift file template, the Foundation framework will be added by means of the import statement, but the template will do little else after you provide a file name and add the file to the project. For this example, I am naming the file DbMgrDAO, of course. Once the file is created, it will be automatically loaded in the editor.

To set up the class, you will need to add the class definition statement below the import instruction. You'll probably notice that I have added the UIKit framework as well as the Foundation framework. The class will classes from this framework to interact with DetailViewController later. You add it now, however.

Directly below the import statements, add the public class DbMgrDao:NSObject class definition statement followed by a curly brace. The Xcode editor will add the closing curly brace for you when you press the Enter key.

```
import Foundation
import UIKit

public class DbMgrDAO:NSObject{

}
```

Immediately following the class definition, I will add some properties variables. The db variable has a COpaquePointer data type. This data type is a wrapper for the C opaque pointer, which is a pointer for unknown data types. SQLite make use of pointers throughout its API. The dbPath variable is an NSURL data type and will be used to hold the path for the stored database that is open. To pass queries to the SQLite database engine, you will need a prepared statement, which is set up using a COpaquePointer as well and is named sqlStatement.

```
    var db:CopaquePointer?=nil
    var dbPath:URL = URL()
    var sqlStatement:CopaquePointer?=nil
    var dbErr: UnsafeMutablePointer<UnsafeMutablePointer<Int8>>?= nil
    var errmsg:String=""

    override init() {
        //code here
    }
```

The populateIndexView Function

Following the init function, I define the populateIndexView function. The first parameter, defined as AnyObject, is actually the DetailViewController. The second parameter is a SELECT query that will be used to populate the TableView in the MasterViewController. The function will return an array of Strings. I could replace this with a custom type property, but this design works fine for my demonstration needs in this app, especially since I am only returning a list of database schema names from the sqlite_master table.

After defining the return array variable, I create a DetailViewController variable and assign it the actual DetailViewController object from the DetailViewController by using the isKindOfClass method.

Next, I check to see if the database is still open and open it if necessary. Otherwise, I send the preparedStatement and query and sqlStatement and query respectively to the database engine for execution. Notice how the Swift string is converted to the char data type that is required by SQLite using the cStringUsingEncoding (NSUTF8StringEncoding) method of the String class. If any errors are thrown, I assign them to my errmsg variable by passing sqlite3.errmsg(db) to fromCString.

The preparedStatement in SQLite is created by using the sqlite3_prepare_v2 function. Once the results are all retrieved, you need to call the sqlite3_finalize function that takes preparedStatement as an argument. Once the sqlite3_prepare_v2 function is executed and the return status code, Enums.SQliteStatus.Code—which you will need to get the Swift enum's rawValue—is equal to OK or 0, you can step through the result set using the sqlite3_step function, as seen in the code that follows. You can loop the values by checking the return code, which should be 0 or OK. For this app, I am creating an instance of the Db class for each iteration and assigning its name property the value of the returned name column in the query. Converting the C string to a Swift string requires some fancy footwork.

To retrieve a column value, you need to use one of the preparedStatements column methods of the corresponding type. Since the name column in the database I will create later has a varchar data type, I need to use the sqlite3_column_text method. This method returns a C unsigned char, which we will need to cast using the unsafePointer with an Int8 primitive type that is a char. This value is then converted to a Swift string using the cString method that we saw before.

■ **Note** The process of binding Swift data to SQLite columns is pretty much the same for the other data types, as I outlined in the previous paragraph. Check out the SQLite Prepared Statement web page (https://www.sqlite.org/c3ref/stmt.html) for the complete list of methods and functions and the corresponding argument requirements and return types.

With the result in hand, I only need to assign the value to the name property of the Db object and append this value to the result-set array.

The array is returned to the calling array variable in the MasterViewController, which will update the TableView's data source and reload the data into the UITableView.

That completes the discussion on populating a data source for the TableView. I demonstrated how to retrieve values from a SQLite database. As I mentioned earlier, we will explore SELECT statements later on in the chapter devoted to it, along with its API.

```
func populateIndexView(_ sender:AnyObject, query:String)->Array<String>{
        var resultset = [String]()
                 let sourceVC:DetailViewController = nil;
                 if( sender.sourceViewController.isKindOfClass(DetailViewController)){
               sourceVC:DetailViewController = sender as! DetailViewController
        }

        if sqlite3_open(dbPath.path!, &db) != SQLITE_OK {
                              errmsg = "error opening database"
                          print(errmsg)
                              sourceVC.detailSQLiteQueryField.text = errmsg

        }
```

```
    let statuscode = sqlite3_prepare_v2(db, query.cString(using: String.Encoding.utf8)!,
    -1, &sqlStatement, nil)

    if(statuscode != Enums.SQLiteStatusCode.ok.rawValue){
        errmsg = String (cString: sqlite3_errmsg(db))!
        print("error preparing select: \(errmsg)")

        if( sender.sourceViewController.isKind(of: DetailViewController.self)){
            let sourceVC:DetailViewController = sender as! DetailViewController

           sourceVC.detailSQLiteQueryField.text = errmsg
        }

    }else{
        while (sqlite3_step(sqlStatement)==Enums.SQLiteStatusCode.row.rawValue){
            let dbVal:Db=Db.init()

             dbVal.name = String (cString: UnsafePointer<Int8>(sqlite3_column_
             text(sqlStatement, 1)))!
            resultset.append(dbVal.name)
        }

        sqlite3_finalize(sqlStatement);
    }

    return resultset
}
```

The initViewIndex function

The next Swift function we will look at is `initViewIndex`. This function has a singular duty: to peruse the Documents directory and retrieve a list of SQLite databases. This list is inserted into an array called `sqliteDbs` and is returned to the calling object. This function is called when the app is loaded from the `viewDidLoad` function in the `MasterViewController`.

```
func initViewIndex()->Array<String>{
    var sqliteDbs:[String]=[]
    let documentsDir =  FileManager.default.urlsForDirectory(.documentDirectory,
    inDomains: .userDomainMask).first!

    do {
        let directoryUrls = try  FileManager.default.contentsOfDirectory(at:
        documentsDir, includingPropertiesForKeys: nil, options: FileManager.
        DirectoryEnumerationOptions())

        sqliteDbs = directoryUrls.filter{$0.pathExtension! == "sqlite" }.map{
        $0.lastPathComponent! }
```

```
        } catch let error as NSError {
            print(error.localizedDescription)
        }
        return sqliteDbs

    }
```

The logic in the function is pretty easy. It starts by getting a handle on the Documents directory through the `NSFileManager.default` class and function. Afterward, it attempts to retrieve all the contents of the Documents directory, including any sub-directories, by using the `contentsOfDirectoryAtURL` method in the `NSFileManager` class.

A filter is applied to the list so as to return only the files with a .sqlite extension (I am taking a leap of faith here that all SQLite databases have a .sqlite extension, which may not be the case in a production environment). The map is used to extrapolate only the last part of the returned paths.

The filter and map design have been reworked in the latest version of Swift to apply a `mapReduce` pattern of sorts. The design pattern eliminates the need to iterate through an array, hence it is much more efficient. The filter uses a Boolean expression to test and return the reduced list. The `map` function is used to extract or perform a transformation on portions of a collection that has been filtered.

The executeQuery function

The `executeQuery`, as its name implies, executes SQLite queries. I make extensive use of this function in association with the Execute Query button in the `DetailedViewController` that we will explore later. The function follows a path similar to that of the other functions that perform SQLite operations—namely, it ensures the database is open or opens it as needed using the `sqlite3_open` command. If it can't open the database—or create it, for that matter—it returns an error message that will be displayed in the Editor field in the `DetailViewController`.

```
func executeQuery(_ sender:AnyObject, query:String?)->String{
let sourceVC:DetailViewController = nil;
                if( sender.sourceViewController.isKindOfClass(DetailViewController)){
               sourceVC = sender as! DetailViewController
        }
        var statuscode:integer_t=0
        //There is no error checking but you should have it in a production app
        //You should see if database is present and open also.
        if sqlite3_open(dbPath.path!, &db) != Enums.SQLiteStatusCode.ok.rawValue {
            errmsg = "error opening database or database does not exist"
        }else{
            let query = query!.replacingOccurrences(of: "\n", with: "")
            statuscode = sqlite3_exec(db, query.cString(using: String.Encoding.utf8)!, nil,
            nil, dbErr);
            if(statuscode != Enums.SQLiteStatusCode.ok.rawValue){
                errmsg = String.fromCString(sqlite3_errmsg(db))!
            }else{
                errmsg = "query was successful"
            }
        }
        return errmsg
    }
```

Next, the function preps the query string by removing the newline character \n using stringByReplacingOccurencesOfString, because, if you remember, the templates that I defined included the newline character, as I wanted them to display properly in the SQL editor, which I will show later. Of course, if the query string doesn't include any of these characters then this line of code won't do anything.

The important line of code is the sqlite3_exec command. This command is really great, since it encapsulates sqlite3_prepare_v2, sqlite3_step, and sqlite3_finalize. I tend to use this command when I am doing inserts, updates, or deletes and opt to use the longer version for selects simply because I find I have more control to map the retuning columns to a custom data type. Using sqlite3_exec with a select query is possible; however, you need to provide a callback function in order for the third argument to handle the result set. The command returns a status code, which I then use to update the editor on the status of the transaction.

The openSQLiteDatabase Function

The openSQliteDatabase function is a simple function and is similar to some of the other functions. Its sole duty is to create the database and open it using a user-supplied file name from the editor (UITextArea) field in the DetailViewController. As we have seen before, the SQLite database is created in the Documents directory, since this is one of the few places in the apps sandbox that has read/write permissions. If you create the file in the root of the app, it will automatically be read-only, and, to make matters worse, you or your app users won't receive any error messages saying that it not writable.

```
func openSQLiteDatabase(databaseName:String)->Enums.SQLiteStatusCode{

    //SQLite database is always created in Documents directory
    //I am assuming that the database name has the .sqlite extension for the sake of
      simplicity
    //Also, you would need to check if databaseName is not empty
    let dirManager = FileManager.default()
    do {
        let directoryURL = try dirManager.urlForDirectory(FileManager.
        SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
        userDomainMask, appropriateFor: nil, create: true)

        dbPath = try! directoryURL.appendingPathComponent(databaseName)
    } catch let err as NSError {
        print("Error: \(err.domain)")
    }
    return Enums.SQLiteStatusCode(rawValue: sqlite3_open(dbPath.absoluteString.cString
    (using: String.Encoding.utf8)!, &db))!
}
```

The function gets a handle on the Document directory and appends the name of the database file to the path using URLByAppendingPathComponent. Then the sqlite3_open command is used to create the database, open it, and return the status code. Notice how the path has to be converted to the char data type that SQLite understands using absoluteString.cStringUsingEncoding and the UTF8 encoding.

That completes the logic for the DbMgrDAO class. The next two sections will deal with setting up the MasterViewController, the DetailViewController, and, finally, the UI.

The MasterViewController

The MasterViewController uses a UITableView component. The TableView displays both databases as well as their corresponding schemas in a multiple-section view. As databases are created, they are added to the TableView's data source, and the display is reloaded. When a database file is selected, the corresponding schema is subsequently retrieved from the database using a SELECT query and is displayed in the section associated with either a table, view, index, or trigger schema element.

There are a number of functions that need to be implemented for the UITableView, its data source, and its delegate, along with several custom functions.

```
import UIKit

class MasterViewController: UITableViewController {

    var detailViewController: DetailViewController? = nil
    var objects = [AnyObject]()
        //inmutable array
    let schemaSections = ["Databases", "Tables", "Views", "Indexes", "Triggers"] var
    schemaDetailsItems=[[String]]()
    var tableArray:[String] = []
    var viewArray:[String] = []
    var indexArray:[String] = []
    var triggerArray:[String] = []
    var dao:DbMgrDAO = DbMgrDAO.init()
```

Let's start at the beginning with the custom properties. The first variable is detailViewController in order to get a handle on the DetailViewController so that I may pass status messages back to the editor without going through the segue, as is the normal documented route. Yes, that route is bi-directional. The objects variable is a catch-all bucket for an array for every type of object. Then, I have the schemaSection immutable array to define the table sections for the databases and corresponding schemas. schemaDetailsItems is a mutable array for the schemas categorized by tables, views, indexes, and triggers. The tableArray, viewArray, indexArray, and triggerArray, as you might suspect, are the arrays for the section details. It is almost like creating a JSON data structure, since these last four arrays will be appended to the schemaDetailsItems array, as will be seen in the didViewLoad function. This variable is an instance of the DbMgrDAO class. I will use it to call the various functions I mentioned in the previous section.

The viewDidLoad Function Implementation

```
override func viewDidLoad() {
        super.viewDidLoad()

        schemaDetailsItems.append(dao.initViewIndex())
        schemaDetailsItems.append(tableArray)
        schemaDetailsItems.append(viewArray)
        schemaDetailsItems.append(indexArray)
        schemaDetailsItems.append(triggerArray)

        if let split = self.splitViewController {
            let controllers = split.viewControllers
```

```
            self.detailViewController = (controllers[controllers.count-1] as!
            UINavigationController).topViewController as? DetailViewController
        }
    }
```

From the preceding code, you can extrapolate the knowledge of how to populate the `UITableView` data source in the `MasterViewController`. First, `initViewIndex` is called when the app is loaded from the `AppDelegate` object, which is called when the app is launched from the `main.swift` file. The results of `initViewIndex` are appended to the `schemaDetailsItems` array. I also append empty arrays for the other sections; otherwise, the Swift compiler complains that the other sections are empty or non-existent.

The remaining code sets up the `splitViewController`, the navigation, and making the `detailViewController` the initial or top controller. You don't need to change or touch this.

I am presenting one option, or technique, on populating a `TableView`. However, you can also set up a `UITableViewCell` that can act as the controller for the cell. In my case, I am using two data sources to build the main data source, so it would be hard to define a cell controller, but not impossible.

I won't touch on the functions that I am not using but that are implemented as part of the `UITableView` implementation, namely `viewWillAppear` and `didReceiveMemoryWarning`.

The Data Source Functions' Implementations

The following functions are required in order to set up and display the data in the `TableView`. The `titleForHeaderInSection` returns the Section Title, so, for this app, this would be Databases, Tables, Views, Indexes, and Triggers. The `numberOfSectionsinTableView` function will how many sections the `TableView` will contain. For this app, it will be 5, which is the number of elements in the `schemaSections` array. `numberOfRowsInSection` configures the number of rows to display. This is usually the count of elements in the array that is serving as the data source. Since I am using nested arrays, I will use the number of rows per section as `schemaDetailsItems [section].count`. The last required function, `cellForRowAtIndexPath`, defines the cell display. Here, I am setting the cell to display the value found in the data source at the `IndexPath.section` of `schemaDetailsItems`:

```
    override func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) ->
    String?
        return self.schemaSections [section ]

    }

    override func numberOfSections(in tableView: UITableView) -> Int {
        return self.schemaSections.count
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) ->
    Int {
        return schemaDetailsItems [section].count
    }

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
        let cell = tableView.dequeueReusableCell (withIdentifier: "Cell", for: indexPath)
```

```
        cell.textLabel?.text = self.schemaDetailsItems[(indexPath as NSIndexPath).section]
        [(indexPath as NSInpexPath).row]

        return cell
    }
```

When you need to navigate between scenes, you do so by defining threads called *segues* between the various scenes. When the app moves from one view (scene) to another, the `prepareForSeque` function is called. This function is very important, as it also provides a transport for data and objects to the next `ViewController`. However, `prepareForSegue` can only be used in one direction. If you want to return to the calling `ViewController`, you need unwind from the scene, which entails creating an `unWindFromSegue` function. The concept is similar to backing out of the history path in an Internet browser. However, with unwinding you can only unwind one scene at a time and in the order that you moved forward.

I have opted to use the `tableView didSelectRowAtIndexPath`, which returns the selected row. This method is provided through the delegate. The `selectDb` variable stores the selected value at the specified row index. After checking to ensure that the database is open, I query the `sqlite_master` table using a `SELECT` statement and assign the results to individual mutable arrays. I then append these arrays to the `schemaDetailsItems` array after I remove any existing items. If you don't, you will get a very long array with each click, and your items won't display in the `TableView` since this array won't line up with the `schemaSections` array. Once the data source is set, I reload the `TableView` to refresh the view with the new data.

```
    override func tableView(tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

        var selectdb = schemaDetailsItems[0][(indexPath as NSIndexPath).row]

        if(dao.openSQLiteDatabase(selectdb[0]).rawValue == Enums.SQLiteStatusCode.
        ok.rawValue){
            let db:Db = Db.init()
            let tables:String = db.selectDbSchemaListByType("table") as String
            let views:String = db.selectDbSchemaListByType("view") as String
            let indices:String = db.selectDbSchemaListByType("index") as String
            let triggers:String = db.selectDbSchemaListByType("trigger") as String

            tableArray = dao.populateIndexView(self, query:tables)
            viewArray = dao.populateIndexView(self, query:views)
            indexArray = dao.populateIndexView(self, query:indices)
            triggerArray = dao.populateIndexView(self, query:triggers)
        }

       schemaDetailsItems.removeAll()
        schemaDetailsItems.append(dao.initViewIndex())
        schemaDetailsItems.append(tableArray)
        schemaDetailsItems.append(viewArray)
        schemaDetailsItems.append(indexArray)
        schemaDetailsItems.append(triggerArray)
        tableView.reloadData()
    }
```

The DetailViewController

The `DetailViewController` contains buttons to create and open a SQLite database and load SQL templates as well as to execute the queries to create the database schema.

The following code is the app's implementation of the processing logic as it relates to the interaction with the user, the controllers, and the model. Since the `DetailViewController` is a subclass of the `UIViewController`, there are several functions that are provided with the class.

From the code, you'll notice that I create an instance of the `DbMgrDAO` class as well as an instance of the `Db` class. I call their `init` functions to initiate the classes.

The createDbButton @IBAction

The `createDbButton` is an `@IBAction`. As we have seen in the section on building the UI, the function is connected to the `UIButton` in the UI Detail Scene. I assign the value from the `detailSQLiteQueryField` to the `dbname` constant. This value will be passed to the `openSQLiteDatabase` method that we previously discussed to create and open the database file, thus creating a connection with the database. If the return value is OK or 0, I will display a success or confirmation message in the same `detailSQLiteQueryField`.

```
//
//  DetailViewController.swift
//  Db Mgr
//
//  Created by Kevin Languedoc on 2016-05-20.
//  Copyright © 2016 Kevin Languedoc. All rights reserved.
//

import UIKit

class DetailViewController: UIViewController {
    let dbDAO:DbMgrDAO = DbMgrDAO.init()
    let database:Db = Db.init()
    var dbname:String = ""
    var dbStatusMsg:String = ""

    @IBAction func createDbButton(_ sender: AnyObject) {
        dbname = self.detailSQLiteQueryField.text
        if(dbDAO.openSQLiteDatabase(dbname).rawValue == Enums.SQLiteStatusCode.ok.rawValue){
            self.detailSQLiteQueryField.text = "SQLite database \(self.
            detailSQLiteQueryField.text) is created and open"

            if let delegate = self.delegate {
                delegate.didOpenDatabase(self, databaseName: dbnane)
            }

        }
    }
```

That is pretty much it for the button. Let's look at the remaining `IBActions` functions that are connected to the buttons in the Detail scene next.

The Create Database Schema @IBAction Buttons

The first four functions are very simple. They display the corresponding SQL queries for the database elements that can be created in a SQLite database: tables, views, triggers, and indexes. The fifth function calls the executeQuery function, passing the contents of the detailSQLiteQueryField to it. The returned message is assigned to the same field.

Each of the button functions is assigned either the tableDef, viewDef, triggerDef, or indexDef query templates from the database object, which is an instance of the Db class.

```
@IBAction func createTableButton(_ sender: AnyObject) {
    self.detailSQLiteQueryField.text = database.tableDef
}

@IBAction func createViewButton(_ sender: AnyObject) {
    self.detailSQLiteQueryField.text = database.viewDef
}

@IBAction func createTriggerButton(_ sender: AnyObject) {
    self.detailSQLiteQueryField.text = database.triggerDef
}

@IBAction func createIndexButton(_ sender: AnyObject) {
    self.detailSQLiteQueryField.text = database.indexDef
}
@IBAction func executeQueryButton(_ sender: AnyObject) {
    self.detailSQLiteQueryField.text =  dbDAO.executeQuery(self.detailSQLiteQueryField.
    text)
}
```

The detailSQLiteQueryField @IBOutlet

The detailSQLiteQueryField is an IBOutlet variable associated with the UITextArea component of the same name in the Detail scene. It is the main work area of the UI, loading and editing queries as well as providing the SQLite database file to create and/or open.

The variable's signature is completely defined through the Interface Builder (IB). Keep in mind that you can create the same functionality programmatically.

```
@IBOutlet weak var detailSQLiteQueryField: UITextView!
```

Set Up the View

When the Detail scene is first loaded after the app is launched, the detailItem and the configureView functions are called from the viewDidLoad function. These are also called when prepareForSeque is called from the MasterViewController.

The modifications I did here were to replace the UITextfield that is included with the Master-Detail template with the detailSQLiteQueryField in order to receive information through the prepareForSegue function, and to add the code to open the selected database from the MasterViewController TableView, passing the value of detail.description. The status message is assigned to a variable dbStatusMsg instead

of being passed directly to the `detailSQLiteQueryField`, because the field is not yet created when the function is called. Instead, I assign the value of `dbStatusMsg` to the field in the `viewWillAppear` function.

```
    var detailItem: AnyObject? {
        didSet {
            // Update the view.
            self.configureView()
        }
    }

func configureView() {
        if let detail = self.detailItem {
            if(dbDAO.openSQLiteDatabase(detail.description).rawValue == Enums.
            SQLiteStatusCode.ok.rawValue){
                dbStatusMsg = "SQLite database \(detail.description) is created and open"
            }
        }
    }
```

The viewDidLoad Function Implementation

I replaced all the code that is included in the `viewdidLoad` function, except for the `self.configureView`, with some setup code to set up the `detailSQLiteQueryField`.

```
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        // Update the user interface for the detail item.
        self.detailSQLiteQueryField.layer.borderWidth = 1.0
        self.detailSQLiteQueryField.layer.borderColor = UIColor.gray().cgColor
        self.configureView()
    }
```

The viewWillAppear Function Implementation

When the `viewWillAppear` function is called, I assign the value of `dbStatusMsg` to the `detailSQLiteQueryField`, thus ensuring that the field is actually available.

```
   override func viewWillAppear(_ animated: Bool) {
        self.detailSQLiteQueryField.text = dbname
    }
```

Building the Winery Database

To finish this chapter, I want to demonstrate creating a database, which I will use for chapters 5–9. These chapters will perform CRUD operations using an app for wines.

Create the Winery.sqlite File

The task is really easy. Enter the `Winery.sqlite` filename in the `detailSQLiteQueryField` and click on the Create/Open Database button (`createDbButton`). I provided code snippets from the `createDbButton` function in the `DetailViewController` as well as the `openSQLiteDatabase` function in the `DbMgrDAO` class for reference.

In the `createDbButton` function, the `openSQLiteDatabase` is called, passing the file name `Winery.sqlite` for its required parameter. You have probably noticed that there isn't any error checking beyond what is required by the SQLite functions. Also, I optimistically assume that the return code will be 0 or successful and display a successful creation message.

```
@IBAction func createDbButton(sender: AnyObject) {
        dbname = self.detailSQLiteQueryField.text
        if(dbDAO.openSQLiteDatabase(dbname).rawValue == Enums.SQLiteStatusCode.ok.rawValue){
            self.detailSQLiteQueryField.text = "SQLite database \(self.
            detailSQLiteQueryField.text) is created and open"
        }
    }
```

In `openSQLiteDatabase`, you may recall that I get a handle on the Documents directory because it is one of the few places in an app's sandbox that is writeable, and then I call the `sqlite3_open` SQLite function to create and/or open the database at the requested path. If the file name at the requested path doesn't exist, then `sqlite3_open` will create the database file and open it; otherwise, the function will attempt to open the database file, and by doing so establish a connection.

```
func openSQLiteDatabase(databaseName:String)->Enums.SQLiteStatusCode{
        let dirManager = FileManager.default()
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)
            dbPath = try! directoryURL.appendingPathComponent(databaseName)
        } catch let err as NSError {
            print("Error: \(err.domain)")
        }
        return Enums.SQLiteStatusCode(rawValue: sqlite3_open(dbPath.absoluteString.cString
        (String.encoding.utf8)!, &db))!
    }
```

With the database in hand, I will now create the tables and views.

Adding the Tables

Recall the Create Table and Create View buttons in the UI? If you click on these, they will load the templates in the `Db` class into the `detailSQLiteQueryField` field to be edited. Once the queries are to your liking, you need only click on the Execute Query button (`executeQueryButton`) to execute the query through the database engine and create the database structure in the open database file.

For the `winery.sqlite` database and app, I will need two tables: wine and producer. I could easily extend this with a few more tables and views and triggers. I won't go into joins or foreign keys here, as I will revisit them in detail in chapter 5 on inserting records. Suffice to say that the two tables will have a foreign-key relationship, and the view will use a join to access data from both for display.

The first table I will create is the `producer` table. This table will contain the data on the wine producers. The second table, `wine`, will contain information on the bottles of wine. The schema is outlined in Table 3-1. The foreign key will be in the `wine` table and is defined by the `Producer_id` field.

Table 3-1. *Winery Schema*

wine	producer
id	id
Producer_id	name
name	country
Rating	region
type	

To create the table in the `winery.sqlite` database, follow these steps:

1. From the running app (iOS Simulator or installed on iPad), enter "winery.sqlite" in the editor field and click on the Create/Open Database button.
2. If the database is successfully opened, you will get a success message back that replaces the text in the same field.
3. Next, click on the Create Table button to load the Create Table template into the editor field.
4. Modify the query so that it looks like this:

```
CREATE TABLE IF NOT EXISTS main.producer (
 Id INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL  UNIQUE ,
Producer_id integer,
 Name VARCHAR,
Country VARCHAR,
Region VARCHAR,
Foreign Key (Producer_id) references producer(id))
```

5. Click on the Execute Query button to create the table. You should receive a "query is successful" confirmation message.
6. Repeat the same process for the `wine` table. The following is the query to use:

```
CREATE TABLE IF NOT EXISTS main.producer (
 Id INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL  UNIQUE ,
 Name VARCHAR,
Country VARCHAR,
Region VARCHAR

)
```

Again, you should get the "query is successful" confirmation message. With the two tables in place, all that remains is the view to display the results. The next section will provide the details for the view.

Adding the View

Since this is a very simple database and app, I will create just one view that will select the columns from each table using a join. The code is as follows:

```
CREATE VIEW IF NOT EXISTS wines AS SELECT w.name, w.rating, w.type, p.producer, p.country,
p.region FROM main.wine w INNER JOIN main.producer p on w.id = p.id
```

SQLIte supports three types of joins. You can use the joins as just seen, or you can use the `USING` (columns array) clause to build the predicate for data selection and relationship definition:

- INNER JOIN
- CROSS JOIN
- OUTER JOIN

Summary

That completes this chapter on creating databases during runtime. We looked at designing and building a database manager app that is capable of executing SQLite queries to create tables, views, indexes, and triggers. We also looked at creating a database with two tables and a view using the database manager app.

In the next chapter, I will add functionality to the Db Mgr app to modify and delete database elements.

CHAPTER 4

Altering Databases and Other Features

SQLite doesn't have an extensive API for modifying databases. Nevertheless, it does still offer functions to alter tables, and we can still alter views, triggers, and indexes as well as functions.

This chapter will focus on demonstrating SQLite's altering capabilities as well as on modifying a database using the platform's other tools, including the following:

- Altering table
- Modifying views
- Modifying indexes
- Modifying triggers
- Re-indexing a table
- Deleting tables
- Deleting views
- Deleting indexes
- Deleting triggers

In this section of the chapter, I will add the modifying database functionality as well as explore the other unique features of SQLite; for instance:

- Collation sequences
- The JSON extension
- SQLite functions
- Creating custom SQLite functions
- Pragma expressions
- SQLite limits
- SQLite database corruption issues

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_4

Modifying Tables

Three options are available for modifying a SQLite table. You can rename a table using the RENAME command, add a column using the ALTER command, and re-index using the REINDEX command. There is no function or command to alter a column, but it is still possible.

You also need to keep in mind how to handle foreign keys and indexes. You can't directly alter these using a built-in function or command, but it is still possible. Let us start with how to rename a table.

Renaming a Table

Renaming a table is a very simple task, as you only need to issue the RENAME clause in the ALTER TABLE SQLite command. However, a couple of caveats must be respected. First, you cannot move or copy a table from one database to another, either within the same file or in another database file. While we will look at this feature in more depth in a later chapter, SQLite allows for multiple databases in a single database file. Second, if you have foreign-key constraints defined, you will need to disable them first.

Foreign keys with references will be renamed as a result the renaming, both in the table being altered and in the referenced table. However, any indexes, views, and/or triggers that reference a table that is renamed must be manually modified.

Simple Table Renaming

Except for constraints, renaming a table only requires the following, for example:

```
ALTER TABLE main.producer RENAME main.wineries
```

Complex Table Renaming

Of course, if you have constraints on a table, or on indexes, or if you have triggers and views that reference the table that you are renaming, you should use the following procedure and make the alterations.

Run a SELECT query on the sqlite_master table to get a copy of the query you used to create the indexes, views, and triggers. You can use a query similar to this:

```
SELECT sql from main.sqlite_master
```

This query would return all SQL statements for each type of element in the database. I recommend saving this query either to a file or another database so that you have a reference to modify and re-create these elements later. You could also just extract the SQL statements for specific elements, like a view or trigger, in which case the SQL SELECT query would look something like the following:

```
SELECT sql FROM main.sqlite_master WHERE type = 'view' or type = 'trigger'
```

This type of query would return all the statements associated with these database objects.

Once you have these queries in hand, you can disable any constraints by issuing the following command:

```
PRAGMA foreign_keys=OFF
```

Followed by the rename statement from earlier.

Once the renaming is complete, you re-create the indexes, views, and triggers using the CREATE function for each of the objects after you have dropped them from the database. To drop, you simply use the DROP command, as follows:

```
DROP viewname
DROP triggername
DROP indexname
```

Finally, re-enable the foreign keys using the PRAGMA foreign_keys= ON command, as follows:

```
PRAGMA foreign_keys=ON
```

These sequences of SQL queries can all be executed from the Db Mgr app using the Rename table menu command. Finally, you should run an UPDATE query to update the sqlite_master table with any changes to your table, as follows:

```
UPDATE main.sqlite_master
SET sql = ‘ new creatw table query’
WHERE type = ‘table’
AND name = ‘your table name’
```

■ **Warning** If you run an update on the sqlite_master and your update query has syntax errors, you will corrupt the sqlite_master table and your database. It is best to test your update query on a blank database or on a test table with a similar structure as the sqlite_master table to ensure that the update query works as expected.

In the next section, I will show how to add columns to an existing table.

Adding Columns

Adding columns to a SQLite database table is the second actual way of altering a SQLite database table. By this I mean that the ALTER command is only available for these two use cases. As with the renaming, the Add Column functionality has a few caveats that you need to adhere to in order to add columns to a SQLite database table; otherwise, you can define a new column using the CREATE TABLE statement.

- The new column cannot be defined as a primary key.
- The added column cannot have a date-time default value defined using the following functions: CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP.
- If the column is set to NOT NULL, then you must provide a default value.
- If you are adding a column that will be used as reference to a foreign key, then the column must be set to NULL as a default value.
- Each new column must be added individually.

To add a column to an existing table, you simply need to run a SQL query; for example:

```
Alter table main.tablename
Add column columnname datatype default value
```

Here are some actual examples:

```
Create table main.country(id integer  primary key not null autoincrement, name varchar not null)

Alter table main.country
Add column continent varchar null

Alter table main.country
Add column population integer null
```

Once the query is executed using `sqlite3_exec` or `sqlite3_prepare_v2`, `sqlite3_step`, and `sqlite3_finalize`, the alterations will be completed.

Re-indexing a Table

The third type of modification you can do to a table is to re-index its index. You can specify a single index to rebuild, or all of them, if there is more than one in a table. Likewise, if you have a collation sequence to arrange data in a database, you can re-index the collation sequence, and all indexes associated with the collation name will be re-indexed.

Re-indexing an index or collation sequence is accomplished using the `REINDEX` command followed by the schema name or collation name, as follows:

```
REINDEX collation-name
REINDEX main.table_name
REINDEX main.table_index_name
```

It is always useful to rebuild an index on a regular basis or after loading, reloading, or deleting a dataset from a table. This will also ensure that any unused space is removed and the sequence of the index is optimized, thus ensuring the fast data access.

Modifying Views

Views in a SQLite database cannot be modified or altered. You can only `DROP` a view and re-create it. To drop and create a view you can follow this sequence:

```
DROP VIEW schema.view_name
CREATE View schema.view_name as SELECT * or list of columns FROM schema.table_name WHERE
where_clause
```

The column list `WHERE` clause in a view is often used to delimit the amount of data that is stored in a database. The `WHERE` is optional but resolves to a Boolean value.

Modifying Indexes

Like views, an index cannot be altered or modified. You can only `DROP` and `CREATE` an index. If you have more than one index in a table, you will need to `DROP` and `CREATE` them individually. Also, before you drop an index, it is best to get a copy of the SQL `CREATE` query that was used to create the index from the `sqlite_master` table, because this table will be automatically updated once you execute the `DROP` statement, just like it gets updated when you `CREATE` an index.

You can use the following sequence to alter an index:

```
DROP INDEX schema.index_name
CREATE optionally UNIQUE INDEX optionally IF NOT EXIST schema.index_name on schema.table_
name(column(s)) WHERE where_clause
```

The use of UNIQUE and IF NOT EXIST are optional, as is the WHERE clause. The IF NOT EXIST would be redundant here, as the index wouldn't yet exist. If the WHERE clause is used, then the index will be known as a partial index. Also note that you can use an expression instead of a column or sequence of columns to be indexed. In addition, a COLLATION_NAME can be added with a sort order of ASC or DESC to arrange the data in the index.

It is important to remember that when creating or modifying an index, the columns that you use for your index must be in the table where the index is. You cannot use columns in other tables.

Here is an example of using an expression for an index:

```
DROP INDEX main.wine_id
CREATE UNIQUE INDEX main.wine_id on main.wineries(id+name)
```

Using the a collation sequence, you would write a query similar to this example:

```
DROP INDEX main.wine_id
CREATE UNIQUE INDEX main.wine_id
```

You can also use a function call in your expressions as long as the return value is deterministic. In other words, you cannot use functions like RANDOM(). The function can be either one of the functions provided by the SQLite API or a custom function. I will demonstrate functions a little later. For example, you could use the following:

```
DROP INDEX main.wine_id
CREATE INDEX main_wine_id ON main.wineries(coalesce(name))
```

Modifying Triggers

Triggers, as we have seen elsewhere, cannot be altered. You can, however, DROP a trigger and CREATE a new one using any of the accepted API syntax that is permitted. Before dropping a trigger, you should run the SELECT query on the sqlite_master to get the existing SQL query that was used to create the original trigger. If you want to the sqlite_master table with the new version of the trigger, you will need to execute an UPDATE. However, as I warned before, if you execute an UPDATE on the sqlite_master table and your update statement has syntax errors, you run the risk of corrupting the table and scraping your database. It is always best to test on a secondary database first.

What I usually do is either execute a SELECT on the sqlite_master table to get the SQL statement or keep a copy of the SQL statement in a file, because the DROP statement will update the sqlite_master table. For instance, this sequence should be used to replace an existing trigger:

- Retrieve copy of CREATE statement.
- DROP trigger.
- Modify the trigger's logic.
- CREATE the new trigger.
- CREATE a test database.

- UPDATE sqlite_master table in test database.
- Fix syntax issues, if any.
- UPDATE sqlite_master table in your database.

Here are some examples of SQLite triggers that you can execute through the Db Mgr app:

```
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name BEFORE DELETE ON main.table_name FOR EACH ROW WHEN
expression is true or false BEGIN delete-statement
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name BEFORE UPDATE of column ON main.table_name FOR EACH ROW
WHEN expression is true or false BEGIN update-statement
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name BEFORE INSERT ON main.table_name FOR EACH ROW WHEN
expression is true or false BEGIN insert-statement
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name AFTER DELETE ON main.table_name FOR EACH ROW WHEN
expression is true or false BEGIN delete-statement
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name AFTER UPDATE of column ON main.table_name FOR EACH ROW WHEN
expression is true or false BEGIN update-statement
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name BEFORE INSERT ON main.table_name FOR EACH ROW WHEN
expression is true or false BEGIN insert-statement
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name INSTEAD OF DELETE ON main.table_name FOR EACH ROW WHEN
expression is true or false BEGIN delete-statement
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name INSTEAD OF UPDATE of column ON main.table_name FOR EACH ROW
WHEN expression is true or false BEGIN update-statement
DROP TRIGGER main.trigger_name
CREATE TRIGGER main.trigger_name INSTEAD OF INSERT ON main.table_name FOR EACH ROW WHEN
expression is true or false BEGIN insert-statement
```

You can also add a TEMP or TEMPORARY keyword between the CREATE and TRIGGER keywords if you only need a temporary trigger. These triggers will only exist while the database is open. Altering triggers in SQLite is really just creating a new trigger to replace the old one.

Adding and Altering Collation Sequences

Collation sequences are directives on how the data in a database is arranged or sorted. Collation structures are present in the real world, like for cataloguing systems used in libraries, for instance, or medical records.

You can create a collation sequence in SQLite by using the CREATE Collation command, which modifies the database by adding the sequence to the database. The collation sequence can be added when the database is first created, or afterward.

SQLite uses collation sequencing internally to determine the greater of lesser of two values. SQLite has three built-in collation sequence types:

- Binary
- Nocase
- Rtrim

Binary

The BINARY collation sequence algorithm uses the memcmp() C function to compare two text values regardless of string encoding. memcmp() compares the first byte-area size of two strings. The BINARY collate option uses comparison operators (==, <, >, !=, IS, IS NOT, =>, <=) to compare two values.

NoCase

The NoCase collation sequence type compares two text values by converting the 26 ASCII uppercase characters into their lowercase equivalents. Note that it only compares ASCII because full UTF conversion would make the tables manageable.

Rtrim

Lastly, the RTRIM type is the same as BINARY except that it first trims any trailing spaces to the text values to be compared.

To use the collating sequence, you would add COLLATE to the column definition; for example:

```
CREATE TABLE Strings(
Id INTEGER PRIMARY KEY,
String1, //defaults to binary
String2 COLLATE BINARY,
String3 COLLATE NOCASE,
String4 COLLATE RTRIM
)
```

sqlite3_create_collation

Of course, you can create your own collation sequence by implementing the sqlite3_create_collation function. The function has three variants, whose logic is provided here:

```
int sqlite3_create_collation(
  sqlite3,
  const char *zName,
  int eTextRep,
  void *pArg,
  int(*xCompare)(void*,int,const void*,int,const void*)
);
```

The sqlite3_create_collation has five arguments. The first is the pointer to the database connection. In Swift this is represented by a COpaquePointer. The second argument, const char *zName, is the name of the collation sequence module. The third, eTextRep, is the text encoding of the string callback function and must implement one of the following types:

- SQLITE_UTF8
- SQLITE_UTF16LE
- SQLITE_UTF16BE
- SQLITE_UTF16
- SQLITE_UTF16_ALIGNED

The next argument, *pArg, is an application pointer for the first argument of the callback function. The last argument is the callback function. sqlite3_create_collation_v2 is similar to the first except that it provides a Destroy argument, and sqlite3_create_collation16 provides the functionality to create collation sequences in a native 16 bits. See the following:

```
int sqlite3_create_collation_v2(
  sqlite3*,
  const char *zName,
  int eTextRep,
  void *pArg,
  int(*xCompare)(void*,int,const void*,int,const void*),
  void(*xDestroy)(void*)
);
int sqlite3_create_collation16(
  sqlite3*,
  const void *zName,
  int eTextRep,
  void *pArg,
  int(*xCompare)(void*,int,const void*,int,const void*)
);
```

All three functions can add, modify, and delete collation sequences from a SQLite database. In Swift these functions are implemented in the sqlite3.h header file as follows:

```
sqlite3_collation_needed
sqlite3_collation_needed16
```

They both take the same arguments as the versions in C. These functions are abbreviated versions of the former functions and are accessible as well through the standard SQLite C API. You would need to define these functions as COpaquePointers, UnsafeMutablePointer, Int32, and UnsafePointer<Int8> respectively. For the 16-bit version, the last argument would be a 16-bit UnsafePointer rather than an UnsafePointer<Int8>.

The SQLite DELETE Statement

Deleting elements from a SQLite database is implemented through the SQL DROP function. You can use the DROP function to remove tables, views, indexes, and triggers. When you drop a table or an index, the sqlite_master table is updated accordingly. For the views and triggers, you need to execute a DELETE record on the table.

The DROP statements are provided next and can be used as is through the DB Mgr app by entering the DROP function followed by the schema element to remove and the name of the element. Once you click on the Execute button, the element will be removed from the database. Removing or dropping a table will delete any data that is in that table. Of course, the database needs to be open first.

Deleting Tables

```
DROP TABLE schema.table_name
```

Deleting Views

```
DROP VIEW schema.view_name
```

Deleting Indices

```
DROP INDEX schema.index_name
```

Deleting Triggers

```
DROP TRIGGER schema.trigger_name
```

Deleting Collation Sequences

Custom collation functions are deleted when you invoke the `Destroy` argument or when the database connection is closed using the `sqlite3_close` function.

SQLite Functions

Unlike other relational database engines, SQLite doesn't provide an API to create stored procedures, also known as SPROCS. However, you can define your own functions in addition to using the built-in functions in SQLite.

SQLite has several built-in functions that can be categorized as follows:

- Core functions
- Aggregate functions
- Date/time functions
- JSON functions
- Standard functions

Core functions include functions `abs`, `coalesce`, `ifnull`, `instr`, `glob`, `like`, and `length`. They make up the core functionality of the SQLite platform. The Aggregate functions, as the name implies, provide functions like `count`, `avg`, `min`, `max`, `total`, and `sum`, among others. The Date/time functions allow you to obtain and manipulate date and time values. These functions include `date`, `time`, `datetime`, `juliandate`, date modifiers, and operators. For example, you can get the current date as follows:

```
SELECT date('now')
SELECT date('now', 'YYYY-MM-DD')
SELECT date('now', 'MMM/dd/YYYY', '-1 day')
SELECT date('now', 'MM-dd-yyyy', '-1month', '+7 days', '+1 year')
```

The JSON Extension

The JSON functions, officially known as the json1 extension, are a fairly new addition to SQLite. They aren't available with the standard API. You need to install them manually if you need to use them. These functions are loaded at run-time. Using the JSON functions, you can store and parse data in the JSON format. The extension includes the functions listed here:

- `Json(json)`
- `Json_array`
- `Json_array_length(json)`

- Json_array_length(json. Path)
- Json_extract
- Json_insert
- Json_object
- Json_replace
- Json_remove
- Json_set
- Json_type(json)
- Json_type(json, path)
- Json_valid
- Json_group_array
- Json_group_object
- Json_each
- Json_tree

The JSON functions that have json as a first parameter must be a valid JSON object, a number, a string, or a null value. Number and Null are interpreted as SQLite data types. The PATH argument must be a valid, well-formed path value that begins with $.

To load the json1 extension, you need to implement the loadable interface using sqlite3_load_extension, which is available in the C API through the bridge. You also have the sqlite3_enable_load_extension, which can disable the loading of extensions to prevent security leaks. See the following:

```
sqlite3_load_extension(db: COpaquePointer,zFile: UnsafePointer<Int8>, zProc:
UnsafePointer<Int8>, pzErrMsg:UnsafeMutablePointer<UnsafeMutablePointer<Int8>>)

sqlite3_enable_load_extension(db: COpaquePointer, onoff: Int32)
```

The extension functions must be called when the database is open and running so that the extensions can be loaded as needed.

The first argument is the pointer to the SQLite database engine. The second argument is the file for the shared library. For the json1 extension, the file is not included in the default sqlite3.dylib library. The third argument is the entry point for the extension, while the fourth is a pointer for the error message:

```
var db:COpaquePointer?=nil
let lib_file: UnsafePointer<Int8>? = nil
let proc: UnsafePointer<Int8>?=nil
let err_json_msg:UnsafeMutablePointer<UnsafeMutablePointer<Int8>>? = nil

func load_extension()->Void{
      sqlite3_load_extension(db, lib_file, proc, err_json_msg)
    }
```

■ **Note** The json1 sqlite3 extension is not included in the sqlite3 dylib, which references the `sqlite3.h` file in iOS 10. You could attempt to add it by downloading the sqlite3 source code and extracting the `sqlite3.c` file from the ext/misc directory. Then you would need to add it to your project, but this would probably create conflicts with the existing sqlite3 library in iOS. The other option would be to compile a new static Objective-C extension (.h) and add the `sqlite3.h` and `sqlite3.c` files to your project from the sqlite3 source code. Then, you would need to import the `sqlite3.h` file into the Objective-C extension (header). Next, you would need to create a new bridge header file and add it to the Swift compiler bridge configuration.

I have created a reference project using the `sqlite3.c` and `sqlite3.h` files from the actual code instead of the Xcode-supported sqlite3.dylib library.

The project is in GitHub (https://github.com/kevlangdo/load_sqlite_json_extension) and is experimental. I haven't tested it, and it is outside of the context of this book, but it gives an example of how to load the json extension in an iOS project.

Creating Functions using Swift

Other than the SQLite functions included with the API and the extensions, you can create your own and attach them to your database. These extensions are written in C.

For the next iOS app project, the Wineries, I will need a function that can convert liters into ounces and vice versa. The code must be written in C using, in part, the SQLite API to convert values to and from SQLite.

SQLite values are as follows:

```
SQLITE_API const void *SQLITE_STDCALL sqlite3_value_blob(sqlite3_value*);
SQLITE_API int SQLITE_STDCALL sqlite3_value_bytes(sqlite3_value*);
SQLITE_API int SQLITE_STDCALL sqlite3_value_bytes16(sqlite3_value*);
SQLITE_API double SQLITE_STDCALL sqlite3_value_double(sqlite3_value*);
SQLITE_API int SQLITE_STDCALL sqlite3_value_int(sqlite3_value*);
SQLITE_API sqlite3_int64 SQLITE_STDCALL sqlite3_value_int64(sqlite3_value*);
SQLITE_API const unsigned char *SQLITE_STDCALL sqlite3_value_text(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL sqlite3_value_text16(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL sqlite3_value_text16le(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL sqlite3_value_text16be(sqlite3_value*);
SQLITE_API int SQLITE_STDCALL sqlite3_value_type(sqlite3_value*);
SQLITE_API int SQLITE_STDCALL sqlite3_value_numeric_type(sqlite3_value*);
```

To create a SQLite function for Swift, you need to follow these steps:

- As shown in Figure 4-1, add a C file using the C File template under the iOS categories. For my function, I will name it *sizeconverter* because it will convert volume from liters to ounces and vice versa. Xcode will ask if you want to add a header file; say yes.

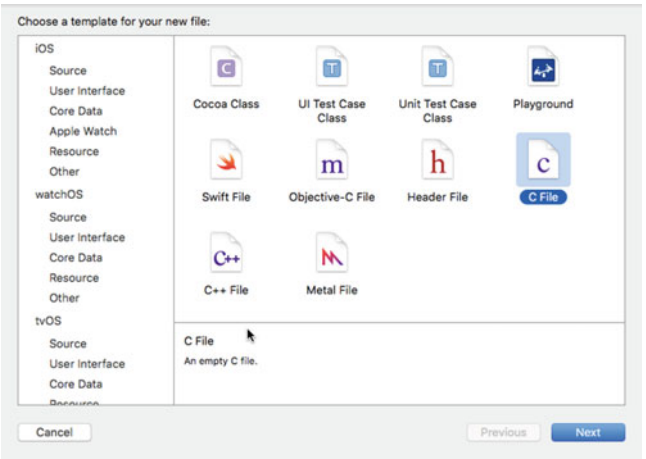

Figure 4-1. *C File template*

- Next, you need to add C logic to the C file and import whatever C library into the header. You can also define your functions' signatures in the header file, which I am doing for this example. For example, I am adding the `#include` directive for the `sqlite3.h` header file.
- Then, you need to add the C file to your bridge file using the `#import` statement (see the code that follows).
- As shown in Figure 4-2, you need to add your header file to Build Phases > Compile Sources.

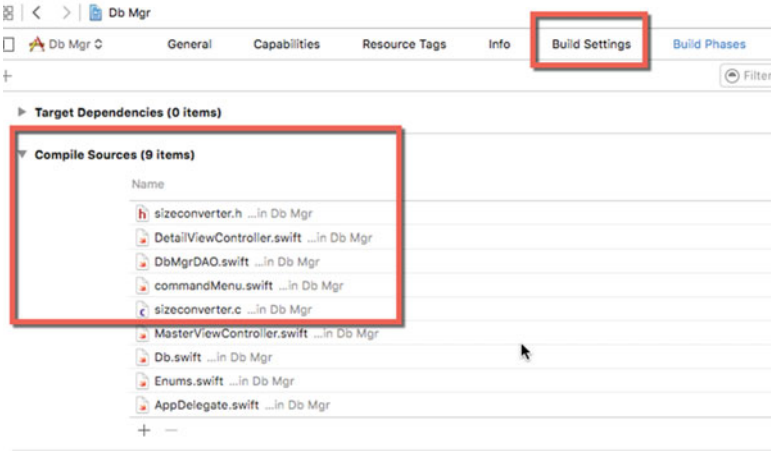

Figure 4-2. *Build phases*

The code for the function (`sizeconverter.c`) is shown here, along with the header file and the changes to the SQLite3Bridge:

```
//
//  SQlite3Bridge.h
//  Db Mgr
//
//  Created by Kevin Languedoc on 2016-05-20.
//  Copyright © 2016 Kevin Languedoc. All rights reserved.
//

#ifndef SQlite3Bridge_h
#define SQlite3Bridge_h

#endif /* SQlite3Bridge_h */

// Add this code to import the sqlite3 header. The code above is supplied by the template
#import <sqlite3.h>
#import "sizeconverter.h"
```

The Adjust SQLite3Bridge Header File

```
//
//  sizeconverter.h
//  Db Mgr
//
//  Created by Kevin Languedoc on 2016-07-03.
//  Copyright © 2016 Kevin Languedoc. All rights reserved.
//

#ifndef sizeconverter_h
#define sizeconverter_h

#include <stdio.h>
#include <sqlite3.h>

static void sizeconverter(sqlite3_context *context, int argc,   sqlite3_value **argv);

#endif /* sizeconverter_h */
```

The sizeconverter header file

```
//  sizeconverter.c
//  Db Mgr
//
//  Created by Kevin Languedoc on 2016-07-03.
//  Copyright © 2016 Kevin Languedoc. All rights reserved.
//

#include "sizeconverter.h"
#include "SQlite3Bridge.h"

static void sizeconverter(sqlite3_context *context, int argc,   sqlite3_value **argv)
{

    double result = 0.0;
    const char *liter;
```

```
    const char *ounce;
    const char *us;
    const char *uk;

    us = "us";
    uk = "uk";
    liter = "l";
    ounce = "o";

    if (argc==3) {
        double from = sqlite3_value_double(argv[0]); // original volume
        const unsigned char *to = sqlite3_value_text(argv[1]); // liters or ounces
        const unsigned char *country = sqlite3_value_text(argv[2]); // us or uk

        const double us_uom = 33.8140226; // 1 Liter = 33.8140226 Ounces [Fluid, US]
        const double uk_uom = 35.195079; // 1 Liter = 35.195 079 Ounces [UK]

        if (country == (const unsigned char*)us && to == (const unsigned char*)liter) {
            result = from * us_uom;
        }else if (country == (const unsigned char*)uk && to == (const unsigned char*)liter){
            result = from * uk_uom;
        }else if (country == (const unsigned char*)us && to == (const unsigned char*)ounce){
            result = from / us_uom;
        }else if (country == (const unsigned char*)uk && to == (const unsigned char*)ounce){
            result = from / uk_uom;
        }

    }
    return sqlite3_result_double(context, result);
}
```

This is a simple C function that takes three arguments: the original volume, the target unit of measure, and the country, which can be United States or United Kingdom, since these have different liter conversion values because their ounces are based on either the Imperial system or the American system. Most other countries use the metric system.

With the function done and configured, we only need to use it in the Db Mgr app and attach it to the database.

Using Functions in a SQLite Database using Swift

Using custom SQLite functions requires using `sqlite3_create_function`. The first argument is the SQLite database pointer, which is a `COpaquePointer`; the second argument is the name of the function as a UTF-8 encoding string; the third parameter is the number of input parameters; and the fourth parameter is the text encoding type the function prefers. In my case, it is UTF-8, which is pretty standard, but it will accept any of the supported encodings. The fifth parameter is an arbitrary pointer that allows access through the function using the `sqlite3_user_data()` function. The last three arguments are pointers that implement the function or aggregate:

```
 func createSQLiteFunction()->Enums.SQLiteStatusCode{
        let funcname:String = “sizeconverter”
        return Enums.SQLiteStatusCode(rawValue:sqlite3_create_function(db, funcname.cString
(String.encoding.utf8)!, 3, SQLITE_UTF8, nil, sizeconverter, nil, nil))!
}
```

Once the function is connected, you can use the function elsewhere in your INSERT, SELECT, or UPDATE queries, which we will look at later in this book in the chapters covering those subjects.

Pragma Statements

Pragma statements are a unique feature of the SQLite platform. These statements are used to set and control environment variables within a SQLite database and environment.

Pragma statements can be used like other queries, using the sqlite3_prepare_v2, sqlite3_step, and sqlite3_finalize functions once the database is open and the connection is established. However, some PRAGMAs will run during slite3_preapre_v2 or sqlite_step or both, depending on the Pragma statement.

For example:

```
let pragma:String = “PRAGMA schema.index_list( table_name )”

If(sqlite3_open(db.path, db) == SQLITE_OK){

If(sqlite3_prepare_v2(db, pragma.cString (String.encoding.utf8)!, -1, &sqlStatement, nil) ==
Enums.SQLiteStatusCode.ok.rawValue}{

}
}
```

SQLite has many Pragma statements that are used more than others (see https://www.sqlite.org/pragma.html for the full list). For instance, here are some useful ones:

- Foreign_key_check
- Foreign_key_list
- Integrity_check
- Automatic_index
- Busy_timeout
- Shrink_memory
- Auto_vacuum

Foreign_key_check

```
PRAGMA schema.foreign_key_check;
PRAGMA schema.foreign_key_check(table-name);
```

The foreign_key_check PRAGMA runs against the database or table to check for any foreign-key violations. It returns a row for each violation, so you could use the sqlite3_step function to retrieve the returning value. The resulting value has four columns, as follows:

- The name of the name containing the reference.
- The second value is the rowindex where the violation occurred.
- The third column is the table that the foreign key refers to.
- The last column is the foreign-key name.

Foreign_key_list

The `Foreign_key_list` PRAGMA returns the list of foreign keys in a database. It returns one row for each foreign-key constraint.

```
PRAGMA foreign_key_list(table-name)
```

Integrity_check

The `integrity_check` PRAGMA does a sanity check on the entire database looking for missing indexes, broken records, out-of-order records, missing pages, and `UNIQUE` or `NOT NULL` violations, among other verifications. The results are returned as a single column describing the issue. If no problems are found, the returning column will contain the "OK" value only. The *N* refers to the max number of errors to return. It defaults to 100.

```
PRAGMA schema.integrity_check;
PRAGMA schema.integrity_check(N)
```

Automatic_index

The `automatic_index` PRAGMA allows the developer to query, set, or clear the automatic index on a table.

```
PRAGMA automatic_index;
PRAGMA automatic_index = boolean;
```

Busy_timeout

This PRAGMA expression enables you to get the current timeout or set the busy timeout for query execution.

```
PRAGMA busy_timeout;
PRAGMA busy_timeout = milliseconds;
```

Shrink_memory

This PRAGMA expression frees up memory as much as possible. This is handy for large databases or long-running queries. This is similar in concept to a garbage collector.

```
PRAGMA shrink_memory
```

Auto-vacuum

This PRAGMA removes empty space from the `page_file`, thus freeing up memory and shrinking the size on disk.

```
PRAGMA schema.auto_vacuum;
PRAGMA schema.auto_vacuum = 0 | NONE | 1 | FULL | 2 | INCREMENTAL;
```

Corrupting a SQLite Database

SQLite is an extremely stable technology and is very robust, but it is still possible to cause database files to become corrupt if you're not careful.

For instance, if you move a database while the connection is open, you run the risk of corrupting the database. Since a SQLite file is a standard binary file, there is nothing that prevents a rogue thread or process from interfacing with. Within the context of an iOS sandbox where the app lives, this would be difficult but not impossible.

Another possible threat is trying to access the same file descriptor after it was closed and reopened. Also, trying to back up or restore a database while it is open can damage the file.

Broken file locks, which are managed by the filesystem, can damage the database file if you try to access it while a faulty lock is in place. Although there are many other corruption issues that can arise through the POSIX and by using different locking mechanisms or having multiple applications try to access the same database, the environment in which a SQLite database operates on iOS somewhat protects it from corruption.

However, one way to easily corrupt a database is to rename it while it is open. You have to be careful when moving a database—say, from the Resources bundle to the Documents directory—that the database is closed before attempting the operation.

Lastly, there are syncing issues that cause damage to a database. This could be an issue if you place a copy of the database on Dropbox, OneDrive, or iCloud or any other Cloud file storage and sharing platform and try to sync the database files while it is open, which can cause damage to the database file. Make sure that the connection is closed before attempting any of these operations.

SQLite Limits

The last word goes to the limits. SQLite is very happy storing large amounts of data within its storage facility but it still does have its limits. Many of the limits imposed on SQLite come from the OS or memory. For instance, memory can be confined to 32-bit or 64-bit. On an iPhone, you have 1 GB of RAM available for the whole device. On an iPad Pro, however, you have 4 GB of RAM. SQLite must work within these small confines.

Another limit can be the overall disk size. Since we are dealing with mobile devices, you have a finite amount of disk space available.

There are other limits that can be tweaked during runtime for the specific needs of the application. The default length of a blob is defined by the macro `SQLITE_MAX_LENGTH` with a value of a 1 billion bytes. However, you can change this default using the `DSQLITE_MAX_LENGTH` flag:

```
-DSQLITE_MAX_LENGTH=123456789
```

You can also change the maximum number of columns, indexes, or views, or the maximum number of `update` and `insert` terms or `where` clause terms, from the default value of 2000 to a max of 32767. But how many databases have you seen reaching these sizes, even on a server?

Another size limit you can change is the length of a query. The size is set at 1000000 through the `DSQLITE_MAX_LENGTH` macro. This value can be increased to 1073741824. As for tables, you can have up to 64 tables. Try that one if you can.

Another interesting limit is the maximum amount of parameters a function can have. The default value is 100, but this value can be modified using the `SQLITE_MAX_FUNCTION_ARG` macro. Also, the maximum amount of compound `SELECT` statements using joins is 500. This value can be changed using the `SQLITE_MAX_COMPOUND_SELECT` macro. You can attach up to 125 databases in a file, while the default is 10. The theoretical maximum number of rows in a table is 2^{64} (18446744073709551616, or about 1.8e+19). However, as the documentation states, this limit can never be reached, since a database file can have a maximum size of 164 terabytes, which far surpasses the physical limits of any iOS device.

Although SQLite has limits on other aspects of its architecture, these limits can never be reached within the confines of any iOS device. You will run out of physical space or resources before you reach SQLite's limits.

Summary

This concludes this chapter. We have explored the different features available in the SQLite API for altering databases. We have also looked at PRAGMA statements and SQLite's built-in functions. Finally, we built a custom SQLite function and added to the Winery database in the DB Mgr app.

The next several chapters will demonstrate how to perform CRUD operations on a SQLite database through the use of an iPhone app. We will also look at how to perform searches.

CHAPTER 5

Inserting Records

The SQL `Insert` statement in SQLite has some interesting features, such as being able to replace an existing record, just like the `Upsert` statement in Oracle's PL/SQL. The expression parameter can either be a literal or the return value from a function. We will explore these interesting features together.

This chapter will demonstrate how to insert data into a SQLite database. I will cover all the supported variations of the API, including:

- The data-binding functions
- Inserting records
- Inserting or replacing records
- The Insert or Rollback option
- The Insert or Ignore option
- The Insert or Abort option
- The Insert or Fail option
- Inserting blobs
- Building an iOS app to insert records

The Data-Binding Functions

The SQLite C API has specific functions for working with data, allowing an application to bind data with the Cocoa Touch corresponding data types. Bound data types are used with prepared statements. The following functions work with all the major primitive data types in regards to text and numbers. For blobs, images, videos, and audio the API provides blob and blob64 as well as zero and zeroblob64.

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_blob64(sqlite3_stmt*, int, const void*, sqlite3_uint64, void(*)(void*));
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_int64(sqlite3_stmt*, int, sqlite3_int64);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int n, void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int, void(*)(void*));
int sqlite3_bind_text64(sqlite3_stmt*, int, const char*, sqlite3_uint64, (*)(void*),
unsigned char encoding);
```


K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_5

```
int sqlite3_bind_value(sqlite3_stmt*, int, const sqlite3_value*);
int sqlite3_bind_zeroblob(sqlite3_stmt*, int, int n);
int sqlite3_bind_zeroblob64(sqlite3_stmt*, int, sqlite3_uint64);
```

For the basic primitive types, the first parameter of the SQLite binding function is the statement that you create using the `sqlite3_stmt` function and that is returned from the `sqlite3_prepare_v2` function. The second parameter is the column number in the table where you want to insert records, which corresponds to a zero-based array, and the last parameter is the value to be inserted. The columns are a 0 (zero)-based array. So column 1 would be 0 and column 2 would be 1, and so on.

For blobs and 64-bit-length functions, the fourth parameter is the number of bytes in the value to be inserted, not the characters. You use blobs to store images, videos, and audio data. The last argument is the destructor.

The SQLite INSERT function

The `INSERT` function is the SQLite implementation of the SQL `INSERT` statement. The SQLite `INSERT` statement comes in three variations:

- You can insert values into a database table without specifying the column names. However, the values being inserted must match the number of columns in the table. This type of insert takes the following form:

  ```
  INSERT into main.table VALUES(values list items)
  ```

- The second type of insert uses a `SELECT` statement as the list of rows to insert. If you omit the column names in the `INSERT` statement then the `SELECT` statement must have the exact same number of columns as in the target table, unless the table columns have default values. Here are three examples of the second type:

  ```
  INSERT INTO main.table
          SELECT * FROM main.otherTable WHERE clause
  INSERT INTO main.table
          SELECT column list FROM main.otherTable (with or without WHERE clause)
  INSERT INTO main.table(column list)
          SELECT column list FROM main.otherTable (with or without WHERE clause)
  ```

Here is a quick, boilerplate example of how to insert records using the SQLite C API and Swift:

```
internal let SQLITE_STATIC = unsafeBitCast(0, to: sqlite3_destructor_type.self)
internal let SQLITE_TRANSIENT = unsafeBitCast(-1, to: sqlite3_destructor_type.self)

func sample(){

        var sqlite3_stmt:COpaquePointer?=nil;
        var sqlite3_db:COpaquePointe?r=nil;
        var txt:String = "some text";
        let integer:Int32 = 500;
        let dbl:Double = 10.45;
        var dbPath:URL = URL()
```

```
        var sqlStatement:COpaquePointer?=nil
        var dbErr: UnsafeMutablePointer<UnsafeMutablePointer<Int8>>? = nil
        var errmsg:String=""
        let dbName = "winery.sqlite"

        //insert query
        let sql:String = "INSERT INTO table(coltext, colint, coldouble) VALUES(?,?,?)";

        let dirManager = FileManager.default()
        //Open db assuming there are no subfolders
        do {
            let documentDirectoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)

            dbPath = documentDirectoryURL.urlLByAppendingPathComponent(dbName)
        } catch let err as NSError {
            print("Error: \(err.domain)")
        }

        sqlite3_open(dbPath.path!, &sqlite3_db);
        sqlite3_prepare_v2(sqlite3_db, sql, 1, &sqlite3_stmt, nil);
        sqlite3_bind_text(sqlite3_stmt, 1, txt.cString(using: String.encoding.ut)!, -1,
        SQLITE_TRANSIENT);
        sqlite3_bind_int(sqlite3_stmt, 2, integer);
        sqlite3_bind_double(sqlite3_stmt, 3, dbl);
        sqlite3_step(sqlite3_stmt);
        sqlite3_finalize(sqlite3_stmt);
        sqlite3_close(sqlite3_db);

    }
```

In the preceding code, `SQLITE_STATIC` is an immutable `sqlite3_static` value pointer and `SQLITE_TRANSIENT` is a pointer that will change in the future, but it is SQLite that will move the pointer as needed.

Within the confines of a function, I create a SQLite statement variable called `sqlite3_stmt` using a `COPaquePointer`. This will be initialized with the string query, `sql`, through the `sqlite3_prepare_v2` function, along with the SQLite database engine variable, `db`, which is created using the `COPaquePointer`. The next three lines of code create some test variables to allow the code to insert values in the database. For the sake of clarity and simplicity, I am creating a String variable, an Int32, and a Double. I will also hard code some values. However, these can be dynamically set using a block, parameters in a method, or as a result of some process or calculation.

The actual SQL `INSERT` query statement is pretty standard, as you can see from the `sql String` variable, which must be converted to a C char (string of chars) later using `cUsingStringEncoding` and `NSUTF8Encoding`. I find this statement works best. I have seen some developers trying to replace the interrogation symbols with actual variable names, but the code can get very messy and hard to maintain, and often SQLite will complain that there are issues with the statement because you need to convert and bind the input values.

The next piece of the puzzle is the path to the database file. In this example, I get the path and appended file name by using the `NSSearchPathDirectory.DocumentDirectory` and `GetDirectory` functions in the `NSFileManager` class. I append the database file name using the `URLByAppendingPathComponent` function, which was changed from the `stringByAppendingPathComponent` method. The preceding code assumes that there are no subfolders, otherwise I would need to store the Document folder and subfolders in an array using the `NSSearchPathForDirectoriesInDomains` function.

The other piece of important information is the fact that you need to create your database in the Documents directory in your app's sandbox. It is the only useful writable directory. If you place the database in the Resources folder, you won't get any errors, but the data will not be written to the database because the folder is read-only.

Insert or Replace

SQLite also has a special clause that allows you to insert/update using the INSERT statement. With SQLite you use the INSERT OR REPLACE statements. If an insert constraint is encountered, the REPLACE clause will delete the existing record and replace it with the new record. In cases where there is a NOT NULL, REPLACE will attempt to replace it with a default value if the REPLACE is trying to insert a NULL value. If no default value is available, the ABORT clause is used instead on that row. The remaining rows aren't affected unless similar conditions are encountered. The general syntax is as follows:

```
INSERT OR REPLACE into schema.table_name(Id, ColText, ColInt, ColDouble) VALUES(1, 'Kevin',
20, 53.6)
```

If the index value, Id = 1, doesn't exist, it will be inserted. Likewise, if it does exist, the record will be updated. You can use Swift data types instead of the primitives when inserting records, as the following example demonstrates:

```
    func replace(){
        let index:Int32 = 1
        let name:String = "kevin"
        let Dbl:Double = 1000.99
        var dbPath:URL = URL();

        let dirManager = FileManager.default()
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)

            dbPath = directoryURL.urlByAppendingPathComponent("database.sqlite")
            if( sqlite3_open(dbPath.absoluteString?.cString(using: String.Encoding.
            utf8)!,&sqlite3_db) == 0){
                let sql:String = "INSERT OR REPLACE INTO schema.simpletable (id, name,
                colDouble)VALUES(\(index),\(name.cString(using: String.Encoding.utf8)), \(Dbl))"

                if(sqlite3_prepare_v2(sqlite3_db, sql.cString(using: String.Encoding.utf8)!,
                -1, &sqlite3_stmt, nil) != SQLITE_OK)
                {
                    print("Problem with prepared statement")

                }else{
                    sqlite3_finalize(sqlite3_stmt);
                    sqlite3_close(sqlite3_db);
                }
            }
```

```
        } catch let err as NSError {
            print("Error: \(err.domain)")
        }

    }
```

This query is similar to the previous one, except that I am using interpolation to construct a new query String.

Insert or Rollback

The rollback option provides you with an elegant way to back out of a transaction if things don't go your way, like, for instance, if there is an issue with the data being inserted into the column.

The `OR REPLACE` is shorthand for the `ON CONFLICT REPLACE` clause, just like the other conflict-handling clauses: `ABORT`, `REPLACE`, `IGNORE`, `FAIL`. For instance, you may not want to replace an existing record. If a duplicate record already exists in the database, you may want to roll back that transaction without generating an error. For general syntax, it is as follows:

```
let sql:String = "INSERT or ROLLBACK INTO table(coltext, colint, coldouble) VALUES(?,?,?)";
```

Again, the query assumes that you will be using SQLite3 data binding to bind data to the values, which is the safest way or pattern to use when working with SQLite3.

Insert or Ignore

The `IGNORE` clause handles the insert constraint by skipping over the problematic row altogether and generates an `SQLITE_CONSTRAINT` error like the other constraint clauses, except for `REPLACE`. The rows before and after are treated normally unless another constraint is encountered. See the following example:

```
let sql:String = "INSERT or IGNORE INTO table(coltext, colint, coldouble) VALUES(?,?,?)";
```

Insert or Abort

The `ABORT` clause aborts the current operation and backs out of the current transaction, allowing your program to continue to handle the other potential inserts or to continue with the runtime logic. The syntax is the same as for the other constraint clauses:

```
let sql:String = "INSERT or ABORT INTO table(coltext, colint, coldouble) VALUES(?,?,?)";
```

Insert or Fail

The last clause is `FAIL`. This clause is triggered when a constraint is encountered and `SQLITE_CONSTRAINT` is called. The previous transactions are maintained, but the current one is cancelled, and subsequent transactions do not occur. Here is an example:

```
let sql:String = "INSERT or FAIL INTO table(coltext, colint, coldouble) VALUES(?,?,?)";
```

Inserting Blobs

Working with binary data in Swift and the Cocoa Touch framework implies that you will be using the NSData class on the iOS side and blob on the SQLite side. In this section, we will look at two functions that will allow you to insert any binary data into a SQLite database. While this is technically possible, I wouldn't recommend using this in a production app, because it would use up a tremendous amount of valuable storage. There might be some instances where you need to provide binary data as part of your part, and this data wouldn't increase in size over time.

The following two functions can be used to insert binary data like videos, images, and audio. Of the two, the sqlite3_bind_blob function will be most always be used. The sqlite3_bind_zeroblob is used to write blobs incrementally into the database instead of storing the full size of the blob into memory all at once. We will demonstrate both:

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_zeroblob(sqlite3_stmt*, int, int n);
```

The first parameter is the SQLite INSERT statement, the second parameter is the index of the parameter to be set, and the third is the value to bind to the statement. If this value is nil, then the fourth parameter is ignored; otherwise, this parameter is the amount of bytes in the buffer, which represents the length of the binary data. The last, or fifth, parameter is a destructor to dispose of the blob once it has been inserted into the database. The following code provides a reference implementation:

```
func insertBlob(){
        var dbPath:URL = URL()
        var bindata:Data = Data()
        var imagePath:String = ""
        var urlObj:URL = URL()
        let binName:String = "Test image"
        let dirManager = FileManager.default()
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)

            dbPath = directoryURL.urlByAppendingPathComponent("database.sqlite")
            if( sqlite3_open(dbPath.absoluteString?.cString(using: String.Encoding.
            utf8)!,&sqlite3_db) == 0){
                let sql:String = "Insert into binaryTbl(PictureName, ImageData) VALUES(?,?)"

                if(sqlite3_prepare_v2(sqlite3_db, sql.cString (using: String.Encoding.
                utf8)!, -1, &sqlite3_stmt, nil) != SQLITE_OK)
                {
                    print("Problem with prepared statement")
                }else{
                    imagePath = Bundle.main.pathForResource("IMG_0095", ofType: "JPG")!
                    urlObj = URL(fileURLWithPath: imagePath)
                    bindata = try! Data(contentsOf: urlObj)!
                    sqlite3_bind_text(sqlite3_stmt, 1, binName.cString (using: String.
                    Encoding.utf8)!, -1, SQLITE_TRANSIENT);
                    sqlite3_bind_blob(sqlite3_stmt, 2, bindata.bytes,Int32(bindata.count),
                    SQLITE_TRANSIENT);
```

```
                    if(sqlite3_step(sqlite3_stmt)==SQLITE_DONE){
                        sqlite3_finalize(sqlite3_stmt);
                        sqlite3_close(sqlite3_db);
                    }
            }
                }
} catch let err as NSError {
                print("Error: \(err.domain)")
            }
        }
```

As you can see from the preceding code, the pattern is similar to the other SQLite code snippets. First, we create either our variables using the `var` keyword or a constant using the `let` keyword. If you make a mistake, Xcode will let you know and offer a suggestion to fix it. The visibility of the variable (or constant) depends on your needs.

Once the variables and constants are created, you attempt to open the SQLite database as usual and set up your SQL query. You execute the query using the `sqlite3_prepare_v2` function, and you bind your data to the right columns.

Binding binary is similar to binding other data types, with a few extra steps. To work with binary data (images, videos, compiled files), you need to use `NSData`, which is a class in the Foundation framework that provides wrappers for the bytes buffers.

When you are storing binary data, you are actually storing bytes. To get the bytes, use the `NSURL` class to retrieve the path to the file and read in the bytes from the file using the `NSData contentsOfURL`. The `sqlite3_bind_blob` needs the bytes contents of the file as well as the length of the bytes buffer in addition to a pointer to the `sqlite3_statement`, column position, and the `SQLITE_TRANSIENT` pointer.

Creating the Winery App

The Winery iPhone app will allow a user to enter information about their favorite bottles of wine and take a photo of the label or bottle. The information will be stored in a SQLite database. In later chapters, we will expand on the app to include other CRUD (Create or Insert, Read or Select, Update, and Delete) operations.

In this chapter, I will create the initial app and add the insert capabilities for the wine information and wineries information. The app will be based on the Tabbed Application template.

Create the Project

From Xcode, create a new project and select the Tabbed Application template under the iOS Application category.

■ **Note** If you are new to Xcode, you can create a project from the launcher if no projects are currently open, or from File menu ➤ New ➤ Project.

Add the Bridge

As always, the first step after creating the project is to create an Objective-C bridge. You could create a bridge individually for each project, or you can use create a share resource copy it to work project. You can also use CocoaPods (`CocoaPods.org`) to install the sqlite3 library and the bridge in your new project. You could also use a project from the CocoaPods repository.

Within the scope of this project, I will simply add the sqlite3 library to my project and manually create a bridge file.

To add the sqlite3 iOS-supported library, follow these steps:

1. From the General Settings tab, locate the Linked Framework and Libraries section.
2. Click the "+" button and enter "sqlite3" in the Search field.
3. Select the `libsqlite3.tbd` library.
4. Click Add to add the library to your project.
5. Next, add a new Objective-C header file using the Header File template.
6. Add an `#import <sqlite3.h>` statement.
7. Under Build Settings, search for Swift.
8. Under the Objective-C Bridge Header entry, add the name of the bridge file preceded by the project folder name, unless you place the file in the root directory of the project (Figure 5-1).

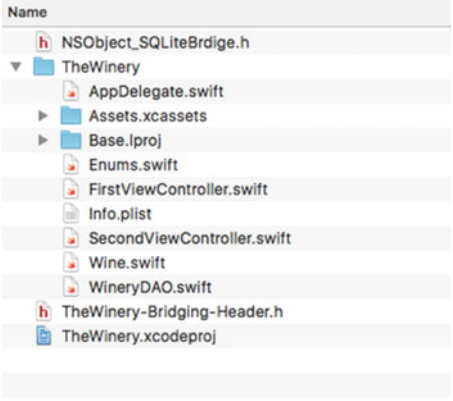

Figure 5-1. *Bridge file Location*

Alternatively, if there is no bridge file in your project, do the following:

1. Select the Objective-C File template under the iOS Source category (Figure 5-2).

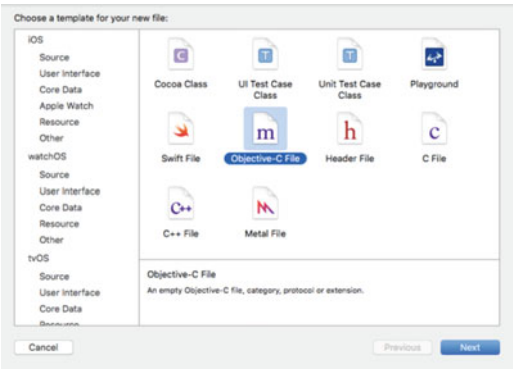

Figure 5-2. *Objective-C File template*

2. Provide a name, such as SQLiteBridge (Figure 5-3).

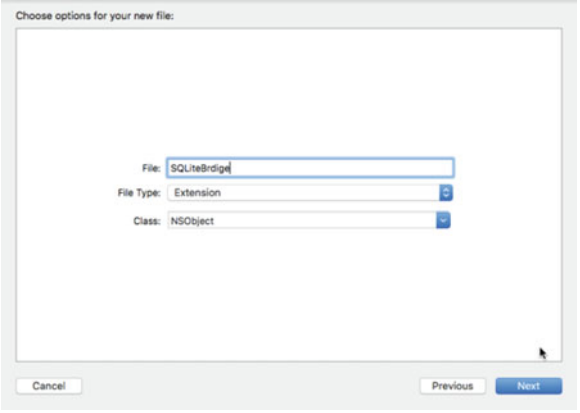

Figure 5-3. *Name and file type*

3. Select the Extension file type and NSObject as the class.
4. Once you click Next and Add, Xcode will ask you to create a bridge file (Figure 5-4).

Figure 5-4. *Create Bridge popup*

5. When you click on the Create Bridging Header button, Xcode will generate the header file and add the reference to the Build Settings for you (Figure 5-5). You can discard the `NSObject_SQLiteBridge.h` file, as it is not needed. The actual bridge file is `TheWinery-Bridging-Header.h`, and it is this file that is included in the Swift Compiler settings for the Objective-C header mapping.

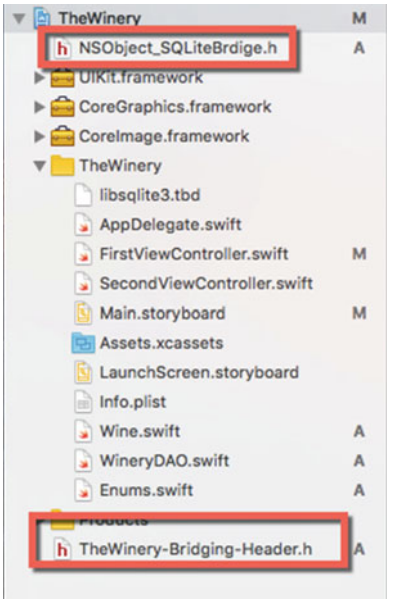

Figure 5-5. *Xcode-created bridge header*

Creating the UI View for Inserting

Before getting into the data model and controllers, I will build the UI and add the `IBOutlets` and `IBActions` to the `FirstViewController` and `SecondViewController`.

Without adding any other code, you can run the app as is and test Swift between the first and second scenes. Xcode provides a lot of boilerplate code through the template for us.

Figure 5-6 provides a view of the layout of the view controllers and navigation controller along with the components that we will add next.

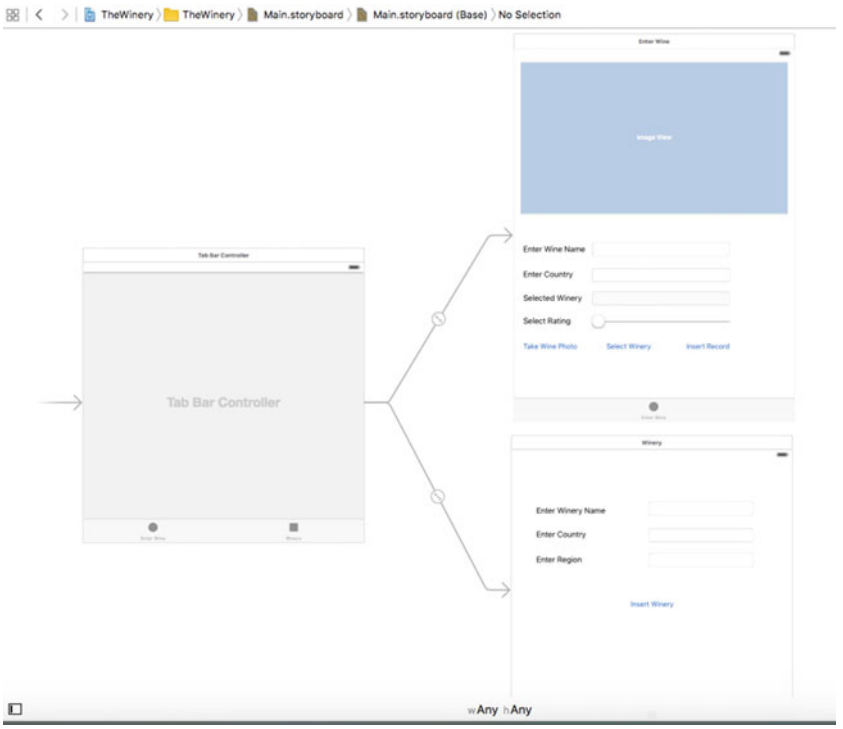

Figure 5-6. *Winery UI in Xcode IB*

The first view controller will include the following UI components:

Element	Name	Connection Type
UIImageView	imageView	IBOutlet
UITextField	wineNameField	IBOutlet
UITextField	countryNameField	IBOutlet
UISlider	wineRating	IBAction
UITextField	selectWineryField	IBOutlet
UIButton	InsertRecordAction	IBAction
UIButton	selectPhoto	IBAction
UIButton	selectWinery	IBAction
UILabel	Enter Wine Name	
UILabel	Enter Country	
UILabel	Select Winery	
UILabel	Select Rating	

To get started, select the `FirstViewController`, and from the Component Pallet on the lower right side of Xcode, select the `ImageViewer` (`UIImageView`) and drop it on the IB canvas. Next, select and open the Constraints tool in the bottom right of the IB window and click on the top and left beams; select both the width and height constraints (Figure 5-7). Click on the Add 4 Components button to set the constraints.

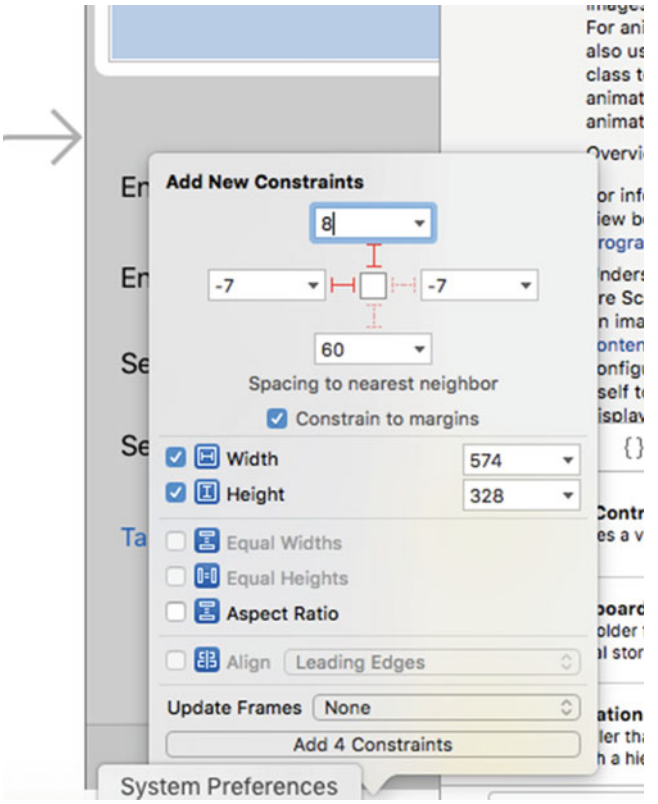

Figure 5-7. *Constraints in IB for ImageViewer*

Continue to build the layout by adding labels (`UILabel`) and fields (`UITextField`) to the UI, like in Figure 5-6. Also add a slider to set the rating, and finish off by adding three buttons (`UIButton`): one to take the photo, one to display a `UIPickerView` of possible wineries, which will have to populate first before adding wines, and one button to insert the record.

As before, select the various components and set the constraints so that they will adapt to the target device. You will need to repeat the process for the `SecondViewController`. This second scene will allow a user to enter new wineries to the database.

The second view controller will include the following elements:

Element	Name	Connection Type
UITextField	EnterWineryField	IBOutlet
UITextField	EnterCountryField	IBOutlet
UITextField	EnterRegionField	IBOutlet
UILabel	Enter Winery Name	

(*continued*)

Element	Name	Connection Type
UILabel	Enter Country	
UILabel	Enter Region	
UIButton	InsertWineryBtn	IBAction

Before moving on to the data model, we need to create connections for the UI `IBOutlet` and `IBAction` elements. To create the connections, first select the `FirstViewController` scene and click on the Identity inspector, which will open the `FirstViewController` file next to the IB window (Figure 5-8).

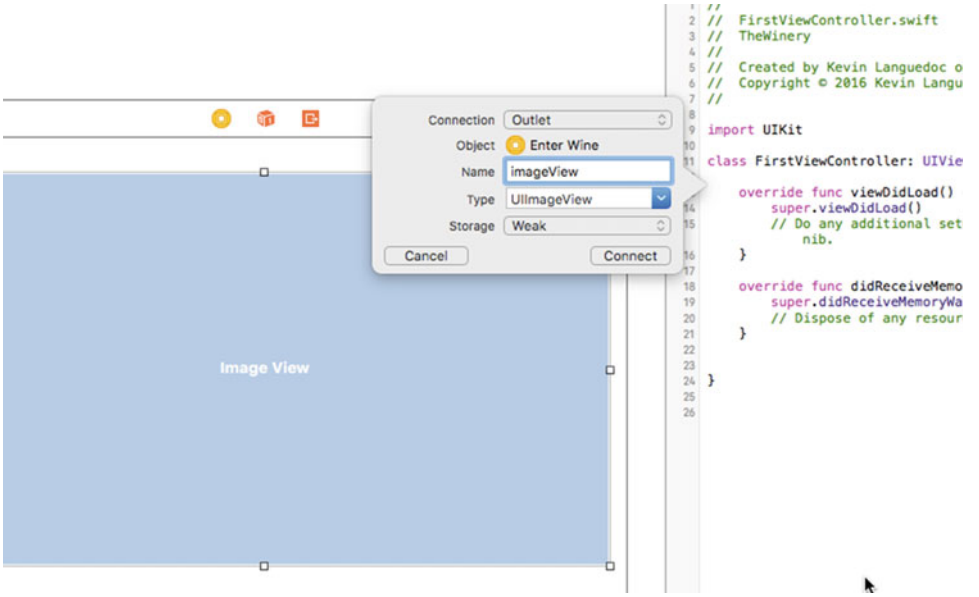

Figure 5-8. *Adding an IBOutlet to the FirstViewController*

Select the `imageView` element and hold down the Control key while dragging it to the open file. When you release the mouse, a popup will appear allowing you to enter the name of the `imageView` element. Click the Connect button, which will create the property.

Repeat the same process for the remaining `UITextFields`. For the `selectWineryField UITextField`, select Attribute inspector and unselect the "Enable" property for the Read Only field. The buttons are `IBAction` so you need to perform an extra step in the Connection popup. For the `IBAction`, you need to change the connection type from outlet to action.

Once you have finished the connections for the first view controller, repeat the process with the second view controller. Select the `SecondViewController` and open the Identity inspector corresponding to the scene and drag and drop the connections. I will add the logic later in the chapter.

Creating the Data Model

Creating our data model requires a series of steps, which are outlined in the following sections.

Add the Wineries Database

With the bridge set up, I will add the `Wineries.sqlite` database. I will create this database through the `AppDelegate` class in the `didFinishWithOptions` function. The code is provided later. The pattern is quite simple, and we have seen this code at several points in the preceding chapters.

I get a handle on the Documents directory using the `FileManager.SearchPathDirectory.documentDirectory` property through the `URLForDirectory` function of the `FileManager` class. I then append the `wineries.sqlite` file name to the `dbPath` variable and pass this value to the `sqlite3_open` function along with the pointer to the sqlite3 database engine.

Each time the app runs after the first time, sqlite3 will simply open the database and establish a connection.

```
var srcPath:URL = URL.init(fileURLWithPath: "")
        var destPath:String = ""
        let dirManager = FileManager.default
        let projectBundle = Bundle.main

        do {
            let resourcePath = projectBundle.path(forResource: "thewinery", ofType:
            "sqlite")
            let documentURL = try dirManager.urls(for: .documentDirectory, in:
            .userDomainMask)

            srcPath = URL(fileURLWithPath: resourcePath!)

            destPath = String(describing: documentURL)

            if !dirManager.fileExists(atPath: destPath) {

                try dirManager.copyItem(at: srcPath, to: URL(fileURLWithPath: destPath))

            }

        } catch let err as NSError {
            print("Error: \(err.domain)")
        }
```

Add the Wine Type

```
import Foundation

class Wine: NSObject {
    var id:Int32 = 0
    var name:String = ""
    var rating:String = ""
    var image:Data = Data()
    var producer:Int32 = 0

    override init(){

    }
}
```

Add the Wineries Type

```
import Foundation

public class Wineries:NSObject{
    var id:Int32 = 0
    var name:String = ""
    var country:String = ""
    var region:String = ""
    var volume:Double = 0.0
    var uom:String = ""

    override init(){

    }
}
```

Add the Database Schema

For the schema proper, I will use a script file, `wineries.sql`, that I created using the Empty file template under the iOS category from the Other section. I added the following SQL script to create two tables:

- Wine
- Wineries

```
CREATE TABLE IF NOT EXISTS main.wineries(
    id integer primary key autoincrement not null,
    name varchar,
    country varchar,
    region varchar,
    volume float,
    uom varchar
)

CREATE TABLE IF NOT EXISTS main.wine(

    id integer primary key autoincrement not null,
    name varchar,
    rating integer
    producer_id integer foreign key references wineries(id)
)
```

To build the table schema for inserts, I will read the file and execute the query using the `sqlite3_prepare_v2` function along with the `sqlite3_step` and `sqlite3_finalize` functions. I will provide the code to execute these SQL scripts in the next section for the `WineryDAO` controller class.

Creating the Controllers

In this section, I will create the controllers.

Add the WineryDAO Class

To create this class, select the Swift file template from the New File interface under the iOS > Source category. Name the file WineryDAO and add the file to the project. In the class, add the `NSObject` subclass and the following functions:

- `buildSchema`
- `createOrOpenDatabase`
- `insertWineRecord`
- `insertWineryRecord`

The class signature should resemble this:

```
class WineryDAO: NSObject{}
```

Before getting to the functions in this class, we need to define some variables, which are listed here. The `dbName` is the SQLite database file name; the `db` is the pointer to the SQLite instance; the `sqlStatement` is the pointer for the `sqlite3_statement` instance; `errMsg` is an `UnsafeMutablePointer` to capture any operational errors thrown by SQLite; `sqlite_static` and `sqlite_transient` are `unsafeBitCast` pointers; `dbPath` is the path to the SQLite database file in the sandbox; and `errStr` is a String variable to manage error messages.

```
let dbName:String=" winery.sqlite "
var db:COpaquePointer?=nil
var sqlStatement:COpaquePointer?=nil
var errMsg: UnsafeMutablePointer<UnsafeMutablePointer<Int8>>? = nil
internal let SQLITE_STATIC = unsafeBitCast(0, to:sqlite3_destructor_type.self)
internal let SQLITE_TRANSIENT = unsafeBitCast(-1, to:sqlite3_destructor_type.self)
var dbPath:NSURL = URL()
var errStr:String = ""
```

The init() function init()

The `init` function is a standard initializer in the Swift language. As you might imagine, it allows the program to set up any variables as the instance of the class is created and loaded into memory. To use the `init` function, you need to override it. In this class, I use the `init` to ensure that the database path is set. I could have also added `sqlite3_open` to actually open the database, but opted instead to place that operation in its own function.

Remembering the corruption issue I mentioned in chapter 3, if you open the same database file more than once at the same time, hence having different threads, you risk corrupting the database. The safest way to ensure that there are no corruption issues is to open and close the database after each operation. The other way is to use `sqlite3_open_v2`, which allows you to set some additional parameters. Using this variation of the function, you can specify if the database will be opened in read mode or read/write mode, and you can use the `SQL_OPEN_FULLMUTEX` flag to open the database in serialized mode, which offers the most protection against file corruption. The other multi-threading option is `SQLITE_OPEN_MUTEX`, which also opens the database in a multi-thread mode as long as the single-thread mode was not set when the database was first created. The default is `SQLITE_OPEN_NOMUTIEX`, which is single-thread operation.

For this app, I will open and close the database with each operation.

```
  override init() {
        /*
         Create SQLite Winery.sqlite database in Documents directory

         */
let dirManager = FileManager.default()
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)
dbPath = try! directoryURL.urlByAppendingPathComponent(dbName)
} catch let err as NSError {
            print("Error: \(err.domain)")
        }
    }
```

The buildSchema Function

With this app, I wanted to try a different approach to building a database schema, so I decided to add the schema definition to a file that is loaded when the app is loaded. This function retrieves the `.sql` file from the Resource directory and executes the queries, which were provided in the previous section. I am using the `sqlite3_exec` function as it nicely encapsulates the different operations to execute a query, namely `sqlite3_prepare_v2`, `sqlite3_step`, and `sqlite3_finalize`. Once the query is executed, the database is closed.

The function is called from the `AppDelegate` when the app is loaded:

```
func buildSchema()->Void{
        if let filepath = Bundle.main.pathForResource("wineries", ofType: "sql") {
            do {
                let script = try NSString(contentsOfFile: filepath, usedEncoding: nil) as
                String
                print(script)
                if sqlite3_open(dbName, &db)==SQLITE_OK {
                    if sqlite3_exec(db, script.cString(using: String.Encoding.utf8)!, nil,
                    nil, errMsg) != SQLITE_OK{
                        print( errStr = String (cString: sqlite3_errmsg(db))!)
                    }
                }else{
                    print("Could not open database " + String(cString.sqlite3_errmsg(db))!)
                }
            } catch let error as NSError {
                print(error.localizedDescription)
            }
        } else {
            print("file not found")
        }
sqlite3_close(db)
}
```

The createOrOpenDatabase Function

The `createOrOpenDatabase` function is called from the `AppDelegate` function when the app is loaded to ensure that the database is present and the schema script is executed. We could encapsulate this operation with "file exists" check to not unnecessarily open and execute the schema script every time the app is loaded.

```
func createOrOpenDatabase()->Enums.SQLiteStatusCode{
 return Enums.SQLiteStatusCode(rawValue: sqlite3_open(dbPath.absoluteString!.cString (using:
String.Encoding.utf8)!, &db))!
    }
```

The insertWineRecord Function

This function is called from the `FirstViewController` through the `insertRecordAction IBAction`. First, we define an insert query and assign it to a constant called `sql`. Next, we open the database that is needed and ensure that it is open with the `SQLITE_OK` status code before executing the `sqlite3_prepare_v2` function, which takes the `sqlite` pointer, the `sql` query, and the `sqlStatement` pointer as arguments.

The special feature of this function is `wine.image`, which converts `NSData` into a blob for insertion into the database. To convert it, we need to get the bytes from the `NSData` and then supply the length of the bytes and convert this to an Int32.

After opening the database and preparing the query statement, we bind the input values to the columns using the appropriate binding functions for the required data type. We execute the query using the `sqlite3_step` function and then clean up the operation using the `sqlite3_finalize` function. All that remains is to close the database using the `sqlite3_close` function:

```
    func insertWineRecord(_ wine:Wine)->Enums.SQLiteStatusCode{
        let sql:String = "INSERT INTO main.wine VALUES(?, ?, ?, ?)"

        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, sql.cString (using:String.Encoding.utf8)!, -1,
            &sqlStatement, nil)==SQLITE_OK){
                sqlite3_bind_value(sqlStatement, 1, nil)
                sqlite3_bind_text(sqlStatement, 2, wine.name.cString(using: String.Encoding.
                utf8)!, -1, SQLITE_TRANSIENT)
                sqlite3_bind_int(sqlStatement, 3, wine.rating)
                sqlite3_bind_int(sqlStatement, 4, wine.producer)
                sqlite3_bind_blob(sqlStatement, 5, wine.image.bytes,Int32(wine.image.count),
                SQLITE_TRANSIENT)
                sqlite3_step(sqlStatement)
                sqlite3_finalize(sqlStatement)
            }
}else{
            print(String(cString: sqlite3_errmsg(db))!)
            return Enums.SQLiteStatusCode.error
        }
         sqlite3_close(db)
        return Enums.SQLiteStatusCode.ok
    }
```

The insertWineryRecord Function

This function inserts winery records from the SecondViewController via the insertWineryBtn. The pattern of this function is identical to the function to insert wines, except for the binding columns, which include binding values for integers as well as for doubles and strings. As we have seen before, the strings need to be converted to string chars using the cStringUsingEncoding property and NSUTF8StringEncoding:

```
    func insertWineryRecord(_ vintner:Wineries) -> Enums.SQLiteStatusCode {
    let sql:String = "INSERT INTO main.winery VALUES(?, ?, ?, ?, ?)"
    if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
        if(sqlite3_prepare_v2(db, sql.cString(using: String.Encoding.utf8)!, -1,
        &sqlStatement, nil)==SQLITE_OK){
            sqlite3_bind_value(sqlStatement, 1, nil)
            sqlite3_bind_text(sqlStatement, 2, vintner.name.cString(using: String.
            Encoding.utf8)!, -1, SQLITE_TRANSIENT)
            sqlite3_bind_text(sqlStatement, 3, vintner.country. cString(using: String.
            Encoding.utf8)!, -1, SQLITE_TRANSIENT)
            sqlite3_bind_text(sqlStatement, 4, vintner.region. cString(using: String.
            Encoding.utf8)!, -1, SQLITE_TRANSIENT)
            sqlite3_bind_double(sqlStatement, 5, vintner.volume)
            sqlite3_bind_text(sqlStatement, 6, vintner.uom. cString(using: String.
            Encoding.utf8)!, -1, SQLITE_TRANSIENT)
            sqlite3_step(sqlStatement)
            sqlite3_finalize(sqlStatement)
        }
    }else{
        print(String(cString: sqlite3_errmsg(db))!)
        return Enums.SQLiteStatusCode.error
    }
    sqlite3_close(db)
    return Enums.SQLiteStatusCode.ok
}
```

The FirstViewController

All iOS UIs have a view controller associated with the scenes in the storyboard. When the app was created, the template created a view controller for each of the scenes in the storyboard.

To implement the UIImagePickerController and the UIPickerView, the app will need to implement UIImagePickerControllerDelegate, UIPickerViewDelegate, and UIPickerViewDataSource delegates and data sources. With each of these, the app needs to implement a certain number of required functions, which we will look at later.

From the code that follows, you can see that we implement several variables and constants. The imageSelector variable is a UIImagePickerController type. It is used to present the UIImagePicker, which will use the building camera as its data source. The imageData is a variable that holds the image data from the camera. It is an NSData type that handles binary data on the iOS platform. The dbDAO is an instance of the WineryDAO that the app will use to interface with the database. wine and vintner are instances of the Wine and Wineries classes that represent the data. The wineriesArray is a mutable array of the Wineries type. This array is the data source for the UIPickerView. Finally, the IBOutlets were discussed earlier and are the connections in the UI.

```
class FirstViewController: UIViewController, UINavigationControllerDelegate,
UIImagePickerControllerDelegate, UIPickerViewDelegate, UIPickerViewDataSource
var imageSelector: UIImagePickerController!
var imageData:Data = Data()
var dbDAO:WineryDAO = WineryDAO()
var wine: Wine = Wine()
var wineriesArray = [Wineries]()
var wineriesPickerView: UIPickerView = UIPickerView()
var vintnor:Wineries = Wineries()
@IBOutlet weak var selectWineryField: UITextField!
@IBOutlet weak var imageView: UIImageView!
@IBOutlet weak var wineNameField: UITextField!
@IBOutlet weak var countryNameField: UITextField!
```

Add Photo Capture Functionality

As we have seen in the "Creating the UI for Inserting" section earlier, we need to be able to capture and insert images (preferably wine-related ones). We have set up the UI and added the `IBActions`, and now we need to add the logic to the `FirstViewController`.

We start by initializing the `imageSelector` object and setting its delegate to the `FirstViewController`, specifying that its source will be the building camera. Finally, the picker is loaded onto the view stack using the `presentViewController`:

```
@IBAction func takePhoto(_ sender: AnyObject) {
        imageSelector =  UIImagePickerController()
        imageSelector.delegate = self
        imageSelector.sourceType = .camera
        present(imageSelector, animated: true, completion: nil)
    }
```

The `imagePickerController` function is part of the image-picker protocol. It tells the delegate that a picture was selected. We need to call the `dismissViewControllerAnimated` to close the picker and return the selected image, which we store in the `imageView.image` property:

```
func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo
info: [String : AnyObject]){
        imageSelector.dismiss(animated: true, completion: nil)
        imageView.image = info[UIImagePickerControllerOriginalImage] as? UIImage
        imageData = UIImagePNGRepresentation(imageView.image!)!
}
```

Once we have the image stored in the `imageView`, it is passed to the `imageData` using `UIImagePNGRepresentation`, which takes `imageView.image` as a parameter. The image data is passed and stored in the database using `insertRecordAction` function, which we will look at next.

Add the Insert Function

The `insertRecordAction` calls the `dbDAO` object, passing in the required parameters to insert it into the database. Most of the data used for the function comes from other functions and is assigned to the `Wine` instance class's properties.

```
    @IBAction func insertRecordAction(_ sender: AnyObject) {

        dbDAO.insertWineRecord(wine))

    }
```

The viewDidLoad Function

This standard `ViewController` function is used after the scene is loaded onto the view stack or, in our case, when the app is launched. The function provides a channel to set up the `wineriesPickerView` by setting up the delegate and assigning the `wineriesArray` data source. Once the setup is complete, the `UIPicker` is added as a subview to the current view controller.

```
override func viewDidLoad() {
        super.viewDidLoad()
        //build data source

        self.wineriesPickerView.isHidden = true
        self.wineriesPickerView.dataSource = self
        self.wineriesPickerView.delegate = self
        self.wineriesPickerView.frame = CGRect(x:100, y:100, width: 100, height: 162)
        self.wineriesPickerView.backgroundColor = UIColor.black()
        self.wineriesPickerView.layer.borderColor = UIColor.white().cgColor
        self.wineriesPickerView.layer.borderWidth = 1
        self.wineriesArray = dbDAO.selectWineriesList()

        //other pickerView code like dataSource and delegate
        self.view.addSubview(wineriesPickerView)

    }
```

Add the Rating UISlider Functionality

The `wineRating` function manages the interaction with the `UISlider`. The `sender` argument returns the float value based on the user's selection, and then this value is converted to an Int32 and assigned to the `wine.rating` property:

```
@IBAction func wineRatingSlider(_ sender: AnyObject) {
        let ratingValue:float_t = sender.value
          wine.rating  = Int32(ratingValue)
}
```

The SecondViewController

The second view controller manages the entry of the wine producers. A user will need to add wineries before wines when the app is running. It is much simpler than the first view controller, as we have to deal with just five `UITextFields` that we defined with the UI. We call the `insertWineryRecord` method of the `dbDAO` class from the `insertWineryBtn IBAction` function, passing the values from the `UITextFields`. We create the `winery` instance and assign the values before passing the object as the only argument for the

insertwineryRecord method. We could have simply passed the values for the UITextFields directly, but I will need to share those values later.

```
@IBOutlet weak var wineryNameField: UITextField!
@IBOutlet weak var countryNameField: UITextField!
@IBOutlet weak var regionNameField: UITextField!
@IBOutlet weak var enterVolume: UITextField!
@IBOutlet weak var enterUoM: UITextField!

var dbDAO:WineryDAO = WineryDAO()
var winery:Wineries = Wineries()

@IBAction func insertWineryBtn(_ sender: AnyObject) {
        winery.name = wineryNameField.text!
        winery.country = countryNameField.text!
        winery.region = regionNameField.text!
        winery.volume = Double(enterVolume.text!)!
        winery.uom = enterUoM.text!
        dbDAO.insertWineryRecord(winery)
    }
...
}
```

All that is left to do is run the app and insert some records.

Running the App

To finish up this chapter, let's fire up the app. To properly test this app and the camera, you need to deploy it on an iPhone. I am using iPhone 6 Plus. To deploy to an iPhone, you need to plug in your phone, provision it on the Apple Developer website, then open the settings on the iPhone. Under the General section, look for the Device Management section and click on "Trust link and enable the app for development."

Inserting Records

It is best to start by entering wineries so that you have some data to choose from in the wineries picker in scene one. With the app running, tap the Winery button in the menu bar.

Inserting Wineries

Figure 5-9 is the Wineries scene. Enter a winery and tap the Insert Winery button.

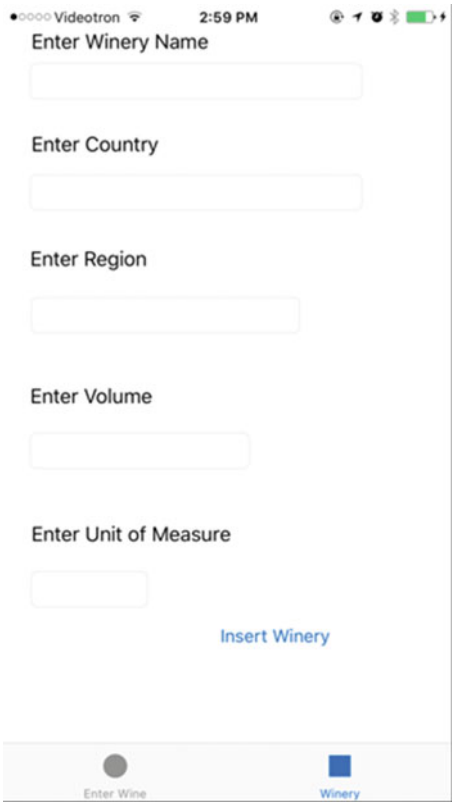

Figure 5-9. *The Winery scene*

Inserting Wines

When using the camera, the device will ask you for permission to access the camera. Click OK to proceed (Figure 5-10).

Figure 5-10. *Access camera from app*

Once you have accessed the camera for the first time, you will be presented with the camera as usual. I took a photo of a Chateauneuf du Pape—la fiole du pape (Figure 5-11). Click on the shutter button. You can cancel to opt to keep the photo. If you accept the photo, it will appear in the `UIImage` viewer. Enter the remaining information and click the Insert Wine button.

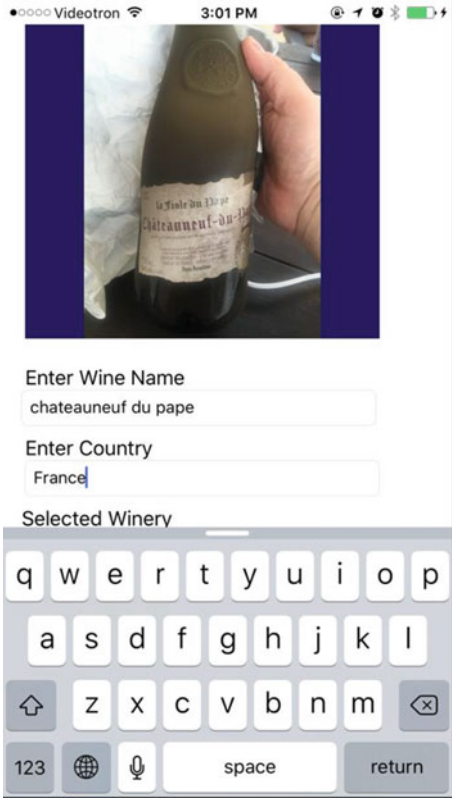

Figure 5-11. *Entering wine information with image*

Summary

That's it for this chapter. We explored the SQLite `INSERT` API including the `OR` clause. I provided sample code to demonstrate how to handle binary data like images, videos, and audio files, and we started to build the Winery app, which will demonstrate the general SQL CRUD operations in SQLite.

In the next chapter, we will continue to build the Winery app by adding `SELECT` operations to the app and testing the selection process.

CHAPTER 6

Selecting Records

The SELECT statement is the workhorse of the SQL language no matter what platform you are using, and it is no different with SQLite. The SELECT statement is used to perform queries against a SQLite table or view. It is also used to populate views in SQLite databases, just as in other relational database systems.

This chapter demonstrates how to use SELECT in Swift by binding data from columns in SQLite tables and assigning them to Swift data-type variables.

The examples demonstrate how to perform SELECT queries to return text and numeric data as well as audio, image, and video data. The Winery app will be retrofitted with SELECT queries to display the list of wineries for the UIPickerView as well as display wines that are stored in the database.

In this chapter, we will explore the following:

- the SELECT statement syntax
- various SELECT use cases
- binding data types
- inner SELECT
- joins
- inline SELECT
- adding SELECT queries to the Winery app

Column Data Types

Here is a list of the data-type binding functions that you can use to assign values from a SELECT query to a C-based variable, a Swift variable, or a custom object's properties:

- `sqlite3_column_blob(sqlite3_stmt, int iCol);`
- `sqlite3_column_bytes(sqlite3_stmt, int iCol);`
- `sqlite3_column_bytes16(sqlite3_stmt, int iCol);`
- `sqlite3_column_double(sqlite3_stmt, int iCol);`
- `sqlite3_column_int(sqlite3_stmt, int iCol);`
- `sqlite3_column_int64(sqlite3_stmt, int iCol);`
- `sqlite3_column_text(sqlite3_stmt, int iCol);`
- `sqlite3_column_text16(sqlite3_stmt, int iCol);`

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_6

- `sqlite3_column_type(sqlite3_stmt, int iCol);`
- `sqlite3_column_value(sqlite3_stmt, int iCol);`
- `sqlite3_bind_zeroblob(sqlite3_stmt, int, int iCol)`

The SELECT Statement

The SQLite `SELECT` statement is used to extract data and information from a SQLite database. The `SELECT` statement is the most complicated in the SQL language, mostly because of all the permutations.

The basic syntax of a `SELECT` query is:

```
SELECT column(s) from main.table
```

You could also use a wildcard to return all the columns in the table:

```
SELECT * from main.table
```

If you wanted or needed to return only a subset of the data from a table, you could use a `WHERE` clause, as in the following example:

```
SELECT * from main.table WHERE column1 = 'some value'
```

The `WHERE` clause could evaluate either custom or standard functions. You could also use the `IN` clause. For example:

```
SELECT id, column1 FROM main.table WHERE column1 IN (SELECT col_id, column1 from main.
second_table WHERE col_id=id)
```

The `SELECT` statement can also include another `SELECT` statement instead of a column; this is known as an *inline select* query, as the following illustrates:

```
SELECT id, column1, (SELECT column FROM main.table )
```

Selecting Data

The first use case will demonstrate how to perform a basic, boilerplate `SELECT` without a `WHERE` clause. The `SELECT` in SQLite is performed through the `select-stmt` function. While you don't interface directly with this function, meaning you don't call this function directly when performing `SELECT` queries, the function does support the standard SQL API on `SELECT`. The following example follows the same pattern of accessing the SQLite database in the Documents path, opening it, and performing a `SELECT` query.

In this example, the `SELECT` query retrieves a result set of contacts and assigns them to a custom Swift data type. The `SELECT` query string is defined using a Swift String variable. Look at the database parameter in the `sqlite3_open` statement as an example of how to use an `NSString` to build a `SELECT` query and pass it to the `sqlite3_stmt` function to be executed by the database engine.

The `SELECT` statement SQL query is passed to the `sqlite3_prepare_v2` function before it can be executed by the database engine. To access the result set, you will need to use the `sqlite3_step` function along with the `SQLITE_ROW` constant to loop through the results; as long as there are records in the result set `SQLITE_ROW` remains true.

Once the result set is exhausted, `sqlite3_finalize` is called to clean up the prepared statement, and `sqlite3_close` closes the database. String data is stored in the database using UTF-8 encoding, so when you need to assign a string value to a Swift variable, you need to use the `NSUTF8StringEncoding` method from the `String` class.

The database values that are retrieved from the result set must be passed or assigned to a variable or custom data type using a special function like `sqlite3_column_text`. There is a `sqlite3_column` for each data type, including blobs. Access to the columns in a table is done through a zero-based array, as in the example code that follows.

Once the values are assigned to either a Swift variable or a custom data-type property, if you wanted to display them in a `UIPickerView` or a `UITableView` you would need to add the values to an array or `NSDictionary`, which is the preferred, most commonly used data source method for these list-based objects. See the example code here:

```
class SelectWithSwift: NSObject {

    var dbPath:URL = URL()
    let db:COpaquePointer?=nil
    let sqlite3_stmt:COpaquePointer?=nil

    override init() {
        let dirManager = FileManager.default()
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManagerSearchPathDirec
            tory.documentDirectory, in FileManager.SearchPathDomainMask.UserDomainMask,
            appropriateFor: nil, create: true)

            dbPath = try! directoryURL.appendingPathComponent("Contacts.sqlite")

        } catch let err as NSError {
            print("Error: \(err.domain)")
        }
    }

    func simpleSelect(){
        var contactList = [Person]()
        let sql:String = "Select id, name, address, city, zip,country from contact"

        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, sql.cString (using:String.Encoding.utf8)!, -1,
            &sqlStatement,  nil) == SQLITE_OK)
            {
                while (sqlite3_step(sqlStatement)==SQLITE_ROW) {
                    let contact:Person = Person()
                    contact.name = String(cString:UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 0)))!
                    contact.address = String(cString:UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 1)))!
                    contact.city = String(cString:UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 2)))!
                    contact.zip = String(cString:UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 3)))!
                    contact.country = String(cString:UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 4)))!contactList.append(contact)

                }
```

```
        }
        }
        sqlite3_finalize(sqlStatement);
        sqlite3_close(db);
    }
```

■ **Note** Obviously, if you are only retrieving one value from the database, like we will see in the example that uses a `WHERE` clause, you don't need to loop through the result set.

Using a Dynamic WHERE Clause

The next use case will demonstrate how to use a `WHERE` clause. Using a `WHERE` clause with your `SELECT` query is a powerful feature in SQL. The `WHERE` clause in SQLite acts no differently than it does in any other database platform. The trick is how to dynamically pass values to the `WHERE` clause's Boolean variables.

Basically, there are two formats you can use to pass values to the `WHERE` clause's variables. You can build an `NSString` string or use string formatters, like in the following example:

```
    func SelectWhereContactInformation(){
        var contactList = [Person]()
        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            let sql:String = "Select id, name, address, city, zip,country from contact where
            name=?";
            if(sqlite3_prepare_v2(db, sql, -1, &sqlStatement,  nil) == SQLITE_OK)
            {
                // input value for the WHERE clause unless you are
                sqlite3_bind_text(sqlStatement, 0, sql, -1, nil);

                //Bind each column in the table to the property names
                //for the contact object.
                while (sqlite3_step(sqlStatement)==SQLITE_ROW) {
                    let contact:Person = Person()
                    contact.name = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 0)))!
                    contact.address = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 1)))!
                    contact.city = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 2)))!
                    contact.zip = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 3)))!
                    contact.country = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 4)))! contactList.append(contact)
                }
            }
}

    sqlite3_finalize(sqlStatement);
    sqlite3_close(db);
    }
```

In the preceding code, the value for the name variable in the WHERE clause gets updated from the sqlite3_bind_text value, which replaces the "?" placeholder in the sql: String variable. You then pass the string to the sqlite3_prepare_v2 method by converting the sql query string to UTF-8, as follows:

```
sql.cString (using: String.Encoding.utf8)!
```

Perform a SELECT using a Sub-Query

The next example I want to show you is how to do a SELECT using a sub-query. Actually, a sub-query is a result set of a SELECT query, and you perform your SELECT query on this result set rather than on the complete table.

Consider this basic example, which shows the basic syntax of a sub-query:

```
SELECT id, name, sales, sales_quota, region FROM Sales WHERE id IN
        (SELECT id FROM Sales WHERE closed_deals > 10000000)
```

From a Swift standpoint, this wouldn't be any different than a normal query, and you could pass the value of the WHERE expression as in previous examples. Formatted as a *String* variable assignment, however, this would look like the following:

```
let intValue:Int = 1000000
        let subquery:String = "SELECT id, name, sales, sales_quota, region FROM Sales WHERE
        id " +
        "IN (SELECT id FROM Sales WHERE closed_deals > \(intValue)"
```

Or you could bind the value as you have seen in previous examples:

```
let intValue:Int = 1000000
        let subquery:String = "SELECT id, name, sales, sales_quota, region FROM Sales WHERE
        id " +
        "IN (SELECT id FROM Sales WHERE closed_deals > ?"
        // sqlite3_prepare_v2 code
sqlite3_bind_int(sqlStatement, 0, Int32(intValue))
```

There are a couple of rules that you need to keep in mind when working with sub-queries. First, the sub-query can only have one column in the SELECT clause unless several columns in the main SELECT match up with the same columns in the sub-query. Second, sub-queries must be enclosed in parentheses. Third, you cannot include an ORDER BY clause in the sub-query. However, you can use ORDER BY in the main SELECT. Fourth, a sub-query that returns more than one row must use a multiple-value operator like IN. Finally, you can only use the BETWEEN clause in a sub-query.

Perform a SELECT using Joins

Joins are an essential operation in SQL, or, more precisely, in a relational database, since we look up data in different tables based on the relationship between those tables. SQLite implements joins in a fashion similar to that of other SQL derivatives. In this section, we will explore the standard join patterns or clauses.

Using an INNER Join

The most common join is the inner join. The inner join uses a matching key (primary–foreign) to create a relationship between two tables. The key doesn't have to have the same name, only the same data type:

```
Select c.id, c.name, r.hotel, r.checkin, r.checkout from customer as C JOIN
reservations as r ON on c.id = r.custid
```

You could also use a `WHERE` clause to achieve the same result:

```
Select c.id, c.name, r.hotel, r.checkin, r.checkout from customer as C JOIN
reservations as r WHERE c.id = r.custid
```

Using a CROSS Join

The `NATURAL JOIN` is similar to the inner join except that the query presumes and expects to find an identical column in each joined table and will any matching column (name and data type) that it finds:

```
Select c.id, c.name, r.hotel, r.checkin, r.checkout from customer as C NATURAL JOIN
reservations AS r
```

Using the OUTER Join

With the left outer join, all the records from the left table—or, in other words, all the records from the table immediately following the `FROM` keyword—even if there are no records in the table on the right.

```
Select c.customerid, c.name, v.visits from customer as C LEFT JOIN Visits v ON c.id=v.custid
```

Select and Display Images

This example demonstrates how to select binary data like images and convert that binary data into the `UIImage` data type that can be displayed in a `UIImageView`:

```
func selectImages(_ filename:String)->Array<UIImage>{
    var imageArr = [UIImage]()

    if(!(sqlite3_open(dbPath.path!, &db) == SQLITE_OK))
    {
        print("An error has occured.");
        return imageArr;
    }else{
        let sql:String = "SELECT id, filename, image FROM images where filename=?";
        if(sqlite3_prepare(db, sql, -1, &sqlStatement, nil) != SQLITE_OK)
        {
            print("Problem with prepared statement");
        }else{
            //WHERE parameter value
            sqlite3_bind_text(sqlStatement, 1,filename, -1,SQLITE_TRANSIENT)
            while (sqlite3_step(sqlStatement)==SQLITE_ROW) {
                let contact:Person = Person()

                let raw:UnsafePointer = sqlite3_column_blob(sqlStatement, 3);
                let rawLen:Int32 = sqlite3_column_bytes(sqlStatement, 3);
                let data:Data = Data(bytes: raw, count: Int(rawLen))
```

```
                    //Convert the binary data into an UIImage
                    contact.avatar = UIImage(data: data)!
                    imageArr.append(contact.avatar)

                }
            }

        }
       return imageArr
    }
```

Select and Playback Audio Records

```
func selectAudioWithPlayback(_ selectedFile:String)->Data{

        var audio:Data = Data()
        let sql:String = "SELECT audioData FROM audios wherefileName= \(selectedFile)"

        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, sql, -1, &sqlStatement, nil)==SQLITE_OK){
                while (sqlite3_step(sqlStatement)==SQLITE_ROW) {
                    let raw:UnsafePointer = sqlite3_column_blob(sqlStatement, 3);
                    let rawLen:Int32 = sqlite3_column_bytes(sqlStatement, 3);
                    audio = Data(bytes: raw, count: Int(rawLen))
                }
            }
            sqlite3_finalize(sqlStatement);
            sqlite3_close(db);
            return audio
        }
    }
```

■ **Note** The following code is for representation only, meaning that it cannot be run from here. Rather, you would need to include it in a view controller attached to an `IBAction`. The self-reference that follows refers to the view controller.

```
func playback(_ selectedAudioFile:NSData){

        do {
            var audioPlayer = try AVAudioPlayer(data:selectedAudioFile)
            audioPlayer.delegate = self;
            audioPlayer.prepareToPlay()
            audioPlayer.play()
        } catch let err as NSError {
            print("Error: \(err.domain)")
        }

    }
```

Select and Display Video Records

Selecting videos for playback follows the same pattern as doing so for images and audio. To perform a SELECT, extract the data using the blob and store the binary data in an NSData object or objects that can be stored in an NSMutableArray or NSDictionary, for example.

However, you can't convert or pass the NSData object directly to the MPMoviePlayerController object for playback like you can with images or audio. You will need to convert the NSData object to a movie format like mp4 by saving it to a movie file in the Documents directory. To save the NSData to an mp4 file, simply write out the data to a file with the .mp4 extension; NSData is simply a container for data, so no transformation is needed.

Alternatively, you could store the NSData to an NString and convert this string to an NSURL object, which you could then use to initialize the MPMoviePlayerController using the initWithContentURL method.

Adding SELECT Functionality to the Winery App

In this section of the chapter, I will add the SELECT functionality to the Winery app. This functionality will include a function to select a winery when entering a wine, another function to select and display a list of wines from the database, as well as a function to display the different wineries. This third function will be enhanced in a later chapter on updating records. Let's start with the wineriesPickerView.

Add the SelectWineries UIPicker

This little function displays the UIPicker so that a user can select a winery from the list:

```
    @IBAction func selectWineryButton(sender: AnyObject) {
         self.wineriesPickerView.hidden = false
    }
```

The viewDidLoad Function

This standard ViewController function runs after the scene is loaded onto the view stack, or in our case when the app is launched. The function provides a channel through which to set up the wineriesPickerView by setting up the delegate and assigning the wineriesArray data source. Once the setup is complete, the UIPicker is added as a sub-view to the current view.

```
override func viewDidLoad() {
        super.viewDidLoad()
        //build data source

        self.wineriesPickerView.isHidden = true
        self.wineriesArray = dbDAO.selectWineriesList()
        self.wineriesPickerView.dataSource = self
        self.wineriesPickerView.delegate = self
        self.wineriesPickerView.frame = CGRect(x:19, y:243, width: 336, height: 216)
        self.wineriesPickerView.backgroundColor = UIColor.white()
        self.wineriesPickerView.layer.borderColor = UIColor.blueColor().cgColor
        self.wineriesPickerView.layer.borderWidth = 1

        //other pickerView code like dataSource and delegate
        self.view.addSubview(wineriesPickerView)
    }
```

The UIPickerView Functions

The following functions are related to the UIPickerView. Some are for the delegate, which provides the interactivity to the UI component, while others identify the number of columns, the title, and the data source, as well as identify which row is selected.

The numberOfComponentsInPickerView function sets the number of columns the UIPicker will display. The pickerView function used along with the numberOfRowsInComponent argument sets the number of rows the UIPickerView will have in its data source. Typically, this return value is the data source array's count property. Next, the pickerView function used with titleForRow displays the actual list of values. didSelectRow returns the item that is selected by the user. Since I have a foreign key for the winery in the wine table, I need to get the ID for the winery, which I am doing with the vintner.id value; this is also the row id in the data source. Then, I set the hidden property to false to hide the picker view again from the user. The next two functions set the height and width of the rows. Finally, the typePickerViewSelected, although I added it in this section, is not a required function. Rather, it is an IBAction that I defined to display the UIPickerView when the user clicks on the Select Winery button.

```
    func numberOfComponents (_ in pickerView: UIPickerView) -> Int {
        return 1
    }

    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) ->
    Int {
        return wineriesArray.count
    }

    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent
    component: Int) -> String?   {
        vintnor = wineriesArray[row] as Wineries
        let pickernames = vintnor.name
        return  pickernames    }

    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent
    component: Int) {
        vintnor = wineriesArray[row] as Wineries
        vintnor.id = Int32(row)
        wineriesPickerView.isHidden = false
    }

    func pickerView(_ pickerView: UIPickerView, widthForComponent component: Int) -> CGFloat
{
        return 300.0
    }

    func pickerView(_ pickerView: UIPickerView, rowHeightForComponent component: Int) ->
    CGFloat {
        return 56.0
    }

    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent
    component: Int) {
        vintnor = wineriesArray[row] as Wineries
        selectWineryField.text = vintnor.name
```

```
        wineriesPickerView.endEditing(true)
        wineriesPickerView.hidden = true
    }
    func pickerView(_ pickerView: UIPickerView, viewForRow row: Int, forComponent component:
    Int, reusing view: UIView?) -> UIView {
        let test:UILabel = UILabel()
        let titleData = wineriesArray[row].name
        let myTitle = AttributedString(string: titleData, attributes: [NSFontAttributeName:U
        IFont(name: "Georgia", size: 15.0)!,NSForegroundColorAttributeName:UIColor.red()])
        test.attributedText = myTitle
        return test

    }
```

The selectWineriesList Function

This function will act as the data source for the `UIPickerView` that is activated when you enter the Select Winery field in the `FirstViewController`. This function will also be called by the `UITableViewController` to display the list of wineries. After you open the database, if it's not already open, the `SELECT` query string is passed to the `sqlite3_prepared_v2` function along with the `sqlite3-stmt` pointer, `sqlStatement`. While there are records, meaning while `sqlite3_step` returns `SQLITE_ROW`, the code will loop and assign the returned values to the wine object's properties.

Unbinding the returned values requires a couple of hoops, but as long you know how to deal with the return data types, everything will move along smoothly. `sqlite3_column_int` returns an Int32 value. You can convert this value to an Int, if needed, by wrapping `sqlite3_column_int` in an `Int()` function. Likewise, for `sqlite3_column_text`, which returns an `UnsafePointer` of an Int8 type, you just need to convert the returned value into an Int8 character, then convert this value to a String by using `fromCString`, which requires an Int8 value.

The returned columns are zero based, so the last parameter in the sqlite3 column-mapping functions is the column number whose value you need returned. See the following:

```
func selectWineriesList()->Array<Wineries>{
        var wineryArray = [Wineries]()

let sql:String = "Select name, country, region, volume, uom from main.winery"

        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, sql, -1, &sqlStatement, nil)==SQLITE_OK){
                while(sqlite3_step(sqlStatement)==SQLITE_ROW){
                    let vintnor:Wineries = Wineries.init()
                    vintnor.name = String(cString:UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 0)))!
                    vintnor.country = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 1)))!
                    vintnor.region = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 2)))!
                    vintnor.volume = sqlite3_column_double(sqlStatement, 3)
                    vintnor.uom = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 4)))!
                    wineryArray.append(vintnor)
                }
            }
```

```
        }
        sqlite3_close(db)
        return wineryArray
    }
```

The selectWineList Function

The `selectWineList` function acts the same way as the previous function. Its one special feature is the blob, or stored image. In order to convert the stored blob, you need the `NSData`, as we saw earlier in this chapter.

The `NSData` function needs the returned data as bytes, and it needs to know the length of the bytes. These values are stored in the `raw` and `rawLen` variables, respectively. The `raw` variable is an `UnsafePointer` because this is the return type of the `sqlite3_column_blob` function, and `rawLen` is an Int32 data type. You then pass these two values to the `NSData` initializer, which is then appended to the `wine.image` property. However, this value is still a binary format. Later in the chapter, the app will convert `NSData` into an `UIImage` and display it in the `UIImageView`. See the following:

```
func selectWineList()->Array<Wine>{
    var wineArray = [Wine]()
        let sql:String = "Select name, rating, image, producer from main.wine"
if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, sql.cString(String.Encoding.utf8)!, -1, &sqlStatement,
            nil)==SQLITE_OK){
                while(sqlite3_step(sqlStatement)==SQLITE_OK){
                    let wine:Wine = Wine.init()

                    wine.name = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 1)))!
                    wine.rating = sqlite3_column_int(sqlStatement, 2)
                    let raw:UnsafePointer = sqlite3_column_blob(sqlStatement, 3);
                    let rawLen:Int32 = sqlite3_column_bytes(sqlStatement, 3);
                    wine.image  = Data(bytes: raw, count: Int(rawLen))
                    wine.producer = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 4)))!
        wineArray.append(wine)
                }
            }
        }
        sqlite3_close(db)
        return wineArray
    }
```

The selectWineryByName Function

The wine producer property, which is the foreign key in the database, is an Int32 value. In order to get the winery name from the database for the Selected Winery field in the `FirstViewController`, the following function performs a `SELECT` statement with a `WHERE` clause, using the ID to locate the record. I pass the `WHERE` clause value using `sqlite3_bind_int` then retrieve the return value and assign it to the `name` property of the `vintner` object using the same method as before. Notice that the last parameter is a 1 instead of a 0. This is because in the `SELECT` query I am requesting two columns: the ID and the name.

```
func selectWineryByName(name:String)->String{
    let vintnor:Wineries = Wineries.init()
    let sql:String = "Select name from main.winery where name=?"
    if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
        if(sqlite3_prepare_v2(db, sql.cString(using: String.Encoding.utf8)!, -1,
        &sqlStatement, nil)==SQLITE_OK){
            sqlite3_bind_text(sqlStatement, 0, vintner.name.cString(String.Encoding.
            utf8)!, -1, SQLITE_TRANSIENT)
            if(sqlite3_step(sqlStatement)==SQLITE_OK){
                vintnor.name = String(cString:UnsafePointer<Int8>(sqlite3_column_
                text(sqlStatement, 1)))!
            }
        }
    }
    sqlite3_close(db)
    return vintnor.name
}
```

Modifying the UI for Displaying Records

With the `SELECT` queries in place, all that is needed is to add the view controllers to the UI and wire everything up. The app is going to need two `TableViewControllers` to display the list of records. These will have to be wired to the `TabBarController`.

Figure 6-1 provides a visual of the additional design elements that are added to the Winery app for the select and display functionality. These new elements include the following:

- Two table view controllers
- Two navigation controllers
- Two `UITableViewCellControllers`
- Three `UILabels` for the `WineList` cell prototype and five for the `WineryList` cell prototype

Adding the UITableViewControllers

We will first add the `UITableViewController` for the wine list, which I will call the Cellar. Select the `main.storyboard` to open it (Figure 6-1). Drag a `UITablewViewController` from the palette on the lower right onto the canvas. Select the `UITableViewController` and then select the Attributes inspector and enter the "Cellar" title in the Title field. To connect the `TableView` to the `TabBarController`, you first need to create a `NavigationViewController` for the `TableViewController`. The easiest way to create the NavigationViewController, other than dragging it from the palette onto the canvas, is to select it and select Editor/Embed/Navigation Controller from the Xcode menu.

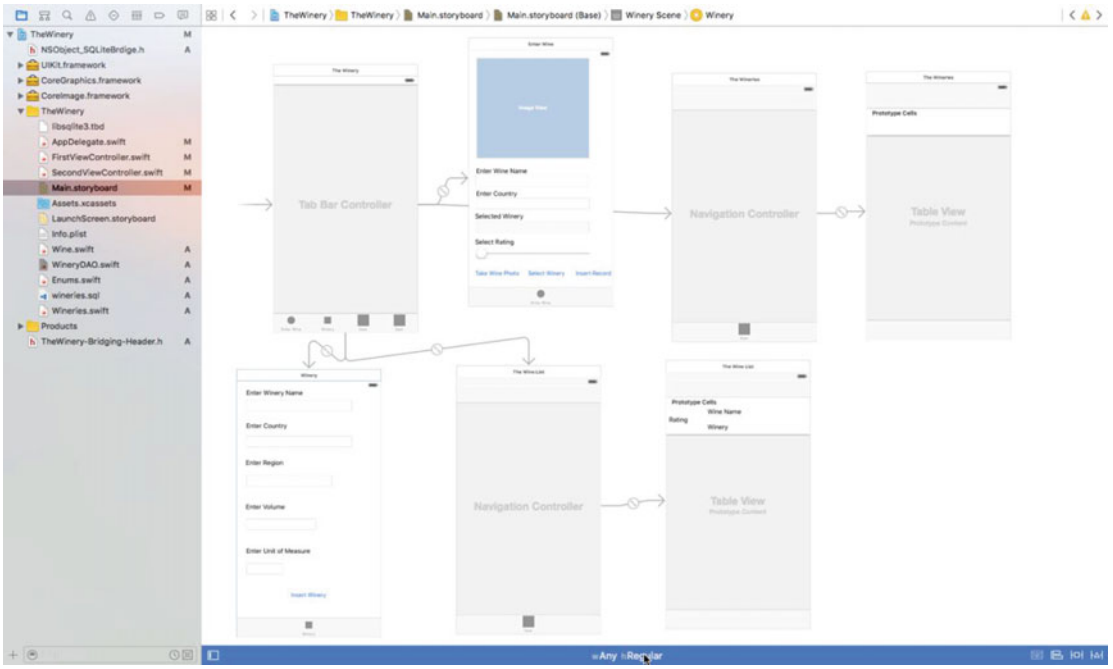

Figure 6-1. *Extended storyboard with TableViewControllers*

Once the navigation controller has been created, select the `TabBarController` and drag (press Control or CTRL and press the left mouse button while dragging a line from the TabBar to the navigation controller) a connection to the navigation controller. Upon releasing the mouse button, a popup will appear. From this popup, you need to select `ViewControllers` from the choices. This will create a connection with the `TabBarController` and will add a navigation item to the existing tab bar. You can change the item's value by selecting the navigation controller and changing the title value in the Attributes inspector. We need to repeat this process for the next `UITableViewController` for the winery list, which will also have a modified `TableViewCell`.

For now, that is all that is needed for this `UITableViewController`. We will add the `TableViewControllers` and the `TableViewCellController` in the next section.

After creating the second `UITableViewController`, using the preceding instructions for the Cellar, drag two `UILabels` onto the Cell Prototype (Figure 6-2). Change the label names to Wine and Winery as in the following figure. Also add an `ImageView` for the wine image. We will add the view controllers in the next section as well as create the `IBOutlets`.

Figure 6-2. *The Cellar TableView cell prototype*

Adding the Navigation Controllers

The final step is to provide an interface between the new UI elements and the `SELECT` functions, and ultimately the SQLite database. This interface consists of `TableViewControllers` and a `TableViewCellController` that is configured or connected to the corresponding UI elements, which will contain the `IBOutlets` that will display the data from the database.

Both `TableViewControllers` are created the same way. From the File menu in Xcode, select New, then File. In the template selector, choose the Cocoa Touch Class template under the iOS Source category (Figure 6-3). In the next screen, enter `WineryListTableViewController` in the Class field. In the Subclass dropdown, select the `UItableViewController` class, and leave or select the Swift language (Figure 6-4). When the class is created, Xcode will append the name of the super class to the name. As shown in Figure 6-5, click the Next button and select the location in the project, then click Create.

■ **Tip** You can also add a new file by right-clicking anywhere in the Project Navigator and selecting New File…

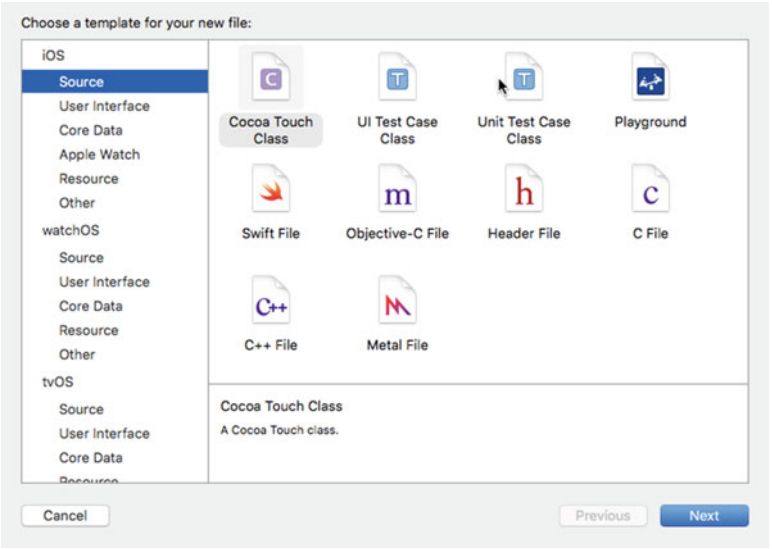

Figure 6-3. *Select the Cocoa Touch Class*

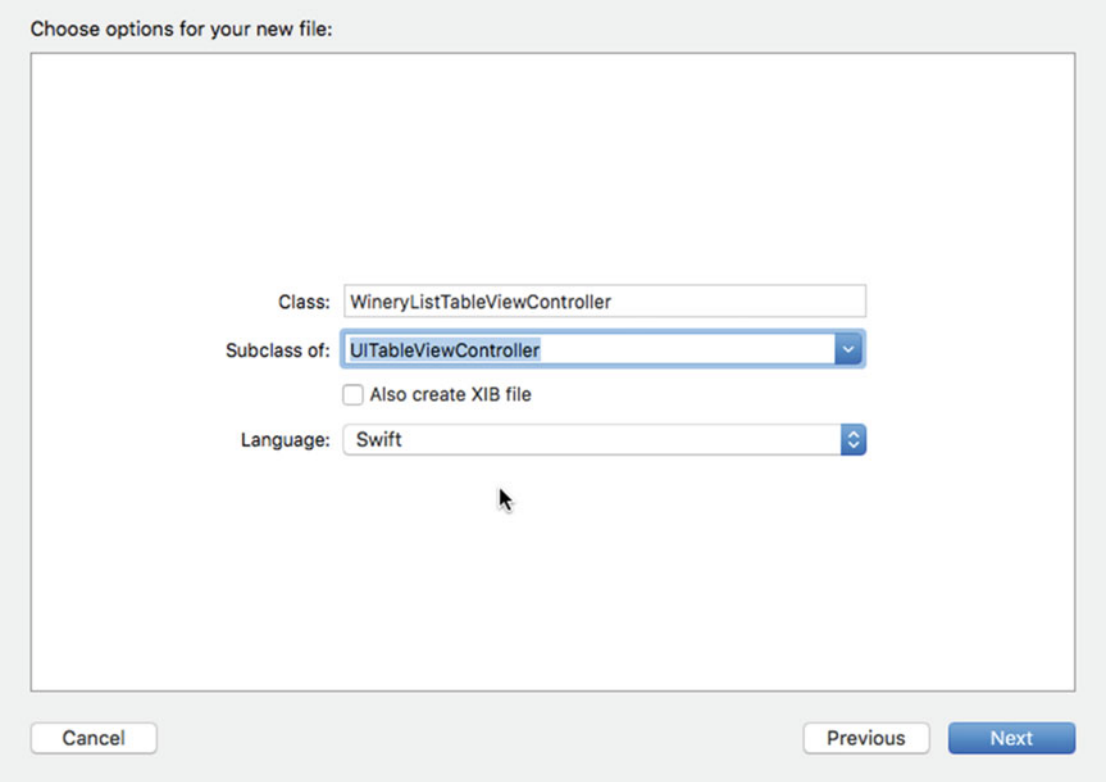

Figure 6-4. *Enter WineryList in Class field*

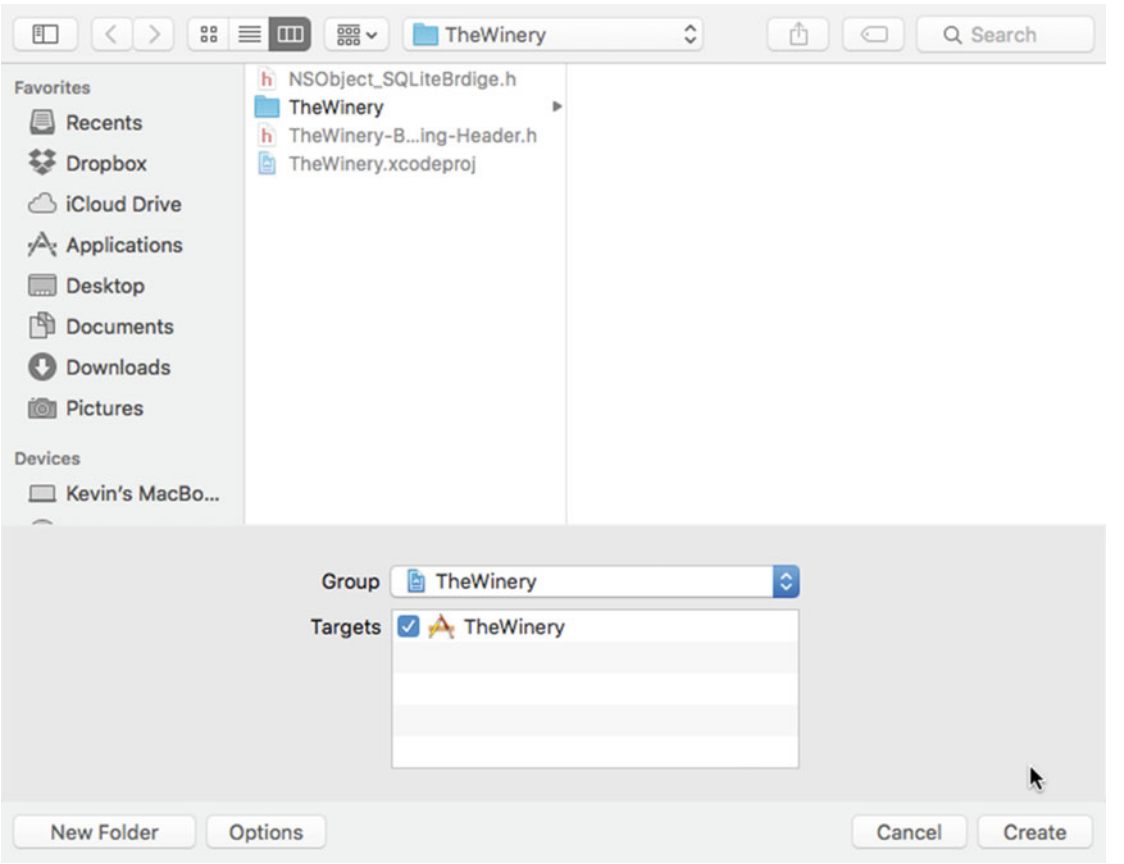

Figure 6-5. *Add the file to the project*

The template provides a lot of code that will need to be modified later. Repeat the same process for the `WineryList` controller, `WineCellTableViewCell`, and `WineryCellTableViewCell` `UITableViewCellController`. Next, we will connect the view controllers and add the `IBOutlets`.

Connect the TableViewControllers and TableViewCellController

To add the `IBOutlets`, you first need to connect the view controllers with their corresponding UI elements. To connect the table view controller, select the `WineList` in the storyboard. Select the Identity inspector on the right side of the Xcode editor, and in the Custom Class field select `WineListTableViewController` (Figure 6-6). Repeat the process with the `WineryListTableViewController`.

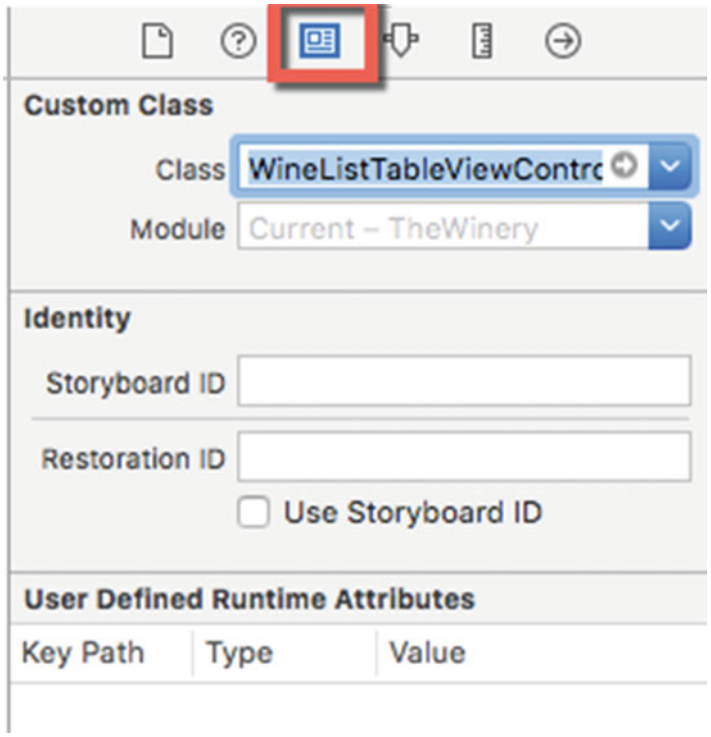

Figure 6-6. *Connect TableViewControllers*

To select the `TableViewCellController`, expand the Document Outline using the icon on the lower left of the IB canvas (Figure 6-7). Select the `TableViewCell`, then select the Identity inspector and choose the `WineCellTableViewController` from the dropdown (Figure 6-8).

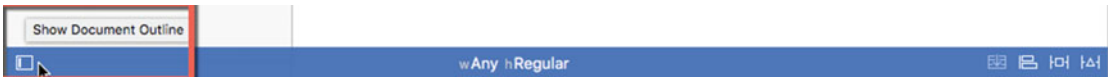

Figure 6-7. *Document Outline selector*

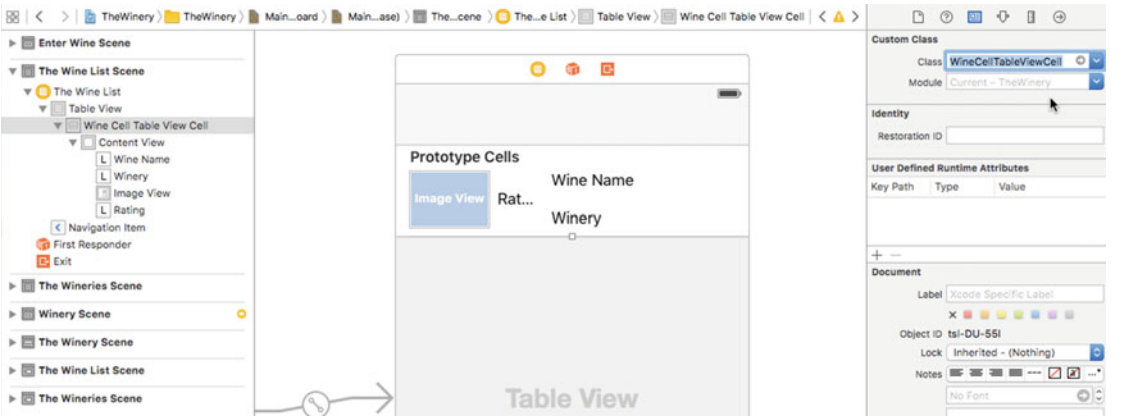

Figure 6-8. *Connect TableViewCellController*

Adding the IBOutlets: WineList Controller

For the `WineList` cell, expand the Document Outline and select the `WineCellTableViewCell` and control-drag a connection to the Assistant Editor (Figure 6-9).

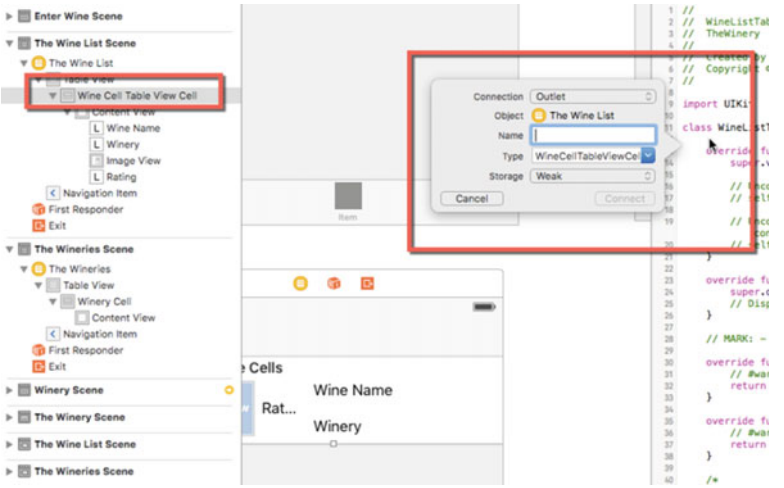

Figure 6-9. *Create WineCell outlet*

Next, in turn, select the `ImageView` and the `UILabels` and create `IBOutlets` for each of them, naming them `wineImgOutlet`, `wineNameOutlet`, and `wineryNameOutlet`.

With the `IBOutlets` set up, we can add the code to display the data in the scenes.

To display the winery information in the corresponding `UITableViewCell`, we will need to add `UILabels` for the winery name, country, region, volume, and unit of measure. Follow the same process as for the `WineCellTableViewCell` and connect them to the `WineryCellTableViewCell`.

```
import UIKit

class WineCellTableViewCell: UITableViewCell {
```

```
    @IBOutlet weak var wineNameOutlet: UILabel!
    @IBOutlet weak var wineryNameOutlet: UILabel!
    @IBOutlet weak var wineRatingOutlet: UILabel!
    @IBOutlet weak var wineImageOutlet: UIImageView!

}
```

Likewise, for the `WineryCell` in the `WineryListTableViewController` (Figure 6-10), add this code:

```
class WineryCellTableViewCell: UITableViewCell {

    @IBOutlet weak var wineryNameOutlet: UILabel!
    @IBOutlet weak var regionOutlet: UILabel!
    @IBOutlet weak var countryOutlet: UILabel!
    @IBOutlet weak var volumeOutlet: UILabel!
    @IBOutlet weak var uomOutlet: UILabel!
```

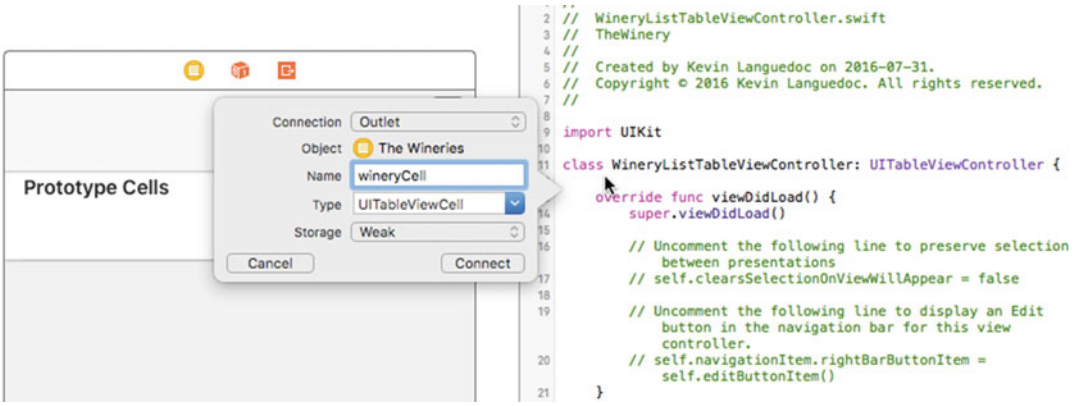

Figure 6-10. *WineryCell IBOutlet*

Add the Business Logic

To fetch the data from the database and display it in the UI, you need to add some code.

The WineListTableViewController

The `WineryListTableViewController` provides the interface between the database and the table view cell controller. For the data source, I define a wine array, `wineListArray`. I populate the array using the `selectWineList` function in the `wineDAO` instance object from the `loadWineList` function. This latter function gets called when the view loads using the `viewDidLoad` standard function of the `UITableViewController`:

```
var wineListArray = [Wine]()
    let wineDAO:WineryDAO = WineryDAO()

    func loadWineList(){
        wineListArray = wineDAO.selectWineList()
    }
```

```
override func viewDidLoad() {
        super.viewDidLoad()
        loadWineList()
      }
```

Next, you need to tell the controller how many sections will be in the table, as well as how many rows the data source will have or will be displayed. Typically, the `numberOfSectionsInTableView` returns 1 to indicate one section. For the number of rows, you typically return the number of elements in the array:

```
  override func numberOfSections(in: tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) ->
    Int {
        return wineListArray.count
    }
```

To display the data in the table, you need reference the cell that we previously set up in the IB and in the `WineCellTableViewCell` controller. You do that by passing the name of the cell identifier to the `dequeueReusableCellIdentifier` property of the table view. Then, cast the returning value as a `WineCellTableViewCell`. Next, you simply fetch each row in the array and assign the values to the cell's `IBOutlets`. This function is called automatically for each row in the array that you defined in the previous function:

```
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "WineCellTableViewCell",
        for: indexPath) as! WineCellTableViewCell

        // Configure the cell...
        let wine = wineListArray[(indexPath as NSIndexPath).row]

        cell.wineRatingOutlet.text = String(wine.rating)
        cell.wineNameOutlet.text = wine.name
        cell.wineryNameOutlet.text = wine.producer
        cell.wineImageOutlet.image =  UIImage.init(data: wine.image as Data)

        return cell
    }
```

The WineryListTableViewController

To set up the data source to the table, you need to define an array. In the sample app, I

```
class WineryListTableViewController: UITableViewController {
    var wineryListArray = [Wineries]()
    let wineDAO:WineryDAO = WineryDAO()
```

```
    func loadWineList(){
        wineryListArray = wineDAO.selectWineriesList()
    }
    override func viewDidLoad() {
        super.viewDidLoad()
        loadWineList()

}
override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
{
        return wineryListArray.count
    }
```

To display the individual rows in the cell's `UILabels`, set up the `cellForRowAt` function. To reference the cell prototype, you need to pass the `WineryCellTableViewCell` cell identifier to the `dequeueReusableCellWithIdentifier` property and cast the table view cell as `WineryCellTableViewCell`. With the cell reference in hand, you need to get each row in the data-source array and assign the values to the components in the cell, which in our case are the `IBOutlets` you set up before.

```
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "WineryCellTableViewCell",
        for: indexPath) as! WineryCellTableViewCell
        let winery:Wineries = wineryListArray[(indexPath as NSIndexPath).row]

        cell.wineryNameOutlet.text = winery.name
        cell.regionOutlet.text = winery.region
        cell.countryOutlet.text = winery.country
        cell.volumeOutlet.text = String(winery.volume)
        cell.uomOutlet.text = winery.uom

        return cell
    }
```

Running the App

With everything now created, fire up the app and actual iPhone, or you can use the camera in the simulator (Figure 6-11).

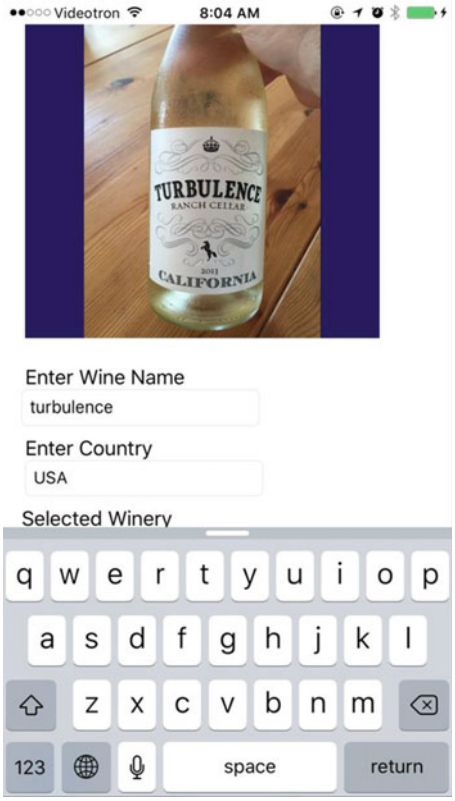

Figure 6-11. *Photo capture*

Next, select the winery and rating and click the Save button to insert it into the record in the database (Figure 6-12).

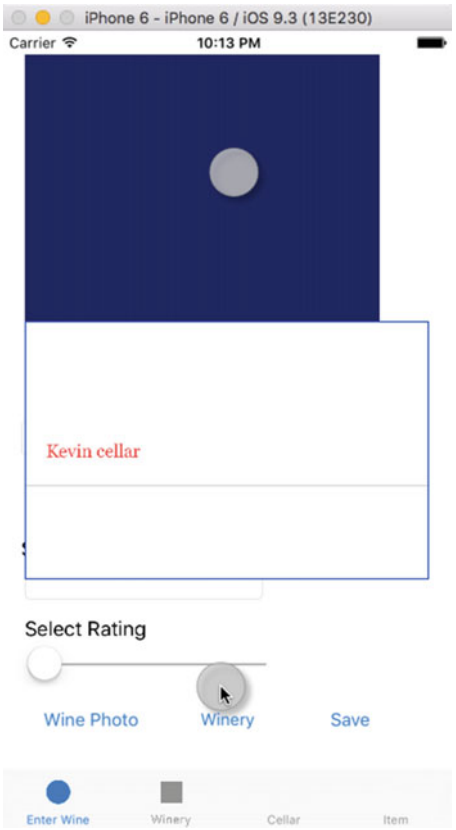

Figure 6-12. *Select Wineries UIPickerView*

To call the `SELECT` functions and display the data in the `UITableViews`, select either the Cellar or Vineyard button in the tab bar (Figure 6-13).

Figure 6-13. *Displaying the list of records*

Summary

The focus of this chapter was using the SQLite `SELECT` statement to perform queries on a SQLite database. The first part was an overview of the API along with some sample code to demonstrate using `SELECT` with Swift. We also looked at performing `SELECT` queries on binary data.

Finally, we added the `SELECT` query's functionality to the Winery app so as to select wines and wineries for the `UITableViewControllers` as well as for the `UIPickerView`.

The next chapter will explore the `UPDATE` statement in SQLite, and we will also add the `UPDATE` functionality to the Winery app.

CHAPTER 7

Updating Records

Updating records in SQLite is very similar to using the UPDATE statement on other platforms like SQL Server or Oracle. However, SQLite offers additional operations for updating records that are similar in nature to the SQLite INSERT statement.

In this chapter, we will discuss the various ways we can use the UPDATE statement in SQLite. These include the following:

- The UPDATE statement
- The WHERE clause
- The LIMIT and ORDER BY clauses
- Using sub-queries
- Using joins
- Using a WHERE clause in a sub-query
- Using the ON Conflict clause
- The OR FALLBACK statement
- The OR ABORT statement
- The OR REPLACE statement
- The OR FAIL statement
- The OR IGNORE statement

The following sections in this chapter will provide working examples of the queries for each of these operations in addition to adding the updating functionality to the Winery app.

SQLite Update Statement

The UPDATE statement is used to update an existing record or records in a SQLite database. If you only need to update a subset of the rows in a table, you can use a WHERE clause to filter or target the records that need to be updated or replaced, as we will see later. The LIMIT clause and the ORDER BY clause can be used to limit the number of rows to be affected by the UPDATE. Likewise, ORDER BY is primarily used to determine which rows are included when using the LIMIT clause.

WHERE, LIMIT, and ORDER BY are optional; however, the WHERE clause is almost always used. The table name should include the schema, especially if you have more than one database attached to the same file. See the following:

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_7

```
UPDATE main.tableName
SET column(s) = expression
WHERE whereClause
LIMIT numer_of_rows
ORDER BY column(s)
```

Other options that can be used with UPDATE are the OR clauses, which include ROLLBACK, FAIL, ABORT, IGNORE, and REPLACE.

The first example query we will look at is the basic UPDATE statement. If you have used UPDATE in SQL or in other platform derivatives of the SQL language like Pl/SQL or T-SQL, you will be right at home, because UPDATE works exactly the same way in SQLite:

```
UPDATE main.CityTemperature
SET Temperature = 1
, Scale = 'C'
, Date = '11-15-2014'
```

In the preceding example, the CityTemperature table is updated to reflect the current temperature reading. The assumption here is that there is only one entry in the database; otherwise, every entry would be updated as well.

In an iOS application, the UPDATE statement could use values captured from the UI fields (IBOutlets) and pass them to the query statement using SQLite binding types, as we have seen before. Using the preceding example, let us suppose there are three fields (UITextField) that are connected to a ViewController using IBOutlets. The code would look like something like this:

```
func updateRecords(){

        let sql:String = "Update main.data set coltext=?"

       if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){

            if(sqlite3_prepare_v2(db, sql.cString(using: String.Encoding.utf8)!, -1,
            &sqlStatement,  nil) == SQLITE_OK)
            {
                sqlite3_bind_text(stmt, 1, txt.cString(using: String.Encoding.utf8)!, -1,
                SQLITE_TRANSIENT);
}

        }

        sqlite3_step(stmt);
        sqlite3_finalize(stmt);
        sqlite3_close(cruddb);

    }
```

In this example, I define a Swift String constant with a placeholder for the new value for the column to be updated. After opening the database, the query string is attached to the sqlite3_prepared_v2 function, and the value is bound using sqlite3_bind_text.

Although functional, this query is not very practical, as it either assumes that there is only one record in the database or that every row will be updated with this new value. The next example will use a WHERE clause to limit the update to a common set of rows or a single row, as the case may be.

UPDATE Using a Where Clause

To make an UPDATE statement more precise, it is often used in conjunction with a WHERE clause. This clause is included at the end of the UPDATE statement, as seen in the following example. Of course, you could include multiple Boolean expressions in the WHERE by using the AND operator or even a standard or custom function.

```
UPDATE CityTemperature
SET Temperature = 1
, Scale = 'C'
, Date = '11-15-2014'
WHERE City = 'Montreal'
```

In Swift, this query could be represented as follows:

```
func updateRecords(){

        let sql:String = "Update Temperature set TemperatureScale =? where City=? and
        Country = ?"

       if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){

            if(sqlite3_prepare_v2(db, sql.cString (using: String.Encoding.utf8)!, -1,
            &sqlStatement,  nil) == SQLITE_OK)
            {
                sqlite3_bind_text(stmt, 1, txt.cString (String.Encoding.utf8)!, -1, SQLITE_
                TRANSIENT);
                sqlite3_bind_text(stmt, 2, utxt.cString (String.Encoding.utf8)!, -1, SQLITE_
                TRANSIENT);
                sqlite3_bind_text(stmt, 3, txt.cString (String.Encoding.utf8)!, -1, SQLITE_
                TRANSIENT);
}
        sqlite3_step(stmt);
        sqlite3_finalize(stmt);
        sqlite3_close(cruddb);

    }
```

In the preceding example, I bind values for each of the value placeholders, which are represented by question marks. The query can have as many binding values as are needed as long as you place the binding values in the same order as the placeholders in the query string.

UPDATE Using a Sub-query

Another way to assign a value to a column is to use a sub-query to fetch the rewired value and assign it to the required column. The sub-query would have to return only one value; otherwise, you would get an error. You could also use a function that evaluates an expression and returns a result. This expression could also include another query, for example, or a calculation of some sort. Let's see an example sub-query:

```
UPDATE CityTemperature
SET Scale = (Select Scale from TemperatureScales where Country = 'Canada')
where Country = ‘Canada’
```

Updating Records Using a Join

You can also use a join to fetch data from another table. In the following example, I use an `INNER JOIN`, but you can also use an `OUTER JOIN` or a `CROSS JOIN`. However, in most cases you would use an `INNER JOIN`.

```
UPDATE CityTemperature
SET TemperatureScale = 'C'
FROM CityTemperature ct INNER JOIN TemperatureScales s
ON ct.id = s.id
WHERE ct.Country = 'Canada'
```

UPDATE Using a Sub-Query in FROM Clause

Another option for an `UPDATE` operation is to use a sub-query in the `WHERE` clause, as the following example demonstrates. Here, the sub-query is named `tmpSelect` and it is the temperature column from this inner select, or sub-query, that is being referenced.

You could use this query in Swift as long as you maintain the order of the placeholders when binding the columns.

```
UPDATE CityTemperature
SET Temperature = tmpSelect.Temperature
FROM (SELECT id, Temperature from TempReadings where id = CityTemperature.Id and City =
'Montreal') as tmpSelect
```

Update On Conflict

SQLite provides five special operations to handle conflicts during a SQL transaction. These are `ROLLBACK`, `IGNORE`, `FAIL`, `ABORT`, and `REPLACE`. These aren't part of the SQL standard, but are provided through the SQLite API. These special statements enable a SQL script to handle conflicts at the column level when updating data. They handle constraints related to primary keys and unique, or not null, columns and values.

For the `UPDATE` command, like for the `INSERT` command, the `ON CONFLICT` is changed to `OR` followed by one of the keywords mentioned. We will look at these commands in the following sections.

Update or Rollback Records

When using the `ROLLBACK`, if the record you are trying to update is violating a constraint such as a primary key, the `ROLLBACK` option can help you gracefully handle the error and move on by rolling back the transaction. If the query is updating more than one record, the `ROLLBACK` will behave like the `ABORT` operation.

```
UPDATE OR ROLLBACK BooksToRead
SET Title = 'To Kill a Mockingbird'
, Author = 'Harper Lee'
WHERE id = 2
```

Update or Abort Records

The `ABORT` option stops the current transactional operation if there is a constraint issue, such as a null record or the data type of the updating value not being correct. The SQLite operation will back out of the current operation but leave any previous transactions intact.

```
UPDATE OR ABORT BooksToRead
SET Title = 'To Kill a Mockingbird'
, Author = 'Harper Lee'
WHERE id = 2
```

Update or Replace Records

When the query encounters a row with a constraint issue, if the `REPLACE` option is being used, the query will delete the constraining row and replace it with a new row, thus removing the constraint. If any `DELETE` triggers are in play, these will also fire.

```
UPDATE OR REPLACE BooksToRead
SET Title = 'To Kill a Mockingbird'
, Author = 'Harper Lee'
WHERE id = 2
```

Update or Fail Records

If the `FAIL` option is used, when a transaction encounters a constraint issue and throws a `SQLITE_CONSTRAINT`, the previous transactions are preserved, if any, but the current transaction that failed and any subsequent transactions won't be processed by the query. The operation stops at the faulty transaction.

```
UPDATE OR FAIL BooksToRead
SET Title = 'To Kill a Mockingbird'
, Author = 'Harper Lee'
WHERE id = 2
```

Update or Ignore Records

When the `IGNORE` option is used, the transaction with the problem is skipped and the remaining rows, if any, are processed.

```
UPDATE OR IGNORE BooksToRead
SET Title = 'To Kill a Mockingbird'
, Author = 'Harper Lee'
WHERE id = 2
```

A Sample SQLite UPDATE Operation in Swift

To demonstrate the `UPDATE` queries, we will add the update functionality to our minimalist iOS app that we have used in the previous chapters. In this installment, we will add the `updateRecords` method to the `CrudOp` custom class, which will update records in the database.

To begin, open the Crud iOS project in Xcode and locate the `CrudOp.h` header file. Add the `updateRecords` method as follows:

```
func updateRecords(){

        let sql:String = "Update data set coltext=? where coltext=?"
```

```
        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){

            if(sqlite3_prepare_v2(db, sql, -1, &sqlStatement,  nil) == SQLITE_OK)
            {
                sqlite3_bind_text(stmt, 1, txt, -1, SQLITE_TRANSIENT);
                sqlite3_bind_text(stmt, 2, utxt, -1, SQLITE_TRANSIENT);
            }
         }
         sqlite3_step(stmt);
         sqlite3_finalize(stmt);
         sqlite3_close(cruddb);

    }
```

As you can see, the code pattern is very similar to that of other SQLite operations. After creating a `sqlite3_stmt` and a local `sqlite3` database object, we create the update query as `const char`. The query could also be an `NSString` object, in which case you don't need to specify the UTF8String encoding to convert the string to the universal character set. Next, we establish a connection to the database engine and open the database using the `cruddatabase NSString` variable. Then, we encode it using the `UTF8string` property and the `sqlite3` object.

With the connection made, we can execute the query by using the `sqlite3_prepare_v2` function and binding the parameter variables for the `set` and `where` clauses. The binding function must match the data type of the column and parameter values or you will get an error, unless you cast the value to the proper type.

The `sqlite3_step` function will execute the statement that we prepared in the previous operation and will finally use `sqlite3_finalize` before closing the connection using `sqlite3_close`. This is the basic setup for the update query in SQLite. Using this pattern, you can do any type of update query.

As with the other CRUD operations, we will call the method through the `segButton IBAction` in the `kcbViewController` custom class of the `ViewController` in the storyboard.

Adding the UPDATE Functionality to the Winery App

The next piece of the winery app's functionality is the ability to update existing records. We will need to modify the `wineyDAO` class and add two functions: one for the wines and the other for the wineries. We will look at these next.

Modifying the WineryDAO Controller

I will add the `wineUpdate` function for the wines and `wineryUpdate` for the wineries. Both will have the same design and will update all columns in the table, regardless of whether that value is changing based on a `WHERE` clause.

Adding the wineUpdate Function

The `wineUpdate` function updates a wine record in the SQLite database. The function takes a `wine` object as an input parameter. After defining an `SQL UPDATE` string, I open the SQLite database using the `sqlite3_open` function as usual. Then, I set up the query using the `sqlite3_prepare_v2` function, which takes the SQLite database pointer, the `sqlite_statement` pointer, and the `sql` query string as input parameters. If the query string is OK, the function will return `SQLITE_OK` status.

Next, you need to bind the wine properties to the input columns and WHERE clause using sqlite3_bind_text for strings, sqlite3_bind_int for the integers, and sqlite3_bind_blob for the image.

Finally, execute the query using the sqlite3_step function, then clean up using sqlite3_finalize and close the database using sqlite3_close. See the following code.

■ **Note** You can use the ternary operator to build the query string and binding values based on the available values if you don't want to replace every value. Also, this code doesn't have any error handling.

```
func WineUpdate(_ wine:Wine){
        let sql:String = "UPDATE main.Wine " +
        "SET name = ?," +
        "rating = ?, " +
        "image = ?, " +
        "producer = ? " +
        "WHERE id = ?"

        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, sql, -1, &sqlStatement, nil)==SQLITE_OK){
                sqlite3_bind_text(sqlStatement, 0, wine.name.cString(using: String.Encoding.
                utf8), -1, SQLITE_TRANSIENT)
                sqlite3_bind_int(sqlStatement, 1, wine.rating)
                sqlite3_bind_blob(sqlStatement, 2, wine.image.bytes, Int32(wine.image.
                count), SQLITE_TRANSIENT)
                sqlite3_bind_text(sqlStatement, 4, wine.producer.cString(uisng: String.
                Encoding.utf8), -1, SQLITE_TRANSIENT)
                sqlite3_bind_int(sqlStatement, 5, wine.id)
                sqlite3_step(sqlStatement)
            }
        }
        sqlite3_close(db)
    }
```

Adding the wineryUpdate Function

The wineryUpdate function follows exactly the same pattern as seen in the previous function. Except for the different columns and table targeted in the SQL query string, as well as the number of binding values, the functionality is identical to the previous one:

```
func WineryUpdate(_ winery:Wineries)->Int32{
let sql:String = "UPDATE winery SET country = '\(winery.country)', region = ' \(winery.
region) ', volume = \(winery.volume), uom = ' \(winery.uom) ' WHERE name = ' \(winery.name)
' ;"
            var status_code:Int32 = 0

            if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
                status_code = sqlite3_prepare_v2(db, sql, -1, &sqlStatement, nil)
                if(status_code==0){
                    status_code = sqlite3_step(sqlStatement)
```

```
                status_code = sqlite3_finalize(sqlStatement)
            }
        }
        sqlite3_close(db)

        return status_code

    }
```

Modifying the UI for Updates

To enable the update functionality that was added to the `wineryDAO` class, some changes need to be made to the UI and corresponding `FirstViewController` and `SecondViewController`.

When a user selects an item from either the Wine list or Winery list, the data will be displayed in the corresponding scene so that the information can be changed. When the Save button is clicked, instead of calling the insert functions, the update functions will be called instead.

Set Up the showWineDetail Segue

Figure 7-1 depicts the layout of the new segue that connects the Cellar scene, which is the `TableViewController` for the list of wines that are stored in the database, with the Enter Wine scene, which is the initial view controller when the app starts.

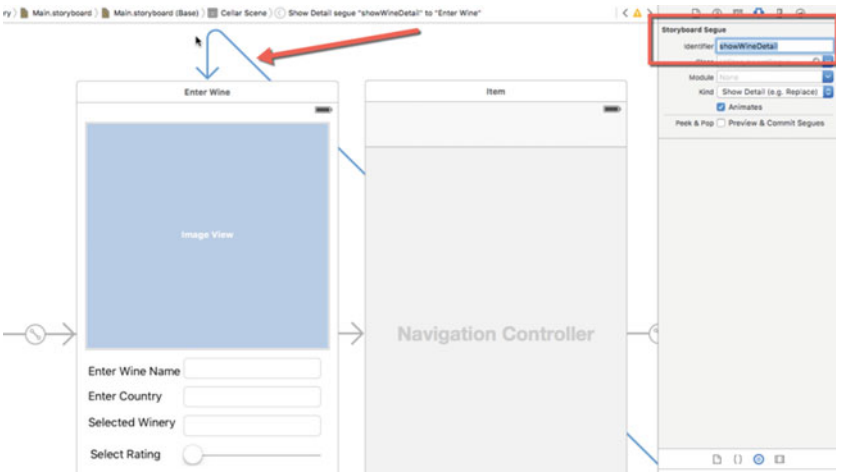

Figure 7-1. *Adding the showWineDetail segue*

To set up the segue, follow these steps:

1. Control + drag a connection from the `WineCellTableViewCell` in the Cellar scene to the Enter Wine scene using the mouse. When you release the mouse button a popup will appear. Select the `showDetail` option.
2. Select the newly created segue and select Attributes inspector. There, you can enter the `showWineDetail`. This is used in the `prepareForSegue` function, which is triggered just before the segue is executed.

WineListTableViewController

Next, switch to the `WineListTableViewController`. We will add the functionality to transfer the data to the Enter Wine scene. To send data to `FirstViewController's` field outlets, we need to uncomment the `prepareForSegue` function, which is `n=one` of the optional function in the standard `TableViewController` class.

Given the fact that we need to send information to the `FirstViewController` when a row is selected in the table view, we first need to define a constant of that type. In the code that follows, I create the `wineViewController` constant and assign the `segue.destinationViewController` property, then cast this as a `FirstViewController` object. Next, I attempt to create a constant for the cell in the table by assigning the sender, as an optional `WineCellTableCell`, to the `wineCell` constant. If the operation is successful, I merely need to assign the selected values to the fields in the `FirstViewController` object.

The value of the `UISlider` can't be directly set, so I created a new function in the `FirstViewController` (see the code snippet that follows) that sets its value. Here, I simply call the `setWineRating` function, passing the value of the rating outlet and casting it as a float.

In addition, I added a new property to the `FirstViewController` called `isEdit`, which I will assign a value of 1. We can now click the Save button to call the `WineUpdate` function rather than the `insertWineRecord` function. The code is listed after the end of this section.

```
override func prepare (_for segue: UIStoryboardSegue, sender: AnyObject?) {
      // Get the new view controller using segue.destinationViewController.
      // Pass the selected object to the new view controller.
      if(segue.identifier == "showWineDetail"){
          let wineViewController = segue.destinationViewController as! FirstViewController
          if let wineCell = sender as? WineCellTableViewCell{

              let indexPath = tableView.indexPath(for: wineCell)!
              let selectedWine = wineListArray[(indexPath as NSIndexPath).row]
              wineViewController.wine = selectedWine
              wineViewController.isEdit = 1
          }
      }
      }
```

However, before you can set the `wineRatingSelected`, which is the `IBOutlet` for the `UISlider` in the storyboard, you need to create the `IBOutlet`.

Follow these steps:

1. Open the storyboard and select the Enter Wine view controller; click on the Identity inspector to open the `FirstViewController` file alongside the IB canvas.
2. Select the `UISlider` in the IB canvas and control + drag a connection over to an empty space in the `FirstViewController` file.
3. As shown in Figure 7-2, release the mouse button and enter the name of the outlet. In this app, I name it `wineRatingSelector`.

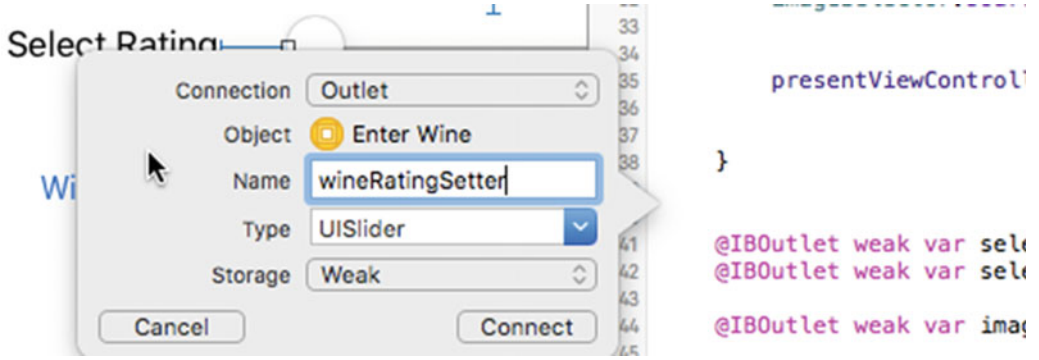

Figure 7-2. *Set the wineRatingSetter IBOutlet connection*

4. Accept the other default values, including the `IBOutlet` connection type
5. Click on the Connect button to create the outlet connection on the `FirstViewController` file, as follows:

```
    @IBOutlet weak var wineRatingSetter: UISlider!
```

6. Now you can access it through the `WineListTableViewController`.

```
var isEdit:Int = -1
//skip code for brevity
    @IBAction func insertRecordAction(sender: AnyObject) {
        if(isEdit==1){
            //we should update
            let editWine = Wine()
            editWine.name = self.wineNameField.text!
            editWine.producer = self.selectWineryField.text!
            editWine.rating = Int32(self.wineRatingSetter.value)
            dbDAO.wineUpdate(editWine)
        }else{
            wine.name = self.wineNameField.text
            wine.producer = self.selectWineryField.text!
            dbDAO.insertWineRecord(wine)
        }
}
```

WineryListTableViewController

You need to follow the same approach for the `WineryListTableViewController`. Figure 7-3 demonstrates the layout of the storyboard with the new segue for this.

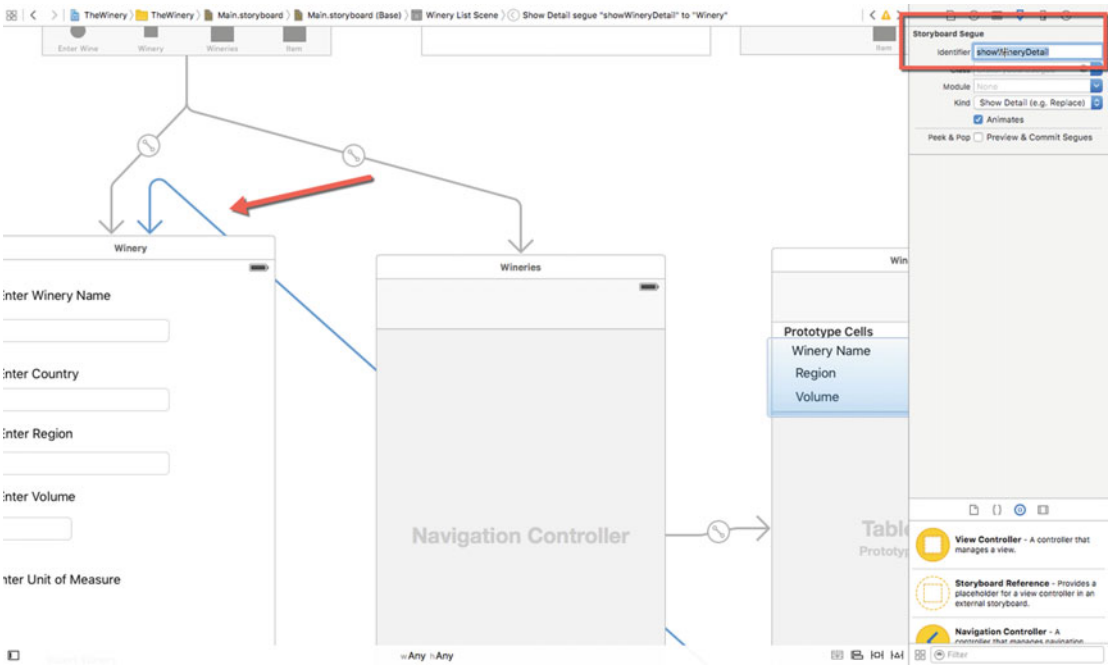

Figure 7-3. *The showWineryDetail segue*

Follow these steps to create the segue for the second view controller:

1. Select the `WineryCellTableViewCell` from the Document Outline view.
2. Then control + drag a connection from the selected cell in UI over to the Winery scene.
3. Release the mouse button and select "Show Detail" in the popup.
4. Next, select the new segue and open the Attributes inspector; name the segue `showWineryDetail` in the Identity field.

With the segue created, we open the `SecondViewController` and uncomment the `prepareForSegue` function near the end of the code.

As before, we need to create a constant with a `segue.destinationViewController` assignment, then cast this as a `SecondViewController`. This will give us the `IBOutlets` in the `SecondViewController`. Next, we need to define a constant for the cell in the Winery List `TableViewController` so that we can access the `IBOutlets` defined there. Then, it's a simple operation of assigning the values of the selected row in the table to the outlets in the `wineryController` object before the segue pushes the `SecondViewController` onto the stack with the corresponding fields populated.

Notice the `isEdit` property, described later, that we are adding to the `SecondViewController` to indicate that we need to run the `WineryUpdate` function rather than the `insertWineryRecord` function.

```
override func prepare(_ for segue: UIStoryboardSegue, sender: AnyObject?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
        if(segue.identifier == "showWineryDetail"){
            let wineryController = segue.destinationViewController as! SecondViewController
            if let wineryCell = sender as? WineryCellTableViewCell {
                let indexPath = tableView.indexPath(for: wineryCell)!
                let selectedWinery = wineryListArray[(indexPath as NSIndexPath).row]
                wineryController.winery = selectedWinery
                wineryController.isEdit = 1

            }
        }
    }
```

As with the previous discussion on saving changes, we have modified the `SecondViewController` by adding the `isEdit` property and setting its initial value to `-1`. Mind you, we could set this value to anything, really. Also, we could have used the `unwind Segue` functionality, but this little bit of code suits our purposes better. In addition, we kept the previous connection as `insertWineryBtn` instead of renaming it something like `saveWinery`, simply to illustrate a possible functionality of saving records to a SQLite database.

```
var isEdit:Int = -1

@IBAction func insertWineryBtn(_ sender: AnyObject) {
winery.name = wineryNameField.text!
    winery.country = countryNameField.text!
    winery.region = regionNameField.text!
    winery.volume = Double(enterVolume.text!)!
    winery.uom = enterUoM.text!
    if(winery.isEdit==1){
        dbDAO.wineryUpdate(winery)
    }else{
        dbDAO.insertWineryRecord(winery)
    }
  isEdit = -1
}
```

With the controllers in place, all that is needed is to run the app to see if everything checks out, and to debug if necessary.

Running the App

For this example, we will fetch a wine record as well as a winery record from the database, update the records, and save them back to the database.

Updating Records

Launch the app on an iPhone and switch to the Winery List (`wineries`). In this example, we have one winery in the list. If we click on it, it should display the contents in the proper fields in the Winery scene. Figure 7-4 displays one record from the winery table in the database.

Figure 7-4. *List of wineries*

When we select the item in the list, the contents are displayed back in the `FirstViewController` for editing (Figure 7-5). After making edits, we click on the Save button to send the data back to the database.

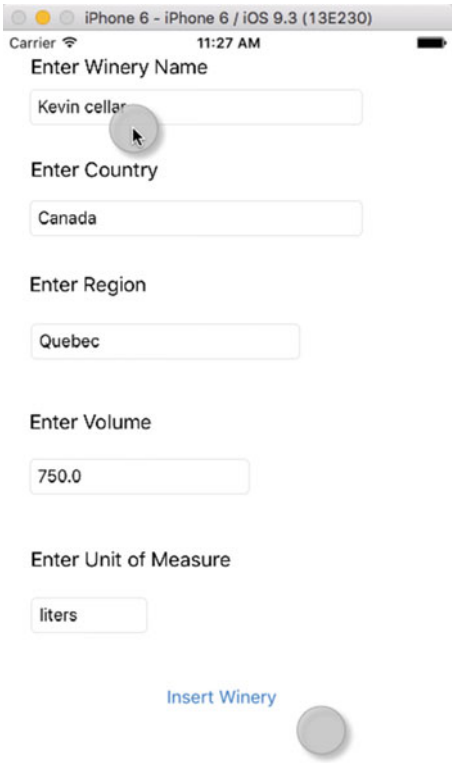

Figure 7-5. *Displaying winery data from segue*

Figure 7-6 displays the changes that are made to the Lamartine bottle of wine.

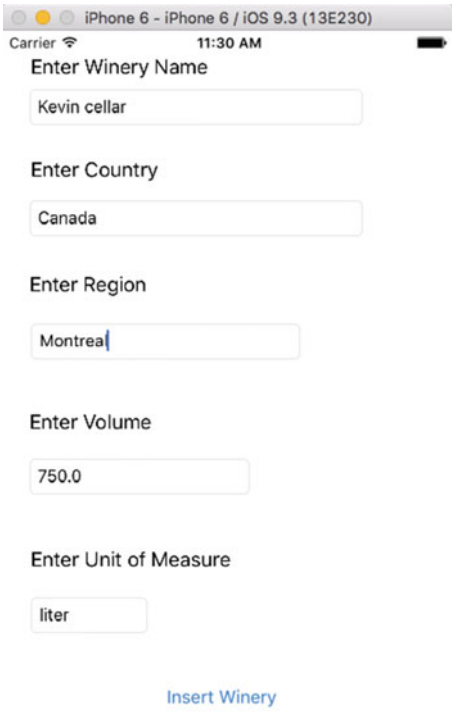

Figure 7-6. *Displaying changes to winery*

Figure 7-7 demonstrates the new data being passed to the `UPDATE` query, and once the `sqlite3_step` is complete the status code contains the value of `0` or `SQLITE_OK`. Figure 7-8 shows the status code of `101`, meaning `SQLITE_DONE`, which means that the query has successfully completed execution.

Figure 7-7. *Updating record with Status Code = 0*

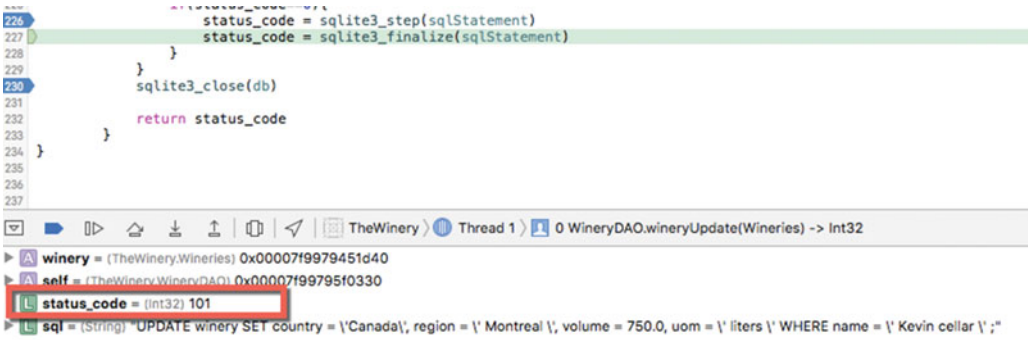

Figure 7-8. *sqlite3_finalize's successful completion status code*

Figure 7-9 lists a few wine entries. There are a few misses, which we will clean up in the next chapter, and the image needs to be rotated 90 degrees, but we have a good fetch from the database.

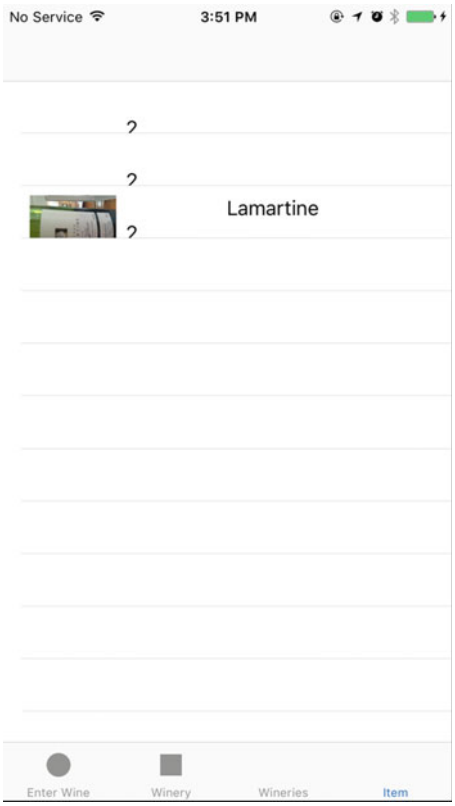

Figure 7-9. *List of wines*

Figure 7-10 displays the selected contents of the Lamartine wine record from the preceding table list.

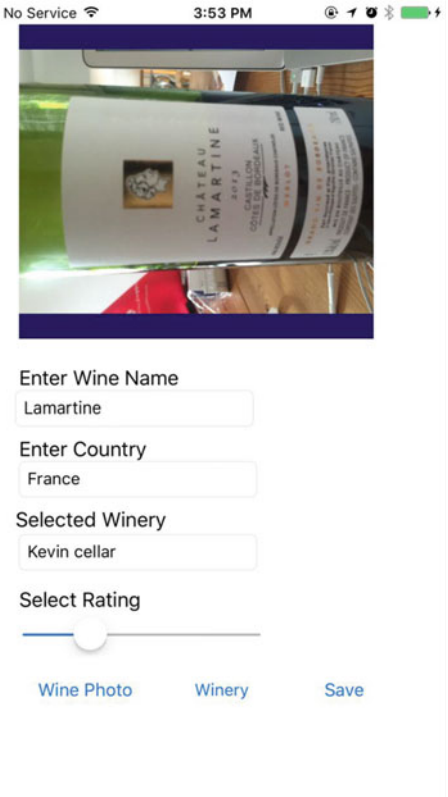

Figure 7-10. *Wine details*

Figure 7-11 shows a status code of `101`, or `SQLITE_DONE`, which signifies that the query was successfully completed.

Figure 7-11. *Successful UPDATE operation*

Summary

This completes the `UPDATE` discussion and how to do additions in the Winery app. We explored the SQLite `UPDATE` in detail, including the `OR` clauses. We looked at how to add `UITableViewControllers` and manage updates and inserts using the same `IBAction`.

The last operation is `DELETE`, which wraps up the CRUD group of operations on the Winery app. The `DELETE` chapter will provide detailed information on the SQLite `DELETE` statement and how we can incorporate the deletion functionality in the `UITableViews`.

CHAPTER 8

Deleting Records

In this chapter, we will discuss the DELETE function in SQLite. In contrast to the other CRUD functions in SQLite, the DELETE function API is the same as in other platforms and implements the basic SQL API, except for the LIMIT clause, which allows a developer to set a limit on the number of rows to delete. We will cover the following:

- The DELETE statement
- DELETE using a WHERE clause
- DELETE restrictions and TRIGGERS
- LIMITS
- A Swift DELETE example

The DELETE Statement in SQLite

The DELETE statement is a standard SQL statement that is used to permanently remove one or more records from a table in a SQLite database. The basic syntax is as follows:

```
DELETE FROM main.tablename
```

or,

```
DELETE FROM main.tablename
WHERE Boolean expression
```

If the WHERE clause is not used, then the entire contents of a given table is deleted.

Using the WHERE Clause

To better control the records that you are deleting, you can use the WHERE clause to specify a Boolean variable. As with other SQL WHERE clauses, you can use the AND keyword to specify more than one column or the OR keyword to specify one column or another. There is no limit on the number of columns that you can add to your WHERE arguments. In addition, you can use NOT, as in NOT IN or NOT EQUAL. You can also use the BETWEEN keyword; for example, to specify a date or number range. By using the WHERE clause along with DELETE, you can limit the number of records that are affected by the query.

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_8

The syntax is as follows:

```
DELETE FROM table
WHERE columnA = 'value'
AND columnB = 'value'
```

You can also write a query as follows:

```
DELETE FROM table
WHERE (columnA = 'value'
AND columnB = 'value') OR
(columnC = 'value'
AND columnD = 'value')
```

Restrictions and Triggers

When using the DELETE statement with a TRIGGER, you are not allowed to use the schema name, only the table name. In other words, you must use the DELETE as follows:

```
DELETE FROM table
```

Rather than:

```
DELETE FROM main.tablename
```

Also, if the DELETE statement in the trigger is not associated with a TEMP table, then the trigger must be in the same database as the trigger. The trigger will search for tables in each of the attached databases in the order in which the database was attached.

DELETE Limits

You can use the LIMITS clause along with the ORDER BY clause to restrict the number of rows to delete. By using the ORDER BY, you can sort the records in either an ascending (ASC) or descending (DESC) order so as to ensure the proper rows are targeted for deletion.

In order to use the LIMIT and ORDER BY clauses, you must enable the SQLITE_ENABLE_UPDATE_DELETE_LIMIT option when compiling the database. Also, it is important to keep in mind that the limits and triggers are not supported for the DELETE statement in SQLite.

A Swift SQLite Delete Example

Using the SQLite DELETE statement in Swift is very easy, as the following example illustrates. After setting the usual variables and constants, we set up the database in the viewDidLoad function and call the setupSampleTable, addRecords, and sampleDelete functions sequentially.

The first two functions create a table and add one record to the table. The main purpose of this code is to demonstrate the DELETE query in Swift, which I do in the sampleDelete function.

```
let dbName:String="chapter8.sqlite"
var db:COpaquePointer?=nil
var sqlStatement:COpaquePointer?=nil
```

```
    var errMsg: UnsafeMutablePointer<UnsafeMutablePointer<Int8>>! = nil
    internal let SQLITE_STATIC = unsafeBitCast(0, for:sqlite3_destructor_type.self)
    internal let SQLITE_TRANSIENT = unsafeBitCast(-1, for:sqlite3_destructor_type.self)
    var dbPath:URL = URL(fileWithPath:"")
    var errStr:String = ""
    override func viewDidLoad() {
        super.viewDidLoad()
        let dirManager = FileManager.default()
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateForL: nil, create: true)
        dbPath = try! directoryURL.appendingPathComponent(dbName)
        } catch let err as NSError {
            print("Error: \(err.domain)")
        }
        self.setupSampleTable()
        self.addRecords()
        self.sampleDelete()

    }
func setupSampleTable(){
        let creatSQL:String = "CREATE TABLE IF NOT EXISTS sample(id int, name varchar)"
        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, creatSQL, -1, &sqlStatement, nil)==SQLITE_OK){
                if(sqlite3_step(sqlStatement)==SQLITE_DONE){
                    print("table created")
                    sqlite3_finalize(sqlStatement)
                    sqlite3_close(db)
                }else{
                    print("unable to create table")
                }
            }
        }
    }

    func addRecords(){
        let insertSQL:String = "INSERT INTO TABLE main.sample (id, name) VALUES(?,?)"
        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, insertSQL, -1, &sqlStatement, nil)==SQLITE_OK){
                sqlite3_bind_int(sqlStatement, 1, 1)
                sqlite3_bind_text(sqlStatement, 2, "kevin", -1, SQLITE_TRANSIENT)
                if(sqlite3_step(sqlStatement)==SQLITE_DONE){
                    print("table created")
                    sqlite3_finalize(sqlStatement)
                    sqlite3_close(db)
                }else{
                    print("unable to create table")
                }
            }
        }
    }
```

As you can see from the code that follows, the DELETE query follows a pattern similar to that of the other CRUD operations. We first set a SQL query string for the DELETE statement, then we attempt to open the database and load the DELETE SQL query string into memory using the sqlite3_prepare_v2 function.

Then we bind the WHERE value using the sqlite3_bind_int function and execute the query using the sqlite3_step function. If we get a successful status result, we clean up the query with sqlite3_finalize and close the database.

```
func sampleDelete(){
        let deleteStmt:String = "DELETE FROM sample WHERE id = ?"

         if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, deleteStmt, -1, &sqlStatement, nil)==SQLITE_OK){
                sqlite3_bind_int(sqlStatement, 1, 1)
                if(sqlite3_step(sqlStatement)==SQLITE_DONE){
                    print("item deleted")
                    sqlite3_finalize(sqlStatement)
                    sqlite3_close(db)
                }else{
                    print("unable to delete")
                }
            }
         }
    }
```

Alternatively, we could perform the same query operation using the sqlite3_exec function, as in the following example. As you may remember, the sqlite3_exec function encapsulates the sqlite3_prepare_v2, sqlite3_step, and sqlite3_finalize functions. See here:

```
func sampleExecDelete(){
        let deleteStmt:String = "DELETE FROM sampleTable WHERE id = ?"
       if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_exec(db, deleteStmt, nil, &sqlStatement, errMsg)==SQLITE_OK){
                sqlite3_bind_int(sqlStatement, 1, 1)
                sqlite3_close(db)
            }else{
                print("unable to delete")
        }
    }
}
```

Adding the Delete Functionality to the Winery App

This is the final installment of the CRUD application that we have been building. To begin adding the DELETE functionality to our app, open the header and the deleteRecords method as shown in the code that follows. The deleteRecords method will take only one parameter for the WHERE clause variable.

To implement the DELETE functionality in the Winery app, we will need to make several modifications to the various view controllers and table view controllers as well as to the storyboard's UITableViewControllers. We will also need to add two functions to the WineryDAO class.

Modifying the WineryDAO Class

In the WineryDAO class, we will need to add two functions to handle the deletes in the Wine table as well as the deletes in the Winery table. Both functions will perform a DELETE statement on the winery database. Both functions will implement the same design using two different approaches. Actually, this code could be replaced by one function, and we could pass the query string as an input parameter, but I decided to use sqlite3_exec for the second function to demonstrate the two ways to execute a query in SQLite.

Add the deleteWineRecord Function

The deleteWineRecord function contains a string constant that defines a DELETE SQL statement and takes one parameter for the WHERE clause.

The SQLite execution operation includes the opening of the SQLite database using the sqlite3_open function, followed by the sqlite3_prepared_v2 function if the database is successfully opened.

If the SQL statement contains no errors and is successfully loaded into memory, the parameter is passed to the query using the sqlite3_bind_text function, and finally the query is executed with the sqlite3_step function.

I will test the function and post the results at the end of the chapter, as usual.

```
func deleteWineRecord(record:String){
    let deleteSQL = "DELETE FROM wine WHERE name = ?"

    if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
        if(sqlite3_prepare_v2(db, deleteSQL, -1, &sqlStatement, nil)==SQLITE_OK){
            sqlite3_bind_text(sqlStatement, 1, record, -1, SQLITE_TRANSIENT)
            if(sqlite3_step(sqlStatement)==SQLITE_DONE){
                print("item deleted")
            }else{
                print("unable to delete")
            }
        }

    }
}
```

Add the deleteWineryRecord Function

As with the previous function, the deleteWineryRecord will delete a record from the SQLite database that is selected in the WineryListTableViewController. The first line of the body of the function contains a SQL string constant that defines a SQL DELETE statement. Notice how the schema is absent from the table definition as per the API requirements.

Next, sqlite3_open is used to open the database, followed by the sqlite3_exec function, which encapsulates the execute API of sqlite3_prepare_v2, sqlite3_step, and sqlite3_finalize.

```
func deleteWineryRecord(record:String){
        let deleteSQL = "DELETE FROM winery WHERE name = ?"
       if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
                if(sqlite3_exec(db, deleteSQL, nil, &sqlStatement, errMsg)==SQLITE_OK){
                   sqlite3_bind_text(sqlStatement, 1, record, -1, SQLITE_TRANSIENT)
                   sqlite3_close(db)
```

```
                }else{
                  print("unable to delete")
                }
        }
    }
```

Modifying the ViewControllers

For this part of the app, we don't need to make any changes to either the `FirstViewController` or the `SecondViewController`.

Modifying the TableViewControllers

Adding the delete functionality requires a couple of changes to both the `WineryListTableController` and the `WineListTableViewController`. We have to enable the Edit button in the menu as well as call the delete functions in the `WineDAO` class.

WineryListTableViewController

For the `WineryListTableViewController`, we enable the Edit button in the `viewDidLoad` method just below the `loadWineList()` function. The `edit.editButtonItem` is assigned to the `self.navigationItem.rightBarButtonItem` method, which you will able to see when the app runs.

```
override func viewDidLoad() {
        super.viewDidLoad()
        loadWineList()

        // Uncomment the following line to display an Edit button in the navigation bar for
        this view controller.
        self.navigationItem.rightBarButtonItem = self.editButtonItem()
    }
```

The second change is to enable the following `tableView` standard function. The code is already included when we create a sub-class of the `UITableViewController`; we only need to uncomment it to enable it.

However, we still need to add some code to actually delete the row from the database, as `deleteRowsAtIndexPath` will remove an item from the array, but if we don't remove the same row from the database, when the view is refreshed, the record will re-appear. We accomplish this by getting the row index of the selected record and retrieving that record from the wineries array. Then, we create an instance object of the winery class and pass the name of the winery to the `deleteWineryRecord` function.

We also add `tableView.beginUpdates` and `tableView.endUpdates` at the beginning and end of the delete operation. These indicate that subsequent operations are performed and terminated.

```
override func tableView(tableView: UITableView, commitEditingStyle editingStyle:
UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
if editingStyle == .delete {
            tableView.beginUpdates()
            // Delete the row from the data source
            let winery:Wineries = wineryListArray[(indexPath as NSIndexPath).row]
```

```
            tableView.deleteRows(at: [indexPath], with: .fade)
             wineDAO.deleteWineryRecord(winery.name)
            tableView.endUpdates()        } else if editingStyle == .Insert {
            // Create a new instance of the appropriate class, insert it into the array, and
            add a new row to the table view
        }
    }
```

WineListTableViewController

In kind, we implement the Edit button for the `WineListTableViewController` as we did with the previous `TableViewController`:

```
override func viewDidLoad() {
        super.viewDidLoad()
        self.loadWineList()
        self.navigationItem.rightBarButtonItem = self.editButtonItem()
    }
```

Again, we implement the following `tableView` function to enable the delete capabilities of the `TableView`. In the `delete editingStyle`, we enclose `tableView.beginUpdates` and `tableView.endUpdates` and fetch the row index from the `wineListArray` before creating an instance object and passing the name value of the selected record to the `deleteWineRecord` function.

```
override func tableView(tableView: UITableView, commitEditingStyle editingStyle:
UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
        if editingStyle == .delete {
            tableView.beginUpdates()
            let wine = wineListArray[(indexPath as NSIndexPath).row]
            // Delete the row from the data source
            tableView.deleteRows(at: [indexPath], with: .fade)
            wineDAO.deleteWineRecord(wine.name)
            tableView.endUpdates()
          } else if editingStyle == .Insert {
            // Create a new instance of the appropriate class, insert it into the array, and
            add a new row to the table view
        }
    }
```

With this code in place, all that is needed now is to run the app and test it.

Modifying the UI for Delete

Modifying the UI

When we enabled the Edit button in the `viewDidLoad` methods of both the table view controllers, we activated the Edit buttons. When you click on the Edit menu item, the UI changes to display the Delete icon. Other than this, no change is needed to the UI. In order to see the Edit and Delete buttons, you will need to run the app, which I will do next.

Running the App

Figure 8-1 is the list of wines in the Cellar `TableViewController`. Instead of selecting an item, click on the "Edit" link. This will switch the view into Edit mode.

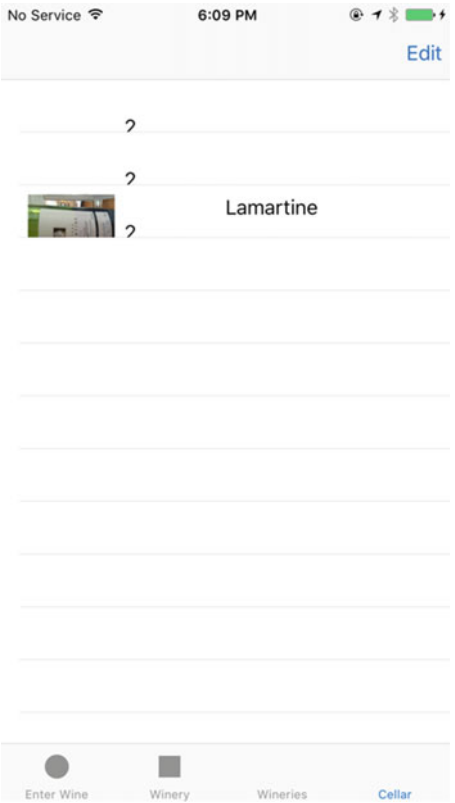

Figure 8-1. *The Cellar TableView*

Figure 8-2 shows the Cellar `TableView` in Edit mode. Notice how the Edit button was converted to *Done*. All this is provided with one line of code in the `viewDidLoad` function. Select one of the empty rows, and the row will display a Delete button.

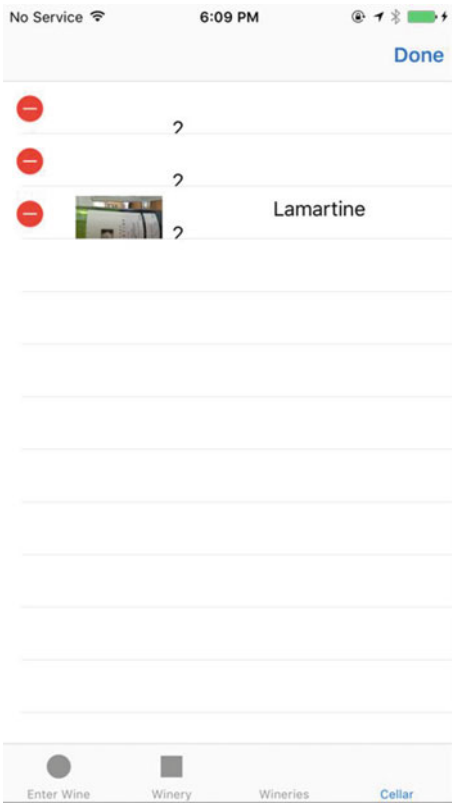

Figure 8-2. *The Cellar TableView in Edit mode*

Figure 8-3 shows the Delete button for the selected row. If you click on the button, the `override tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath)` will be called, and the selected item will be removed from the array using the `deleteRowsAtIndexPaths` function. As you may remember, we also call `deleteWineRecord` to delete the record from the database.

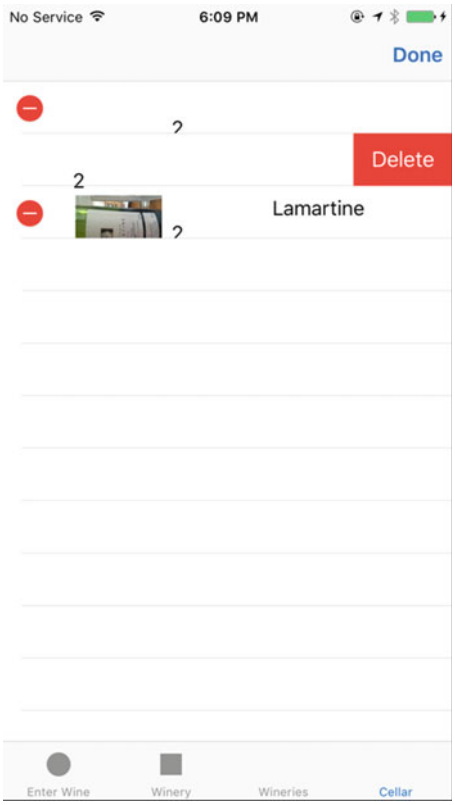

Figure 8-3. *the Selected item for deletion*

Figure 8-4 displays the `TableView` for the Wineries table view. It currently has only one entry. We will repeat the same process as before and click on the Edit button to switch into Edit mode.

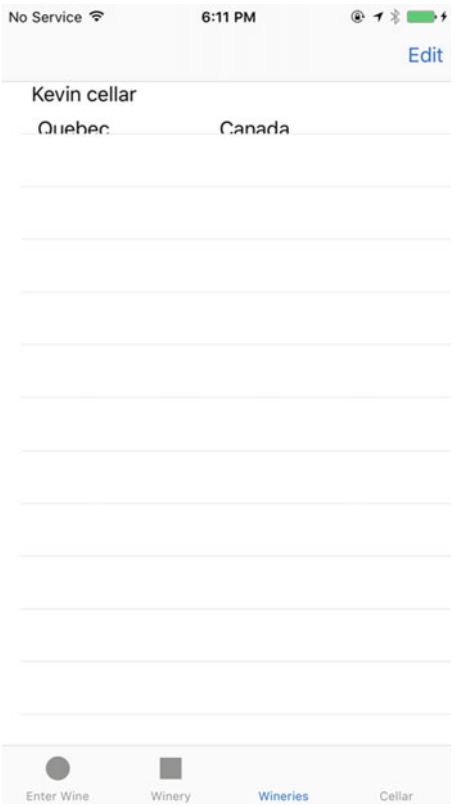

Figure 8-4. *The Wineries TableView*

As shown in Figure 8-5, with the `TableView` in Edit mode, you can select the entry, which will trigger the row to display a Delete button. Also notice the "Done" link, which you use to switch the `TableView` back into Read mode.

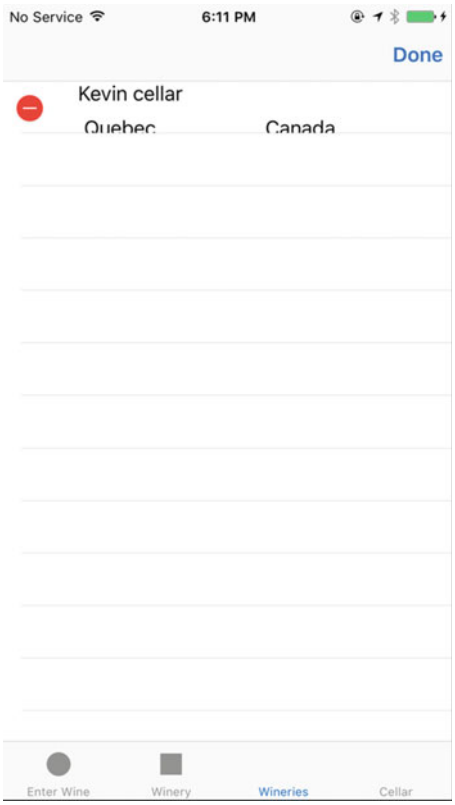

Figure 8-5. *The Wineries in Edit mode*

Figure 8-6 displays the Delete button, and if you click on the button the selected entry will be deleted from both the winery array and the winery table in the database.

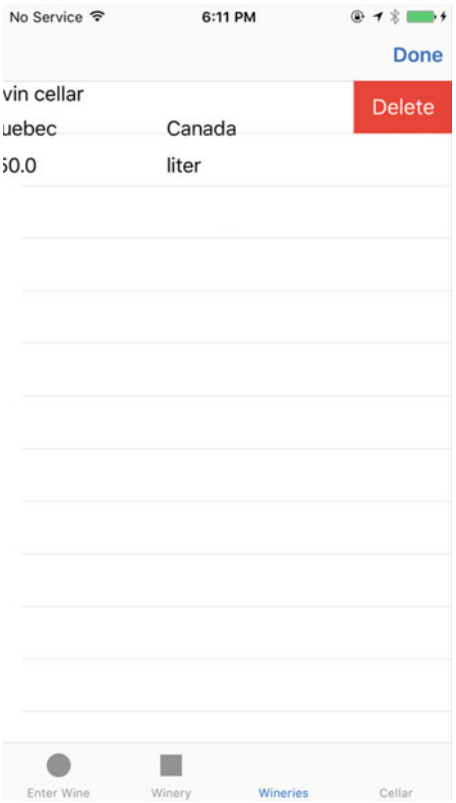

Figure 8-6. *The Wineries TableView with item selected for deletion*

Figure 8-7 displays the absent-entry Wineries `TableView`.

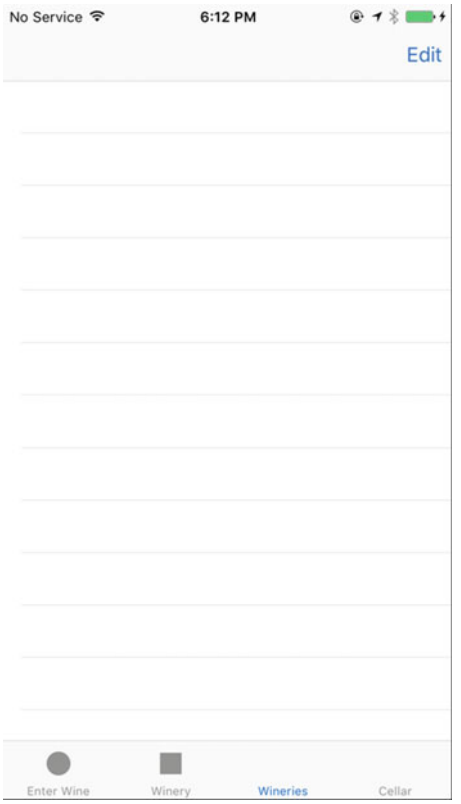

Figure 8-7. *The selected item has been deleted*

Summary

In this chapter we discussed how to use the SQLite DELETE statement to delete records from a SQLite database. Also how we can use a WHERE clause to limit the number of records to delete or to target a certain record or records.

We also explored how to implement the DELETE statement in Swift 3 and finally we added the delete functionality to the Wine app using Swift. In the next chapter, we will implement searching for records in a SQLite database and displaying those records.

CHAPTER 9

Searching for Records in SQLite

This chapter doesn't show any new SQLite APIs. Rather, it focuses on how you can use the SQLite APIs to create an iOS iPhone app for searching for records in a SQLite database. In this chapter, we will explore the following:

- Creating an iOS app
- Creating a SQLite database
- Adding the Search function
- Developing the UI for searches
- Searching for records
- Displaying search results
- Developing a UISearchBar iPhone app

The Search App

This tutorial demonstrates how to use the `SELECT` statement to search for content in a database using the `UISearchBar` and display the results in a `TableView` embedded in a `ViewController`. Figure 9-1 provides a visual of the running app.

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_9

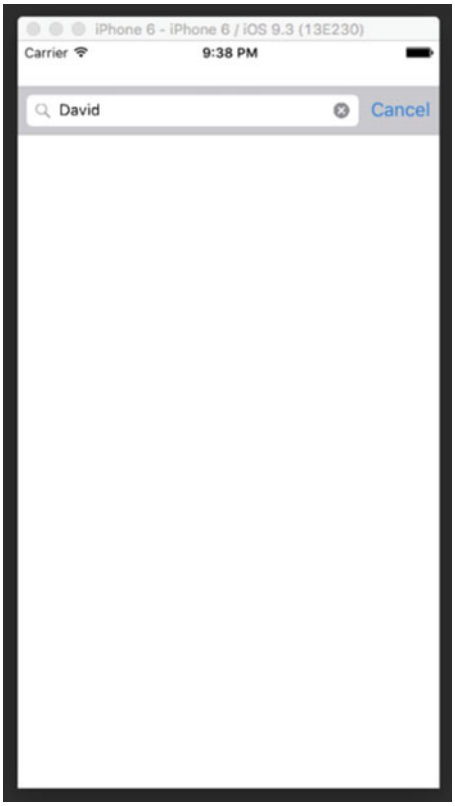

Figure 9-1. *The Search iPhone app*

The `UISearchBar` and `UITableViewController` are embedded in the `ViewController`. The search term is passed to a function in the `ViewController`.

The `UISearchBar` is an iOS component in the UIKit that was introduced in version 2.0 of the Cocoa Touch Framework and iOS SDK. The `UIControl` and protocol have several important features to help a developer quickly implement a Search field in their applications. You can enable a Search button, a Cancel button, and a Bookmark button. The delegate has methods that interact with those buttons that are pressed.

This tutorial will demonstrate how to quickly develop an iPhone app that searches a SQLite database using the `UISearchBar` text field. The database contains a list of names in separate columns. The app will implement a SQL query to search either field, then display the results in a `UITableView`.

The whole application is built using a single-view application template. The SQLite database is built and sample data is added to it using the SQLite Manager in Firefox.

Create the SQLite Database

For this example app, we will create a SQLite database using SQLite Manager in Firefox, which is a free add-on. As Figure 9-2 shows, we create a database titled `dbsearch.sqlite` and one table, `names`, to hold a sample of first and last names. The file should be saved to a convenient location, since it will have to be added to the iOS project later.

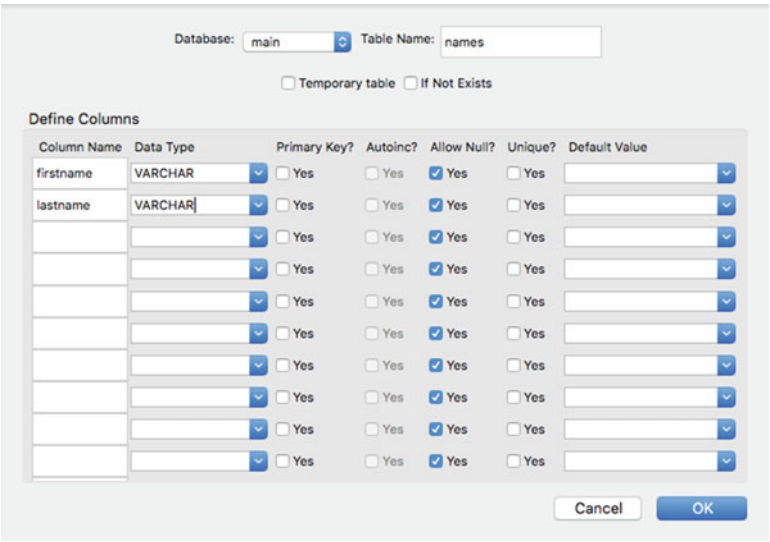

Figure 9-2. *Create dbSearch.sqlite and names table*

Then, we add two columns:

```
firstname:varchar
lastname:varchar
```

Figure 9-3 shows a screenshot of how SQLite Manager in Firefox creates an input screen based on the columns you define. Using SQLite Manager, we add some sampling data so that we can perform a search later. Figure 9-4 shows the sample data entered into the database through SQLite Manager in Firefox.

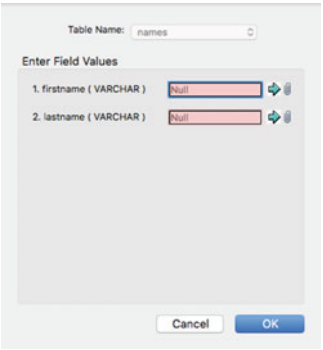

Figure 9-3. *SQLite data input*

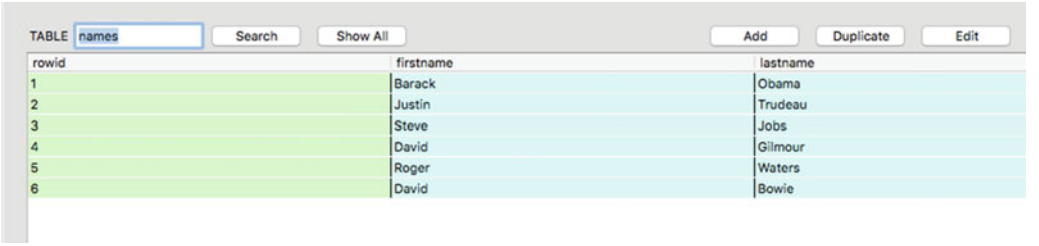

Figure 9-4. *Sample data in dbSearch.sqlite*

Create the iOS/SQLite Project

Figure 9-5 shows where to select a template for the project. Choose the single-view application template from the iOS Project category to create a simple iOS iPhone app. Call the app *SQLiteSearch* and ensure that the Swift language is selected in the language options.

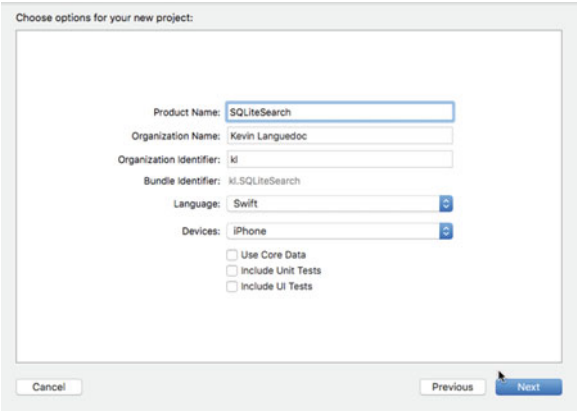

Figure 9-5. *Create the SQLiteSearch app*

Once the project is created, you need to add the sqlite3 library to the project and create the bridge. Figure 9-6 shows how to add the sqlite3 library to the project. Select the project root in the navigator and scroll to the Linked Libraries and Frameworks section on the General tab. Click on the "+" button to bring up the library selector popup. In the Search field, type "sqlite" and then select the libsqlite3.tbd library. Click Add to close the popup and add the library to the project.

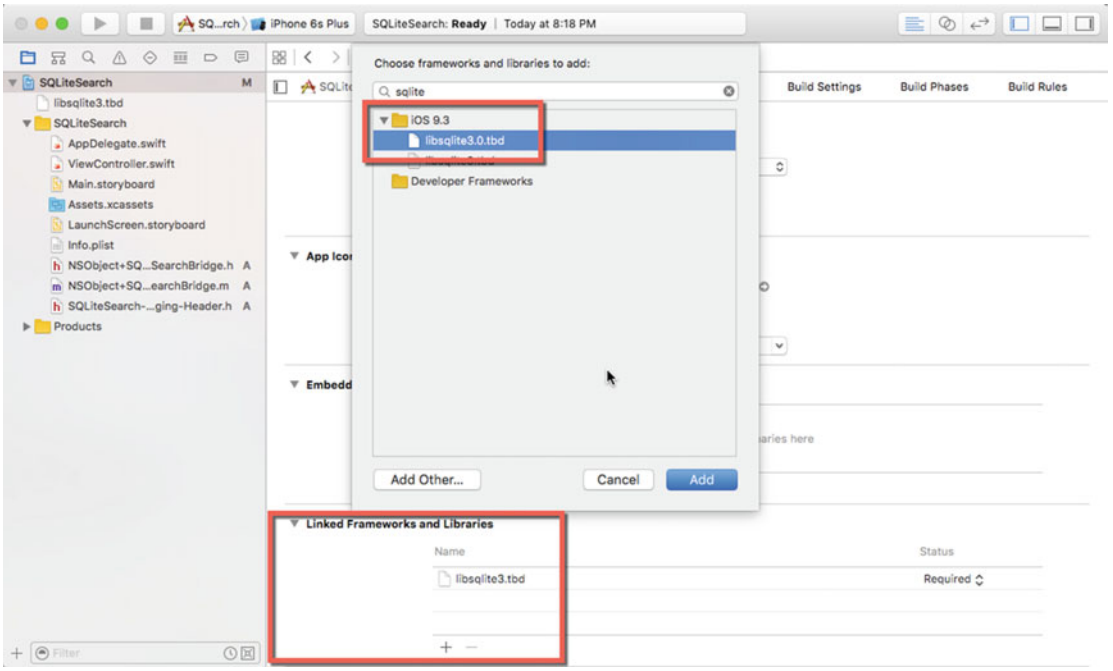

Figure 9-6. *Add the sqlite3 library*

Next, right-click on the `DbSearch` group and select the "Add files to ..." context-menu item. Browse to the location where the `dbsearch.sqlite` file was saved, select the database file, and click Add to begin copying the database to the project. A second popup will appear from which you will need to select the first option: "copy items to destination group's folder (if needed)" to actual the project. This is important, or only a reference will be added. Now that the project is set up, we can build the storyboard and controller logic.

Set Up the Bridge

With the library in place, we must set up the Objective-C bridge. Add a new file to the project by selecting the Cocoa Touch class template from the iOS Source category from the available templates (Figure 9-7), then do the following:

- Name the bridge *SQLiteSearchBridge*
- Select the category for the file types, which will trigger the Add Bridge Interface
- Select the `NSObject` class

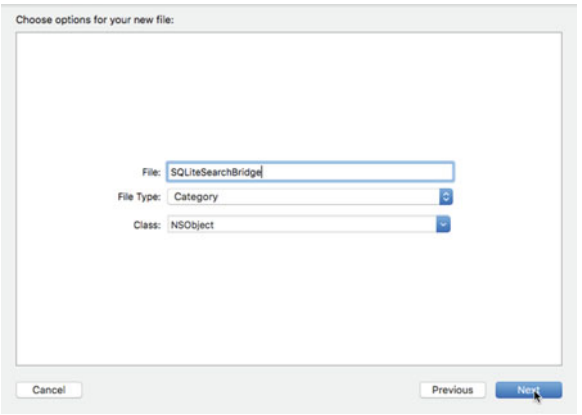

Figure 9-7. *Create the SQLiteSearchBridge*

Figure 9-8 shows the Create Bridge Interface, which appears after adding the file to the project. When we create the bridge using this method, Xcode will create the bridge file and add the file to the Build settings for the Swift Compiler. You can discard the Objective-C header and implementation files that are also created. In this project, the files are `NSObject+SQLiteSearchBridge.h` and `NSObject+SQLiteSearchBridge.m`. We only need the `SQLiteSearch-Bridging-Header.h` file. You need to open this latter file and add the `#import <sqlite3.h>` directive in order to interface with the SQLite3 API.

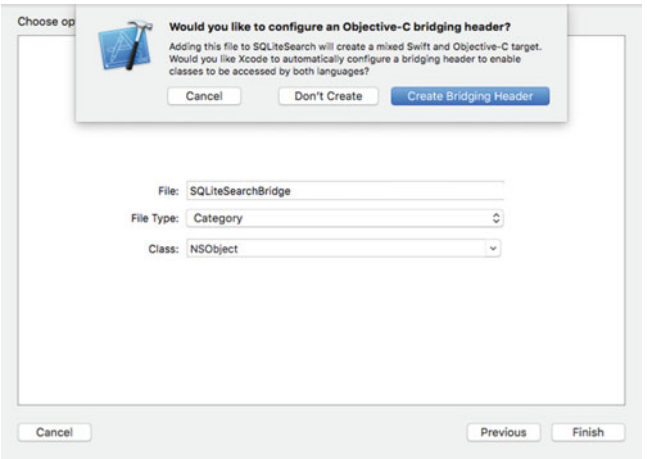

Figure 9-8. *Setting up the bridge in the Xcode*

The Controller Code

Before getting to the interface and adding the controls and the `IBOutlets`, we will need to add the code to the `ViewController` to handle the search operations and interface with the `UISearchBar, UITablewView`, and `UITableViewCell`.

Open the ViewController and add the UISearchBarDelegate, UITableViewDelegate, and UITableViewDataSource next to the class definition (see following code). We will also need to set up the delegates when the app loads, so we need to assign the delegates to self in the viewDidLoad function.

```
import UIKit

class ViewController: UIViewController, UISearchBarDelegate, UITableViewDelegate,
UITableViewDataSource   {

// code skipped for clarity

override func viewDidLoad() {
        super.viewDidLoad()
        searchResults.dataSource = self
        searchResults.delegate = self
        searchField.delegate = self
    }
```

A variable for an array will be used to store that data for the UITableView, hence its data source is needed as well. For this project, the array, nameList, is of the String data type. If you want to manipulate the data in the UITableView, you would interface with its data source. In other words, with the array. Also, we need to create an UnsafeBirCast pointer called SQLITE_TRANSIENT. This is a destructor pointer that is used in the sqlite3_bind_text later in the searchDatabase function.

```
var nameList = [String]()
internal let SQLITE_TRANSIENT = unsafeBitCast(-1, sqlite3_destructor_type.self)
```

For the UITableViewDelegate and UITableViewDataSource protocols, as well as for the UITableView itself, we need to add some required functions. These are described next.

The numberOfSectionsInTableView Function

To display the returned results, we will need to set up the UITableView delegate and data source. numberOfSectionsInTableView tells the table how many sections the table will have. Sections are groups of rows. We will set this to 1:

```
func numberOfSectionsInTableView(tableView: UITableView) -> Int{
    return 1

}
```

The tableView:numberOfRowsInSection Function

numberOfRowsInSection tells the table how many rows to display. It is customary to indicate the number of elements in the array or data source:

```
func tableView(tableView: UITableView,
                numberOfRowsInSection section: Int) -> Int{
    return nameList.count
}
```

The tableView:cellForRowAtIndexPath:indexPath Function

This method configures the UITableCell by getting a handle on the cell prototype in the UITableView and assigning the value at indexPath of the array to the label property of the cell. This method will be called repeatedly for each row that was defined in the numberOfRowsInSection function or the number of objects in the array.

The cell identifier, searchResultCell, will be configured later in the UI. Also, famousName is the UILabel IBOutlet that we will add later in the UITableViewCell. With the cell variable set up and cast as a searchResultCellTableViewCell UITableViewCell type, the other variable, nameObject, is assigned the value from the current array index. This value is then assigned to call.famousName.text:

```
func tableView(tableView: UITableView,
                cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell{
    let cell = tableView.dequeueReusableCellWithIdentifier("searchResultCell", forIndexPath:
        indexPath) as! searchResultCellTableViewCell

    // Configure the cell...
    let nameObj = nameList[indexPath.row]
    cell.famousName.text = nameObj

    return cell
}
```

So, for the UISearchBar delegate, UISearchBarDelegate, which requires the implementation of the searchBarCancelButtonClicked and searchBarSearchButtonClicked methods, the interactivity will respond to the buttons in the UISearchBar. For the UITableView, the UITableViewDelegate and UITableViewDataSource protocols. The function definitions are provided next.

The searchBarCancelButtonClicked Function

The searchBarCancelButtonClicked method will reset not only the UISearchBar field but also its data source and UITableView, and will also dismiss the keyboard. The complete code is provided here:

```
func searchBarCancelButtonClicked(searchBar: UISearchBar){
    self.searchField.text=""
    searchResults.reloadData()
    searchField.resignFirstResponder()
    self.view.endEditing(true)

}
```

The searchBarSearchButtonClicked Function

The searchBarSearchButtonClicked method is similar in design to searchBarCancelButtonClicked except that it calls the searchDatabase method. Then, the code resets the Search field, reloads the data in the UITableView, and resigns the keyboard:

```
func searchBarSearchButtonClicked(searchBar: UISearchBar){
    self.searchDatabase(searchField.text!)
    self.searchField.text=""
    searchResults.reloadData()
```

```
    searchField.resignFirstResponder()
    self.view.endEditing(true)
}
```

The searchDatabase Function

The other addition is a method to interface with the database: `searchDatabase`. This method will take one argument for the search term. Technically, you should be able to pass multiple search terms, which would be split into an array, but for simplicity's sake I am assuming a single-term search term.

The first variable, `fileExist`, is a Boolean. It will allow us to ensure that the `dbSearch.sqlite` file is available in the main application bundle. The `db` variable is a `COpaquePointer` for the SQLite database. Likewise, the `sqlStatement` variable is a `COpaquePointer` for the `sqlite3_stmt` statement.

In order to open the database, we need to get the path to the file in the main application bundle. We are leaving it there because we don't need to write to it and the main bundle is read-only during runtime. The `projectBundle` gets a handle to the `mainBundle` through the `NSBundle` class. Then the `fileMgr` constant is created as an `NSFileManager`. This class handles interactivity with the file system. The `resourcePath` String constant will be assigned the fully qualified path to the database using `pathForResource`.

Once these constants and variables are defined and assigned their initial values, we check to make sure the database file is available using `fileExistsAtPath`, which returns a Boolean value. If the database file exists, the database is opened using the `sqlite3_open` function. `SQLITE_OK` is returned if the database is successfully opened or will print an error message to the console if not.

Once open, we define a SQL `SELECT` query string, `sqlQry`, which takes two arguments in the `WHERE` clause. Both arguments will receive a copy of the `searchTerm` argument that is passed to the function from the `UISearchBar IBOutlet`. We then assign and bind the `inout` values to the SQL query using the `sqlite3_bind_text` function—one for each argument in the `WHERE` clause. Then, the code will execute and return the values if successful. These values are then assigned to the `concatName` constant and in turn are appended to the `nameList` array. Finally, the memory is cleaned up and the database is closed.

```
func searchDatabase(searchTerm:String){
    var fileExist:Bool = false
    var db:COpaquePointer = nil
    var sqlStatement:COpaquePointer=nil

    let projectBundle = NSBundle.mainBundle()
    let fileMgr = NSFileManager.defaultManager()
    let resourcePath = projectBundle.pathForResource("dbsearch", ofType: "sqlite")

    fileExist = fileMgr.fileExistsAtPath(resourcePath!)

    if(fileExist){
        if(!(sqlite3_open(resourcePath!, &db) == SQLITE_OK))
        {
            print("An error has occured.")

        }else{

            let sqlQry = "SELECT firstname,lastname  FROM  names where firstname=? or
            lastname=?"
            if(sqlite3_prepare_v2(db, sqlQry, -1, &sqlStatement, nil) != SQLITE_OK)
            {
                print("Problem with prepared statement " + String(sqlite3_errcode(db)));
```

```
            }
            sqlite3_bind_text(sqlStatement, 0, searchTerm, -1, SQLITE_TRANSIENT)
            sqlite3_bind_text(sqlStatement, 1, searchTerm, -1, SQLITE_TRANSIENT)
            while (sqlite3_step(sqlStatement)==SQLITE_ROW) {

                let concatName:String = String.fromCString(UnsafePointer<Int8>(sqlite3_
                column_text(sqlStatement,0)))! + " " + String.fromCString(UnsafePointer<Int8
                >(sqlite3_column_text(sqlStatement,1)))!

                print("This is the name : " + concatName)
                nameList.append(concatName)
            }
            sqlite3_finalize(sqlStatement);
            sqlite3_close(db);
        }

    }

}
```

With the array populated, the `UITableView` in the `ViewController` will display the results.

The searchResultCellTableViewCell Function

Before we get to the storyboard, we need to add a `UITableViewCell` object called `searchResultCellTableViewCell`. We will add the `IBOutlet` for the `UILabel` that we will add to the cell prototype next.

Develop the Storyboard

The storyboard is going to be very simple, with only a `UISearchBar` control, a `UITableView`, and the corresponding `UITableCell`. Figure 9-9 depicts the `UISearchBar` control that needs to be added to the `ViewController`. Before adding any controls, select the `ViewController` and set Simulator Size to iPhone 4.7 inch. You can adjust the setting in the Attributes inspector.

The UISearchBar & UITableView

Drag a `UISearchBar` onto the canvas and add it to the top of the scene (Figure 9-9). Figure 9-10 shows how to set the attributes for the Cancel and Search buttons. With the control key pressed, open the Attributes inspector and select the following options:

- Shows Search Results button
- Shows Cancel button

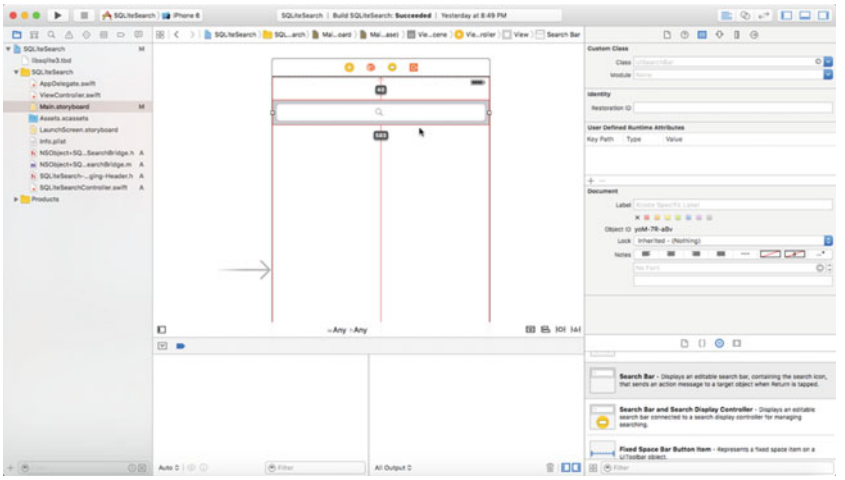

Figure 9-9. *Adding the UISearchBar to the ViewController*

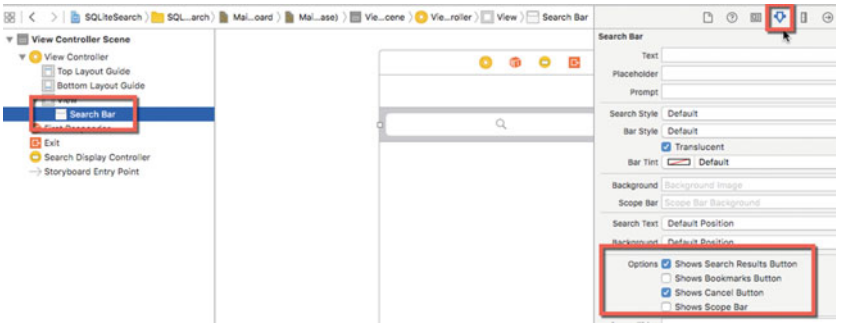

Figure 9-10. *Setting the UISearchBar attributes*

We won't need the other options for this example app. Next, add a `UITableView UIControl` and a `UITableCell` (Figure 9-11). Superimpose the cell control on the table. With the `UITableCell` selected, add an identifier through the Attributes inspector page. Drag a connection from the `UITableView` to the `UIViewController` proxy (yellow globe or circle on the bar at the top of the main scene), as shown in Figure 9-12. When you release the mouse button, a popup will appear to allow you to set the delegate and the data source. Set both of them.

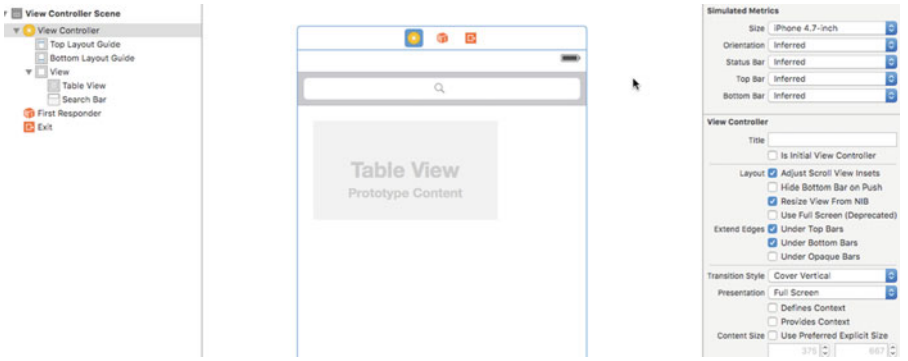

Figure 9-11. *Adding the UITableView and UITableViewCell*

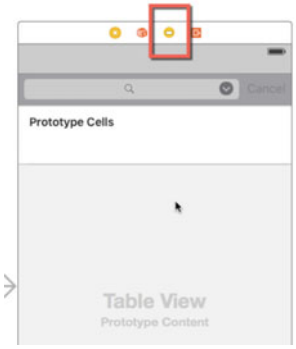

Figure 9-12. *Adding proxy for table delegate and data source*

The IBOutlets

As shown in Figure 9-13, next we will create the `IBOutlets`. Open the Assistant Editor by clicking on the double circle icon in the toolbar. To create the `IBOutlets`, drag a connection (ctrl + drag) using the mouse button to the open header file in the Assistant Editor. Releasing the mouse button activates a popup, thus allowing you to create an `IBOutlet` connection by entering a name for the connection in the appropriate field and clicking on Connect. For the `UISearchBar`, I named the `IBOutlet` *searchField*. Repeat this operation for the `UISearchBar` and the `UITableView`. For the `UITableView`, I named the outlet *searchResults*.

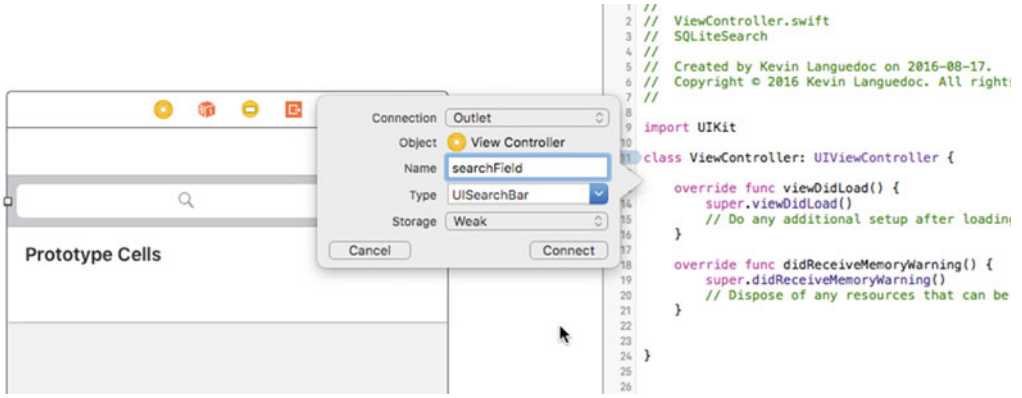

Figure 9-13. *Add the searchField IBOutlet*

```
@IBOutlet weak var searchField: UISearchBar!
@IBOutlet weak var searchResults: UITableView!
```

The Prototype Cell

Finally, for the prototype cell we need to configure the cell identifier and add a `UILabel` to display the search results. To configure the cell identifier, open the document outline and select `searchResultCell` from the document hierarchy (Figure 9-14). Then, open the Attributes inspector, add the `searchResultCell` name, and hit Enter. This value is used in the `ViewController`, as we saw in the previous section.

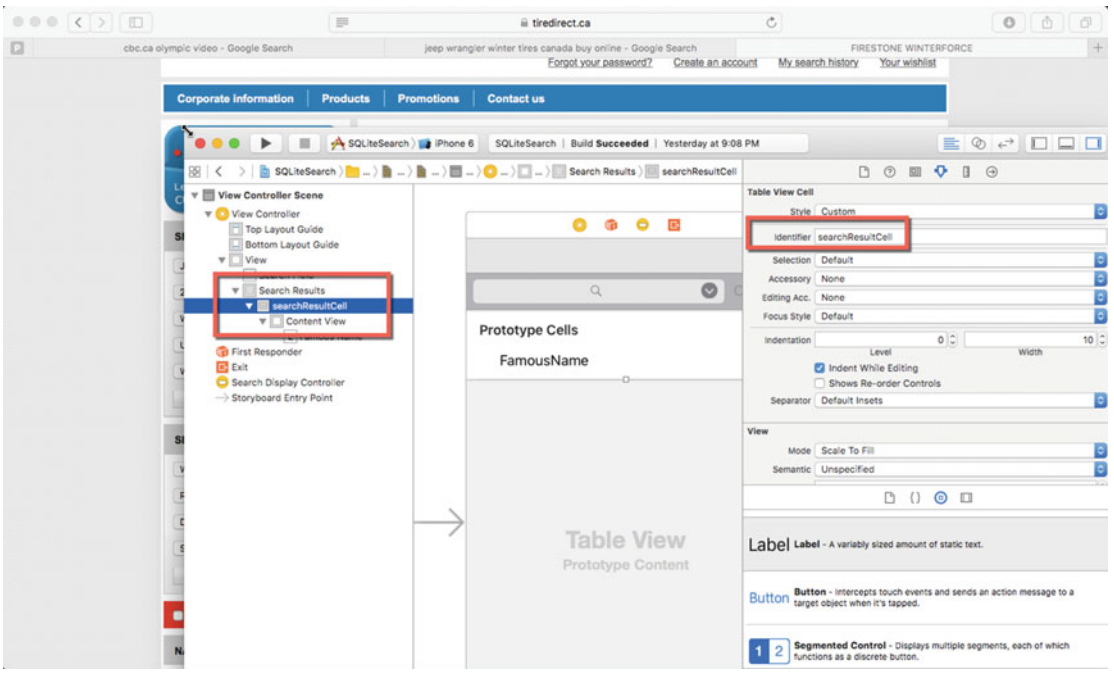

Figure 9-14. *Configuring the prototype cell*

Also add a `UILabel` to the cell and add an `IBOutlet` to the `searchResultCellTableViewCellController` by dragging a connection to the open Identity inspector.

```
import UIKit

class searchResultCellTableViewCell: UITableViewCell {

    @IBOutlet weak var famousName: UILabel!

    //code removed for brevity
}
```

All that is needed now is to run the app and test the functionality.

Running the App

As Figure 9-15 shows, when a user enters a name in the Search field and performs the search by clicking on the Search button, the results, if any, are fetched from the database and displayed in the database. The `UITableView` is reset before each search. For this example, I entered the name David and hit Enter, since I am testing in the simulator.

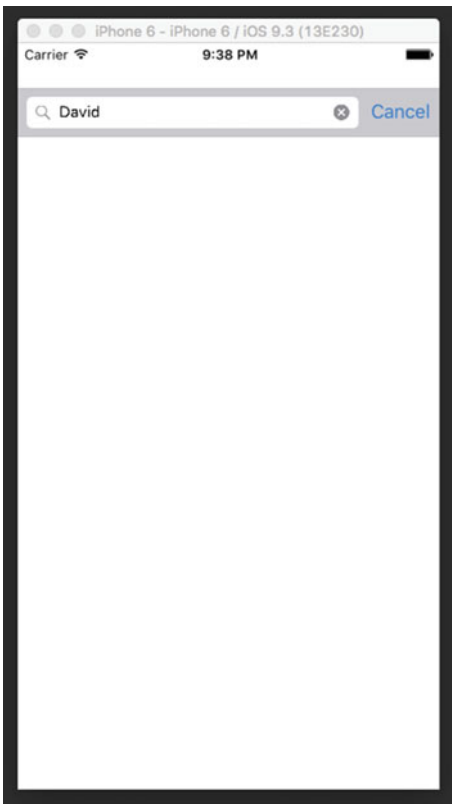

Figure 9-15. *Enter a search term, such as David*

Figure 9-16 displays two names from the database that match our search term.

Figure 9-16. *The search results*

Summary

In this chapter, we revisited the SQLite SELECT statement and implemented a database search function, which can be helpful for locating information in a database. The next chapter will focus on attaching and using multiple databases in a single file.

CHAPTER 10

Working with Multiple Databases

SQLite has a great feature for managing large volumes of data: multi-database apps. In this chapter, you'll see that by creating and attaching multiple databases to the same connection you can improve disk I/O performance and reliability.

Here are the topics covered:

- Overview of the `ATTACH` function
- Detaching databases using `DETACH`
- Multi-database limits
- Creating joins
- Working with `ATTACH AND DETACH` in Swift

The ATTACH Statement

SQLite 3 provides the `ATTACH DATABASE` statement for multi-database queries. Although you could manage multiple database connections via an Objective-C `DAO` class, it is much more efficient to use SQLite's API. Attaching databases means that more than one database is sharing the same database connection. To the SQLite engine, the first database that uses a database connection is known as **`main`**. Any additional databases that are attached must be assigned a new name to distinguish them in the connection pool. The syntax is very straightforward:

```
let attachdb = "ATTACH DATABASE database_file AS schema_name"
```

The value of `database_file` is the file name of the database to attach and `schema_name` is the alias for the database so that the database engine can identify the other databases. It also allows you as a developer to query these additional databases by using their alias.

To attach a SQLite database to an open connection, you first need to open an initial database, thus opening a connection to the SQLite engine. This will be the main database, and you must refer to this database by its schema `main.table_name`. Subsequently, you need to create the other databases as needed, either up front or later. With the databases in hand and after opening the first, or main, database, you can issue the `ATTACH` statement like any other SQL query, specifying the name and path of the database you wish to attach to the main database connection, as well as the alias or schema.

You would then refer to the other attached databases by their schema names, like `seconddb.table_name` or `attacheddb.table_name`. The name of the attached database schema is arbitrary and can be anything you like, except that it must be a single word or a compound word.

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_10

You can also attach in-memory databases. We haven't looked at in-memory databases within the scope of this book; however, you can create an in-memory database using the special `:memory:` keyword when opening a database, as in the following example:

```
sqlite3_open(":memory:", &db);
```

Each time you create an in-memory database using the preceding command, it exists separately from any others. Once the in-memory databases are created, you can attach them as usual, as follows:

```
let temp1 = "ATTACH DATABASE ':memory:' AS temp1;"
```

This will cause the second in-memory database (the second one created) to be attached to the first, or main, one. You can attach as many in-memory databases as are needed, as long as you remember that they will be attached in the sequence in which they were created.

If you are using temporary databases—again, out of the scope of this book—you create the temporary database using the following command:

```
sqlite3_open("", &db)
```

Then, you attach the temporary databases as usual:

```
Let tempdb = "ATTACH DATABASE ‘’ as temp1"
```

Both of these types of databases can be made to cease to exist once the SQLite connection is closed by using the command:

```
sqlite3_close(&db)
```

Also, it is possible to attach the same database more than once using different schema names. However, you must not have the `enable_cache_mode` enabled or you will get an error.

The DETACH Statement

The `DETACH` statement performs the opposite function of the `ATTACH` database. In other words, the command detaches a previously attached database from the open connection. Detaching a physical (`onOdisk`) database, an in-memory database, or a temporary database is performed by issuing the `DETACH` command followed by the schema name, as follows:

```
Let detach_db:String = "DETACH DATABASE schema_name"
```

Like the `ATTACH` statement, you can detach the same database that was attached multiple times using different schemas by using the same schema names to reference the databases to be detached. Like with the `ATTACH` operation, `enable_cache_mode` must be disabled or an error will be thrown.

Multi-Database Limits

Beyond the physical limit, in terms of the storage space of the iOS device where the app resides, the limit of attached databases is set using `SQLITE_LIMIT_ATTACHED` and the `sqlite3_limit` command. The syntax for the command is as follows:

```
int sqlite3_limit(sqlite3*, int id, int newVal);
```

The first parameter is the pointer to the open database connection; the second is one of the limit categories (listed next); and the last parameter is the new limit of the attached databases. For attached databases, you need to use the SQLITE_LIMIT_ATTACHED constant.

If a negative number is used for the third parameter, no changes are made to the database limits.

RUN-TIME LIMIT CATEGORIES	VALUE
SQLITE_LIMIT_LENGTH	0
SQLITE_LIMIT_SQL_LENGTH	1
SQLITE_LIMIT_COLUMN	2
SQLITE_LIMIT_EXPR_DEPTH	3
SQLITE_LIMIT_COMPOUND_SELECT	4
SQLITE_LIMIT_VDBE_OP	5
SQLITE_LIMIT_FUNCTION_ARG	6
SQLITE_LIMIT_ATTACHED	7
SQLITE_LIMIT_LIKE_PATTERN_LENGTH	8
SQLITE_LIMIT_VARIABLE_NUMBER	9
SQLITE_LIMIT_TRIGGER_DEPTH	10
SQLITE_LIMIT_WORKER_THREADS	11

Performing Joins

Building queries that take multiple databases into account is exactly like building joins between tables within the same database. The one caveat that you need to remember is that you need to reference the table or tables in the additional databases by prefixing the table with the database schema name, as the following example demonstrates.

SQLite supports three types of join: INNER JOIN, OUTER JOIN, and CROSS JOIN. All three can be formed using multiple databases that are attached to the same open connection. See this example:

```
SELECT c.name, a.address, a.city FROM mainDb.contacts c INNER JOIN secondDb.addresses a ON
c.name, a.name
```

Attach and Detach in Swift

This is a simple single-view iPhone app to demonstrate and test the ATTACH and DETACH SQLite functionality in Swift. The app will provide a function to create databases (and open them). There will also be a function to ATTACH and DETACH a database from the main database.

For the sake of simplicity, another function will pre-populate the databases using a sampling of city names and the countries in which they are located. A final function will fetch the data from the database and display the data in a TableViewController.

The Project

After the project is created, we need to add the sqlite3 library as we have seen before; it is found in the Linked Libraries and Framework section in General settings.

With the library in place, we will add the bridge using the Objective-C class template and choosing the Category file type, which will trigger Xcode to create the bridge file for us and set up the Build settings for the Swift compiler. Of course, we still need to add the `#import <sqlite3.h>` header reference in the bridge file to actually interface with the sqlite3 library.

With the project set up, we can add the logic to the `ViewController` to handle the multi-database operations.

The ViewController

The `ViewController` is the main entry point for the app's logic. Other than the global variables and constants that are usually needed in order to interface with SQLite, the `ViewController` will have six functions. We will explore these in the following sections.

Since the main view in the storyboard will contain a `UIPickerView` for the created databases, we will need to add the delegate and data source protocols:

```
class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource
```

The Global Variables and Constants

The `ViewController` is created by default when the Single View Application template is used to create an app. To work with SQLite, we need a variable for the database name and path, which is called `dbPath` for this app. The databases array will contain the database names so that we can select a database to attach to the main one later.

We also need a pointer, or `COpaquePointer` variable, for the SQLite database connection and a variable for the SQLite 3 statement. These are named `db` and `sqlStatement` respectively. `SQLITE_TRANSIENT` is a movable pointer for the SQLite destructor, which uses the `sqlite3_destructor_type` type. This has to be cast using the `unsafeBitCast` function in Swift.

```
internal let SQLITE_TRANSIENT = unsafeBitCast(-1, to:sqlite3_destructor_type.self)
var databases = [String]()
var dbPath = URL()
var db:COpaquePointer? = nil
var sqlStatement:COpaquePointer? = nil
var attachDb:String = ""
```

The IBActions and IBOutlets

Although the `IBActions` and `IBOutlets` will be added through the UI storyboard later, I wanted to mention the code that is added to the `IBActions`. The `saveBtn` will call the `createDatabase` function explained later. After the createDatabase function is called, the UIPickerView is re-loaded with the new value in the database array, which is the `UIPickerView`'s data source. The `attachDatabase` function is called from the `AttachDbBtn`, and `detachDatabase` is called from `detachDbBtn`.

The first parameter value, `attachDb`, is populated from the `dbnamePicker pickerView` function that we will look at later. Likewise, for the second parameter's values, we simply concatenate the "schema" string to the name of the `attachDb` variable.

```
@IBOutlet weak var dbnamePicker: UIPickerView!
@IBOutlet weak var dbnameField: UITextField!
@IBAction func saveBtn(sender: AnyObject) {
    self.createDatabase(self.dbnameField.text!)
    dbnamePicker.reloadAllComponents()
}

@IBAction func attachDbBtn(_ sender: AnyObject) {
    let dbame = attachDb.components(separatedBy: ".")[0]
    self.attachDatabase(self.getDbToAttach(attachDb).path!, schemaName: dbame+"_schema")
}

@IBAction func detachDbBtn(_ sender: AnyObject) {
    let dbame = attachDb.components(separatedBy: ".")[0]
    self.detachDatabase(dbame+"schema")

}
```

The viewDidLoad Function

The `viewDidLoad` function sets the delegate and data source for the `UIPickerView` object. Afterward, the code retrieves a list of the sqlite files in the Document directory and populates the databases array:

```
    override func viewDidLoad() {
    super.viewDidLoad()
    dbnamePicker.delegate = self
    dbnamePicker.dataSource = self
    // Get the document directory url
    let documentsUrl =  FileManager.default.urlsForDirectory(.documentDirectory,
    inDomains: .userDomainMask).first!

    do {
        // Get the directory contents urls (including subfolders urls)
        let directoryContents = try FileManager.default.contentsOfDirectory( at:
        documentsUrl, includingPropertiesForKeys: nil, options: [])
        print(directoryContents)

        // if you want to filter the directory contents you can do so like this:
        let sqliteFiles = directoryContents.filter{ $0.pathExtension == "sqlite" }

        databases = sqliteFiles.flatMap({$0.lastPathComponent})
        print("list:", databases)
        self.setDbpath()
    } catch let error as NSError {
        print(error.localizedDescription)
    }

}
```

The getDbToAttach Function

This little function merely gets the name and path of the database to be attached. The code retrieves the path value from the Document directory and stores the value into the `db_to_attach_path` variable:

```
func getDbToAttach(_ db_to_attach:String)->URL{
        let dirManager = FileManager.default)
        var db_to_attach_path = URL()
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)
            db_to_attach_path = try! directoryURL.appendingPathComponent(db_to_attach)
            } catch let err as NSError {
            print("Error: \(err.domain)")
        }
        return db_to_attach_path
    }
```

The setDbpath Function

The `setDbPath` function sets the path value of the selected database name in the `UIPickerView` list and stores it in the `dbPath` variable. This function is called when the app is launched:

```
    func setDbpath(){
        let dirManager = FileManager.default()
        let dbname = "cities.sqlite"
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)
            dbPath = try! directoryURL.appendingPathComponent(dbname)

        } catch let err as NSError {
            print("Error: \(err.domain)")
        }
    }
```

The createDatabase Function

To set up the databases, we will use the `createDatabase` function. The function will get a handle on the Document directory and assign it to `directoryURL`. Next, the database name, using the input parameter *database*, will be assigned to the `dbPath` variable by using the `URLByAppendingPathComponent` function.

Once the database name and path are set, `sqlite3_open` will create the database file and open the database. Any errors are caught by the `NSError` class. See here:

```
    func createDatabase(_ database:String){
        let dirManager = FileManager.default()
        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.SearchPathDirectory.
            documentDirectory, in: FileManager.SearchPathDomainMask.userDomainMask,
            appropriateFor: nil, create: true)
```

```
            dbPath = try! directoryURL.appendingPathComponent (database)
            if(!(sqlite3_open(dbPath.path!, &db) == SQLITE_OK))
            {
                print("Unable to create database")

            }else{
                print("Database: " + database + " successfully created ")
                databases.append(database)
                sqlite3_close(db);
            }
        } catch let err as NSError {
            print("Error: \(err.domain)")
        }
    }
```

The attachDatabase Function

This function's singular purpose is to attach a database to the open connection. To keep things simple, minimal error checking is added. `sqlite3_open` ensures that the main database is opened and that the `ATTACH` query string is passed to the `sqlite3_exec` function. If there are no errors with the query, the input values are bound to the query using Swift inline text binding and the query is executed. The query memory is released, and the connection is closed to complete the operation. See here:

```
func attachDatabase(_ dbName:String, schemaName:String){
        var err:UnsafeMutablePointer<Int8>? = nil
            if(!(sqlite3_open( dbPath.path!, &db) == SQLITE_OK))
            {
                print("An error has occured.")
            }else{
                let dbstatus = sqlite3_open(dbName, &db)
                print(dbstatus)
                if(sqlite3_open(dbName, &db) == SQLITE_OK){
                    let attachSQL = "ATTACH DATABASE '\(dbName)' AS '\(schemaName)'"
                    let status = sqlite3_exec(db, attachSQL, nil, &sqlStatement, &err)
                    if(status != SQLITE_OK)
                    {
                        print("Problem with prepared statement " + String(sqlite3_
                        errcode(db)));
                    }
                    if (status == SQLITE_OK) {
                        print("Database : " + dbName + " attached as " + schemaName)
                    }
             }

                sqlite3_finalize(sqlStatement);
                sqlite3_close(db);
            }
    }
```

The detachDatabase Function

The detachDatabase function is similar in design to attachDatabase. After ensuring that the main database is open using the sqlite#_open function, the DETACH query string is passed to the sqlite3_exec function along with the pointer for the sql3_statement pointer. The name of the schema is passed to the query using the Swift inline text binding method. If the query is successfully executed, the memory is cleaned up and the main database connection is closed. See here:

```
func detachDatabase(_ dbName:String, schemaName:String){
         var err:UnsafeMutablePointer<Int8>? = nil
        if(!(sqlite3_open(dbPath.path!, &db) == SQLITE_OK))
        {
            print("An error has occured.")

        }else{
            let detachSQL = "DETACH DATABASE '\(schemaName)' "
            var status = sqlite3_exec(db, detachSQL, nil, &sqlStatement, &err)

                status = sqlite3_prepare_v2(db, "PRAGMA database_list", -1, &sqlStatement,
                nil)
                while(sqlite3_step(sqlStatement) == SQLITE_ROW){
                    print(sqlite3_column_int(sqlStatement, 0))
                    print(sqlite3_column_text(sqlStatement, 1))

                }

                sqlite3_finalize(sqlStatement);
                sqlite3_close(db);

        }

    }
```

The UIPickerView Functions

The UIPicker functions include numberOfComponentsInPickerView, which configures the number of columns in the UIPickerView. The pickerView with the numberOfRowsInComponent specifies how many rows will be displayed in the UIPickerView. This return value is usually the data source's element count. titleForRow displays the actual value in the UIPickerView for each element in the data source. widthForComponent and rowHeightForComponent are helpers to set the width and height of the cells that display the elements in the UIPickerView. Finally, didSelectRow returns the selected value. This is used to select the database to attach. See here:

```
func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return 1
    }

    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        return databases.count
    }
```

```
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component:
Int) -> String?     {
    let dbname:String = databases[row]
    return  dbname
}

func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent
component: Int) {
    attachDb = databases[row]
}

func pickerView(_ pickerView: UIPickerView, widthForComponent component: Int) -> CGFloat {
    return 250.0
}

func pickerView(_ pickerView: UIPickerView, rowHeightForComponent component: Int) ->
CGFloat {
    return 50.0
}
```

Building the UI

Figure 10-1 shows the application's minimalist design, which allows a user to create a database or select a database from the `UIPicklist`. The UI features a `UITextField` to allow a user to create a new database by clicking the Save button.

The `UIPicklist` contains a list of databases that are stored in the Documents directory. To attach or detach a database, a user selects a database from the list and clicks on the appropriate button. As you may have noticed, these databases don't contain any tables or other design elements. The app's purpose is demonstrating the minimal code needed to attaching and detaching databases.

For this project, you are going to need the following components, which are laid out as in Figure 10-1:

Name	Connection Type	Design Element
Create SQLite Db	None	UILabel
DbnameField	IBOutlet	UITextField
Save	IBAction	UIButton
DBNamePicker	IBOutlet	UIPickerList
Attach	IBAction	UIButton
Detach	IBAction	UIButton

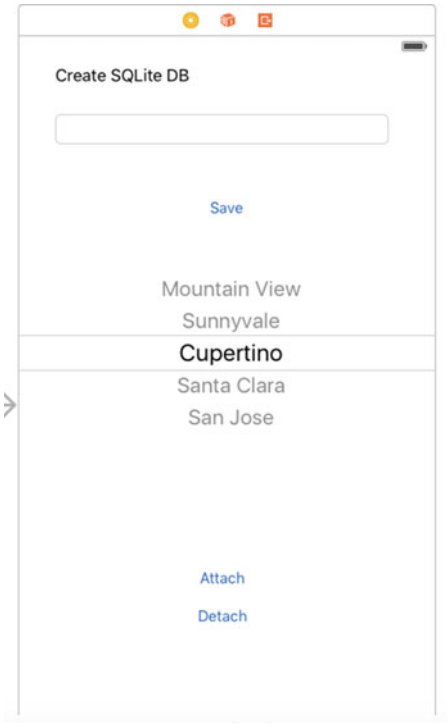

Figure 10-1. *UI design*

Once the design elements are in place, we need to create connections with the `ViewController`. Open the Identity assistant and control + drag a connection to the open `ViewController` Swift files above the `viewDidLoad` function, as we previously discussed in the `ViewController` section (Figure 10-2).

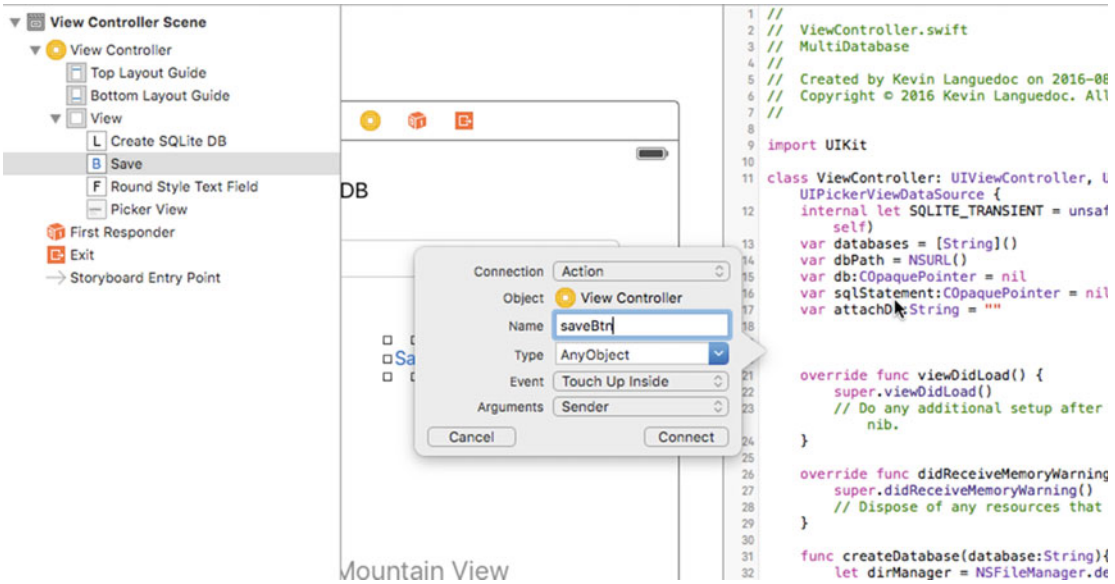

Figure 10-2. *Adding the IBActions and IBOutlets*

Running the App

Figure 10-3 illustrates the running app. For this example, I have added two databases: `cities.sqlite` and `countries.sqlite`. A third one was added for testing purposes. The `cities.sqlite` database is the "main" database and is opened when the app is launched. By selecting `countries.sqlite` and clicking the Save button, the `countries.sqlite` database is attached.

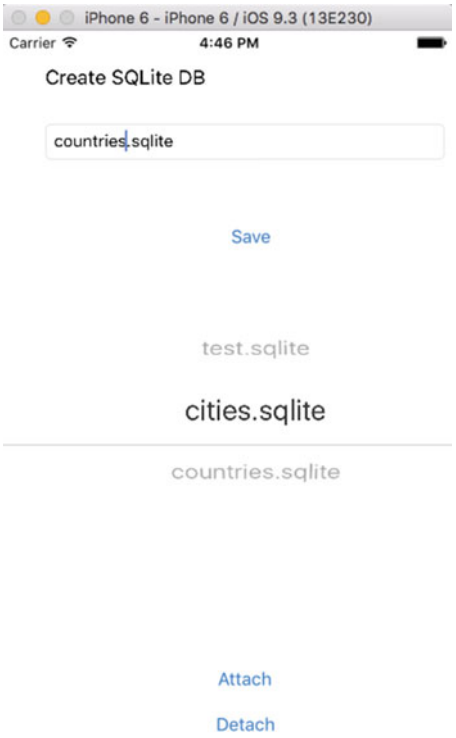

Figure 10-3. *The running app*

Figure 10-4 shows output for the attached database. After the `sqlite3_exec` function is executed, the `SQLITE_OK` or `0` status code is returned stating whether the operation was successful. Likewise for the detach operation, which is executed through the `detachDatabase` function (Figure 10-5).

Figure 10-4. *The attach status*

Figure 10-5 shows the output of the PRAGMA `database_list` command, which returns the list of databases attached to a connection. The output console shows the two database pointers and their index number in the array.

Figure 10-5. *PRAGMA database_list output*

Figure 10-6 shows the output of the detached database, `countries_schema`, which was previously attached.

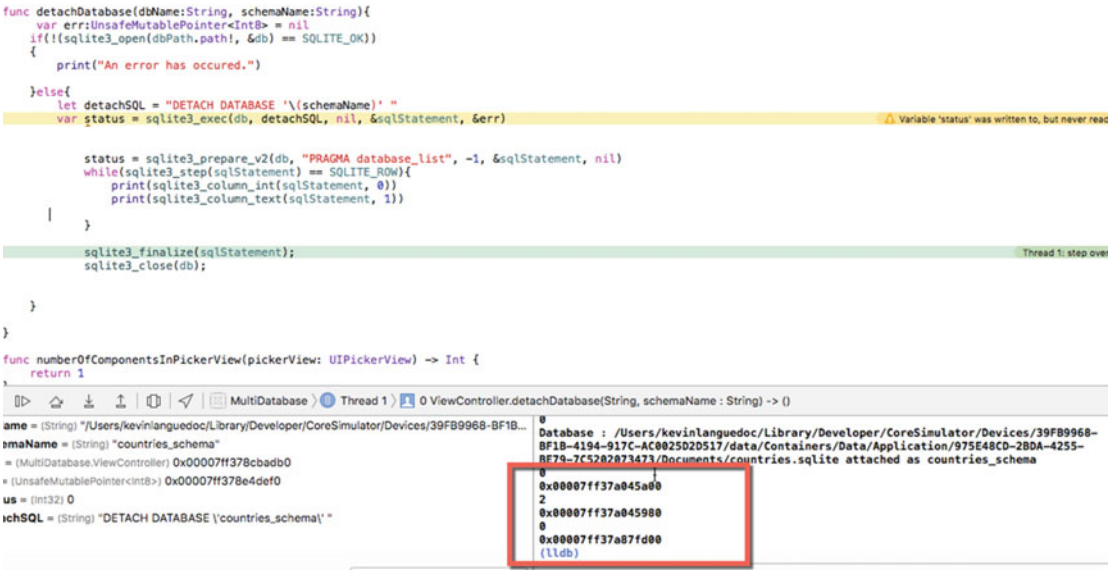

Figure 10-6. *The database_list output minus the second database*

Summary

In this chapter, you saw that attaching and detaching databases is a simple process. Although using multiple databases is not always necessary depending on your app's needs, doing so can improve disk I/O performance and reliability.

CHAPTER 11

Backing Up SQLite Databases

- Copy contents to new database
- Backing up on-disk databases
- Backing up in-memory databases

Overview of the SQLite Backup Methods

Prior to the introduction of the Backup API, the only way to perform a backup was to make a copy. While this is still a viable option, it does have its drawbacks, like trying to make a copy while there is a lock on the open database, for instance. Another drawback is if the app stops running before the copy is finished, the app would need to delete the copy and start over. In addition to those two examples, there are numerous ways a copy process can fail and possibly corrupt the database files. In short, there isn't any kind of intelligence built in to this option.

But in a jam, this option will and does work. In a Swift iOS setting, your app would need to use the Swift `FileManager` class to physically make a copy of the database and move it off to another location. Then, you would need to figure out how much data to keep in the database and then remove excess data from the production database to free up space, if needed. To regain some space, you could also run the `VACUUM PRAGMA` option.

Make a Backup Copy

The following code snippet is an example of a possible way to perform a backup using `FileManager`. The logic is quite simple:

1. First, you ensure that the database is closed using the `sqlite3_close` command.
2. Then, you get a handle on the file system using the `FileManager.defaultManager` property.
3. Next, you use the `copyItemAtPath` to physically make a copy of the file.
4. Then, you remove *n*th records from the database, if needed.
5. Finally, you run `VACUUM` to remove the empty space and re-index the indexes for good measure.

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_11

```
func backupCopyDatabase(){
        if sqlite3_close(db) == SQLITE_OK{
            let fileMgr = FileManager.default

            do {
                try fileMgr.copyItem(atPath: self.sourcedb, toPath: self.targetdb)
                let deleteSQL = "DELETE FROM main.table"
                if(sqlite3_exec(db, deleteSQL, nil, sqlStatement, err) == SQLITE_OK){
                    if(sqlite3_exec(db, "VACUUM", nil, &sqlStatement, &err) == SQLITE_OK){
                        print("database compressed")
                    }
                })

            }
            catch let error as NSError {
                print("Backup error: \(error)")
            }

        }

    }
```

Back Up In-Memory SQLite Databases

SQLite has a Backup API that can support backups on running databases and in-memory databases. The backup works by copying the contents from one database into another SQLite database. When the API copies the contents from the source to the target, it will overwrite the contents in the target, so you may need to make different copies if you need to preserve all versions of the content. You can also restore a database in a similar fashion.

The Backup API allows you to make backups while there is still a shared lock on the source database. If you lose power or have other issues, like having an app crash, the Backup API will remember where it left off and continue from that point.

Backing up a database using the API has three distinct steps:

- Initialize using `sqlite3_backup_init`.
- Execute the backup with `sqlite3_backup_step`.
- Finally, clean up the operation using `sqlite3_backup_finish`.

To check to see if there are still records to back up, you can use the `sqlite3_backup_remaining` function, which returns the number of records remaining in the database. The `sqlite3_backup_pagecount` provides the total count of pages to be backed up in the source database.

It is important to ensure there aren't any locks prior to starting a backup; otherwise, SQLite will immediately return a `SQLITE_BUSY` error. To avoid any conflicts, it is best to test to see if the database is accessible for backup. To run these tests, the code should implement `sqlite3_busy_handler` or `sqlite3_busy_timeout`.

The former calls a callback function in the event that an error occurs. The latter sets a timer in milliseconds in case of a lock situation. It can be called repeatedly until the lock is removed.

Back Up On-Disk SQLite Databases

Backing up on disk is similar to the first option, but uses the SQLite API. With this option, the app is actually moving data from the source database to the target database. Both databases are physical files. In theory, the target database could be on a remote server or in the same Document directory as the source.

The on-disk operation uses four different functions to perform the backup:

- `sqlite3_backup_init`
- `sqlite3_backup_step`
- `sqlite3_sleep`
- `sqlite3_backup_remaining`
- `sqlite3_backup_pagecount`
- `sqlite_backup_finish`

`sqlite3_backup_init` is called to create the `sqlite3_backup` object. The function takes a source database pointer and a target database pointer as arguments. Either one of these pointers can be for an in-memory database and the other for a physical database, or both can be physical database pointers.

The next operation is the `sqlite3_backup_step` function, which is repeatedly called to copy the *n*th number of pages specified in argument 5 to the target database. At the end of each iteration, the app needs to call `sqlite3_sleep`, which freezes the process for 250 ms in order for the write process to complete. `sqlite3_backup_remaining` returns the remaining amount of pages in the database after the `sqlite3_backup_step` function has executed. I usually call the `sqlite3_backup_pagecount` function at the beginning of the process to determine how many pages are in the database, as the sample will demonstrate later in this chapter, and then call the `sqlite3_backup_remaining` function after each iteration.

Once the source database is completely backed up, the `sqlite3_backup_finish` function is called to clean up the resources allocated with the `sqlite3_backup_init` function.

The Backup App

To demonstrate the various APIs for backup, I will create a single-view iOS iPhone app. The app will implement two methods for the SQLite Backup APIs in addition to `backupCopyDatabase` from earlier.

I am using a copy of the Chinook database (`https://chinookdatabase.codeplex.com/`) for sample data. You will need to import the database into the project if you intend to try out the code. To make the database writeable, it needs to be copied into the Documents directory, so add the method `copyDatabaseIntoDocuments` to the project.

Figure 11-1 shows the second screen, in which you enter a name for the Single View Application project. For this example, I will name the project BackupSQLite. The language is Swift, of course, and the target device is iPhone. You can leave the other options as is.

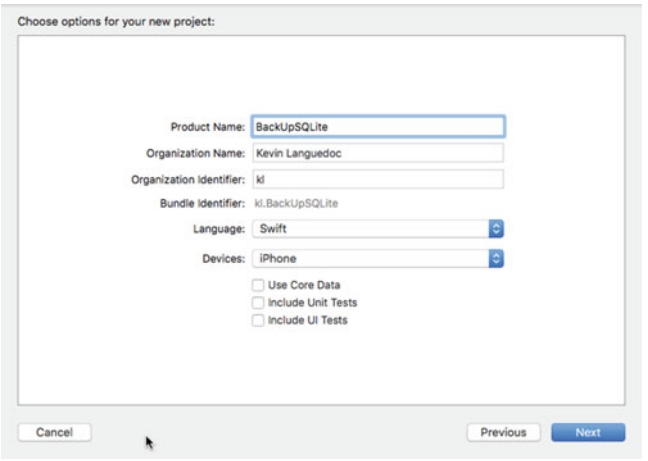

Figure 11-1. *The Backup Project: Single View Template*

As we will be working with SQLite databases and the SQLite Backup API, we will need to create a bridge, as usual. Select the Objective-C File template and click Next to move to the next screen, where you will need to specify a name for the bridge and the file type, which should be Category (Figure 11-2). Choosing this option will initiate Xcode, which will request to create the bridge file for you and set up the Swift Build settings.

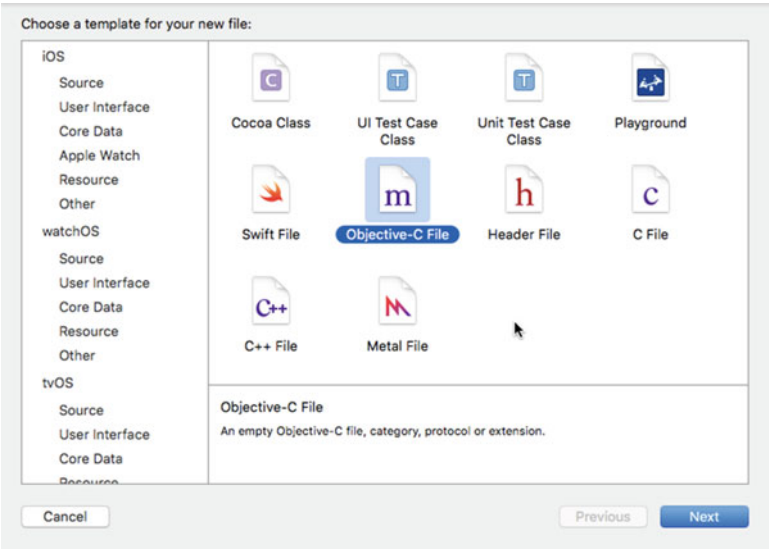

Figure 11-2. *The bridge interface*

Figure 11-3 provides a screenshot of the options for the bridge file. The first field is for the name of the file. In this case it is BackupBridge. Selecting the Category file type will trigger the Create Bridge interface once the file is saved to the project.

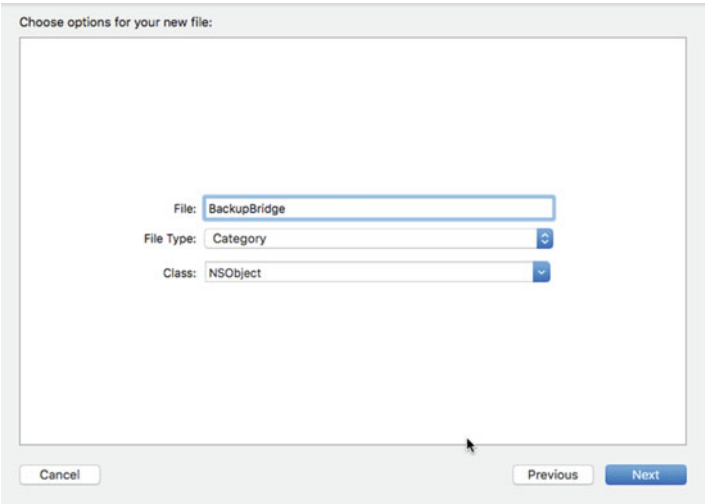

Figure 11-3. *The BackupBridge file*

Figure 11-4 shows the interface that allows Xcode to set up the bridge file and adds the setting to the Build settings.

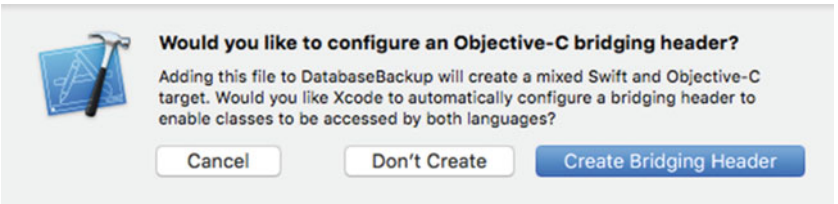

Figure 11-4. *The Create Bridge Header interface*

Figure 11-5 provides a visual of the Swift Compiler Build settings that are automatically configured by the Objective-C bridge setup process.

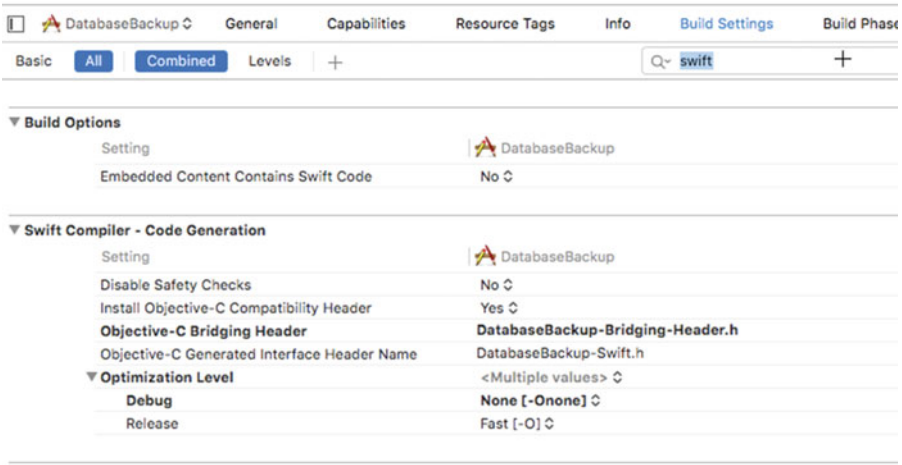

Figure 11-5. *The Swift Compiler Build settings*

Remember to add the sqlite3 library to the Linked Libraries (Figure 11-6), as we need to use the sqlite library.

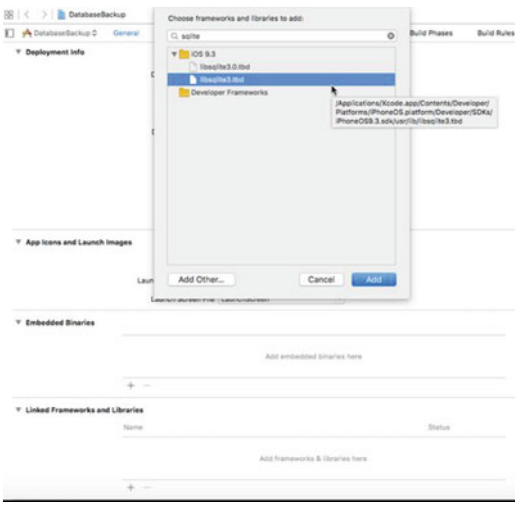

Figure 11-6. *Add the sqlite library*

Next, add the header to the `DatabaseBackup-Bridging-Header.h` header file using the `import` keyword, as in the code snippet here:

```
#import <sqlite3.h>
```

With the bridge in place, we can add the code to perform backups.

The ViewController

The `ViewController` is the main controller for the application. We add three database pointers and a pointer to the `sqlite3_backup` object. We also add a `FileManager` to perform file operations like copying the database files to the Documents directory. The source and target databases are defined as `sourcedb` and `targetdb`, and the last two `COpaquePointers` are for the `sqlite3_statement` and sqlite3 database errors.

```
    var db:OpaquePointer? = nil  // SQLite database connection for sourceDB filename
    var bakdb :OpaquePointer? = nil  // the SQLite Backup Object
    var filedb :OpaquePointer? = nil  // the SQLite Backup Object
    let fileMgr:FileManager = FileManager.default
    let sourcedb:String = "Chinook_Sqlite.sqlite" // The SQLite database to be backed up
    let targetdb:String = "backup_chinook.sqlite"
    var sqlStatement:OpaquePointer? = nil
    var err:UnsafeMutablePointer<Int8>? = nil
```

The viewDidLoad and copyDatabaseIntoDocument Functions

In the `viewDidLoad` function, we call the `copyDatabaseIntoDocuments` function for each of the databases. This function simply gets a handle on the files and on the Document directory and uses `openItem` to copy the files to the Document directory if the files don't already exist:

```
override func viewDidLoad() {
        super.viewDidLoad()
        self.copyDatabaseIntoDocuments(sourcedb)
        self.copyDatabaseIntoDocuments(targetdb)
    }

func copyDatabaseIntoDocument(_ dbFilename:String){

        var srcPath:URL
        var destPath:URL
        let dirManager = FileManager.default
        let projectBundle = Bundle.main
       do {
            let resourcePath = projectBundle.pathForResource(dbFilename.
            components(separatedBy: ".")[0], ofType: "sqlite")
            let documentURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)

            srcPath = URL(fileURLWithPath: resourcePath!)
            destPath = try! documentURL.appendingPathComponent(dbFilename)

            if !dirManager.fileExists(atPath: destPath.path!) {

                try dirManager.copyItem(at: srcPath, to: destPath)

            }
```

```
        } catch let err as NSError {
            print("Error: \(err.domain)")
        }

    }
```

The getDatabasePath Function

Finally, `getDatabasePath` is a helper function to get the fully qualified path for each of the database files. The function is used in the `copyDatabaseIntoDocument` function:

```
    func getDatabasePath(_ database:String)->URL{
        var dbfile:URL = URL.init(fileURLWithPath:"")
        let dirManager = FileManager.default

        do {
            let directoryURL = try dirManager.urlForDirectory(FileManager.
            SearchPathDirectory.documentDirectory, in: FileManager.SearchPathDomainMask.
            userDomainMask, appropriateFor: nil, create: true)

            dbfile = try! directoryURL.appendingPathComponent(database)

        } catch let err as NSError {
            print("Error: \(err.domain)")
        }
       return dbfile
    }
```

Back Up a Running Database

To demonstrate how to back up a running database, you will need to get a handle on the running database by using the open command and initializing the backup process using `sqlite3_backup_init`. With a running database you can run into locks, and so you will need to step through the records in the database, all the while checking to see if there is a lock on the database. SQLite provides the `sqlite3_sleep function`, which is usually set to 250 ms. With each iteration, the remaining-records count is fetched until nine are left. See here:

```
    func backupRunningDatabase(){
        var rc:Int32 = -1
        var remaining:Int32 = 0
        var page_count:Int32 = 0

  if(sqlite3_open(self.getDatabasePath(sourcedb).path!, &db)==SQLITE_OK){
            if(sqlite3_open(self.getDatabasePath(targetdb).path!, &filedb) == SQLITE_OK){
            bakdb = sqlite3_backup_init(filedb, "main", db, "main");
            remaining = sqlite3_backup_remaining(db)
```

```
        while(remaining != 0){
            rc = sqlite3_backup_step(bakdb, 10) //copy 10 pages to backup db
            if(rc == SQLITE_OK){
                remaining = sqlite3_backup_remaining(db)
                page_count = sqlite3_backup_pagecount(db)

                if( rc==SQLITE_OK || rc==SQLITE_BUSY || rc==SQLITE_LOCKED ){
                    sqlite3_sleep(250);
                }
            }

        }
         sqlite3_backup_finish(bakdb)
      }
    }
     sqlite3_close(db)
     sqlite3_close(filedb)
   }
```

Backup an In-Memory Database

Creating an in-memory backup is slightly different than backing up a running database. From the code that follows, you can see how simple it is to create an in-memory backup of your database. All you need is a pointer to your database and a pointer to the backup. Then, step through the database until all records have been backed up into the in-memory database:

```
func backupInMemory(){
    if(sqlite3_open(sourcedb, &db)==SQLITE_OK){
    if(sqlite3_open("file::memory:", &db) == SQLITE_OK){
             bakdb = sqlite3_backup_init(filedb, "main", db, "main")
            sqlite3_backup_step(bakdb, -1)
            sqlite3_backup_finish(bakdb)

            sqlite3_close(db);
            sqlite3_close(filedb)
        }
    }
}
```

Building the UI

As seen in Figure 11-7, the UI is very simple, with only two buttons, which are used to call the `backupInMemory` and `backupRunningDatabase` functions. These will be connected to the `ViewController` as `IBActions`.

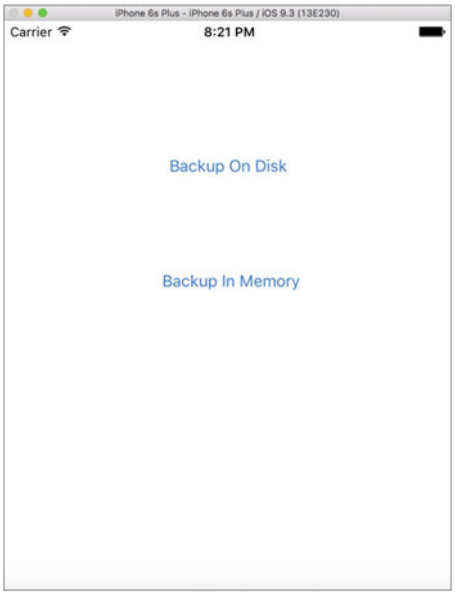

Figure 11-7. *The app's UI*

To create the connecting `IBActions`, open the Identity assistant in Xcode and control + drag a connection line from the `UIButton` to the open `ViewController` file. In Figure 11-8, you can see that releasing the mouse button triggers a popup, allowing you to enter the name of the `IBAction`, which will be `backupOnDisk`, for the corresponding `UIButton`.

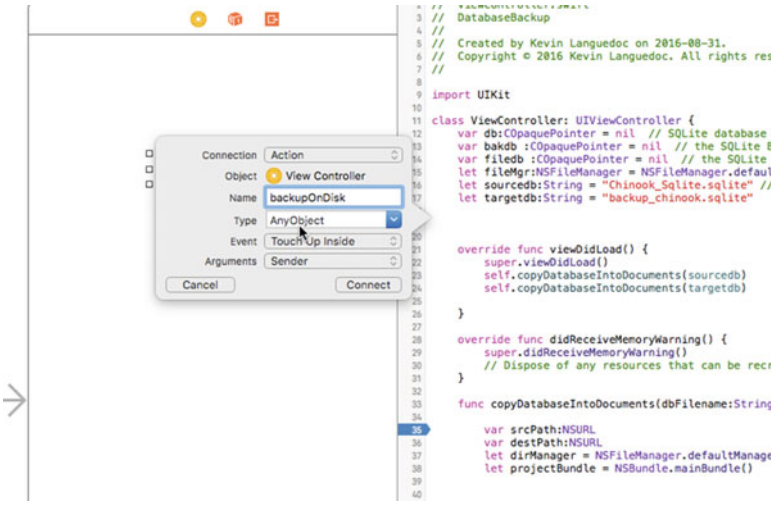

Figure 11-8. *The backupOnDisk IBAction*

For the Backup In Memory button, name the `IBAction backupInMemory`. The code for these two `IBActions` is provided following Figure 11-9.

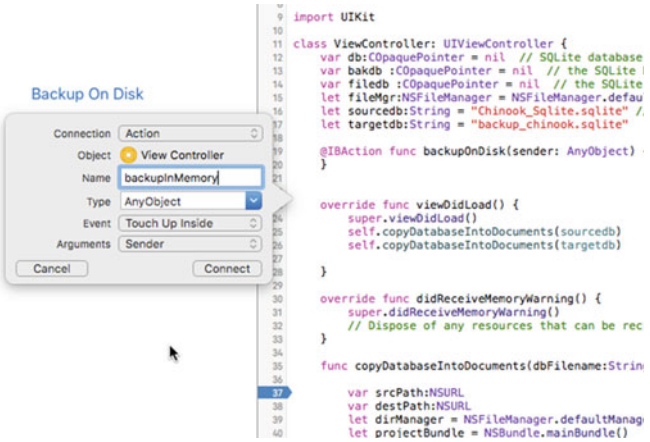

Figure 11-9. *The backupInMemory IBAction*

Finally, both `IBActions` call their respective backup functions in the `ViewController`.

```
@IBAction func backupOnDisk(_ sender: AnyObject) {
        self.backupRunningDatabase()
    }

    @IBAction func backupInMemory(_ sender: AnyObject) {
        self.backupInMemory()
    }
```

All that will remain after the backup completes is to determine where to store the backup. On a mobile device, the options are slim, so you would need to export these files to another location like iCloud or some other similar service. The files can also be moved to a corporate network.

Summary

The last chapter will cover the analysis of SQLite databases.

CHAPTER 12

Analyzing SQLite Databases

This last chapter will focus on the different tools the SQLite platform provides to help you analyze your app's databases. Except for the ANALYZE statement, which can be executed with the sqlite3_exec command, the other tools are external software programs. Specifically, in this chapter we will discuss and explore, through examples, the following technologies:

- The ANALYZE statement
- The sqldiff tool
- The sqlite3_analyzer tool

Other than the ANALYZE statement, which we will explore in Swift, all the other tools are external tools that can help you analyze your databases for support and/or for development optimization.

The Analyze Statement

The role of the ANALYZE statement is to gather information on tables and indexes in a database through statistics. SQLite accomplishes this task by creating a sqlite_stats1 table in your app's database when the ANALYZE statement is executed. SQLite passes this information to the SQLite Query Optimizer, which in turn uses the collected information to use the best query algorithm for the best performance.

If you build the database or enable SQLITE3_ENABLE_STAT3 or SQLITE3_ENABLE_STAT4, additional histogram information is gathered and stored in the sqlite3_stat3 and sqlite3_stat4 tables respectively.

To run the ANALYZE functionality in a database, you simply need to execute the ANALYZE statement, as follows:

```
ANALYZE schema
```

This will create a stat1 table for the whole database. If you only want to target a table, you could issue a command like:

```
ANALYZE schema.tablename
```

Of course, you can also build a stat1 table for an index in a table using the following command:

```
ANALYZE schema with or without the schema prefix:

ANALYZE schema.indexname
```

or,

```
ANALYZE indexname
```


K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4_12

In Swift, the ANALYZE statement can be executed like any other query, as follows:

```
func analyzeDatabase(_ name:String){

        let sql:String = "ANALYZE Chinook_Sqlite.sqlite"
        if(sqlite3_open(dbPath.path!, &db)==SQLITE_OK){
            if(sqlite3_prepare_v2(db, sql.cString(using: String.Encoding.utf8)!, -1,
            &sqlStatement, nil)==SQLITE_OK){
        while(sqlite3_step(sqlStatement)==SQLITE_ROW){
                    let output = String(cString: UnsafePointer<Int8>(sqlite3_column_
                    text(sqlStatement, 0)))
                       print(output)
                   }
            }
        }
        sqlite3_close(db)
}
```

The sqldiff Tool

The sqldiff utility is an external application from SQLite. You can download it along with SQLite3_Analyzer—and SQLite, for that matter–from the download page at www.sqlite.org. Look for the zip package for OSX x86.

You only need to unzip the compressed file to a handy directory on your OSX machine. These tools are equally available for Windows and Linux. The instructions that follow are for OSX only; however, I am sure they work the same way on Windows and Linux.

The sqldiff utility is used to compare two SQLite databases and generates a SQL script to convert the source database (database1) into the target database (database2). The sqldiff tool is easy to use. A sample output is shown in Figure 12-1. From a Terminal window, navigate to the directory where you unzipped the utility, and sqldiff will issue the following command:

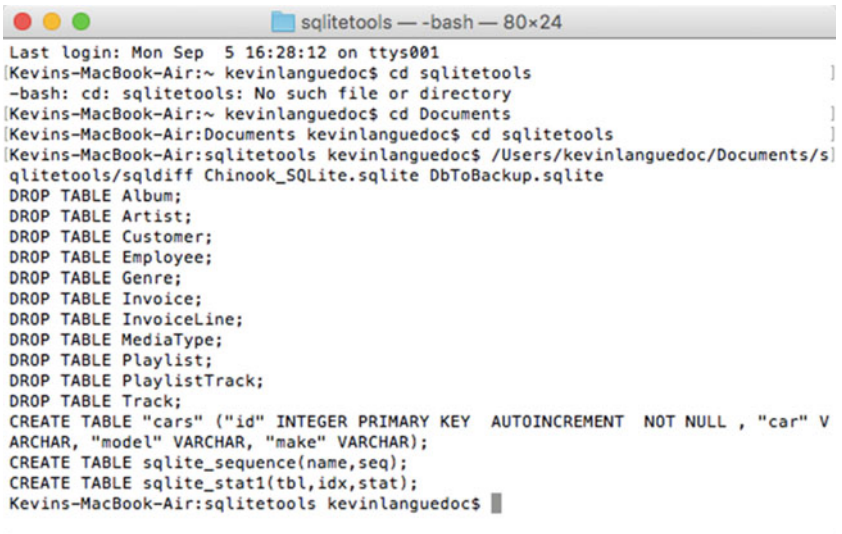

Figure 12-1. *sqldiff output to standard output*

```
sqldiff database1.sqlite databse2.sqlite
```

To illustrate, I will run the utility on the `Chinook_SQLite.sqlite` database that I used for the backups. The second database is `DbToBackup.sqlite`, which is another empty database I created for the backups.

The utility has a number of options that you can use as well. For instance:

- `--changeset FILE`
- `--lib or L`
- `--primaryKey`
- `--schema`
- `--summary`
- `--table TABLENAME`
- `--transaction`
- `--vtab`

The `-changeset` option re-directs the output to a file. The `-lib` option loads a user-defined library prior to comparing the databases, like `collating_sequences`, for instance. If you prefer using the primary key in a table instead of the rowed, then `-primaryKey` is the way to go. Using the `-schema` flag, you can show the differences in the schema only, excluding the content. To pinpoint the changes that have occurred in two tables, use `-summary`. However, the actual changes won't be displayed. Using the `-table` option allows you to compare specific table content. The `-transaction` option allows you to generate one large transaction for the complete operation. Finally, the `-vtab` choice works with virtual tables like FTS3, FTS5, and rtree tables.

The sqldiff utility works by comparing pairwise rowids, unless you are using the `-primaryKey` option. The output is generated as updates if the content is in similar tables. If the two databases have distinct tables, then the source tables are dropped (`DELETE`) and new ones created, along with using `INSERT` to insert the content.

There are, of course, some limitations with the current version of the tool. For instance, the utility only works on tables, and rowids must be accessible, unless you are using the `-primaryKey` option. Also, the content of virtual tables isn't compared unless it results in a physical table. However, using the tool in this capacity can create corrupted databases.

The sqlite3_analyzer tool

The sqlite3_analyzer is a handy tool used to measure the effective use of the space in the tables in the target database. Like the sqldiff tool before, the sqlite3_analyzer is a command-line tool that is included in the same download package as the sqldiff tool.

The utility generates a text (ASCII)-based report in a text file. It is a human-readable file. To demonstrate, I will run the tool with the `Chinook_Sqlite.sqlite` database file from before.

From a Terminal window, navigate to the directory where the sqlite3_analyzer tool is situated and issue the following command:

```
sqlite3_analyzer database.sqlite
```

For example, here is the output of the `DbToBackup.sqlite` database:

```
Last login: Mon Sep  5 17:04:01 on ttys001
Kevins-MacBook-Air:~ kevinlanguedoc$ /Users/kevinlanguedoc/Documents/sqlitetools/sqlite3_
analyzer /Users/kevinlanguedoc/Documents/sqlitetools/DbToBackup.sqlite
/** Disk-Space Utilization Report For /Users/kevinlanguedoc/Documents/sqlitetools/
DbToBackup.sqlite
```

```
Page size in bytes.............................. 32768
Pages in the whole file (measured)............... 4
Pages in the whole file (calculated)............. 4
Pages that store data............................ 4          100.0%
Pages on the freelist (per header)............... 0            0.0%
Pages on the freelist (calculated)............... 0            0.0%
Pages of auto-vacuum overhead.................... 0            0.0%
Number of tables in the database................. 4
Number of indices................................ 0
Number of defined indices........................ 0
Number of implied indices........................ 0
Size of the file in bytes........................ 131072
Bytes of user payload stored..................... 0            0.0%

*** Page counts for all tables with their indices ****************************

CARS............................................ 1            25.0%
SQLITE_MASTER................................... 1            25.0%
SQLITE_SEQUENCE................................. 1            25.0%
SQLITE_STAT1.................................... 1            25.0%

*** Page counts for all tables and indices separately ************************

CARS............................................ 1            25.0%
SQLITE_MASTER................................... 1            25.0%
SQLITE_SEQUENCE................................. 1            25.0%
SQLITE_STAT1.................................... 1            25.0%

*** All tables ***************************************************************

Percentage of total database..................... 100.0%
Number of entries................................ 3
Bytes of storage consumed........................ 131072
Bytes of payload................................. 296          0.23%
Average payload per entry........................ 98.67
Average unused bytes per entry................... 43543.67
Maximum payload per entry........................ 141
Entries that use overflow........................ 0            0.0%
Primary pages used............................... 4
Overflow pages used.............................. 0
Total pages used................................. 4
Unused bytes on primary pages.................... 130631       99.66%
Unused bytes on overflow pages................... 0
Unused bytes on all pages........................ 130631       99.66%

*** Table CARS ***************************************************************

Percentage of total database..................... 25.0%
Number of entries................................ 0
Bytes of storage consumed........................ 32768
Bytes of payload................................. 0            0.0%
B-tree depth..................................... 1
```

```
Average payload per entry..... 0.0
Average unused bytes per entry..... 0.0
Maximum payload per entry..... 0
Entries that use overflow..... 0
Primary pages used..... 1
Overflow pages used..... 0
Total pages used..... 1
Unused bytes on primary pages..... 32760       99.976%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 32760       99.976%

*** Table SQLITE_MASTER ***************************************************

Percentage of total database..... 25.0%
Number of entries..... 3
Bytes of storage consumed..... 32768
Bytes of payload..... 296          0.90%
B-tree depth..... 1
Average payload per entry..... 98.67
Average unused bytes per entry..... 10783.67
Maximum payload per entry..... 141
Entries that use overflow..... 0            0.0%
Primary pages used..... 1
Overflow pages used..... 0
Total pages used..... 1
Unused bytes on primary pages..... 32351       98.7%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 32351       98.7%

*** Table SQLITE_SEQUENCE *************************************************

Percentage of total database..... 25.0%
Number of entries..... 0
Bytes of storage consumed..... 32768
Bytes of payload..... 0            0.0%
B-tree depth..... 1
Average payload per entry..... 0.0
Average unused bytes per entry..... 0.0
Maximum payload per entry..... 0
Entries that use overflow..... 0
Primary pages used..... 1
Overflow pages used..... 0
Total pages used..... 1
Unused bytes on primary pages..... 32760       99.976%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 32760       99.976%

*** Table SQLITE_STAT1 ****************************************************

Percentage of total database..... 25.0%
Number of entries..... 0
Bytes of storage consumed..... 32768
```

```
Bytes of payload..................................... 0           0.0%
B-tree depth......................................... 1
Average payload per entry............................ 0.0
Average unused bytes per entry....................... 0.0
Maximum payload per entry............................ 0
Entries that use overflow............................ 0
Primary pages used................................... 1
Overflow pages used.................................. 0
Total pages used..................................... 1
Unused bytes on primary pages........................ 32760       99.976%
Unused bytes on overflow pages....................... 0
Unused bytes on all pages............................ 32760       99.976%

*** Definitions **************************************************************

Page size in bytes

    The number of bytes in a single page of the database file.
    Usually 1024.

Number of pages in the whole file

    The number of 32768-byte pages that go into forming the complete
    database

Pages that store data

    The number of pages that store data, either as primary B*Tree pages or
    as overflow pages. The number at the right is the data pages divided by
    the total number of pages in the file.

Pages on the freelist

    The number of pages that are not currently in use but are reserved for
    future use. The percentage at the right is the number of freelist pages
    divided by the total number of pages in the file.

Pages of auto-vacuum overhead

    The number of pages that store data used by the database to facilitate
    auto-vacuum. This is zero for databases that do not support auto-vacuum.

Number of tables in the database

    The number of tables in the database, including the SQLITE_MASTER table
    used to store schema information.

Number of indices

    The total number of indices in the database.

Number of defined indices

    The number of indices created using an explicit CREATE INDEX statement.
```

```
Number of implied indices
    The number of indices used to implement PRIMARY KEY or UNIQUE constraints
    on tables.

Size of the file in bytes

    The total amount of disk space used by the entire database files.

Bytes of user payload stored

    The total number of bytes of user payload stored in the database. The
    schema information in the SQLITE_MASTER table is not counted when
    computing this number. The percentage at the right shows the payload
    divided by the total file size.

Percentage of total database

    The amount of the complete database file that is devoted to storing
    information described by this category.

Number of entries

    The total number of B-Tree key/value pairs stored under this category.

Bytes of storage consumed

    The total amount of disk space required to store all B-Tree entries
    under this category. This is the total number of pages used times
    the pages size.

Bytes of payload

    The amount of payload stored under this category. Payload is the data
    part of table entries and the key part of index entries. The percentage
    at the right is the bytes of payload divided by the bytes of storage
    consumed.

Average payload per entry

    The average amount of payload on each entry. This is just the bytes of
    payload divided by the number of entries.

Average unused bytes per entry

    The average amount of free space remaining on all pages under this
    category on a per-entry basis. This is the number of unused bytes on
    all pages divided by the number of entries.

Non-sequential pages

    The number of pages in the table or index that are out of sequence.
    Many filesystems are optimized for sequential file access, so a small
    number of non-sequential pages might result in faster queries,
```

```
    especially for larger database files that do not fit in the disk cache.
    Note that after running VACUUM, the root page of each table or index is
    at the beginning of the database file and all other pages are in a
    separate part of the database file, resulting in a single non-
    sequential page.

Maximum payload per entry

    The largest payload size of any entry.

Entries that use overflow

    The number of entries that use one or more overflow pages.

Total pages used

    This is the number of pages used to hold all information in the current
    category. This is the sum of index, primary, and overflow pages.

Index pages used

    This is the number of pages in a table B-tree that hold only key (rowid)
    information and no data.

Primary pages used

    This is the number of B-tree pages that hold both key information and data.

Overflow pages used

    The total number of overflow pages used for this category.

Unused bytes on index pages

    The total number of bytes of unused space on all index pages. The
    percentage at the right is the number of unused bytes divided by the
    total number of bytes on index pages.

Unused bytes on primary pages

    The total number of bytes of unused space on all primary pages. The
    percentage at the right is the number of unused bytes divided by the
    total number of bytes on primary pages.

Unused bytes on overflow pages

    The total number of bytes of unused space on all overflow pages. The
    percentage at the right is the number of unused bytes divided by the
    total number of bytes on overflow pages.

Unused bytes on all pages
```

```
    The total number of bytes of unused space on all primary and overflow
    pages. The percentage at the right is the number of unused bytes

    divided by the total number of bytes.
*******************************************************************************
The entire text of this report can be sourced into any SQL database
engine for further analysis. All of the text above is an SQL comment.
The data used to generate this report follows:
*/
BEGIN;
CREATE TABLE space_used(
   name clob,        -- Name of a table or index in the database file
   tblname clob,     -- Name of associated table
   is_index boolean, -- TRUE if it is an index, false for a table
   is_without_rowid boolean, -- TRUE if WITHOUT ROWID table
   nentry int,       -- Number of entries in the BTree
   leaf_entries int, -- Number of leaf entries
   depth int,        -- Depth of the b-tree
   payload int,      -- Total amount of data stored in this table or index
   ovfl_payload int, -- Total amount of data stored on overflow pages
   ovfl_cnt int,     -- Number of entries that use overflow
   mx_payload int,   -- Maximum payload size
   int_pages int,    -- Number of interior pages used
   leaf_pages int,   -- Number of leaf pages used
   ovfl_pages int,   -- Number of overflow pages used
   int_unused int,   -- Number of unused bytes on interior pages
   leaf_unused int,  -- Number of unused bytes on primary pages
   ovfl_unused int,  -- Number of unused bytes on overflow pages
   gap_cnt int,      -- Number of gaps in the page layout
   compressed_size int  -- Total bytes stored on disk
);
INSERT INTO space_used VALUES('sqlite_master','sqlite_
master',0,0,3,3,1,296,0,0,141,0,1,0,0,32351,0,0,32768);
INSERT INTO space_used VALUES('sqlite_stat1','sqlite_stat1',0,0,0,0,1,0,0,0,0,0,1,0,0,32760
,0,0,32768);
INSERT INTO space_used VALUES('cars','cars',0,0,0,0,1,0,0,0,0,0,1,0,0,32760,0,0,32768);
INSERT INTO space_used VALUES('sqlite_sequence','sqlite_
sequence',0,0,0,0,1,0,0,0,0,0,1,0,0,32760,0,0,32768);
COMMIT;
```

This analysis report provides very detailed information on the use of space in the pages that are in the database. What follows is the extract on the Cars table. The primary concern to me as a database developer is the amount of free space in the database's table. In this case, it is at 99.976 %, which is fantastic. This means that there isn't any bloat in the pages.

A page is a unit of storage in a database—any database, really. In this table, only one page is used. However, that database is free to use as many pages as is needed to efficiently manage itself. When there is a lot of activity from inserts, deletes, and updates, the database can be become bloated with unused space in the pages.

In order to regain that space and optimize the database, you can execute the `VACUUM` command on the database, as we have discussed before, which removes empty space. You can also re-index the tables to optimize table I/O efficiency.

```
*** Table CARS ***************************************************************

Percentage of total database..... 25.0%
Number of entries..............................  0
Bytes of storage consumed..... 32768
Bytes of payload..............................  0            0.0%
B-tree depth..................................  1
Average payload per entry..... 0.0
Average unused bytes per entry..... 0.0
Maximum payload per entry..... 0
Entries that use overflow..... 0
Primary pages used.............................  1
Overflow pages used............................  0
Total pages used...............................  1
Unused bytes on primary pages..... 32760       99.976%
Unused bytes on overflow pages..... 0
Unused bytes on all pages..... 32760       99.976%
```

Summary

This chapter is different from the other chapters in that the focus was primarily outside of Xcode and Swift. Most of these tools that we looked at are run in the Terminal. The sqldiff lets us compare two databases and copy the schema and content from one to another. The sqlite3_analyzer tool generates a report on how the space is used in the pages in the database. We also looked at the `ANALYZE` statement, which creates a `stat1` table to store statistics on the tables, which are then used in the SQLite Query Optimizer to select the best algorithm to use to perform various queries against a database.

I hope this book serves you well.

Index

A

ANALYZE statement, 187–188
attachDatabase function, 167
ATTACH DATABASE statement, 161–162
ATTACH SQLite functionality, 163–170

B

Backup API
    BackupBridge file, 179
    bridge interface, 178–179
    FileManager, 175–176
    in-memory databases, 176
    on-disk operation, 177
    sqlite library, 179–180
    Swift Compiler Build settings, 179–180
    UI building, 183–185
    View Application project, 177–178
    ViewController, 181–183

C

cellForRowAt, 37
Chinook_SQLite.sqlite database, 189
Collation sequences, 50
    BINARY collation, 51
    NoCase collation, 51
    RTRIM type, 51
    sqlite3_create_collation function, 51
Controllers, 77
    buildSchema, 79
    FirstViewController, 81
    init function, 78
    insertWineRecord function, 80
    insertWineryRecord function, 80
    photo capture function, 82
        imagePickerController function, 82
        insertRecordAction, 82
        UISlider, 83
        viewDidLoad function, 82
        wineRating function, 83
    SecondViewController, 83
    WineryDAO class, 77
copyDatabaseIntoDocuments function, 181
createDatabase function, 166
createOrOpenDatabase function, 79
CROSS JOIN, 163

D

Data model, 75
    database schema, 77
    wineries database, 76
        documentDirectory, 76
        SearchPathDirectory, 76
        types, 76
DbMgrDAO controller, 30
    contentsOfDirectoryAtURL method, 34
    COpaquePointer, 31
    DetailViewController, 32
    executeQuery function, 34
    initViewIndex function, 33
    NSFileManager class, 34
    NSFileManager.defaultManage classr, 34
    openSQliteDatabase function, 35
    populateIndexView function, 31
    preparedStatement, 32
    sqlite3_column_text method, 32
    stringByReplacingOccurencesOfString, 35
    TableView, 32
    viewDidLoad function, 33
Db Swift class, 27
    CREATE TABLE statement, 28
    Enums class, 30
    executeQuery method, 29
    insert statement, 28
    selectDbSchemaListByType method, 29
    selectDbSchemaStructure method, 29
    tableDef, 28
db_to_attach_path variable, 166
DELETE function
    collation sequences, 53
    DELETE statement, 131

K. Languedoc, *Build iOS Database Apps with Swift and SQLite*, DOI 10.1007/978-1-4842-2232-4

DELETE function (*cont.*)
indices, 53
LIMITS clause, 132
sampleDelete function, 132–134
tables, 52
TRIGGER, 53, 132
views, 52
WHERE clause, 131
Winery app
Cellar TableView, 138–139
commitEditingStyle, 139
deleteRowsAtIndexPaths, 139
deleteWineRecord, 139–140
Edit mode, 141–142
UITableView, 139
UITableViewCellEditingStyle, 139
ViewControllers, 136
viewDidLoad method, 137
Wineries TableView, 140–144
WineryDAO Class, 135–136
WineryListTableController, 136–137
dequeueReusableCellWithIdentifier, 152
detachDatabase function, 168
DETACH SQLite functionality, 163–170
DETACH statement, 162
DetailViewController, 39
createDbButton, 39
dbStatusMsg, 41
detailSQLiteQueryField method, 39–40
@IBAction Buttons, 40
openSQLiteDatabase method, 39
prepareForSeque, 40
viewDidLoad function, 41
viewWillAppear function, 41

E

executeQuery function, 34

F

FileManager, 175–176

G, H

getDatabasePath function, 182

I, J, K

IBAction, 164, 184
IBOutlets, 157, 164
Index, 48
DROP INDEX, 49
UNIQUE INDEX, 49
INNER JOIN, 163
INSERT statement, 64
ABORT clause, 67
blobs, 68
data-binding functions, 63
DocumentDirectory, 65
IGNORE clause, 67
last clause, 67
NSFileManager, 65
NSSearchPathDirectory, 65
NSSearchPathForDirectoriesInDomains, 65
records, 84–85
REPLACE statements, 66
rollback, 67
iOS app
Create Bridge interface, 150
database creation, 146–147
project creation, 148–149
SQLiteSearchBridge, 149–150
UISearchBar, 146
ViewController, 150–159
project creation, 22
AppDelegate application, 23
bundle, 23
database, 22
didFinishLaunchWithOption, 23
documents directory, 23
finder, 23
Master-Detail template, 22
resources, 22

L

LIMITS clause, 132

M

MasterViewController, 36
cellForRowAtIndexPath, 37
numberOfRowsInSection, 37
schemaDetailsItems, 36
segue, 36
titleForHeaderInSection, 37
UITableView component, 36
unWindFromSegue function, 38
viewDidLoad function, 36

N

NSObject + SQLiteSearchBridge, 150
numberOfRowsInSection, 37, 151

O

OUTER JOIN, 163

P, Q

PRAGMA database, 172–173
Pragma statements, 59
    automatic_index, 60
    auto-vacuum, 60
    busy timeout, 60
    foreign_key_check, 59
    Foreign_key_list, 60
    integrity_check, 60
    shrink_memory, 60
Prototype, 158

R

REPLACE statements, 66
RTRIM type, 51

S

searchBarCancelButtonClicked method, 152
searchBarSearchButtonClicked
        method, 152
searchDatabase function, 153–154
searchResultCellTableViewCell, 154
SELECT statement, 89
    cross join, 94
    data-type binding functions, 89
    display images, 94
    IBOutlets, 106
    inner join, 93
    list of records, 112
    MPMoviePlayerController, 96
    navigation controllers, 102
    outer join, 94
    playback audio records, 95
    selectWineList function, 99
    selectWineriesList function, 98
    SelectWineries UIPicker, 96
    selectWineryByName function, 99
    SQLITE_ROW constant, 90
    TableViewControllers, 102
    UIPickerView function, 97
    UITableViewController, 100
    using sub query, 93
    ViewController function, 96
    WHERE clause, 90, 92
    WineListTableViewController, 104
    Wineries UIPickerView, 111
    WineryListTableViewController, 107–108
setDbPath function, 166
sqldiff, 188–189
SQLite functions
    corrupt, 61
    JSON functions, 53, 55
    limits, 61
    sqlite3_analyzer, 189–196
    sqlite3_enable_load_extension, 54
    sqlite3_exec function, 171
    sqlite3_load_extension, 54
    using Swift
        build phases, 56
        compile sources, 56
        header File, 57
        sizeconverter, 57
        sqlite3_user_data() function, 58
        values, 55
SQLite library, 1
SQLITE_LIMIT_ATTACHED, 162
SQLite Manager, 1, 147
SQLite Manager, firefox, 11
    Db Settings, 13
    directory selector, 13
    index, 11, 18
        definition, 19
        sqlite_autoindex_book, 18
    menu option, 12
    Sqlite_master, 14
        browse & search, 14
        schema, 14
    sqlite_sequence, 14
        ATTACH command, 14
        AUTOINCREMENT property, 14
        DETACH command, 14
        ROWINDEX, 14
    tables and columns, 11, 15
        definition, 16
        dropping, 17
        INSERT/UPDATE, 16
        primary key, 16
        re-indexing, 17
        table name, 17
    terminal, 11
    tools menu, 12
    trigger, 11, 19, 21
    view, 11, 19
        CREATE VIEW, 20
        select statement, 20
    windows command, 11
SQLiteSearch-Bridging-Header, 150
SQLITE_TRANSIENT, 151
Storyboard, 154
Swift iOS application, 2
    add SQLite 3 library, 3
    bridge, 5
        header file, 5
        Swift compiler, 7
    DetailViewController, 3
    MasterViewController, 3
    wrapper functions, 7

Swift iOS application (*cont.*)
DbMgrDAO class, 8
executeQuery function, 10
init function, 9

T

Tables, modification
add columns, 47
ALTER command, 46–47
DROP command, 47
foreign keys, 46
REINDEX command, 46, 48
RENAME command, 46
UPDATE command, 47
Triggers, 49

U

UIButtons, 25
Assistant Editor, 26
Attributes inspector, 26
detailSQLiteQueryField, 25, 27
@IBActions, 26
Interface Builder, 25
repeat process, 26
UIPickerView, 168
UITableViewCell, 106, 150
umberOfSectionsInTableView, 151
UPDATE statement
ABORT option, 116
CityTemperature table, 114
FAIL option, 117
FirstViewController, 125–126
IBOutlets, 114
IGNORE option, 117
JOIN, 116
Lamartine wine, 126–127, 129
ON CONFLICT, 116
OR clauses, 114
Pl/SQL/T-SQL, 114
REPLACE option, 117
ROLLBACK, 116
SQLITE_DONE, 129–130
SQLITE_OK, 127–128
sub-query, 115
Swift String constant, 114
UI fields, 114
updateRecords method, 117
WHERE clause, 113, 115–116
wineries, 125
wines list, 128
wineyDAO class, 118
columns updation, 118–119
segue.destinationViewController, 123–124
showWineDetail segue, 120
WineListTableViewController, 121–122
WineryListTableViewController, 122–123

V

viewDidLoad function, 151, 165, 181
Views, modification, 48

W, X, Y, Z

Winery application, 69
add bridge, 69
constraints, 74
creation, 69
data model, 75
imageView, 75
UI components, 73
view controller, 72
Winery database
CRUD operations, 41
createDbButton function, 42
detailSQLiteQueryField, 42
openSQLiteDatabase, 42
table creation, 42
view creation, 44