LEARNING CORE DATA FOR iOS WITH SWIFT

A Hands-On Guide to Building Core Data Applications

TIM ROADLEY
Learning Core Data for iOS with Swift
The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you’ve learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Visit informit.com/learningseries for a complete list of available publications.
Learning Core Data for iOS with Swift

Tim Roadley
The more I learn, the more I learn I need to learn more.

I dedicate this book to my wonderful wife, Tracey, who has given up many nights and weekends with me to help make this book a reality. Finally, we can sit back and relax together again! I’m sure Tyler & Taliah will let us....
Contents at a Glance

Preface xv

1 Your First Core Data Application 1
2 Managed Object Model Basics 17
3 Managed Object Model Migration 43
4 Managed Object Model Expansion 73
5 Table Views 91
6 Views 121
7 Picker Views 151
8 Preloading Data 177
9 Deep Copy 201
10 Performance 221
11 Background Processing 249
12 Search 265
13 iCloud 271
14 Taming iCloud 283
15 CloudKit Sync: Uploading Objects 307
16 CloudKit Sync: Downloading Changes and Handling Deletions 337

A Preparing the Groceries Application 369
B Finalizing the Groceries Application 377
Table of Contents

Preface xv

1 Your First Core Data Application 1
   What Is Core Data? 1
      Persistent Store Coordinator 2
      Managed Object Model 4
      Managed Object Context 4
   When to Use Core Data 5
   Introducing the Groceries Application 5
   Adding Core Data to an Existing Application 6
      Implementing the Core Data Helper 7
      Adding Paths 10
      Adding the Core Data Stack 10
      Adding the Setup Section 12
      Adding the Saving Section 13
   Adding a Managed Object Model File 14
   Summary 15
   Exercises 15

2 Managed Object Model Basics 17
   What Is a Managed Object Model? 17
   Entities 18
   Attributes 20
      Integer 16/32/64 21
   Float and Double 22
      Decimal 23
      String 23
      Boolean 23
      Date 23
      Binary Data 23
      Transformable 24
   Attribute Settings 25
   Subclassing NSManagedObject 27
   Scalar Properties for Primitive Data Types 28
   Introducing the Demo Function 29
<table>
<thead>
<tr>
<th>#</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Managed Object Model Migration</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Changing a Managed Object Model</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Adding a Model Version</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Lightweight Migration</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>Default Migration</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Migration Manager</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Introducing MigrationVC.swift (Migration View</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Controller Code)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Introducing CDMigration.swift (Core Data Migration Code)</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>71</td>
</tr>
<tr>
<td>4</td>
<td>Managed Object Model Expansion</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Relationships</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Delete Rules</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>Introducing CDOperation.swift (Core Data Operation)</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>Delete Rule Testing</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Entity Inheritance</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>89</td>
</tr>
<tr>
<td>5</td>
<td>Table Views</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Table View Basics</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Core Data Table Views</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>Introducing CDTableViewController</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>Expanding CDTableViewController</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>DATASOURCE: UITableViewController</td>
<td>98</td>
</tr>
</tbody>
</table>
7 Picker Views 151
   Picker View Basics 151
   Introducing CDPickerTextField 152
   Introducing UnitPickerTF 157
      Creating the Unit Picker 159
      Connecting the Unit Picker 160
      Configuring ItemVC for the Unit Picker 161
   Introducing LocationAtHomePickerTF 165
   Introducing LocationAtShopPickerTF 167
   Using the Location Pickers 168
      Connecting the Location Pickers 169
      Configuring ItemVC for the Location Pickers 170
   Picker-Avoiding Text Field 173
   Summary 176
   Exercises 176

8 Preloading Data 177
   Including Default Data 177
   Indicating Whether an Import Is Required 178
   Importing from XML 182
   Creating an Import Context 184
   Preventing Duplicate Default Data 185
   Triggering a Default Data Import 186
   Finding or Creating Managed Objects 188
   Mapping XML Data to Entity Attributes 192
   Importing from a Persistent Store 195
      Using the Default Data Store as the Initial Store 196
   Summary 199
   Exercises 200

9 Deep Copy 201
   The Deep Copy Process 201
   Configuring a Source Stack 204
      Configuring the Source Coordinator 204
      Configuring the Source Context 205
      Configuring the Source Store 205
Contents

Enhancing CDImporter 206
    Identifying Unique Attributes 207
    Object Info 207
    Copying a Unique Object 208
    Establishing a To-One Relationship 210
    Establishing a To-Many Relationship 211
    Establishing an Ordered To-Many Relationship 212
    Copying Relationships 213
    Deep Copy Entities 215
    Triggering a Deep Copy 216
Summary 219
Exercises 219

10 Performance 221
    Identifying Performance Issues 221
    Implementing the Camera 222
        Implementing the Image Picker Controller Delegates 224
    Generating Test Data 228
        Merge Policies 230
    Measuring Performance with SQLDebug 232
    Measuring Performance with Instruments 233
    Improving Performance 236
        Model Optimization 238
            Handling Large Objects 239
    Cleaning Up 246
Summary 246
Exercises 247

11 Background Processing 249
    Implementing Background Save 249
    Configuring an Import Context Parent 253
    Faulting Objects 254
    Generating Thumbnails 257
Summary 261
Exercises 261
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Search</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td>Updating CDTableViewCell</td>
<td>266</td>
</tr>
<tr>
<td></td>
<td>Updating PrepareTVC</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>Search Optimization</td>
<td>269</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>270</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>270</td>
</tr>
<tr>
<td>13</td>
<td>iCloud</td>
<td>271</td>
</tr>
<tr>
<td></td>
<td>iCloud Basics</td>
<td>271</td>
</tr>
<tr>
<td></td>
<td>Enabling iCloud</td>
<td>273</td>
</tr>
<tr>
<td></td>
<td>Updating CDHelper for iCloud</td>
<td>274</td>
</tr>
<tr>
<td></td>
<td>Adding an iCloud Store</td>
<td>275</td>
</tr>
<tr>
<td></td>
<td>Handling iCloud Notifications</td>
<td>277</td>
</tr>
<tr>
<td></td>
<td>The Debug Navigator</td>
<td>280</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>281</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>282</td>
</tr>
<tr>
<td>14</td>
<td>Taming iCloud</td>
<td>283</td>
</tr>
<tr>
<td></td>
<td>De-Duplication</td>
<td>283</td>
</tr>
<tr>
<td></td>
<td>Identifying Duplicates</td>
<td>284</td>
</tr>
<tr>
<td></td>
<td>Deleting Duplicates</td>
<td>286</td>
</tr>
<tr>
<td></td>
<td>Triggering De-Duplication</td>
<td>291</td>
</tr>
<tr>
<td></td>
<td>Seeding</td>
<td>292</td>
</tr>
<tr>
<td></td>
<td>Preparing Seed Variables</td>
<td>293</td>
</tr>
<tr>
<td></td>
<td>Adding Seed Helper Functions</td>
<td>295</td>
</tr>
<tr>
<td></td>
<td>Developing with a Clean Slate</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>Configurations</td>
<td>302</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>303</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>303</td>
</tr>
<tr>
<td>15</td>
<td>CloudKit Sync: Uploading Objects</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>Introducing CloudKit</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>CloudKit Database Synchronization Limitations</td>
<td>309</td>
</tr>
<tr>
<td></td>
<td>Introducing CDCloudSync</td>
<td>309</td>
</tr>
<tr>
<td></td>
<td>Public Data Groups</td>
<td>311</td>
</tr>
<tr>
<td></td>
<td>Cache Status</td>
<td>316</td>
</tr>
<tr>
<td></td>
<td>CloudKit Building Blocks</td>
<td>320</td>
</tr>
</tbody>
</table>
Uploading New Records and Relationships 325
Adding Synchronization Logic 327
Preparing the Managed Object Model for Sync 331
Summary 335
Exercises 335

16 CloudKit Sync: Downloading Changes and Handling Deletions 337
CloudKit Building Blocks (Continued) 337
Change Synchronization 342
Deletion Synchronization 350
Quality Assurance 354
Testing the Network 357
Updating Synchronization Logic 359
Adding CDCloudSync to Your Own Applications 361
Summary 367
Exercises 367
Thank You! 367

A Preparing the Groceries Application 369
New Xcode Project 369
Storyboard Design 370
App Icons and Launch Images 373
Introducing GenericVC 374

B Finalizing the Groceries Application 377
New Features 377
Photo Library and Photo Deletion Support 378
Favorites 380
Icon Badge 384
Location De-Duplication Logic 385
Acknowledgments

A resounding thank-you first goes out to Trina MacDonald for giving me the opportunity to write this book originally and again for Swift. Her guidance throughout the whole process has been invaluable, as has the assistance of Michael Thurston and the technical reviews from Ash Furrow, Carl Brown, and Niklas Saers. You guys saved this book from a few bugs that crept through on those late nights and also provided some great insight and coding technique suggestions with Swift.

About the Author

Tim Roadley is a family man with strong technical focus in career and personal goals. From managing IT infrastructure to developing apps and writing books, if it’s complex, he’s in his element. He currently works for TKH Group, primarily focused on system integration software used by large clients such as Westfield. Prior roles include implementations of various IT systems at government departments and major banks, most notably the implementation of a payments switch that drives the RediATM network in Australia. In his downtime, he enjoys spending time with his wonderful wife, Tracey (when he can pull her away from her Thermomix), and two lovely children, Tyler and Taliah.
Every day, millions of Apple devices run applications, or apps, that rely on Core Data. This has led to a mature, stable, and incredibly fast platform for apps to access their data. Core Data itself is not a database. In fact, Core Data is a framework that, among other things, automates how you interact with a database. Instead of writing SQL code, you work with managed objects. All the associated SQL you would otherwise have to write yourself is generated automatically. This leaves you with all the benefits of a relational database without the headache of writing, testing, and optimizing SQL queries within your code. The SQL code generated automatically “under the hood” is the product of years of refinement and optimization by Apple’s masterful engineers. Using Core Data not only speeds up your own application development time, it also significantly reduces the amount of code you have to write.

Here are some notable features of Core Data:

- Change management (undo and redo)
- Relationships
- Data model versioning and migration
- Efficient fetching (through batching and faulting)
- Efficient filtering (through predicates)
- Data consistency and validation

With this book, you are introduced to Core Data features and best practices. As you progress through the chapters, you also build a fully functional Core Data iPhone app from scratch. Each key piece of information is explained in succinct detail so you can apply what you’ve learned straight away. The sample application built throughout this book has been especially designed to demonstrate as many aspects of Core Data as possible. At the same time it is a completely real-world application available on the App Store today. This should make it easier to absorb concepts as you relate them to real-life scenarios.

The arrival of iOS 9 offers major improvements in the speed, reliability, and simplicity of Core Data integration with iCloud and introduction of CloudKit. I encourage anyone who has previously given up on this technology to give it another go, because you will be pleasantly surprised.

If you have feedback, bug fixes, corrections, or anything else you would like to contribute to a future edition, please contact me at timroadley@icloud.com. Finally, thank you for taking an interest in this book. If you like it, please tweet or post to Facebook about it! I have put a lot of
effort into meticulously crafting it, so I truly hope it helps you on your way to mastering this brilliant technology.

—Tim Roadley (@TimRoadley), September 2015

Who Is This Book For?

This book is aimed at Swift programmers who want to learn how to efficiently manage data in their iOS apps. Prior experience with databases may help you pick up some topics faster, yet is not essential knowledge. As old habits die hard, some SQL programmers may find it more difficult to wrap their heads around some topics.

Beyond the basics, the closing chapters explain how to build a CloudKit solution for sharing data with multiple users. This is something previously impossible with old iCloud technology. Whatever your scenario, don’t worry. Every step is explained and demonstrated clearly.

What You’ll Need

As a Swift programmer, it is expected that you already have a reasonably modern Mac or MacBook running Xcode 7 or above. You should also be familiar with Xcode and have an iOS device to test with. This is particularly true once you reach Chapter 10, “Performance,” which is all about device performance.

Swift is new to everyone, so it’s likely you’ll encounter code and techniques you haven’t seen before. Fortunately, you’ll be guided step-by-step through the entire process of creating a real-world Swift Core Data application for iOS, so your experience with Swift will grow throughout.

The following resources are suggested as an accompaniment to this book:

- Swift, by Apple (search iBooks Store within the iBooks App)
- Learning Swift Programming, by Jacob Schatz (search amazon.com)

How This Book Is Organized

This book takes you through the entire process of building the Groceries application, which is available from the App Store today. The Groceries implementation throughout the book demonstrates Core Data integration with iCloud and CloudKit. Each chapter in this book builds on the last, so you’re introduced to topics in the order you need to implement them. Along the way you build helper classes that simplify redeployment of what you’ve learned into your own applications. In fact, the exercises at the end of the iCloud and CloudKit chapters guide you through this redeployment of the helper classes into a brand new app. Using this as an example you can integrate Core Data, iCloud, and CloudKit into your own applications in no time!
Here's a brief summary of what you find in each chapter:

- **Chapter 1, “Your First Core Data Application”**—The groundwork is laid as the fundamental concepts of Core Data are introduced. You learn what Core Data is, and just as importantly, what it isn’t. In addition, Core Data integration with an existing application is demonstrated as the CDHelper class is implemented.

- **Chapter 2, “Managed Object Model Basics”**—Data models are introduced as parallels are drawn between traditional database schema design and Core Data. You learn how to configure a basic managed object model as entities and attributes are discussed, along with accompanying advice on choosing the right data types. Inserting, fetching, filtering, sorting, and deleting managed objects are also covered and followed up with an introduction to fetch request templates.

- **Chapter 3, “Managed Object Model Migration”**—Experience lightweight migration, default migration, and using a migration manager to display migration progress. You learn how to make an informed decision when deciding between migration options for your own applications and become comfortable with the model-versioning capabilities of Core Data.

- **Chapter 4, “Managed Object Model Expansion”**—The true power of a relational data model is unlocked as different types of relationships are explained and added to Groceries. Other model features such as abstract and parent entities are also covered, along with techniques for dealing with data validation errors.

- **Chapter 5, “Table Views”**—The application really comes to life as Core Data is used to drive memory-efficient and highly performing table views with a fetched results controller. Of course, most of the generic legwork is put into a reusable table view controller subclass called CDTableViewController. By dropping this class into your own applications, you can easily deploy Core Data–driven table views yourself.

- **Chapter 6, “Views”**—Working with managed objects takes a front seat as you’re shown how to pass them around the application. Objects selected on a table view are passed to a second view, ready for editing. The editing interface is added to Groceries, demonstrating how to work with objects and then save them back to the persistent store.

- **Chapter 7, “Picker Views”**—As a nice touch, Core Data–driven picker views are added to the editing views. Picker views allow the user to quickly assign existing items to a unit of measurement, home location, or shop location. A special reusable text field subclass called CDPickerTextField is introduced, which replaces the keyboard with a Core Data picker view whenever an associated text field is tapped.

- **Chapter 8, “Preloading Data”**—Techniques for generating a persistent store full of default data from XML are explained and demonstrated in this chapter as the generic CDImporter helper class is introduced. Once you have a persistent store to include with a shipping application, you then see how to determine whether a default data import is required or even desired by the user.
• Chapter 9, “Deep Copy”—A highly flexible and fine-grained alternative to migratePersistentStore, deep copy enables you to copy objects and relationships from selected entities between persistent stores. In this chapter, the CDImporter helper class is enhanced with the deep copy capability.

• Chapter 10, “Performance”—Gain experience with Instruments as you identify and eliminate performance issues caused by the common pitfalls of a Core Data application. The camera functionality is introduced to highlight these issues and demonstrates just how important good model design is to a well-performing application.

• Chapter 11, “Background Processing”—Top-notch performance requires intensive tasks be offloaded to a background thread. You learn just how easy it is to run processes in the background as the example of photo thumbnail generation is added with a generic helper class called CDThumbmaller. You also learn how to keep memory usage low with another helper class, called CDFaulter.

• Chapter 12, “Search”—This chapter shows you how to integrate a UISearchController with Core Data as you implement efficient search in CDTableViewController.

• Chapter 13, “iCloud”—Enjoy the easiest, most reliable Core Data integration with iCloud yet. You learn how to handle multiple accounts and varying preferences on using iCloud without missing a beat.

• Chapter 14, “Taming iCloud”—Take iCloud integration to the next level with entity-level seeding and unique object de-duplication. This chapter shows you how to emulate first-time iCloud use by resetting ubiquitous content globally, the right way.

• Chapter 15, “CloudKit Sync: Uploading Objects”—This chapter shows you how to leverage a public CloudKit database to keep a small data set synchronized across the devices of a group of iCloud users. Part 1 lays the foundation for synchronization as the capability to automatically upload new NSManagedObjects is added.

• Chapter 16, “CloudKit Sync: Downloading Changes and Handling Deletions”—The synchronization implementation is finalized as deletion support and the capability to download changed CloudKit records are added.

• Appendix A, “Preparing the Groceries Application”—Every (non–Core Data) step involved in preparing the starting-point application for Chapter 1 is documented here for completeness.

• Appendix B, “Finalizing the Groceries Application”—Every (non–Core Data) step involved in finalizing the application for the App Store is documented here for completeness.

Getting the Sample Code
The sample code built throughout this book is available for download from timroadley.com. Links are given in each chapter, or you can use Table P.1 as a reference, which is arranged in the order of implementation.
Table P.1  Groceries Code

<table>
<thead>
<tr>
<th>Final Code</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterAppendixA.zip">http://timroadley.com/LCDwS/Groceries-AfterAppendixA.zip</a></td>
</tr>
<tr>
<td>Chapter 1</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter01.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter01.zip</a></td>
</tr>
<tr>
<td>Chapter 2</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter02.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter02.zip</a></td>
</tr>
<tr>
<td>Chapter 3</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter03.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter03.zip</a></td>
</tr>
<tr>
<td>Chapter 4</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter04.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter04.zip</a></td>
</tr>
<tr>
<td>Chapter 5</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter05.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter05.zip</a></td>
</tr>
<tr>
<td>Chapter 6</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter06.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter06.zip</a></td>
</tr>
<tr>
<td>Chapter 7</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter07.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter07.zip</a></td>
</tr>
<tr>
<td>Chapter 8</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter08.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter08.zip</a></td>
</tr>
<tr>
<td>Chapter 9</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter09.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter09.zip</a></td>
</tr>
<tr>
<td>Chapter 10</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter10.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter10.zip</a></td>
</tr>
<tr>
<td>Chapter 11</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter11.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter11.zip</a></td>
</tr>
<tr>
<td>Chapter 12</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter12.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter12.zip</a></td>
</tr>
<tr>
<td>Chapter 13</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter13.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter13.zip</a></td>
</tr>
<tr>
<td>Chapter 14</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter14.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter14.zip</a></td>
</tr>
<tr>
<td>Chapter 14 “Mini-project”</td>
<td><a href="http://timroadley.com/LCDwS/EZiCloud.zip">http://timroadley.com/LCDwS/EZiCloud.zip</a></td>
</tr>
<tr>
<td>Chapter 15</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter15.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter15.zip</a></td>
</tr>
<tr>
<td>Chapter 16</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterChapter16.zip">http://timroadley.com/LCDwS/Groceries-AfterChapter16.zip</a></td>
</tr>
<tr>
<td>Chapter 16 “Mini-project”</td>
<td><a href="http://timroadley.com/LCDwS/EZCloudKit.zip">http://timroadley.com/LCDwS/EZCloudKit.zip</a></td>
</tr>
<tr>
<td>Appendix B “Final Project”</td>
<td><a href="http://timroadley.com/LCDwS/Groceries-AfterAppendixB.zip">http://timroadley.com/LCDwS/Groceries-AfterAppendixB.zip</a></td>
</tr>
<tr>
<td>Helper classes, for your own projects, with CloudKit support</td>
<td><a href="http://timroadley.com/LCDwS/Generic%20Core%20Data%20Classes.zip">http://timroadley.com/LCDwS/Generic%20Core%20Data%20Classes.zip</a></td>
</tr>
</tbody>
</table>

**Conventions Used in This Book**

**Code Breaks**

Occasionally lines of code in the chapters are too long to fit on the printed page. Where that occurs, a code-continuation arrow (➡) has been used to mark the continuation. For example:

```swift
let _localStore = self.coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
    configuration: nil, URL: self.localStoreURL, options: options, error: &error)
```
Sample Code Updates

When it’s time to apply what you’ve learned to the sample application, the instructions are preceded by the words “Update Groceries as follows”. You should apply the subsequent steps to the sample application as instructed. The instructions make heavy use of bold formatting to clarify the interface elements you can expect to interact with. For example:

Update Groceries as follows to link to the Core Data Framework:

1. Select the **Groceries Target**, as shown in Figure 1.2.
2. Click the + found in the **Linked Frameworks and Libraries** section of the **General** tab and then link to the **CoreData.framework**, as shown in Figure 1.2.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we’re doing right, what we could do better, what areas you’d like to see us publish in, and any other words of wisdom you’re willing to pass our way.

You can email or write me directly to let me know what you did or didn’t like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book’s title and author as well as your name and phone number or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: trina.macdonald@pearson.com

Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116

Register your copy of *Learning Core Data for iOS with Swift* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN 9780134120034 and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”
In Chapter 2, “Managed Object Model Basics,” the fundamentals of managed object models were introduced, yet you were constrained to just one entity and a few attributes. The next logical step is to add to the model; however, this requires a number of preliminary steps to prevent errors caused by these changes. This chapter shows how to add model versions and model mappings, and it demonstrates different migration techniques you can choose when upgrading a model.

**Changing a Managed Object Model**

As an application evolves, its managed object model probably needs to change, too. Simple changes, such as attribute defaults, validation rules, and fetch request templates can be modified without consequence. Other more structural changes require that persistent stores be migrated to new model versions. If a persistent store doesn’t have the appropriate mappings and settings required to migrate data from one version to the next, the application throws a “store is incompatible” error.
Chapter 3  Managed Object Model Migration

**Note**
To continue building the sample application, you need to have added the previous chapter’s code to Groceries. Alternatively, you may download, unzip, and use the project up to this point from http://www.timroadley.com/LCDws/Groceries-AfterChapter02.zip. Any time you start using an Xcode project from a ZIP file, it’s good practice to hold down option and click **Product > Clean Build Folder**. This practice ensures there’s no residual cache from previous projects using the same name. It is recommended that you use the iOS Simulator when following this chapter so you can inspect the contents of the SQLite database files easily.

Update Groceries as follows to generate a “store is incompatible” error:

1. Run Groceries once to ensure the existing model has been used to create the persistent store. You should see the file system location of the store printed to the console log.
2. Select **Model.xcdatamodeld** in Xcode.
3. Add a new entity and rename it to **Measurement**.
4. Select the **Measurement** entity and add an attribute called **abc**. Set its type to **String**.
5. Rerun the application and examine the console log. You should now have generated arguably one of the most common Core Data errors, as shown in Figure 3.1. If this error has not appeared, delete the application and then click **Product > Clean** and retry from step 1.

![Image](image.png)

Figure 3.1  Store incompatibility

This error isn’t an issue when an application is in its initial development phase. To get past it, you can just delete the application from the device and run it again from Xcode. When the application is run for the first time after being deleted, the persistent store is created based on the latest model. This makes the store compatible with the model, so the application won’t throw the error anymore. **However**, it won’t have any old data in it. As such, this scenario is unacceptable for any application already available on the App Store. There are a few approaches to migrating existing persistent stores, and the migration path you choose is driven by the complexity of the changes and whether you’re using iCloud. Whatever you do, you first need to become familiar with model versioning.
Update Groceries as follows to revert to the original model:

1. Select `Model.xcdatamodeld`.
2. Delete the `Measurement` entity.
3. Rerun the application, which now should not throw an error.

**Adding a Model Version**

To avoid the error shown in Figure 3.1, you need to create a new model version before making changes to the model. Ongoing, you should not remove old versions of a model. Old model versions are needed to help migrate incompatible persistent stores to the current model version. If there are no existing persistent stores on customer devices, you can ignore model versioning until your application is on the App Store.

Update Groceries as follows to add a model version:

1. Select `Model.xcdatamodeld`.
2. Click `Editor > Add Model Version`.
3. Click `Finish` to accept `Model 2` as the version name.

You should now have two model versions, as shown in Figure 3.2.

The new model `Model 2.xcdatamodel` starts out as a replica of `Model.xcdatamodel`. This unfortunately makes it easy to modify the wrong version unintentionally. Before you edit a model, you should triple-check you have selected the correct one. You may want to get into the
habit of taking a snapshot, committing to source control, or even backing up the whole project prior to editing a model. Note that the check mark in the green circle represents the current model, which is the model version used at runtime.

Update Groceries as follows to reintroduce the `Measurement` entity:

1. Optionally take a snapshot or back up the Groceries project.
2. Select `Model 2.xcdatamodel`.
3. Add a new entity and rename it to `Measurement`.
4. Select the `Measurement` entity and add an attribute called `abc`. Set its type to `String`.

After you add the new model version, you still need to set it as the current version before it is used by the application.

Update Groceries as follows to change the current model version:

1. Select `Model.xcdatamodel` (not Model.xcdatamodel).
2. Click View > Utilities > Show File Inspector (or press Option+⌘+1).
3. Set the Current Model Version to Model 2, as shown at the bottom of Figure 3.3.

![Figure 3.3 Setting the current model](image)
Before you can successfully launch the application, you need to configure migration options to
tell Core Data how to migrate. Feel free to launch it again to generate the incompatible store
error in the meantime.

**Lightweight Migration**

Whenever a new model is set as the current version, existing persistent stores must be migrated
to use them. This is because the persistent store coordinator tries to use the current model
to open the existing store, which fails if the store was created using a previous version of the
model. The process of store migration can be handled automatically by passing an options
dictionary to a persistent store coordinator when a store is added:

- When the `NSMigratePersistentStoresAutomaticallyOption` is true (1) and passed to
  a persistent store coordinator, Core Data automatically attempts to migrate incompatible
  persistent stores to the current model.
- When the `NSInferMappingModelAutomaticallyOption` is true (1) and passed to a
  persistent store coordinator, Core Data automatically attempts to infer a best guess
  at what attributes from the *source* model entities should end up as attributes in the
  *destination* model entities.

Using those persistent store coordinator options together is called **lightweight migration** and
is demonstrated in bold in Listing 3.1. These options are set in an updated `localStore` variable
of `CDHelper.swift`. Note that if you’re using iCloud, this is your only choice for migration.

Listing 3.1  **The Local Store (CDHelper.swift localStore)**

```swift
// MARK: - STORE
lazy var localStore: NSPersistentStore? = {
    let options: [NSObject: AnyObject] = [
        NSSQLitePragmasOption: "journal_mode": "DELETE",
        NSMigratePersistentStoresAutomaticallyOption: 1,
        NSInferMappingModelAutomaticallyOption: 1
    ]

    var _localStore: NSPersistentStore?
    do {
        _localStore = try self.coordinator.addPersistentStoreWithType
        (NSSQLiteStoreType, configuration: nil, URL: self.localStoreURL, options: options)
        return _localStore
    } catch {
        return nil
    }
}()
```
Update Groceries as follows to enable lightweight migration:

1. Replace the existing `localStore` variable code in `CDHelper.swift` with the code from Listing 3.1. The key change to be aware of is the introduction of the bold code.

2. Rerun the application, which should not throw an error.

From now on, any time you set a new model as the current version and lightweight migration is enabled, the migration should occur transparently.

Before other migration types can be demonstrated, some test data needs to be generated. Listing 3.2 contains code that generates managed objects based on the `Measurement` entity. You may notice that this code blocks the user interface until it finishes because the context runs on the main thread. More appropriate ways to insert test data in the background are demonstrated later in the book.

**Listing 3.2 Inserting Test Measurement Data (AppDelegate.swift demo)**

```swift
func demo () {
    let context = CDHelper.shared.context
    for i in 0...50000 {
        if let newMeasurement = NSEntityDescription.insertNewObjectForEntityForName
            ("Measurement", inManagedObjectContext: context) as? Measurement {
            newMeasurement.abc = "--> LOTS OF TEST DATA x\(i)"
            print("Inserted \(newMeasurement.abc!)")
        }
    }
    CDHelper.saveSharedContext()
}
```

Update Groceries as follows to generate test data:

1. Create an `NSManagedObject` subclass of the `Measurement` entity. As discussed in Chapter 2, this is achieved by first selecting the entity and then clicking `Editor > Create NSManagedObject Subclass...` and following the prompts. When it comes time to save the class file, don’t forget to save the file in the `Data Model` group and check the `Groceries` target.

2. Replace the `demo` function in `AppDelegate.swift` with the code from Listing 3.2.

3. Run the application once. This inserts a lot of test data into the persistent store, which you can monitor by examining the console log. This may take a little while, depending on the speed of your machine. It’s important to have a fair amount of data in the persistent store to demonstrate the speed of migrations later. Note that the table view still remains blank because it has not yet been configured to display anything.
The next step is to reconfigure the `demo` function to show some of what’s in the persistent store. The code shown in Listing 3.3 fetches a small sample of `Measurement` data. Notice that a new option is included that limits fetched results to 50. This is great for limiting how many results are fetched from large data sets, and it is even more powerful when mixed with sorting to generate a Top-50, for example.

**Listing 3.3  Fetching Test Measurement Data (AppDelegate.swift demo)**

```swift
func demo () {
    let context = CDHelper.shared.context
    let request = NSFetchRequest(entityName: "Measurement")
    request.fetchLimit = 50

    do {
        if let measurements = try context.executeFetchRequest(request) as? [Measurement] {
            for measurement in measurements {
                print("Fetched Measurement Object \(measurement.abc!)")
            }
        }
    } catch {
        print("ERROR executing a fetch request: \(error)")
    }
}
```

Update Groceries as follows to print a sample of the store contents to the console log:

1. Replace the `demo` function in `AppDelegate.swift` with the code from Listing 3.3.
2. Run the application. The console log should show 50 rows of seemingly random measurement objects.
3. Examine the contents of the `LocalStore.sqlite` file using **SQLite Database Browser**, as explained previously in Chapter 2. Figure 3.4 shows the expected results when viewing the `ZMEASUREMENT` table, which is the data for the Measurement entity.

Close the SQLite Database Browser before continuing.
Chapter 3  Managed Object Model Migration

Default Migration

Sometimes, you need more control than what lightweight migration offers. Let’s say, for instance, you want to replace the Measurement entity with another entity called Amount. You also want the abc attribute from the Measurement entity to end up as an xyz attribute in the Amount entity. Any existing abc data should also be migrated to the xyz attribute. To achieve these requirements, you need to create a model mapping to manually specify what maps to where.

When the persistent store option NSInferMappingModelAutomaticallyOption is true (1), Core Data still checks to see whether there are any model-mapping files it should use before trying to infer automatically. It is recommended that you disable this setting while you’re testing a mapping model. This way, you can be certain that the mapping model is being used and is functioning correctly.

Update Groceries as follows to disable automatic model mapping:

1. Set the NSInferMappingModelAutomaticallyOption option in the localStore variable of CDHelper.swift to false by changing the 1 to a 0.

Update Groceries as follows to add a new model in preparation for the migration from the Measurement entity to the Amount entity:
1. Optionally take a snapshot or back up the project.
2. Add a new model version called Model 3 based on Model 2 (Editor > Add Model Version...).
3. Set Model 3 as the current model version.
4. Select Model 3.xcdatamodel.
5. Delete the Measurement entity.
6. Add a new entity called Amount with a String attribute called xyz.
7. Create an NSManagedObject subclass of the Amount entity. When it comes time to save the class file, don’t forget to save the file in the Data Model group and check the Groceries target.
8. Replace the demo function of AppDelegate.swift with the code from Listing 3.4. Similar to the code being replaced, this code simply fetches a small sample of Amount data instead of Measurement data.
9. Run the application, which should throw the “Can’t find mapping model for migration” error shown in Figure 3.5.

Listing 3.4  Fetching Test Amount Data (AppDelegate.swift demo)

```swift
func demo () {
    let context = CDHelper.shared.context
    let request = NSFetchRequest(entityName: "Amount")
    request.fetchLimit = 50

    do {
        if let amounts = try context.executeFetchRequest(request) as? [Amount] {
            for amount in amounts {
                print("Fetched Amount Object \(amount.xyz!)")
            }
        } catch {
            print("ERROR executing a fetch request: \(error)")
        }
    }
}
```

To resolve the error shown in Figure 3.5, you need to create a mapping model that shows where attributes map. Specifically, the requirement is to map the old Measurement abc attribute to the new Amount xyz attribute.
Update Groceries as follows to add a new mapping model:

1. Ensure the Data Model group is selected.
2. Click File > New > File....
3. Select iOS > Core Data > Mapping Model and then click Next.
4. Select Model 2.xcdatamodel as the Source Data Model and then click Next.
5. Select Model 3.xcdatamodel as the Target Data Model and then click Next.
6. Set the mapping model name to save as Model2toModel3.
7. Ensure the Groceries target is checked and then click Create.
8. Select Model2toModel3.xcmappingmodel.

You should now be presented with the model-mapping editor, as shown in Figure 3.6.

The mappings you're presented with are a best guess based on what Core Data can infer on its own. On the left you should see Entity Mappings, showing what source entities map to what
destination entities. You should also see in Figure 3.6 how the source Item entity has already inferred that it should map to the destination Item entity, which is a fair assumption. The naming standard of an entity mapping is SourceToDestination. With this in mind, notice the Amount entity doesn’t seem to have a source entity because it never existed in the source model.

Update Groceries as follows to map the old Measurement entity to the new Amount entity:

1. Select the Amount entity mapping.
2. Click View > Utilities > Show Mapping Model Inspector (if that’s not visible in the menu system, press Option+⌘+3). You need to be able to see the pane shown on the right in Figure 3.7.
3. Set the Source of the Amount entity mapping to Measurement. The expected result is shown in Figure 3.7.

![Figure 3.7 Custom entity mapping of MeasurementToAmount](image)

Because Measurement was selected as the source entity for the Amount destination entity, the Entity Mapping Name was automatically renamed to MeasurementToAmount. In addition, the mapping type changed from Add to Transform. For more complex implementations, you can specify a custom policy in the form of an NSEntityMigrationPolicy subclass. By overriding createDestinationInstancesForSourceInstance in the subclass, you can manipulate the data that’s migrated. For example, you could intercept the values of the abc attribute, set them all to title case, and then migrate them to the xyz attribute.

The Source Fetch option shown at the bottom right of Figure 3.7 allows you to limit the migrated data to the results of a predicated (filtered) fetch. This is useful if you only want a subset of the existing data to be migrated. The predicate format you use here is the same as the format you would use when normally configuring a predicate, except you use $source variables. An example of a predicate that would filter out nil source data from the abc attribute is $source.abc != nil.

Select the ItemToItem entity mapping shown previously in Figure 3.6 and examine its attribute mappings. Notice how each destination attribute has a Value Expression set. Now examine the
MeasurementToAmount entity mapping. Notice there’s no value expression for the xyz destination attribute. This means that the xyz attribute has no source attribute, and you need to set one using the same format used in the ItemToItem entity mapping. The original requirement was to map the abc attribute to the xyz attribute, so that’s what needs configuring here.

Update Groceries as follows to set an appropriate value expression for the xyz destination attribute:

1. Set the Value Expression for the xyz destination attribute of the MeasurementToAmount entity mapping to $source.abc.
2. Run the application. So long as the migration has been successful, you should see the expected result in the console log, as shown in Figure 3.8.

![Figure 3.8 Results of a successfully mapped model](image)

To verify the migration has persisted to the store, examine the contents of the LocalStore.sqlite file using the techniques discussed in Chapter 2. The expected result is shown in Figure 3.9, which illustrates the new ZAMOUNT table (that is, Amount entity) with the data from the old Measurement entity.

Close the SQLite Database Browser before continuing.

**Migration Manager**

Instead of letting a persistent store coordinator perform store migrations, you may want to manually migrate stores using an instance of NSMigrationManager. Using a migration manager still uses a mapping model; however, the difference is you have total control over the migration and the ability to report progress. To be certain that migration is being handled manually, automatic migration should be disabled.

Update Groceries as follows to disable automatic migration:

1. Set the NSMigratePersistentStoresAutomaticallyOption option in the localStore variable of CDHelper.swift to false by changing the 1 to a 0.
Reporting on the progress of a migration is useful for keeping the user informed (and less annoyed) about a slow launch. Although most migrations should be fast, some large databases requiring complex changes can take a while to migrate. To keep the user interface responsive, the migration must be performed on a background thread. At the same time, the user interface has to be responsive to provide updates to the user. The challenge is to prevent the user from attempting to use the application during the migration. This is because the data won’t be ready yet, so you don’t want the user staring at a blank screen wondering what’s going on. This is where a migration progress view comes into play.

Update Groceries as follows to configure a migration View Controller:

1. Select `Main.storyboard`.
2. Drag a new `View Controller` onto the storyboard, placing it above the existing Navigation Controller.
3. Drag a new `Label` and `Progress View` onto the new View Controller.
4. Position the `Progress View` directly in the center of the View Controller and then position the `Label` above it in the center.
5. Widen the `Label` and `Progress View` to the width of the View Controller margins, as shown in the center of Figure 3.10.
6. Configure the **Label** with **Centered** text that reads **Migration Progress 0%**, as shown in the center of Figure 3.10.

7. Configure the **Progress View** progress to 0.

8. Select the View Controller and set its **Storyboard ID** to migration using **Identity Inspector** (**Option**+**⌘**+3).

9. Optionally configure the following layout constraints by holding down the control key and dragging from the progress bar toward the applicable margin. You may skip this step if you’re uncomfortable with constraints as it is not critical.
   - Leading Space to Container Margin
   - Trailing Space to Container Margin
   - Center Vertically In Container

10. Optionally configure the following layout constraints by holding down the control key and dragging from the progress label toward the applicable margin. You may skip this step if you’re uncomfortable with constraints as it is not critical.
    - Leading Space to Container Margin
    - Trailing Space to Container Margin
    - Vertical Spacing from the progress bar

![Figure 3.10 Migration View Controller](image)

**Introducing MigrationVC.swift (Migration View Controller Code)**

The new migration View Controller has UILabel and UIProgressView interface elements that need updating during a migration. This means that a way to refer to these interface elements in code is required. A new UIViewController subclass called MigrationVC should be created for this purpose.
Update Groceries as follows to add a MigrationVC file to a new group:

1. Right-click the existing Groceries group and then select New Group.
2. Set the new group name to View Controllers. This group will contain all of the view controllers. As a side note, feel free to move ViewController.swift to the trash because it is no longer required.
3. Select the View Controllers group.
4. Click File > New > File....
5. Create a new iOS > Source > Cocoa Touch Class and then click Next.
6. Set the subclass to UIViewController and the filename to MigrationVC.
7. Ensure the language is Swift and then click Next.
8. Ensure the Groceries target is checked and that the new file will be saved in the Groceries project directory; then click Create.
9. Select Main.storyboard.
10. Set the Custom Class of the new migration View Controller to MigrationVC using Identity Inspector (Option+⌘+3) while the View Controller is selected. This is in the same place as where the Storyboard ID was set.
11. Show the Assistant Editor by clicking View > Assistant Editor > Show Assistant Editor (or pressing Option+⌘+Return).
12. Ensure the Assistant Editor is automatically showing MigrationVC.swift. The top-right of Figure 3.11 shows what this looks like. If you need to, just click Manual or Automatic while the migration View Controller is selected and select MigrationVC.swift.
13. Hold down the control key while dragging a line from the migration progress label to the code in MigrationVC.swift on the line before the viewDidLoad function. When you let go of the mouse button, a pop-up appears. In the pop-up, set the Name to label and ensure the Storage is set to Strong before clicking Connect. Figure 3.11 shows the intended configuration.
14. Repeat the technique in step 13 to create a linked UIProgressView variable from the progress view called progressView.

There should now be an @IBOutlet called label and an @IBOutlet called progressView in MigrationVC.swift. You may now switch back to the Standard Editor (⌘ + return).

When a migration occurs, notifications that communicate progress need to be sent. For the progress bar to reflect the progress, a new function is required in MigrationVC.swift. In addition, this function needs to be called every time a progress update is observed. Listing 3.5 shows the new code involved in bold.
import UIKit

class MigrationVC: UIViewController {

@IBOutlet var label: UILabel!
@IBOutlet var progressView: UIProgressView!

// MARK: - MIGRATION
func progressChanged (note:AnyObject?) {
    if let _note = note as? NSNotification {
        if let progress = _note.object as? NSNumber {
            let progressFloat:Float = round(progress.floatValue * 100)
            let text = "Migration Progress: \(progressFloat)%"
            print(text)
            dispatch_async(dispatch_get_main_queue(), {
                self.label.text = text
                self.progressView.progress = progress.floatValue
            })
        } else {print("(__FUNCTION__) FAILED to get progress")}
        } else {print("(__FUNCTION__) FAILED to get note")}
    }

override func viewDidLoad() {
    super.viewDidLoad()
    NotificationCenter.defaultCenter().addObserver(self, selector:
"progressChanged:", name: "migrationProgress", object: nil)
    }

Figure 3.11 Creating storyboard-linked properties to MigrationVC.swift

Listing 3.5 Migration View Controller (MigrationVC.swift)

import UIKit

class MigrationVC: UIViewController {

@IBOutlet var label: UILabel!
@IBOutlet var progressView: UIProgressView!

// MARK: - MIGRATION
func progressChanged (note:AnyObject?) {
    if let _note = note as? NSNotification {
        if let progress = _note.object as? NSNumber {
            let progressFloat:Float = round(progress.floatValue * 100)
            let text = "Migration Progress: \(progressFloat)%"
            print(text)
            dispatch_async(dispatch_get_main_queue(), {
                self.label.text = text
                self.progressView.progress = progress.floatValue
            })
        } else {print("(__FUNCTION__) FAILED to get progress")}
        } else {print("(__FUNCTION__) FAILED to get note")}
    }

override func viewDidLoad() {
    super.viewDidLoad()
    NotificationCenter.defaultCenter().addObserver(self, selector:
"progressChanged:", name: "migrationProgress", object: nil)
    }

Figure 3.11 Creating storyboard-linked properties to MigrationVC.swift

Listing 3.5 Migration View Controller (MigrationVC.swift)
The `progressChanged` function simply unwraps the progress notification and constructs a string with the migration completion percentage. It then updates the user interface with this information. Of course, none of this can happen without first adding an observer of the `migrationProgress` variable in the `viewDidLoad` function. When the view deinitializes, it is unregistered as an observer of the `migrationProgress` variable.

Update Groceries as follows to ensure migration progress is reported to the user:

1. Replace all code in `MigrationVC.swift` with the code from Listing 3.5.

The user interface is now positioned to report migration progress to the user. The next step is to implement the code required to perform a manual migration.

### Introducing `CDMigration.swift` (Core Data Migration Code)

To keep `CDHelper.swift` small, the code required to perform a managed migration is put in a new class called `CDMigration.swift`. The starting point to this class is shown in Listing 3.6.

```swift
import UIKit
import CoreData

private let _sharedCDMigration = CDMigration()

class CDMigration: NSObject {

  // MARK: - SHARED INSTANCE
  class var shared: CDMigration {
    return _sharedCDMigration
  }

  // MARK: - MANAGED MIGRATION
  func performMigration() {
    // Code to perform migration...}
}
```

Just like `CDHelper.swift`, `CDMigration.swift` has a `shared` function that makes it easy to use because you can call it from anywhere in the project via `CDMigration.shared`.

Update Groceries as follows to implement `CDMigration.swift`:

1. Select the Generic Core Data Classes group.
2. Click File > New > File....
3. Create a new iOS > Source > Swift File and then click Next.
4. Set the filename to `CDMigration` and ensure the `Groceries` target is checked.

5. Ensure the `Groceries` project directory is open and then click **Create**.

6. Replace the contents of `CDMigration.swift` with the code from Listing 3.6.

To handle migrations manually, three supporting functions are required. One function checks that a given store exists and another checks that it needs migrating. A successful migration generates a separate compatible store, so as soon as migration completes, this new store needs to replace the incompatible one. The final supporting function does exactly that—it replaces the incompatible store with the migrated store. Listing 3.7 shows the code involved with these three supporting functions.

Listing 3.7  Migration View Controller Supporting Functions (`CDMigration.swift`)

```swift
// MARK: - SUPPORTING FUNCTIONS
func storeExistsAtPath(storeURL: NSURL) -> Bool {
    if let _storePath = storeURL.path {
        if NSFileManager.defaultManager().fileExistsAtPath(_storePath) {
            return true
        }
    } else {print("\(__FUNCTION__) FAILED to get store path")}
    return false
}

func store(storeURL: NSURL, isCompatibleWithModel model: NSManagedObjectModel) -> Bool {
    if self.storeExistsAtPath(storeURL) == false {
        return true // prevent migration of a store that does not exist
    }

    do {
        var _metadata: [String : AnyObject]?
        _metadata = try NSPersistentStoreCoordinator.metadataForPersistentStoreOfType
            (NSSQLiteStoreType, URL: storeURL, options: nil)
        if let metadata = _metadata {
            if model.isConfiguration(nil, compatibleWithStoreMetadata: metadata) {
                print("The store is compatible with the current version of the model")
                return true
            }
        } else {print("\(__FUNCTION__) FAILED to get metadata")}
    } catch {
        print("ERROR getting metadata from \(storeURL) \{error}\")
    }
    print("The store is NOT compatible with the current version of the model")
    return false
}
```
func replaceStore(oldStore: NSURL, newStore: NSURL) throws {

    let manager = NSFileManager.defaultManager()

do {
        try manager.removeItemAtURL(oldStore)
        try manager.moveItemAtURL(newStore, toURL: oldStore)
    }
}

The `storeExistsAtPath` function uses `NSFileManager` to determine whether a store exists at the given URL. It returns a `Bool` indicating the result.

The `store:isCompatibleWithModel` function first checks that a store exists at the given path. If there is no store, `true` is returned because this prevents a migration from being attempted. If a store exists at the given URL, it is checked for model compatibility against the given model. To do this, the model used to create the store is drawn from the store's metadata and then compared to the given model via its `isConfiguration:compatibleWithStoreMetadata` function.

The `replaceStore` function uses `NSFileManager` to remove the incompatible store from the file system and then replaces it with the compatible store.

Update Groceries as follows to implement a new SUPPORTING FUNCTIONS section:

1. Add the code from Listing 3.7 to the bottom of `CDMigration.swift` before the last curly brace.

When a migration is in progress, the value of the migration manager's `migrationProgress` variable is constantly updated. This is information that the user needs to see, so a function is required to react whenever the `migrationProgress` value changes. Listing 3.8 shows a new function that posts a notification whenever this value changes.

Listing 3.8  Migration View Controller Progress Reporting (`CDMigration.swift`)  
// MARK: - PROGRESS REPORTING
override func observeValueForKeyPath(keyPath: String?, ofObject object: AnyObject?, change: [String : AnyObject]?, context: UnsafeMutablePointer<Void>) {

    if object is NSMigrationManager, let manager = object as? NSMigrationManager {

        if let notification = keyPath {
            NotificationCenter.defaultCenter().postNotificationName(notification, object: NSNumber(float: manager.migrationProgress))
        } else { print("observeValueForKeyPath did not receive a NSMigrationManager class")
    }
}
Update Groceries as follows to implement a new **PROGRESS REPORTING** section:

1. Add the code from Listing 3.8 to the bottom of CDMigration.swift before the last curly brace.

The next function is where the actual migration happens. Most of this function is used to gather all the pieces required to perform a migration. Listing 3.9 shows the code involved.

### Listing 3.9  **Migration (CDMigration.swift migrateStore)**

```swift
// MARK: - MIGRATION
func migrateStore(store: NSURL, sourceModel: NSManagedObjectModel, destinationModel: NSManagedObjectModel) {
    if let tempdir = store.URLByDeletingLastPathComponent {
        let tempStore = tempdir.URLByAppendingPathComponent("Temp.sqlite")
        let mappingModel = NSMappingModel(fromBundles: nil, forSourceModel: sourceModel, destinationModel: destinationModel)
        let migrationManager = NSMigrationManager(sourceModel: sourceModel, destinationModel: destinationModel)
        migrationManager.addObserver(self, forKeyPath: "migrationProgress", options: .new, context: nil)
        do {
            try migrationManager.migrateStoreFromURL(store, type: NSSQLiteStoreType, options: nil, withMappingModel: mappingModel, toDestinationURL: tempStore, destinationType: NSSQLiteStoreType, destinationOptions: nil)
            try replaceStore(store, newStore: tempStore)
            print("SUCCESSFULLY MIGRATED \(store) to the Current Model")
        } catch {
            print("FAILED MIGRATION: \(error)"
        }
        migrationManager.removeObserver(self, forKeyPath: "migrationProgress")
    } else {
    print("\(__FUNCTION__) FAILED to prepare temporary directory")
} }
```

The `migrateStore` function needs to be given a store to migrate, a source model to migrate from, and destination model to migrate to. The source model could have been taken from the given store’s metadata; however, seeing as this step is performed first in another function, this approach saves repeated code.
The first thing migrateStore does is prepare four variables:

- The tempdir variable holds the URL to the given store and is used to build a URL to a temporary store used for migration.
- The tempStore variable holds the URL to the temporary store used for migration.
- The mappingModel variable holds an instance of NSMappingModel specific to the models being migrated from and to. The migration will fail without a mapping model.
- The migrationManager variable holds an instance of NSMigrationManager based on the source and destination models. An observer is added for the migrationProgress variable so that the observeValueForKeyPath function is called whenever the migrationProgress variable changes.

All these variables are then used to make a call to the migrateStoreFromURL function, which is responsible for migrating the given store to be compatible with the destination model. Once this is complete, the old incompatible store is removed and the new compatible store is put in its place.

Update Groceries as follows to implement a new MIGRATION section:

1. Add the code from Listing 3.9 to the bottom of CDMigration.swift before the final closing curly brace.

The migration code that has just been implemented needs to be called from a background thread so that the user interface can be updated without freezing. This, along with the instantiation of the progress view that the user sees, is shown in Listing 3.10.

Listing 3.10  Migration Progress (CDMigration.swift migrateStoreWithProgressUI)

```swift
func migrateStoreWithProgressUI(store: NSURL, sourceModel: NSManagedObjectModel, destinationModel: NSManagedObjectModel) {

    // Show migration progress view preventing the user from using the app
    let storyboard = UIStoryboard(name: "Main", bundle: nil)

    if let initialVC = UIApplication.sharedApplication().keyWindow?.rootViewController as? UINavigationController {
        if let migrationVC = storyboard.instantiateViewController(withIdentifier: "migration") as? MigrationVC {
            initialVC.presentViewController(migrationVC, animated: false, completion: {
                dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0), {
                    print("BACKGROUND Migration started...")
                    self.migrateStore(store, sourceModel: sourceModel, destinationModel: destinationModel)
                    dispatch_async(dispatch_get_main_queue(), {
                        // ... other code ...
                    })
            })
        }
    }
}
```
Chapter 3  Managed Object Model Migration

---

// trigger the stack setup again, this time with the upgraded
store
let _ = CDHelper.shared.localStore
dispatch_after(2, dispatch_get_main_queue(), {
migrationVC.dismissViewControllerAnimated(false,
completion: nil)
})
})
})
})
} else {print("FAILED to find a view controller with a story board id of
"'migration'")
} else {print("FAILED to find the root view controller, which is supposed to be a
navigation controller")
}

The migrateStoreWithProgressUI function uses a storyboard identifier to instantiate and
present the migration view. Once the view is blocking user interaction the migration can begin.
The migrateStore function is called on a background thread. Once migration is complete, the
localStore is loaded as usual, the migration view is dismissed, and normal use of the applica-
tion can resume.

Update Groceries as follows to implement the migrateStoreWithProgressUI function:

1. Add the code from Listing 3.10 to the MIGRATION section at the bottom of
CDMigration.swift before the last curly brace.

The final piece of code required in CDMigration.swift is used to migrate the store if neces-
sary. This function is called from the setupCoreData function of CDHelper.swift, which is
run as a part of initialization. Listing 3.11 shows the code involved.

Listing 3.11  Migration (CDMigration.swift migrateStoreIfNecessary)

```swift
func migrateStoreIfNecessary (storeURL: NSURL, destinationModel: NSManagedObjectModel) {
    if storeExistsAtPath(storeURL) == false {
        return
    }

    if store(storeURL, isCompatibleWithModel: destinationModel) {
        return
    }

do {
    var _metadata:[String : AnyObject]?
    _metadata = try NSPersistentStoreCoordinator.metadataForPersistentStoreOfType
    (NSSQLiteStoreType, URL: storeURL, options: nil)
    ```
if let metadata = _metadata, let sourceModel = 
  NSManagedObjectModel.mergedModelFromBundles([NSBundle.mainBundle()], 
  forStoreMetadata: metadata) {
    self.migrateStoreWithProgressUI(storeURL, sourceModel: sourceModel, 
    destinationModel: destinationModel)
  } catch {
    print("\(__FUNCTION__) FAILED to get metadata \(error)")
  }
}

Once it’s established that the given store exists, a model compatibility check is performed and 
the store is migrated if necessary. The model used to create the given store is drawn from the 
store’s metadata. This is then given to the migrateStoreWithProgressUI function.

Update Groceries as follows to implement the migrateStoreIfNecessary function:

1. Add the code from Listing 3.11 to MIGRATION section at the bottom of 
   CDMigration.swift before the last curly brace.

When CDHelper.swift initializes, a call is made to setup CoreData. This is an ideal time to 
check that the localStore is compatible with the current model, before it’s needed. The new 
code required in the setup CoreData is shown in bold in Listing 3.12.

Listing 3.12 Migration During Setup (CDHelper.swift setup CoreData)

```
func setup CoreData() {
    // Model Migration
    if let _localStoreURL = self.localStoreURL {
        CDMigration.shared.migrateStoreIfNecessary(_localStoreURL, destinationModel: 
        self.model)
    }

    // Load Local Store
    _ = self.localStore
}
```

Update Groceries as follows to ensure a manual migration is triggered as required:

1. Replace the setup CoreData function of CDHelper.swift with the code from 
   Listing 3.12.

Currently the localStore variable of CDHelper.swift always tries to return a store. If it tried 
to return a store that wasn’t compatible with the current model, the application would throw 
an error. To prevent this, a check is needed to see whether the store needs migrating before it 
is loaded. This check is needed only when migration is handled manually, so the bold code in 
Listing 3.13 wouldn’t be required otherwise.
Listing 3.13  Triggering Migration Manager (CDHelper.swift localStore)

```swift
lazy var localStore: NSPersistentStore? = {

    let useMigrationManager = true
    if let _localStoreURL = self.localStoreURL {
        if useMigrationManager == true &&
            CDMigration.shared.storeExistsAtPath(_localStoreURL) &&
            CDMigration.shared.store(_localStoreURL, isCompatibleWithModel:
                self.model) == false {
            return nil // Don't return a store if it's not compatible with the model
        }
    }

    let options:[NSObject:AnyObject] = [NSSQLitePragmasOption: ["journal_mode":"DELETE"],
        NSMigratePersistentStoresAutomaticallyOption:0,
        NSInferMappingModelAutomaticallyOption:0]
    var _localStore: NSPersistentStore?
    do {
        _localStore = try self.coordinator.addPersistentStoreWithType
            (NSSQLiteStoreType, configuration: nil, URL: self.localStoreURL, options: options)
        return _localStore
    } catch {
        return nil
    }

}()(
```

Update Groceries as follows to ensure a `localStore` is not returned when the migration manager is used and a manual migration is required:

1. Replace the `localStore` variable in `CDHelper.swift` with the code from Listing 3.13.

For progress to be shown to the user, the interface needs to be ready before a migration is triggered. This means that the first call to anything Core Data related should be made from an existing view after it has loaded. To demonstrate the migration process, a small amount of code needs to be applied to the existing table view. The only table view in the application so far is the Prepare table view, where the user adds items to the shopping list. Listing 3.14 shows the minimal code involved that triggers a store model migration. Note that the table view won’t be configured to display anything until later in the book.

Listing 3.14  The Prepare Table View Controller (PrepareTVC.swift)

```swift
import UIKit

class PrepareTVC: UITableViewController {

    override func viewDidLoad() {
```

```swift
 Update Groceries as follows to ensure a `localStore` is not returned when the migration manager is used and a manual migration is required:

1. Replace the `localStore` variable in `CDHelper.swift` with the code from Listing 3.13.

For progress to be shown to the user, the interface needs to be ready before a migration is triggered. This means that the first call to anything Core Data related should be made from an existing view after it has loaded. To demonstrate the migration process, a small amount of code needs to be applied to the existing table view. The only table view in the application so far is the Prepare table view, where the user adds items to the shopping list. Listing 3.14 shows the minimal code involved that triggers a store model migration. Note that the table view won’t be configured to display anything until later in the book.

Listing 3.14  The Prepare Table View Controller (PrepareTVC.swift)

```swift
import UIKit

class PrepareTVC: UITableViewController {

    override func viewDidLoad() {
```
super.viewDidLoad()
{
    super.viewDidLoad()
    // Trigger Demo Code
    if let appDelegate = UIApplication.sharedApplication().delegate as? AppDelegate {
        appDelegate.demo()
    }
}

override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)
    // Trigger Demo Code
    if let appDelegate = UIApplication.sharedApplication().delegate as? AppDelegate {
        appDelegate.demo()
    }
}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return 0
}

override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return 0
}

The PrepareTVC.swift code is the bare minimum code required to show a table view. You might notice that the view is configured to return no rows or sections, as the intent for the moment is just to show model migration progress.

Update Groceries as follows to implement PrepareTVC.swift:

1. Right-click the existing Groceries group and then select New Group.
2. Set the new group name to Table View Controllers.
3. Select the Table View Controllers group.
4. Click File > New > File....
5. Create a new iOS > Source > Swift File and then click Next.
6. Set the filename to PrepareTVC and ensure the Groceries target is checked.
7. Ensure the Groceries project directory is open and then click Create.
8. Replace the contents of PrepareTVC.swift with the code from Listing 3.14.
9. Select Main.storyboard.
10. Set the Custom Class of the Table View Controller to PrepareTVC using Identity Inspector (Option+⌘+3) while the Table View Controller is selected.
11. Remove the call to CDHelper.shared from the application:didFinishLaunchingWithOptions function of AppDelegate.swift. This code would otherwise trigger a migration before the user interface was ready.
12. Remove the call to `demo()` from the `application:applicationDidBecomeActive` function of `AppDelegate.swift`. This code would otherwise trigger a migration before the user interface was ready.

Almost everything is in place to perform a manual migration; however, a new managed object model and mapping model are required to show what attributes map to where.

Update Groceries as follows to prepare the new model:

1. Add a model version called **Model 4** based on **Model 3**.
2. Set **Model 4** as the current model.
3. Select **Model 4.xcdatamodel**.
4. Delete the **Amount** entity.
5. Add a new entity called **Unit** with a **String** attribute called **name**.
6. Set the default value of the **name** attribute to **New Unit**.
7. Create an **NSManagedObject** subclass of the **Unit** entity. When it comes time to save the class file, don’t forget to check the **Groceries** target and ensure that the **Data Model** group is selected.
8. Create a new mapping model with **Model 3** as the source and **Model 4** as the target. When it comes time to save the mapping model file, don’t forget to check the **Groceries** target and save the mapping model as **Model3toModel4**.
9. Select **Model3toModel4.xcmappingmodel**.
10. Select the **Unit** entity mapping.
11. Set the **Source** of the **Unit** entity to **Amount** and the **Value Expression** of the **name** destination attribute to `$source.xyz`. You should see the **Unit** entity mapping automatically renamed to **AmountToUnit**, as shown in Figure 3.12.

![Figure 3.12 Mapping model for AmountToUnit](image)
You’re almost ready to perform a migration; however, the fetch request in the demo function still refers to the old Amount entity. Listing 3.15 shows an updated version of this function.

Listing 3.15  Fetching Test Unit Data (AppDelegate.swift demo)

```swift
func demo () {
    let context = CDHelper.shared.context
    let request = NSFetchRequest(entityName: "Unit")
    request.fetchLimit = 50
    do {
        if let units = try context.executeFetchRequest(request) as? [Unit] {
            for unit in units {
                print("Fetched Unit Object \(unit.name!)")
            }
        }
    } catch {
        print("ERROR executing a fetch request: \(error)")
    }
}
```

Update Groceries as follows to refer to the Unit entity instead of the Amount entity:

1. Replace the demo function of AppDelegate.swift with the code shown in Listing 3.15. This code just fetches 50 Unit objects from the persistent store.

The migration manager is finally ready! Run the application and pay close attention! You should see the migration manager flash before your eyes, alerting you to the progress of the migration. The progress is also shown in the console log (see Figure 3.13).

![Migration Progress](image.png)

Figure 3.13  Visible migration progress
Examine the contents of the ZUNIT table in the `LocalStore.sqlite` file using the techniques discussed in Chapter 2. The expected result is shown in Figure 3.14.

![Figure 3.14 Successful use of migration manager](image)

If you reproduced the results shown in Figure 3.14, give yourself a pat on the back because you successfully implemented three types of model migration! The rest of the book uses lightweight migrations, so it needs to be re-enabled. Before you continue, close the SQLite Database Browser.

Update Groceries as follows to re-enable lightweight migration:

1. Set `useMigrationManager` to `false` in the `localStore` variable of `CDHelper.swift`.
2. Set the `NSMigratePersistentStoresAutomaticallyOption` option in the `localStore` variable of `CDHelper.swift` to `true` by changing the `0` to a `1`.
3. Set the `NSInferMappingModelAutomaticallyOption` option in the `localStore` variable of `CDHelper.swift` to `true` by changing the `0` to a `1`.
4. Comment out the call to `migrateStoreIfNecessary` from the `setupCoreData` function of `CDHelper.swift`.
5. Replace the code in the `demo` function of `AppDelegate.swift` with a call to `CDHelper.shared`. This ensures that the Core Data stack is set up without a reliance on particular entities.
The old mapping models and `NSManagedObject` subclasses of entities that don’t exist anymore are no longer needed. Although you could remove them, leave them in the project for reference sake.

**Summary**

You’ve now experienced lightweight migration, default migration, and using a migration manager to display progress. You should now be able to make an informed decision when determining between migration options for your own applications. Don’t forget that the only migration option for iCloud-enabled Core Data applications is lightweight migration. Adding model versions should now be a familiar procedure because the model has changed several times already.

**Exercises**

Why not build on what you’ve learned by experimenting?

1. Set the current model version to **Model 3** and run the application. It should not throw an error because the downgrade of data is inferred automatically. Note that this is only because `NSInferMappingModelAutomaticallyOption` has been re-enabled. In reality, you would need a **Model4toModel3** mapping model to map attributes properly.

2. Examine the contents of the ZAMOUNT table in `LocalStore.sqlite` and you’ll notice something critical: Where has all the data gone? There was no mapping model, so all the ZUNIT data was lost during the downgrade!

3. Set the current model to **Model 4** and then run the application to trigger an automatic lightweight migration. This should be fast because there is no data in the store.