More iPhone Development with Swift

Exploring the iOS SDK

David Mark | Jayant Varma | Jeff LaMarche | Alex Horovitz | Kevin Kim

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

About the Authors...xix

About the Technical Reviewer xxi

Acknowledgments xxiii

■Chapter 1: Here We Go Round Again.. 1

■Chapter 2: Core Data: What, Why, and How 9

■Chapter 3: A Super Start: Adding, Displaying, and Deleting Data........................ 43

■Chapter 4: The Devil in the Detail View 89

■Chapter 5: Preparing for Change: Migrations and Versioning........................... 127

■Chapter 6: Custom Managed Objects 137

■Chapter 7: Relationships, Fetched Properties, and Expressions 171

■Chapter 8: Behind Every iCloud 219

■Chapter 9: Peer-to-Peer Using Multipeer Connectivity.................................... 239

■Chapter 10: Map Kit.. 281

■Chapter 11: Messaging: Mail, Social, and iMessage 311

■Chapter 12: Media Library Access and Playback.. 329

■ Chapter 13: Lights, Camera, and Action 383
■ Chapter 14: Interface Builder and Storyboards 405
■ Chapter 15: Unit Testing, Debugging, and Instruments 425
■ Chapter 16: The Road Goes Ever On 453

Index 459

Chapter 1

Here We Go Round Again

So, you're still creating iPhone applications, huh? Great! iOS and the App Store have enjoyed tremendous success, fundamentally changing the way mobile applications are delivered and completely changing what people expect from their mobile devices. Since the first release of the iOS Software Development Kit (SDK) way back in March 2008, Apple has been busily adding new functionality and improving what was already there. It's no less exciting a platform than it was back when it was first introduced. In fact, in many ways, it's more exciting because Apple keeps expanding the amount of functionality available to third-party developers like us.

Since the last release of this book, *More iOS 6 Development* (Apress, 2012), Apple has released a number of frameworks, tools, and services. These include, but aren't limited to, the following:

- *Core frameworks*: Core Motion, Core Telephony, Core Media, Core View, Core MIDI, Core Image, and Core Bluetooth
- *Utility frameworks*: Event Kit, Quick Look Framework, Assets Library, Image I/O, Printing, AirPlay, Accounts and Social Frameworks, Pass Kit, AVKit
- *Services and their frameworks*: iAds, Game Center, iCloud, Newsstand
- *Developer-centric enhancements*: Blocks, Grand Central Dispatch (GCD), Storyboards, Collection Views, UI State Preservation, Auto Layout, UIAutomation

Obviously, there are too many changes to cover completely in a single book. But we'll try our best to make you comfortable with the ones that you'll most likely need to know.

What This Book Is

This book is a guide to help you continue down the path to creating better iOS applications. In *Beginning iPhone Development with Swift*, the goal was to get you past the initial learning curve and to help you get your arms around the fundamentals of building your first iOS applications. In this book, we're assuming you already know the basics. So, in addition to showing you how to use several of the new iOS APIs, we're also going to weave in some more advanced techniques that you'll need as your iOS development efforts grow in size and complexity.

In *Beginning iPhone Development with Swift*, every chapter was self-contained, each presenting its own unique project or set of projects. We'll be using a similar approach in the second half of this book, but in Chapters 2 through 8, we'll focus on a single, evolving Core Data application. Each chapter will cover a specific area of Core Data functionality as we expand the application. We'll also be strongly emphasizing techniques that will keep your application from becoming unwieldy and hard to manage as it gets larger.

What You Need To Know

This book assumes you already have some programming knowledge and that you have a basic understanding of the iOS SDK, either because you've worked through *Beginning iPhone Development with Swift* or because you've gained a similar foundation from other sources. We assume you've experimented a little with the SDK, perhaps written a small program or two on your own, and have a general feel for Xcode. You might want to quickly review *Beginning iPhone Development with Swift*.

COMPLETELY NEW TO IOS?

If you are completely new to iOS development, there are other books you probably should read before this one. If you don't already understand the basics of programming and the syntax of the C language, you should check out *Learn C on the Mac for OS X and iOS* by David Mark and James Bucanek, which is a comprehensive introduction to the C language for Macintosh programmers.

If you already understand C but don't have any experience programming with objects, check out *Learn Objective-C on the Mac*, an excellent and approachable introduction to Objective-C by Mac programming experts Scott Knaster, Waqar Malik, and Mark Dalrymple.

If you also need to learn Swift, there is a book for that too; you can refer to *Learn Swift on the Mac* by Waqar Malik

There is a comprehensive list of resources in Chapter 16 of this book in case you want to read and learn more before you continue with this book.

What You Need Before You Can Begin

Before you can write software for iOS devices, you need a few things. For starters, you need an Intel-based Macintosh running Yosemite (Mac OS X 10.10 or newer). Any Macintosh computer—laptop or desktop—that has been released since 2009 should work just fine, but make sure your machine is Intel-based and is capable of running Yosemite.

This may seem obvious, but you'll also need an iPhone (5S/5C or newer) or an iPad (iPad 2 or newer) capable of running iOS 8.*x*. While much of your code can be tested using the iPhone/iPad simulator, not all programs will run in the simulator. And you'll want to thoroughly test any application you create on an actual device before you ever consider releasing it to the public.

Finally, you'll need to sign up to become a Registered iOS Developer. If you're already a Registered iOS Developer, go ahead and download the latest and greatest iPhone development tools; then skip ahead to the next section.

If you're new to Apple's Registered iOS Developer programs, navigate to `http://developer.apple.com/ios/`, which will bring you to a page similar to that shown in Figure 1-1. Just below the iOS Dev Center banner, on the right side of the page, you'll find links labeled Log in and Register. Click the Register link. On the page that appears, click the Continue button. Follow the sequence of instructions to use your existing Apple ID or create a new one.

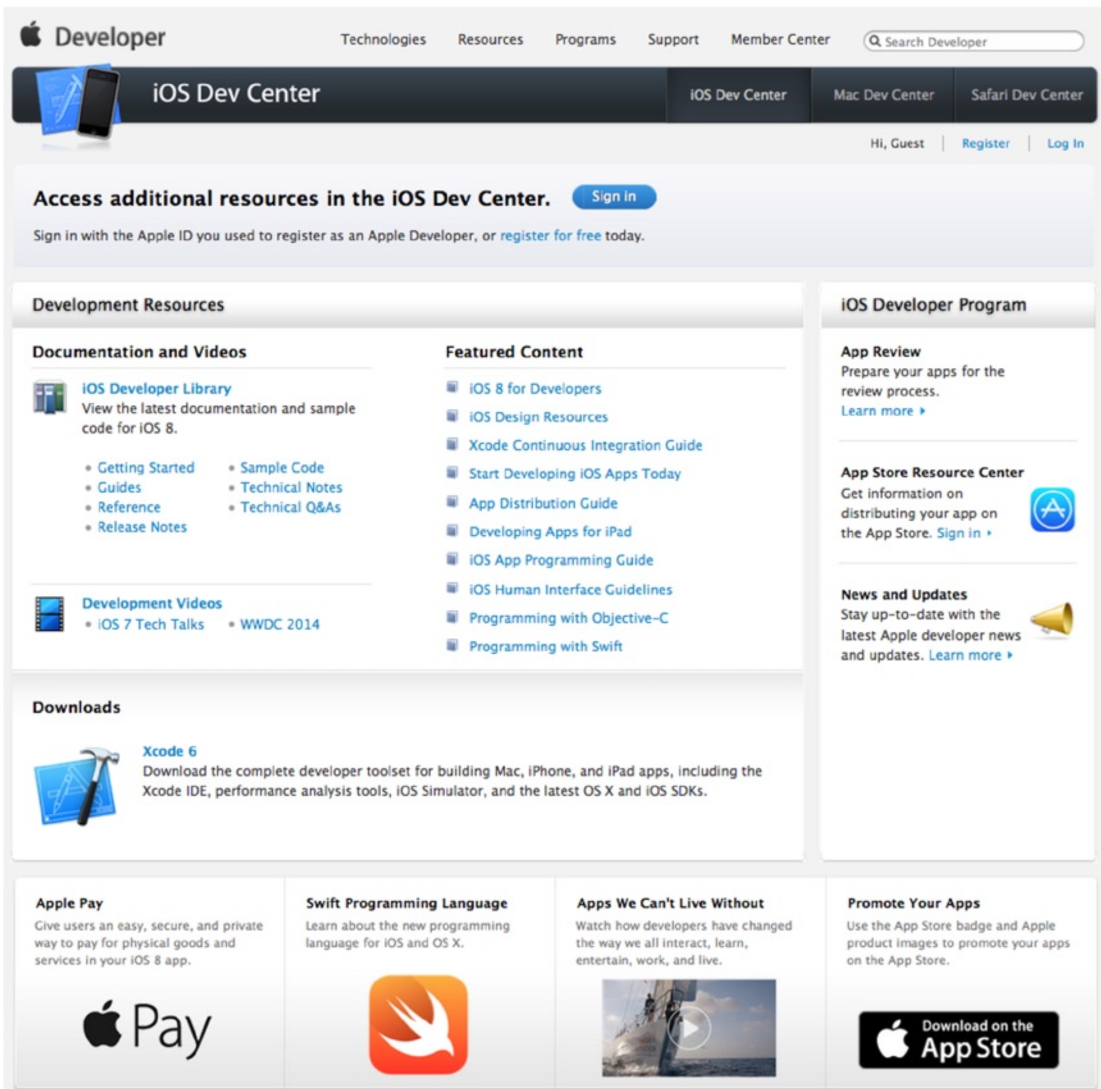

***Figure 1-1.** Apple's iOS Dev Center web site*

At some point, as you register, you'll be given a choice of several paths, all of which will lead you to the SDK download page. The three choices are free, commercial, and enterprise. All three options give you access to the iOS SDK and Xcode, Apple's integrated development environment (IDE). Xcode includes tools for creating and debugging source code, compiling applications, and performance-tuning the applications you've written. Please note that although you get at Xcode through the developer site, your Xcode distribution will be made available to you via the App Store.

The free option is, as its name implies, free. It lets you develop iOS apps that run on a software-only simulator but does not allow you to download those apps to your iPhone, iPod touch, or iPad, nor sell your apps on Apple's App Store. In addition, some programs in this book will run only on your device, not in the simulator, which means you will not be able to run them if you choose the free solution. That said, the free solution is a fine place to start if you don't mind learning without doing for those programs that won't run in the simulator.

The other two options are to sign up for an iOS Developer Program, either the Standard (commercial) Program or the Enterprise Program. The Standard Program costs $99. It provides a host of development tools and resources, technical support, distribution of your application via Apple's App Store, and, most important, the ability to test and debug your code on an iPhone rather than just in the simulator. The Enterprise Program, which costs $299, is designed for companies developing proprietary, in-house applications for the iPhone, iPod touch, and iPad. For more details on these two programs, check out `http://developer.apple.com/programs/`. (Prices are in USD and might vary based on the country that you reside in along with the formalities that Apple might require to enroll in the developer program.)

> **Note** If you are going to sign up for the Standard or Enterprise Program, you should go do it right now. It can take a while to get approved, and you'll need that approval to be able to run applications on your iPhone. Don't worry, though—the projects in the early chapters of this book will run just fine on the iPhone simulator.

Because iOS devices are connected mobile devices that utilize a third party's wireless infrastructure, Apple has placed far more restrictions on iOS developers than it ever has on Macintosh developers, who are able to write and distribute programs with absolutely no oversight or approval from Apple except when selling on the App Store. Apple is not doing this to be mean but rather to minimize the chances of people distributing malicious or poorly written programs that could degrade performance on the shared network. It may seem like a lot of hoops to jump through, but Apple has gone through quite an effort to make the process as painless as possible.

What's In This Book

As we said earlier, Chapters 2 through 7 of this book focus on Core Data, Apple's primary persistence framework. The rest of the chapters cover specific areas of functionality either that are new with iOS SDK or that were simply too advanced to include in *Beginning iPhone Development with Swift*.

Here is a brief overview of the chapters that follow:

- *Chapter 2, "Core Data: What, Why, and How"*: In this chapter, we'll introduce you to Core Data. You'll learn why Core Data is a vital part of your iPhone development arsenal. We'll dissect a simple Core Data application and show you how all the individual parts of a Core Data–backed application fit together.
- *Chapter 3, "A Super Start*: *Adding, Displaying, and Deleting Data"*: Once you have a firm grasp on Core Data's terminology and architecture, you'll learn how to do some basic tasks, including inserting, searching for, and retrieving data.
- *Chapter 4, "The Devil in the Detail View"*: In this chapter, you'll learn how to let your users edit and change the data stored by Core Data. We'll explore techniques for building generic, reusable views so you can leverage the same code to present different types of data.
- *Chapter 5, "Preparing for Change*: *Migrations and Versioning"*: Here, we'll look at Apple tools that you can use to change your application's data model, while still allowing your users to continue using their data from previous versions of your application.
- *Chapter 6, "Custom Managed Objects"*: To really unlock the power of Core Data, you can subclass the class used to represent specific instances of data. In this chapter, you'll learn how to use custom managed objects and see some benefits of doing so.
- *Chapter 7, "Relationships, Fetched Properties, and Expressions"*: In this final chapter on Core Data, you'll learn about some mechanisms that allow you to expand your applications in powerful ways. You'll refactor the application you built in the previous chapters so that you don't need to add new classes as you expand your data model.
- *Chapter 8, "Behind Every iCloud"*: The iCloud Storage APIs are among the coolest features of iOS. The iCloud APIs will let your apps store documents and key-value data in iCloud. iCloud will wirelessly push documents to a user's device automatically and update the documents when changed on any device—automatically. You'll enhance your Core Data application to store information on iCloud.
- *Chapter 9, "Peer-to-Peer Over Bluetooth Using Multipeer Connectivity"*: The Multipeer Connectivity framework makes it easy to create programs that communicate over Bluetooth and WiFi, such as multiplayer games for the iPhone and iPad. You'll explore Multipeer Connectivity by building a simple two-player game.
- *Chapter 10, "MapKit"*: This chapter explores another great new piece of functionality added to the iOS SDK: an enhanced CoreLocation. This framework now includes support for both forward and reverse geocoding location data. You will be able to convert back and forth between a set of map coordinates and information about the street, city,

and country (and so on) at that coordinate. Plus, you'll explore how all this interoperates with enhanced MapKit.

- *Chapter 11, "Messaging: Mail, Social, and iMessage"*: Your ability to get your message out has gone beyond e-mail. In this chapter, we'll take you through the core options of Mail, the Social Framework, and iMessage, and you'll see how to leverage each appropriately.
- *Chapter 12, "Media Library Access and Playback"*: It's now possible to programmatically get access to your users' complete library of audio tracks stored on their iPhone or iPod touch. In this chapter, you'll look at the various techniques used to find, retrieve, and play music and other audio tracks.
- *Chapter 13, "Lights, Camera and Action"*: In this chapter, you'll be taking a detailed look into the AVFoundation framework, which provides a standard set of APIs and classes for iOS applications to play audio and video and even capture the same. In addition to the basic interfaces of this framework, you will utilize some additions for managing capturing, saving images, and audio.
- *Chapter 14, "Interface Builder and Storyboards"*: The new additions to Interface Builder allow you to have live previews and create custom controls to use in your projects. You will create custom transitions between your views and view controllers.
- *Chapter 15, "Unit Testing, Debugging, and Instruments"*: No program is ever perfect. Bugs and defects are a natural part of the programming process. In this chapter, you'll learn various techniques for preventing, finding, and fixing bugs in iOS SDK programs.
- *Chapter 16, "The Road Goes Ever On. . ."*: Sadly, every journey must come to an end. We'll wrap up this book with fond farewells and some resources we hope you'll find useful.

iOS is an incredible computing platform, an ever-expanding frontier for your development pleasure. In this book, we're going to take you further down the iPhone development road, digging deeper into the SDK, touching on new and, in some cases, more advanced topics.

Read the book and be sure to build the projects yourself—don't just copy them from the archive and run them once or twice. You'll learn most by doing. Make sure you understand what you did, and why, before moving on to the next project. Don't be afraid to make changes to the code. Experiment, tweak the code, and observe the results. Rinse and repeat.

Got your iOS SDK installed? Turn the page, put on some iTunes, and let's go. Your continuing journey awaits.

Chapter 2

Core Data: What, Why, and How

Core Data is a framework and set of tools that allow you to save (or persist) your application's data to an iOS device's file system automatically. Core Data is an implementation of something called *object-relational mapping* (ORM). This is just a fancy way of saying that Core Data allows you to interact with your Swift objects without having to worry about how the data from those objects is stored and retrieved from persistent data stores such as relational databases (such as SQLite) or flat files.

Core Data can seem like magic when you first start using it. Core Data objects are, for the most part, handled just like plain old objects, and they seem to know how to retrieve and save themselves automagically. You won't create SQL strings or make file management calls, ever. Core Data insulates you from some complex and difficult programming tasks, which is great for you. By using Core Data, you can develop applications with complex data models much, much faster than you could using straight SQLite, object archiving, or flat files.

Technologies that hide complexity the way Core Data does can encourage "voodoo programming," that most dangerous of programming practices where you include code in your application that you don't necessarily understand. Sometimes that mystery code arrives in the form of a project template. Or, perhaps you download a utilities library that does a task for you that you just don't have the time or expertise to do for yourself. That voodoo code does what you need it to do, and you don't have the time or inclination to step through it and figure it out, so it just sits there, working its magic...until it breaks. As a general rule, if you find yourself with code in your own application that you don't fully understand, it's a sign you should go do a little research, or at least find a more experienced peer to help you get a handle on your mystery code.

The point is that Core Data is one of those complex technologies that can easily turn into a source of mystery code that will make its way into many of your projects. Although you don't need to know exactly how Core Data accomplishes everything it does, you should invest some time and effort into understanding the overall Core Data architecture.

This chapter starts with a brief history of Core Data, and then it dives into a Core Data application. By building a Core Data application with Xcode, you'll find it much easier to understand the more complex Core Data projects you'll find in the following chapters.

A Brief History of Core Data

Core Data has been around for quite some time, but it became available on iOS with the release of iPhone SDK 3.0. Core Data was originally introduced with Mac OS X 10.4 (Tiger), but some of the DNA in Core Data actually goes back about 15 years to a NeXT framework called Enterprise Objects Framework (EOF), which was part of the toolset that shipped with NeXT's WebObjects web application server.

EOF was designed to work with remote data sources, and it was a pretty revolutionary tool when it first came out. Although there are now many good ORM tools for almost every language, when WebObjects was in its infancy, most web applications were written to use handcrafted SQL or file system calls to persist their data. Back then, writing web applications was incredibly time- and labor-intensive. WebObjects, in part because of EOF, cut the development time needed to create complex web applications by an order of magnitude.

In addition to being part of WebObjects, EOF was also used by NeXTSTEP, which was the predecessor to Cocoa. When Apple bought NeXT, the Apple developers used many of the concepts from EOF to develop Core Data. Core Data does for desktop applications what EOF had previously done for web applications: it dramatically increases developer productivity by removing the need to write file system code or interact with an embedded database.

Let's start building your Core Data application.

Creating a Core Data Application

Fire up Xcode and create a new Xcode project. There are many ways to do this. When you start Xcode, you may get the Xcode startup window (Figure 2-1). You can just click "Create a New Xcode project." Or you can select **File ➤ New ➤ Project**. Or you can use the keyboard shortcut ⇧⌘N—whatever floats your boat. Going forward, we're going to mention the options available in the Xcode window or the menu options, but we won't use the keyboard shortcut. If you know and prefer the keyboard shortcuts, feel free to use them. Let's get back to building your app.

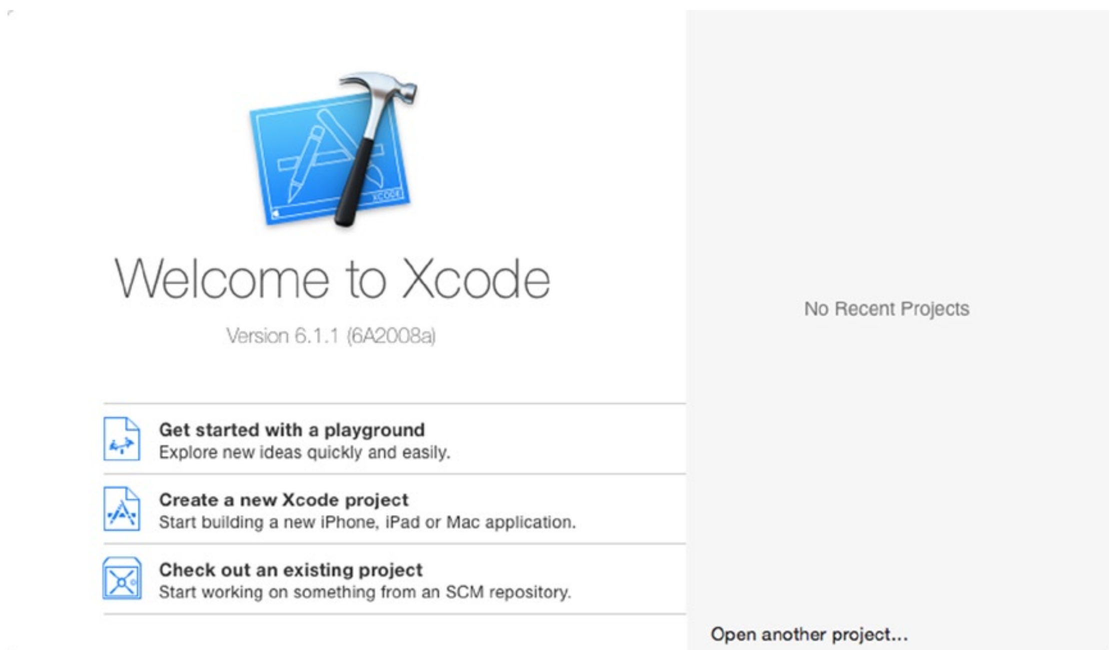

***Figure 2-1.** Xcode startup window*

Xcode will open a project workspace and display the Project Template sheet (Figure 2-2). On the left are the possible template headings: iOS and OS X. Each heading has a bunch of template groups. Select the Application template group under the iOS heading and then select Master-Detail Application template on the right. On the bottom right, there's a short description of the template. Click the Next button to move the next sheet.

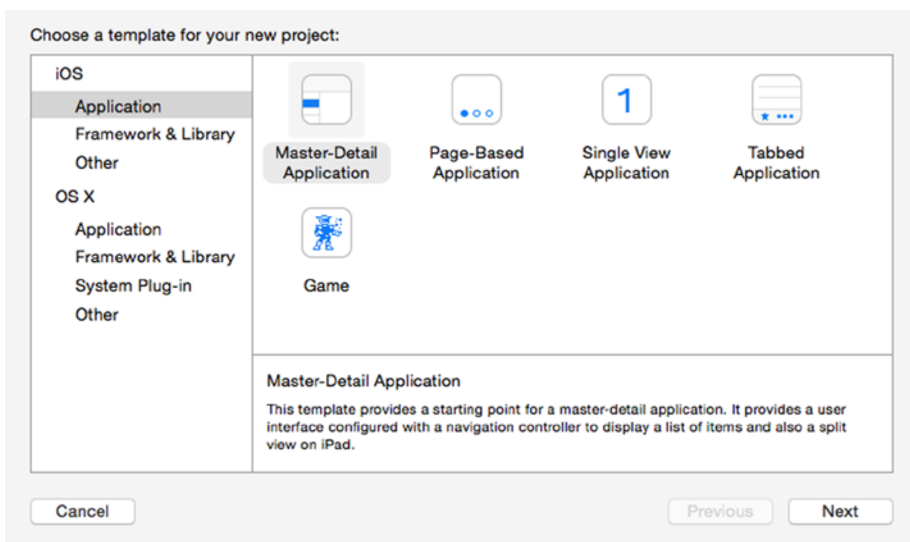

***Figure 2-2.** Project Template sheet*

The next sheet is the Project Configuration sheet (Figure 2-3). You'll be asked to provide a product name; enter **CoreDataApp**. The Organization Name and Company Identifier fields will be set automatically by Xcode; by default these will read MyCompanyName and com.mycompanyname. You can change these to whatever you like, but for Company Identifier, Apple recommends using the reverse domain name style (such as com.oz-apps).

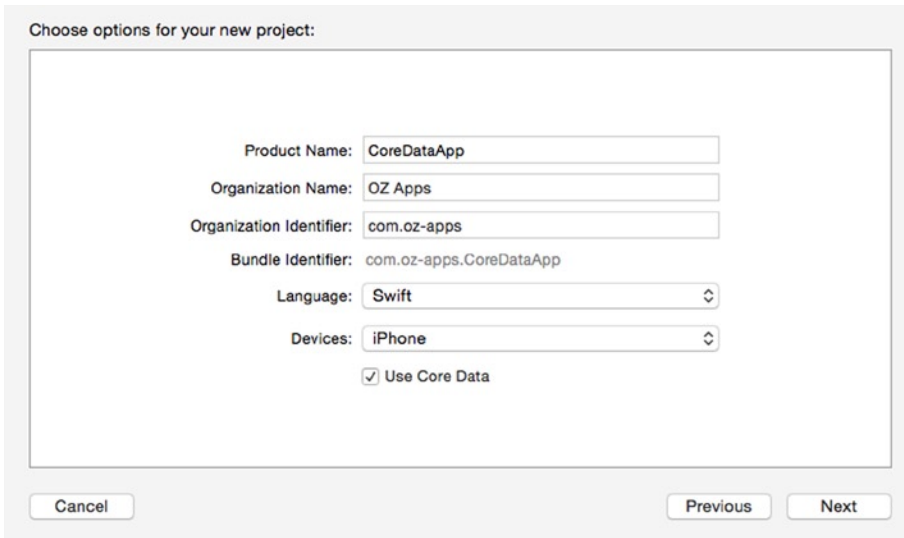

***Figure 2-3.** Project Configuration sheet*

Note that the Bundle Identifier field is not editable; rather, it's populated by the values from the Company Identifier and Product Name fields.

The Devices drop-down field lists the possible target devices for this project: iPad, iPhone, or Universal. The first two are self-explanatory. Universal is for applications that will run on both the iPad and iPhone. It's a blessing and a curse to have to a single project that can support both iPads and iPhones. But for the purposes of this book, you'll stick with iPhone. You obviously want to use Core Data, so select its check box. Finally, make sure that you have Swift selected as the language.

Click Next and choose a location to save your project (Figure 2-4). The check box on the bottom will set up your project to use Git (`www.git-scm.com`), a free, open source version control system. We won't discuss it, but if you don't know about version control or Git, we suggest you get familiar with them. Click Create. Xcode should create your project, and it should look like Figure 2-5.

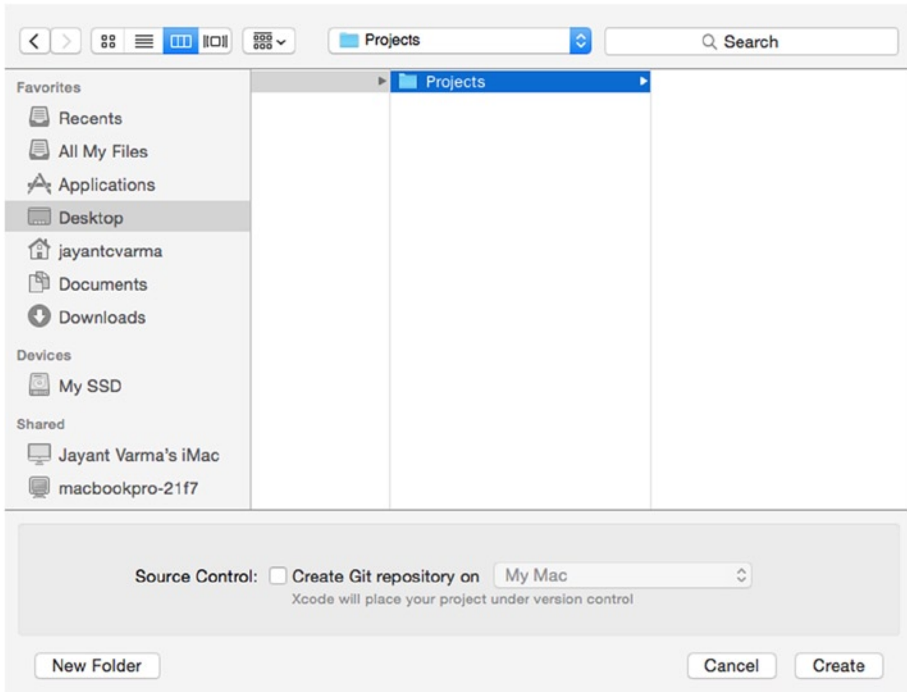

***Figure 2-4.** Choose a location to put your project*

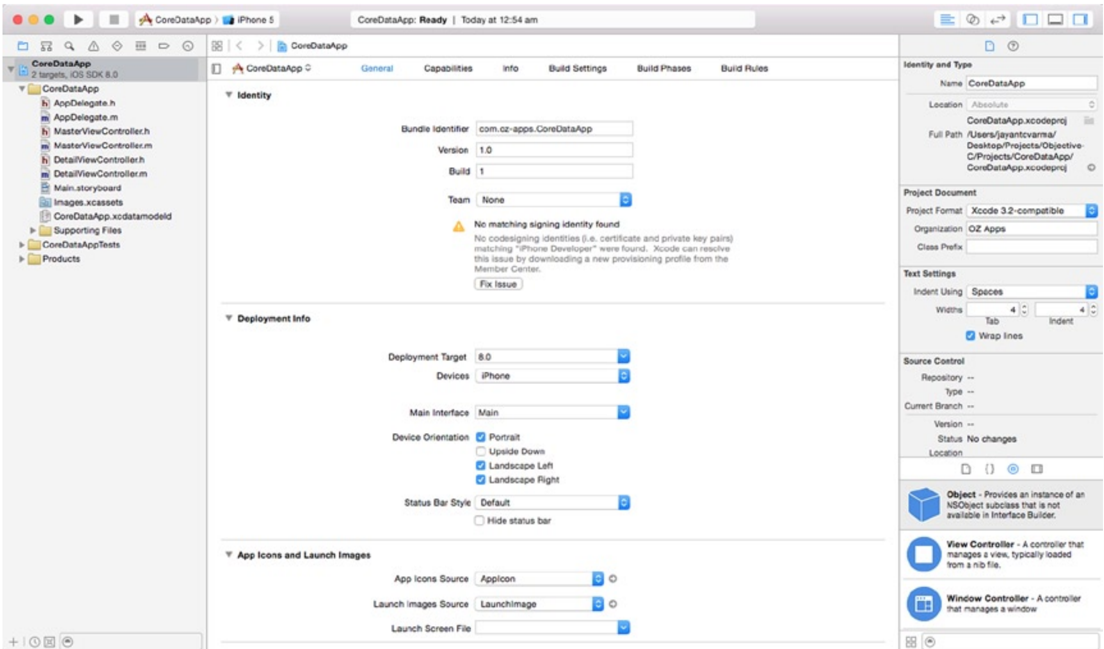

***Figure 2-5.** Voilà, your project is ready!*

Build and run the application. Either press the Run button on the toolbar or select **Product ➤ Run**. The simulator should appear. Press the Add (+) button in the upper right. A new row will insert into the table that shows the exact date and time the `Add` button was *pressed* (Figure 2-6). You can also use the `Edit` button to delete rows. Exciting, huh?

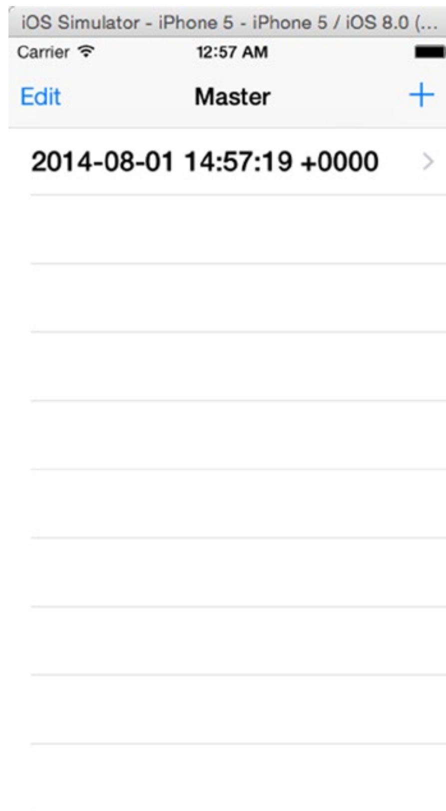

Figure 2-6. *CoreDataApp in action*

Under the hood of this simple application, a lot is happening. Think about it: without adding a single class or any code to persist data to a file or interact with a database, pressing the Add button created an object, populated it with data, and saved it to a SQLite database created for you automatically. There's plenty of free functionality here.

Now that you've seen an application in action, let's take a look at what's going on behind the scenes.

Core Data Concepts and Terminology

Like most complex technologies, Core Data has its own terminology that can be a bit confusing to newcomers. Let's break down the mystery and get your arms around Core Data's nomenclature.

Figure 2-7 shows a simplified, high-level diagram of the Core Data architecture. Don't expect it all to make sense now, but as you look at different pieces, you might want to refer to this diagram to cement your understanding of how they fit together.

There are five key concepts to focus on here. As you proceed through this chapter, make sure you understand each of the following:

- Data model
- Persistent store
- Persistent store coordinator
- Managed object and managed object context
- Fetch request

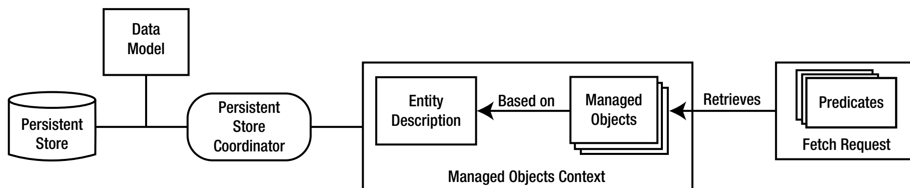

Figure 2-7. *The Core Data architecture*

Once again, don't let the names throw you. Follow along and you'll see how all these pieces fit together.

The Data Model

What is a data model? In an abstract sense, it's an attempt to define the organization of data and the relationship between the organized data components. In Core Data, the data model defines the data structure of objects, the organization of those objects, the relationships between those objects, and the behavior of those objects. Xcode allows you, via the model editor and inspector, to specify your data model for use in your application.

If you expand the CoreDataApp group in the Navigator content pane, you'll see a file called `CoreDataApp.xcdatamodel`. This file is the default data model for your project. Xcode created this file for you because you selected the Use Core Data check box in the Project Configuration sheet. Single-click `CoreDataApp.xcdatamodel` to bring up Xcode's model editor. Make sure the Utility pane is visible (it should be the third button on the View bar) and select the inspector. Your Xcode window should look like Figure 2-8.

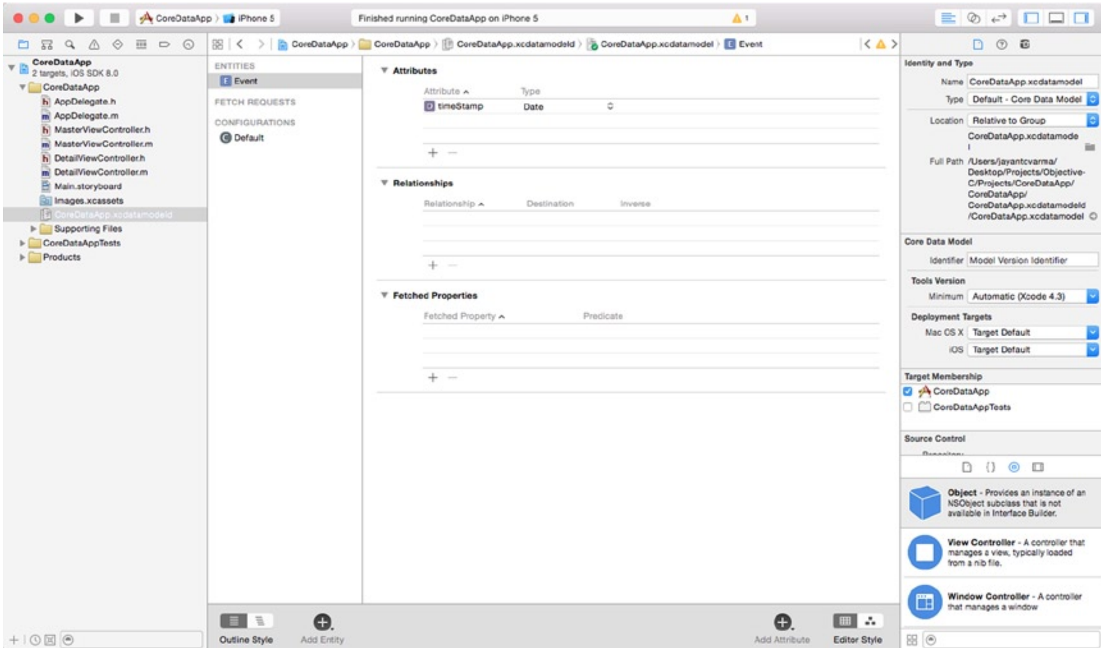

Figure 2-8. Xcode with the model editor and inspector

When you selected the data model file, `CoreDataApp.xcdatamodel`, the Editor pane changed to present the Core Data model editor (Figure 2-9). Along the top, the jump bar remains unchanged. Along the left, the gutter has been replaced by a wider pane, the Top-Level Components pane. The Top-Level Components pane outlines the entities, fetch requests, and configurations defined in the data model (we'll cover these in detail in a little bit). You can add a new entity by using the Add Entity button at the bottom of the Top-Level Components pane. Alternately, you can use the **Editor ➤ Add Entity** menu option. If you click and hold the Add Entity button, you will be presented with a pop-up menu of choices: Add Entity, Add Fetch Request, and Add Configuration. Whatever option you choose, the single-click behavior of the button will change to that component, and the label of the button will change to reflect this behavior. You can find the menu equivalents for adding fetch requests and configurations under the **Editor ➤ Add Entity** menu item.

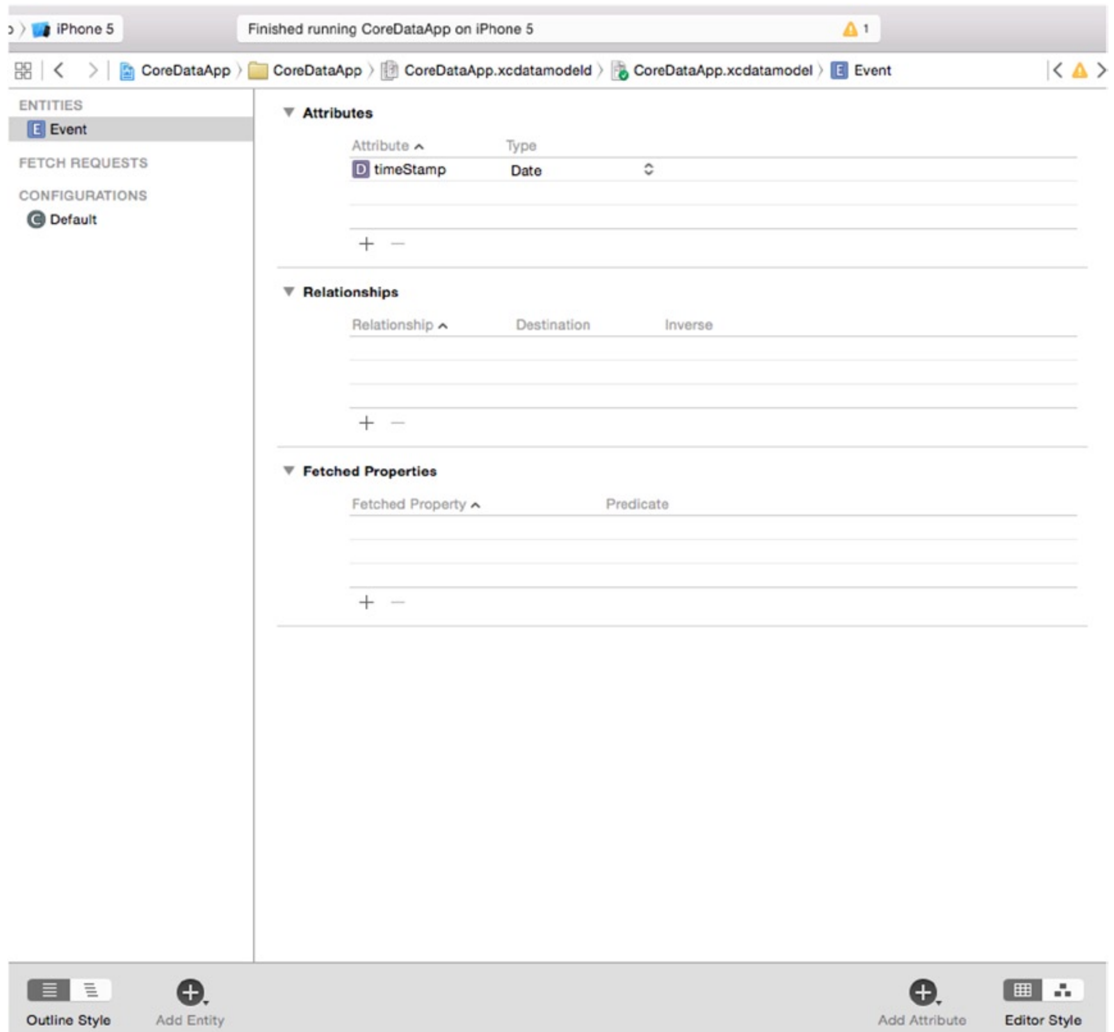

***Figure 2-9.** A close look at the model editor*

The Top-Level Components pane has two styles: list and hierarchical. You can toggle between these two styles by using the Outline Style selector group found at the bottom of the Top-Level Components pane. Switching styles with the CoreDataApp data model won't change anything in the Top-Level Components pane; there's only one entity and one configuration, so there's no hierarchy to be shown. If you had a component that depended on another component, you'd see the hierarchical relationship between the two with the hierarchical outline style.

The bulk of the Editor pane is taken up by the Detail editor. The Detail editor has two editor styles: table and graph. By default (and pictured in Figure 2-9), the Detail editor is in table style. You can toggle between these styles by using the Editor Style selector group on the bottom right of the Editor pane. Try it. You can see the difference in the two styles.

When you select an entity in the Top-Level Components pane, the Detail editor will display, in table style, three tables: `Attributes`, `Relationships`, and `Fetched Properties`. Again, we'll cover these in detail in a little bit. You can add a new attribute by using the Add Attribute button below the Detail editor. Similar to the Add Entity button, a click-and-hold will reveal a pop-up menu of choices: Add Attribute, Add Relationship, and Add Fetched Property. Again, the single-click behavior of this button will change depending on your choice, with its label reflecting that behavior. Under the Editor menu there are three menu items: Add Attribute, Add Relationship, and Add Fetched Property. These are active only when an entity is selected in the Top-Level Components pane.

If you switch the Detail editor to graph style, you'll see a large grid with a single rounded rectangle in the center. This rounded rectangle represents the entity in the Top-Level Components pane. The template you used for this project creates a single entity, Event. Selecting Event in the Top Level Components pane is the same as selecting the rounded rectangle in the graph view.

Try it. Click outside the entity in the Detail editor grid to deselect it and then click the Event line in the Top-Level Components pane. The entity in the graph view will also be selected. The Top Level Components pane and the graph view show two different views of the same entity list.

When unselected, the title bar and lines of the Event entity square should be pink. If you select the Event entity in the Top-Level Components pane, the Event entity in the Detail editor should change color to a blue, indicating it's selected. Now click anywhere on the Detail editor grid, outside the Event rounded square. The Event entity should be deselected in the Top Level Components pane and should change color in the Detail editor. If you click the Event entity in the Detail editor, it will be selected again. When selected, the Event entity should have a resize handle (or dot) on the left and right sides, allowing you to resize its width.

You are currently given the Event entity. It has a single attribute, named timeStamp, and no relationships. The Event entity was created as part of this template. As you design your own data models, you'll most likely delete the Event entity and create your own entities from scratch. A moment ago, you ran your Core Data sample application in the simulator. When you pressed the + icon, a new instance of Event was created. Entities, which we'll look at more closely in a few pages, replace the Swift data model class you would otherwise use to hold your data. We'll get back to the model editor in just a minute to see how it works. For now, just remember that the persistent store is where Core Data stores its data, and the data model defines the form of that data. Also remember that every persistent store has one, and only one, data model.

The inspector provides greater detail for the items selected in the model editor. Since each item could have a different view in the inspector, we'll discuss the details as we discuss the components and their properties. That being said, let's discuss the three top-level components: entities, fetch requests, and configurations.

Entities

An entity can be thought of as the Core Data analog to a Swift class declaration. In fact, when using an entity in your application, you essentially treat it like a Swift class with some Core Data–specific implementation. You use the model editor to define the properties that define your entity. Each entity is given a name (in this case, Event), which must begin with a capital letter. When you ran CoreDataApp earlier, every time you pressed the Add (+) button, a new instance of Event was instantiated and stored in the application's persistent store.

Make sure the Utility pane is exposed, and select the Event entity. Now look at the inspector in the Utility pane (make sure the inspector is showing by selecting the Inspector button in the Inspector selector bar). Note that the Inspector pane now allows you to edit or change aspects of the entity (Figure 2-10). We'll get to the details of the inspector later.

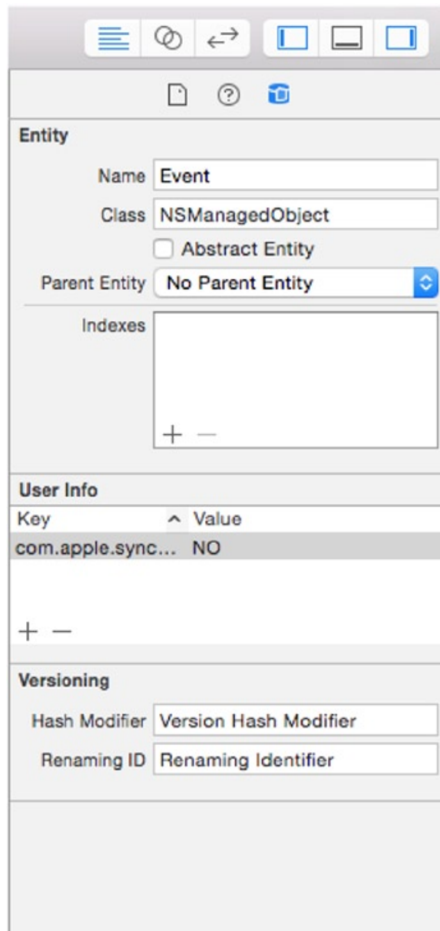

Figure 2-10. The inspector for the Event entity

Properties

While the Editor pane lists all the data model's entities, the Inspector pane allows you to "inspect" the properties that belong to the selected entity. An entity can consist of any number of properties. There are three different types of properties: attributes, relationships, and fetched properties. When you select an entity's property in the model editor, the property's details are displayed in the Inspector pane.

Attributes

The property that you'll use the most when creating entities is the attribute, which serves the same function in a Core Data entity as an instance variable does in a Swift class: they both hold data. If you look at your model editor (or at Figure 2-10), you'll see that the Event entity has one attribute named timeStamp. The timeStamp attribute holds the date and time when a given Event instance was created. In your sample application, when you click the + button, a new row is added to the table displaying a single Event's timeStamp.

Just like an instance variable, each attribute has a type. There are two ways to set an attribute's type. When the model editor is using the table style, you can change an attributes type in the `Attributes` table in the Detail editor (Figure 2-11). In your current application, the timeStamp attribute is set to the Date type. If you click the `Date` cell, you'll see a pop-up menu. That pop-up menu shows the possible attribute types. You'll look at the different attribute types in the next few chapters when you begin building your own data models.

***Figure 2-11.** Attributes table in the model editor, table style*

Make sure that the timeStamp attribute is still selected, and take a look at the inspector (Figure 2-12). Notice among the fields there is an Attribute Type field with a pop-up button. Click the button, and a pop-up menu will appear. It should contain the attribute's types you saw in the `Attribute` table. Make sure the attribute type is set to date.

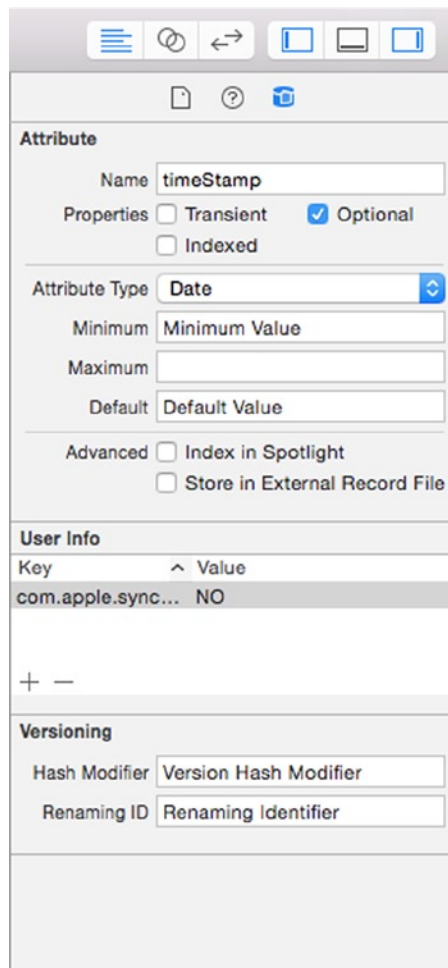

***Figure 2-12.** Inspector for the timeStamp attribute*

A date attribute, such as timeStamp, corresponds to an instance of NSDate. If you want to set a new value for a date attribute, you need to provide an instance of NSDate to do so. A string attribute corresponds to an instance of NSString, and most of the numeric types correspond to an instance of NSNumber.

> **Tip** Don't worry too much about all the other buttons, text fields, and check boxes in the model editor. As you make your way through the next few chapters, you'll get a sense of what each does.

Relationships

As the name implies, a relationship defines the associations between two different entities. In the template application, no relationships are defined for the Event entity. We'll begin discussing relationships in Chapter 7, but here's an example just to give you a sense of how they work.

Suppose you created an Employee entity and wanted to reflect each Employee's employer in the data structure. You could just include an employer attribute, perhaps an `NSString`, in the Employee entity, but that would be pretty limiting. A more flexible approach would be to create an Employer entity, and then create a relationship between the Employee and Employer entities.

Relationships can be to one or to many, and they are designed to link specific objects. The relationship from Employee to Employer might be a to-one relationship, if you assume that your Employees do not moonlight and have only a single job. On the other hand, the relationship from Employer to Employee is to many since an Employer might employ many Employees.

To put this in Swift terms, a to-one relationship is like using an instance variable to hold a pointer to an instance of another Swift class. A to-many relationship is more like using a pointer to a collection class like `NSMutableArray` or `NSSet`, which can contain multiple objects.

Fetched Properties

A fetched property is like a query that originates with a single managed object. For example, suppose you add a birthdate attribute to Employee. You might add a fetched property called sameBirthdate to find all Employees with the same date of birth as the current Employee.

Unlike relationships, fetched properties are not loaded along with the object. For example, if Employee has a relationship to Employer, when an Employee instance is loaded, the corresponding Employer instance will be loaded, too. But when an Employee is loaded, sameBirthdate is not evaluated. This is a form of lazy loading. You'll learn more about fetched properties in Chapter 7.

Fetch Requests

While a fetched property is like a query that originates with a single managed object, a fetch request is more like a class method that implements a canned query. For example, you might build a fetch request named `canChangeLightBulb` that returns a list of Employees who are taller than 80 inches (about 2 meters). You can run the fetch request any time you need a lightbulb changed. When you run it, Core Data searches the persistent store to find the current list of potential lightbulb-changing Employees.

You will create many fetch requests programmatically in the next few chapters, and you'll be looking at a simple one a little later in this chapter in the "Creating a Fetched Results Controller" section.

Configurations

A configuration is a set of entities. Different configurations may contain the same entity. Configurations are used to define which entities are stored in which persistent store. Most of the time, you won't need anything other than the default configuration. We won't cover using multiple configurations in this book. If you want to learn more, check the Apple Developer site or *Pro Core Data for iOS*.

The Data Model Class: NSManagedObjectModel

Although you won't typically access your application's data model directly, you should be aware of the fact that there is a class that represents the data model in memory. This class is called `NSManagedObjectModel`, and the template automatically creates an instance of `NSManagedObjectModel` based on the data model file in your project. Let's take a look at the code that creates it now.

In the Navigation pane, open the CoreDataApp group and `AppDelegate.swift`. In the Editor jump bar, click the last menu (it should read No Selection) to bring up a list of the methods in this class (see Figure 2-13). Select `managedObjectModel` in the Core Data Stack section, which will take you to the method that creates the object model based on the `CoreDataApp.xcdatamodel` file.

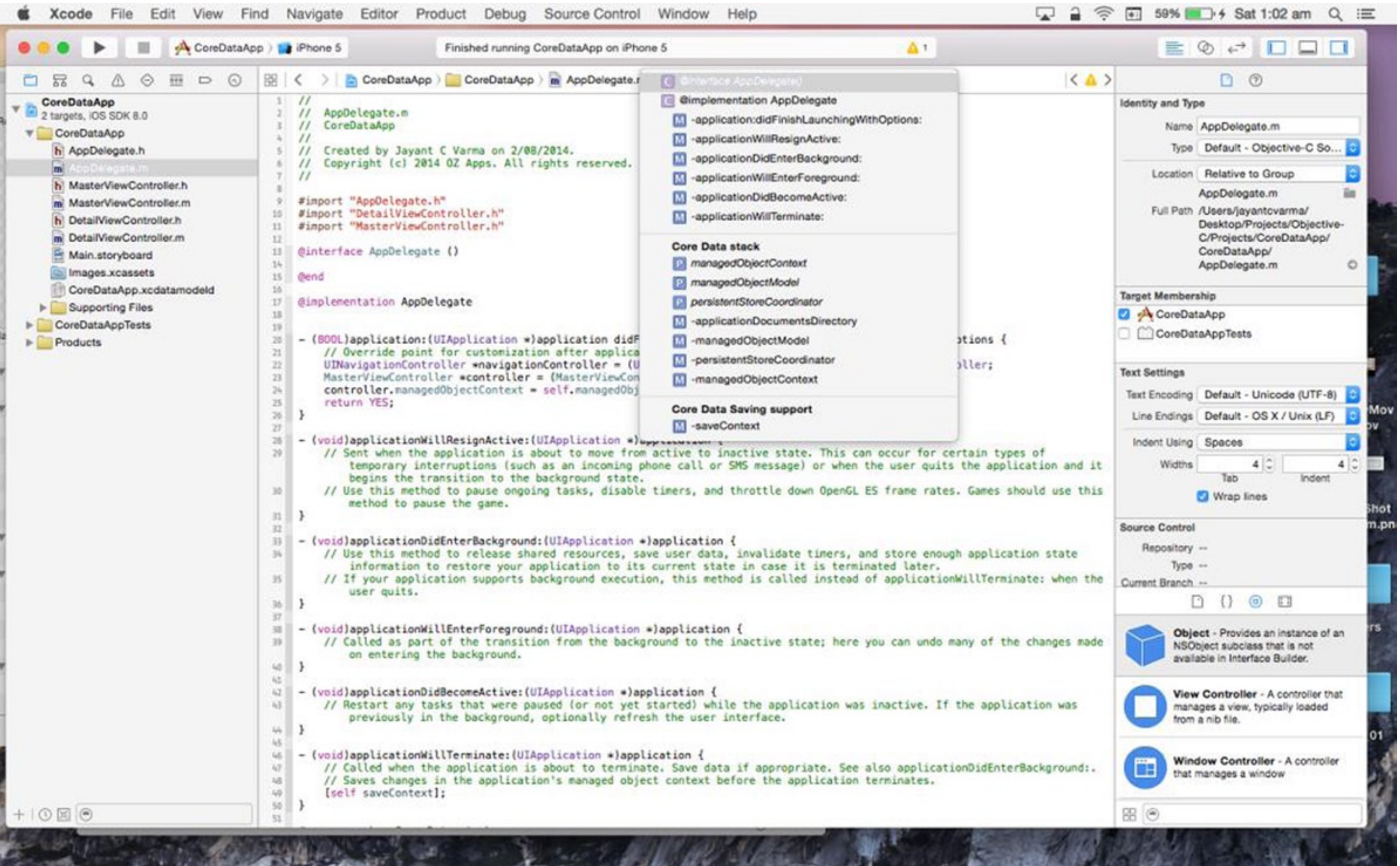

Figure 2-13. Setting the editor pane to show counterparts will allow you to see the declaration and implementation

The method should look like this:

```
lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not optional.
    It is a fatal error for the application not to be able to find and load its model.
    let modelURL = NSBundle.mainBundle().URLForResource("CoreDataApp",
    withExtension: "momd")
    return NSManagedObjectModel(contentsOfURL: modelURL)
}()
```

Using the lazy construct of Swift, the `managedObjectModel` variable is created of type `NSManagedObjectModel` when required. The variable is instantiated the first time it is called.

> **Tip** The data model class is called `NSManagedObjectModel` because, as you'll see a little later in the chapter, instances of data in Core Data are called *managed objects*.

While it creates the `managedObjectModel`, it also sets the `modelObject` with the contents of the data model called `CoreDataApp.momd` as the default with the following code:

```
Let modeURL = NSBundle.mainBundle().URLForResource("CoreDataApp", withExtension:"momd")
```

Remember how we said that a persistent store was associated with a single data model? Well, that's true, but it doesn't tell the whole story. You can combine multiple `.xcdatamodel` files into a single instance of `NSManagedObjectModel`, creating a single data model that combines all the entities from multiple files. If you are planning on having more than one model, you can use the `mergedModelFromBundles` class method of the `NSManagedObjectModel`.

This function will take all `.xcdatamodel` files that might be in your Xcode project and combine them into a single instance of `NSManagedObjectModel`:

```
return NSManagedObjectModel.mergedModelFromBundles(nil)
```

So, for example, if you create a second data model file and add it to your project, that new file will be combined with `CoreDataApp.xcdatamodel` into a single managed object model that contains the contents of both files. This allows you to split up your application's data model into multiple smaller and more manageable files.

The vast majority of iOS applications that use Core Data have a single persistent store and a single data model, so the default template code will work beautifully most of the time. That said, Core Data does support the use of multiple persistent stores. You could, for example, design your application to store some of its data in a SQLite persistent store and some of it in a binary flat file. If you find that you need to use multiple data models, remember to change the template code here to load the managed object models individually using `mergedModelFromBundles`.

The Persistent Store and Persistent Store Coordinator

The persistent store, which is sometimes referred to as a *backing store*, is where Core Data stores its data. By default, on iOS devices Core Data uses a SQLite database contained in your application's `Documents` folder as its persistent store. But this can be changed without impacting any of the other code you write by tweaking a single line of code. We'll show you the actual line of code to change in a few moments.

> **Caution** Do not change the type of persistent store once you have posted your application to the App Store. If you must change it for any reason, you will need to write code to migrate data from the old persistent store to the new one, or else your users will lose all of their data—something that will almost always make them quite unhappy.

Every persistent store is associated with a single data model, which defines the types of data that the persistent store can store.

The persistent store isn't actually represented by a Swift class. Instead, a class called `NSPersistentStoreCoordinator` controls access to the persistent store. In essence, it takes all the calls coming from different classes that trigger reads or writes to the persistent store and serializes them so that multiple calls against the same file are not being made at the same time, which could result in problems because of file or database locking.

As is the case with the managed object model, the template provides you with a method in the application delegate that creates and returns an instance of a persistent store coordinator. Other than creating the store and associating it with a data model and a location on disk (which is done for you in the template), you will rarely need to interact with the persistent store coordinator directly. You'll use high-level Core Data calls, and Core Data will interact with the persistent store coordinator to retrieve or save the data.

Let's take a look at the method that returns the persistent store coordinator. In `AppDelegate.swift`, select `persistentStoreCoordinator` from the function pop-up menu. Here's the method:

```
lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator? = {
    // The persistent store coordinator for the application. This implementation creates and
    return a coordinator, having added the store for the application to it. This property is
    optional since there are legitimate error conditions that could cause the creation of
    the store to fail.
    // Create the coordinator and store
    var coordinator: NSPersistentStoreCoordinator? = NSPersistentStoreCoordinator
    (managedObjectModel: self.managedObjectModel)
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent
    ("CoreDataApp.sqlite")
    var error: NSError? = nil
    var failureReason = "There was an error creating or loading the application's
    saved data."
```

```
    if coordinator!.addPersistentStoreWithType(NSSQLiteStoreType,
                                    configuration: nil,
                                              URL: url,
                                          options: nil,
                                            error: &error) == nil {
        coordinator = nil
        // Report any error we got.
        let dict = NSMutableDictionary()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's saved data"
        dict[NSLocalizedFailureReasonErrorKey] = failureReason
        dict[NSUnderlyingErrorKey] = error
        error = NSError.errorWithDomain("YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and terminate. You should
        not use this function in a shipping application, although it may be useful during
        development.
        NSLog("Unresolved error \(error), \(error!.userInfo)")
        abort()
    }

    return coordinator
}()
```

As with the managed object model, this `persistentStoreCoordinator` accessor method uses lazy loading and doesn't instantiate the persistent store coordinator until the first time it is accessed. It is prefixed with the lazy keyword. Then it creates a path to a file called `CoreDataApp.sqlite` in the `Documents` directory in your application's sandbox. The template will always create a filename based on your project's name. If you want to use a different name, you can change it here, though it generally doesn't matter what you call the file since the user will never see it.

> **Caution** If you do decide to change the filename, make sure you don't change it after you've posted your application to the App Store, or else future updates will cause your users to lose all of their data.

Take a look at this line of code:

```
if coordinator!.addPersistentStoreWithType(NSSQLiteStoreType,
                                configuration: nil,
                                          URL: url,
                                      options: nil,
                                        error: &error) == nil {
```

The first parameter to this method, `NSSQLiteStoreType`, determines the type of the persistent store. `NSSQLiteStoreType` is a constant that tells Core Data to use a SQLite database for its persistent store. If you want your application to use a single, binary flat file instead of a SQLite database, you could specify the constant `NSBinaryStoreType` instead of `NSSQLiteStoreType`. The vast majority of the time, the default setting is the best choice, so unless you have a compelling reason to change it, leave it alone.