Mastering Core Data With Swift

Written by Bart Jacobs

Cocoacasts

Mastering Core Data With Swift

Bart Jacobs

This book is for sale at http://leanpub.com/mastering-core-data-with-swift

This version was published on 2017-07-13

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

Table of Contents

Welcome

Welcome to **Mastering Core Data With Swift**. In this book, you'll learn the ins and outs of Apple's popular Core Data framework. Even though we'll be building an iOS application, the Core Data framework is available on iOS, tvOS, macOS, and watchOS, and the contents of this book apply to each of these platforms.

Xcode 9 and Swift 4

In this book, we use **Xcode 9** and **Swift 4**. Xcode 8 and Swift 3 introduced a number of significant improvements that make working with Core Data more intuitive and more enjoyable. Make sure to have Xcode 8 or Xcode 9 installed to follow along. Everything you learn in this book applies to both Swift 3 and Swift 4.

What You'll Learn

Before we start writing code, we take a look at the Core Data framework itself. We find out what Core Data **is** and **isn't**, and we explore the heart of every Core Data application, the **Core Data stack**.

In this book, we build **Notes**, an iOS application that manages a list of notes. Notes is a simple iOS application, yet it contains all the ingredients we need to learn about the Core Data framework, from creating and deleting records to managing many-to-many relationships.

We also take a close look at the brains of a Core Data application, the **data model**. We discuss data model **versioning** and **migrations**. These concepts are essential for every Core Data application.

Core Data records are represented by managed objects. You learn how to create them, fetch them from a persistent store, and delete them if they're no longer needed.

Mastering Core Data With Swift also covers a few more advanced topics. Even though these topics are more advanced, they're essential if you work with Core Data. We talk in detail about the `NSFetchedResultsController` class and, at the end of this book, I introduce you to the brand new `NSPersistentContainer` class, a recent addition to the framework.

Last but not least, we take a deep dive into Core Data and concurrency, an often overlooked topic. This is another essential topic for anyone working with Core Data. Don't skip this.

That's a lot to cover, but I'm here to guide you along the way. If you have any feedback or questions, reach out to me via email (bart@cocoacasts.com) or Twitter (@_bartjacobs). I'm here to help.

How to Use This Book

If you'd like to follow along, I recommend downloading the source files that come with this book. The chapters that include code each have a starter project and a finished project. This makes it easy to follow along or pick a random chapter from the book. Click here to download the source files for this book.

If you're new to Core Data, then I recommend reading every chapter of the book. Over the years, I have taught thousands of developers about the Core Data framework. From that experience, I developed a roadmap for teaching Core Data. This book is the result of that roadmap.

Not everyone likes books. If you prefer video, then you may be interested in a video course in which I teach the Core Data framework. The content is virtually identical. The only difference is that you can see how I build Notes using the Core Data framework. You can find the video course on the Cocoacasts website.

1 What Is Core Data

Developers new to Core Data often don't take the time to learn about the framework. Not knowing what Core Data is, makes it hard and frustrating to wrap your head around the ins and outs of the framework. I'd like to start by spending a few minutes exploring the nature of Core Data and, more importantly, explain to you what Core Data **is** and **isn't**.

Core Data is a framework developed and maintained by Apple. It's been around for more than a decade and first made its appearance on macOS with the release of OS X Tiger in 2005. In 2009, the company made the framework available on iOS with the release of iOS 3. Today, Core Data is available on iOS, tvOS, macOS, and watchOS.

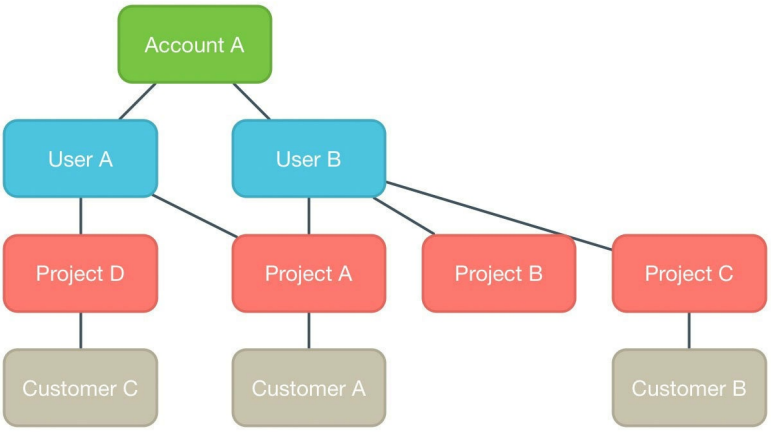
Core Data is the **M** in **MVC**, the model layer of your application. Even though Core Data can persist data to disk, data persistence is actually an optional feature of the framework. Core Data is first and foremost a framework for managing an object graph.

You've probably heard and read about Core Data before taking this course. That means that you may already know that Core Data is **not a database** and that it manages your application's **object graph**. Both statements are true. But what do they really mean?

Core Data Manages an Object Graph

Remember that Core Data is first and foremost an **object graph manager**. But what is an object graph?

An object graph is nothing more than a collection of objects that are connected with one another. The Core Data framework excels at managing complex object graphs.



What Is an Object Graph

The Core Data framework takes care of managing the life cycle of the objects in the object graph. It can optionally persist the object graph to disk and it also offers a powerful interface for searching the object graph it manages.

But Core Data is much more than that. The framework adds a number of other compelling features, such as input validation, data model versioning, and change tracking.

Even though Core Data is a perfect fit for a wide range of applications, not every application should use Core Data.

When to Use Core Data

If you're in need of a lightweight model layer, then Core Data shouldn't be your first choice. There are many, lightweight libraries that provide this type of functionality.

And if you're looking for a SQLite wrapper, then Core Data is also not what you need. For a lightweight, performant SQLite wrapper, I highly recommend Gus Mueller's FMDB. This robust, mature library provides an object-oriented interface for interacting with SQLite.

Core Data & SQLite

Core Data is an excellent choice if you want a solution that manages the model layer of your application. Developers new to Core Data are often confused by the differences between SQLite and Core Data.

If you wonder whether you need Core Data or SQLite, you're asking the wrong question. Remember that Core Data is not a database.

SQLite is a lightweight database that's incredibly performant, and, therefore, a good fit for mobile applications. Even though SQLite is advertised as a

relational database, it's important to realize that the developer is in charge of maintaining the relationships between records stored in the database.

Core Data Goes Much Further

Core Data provides an abstraction that allows developers to interact with the model layer in an object-oriented manner. Every record you interact with is an object.

Core Data is responsible for the integrity of the object graph. It ensures the object graph is kept up to date.

Drawbacks

Even though Core Data is a fantastic framework, there are several drawbacks. These drawbacks are directly related to the nature of Core Data and how it works.

Performance

Core Data can only do its magic because it keeps the object graph it manages in memory. This means that it can only operate on records once they are in memory. This is very different from performing a SQL query on a database. If you want to delete thousands of records, Core Data first needs to load each record into memory. It goes without saying that this results in memory and performance issues if done incorrectly.

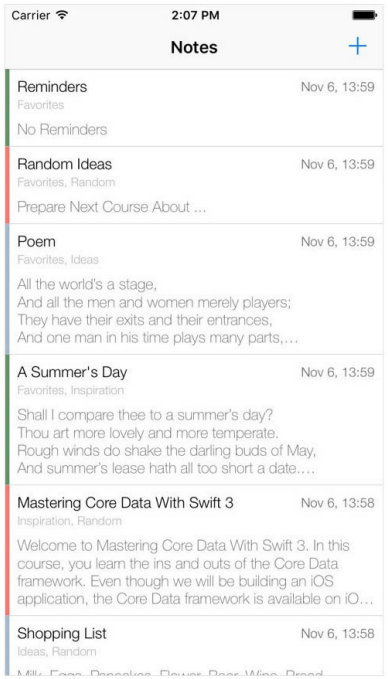
Multithreading

Another important limitation is the threading model of Core Data. The framework expects to be run on a single thread. Fortunately, Core Data has evolved dramatically over the years and the framework has put various solutions in place to make working with Core Data in a multithreaded environment much safer and much easier.

For applications that need to manage a complex object graph, Core Data is a great fit. If you only need to store a handful of unrelated objects, then you may be better off with a lightweight solution or the user defaults system.

2 Building Notes

Notes is a simple application for iOS that manages a list of notes. You can add notes, update notes, and delete notes.



Building Notes

Users can also take advantage of categories to organize their notes. A user can add, update, and delete categories. Each category has a color to make it easier to see what category a note belongs to. A note can belong to one category and a category can have multiple notes.

A note has zero or more tags. The tags of a note are listed below the title of the note. Adding, updating, and removing tags is pretty straightforward.

The user's notes are sorted by last modified date. The most recently modified note appears at the top of the table view.

Even though Notes is a simple application, it's ideal for learning the ropes of the Core Data framework. The data model contains the ingredients of a typical Core Data application with one-to-many and many-to-many relationships.

In this book, we primarily focus on the aspects that relate to Core Data. We won't focus on building the user interface unless it's necessary to explain a concept of the Core Data framework. That is Notes in a nutshell.

In the next chapter, we start our journey by exploring the Core Data stack, the heart of every Core Data application.

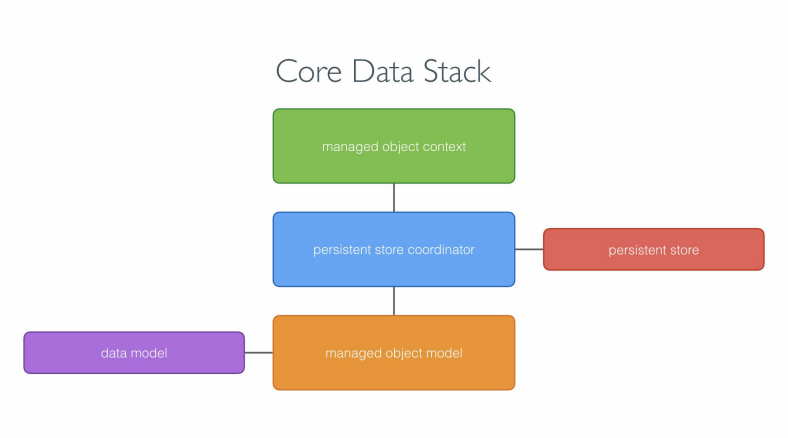
3 Exploring the Core Data Stack

Earlier in this book, we learned what Core Data is and isn't. In this chapter, we zoom in on the building blocks of the Core Data framework.

As I mentioned earlier, it's key that you understand how the various classes that make Core Data tick play together. The star players of the Core Data framework are:

- the managed object model
- the managed object context
- the persistent store coordinator

This diagram shows how these classes relate to one another. We'll use this diagram as a guideline in this chapter.

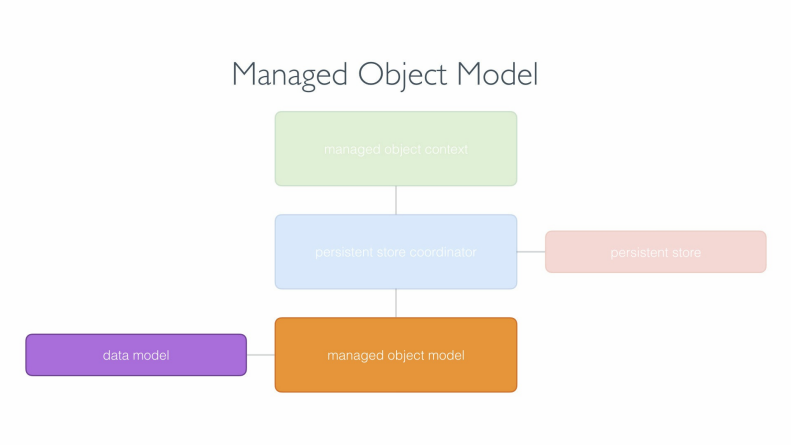


Core Data Stack

Managed Object Model

The managed object model is an instance of the `NSManagedObjectModel` class. A typical Core Data application has one instance of the `NSManagedObjectModel` class, but it's possible to have multiple. The `NSManagedObjectModel` instance represents the data model of the Core Data application.

This diagram shows that the managed object model is connected to the data model. The data model is represented by a file in the application bundle that contains the data schema of the application. This is something we revisit later in this book when we start working with Core Data.



Managed Object Model

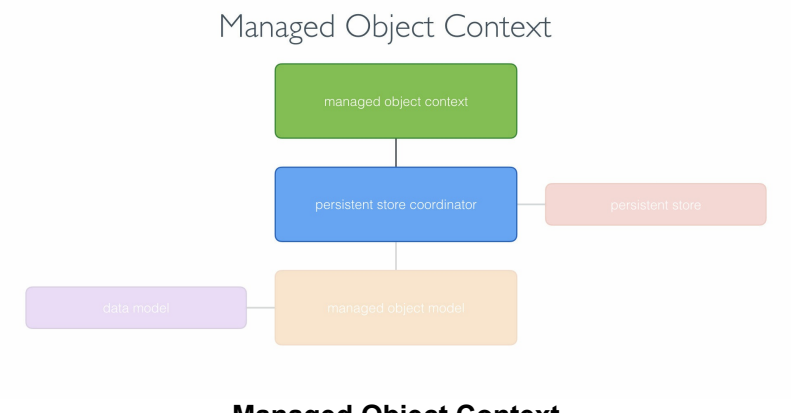
The data model is represented by a file in the application bundle that contains the data schema of the application. The data schema is nothing more than a collection of entities. An entity can have attributes and relationships, which make up the data model of the application.

We explore the data model in more detail later. For now, remember that the managed object model is an instance of the `NSManagedObjectModel` class and represents the data model of the Core Data application.

Managed Object Context

A managed object context is represented by an instance of the `NSManagedObjectContext` class. A Core Data application has one or more managed object contexts. Each managed object context manages a collection of model objects, instances of the `NSManagedObject` class.

The managed object context receives the model objects through a persistent store coordinator as you can see in this diagram. A managed object context keeps a reference to the persistent store coordinator of the application.



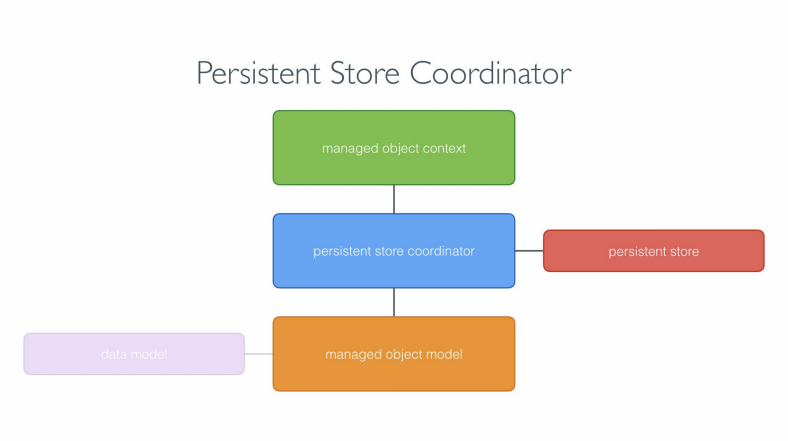
Managed Object Context

The managed object context is the object you interact with most. It creates, reads, updates, and deletes model objects. From a developer's perspective,

the `NSManagedObjectContext` class is the workhorse of the Core Data framework.

Persistent Store Coordinator

The persistent store coordinator is represented by an instance of the `NSPersistentStoreCoordinator` class and it plays a key role in every Core Data application.



Persistent Store Coordinator

While it's possible to have multiple persistent store coordinators, most applications have only one. Very, very rarely is there a need to have multiple persistent store coordinators in an application.

The persistent store coordinator keeps a reference to the managed object model and every parent managed object context keeps a reference to the persistent store coordinator.

But wait … what's a *parent* managed object context? Later in this book, we take a closer look at parent and child managed object contexts. Don't worry about this for now.

The above diagram also tells us that the persistent store coordinator is connected to one or more persistent stores. What's a persistent store?

Remember that Core Data manages an object graph. The framework is only useful if the persistent store coordinator is connected to one or more persistent stores.

Out of the box, Core Data supports three persistent store types:

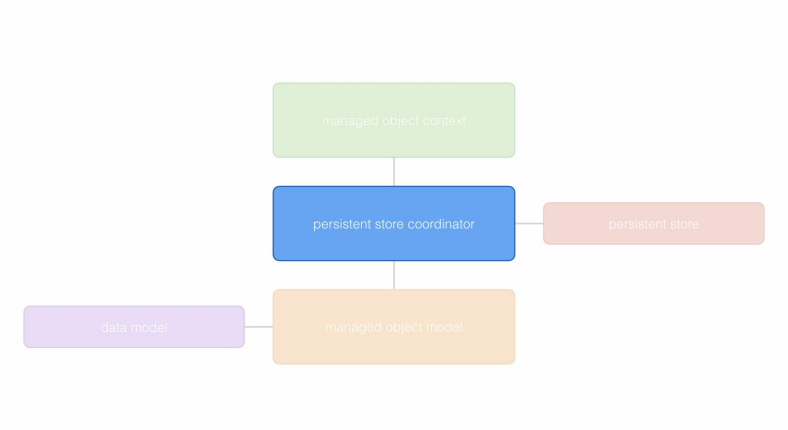
- a SQLite database
- a binary store
- an in-memory store

Each persistent store type has its pros and cons. Most applications use a SQLite database as their persistent store. As we saw in the previous chapter, SQLite is lightweight and very fast. It's great for mobile and desktop applications.

Now that we know what the Core Data stack consists of, it's time to explore how it operates in an application.

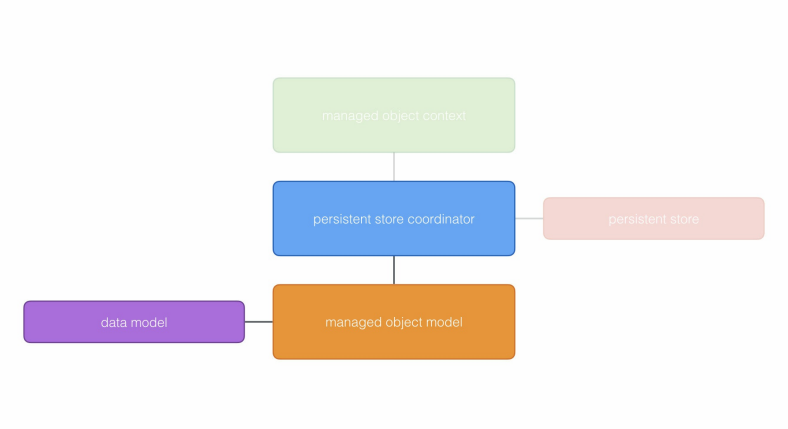
How Does Core Data Work

The heart of the Core Data stack is the **persistent store coordinator**. The persistent store coordinator is instantiated first when the Core Data stack is created.



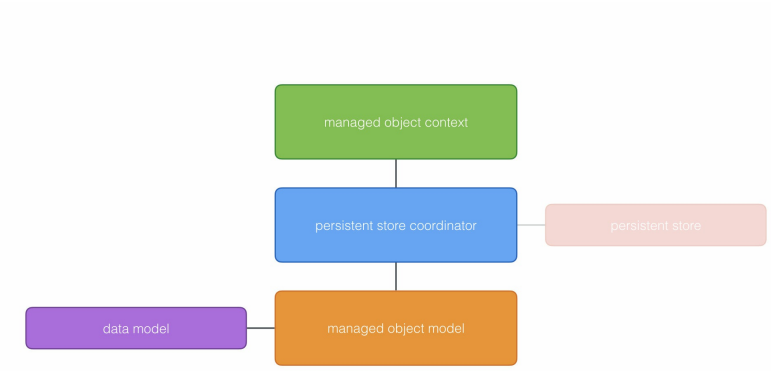
The persistent store coordinator is instantiated first.

But to create the persistent store coordinator, we need a **managed object model**. Why is that? The persistent store coordinator needs to know what the data schema of the application looks like.



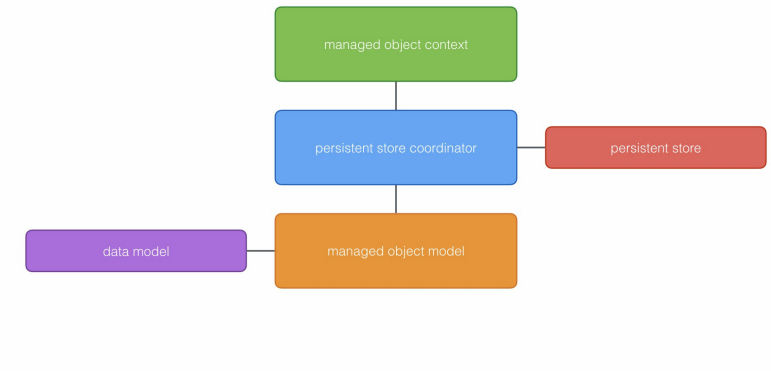
The persistent store coordinator needs a managed object model.

After setting up the persistent store coordinator and the managed object model, the workhorse of the Core Data stack is initialized, the **managed object context**. Remember that a managed object context keeps a reference to the persistent store coordinator.



The managed object context is the workhorse of the Core Data stack.

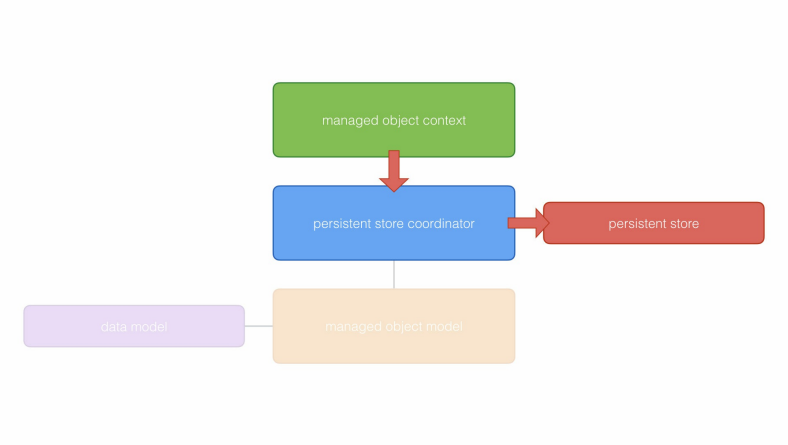
With the Core Data stack set up, the application is ready to use Core Data to interact with the application's persistent store. In most cases, your application interacts with the persistent store coordinator through the managed object context.



Your application interacts with the persistent store coordinator through the managed object context.

You will rarely, if ever, directly interact with the persistent store coordinator or the managed object model. As I mentioned earlier, the `NSManagedObjectContext` class is the class you interact with most frequently.

The managed object context is used to create, read, update, and delete records. When the changes made in the managed object context are saved, the managed object context pushes them to the persistent store coordinator, which sends the changes to the corresponding persistent store.



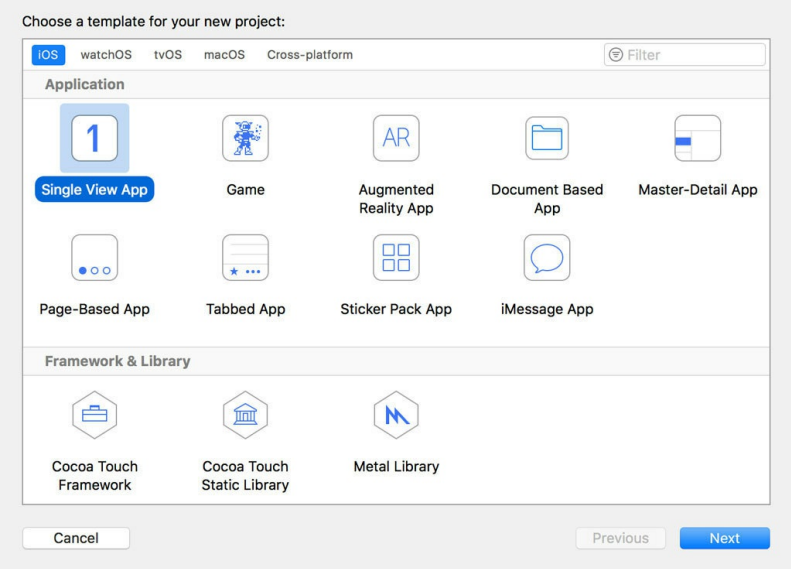
The managed object context pushes changes to the persistent store coordinator, which sends them to the persistent store.

If your application has multiple persistent stores, the persistent store coordinator figures out which persistent store needs to store the changes of the managed object context.

Now that you know what Core Data is and how the Core Data stack is set up, it’s time to write some code. In the next chapters, we create a Core Data stack and explore the classes we discussed in this chapter.

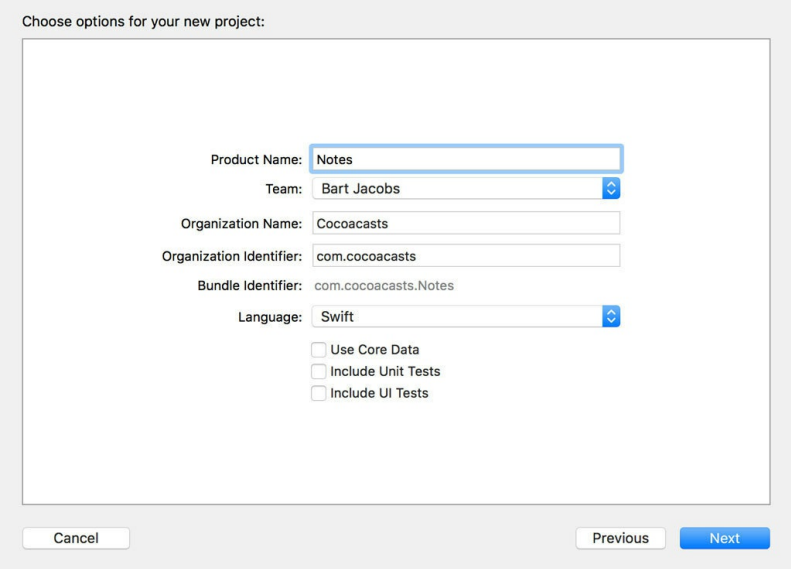
4 Creating the Project

Before we set up the Core Data stack, we need to create the project for Notes. Open Xcode and create a new project based on the **Single View Application** template.



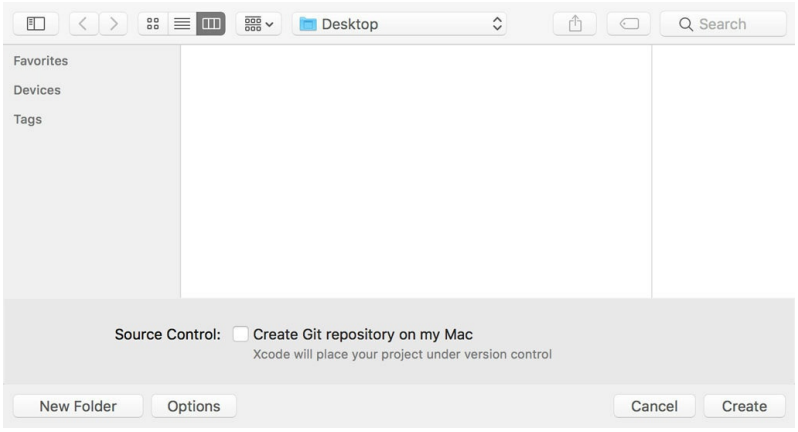
Choosing the Single View Application Template

Name the project **Notes**, set **Language** to **Swift**, and, if you're using Xcode 8, set **Devices** to **iPhone**. Make sure **Use Core Data** is unchecked. We're going to start from scratch.



Configuring the Project

Choose where you want to store the project and click **Create**.

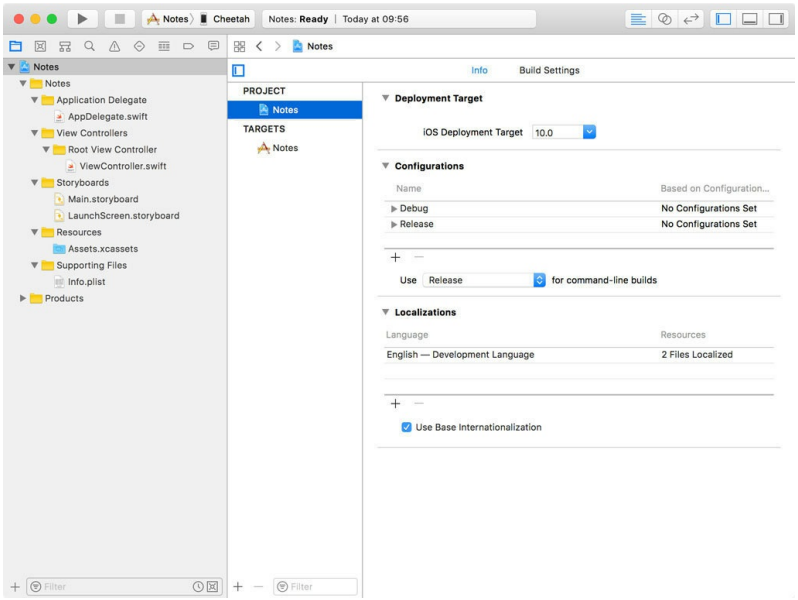


Creating the Project

Before we start writing code, I want to do some housekeeping by modifying the structure of the project. The first thing I do when I start a new project is create groups for the files and folders of the project. These are the groups I create in the **Project Navigator**:

- Application Delegate
- View Controllers
  - Root View Controller
- Storyboards
- Resources
- Supporting Files

This is what the result looks like in the **Project Navigator**. That looks a lot better. Doesn't it?



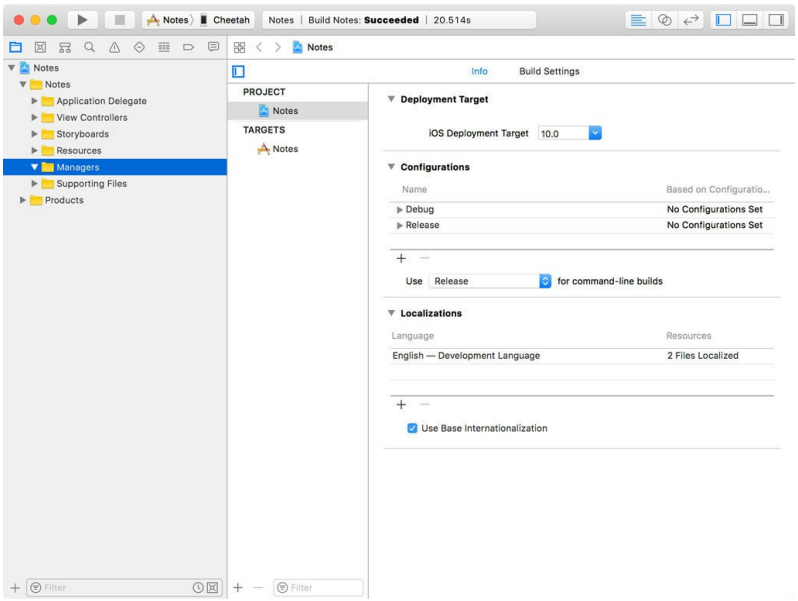
Updating the Project Structure

For this project, I've set the **Deployment Target** of the project to **10.0**. In the next chapter, we set up the Core Data stack of the project.

5 Setting Up the Core Data Stack

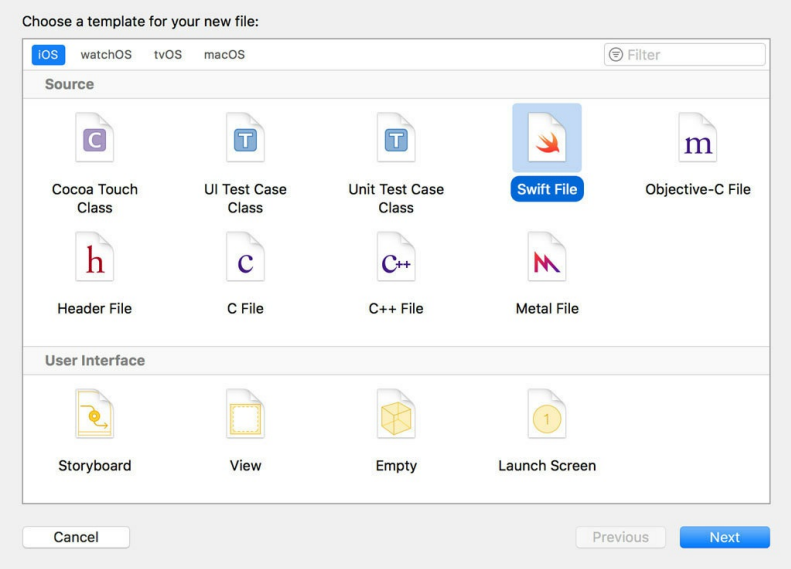
It's time to write some code. Had we checked the **Use Core Data** checkbox during the setup of the project, Xcode would have put the code for the Core Data stack in the **application delegate**. This is something I don't like and we won't be cluttering the application delegate with the setup of the Core Data stack.

Instead, we're going to create a separate class responsible for setting up and managing the Core Data stack. Create a new group and name it **Managers**.

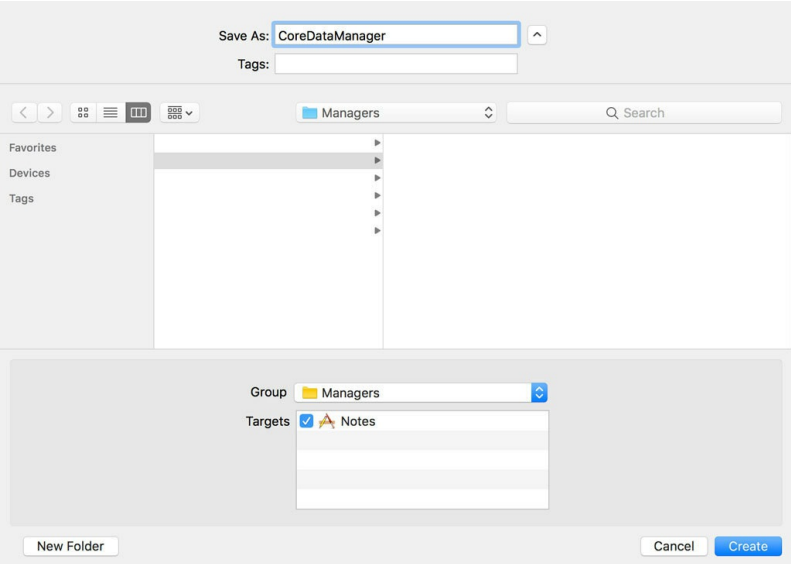


Creating the Managers Group

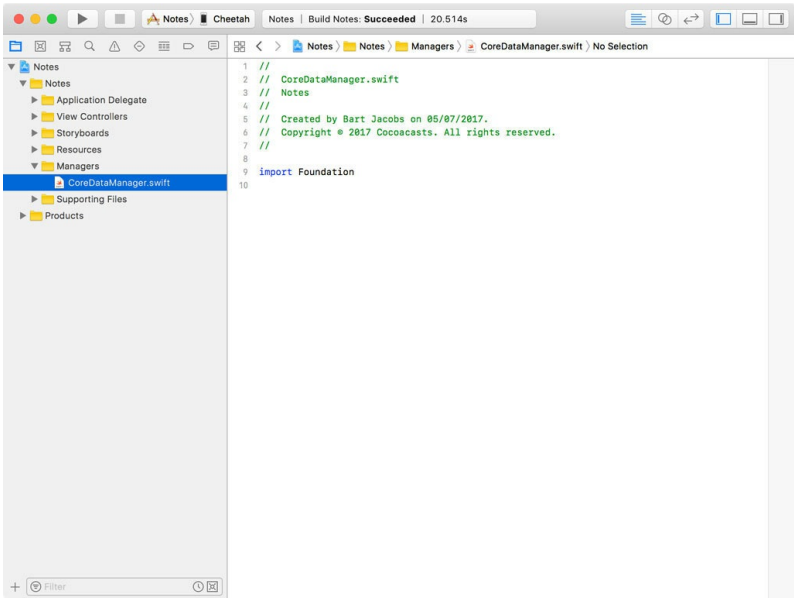
Create a new Swift file in the **Managers** group and name the file **CoreDataManager.swift**. The `CoreDataManager` class is in charge of the Core Data stack of the application.



Choosing the Swift File Template



Creating CoreDataManager.swift



Creating CoreDataManager.swift

Replace the import statement for the **Foundation** framework with an import statement for the **Core Data** framework.

```
import CoreData
```

Next, we define the class itself. Note that we mark the `CoreDataManager` class as final. It's not intended to be subclassed.

```
import CoreData

final class CoreDataManager {

}
```

We're going to keep the implementation straightforward. The only information we're going to give the Core Data manager is the name of the data model. We first create a property for the name of the data model. The property is of type `String`.

```
import CoreData

final class CoreDataManager {

    // MARK: - Properties

    private let modelName: String

}
```

The designated initializer of the class accepts the name of the data model as an argument.

```
import CoreData

final class CoreDataManager {

    // MARK: - Properties

    private let modelName: String

    // MARK: - Initialization

    init(modelName: String) {
        self.modelName = modelName
    }

}
```

Remember that we need to instantiate three objects to set up the Core Data stack:

- a managed object model
- a managed object context
- a persistent store coordinator

Let's start by creating a lazy property for each of these objects. The properties are marked `private`. But notice that we only mark the setter of the `managedObjectContext` property private. The managed object context of the Core Data manager should be accessible by other objects that need access to the Core Data stack. Remember that the managed object context is the object we will be working with most frequently. It's the workhorse of the Core Data stack.

```
private(set) lazy var managedObjectContext: NSManagedObjectContext =\
 {}()

private lazy var managedObjectModel: NSManagedObjectModel = {}()

private lazy var persistentStoreCoordinator: NSPersistentStoreCoordi\
nator = {}()
```

Managed Object Context

Let's start with the implementation of the `managedObjectContext` property. We initialize an instance of the `NSManagedObjectContext` class by invoking its designated initializer, `init(concurrencyType:)`. This initializer accepts an argument of type `NSManagedObjectContextConcurrencyType`. We pass in `mainQueueConcurrencyType`, which means the managed object context is associated with the main queue or the main thread of the application. We learn more about threading later in this book. Don't worry about this for now.

```
// Initialize Managed Object Context
let managedObjectContext = NSManagedObjectContext(concurrencyType: .\
mainQueueConcurrencyType)
```

Remember that every parent managed object context keeps a reference to the persistent store coordinator of the Core Data stack. This means we need to set the `persistentStoreCoordinator` property of the managed object context.

```
// Configure Managed Object Context
managedObjectContext.persistentStoreCoordinator = self.persistentSto\
reCoordinator
```

And we return the managed object context from the closure.

```
private(set) lazy var managedObjectContext: NSManagedObjectContext =\
 {
    // Initialize Managed Object Context
    let managedObjectContext = NSManagedObjectContext(concurrencyTyp\
e: .mainQueueConcurrencyType)

    // Configure Managed Object Context
    managedObjectContext.persistentStoreCoordinator = self.persisten\
tStoreCoordinator

```

```
    return managedObjectContext
}()
```

Managed Object Model

Initializing the managed object model is easy. We ask the application bundle for the URL of the data model and we use the URL to instantiate an instance of the `NSManagedObjectModel` class.

```
// Fetch Model URL
guard let modelURL = Bundle.main.url(forResource: self.modelName, wi\
thExtension: "momd") else {
    fatalError("Unable to Find Data Model")
}

// Initialize Managed Object Model
guard let managedObjectModel = NSManagedObjectModel(contentsOf: mode\
lURL) else {
    fatalError("Unable to Load Data Model")
}
```

We return the managed object model from the closure.

```
private lazy var managedObjectModel: NSManagedObjectModel = {
    // Fetch Model URL
    guard let modelURL = Bundle.main.url(forResource: self.modelName\
, withExtension: "momd") else {
        fatalError("Unable to Find Data Model")
    }

    // Initialize Managed Object Model
    guard let managedObjectModel = NSManagedObjectModel(contentsOf: \
modelURL) else {
        fatalError("Unable to Load Data Model")
    }

    return managedObjectModel
}()
```

We throw a fatal error if the application is unable to find the data model in the application bundle or if we're unable to instantiate the managed object model. Why is that? Because this should never happen in production. If the data model isn't present in the application bundle or the application is unable to load the data model from the application bundle, we have bigger problems to worry about.

Notice that we ask the application bundle for the URL of a resource with an **momd** extension. This is the compiled version of the data model. We discuss the data model in more detail later in this book.

Persistent Store Coordinator

The last piece of the puzzle is the persistent store coordinator. This is a bit more complicated. We first instantiate an instance of the `NSPersistentStoreCoordinator` class using the managed object model. But that's only the first step.

```
// Initialize Persistent Store Coordinator
let persistentStoreCoordinator = NSPersistentStoreCoordinator(manage\
dObjectModel: self.managedObjectModel)
```

The Core Data stack is only functional once the persistent store is added to the persistent store coordinator. We start by creating the URL for the persistent store. There are several locations for storing the persistent store. In this example, we store the persistent store in the **Documents** directory of the application's sandbox. But you could also store it in the **Library** directory.

We append **sqlite** to the name of the data model because we're going to use a SQLite database as the persistent store. Remember that Core Data supports SQLite databases out of the box.

```
// Helpers
let fileManager = FileManager.default
let storeName = "\(self.modelName).sqlite"

// URL Documents Directory
let documentsDirectoryURL = fileManager.urls(for: .documentDirectory\
, in: .userDomainMask)[0]

// URL Persistent Store
let persistentStoreURL = documentsDirectoryURL.appendingPathComponen\
t(storeName)
```

Because adding a persistent store is an operation that can fail, we need to perform it in a `do-catch` statement. To add a persistent store we invoke `addPersistentStore(ofType:configurationName:at:options:)` on the persistent store coordinator. That's quite a mouthful.

This method accepts four arguments:

- the type of the persistent store, SQLite in this example
- an optional configuration
- the location of the persistent store
- an optional dictionary of options

```
do {
    // Add Persistent Store
    let options = [ NSMigratePersistentStoresAutomaticallyOption : t\
rue, NSInferMappingModelAutomaticallyOption : true ]
    try persistentStoreCoordinator.addPersistentStore(ofType: NSSQLi\
teStoreType, configurationName: nil, at: persistentStoreURL, options\
: options)
```

```
8
9 } catch {}
```

The second parameter, the configuration, isn't important for this discussion. The fourth argument, the options dictionary, is something we discuss later in this book.

If the persistent store coordinator cannot find a persistent store at the location we specified, it creates one for us. If a persistent store already exists at the specified location, it's added to the persistent store coordinator. This means that the persistent store is automatically created the first time a user launches your application. The second time, Core Data looks for the persistent store, finds it at the specified location, and adds it to the persistent store coordinator. The framework handles this for you.

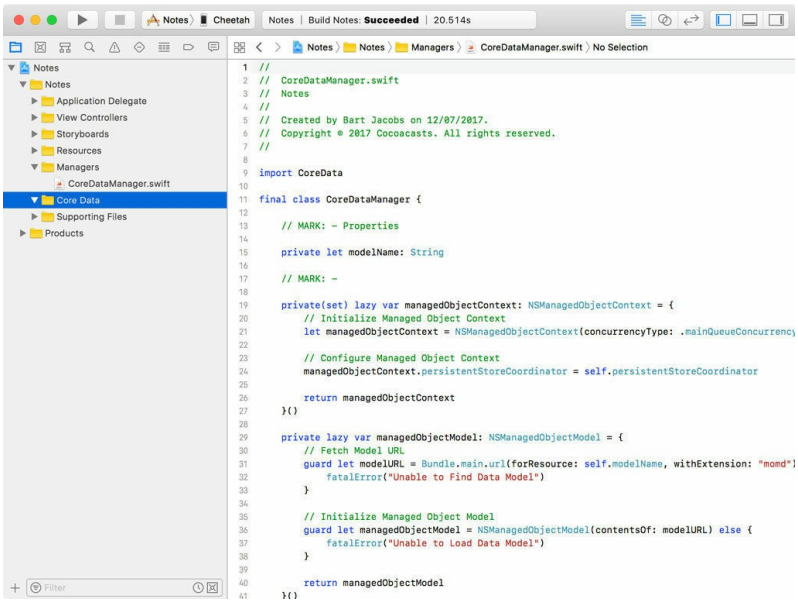
In the `catch` clause, we print the error to the console if the operation failed. We return the persistent store coordinator from the closure.

```
1 private lazy var persistentStoreCoordinator: NSPersistentStoreCoordi\
2 nator = {
3     // Initialize Persistent Store Coordinator
4     let persistentStoreCoordinator = NSPersistentStoreCoordinator(ma\
5 nagedObjectModel: self.managedObjectModel)
6
7     // Helpers
8     let fileManager = FileManager.default
9     let storeName = "\(self.modelName).sqlite"
10
11    // URL Documents Directory
12    let documentsDirectoryURL = fileManager.urls(for: .documentDirec\
13 tory, in: .userDomainMask)[0]
14
15    // URL Persistent Store
16    let persistentStoreURL = documentsDirectoryURL.appendingPathComp\
17 onent(storeName)
18
19    do {
20        // Add Persistent Store
21        try persistentStoreCoordinator.addPersistentStore(ofType: NS\
22 SQLiteStoreType, configurationName: nil, at: persistentStoreURL, opt\
23 ions: nil)
24
25    } catch {
26        fatalError("Unable to Add Persistent Store")
27    }
28
29    return persistentStoreCoordinator
30 }()
```

We now have a working Core Data stack, but we're currently assuming that everything is working fine all the time. Later in this book, we make the Core Data manager more robust. Right now we just want to set up a Core Data stack to make sure we have something to work with.

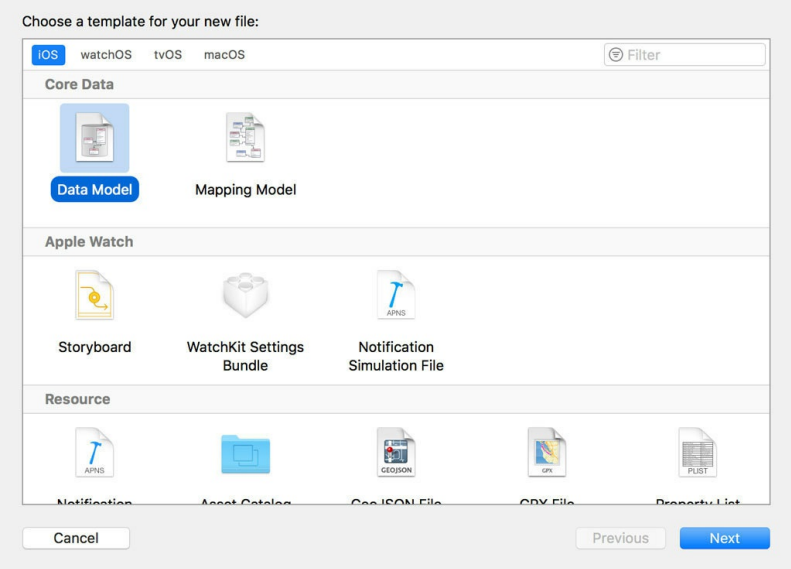
Adding a Data Model

Before we can take the Core Data manager for a spin, we need to add a data model to the project. Create a new group for the data model and name it **Core Data**.



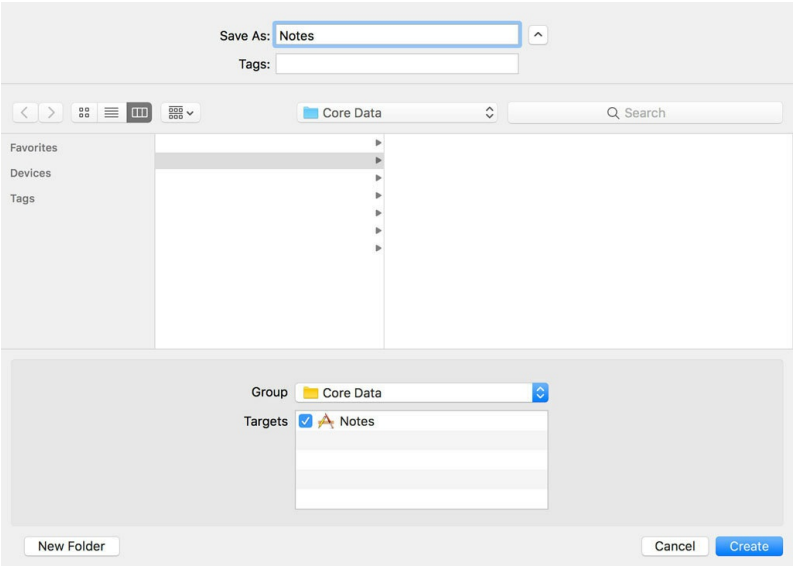
Creating the Core Data Group

Create a new file and choose the **Data Model** template from the **iOS > Core Data** section.

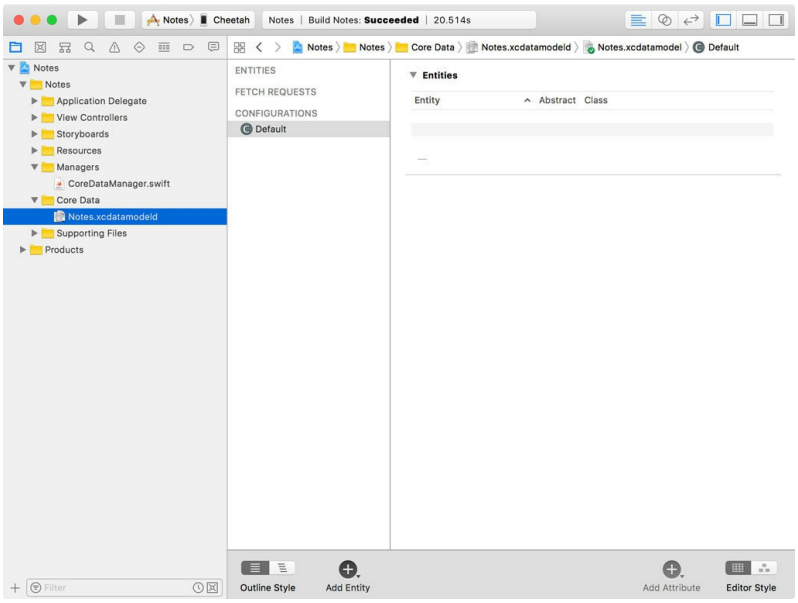


Choosing the Data Model Template

Name the data model **Notes** and click **Create**.



Naming the Data Model Notes



Creating the Data Model

Notice that the extension of the data model is **xcdatamodeld**. This is different from the extension we used earlier in the `managedObjectModel` property. The **xcdatamodeld** file isn't included in the compiled application. The **xcdatamodeld** file is compiled into an **momd** file and it's the latter that's included in the compiled application. Only what is absolutely essential is included in the **momd** file.

Setting Up the Core Data Stack

Open **AppDelegate.swift** and instantiate an instance of the `CoreDataManager` class in the `application(_:didFinishLaunchingWithOptions:)` method. We print the value of the `managedObjectContext` property to the console to make sure the Core Data stack was successfully set up.

```
1 import UIKit
2
3 @UIApplicationMain
4 class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
5
6       // MARK: - Properties
7
8       var window: UIWindow?
9
10      // MARK: - Application Life Cycle
11
12      func application(_ application: UIApplication, didFinishLaunchin\
13  gWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -\
14  > Bool {
15          let coreDataManager = CoreDataManager(modelName: "Notes")
16          print(coreDataManager.managedObjectContext)
17          return true
18      }
19
20  }
```

Build and run the application and inspect the output in the console. The output should looks something like this.

```
1 <NSManagedObjectContext: 0x6180001cdd40>
```

Great. That seems to work. In the next chapter, we use dependency injection to pass the Core Data manager from the application delegate to the root view controller.

6 Core Data and Dependency Injection

I'm not going to lie. I don't like singletons. Singletons are fine if they're used correctly, but I don't like singletons for convenience. They almost always lead to problems down the line.

This means that the Core Data manager isn't going to be a singleton. We're going to create an instance in the application delegate and inject it into the root view controller. Dependency injection is surprisingly easy if you break it down to its bare essentials.

We first open **ViewController.swift** and create a property for the Core Data manager. The property is an optional because we set it after the view controller is initialized. That's a drawback I'm happy to accept.

ViewController.swift

```
1  import UIKit
2
3  class ViewController: UIViewController {
4
5      // MARK: - Properties
6
7      var coreDataManager: CoreDataManager?
8
9      ...
10
11 }
```

Next, we open **AppDelegate.swift** and declare a private constant property, `coreDataManager`, of type `CoreDataManager`. We instantiate a `CoreDataManager` instance and assign it to the property.

AppDelegate.swift

```
1  import UIKit
2
3  @UIApplicationMain
4  class AppDelegate: UIResponder, UIApplicationDelegate {
5
6      // MARK: - Properties
7
8      var window: UIWindow?
9
10     // MARK: -
11
12     private let coreDataManager = CoreDataManager(modelName: "Notes")
13
```

```
    ...

}
```

The magic happens in the `application(_:didFinishLaunchingWithOptions:)` method. We load the main storyboard and instantiate its initial view controller. We expect the initial view controller to be an instance of the `ViewController` class. If it isn't, we throw a fatal error.

AppDelegate.swift

```
// Load Storyboard
let storyboard = UIStoryboard(name: "Main", bundle: Bundle.main)

// Instantiate Initial View Controller
guard let initialViewController = storyboard.instantiateInitialViewC\
ontroller() as? ViewController else {
    fatalError("Unable to Configure Initial View Controller")
}
```

We configure the initial view controller by setting its `coreDataManager` property. In other words, we inject the Core Data manager into the view controller. That's all dependency injection is, setting an object's instance variables.

AppDelegate.swift

```
// Configure Initial View Controller
initialViewController.coreDataManager = coreDataManager
```

Last but not least, we set the `rootViewController` property of the `window` property of the application delegate.

AppDelegate.swift

```
// Configure Window
window?.rootViewController = initialViewController
```

This is what the implementation of `application(_:didFinishLaunchingWithOptions:)` looks like when you're finished.

AppDelegate.swift

```
func application(_ application: UIApplication, didFinishLaunchingWit\
hOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bo\
ol {
    // Load Storyboard
    let storyboard = UIStoryboard(name: "Main", bundle: Bundle.main)

    // Instantiate Initial View Controller
```

```
    guard let initialViewController = storyboard.instantiateInitialV\
iewController() as? ViewController else {
        fatalError("Unable to Configure Initial View Controller")
    }

    // Configure Initial View Controller
    initialViewController.coreDataManager = coreDataManager

    // Configure Window
    window?.rootViewController = initialViewController

    return true
}
```

Let's give it a try by adding a print statement to the `viewDidLoad()` method of the `ViewController` class. Build and run the application and inspect the output in the console.

ViewController.swift

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

    print(coreDataManager?.managedObjectContext ?? "No Managed Objec\
t Context")
}
```

This example illustrates how easy it is to use dependency injection to pass objects around without relying on singletons. Truth be told, there's no need to instantiate the Core Data manager in the application delegate, but I hope it shows that singletons aren't always the only solution.

Before we move on, I'd like to move the instantiation of the Core Data manager to the view controller. This makes more sense. The application delegate shouldn't be bothered with anything related to Core Data.

ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    // MARK: - Properties

    private var coreDataManager = CoreDataManager(modelName: "Notes")

    // MARK: - View Life Cycle

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

    // MARK: - Navigation

    override func prepare(for segue: UIStoryboardSegue, sender: Any?\
```

```
) {

    }

}
```

AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    // MARK: - Properties

    var window: UIWindow?

    // MARK: - Application Life Cycle

    func application(_ application: UIApplication, didFinishLaunchin\
gWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -\
> Bool {
        return true
    }

}
```

We also have the advantage that the `coreDataManager` property is no longer an optional. Great. In the next chapters, we take a close look at data models.

7 Data Model, Entities, and Attributes

The data model is a key component of the Core Data stack and an integral part of every Core Data application. In this chapter, we explore the data model and learn about entities and attributes.

Compiling the Data Model

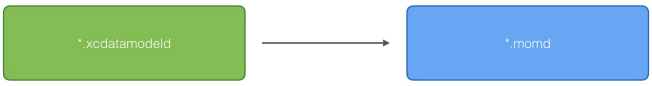
When the application sets up the Core Data stack, the managed object model loads the data model from the application bundle. The code responsible for this lives in the closure of the `managedObjectModel` property of the `CoreDataManager` class.

CoreDataManager.swift

```
1  private lazy var managedObjectModel: NSManagedObjectModel = {
2      // Fetch Model URL
3      guard let modelURL = Bundle.main.url(forResource: self.modelName\
4  , withExtension: "momd") else {
5          fatalError("Unable to Find Data Model")
6      }
7
8      // Initialize Managed Object Model
9      guard let managedObjectModel = NSManagedObjectModel(contentsOf: \
10 modelURL) else {
11         fatalError("Unable to Load Data Model")
12     }
13
14     return managedObjectModel
15 }()
```

The file that's loaded from the application bundle is named **Notes** and has an **momd** extension. The extension stands for **managed object model document**. As we learned earlier, the file extension of the data model in the **Project Navigator** is different, **xcdatamodeld**. And yet if you run the application, the Core Data stack is set up without issues. How's that possible?

When the application is built, the data model file you see in the **Project Navigator** is compiled into an **momd** file. Why is that necessary? Why does the data model need to be compiled?



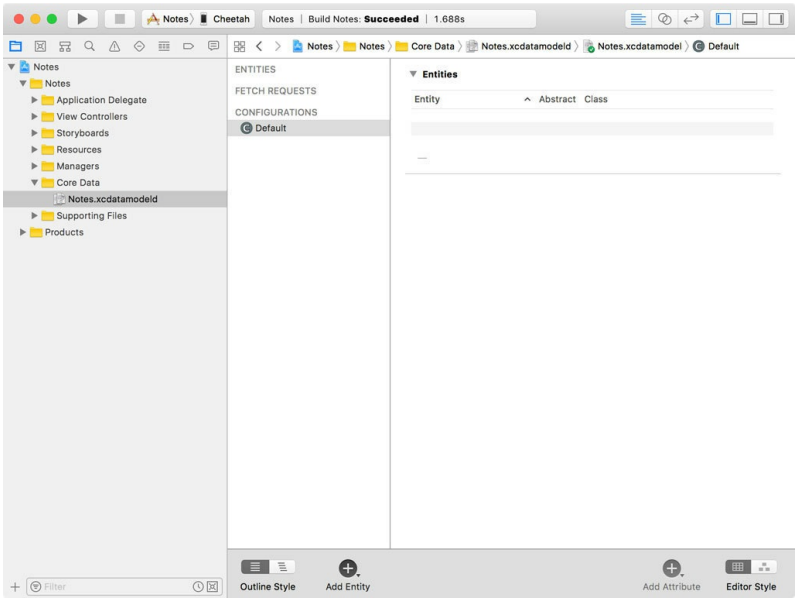
Compiling the Data Model

The **xcdatamodeld** file is the file we edit during development. We use it for defining the application's data model. It shows us a visual representation of the data model as we'll see in a moment. Much of the information in the **xcdatamodeld** file isn't needed for Core Data to do its work.

At compile time, Xcode collects the data it needs from the **xcdatamodeld** file and creates an **momd** file. It's the **momd** file that's included in the compiled application. The resulting **momd** file is much smaller and it only contains what's absolutely essential for Core Data to infer the data model.

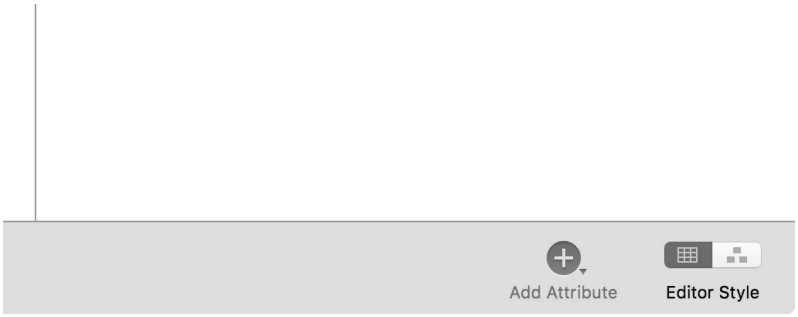
Exploring the Data Model Editor

To edit the data model, Xcode ships with a powerful data model editor. In the **Project Navigator** on the left, select **Notes.xcdatamodeld**. The data model editor should automatically open, showing us the data model.



Exploring Xcode's Data Model Editor

The data model editor has two styles, **table** and **graph**. You can toggle between these styles with the control in the lower right of the data model editor.



Toggling Between Editor Styles

The table style is useful for adding and editing entities, attributes, and relationships. Most of your time is spent in the table style. The graph style is ideal for visualizing relationships between entities. But we first need to answer the question “What is an entity?”

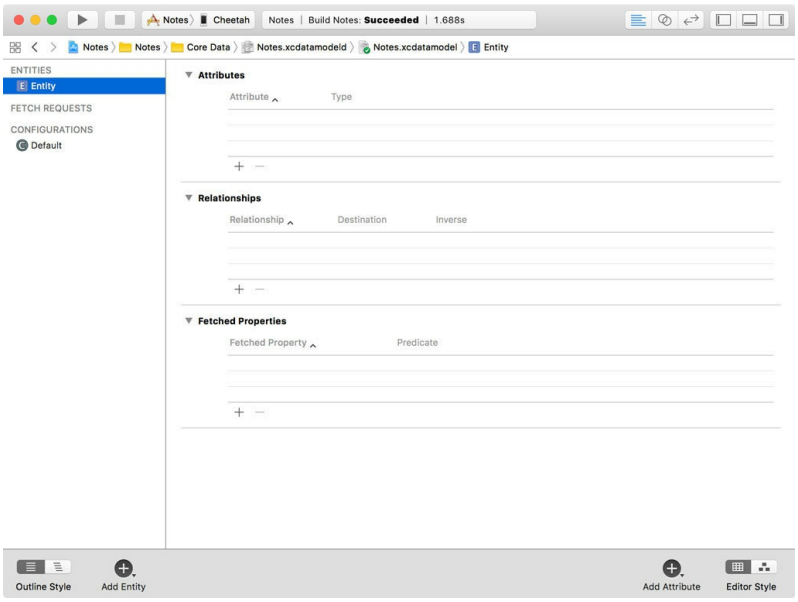
What Is an Entity

Even though Core Data isn’t a database, you can think of an entity as a table in a database. An entity has a name and properties. A property is either an **attribute** or a **relationship**. For example, an entity named **Person** can have an attribute **firstName** and **lastName**. It could also have a relationship **address** that points to an **Address** entity. We discuss relationships in more detail in the next chapter.

What Is an Entity

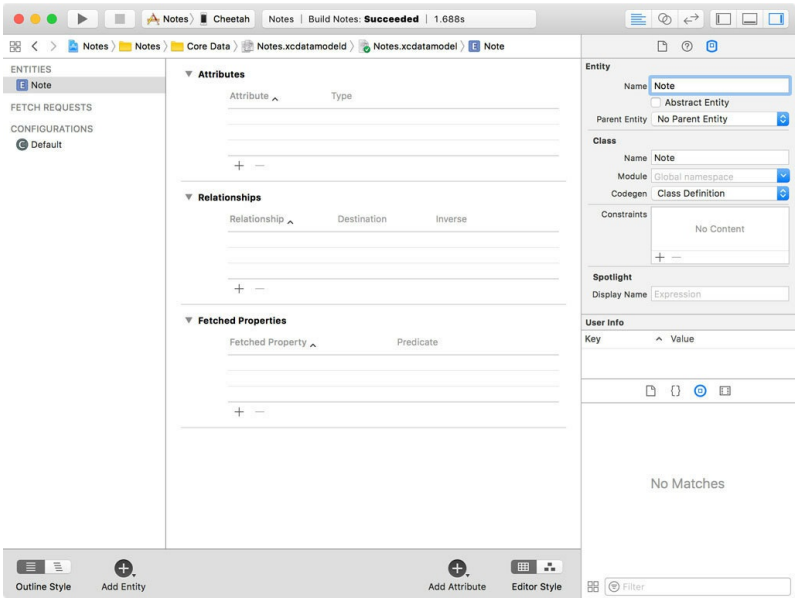
Creating an Entity

You can add an entity by clicking the **Add Entity** button at the bottom of the data model editor. The editor’s table style is split into a navigator on the left and a detail view on the right. In the **Entities** section of the navigator, you should see an entity named **Entity**.



Adding an Entity

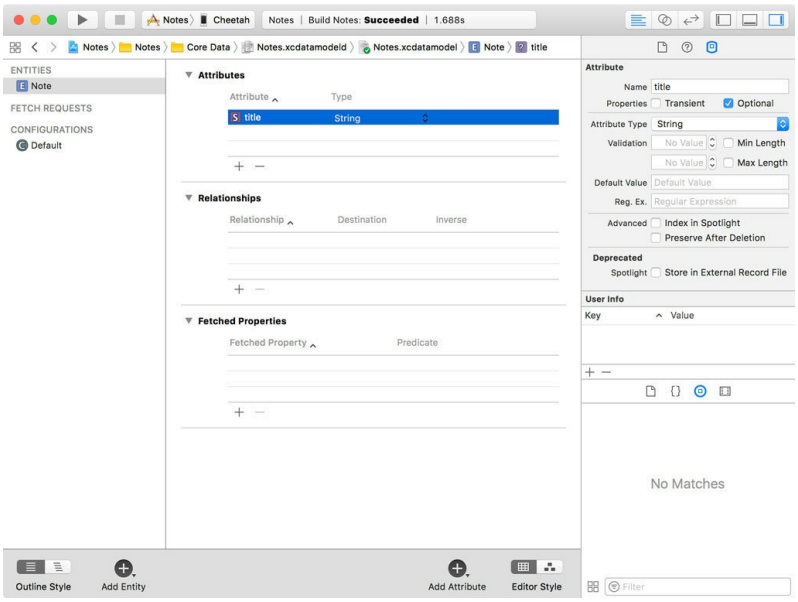
Select the entity, open the **Utilities** pane on the right, and select the **Data Model Inspector**. The inspector shows the details of the entity. Set the name of the entity to **Note**.



Editing an Entity

Creating an Attribute

With the **Note** entity selected, click the **Add Attribute** button at the bottom to add an attribute to the entity. In the **Attributes** table, set **Attribute** to **title** and **Type** to **String**. You can inspect the details of an attribute by selecting it and opening the **Data Model Inspector** in the **Utilities** pane on the right.



Adding an Attribute

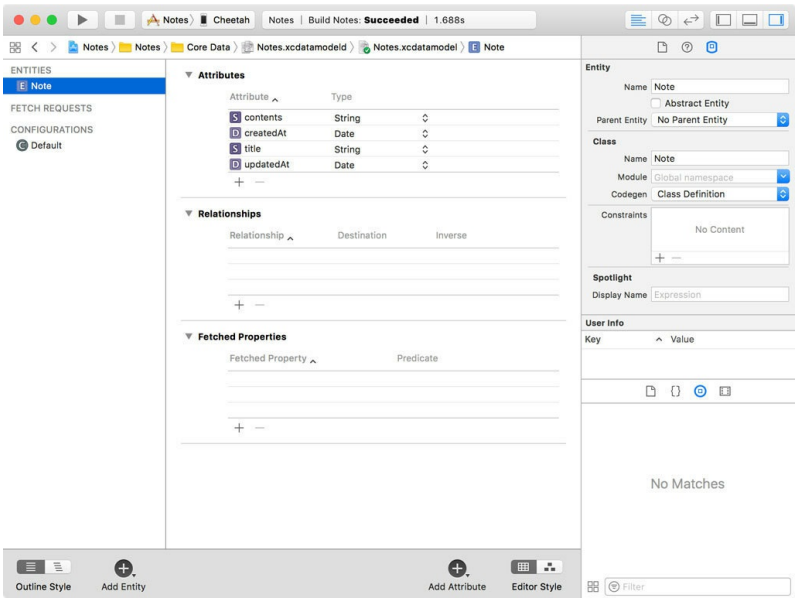
In the **Data Model Inspector**, you can see that an attribute has a number of configuration options, such as validation rules, a default value, and indexing options. Some of the options differ from type to type.

The checkbox **Optional** is checked by default. This indicates that the attribute **title** is optional for the **Note** entity. What does that mean?

If we create a note and the value of the **title** attribute isn't set, Core Data won't complain because the attribute is marked as optional. It's fine if the note record doesn't have a title. If you make the attribute required by unchecking the checkbox, Core Data throws an error if you try to save a note that doesn't have a title.

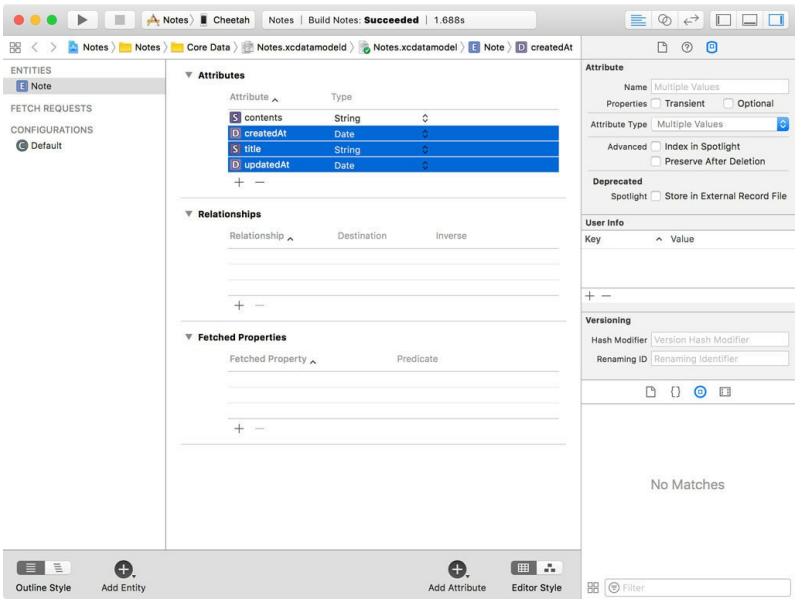
Before we move, we're going to add three more attributes:

- **contents** of type **String**
- **createdAt** of type **Date**
- **updatedAt** of type **Date**



Adding Attributes to the Note Entity

We make the `title`, `createdAt`, and `updatedAt` attributes required by unchecking the **Optional** checkbox in the **Data Model Inspector** on the right.



Requiring Attributes

In the next chapter, we focus on relationships, another powerful feature of Core Data.

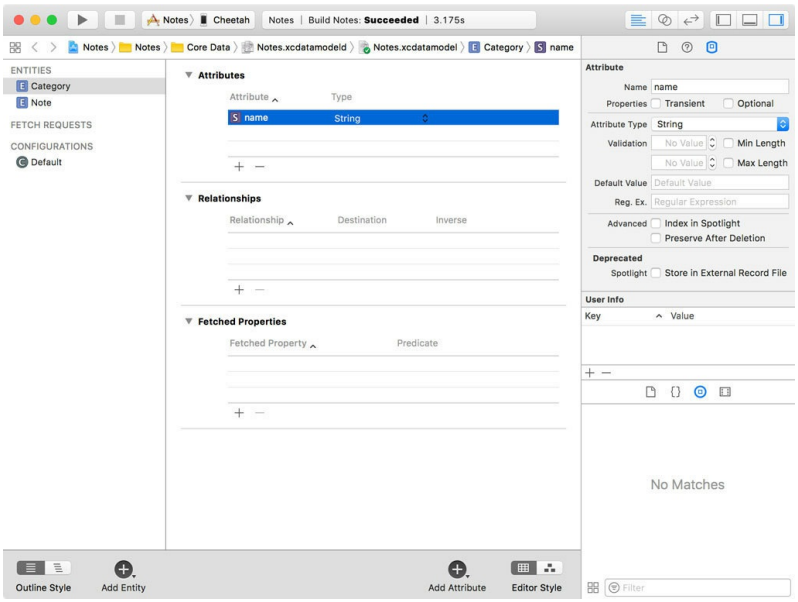
8 Data Model, Entities, and Relationships

Core Data is great at managing relationships. In this chapter, we continue our exploration of the data model by taking a close look at relationships.

Adding More Entities

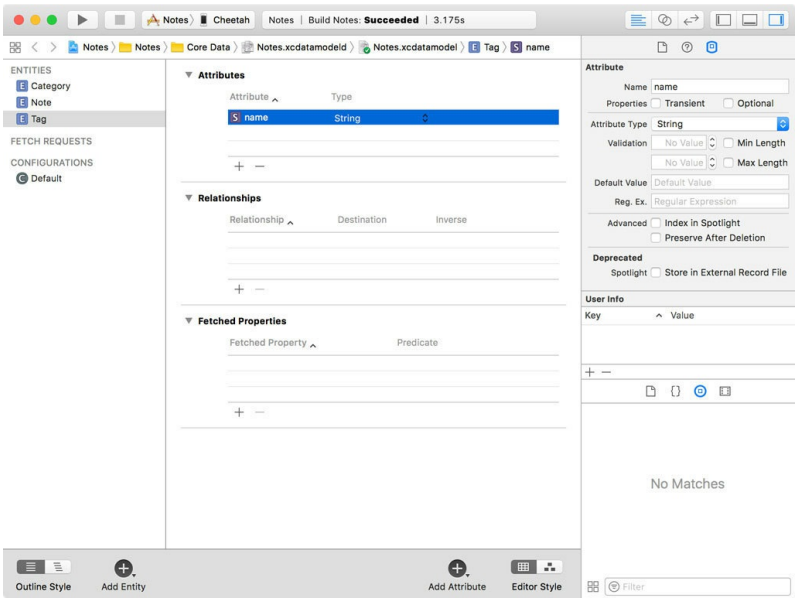
Later in this book, we add the ability to tag and categorize notes. To implement these features, we need to define relationships between entities.

Before we can define relationships, we need to create a few more entities. Create a new entity and name it **Category**. The entity has one attribute, **name** of type **String**. Select the **name** attribute and uncheck **Optional** in the **Data Model Inspector** on the right. You learned how to do this in the previous chapter.



Adding the Category Entity

Create another entity and name it **Tag**. This entity also has one attribute, **name** of type **String**. Select the **name** attribute and uncheck **Optional** in the **Data Model Inspector** on the right.



Adding the Tag Entity

Defining Relationships

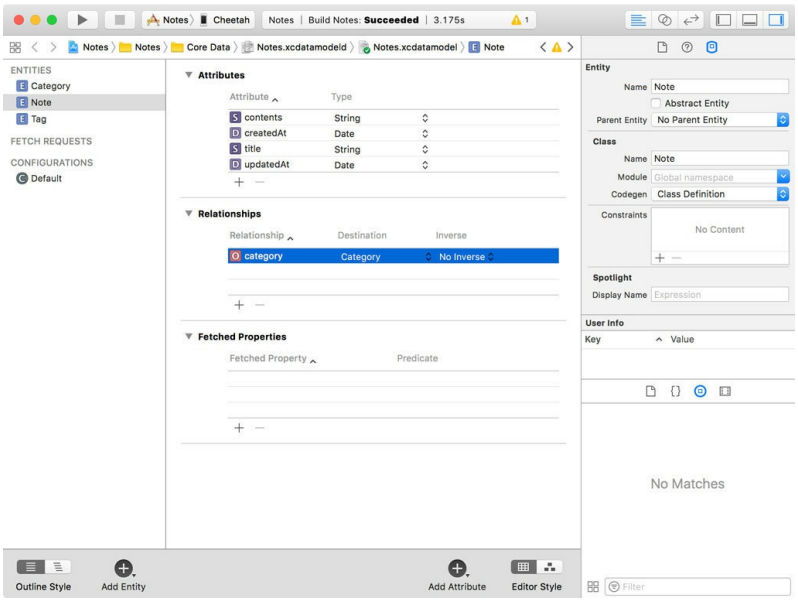
Before we start defining relationships, we need to take a moment to talk about the relationships between the entities we defined. A note can only belong to one category. But a category can have multiple notes. A note can have multiple tags and a tag can belong to multiple notes.

When working with data models, it's important to spend some time planning and reflecting what the data model should look like. I cannot emphasize this enough. Once you've decided on the data model, it's often expensive to make drastic changes to the data model. We'll learn more about what that means for Core Data when we discuss data model migrations.

In the previous chapter, we learned that an entity has properties. The properties of an entity are its attributes and relationships. We now create the relationships we defined earlier in this chapter.

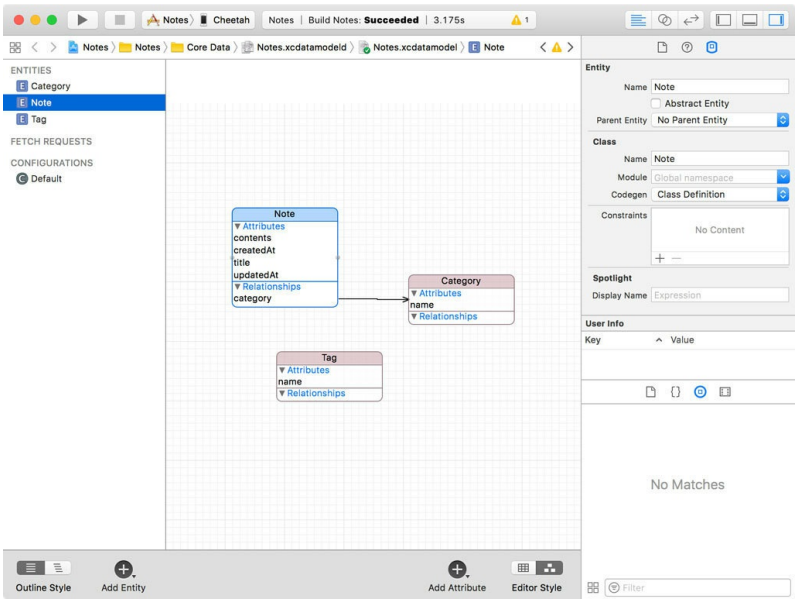
Creating a To-One Relationship

A note belongs to a category. This means that we need to create a relationship that links a note to a category. Open **Notes.xcdatamodeld** and switch to the editor's **table** style. Select the **Note** entity and click the plus button at the bottom of the **Relationships** table to add a relationship to the entity. Set **Relationship** to **category** and **Destination** to the **Category** entity. Leave **Inverse** empty for now.



Creating a Relationship

Switch to the editor's **graph** style to see the visual representation of the relationship we defined.



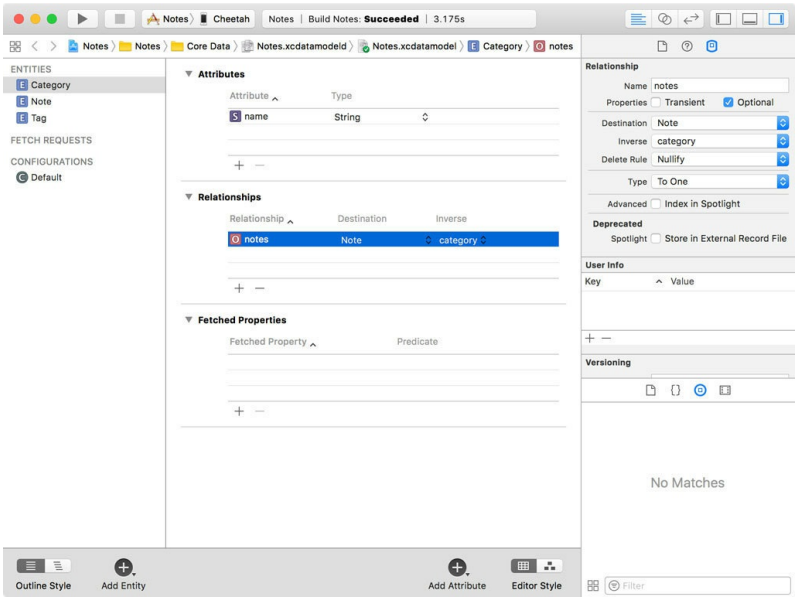
Visualizing the Relationship in the Data Model Editor

Before we move on, there are a number of details we need to discuss. The relationship we added is a **to-one** relationship, a note can belong to only one category. This is visualized in the data model graph by the single arrow pointing *from* the **Note** entity *to* the **Category** entity.

Notice that the arrow points from **Note** to **Category**. There is no arrow pointing back from **Category** to **Note**. There is no **inverse** relationship because we haven't defined one yet. This implies that the category the note belongs to doesn't know that the note belongs to the category. That's not what we want, though. A category should know what notes it contains. Right?

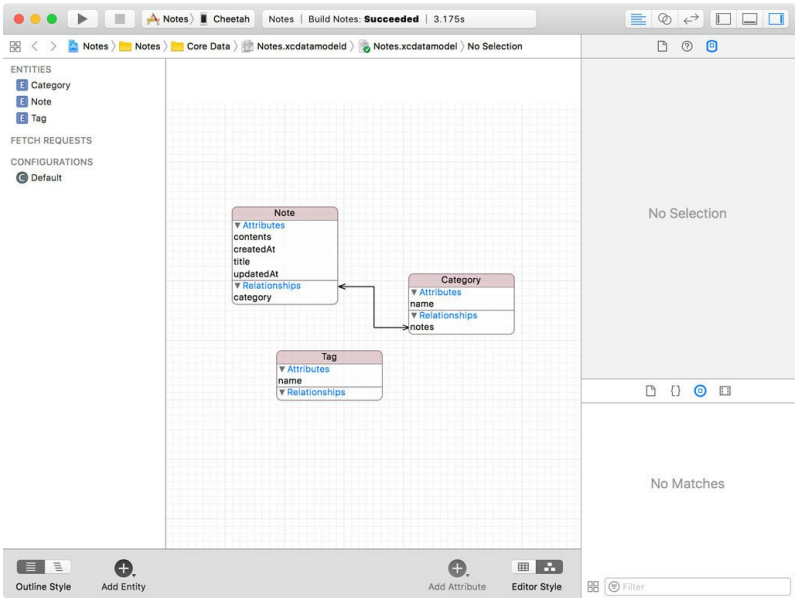
Creating an Inverse Relationship

Switch to the data model editor's table style and select the **Category** entity. Add a relationship and name it **notes**, plural. Set **Destination** to **Note** and set **Inverse** to **category**. By setting the inverse relationship to **category**, the inverse relationship of the **notes** relationship, which we left blank, is automatically set to the **category** relationship of the **Note** entity.



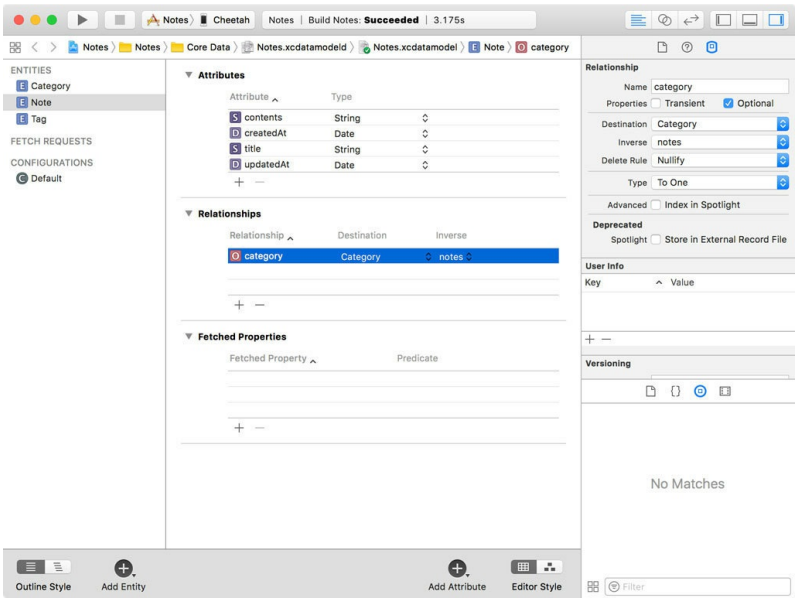
Creating an Inverse Relationship

Switch to the data model editor's graph style to see what that looks like. The connection between **Category** and **Note** contains arrows pointing to and from each entity.



Visualizing the Relationships in the Data Model Editor

Switch back to the data model editor's table style and select the **Note** entity. Notice that the inverse relationship of the **category** relationship is automatically set to **notes** because we set the inverse relationship of the **notes** relationship to **category**. Core Data is clever enough to take care of this small but important detail for us.

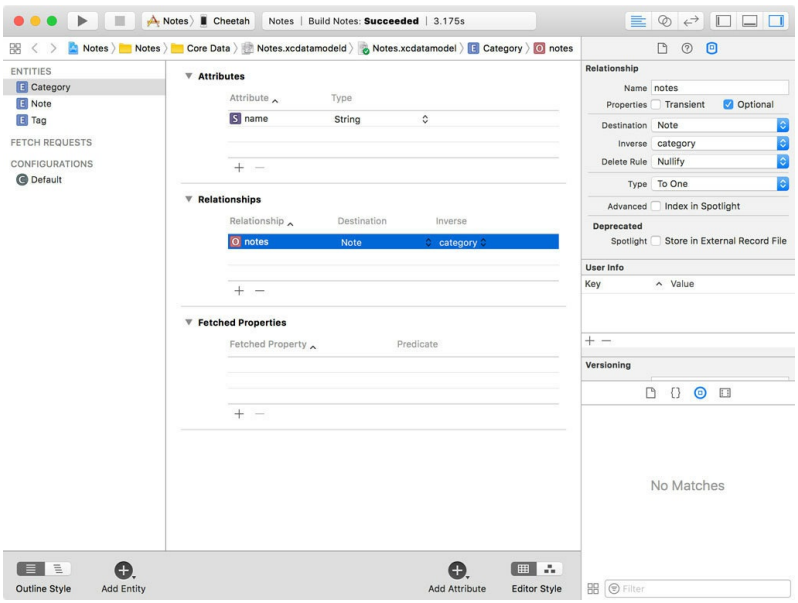


Creating an Inverse Relationship

Creating a To-Many Relationship

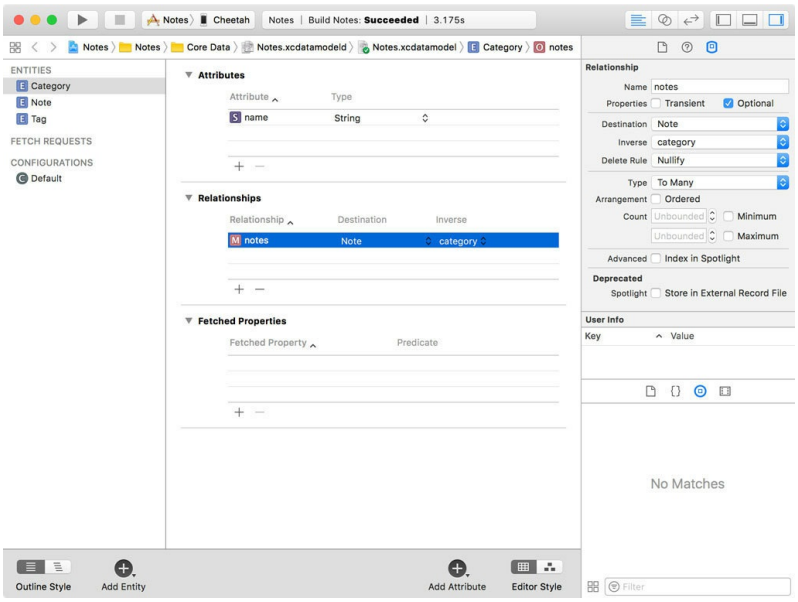
Both **category** and **notes** are to-one relationships. This severely limits the data model. A category with only one note isn't much of a category. It's time to fix that.

Select the **notes** relationship of the **Category** entity in the data model editor's table style. Open the **Data Model Inspector** in the **Utilities** pane on the right to see the details of the relationship.



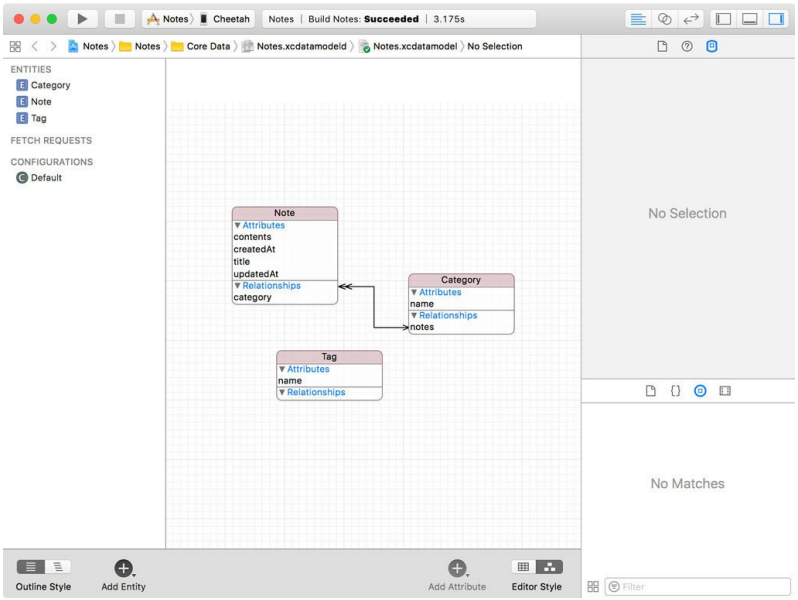
Inspecting the Relationship's Details

In the **Data Model Inspector**, you can modify the relationship's destination and inverse relationship. You can also modify the relationship's **type** or **cardinality**. Core Data supports **To-One** and **To-Many** relationships. Set **Type** to **To-Many**.



Modifying the Relationship Type

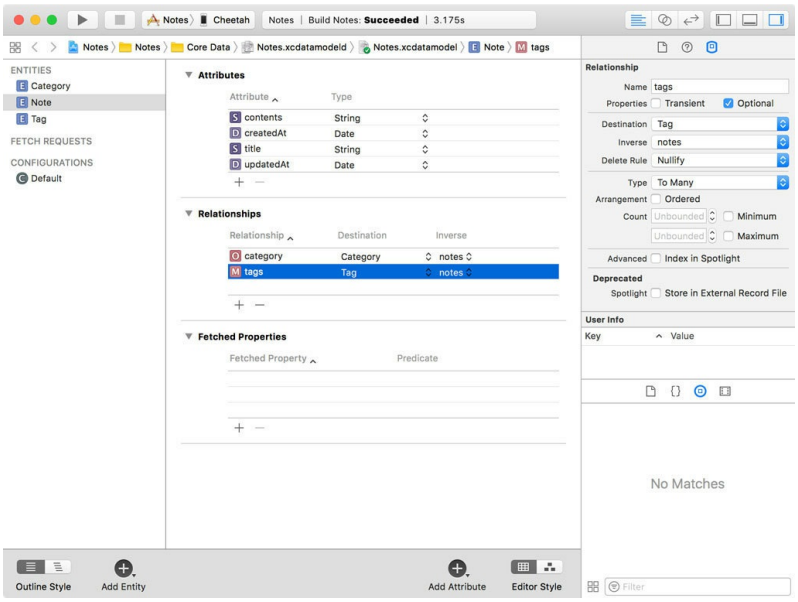
Switch back to the data model editor's graph style to see what has changed. The relationship from **Category** to **Note** now has two arrows, indicating that **notes** is a **To-Many** relationship.



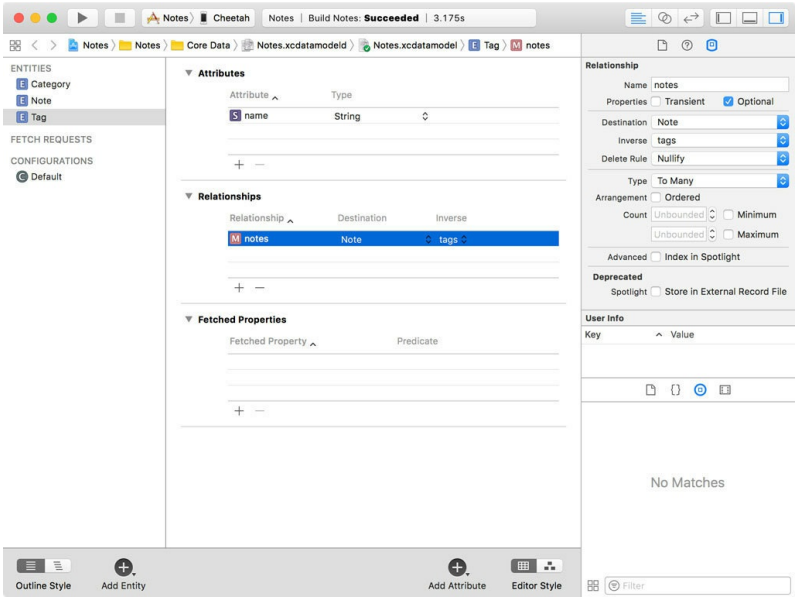
Visualizing the Relationships in the Data Model Editor

Creating a Many-To-Many Relationship

We can now focus on the relationship the **Note** entity has with the **Tag** entity. We already defined the relationships earlier in this chapter. A tag can be linked to multiple notes and a note can have multiple tags. This means we need to add a **To-Many** relationship to the **Note** entity and name it **tags** and we need to add a **To-Many** relationship to the **Tag** entity and name it **notes**.



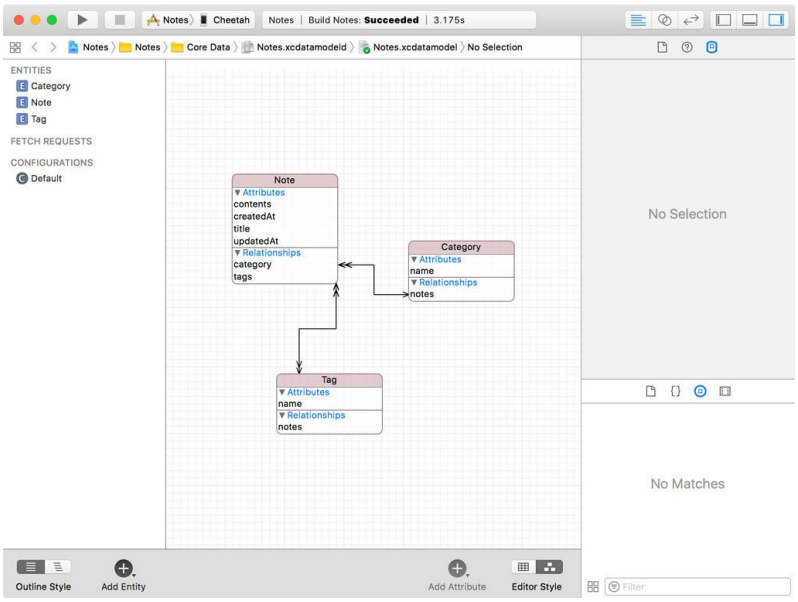
Creating a Many-To-Many Relationship



Creating a Many-To-Many Relationship

Naming conventions are very important, especially if you're working in a team. By pluralizing the relationship name of a **To-Many** relationship, the cardinality of the relationship is immediately obvious.

This is what the data model graph now looks like. The relationship that links the **Note** and **Tag** entities is a **Many-To-Many** relationship.



Creating a Many-To-Many Relationship

In the next chapter, we further discuss relationships and how we can configure relationships in the data model editor.

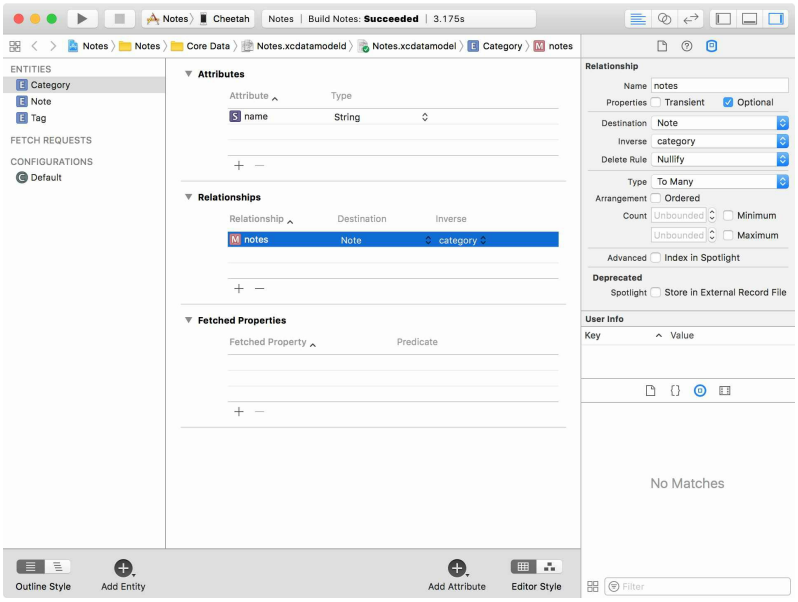
9 Configuring Relationships

Core Data is much more than a database and this becomes clear when you start working with relationships. Relationships in Core Data are powerful because the framework does a lot of the heavy lifting for us.

Delete Rules

Delete rules are one of the conveniences that make working with Core Data great. Every relationship has a delete rule. It defines what happens when the record that *owns* the relationship is deleted. Let me explain what that means.

Select the **notes** relationship of the **Category** entity and open the **Data Model Inspector** on the right.



Every relationship has a delete rule.

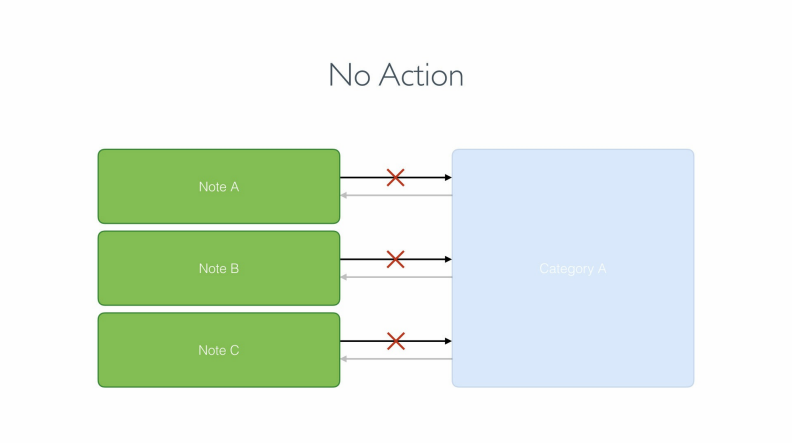
By default, the delete rule of a relationship is set to **Nullify**. Core Data supports four delete rules:

- No Action
- Nullify
- Cascade
- Deny

No Action

If the delete rule of a relationship is set to **No Action**, nothing happens when the *owner* of the relationship is deleted. Let me illustrate this with an example.

We have a category record that contains several note records. If the category record is deleted, the note records are not notified of this event. Every note record thinks it's still associated with the deleted category.

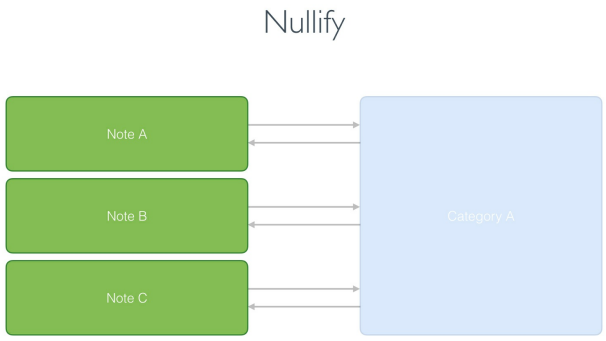


No Action Delete Rule

I've never had a need to use this delete rule. In most situations, you want to take some action when a record is deleted. And that's where the other delete rules come into play.

Nullify

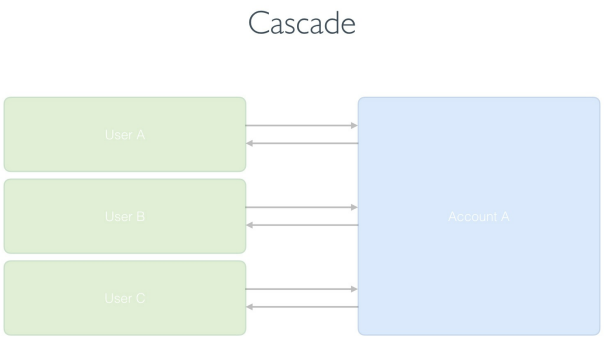
If the delete rule of a relationship is set to **Nullify**, the *destination* of the relationship is nullified when the *owner* of the relationship is deleted. If a category has several notes and the category is deleted, the relationship pointing from the notes to the category is nullified. This is the default delete rule and the delete rule you will find yourself using most often.



Nullify Delete Rule

Cascade

Cascade is useful if the data model includes one or more dependencies. For example, if an account record has a relationship to one or more user records, it may be desirable to delete every user if the account the user belongs to is deleted. In other words, the deletion of the account record *cascades* or trickles down to the user records it is linked to.



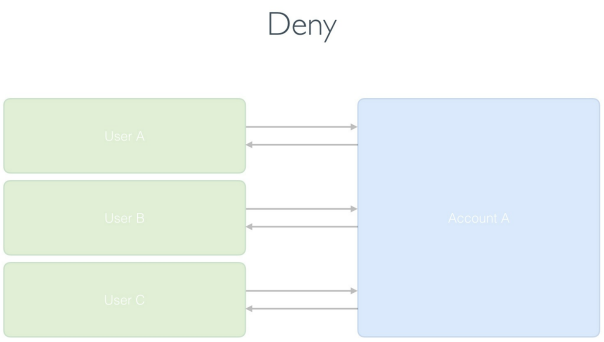
Cascade Delete Rule

If you're dealing with a many-to-many relationship, though, this is often not what you want. If a tag with several notes is deleted, you don't want to delete every note linked to the tag. The notes could be associated with other tags, for example.

Deny

Deny is another powerful and useful pattern. It's the opposite of the **Cascade** delete rule. Instead of cascading the deletion of a record, it *prevents* the deletion of a record.

For example, if an account is associated with several users, the account can only be deleted if it's no longer tied to any users. This configuration prevents the scenario in which users are no longer associated with an account.



Deny Delete Rule

Evaluating the Data Model

With the above in mind, you may be wondering if we need to change the delete rule of any of the relationships defined in the data model. Let's take a look. The default delete rule of a relationship is **Nullify**, which means every relationship we defined has its delete rule set to **Nullify**. Is that what we want?

Let's start with the relationships of the **Note** and **Category** entities. If the user deletes a note, should the application automatically delete the note's category? No. This means that **Nullify** is the correct choice for the **category** relationship of the **Note** entity.

The inverse is also true. If the user deletes a category, should the application automatically delete every note linked to that category? No. That's not the behavior the user expects. **Nullify** is the delete rule we stick with.

The same logic applies to the relationships of the **Note** and **Tag** entities. Deleting a note or a tag shouldn't result in the deletion of any records associated with the deleted record. We stick with **Nullify** for both relationships.

Another Scenario

But this isn't the only possible configuration. Let's assume a note must have a category. This means we mark the **category** relationship of the **Note** entity as required by unchecking the **Optional** checkbox in the **Data Model Inspector**. What's different? What would or should happen if a category is deleted?

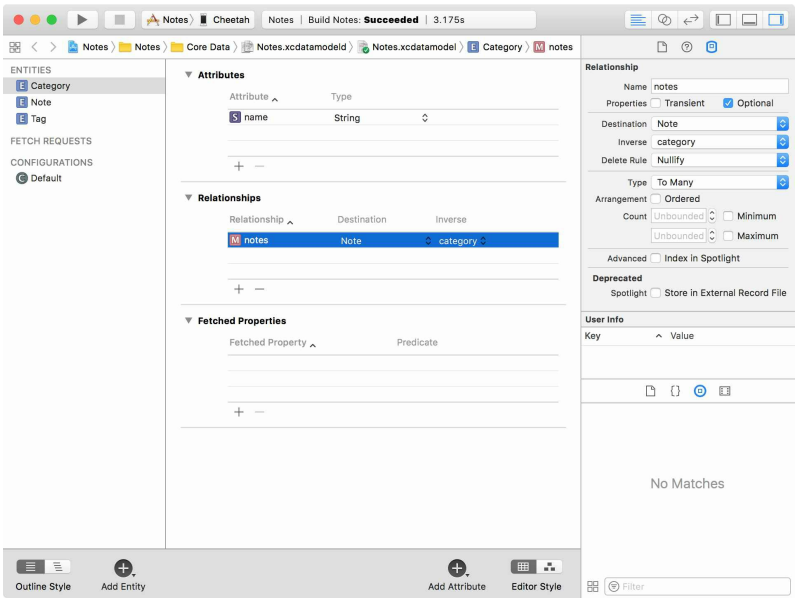
Nothing changes if the user deletes a note. The delete rule of the **category** relationship should be set to **Nullify**. But what happens if the user deletes a category and the **notes** relationship is set to **Nullify**? Every note is required to be linked to a category. To avoid that notes end up without a category, we could set the delete rule to **Deny** to make sure a category with one or more notes cannot be deleted.

While this is a possible option, it would most likely frustrate the user because it would result in an unpleasant and odd user experience. We could offer the user the option to delete every note linked to the category when the category is deleted, but that too would be an odd workaround.

With this example, I want to emphasize the importance of spending time thinking about the data model of your Core Data application. Create an outline and list the possible scenarios the user might encounter. It's easy to make changes to the data model during development. It's hard and a pain to make changes when your application is shipped and in the hands of hundreds or thousands of users.

More Configuration

If you take a closer look at the **Data Model Inspector**, you can see that Xcode offers us a few more options to configure relationships. We can, for example, define validation rules. We could, for example, define a minimum and maximum number of tags for a note record.



Configuring Relationships

There are also a number of more advanced configuration options we won't cover in this book, such as indexing and versioning options. Core Data is a very powerful framework and it has gained this power over more than ten years. It's a proven solution with many, many possibilities.

10 Working With Managed Objects

Core Data records are represented by the `NSManagedObject` class, a key class of the framework. In this chapter, you learn how to create a managed object, what classes are involved, and how a managed object is saved to a persistent store.

What Is a Managed Object

At first glance, `NSManagedObject` instances may look like glorified dictionaries. All they seem to do is manage a collection of key-value pairs. It's true that the `NSManagedObject` class is a generic class, but it implements the fundamental behavior required for Core Data model objects. Let me explain what that means.

Creating a Managed Object

Open **ViewController.swift** and add an import statement for the **Core Data** framework. In the `viewDidLoad()` method, we're going to create a managed object to learn more about the inner workings of the `NSManagedObject` class.

ViewController.swift

```
1 import CoreData
```

To create an instance of the `NSManagedObject` class, we need two ingredients:

- an entity description
- a managed object context

Entity Description

Every managed object has an entity description, an instance of the `NSEntityDescription` class. The entity description is accessible through the `entity` property of the managed object.

Earlier in this book, you learned what an entity is and we created several entities in the data model. An instance of the `NSEntityDescription` class represents an entity of the data model. As the name of the class implies, an `NSEntityDescription` instance *describes* an entity.

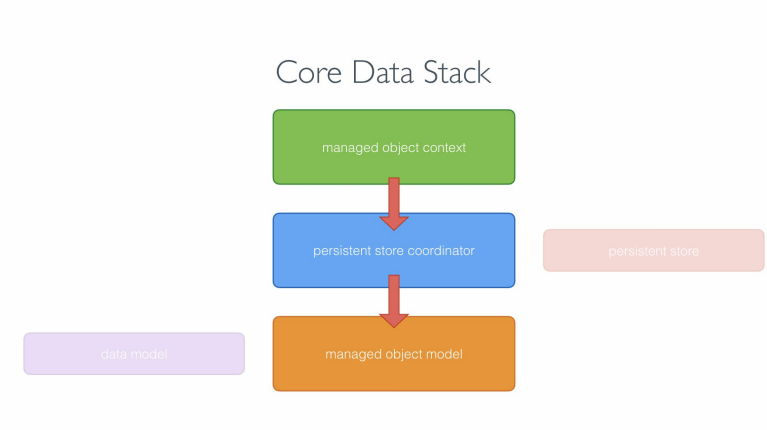
The entity description refers to a specific entity in the data model and it knows about the attributes and relationships of that entity. Every managed object is associated with an entity description.

We can create an entity description by invoking a class method of the `NSEntityDescription` class, `entity(forEntityName:in:)`. We pass this method the **name** of the entity and a **managed object context**.

ViewController.swift

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

    if let entityDescription = NSEntityDescription.entity(forEntityN\
ame: "Note", in: coreDataManager.managedObjectContext) {

    }
}
```

Why do we need to pass it a managed object context? Remember that we rarely, if ever, directly access the managed object model of the Core Data stack. We use the managed object context as a **proxy** to access the managed object model. It's the managed object model that contains the information about the entities of the data model.



The managed object context acts as a proxy for the managed object model.

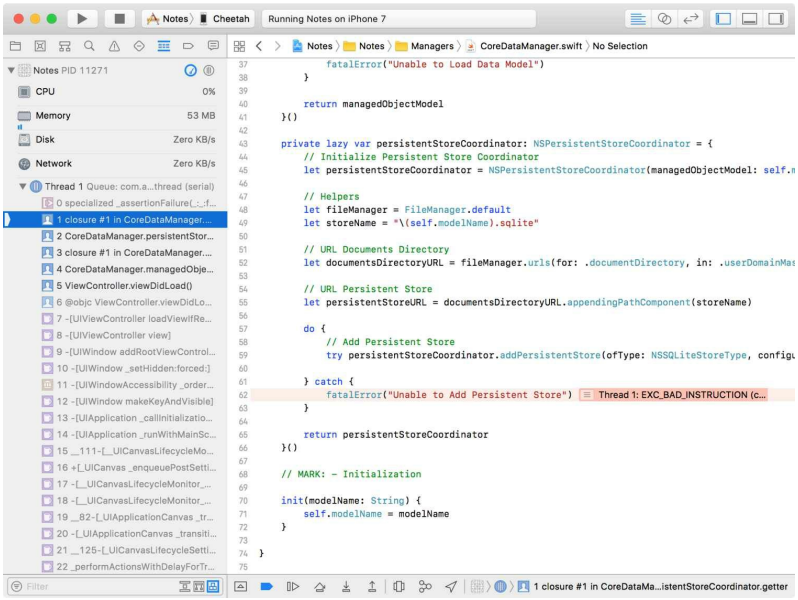
Notice that the class method of the `NSEntityDescription` class returns an optional. What is that about? Core Data needs to make sure that you can only create managed objects for entities that exist in the data model. If we pass `entity(forEntityName:in:)` a name of an entity that doesn't exist in the data model, the entity description can't be created and `entity(forEntityName:in:)` returns `nil`.

We should now have a valid entity description. Let's print the name and the properties of the entity description to the console.

ViewController.swift

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

    if let entityDescription = NSEntityDescription.entity(forEntityName: "Note", in: coreDataManager.managedObjectContext) {
        print(entityDescription.name ?? "No Name")
        print(entityDescription.properties)
    }
}
```

Run the application and inspect the output in the console. Oh. It seems the application crashed.



The persistent store coordinator was unable to add the persistent store.

The persistent store coordinator was unable to add the persistent store. If you encounter the same crash, remove the application from the device or simulator and run the application again. We explore the reason of this crash later in this book. This is what the output in the console looks like.

Console

```
Note
[(<NSAttributeDescription: 0x6000000ed980>), name contents, isOptional 1, isTransient 0, entity Note, renamingIdentifier contents, validation predicates (
), warnings (
), versionHashModifier (null)
 userInfo {
}, attributeType 700 , attributeValueClassName NSString, defaultValue (null), (<NSAttributeDescription: 0x6000000eda80>), name createdAt, isOptional 1, isTransient 0, entity Note, renamingIdentifier createdAt, validation predicates (
), warnings (
), versionHashModifier (null)
 userInfo {
```

```
}, attributeType 900 , attributeValueClassName NSDate, defaultValue \
(null), (<NSAttributeDescription: 0x6000000ed700>), name title, isOp\
tional 1, isTransient 0, entity Note, renamingIdentifier title, vali\
dation predicates (
), warnings (
), versionHashModifier (null)
 userInfo {
}, attributeType 700 , attributeValueClassName NSString, defaultValu\
e (null), (<NSAttributeDescription: 0x6000000ed900>), name updatedAt\
, isOptional 1, isTransient 0, entity Note, renamingIdentifier updat\
edAt, validation predicates (
), warnings (
), versionHashModifier (null)
 userInfo {
}, attributeType 900 , attributeValueClassName NSDate, defaultValue \
(null), (<NSRelationshipDescription: 0x600000130ea0>), name category\
, isOptional 1, isTransient 0, entity Note, renamingIdentifier categ\
ory, validation predicates (
), warnings (
), versionHashModifier (null)
 userInfo {
}, destination entity Category, inverseRelationship notes, minCount \
0, maxCount 1, isOrdered 0, deleteRule 1, (<NSRelationshipDescriptio\
n: 0x600000130e00>), name tags, isOptional 1, isTransient 0, entity \
Note, renamingIdentifier tags, validation predicates (
), warnings (
), versionHashModifier (null)
 userInfo {
}, destination entity Tag, inverseRelationship notes, minCount 0, ma\
xCount 0, isOrdered 0, deleteRule 1]
```

That looks like a lot of gibberish. If you take a closer look, though, it makes more sense. The first line tells us the name of the entity is **Note**. We can also see that the entity has four attributes and two relationships.

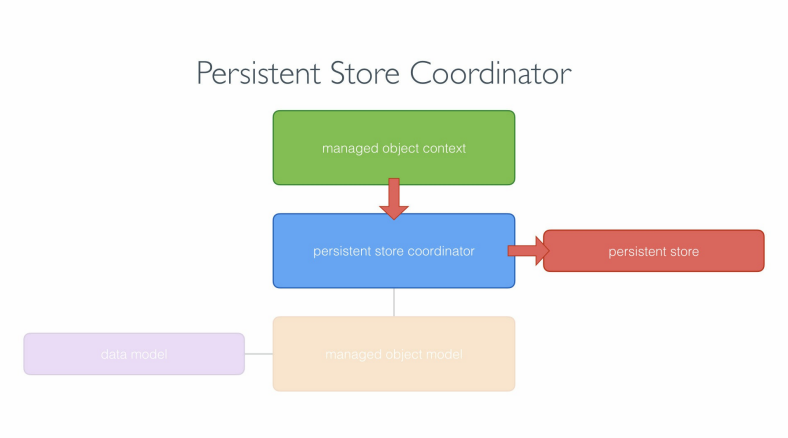
Managed Object Context

To create a managed object, we also need a managed object context. What? Why do we need another reference to a managed object context? We already referenced a managed object context to create the entity description. Remember?

A managed object is *always* associated with a managed object context. There are no exceptions to this rule. Remember that a managed object context manages a number of records or managed objects. As a developer, you primarily interact with managed objects and the managed object context they belong to.

Why is a managed object context important? Remember from earlier in this book that the persistent store coordinator bridges the gap between the persistent store and the managed object context. In the managed object context, records are created, updated, and deleted. Because the managed

object context is unaware of the persistent store, it pushes its changes to the persistent store coordinator, which updates the persistent store.



The managed object context is unaware of the persistent store. It it pushes its changes to the persistent store coordinator.

Creating a Managed Object

With an entity description and a managed object context, we now have the ingredients to create a managed object. We do this by invoking the `init(entity:insertInto:)` initializer. The result is a managed object for the **Note** entity.

ViewController.swift

```
1 // Initialize Managed Object
2 let note = NSManagedObject(entity: entityDescription, insertInto: co\
3 reDataManager.managedObjectContext)
```

The entity description and managed object context are both available as properties on the managed object. Add a print statement for the managed object, run the application again, and inspect the output in the console.

ViewController.swift

```
1  override func viewDidLoad() {
2      super.viewDidLoad()
3
4      if let entityDescription = NSEntityDescription.entity(forEntityN\
5  ame: "Note", in: coreDataManager.managedObjectContext) {
6          print(entityDescription.name ?? "No Name")
7          print(entityDescription.properties)
8
9          // Initialize Managed Object
10         let note = NSManagedObject(entity: entityDescription, insert\
11 Into: coreDataManager.managedObjectContext)
12
13         print(note)
14     }
15 }
```

```
<Note: 0x6080000954a0> (entity: Note; id: 0x608000221440 <x-coredata\
:///Note/tE5C54D25-1766-4940-A0E5-F9361FA778052> ; data: {
    category = nil;
    contents = nil;
    createdAt = nil;
    tags =      (
    );
    title = nil;
    updatedAt = nil;
})
```

The output shows us that the managed object we created doesn't have values for any of its attributes or relationships. It also tells us that no tag or category records are associated with the note record.

Working With a Managed Object

Before we can save the managed object to the persistent store, we need to set the required properties of the managed object. This means we need to set the `title`, `createdAt`, and `updateAt` attributes.

This is easy. To set a value, we invoke `setValue(_:forKey:)` and we pass in the value for the property and the name of the property.

ViewController.swift

```
// Initialize Managed Object
let note = NSManagedObject(entity: entityDescription, insertInto: co\
reDataManager.managedObjectContext)

// Configure Managed Object
note.setValue("My First Note", forKey: "title")
note.setValue(NSDate(), forKey: "createdAt")
note.setValue(NSDate(), forKey: "updatedAt")
```

If we run the application again and print the value of `note` to the console, we see that the managed object contains the data we set.

```
<Note: 0x61800009f9a0> (entity: Note; id: 0x61800003afa0 <x-coredata\
:///Note/t6B25F688-928B-4DDF-BE67-564B8FBE0DCB2> ; data: {
    category = nil;
    contents = nil;
    createdAt = "2017-07-05 12:25:59 +0000";
    tags =      (
    );
    title = "My First Note";
    updatedAt = "2017-07-05 12:25:59 +0000";
})
```

Saving the Managed Object Context

We've successfully created a managed object, a note record, and inserted it into a managed object context. At the moment, the managed object only lives in the managed object context it was inserted into. The persistent store isn't aware of the managed object we created.

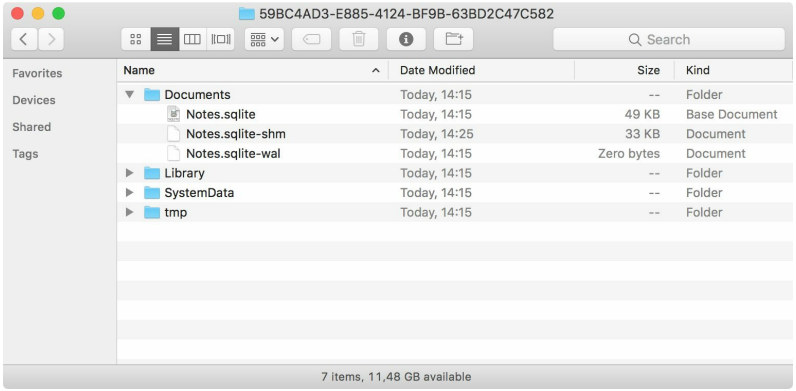
To push the managed object to the persistent store, we need to save the managed object context. Remember that a managed object context is a workspace that allows us to work with managed objects. Any changes we make to the managed object are only pushed to the persistent store if we save the managed object context the managed object belongs to. We do this by invoking `save()` on the managed object context.

ViewController.swift

```
// Initialize Managed Object
let note = NSManagedObject(entity: entityDescription, insertInto: co\
reDataManager.managedObjectContext)

// Configure Managed Object
note.setValue("My First Note", forKey: "title")
note.setValue(NSDate(), forKey: "createdAt")
note.setValue(NSDate(), forKey: "updatedAt")

print(note)

do {
    try coreDataManager.managedObjectContext.save()
} catch {
    print("Unable to Save Managed Object Context")
    print("\(error), \(error.localizedDescription)")
}
```

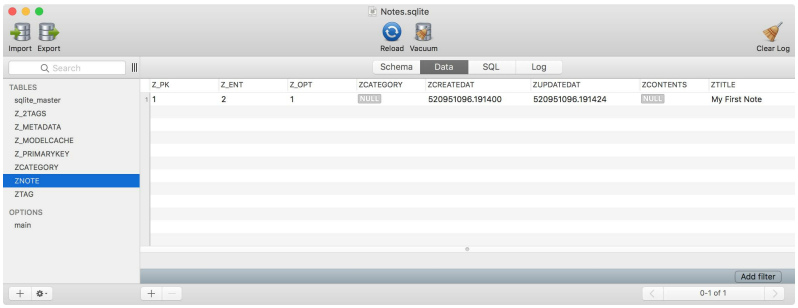
Because `save()` is a throwing method, we wrap it in a `do-catch` statement. Any errors are handled in the `catch` clause.

We can inspect the persistent store to verify that the save operation was successful. I use SimPholders to make this easier. It allows me to quickly inspect the sandbox of any application installed in the simulator.



Inspecting the Application's Container

As you can see, the **Documents** directory contains a SQLite database and, if we open it, we can see that the note is saved to the database. The structure of the database isn't something we need to worry about. That's the task of Core Data. You should never directly interact with the persistent store of a Core Data application.



Inspecting the Application's Persistent Store

Even though we only created a note record in this chapter, we learned a lot about how Core Data works under the hood. Knowing this is important for debugging problems you encounter along the way. And believe me, you *will* run into problems at some point. If you understand the fundamentals of the framework, you're in a much better position to solve any issues that arise. In the next chapter, we continue working with managed objects.

11 Subclassing NSManagedObject

In the previous chapter, we used the `NSManagedObject` class to represent and interact with records stored in the persistent store. This works fine, but the syntax is verbose, we can't take advantage of Xcode's autocompletion, and type safety is also an issue.

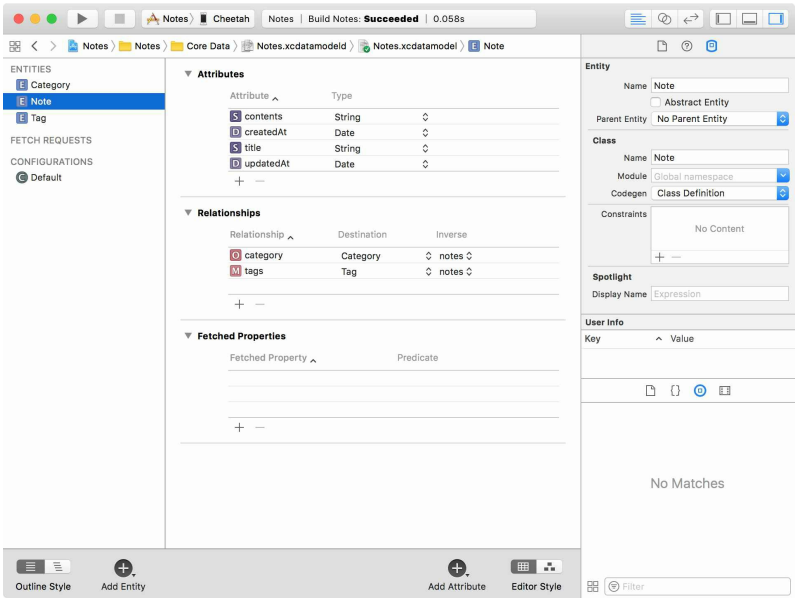
If we want to access the title of the note record, for example, we need to invoke `value(forKey:)` on the record and cast the result to a `String` object.

```
1 if let title = note.value(forKey: "title") as? String {
2     print(title)
3 }
```

This isn't pretty and it gets old very quickly. Fortunately, there's a solution, subclassing `NSManagedObject`.

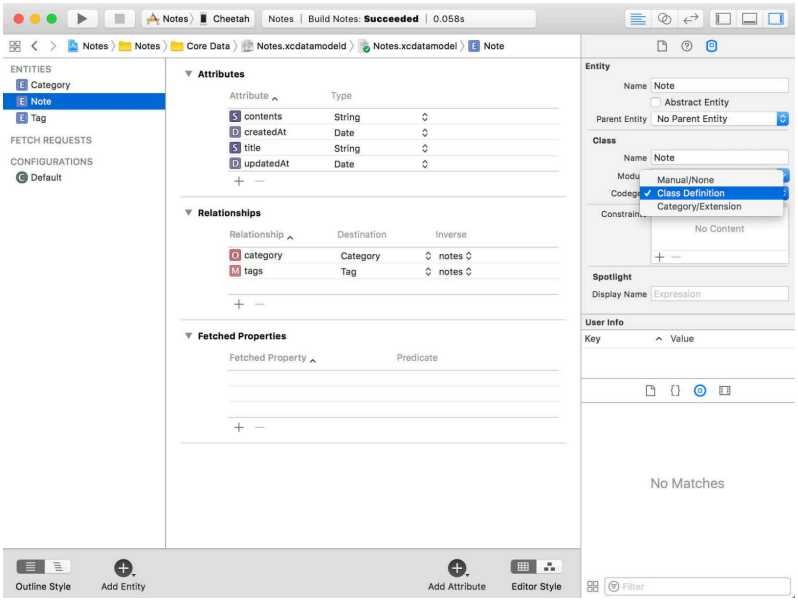
Code Generation

Prior to Xcode 8, developers needed to manually create subclasses for each entity. This is no longer necessary. As of Xcode 8, there's a much cleaner solution. Revisit the data model and select the **Note** entity. Open the **Data Model Inspector** on the right and take a look at the **Class** section. The **Codegen** field is the setting we're interested in. As of Xcode 8.1, this is set by default to **Class Definition**.



Core Data Code Generation

The possible options are **Manual/None**, **Class Definition**, and **Category/Extension**.



Core Data Code Generation Options

By setting it to **Class Definition**, a `NSManagedObject` subclass is automatically generated for us and stored in the **DerivedData** folder, not in the project itself. And that's a good thing. We don't want clutter the project with files Xcode automatically generates.

Because the default setting is **Class Definition**, it means we can already access a class named `Note` without making any changes. Swift adds even more magic to Core Data and `NSManagedObject` subclasses. Let me show you what that magic looks like.

Convenience Methods

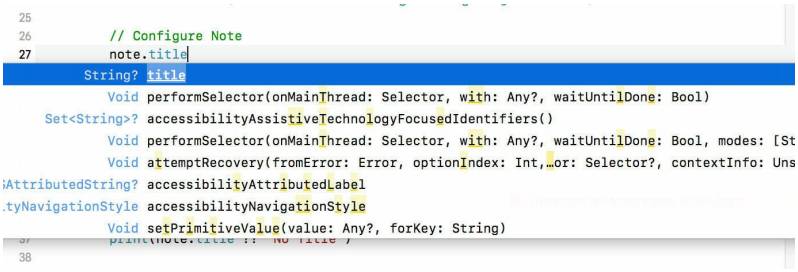
Because the `Note` class knows what entity it is linked to, the initializer no longer requires an `NSEntityDescription` instance. We only need to specify the managed object context for the managed object.

```
1 // Initialize Note
2 let note = Note(context: coreDataManager.managedObjectContext)
```

Populating the note record is concise, type safe, and we benefit from Xcode's autocompletion. This is a significant improvement.

```
1 // Configure Note
2 note.title = "My Second Note"
3 note.createdAt = Date()
4 note.updatedAt = Date()
```

But notice that the properties of the `Note` class are optionals. Why is that?



Properties Are Optionals

It's important to understand that the optionality of the properties has nothing to do with the **Optional** checkbox in the **Data Model Inspector**. These are separate issues.

Why are the properties optionals? When a managed object is created, the value of each property is set to `nil`. This is only possible if the properties are optionals. It's that simple.

I agree that this is unfortunate, but that's the current state of Core Data. The framework started its life as an Objective-C framework and this is a side effect when it's used in combination with Swift.

In Objective-C, this isn't an issue. In Swift, every property of a class or struct needs to have a valid initial value by the time the instance is created. And that means that the default value of a property of a managed object is `nil` hence the optionality.

What to Remember

What I want you to remember from this chapter is that an `NSManagedObject` subclass is automatically created for every entity as of Xcode 8.1 and that the optionality of the properties of an `NSManagedObject` subclass isn't linked to the **Optional** checkbox in the **Data Model Editor**.

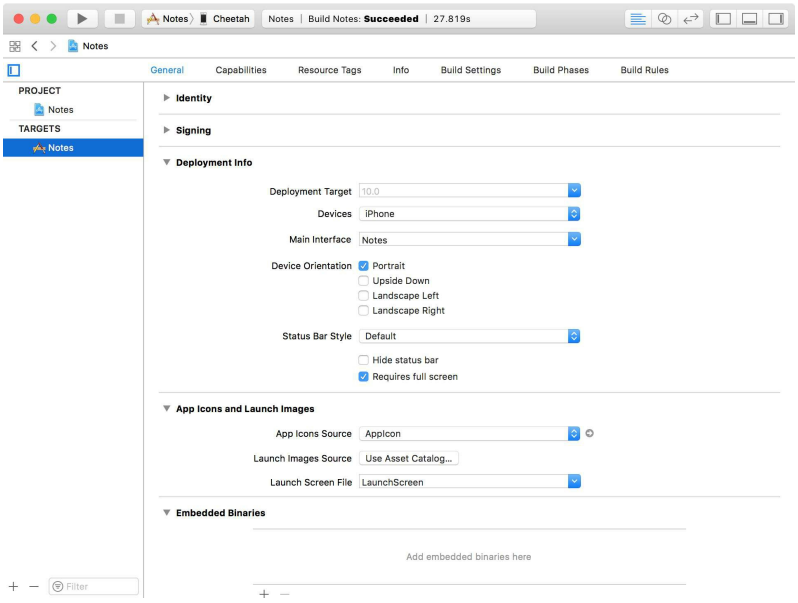
We now know enough to continue building **Notes**. In the next chapters, we add the ability to create, read, update, and delete notes.

12 Adding a Note

In the next chapters, we add the ability to create, read, update, and delete notes. Before we start, we need to make some preparations.

Target Configuration

Open the project in the **Project Navigator**, choose the **Notes** target, and select the **General** tab at the top. Set **Devices** to **iPhone**, **Device Orientation** to **Portrait**, and check **Requires full screen**.

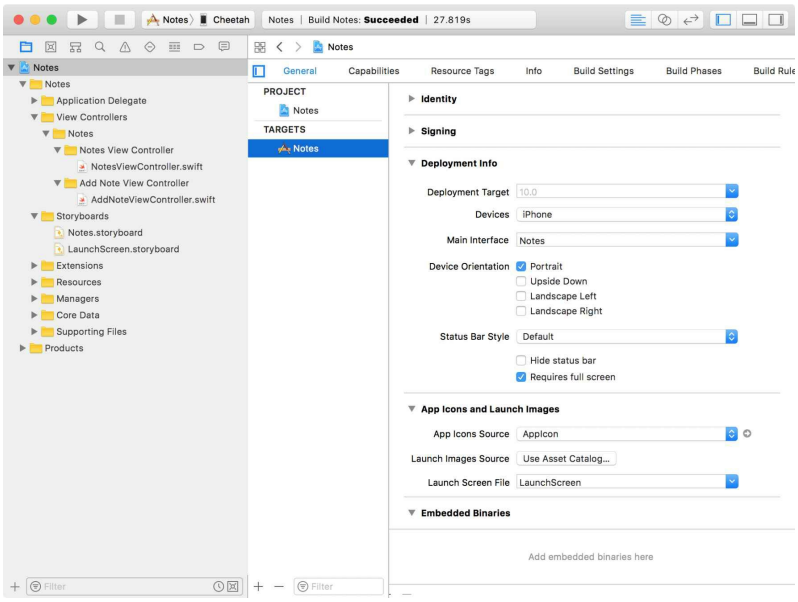


Configuring the Target

View Controllers

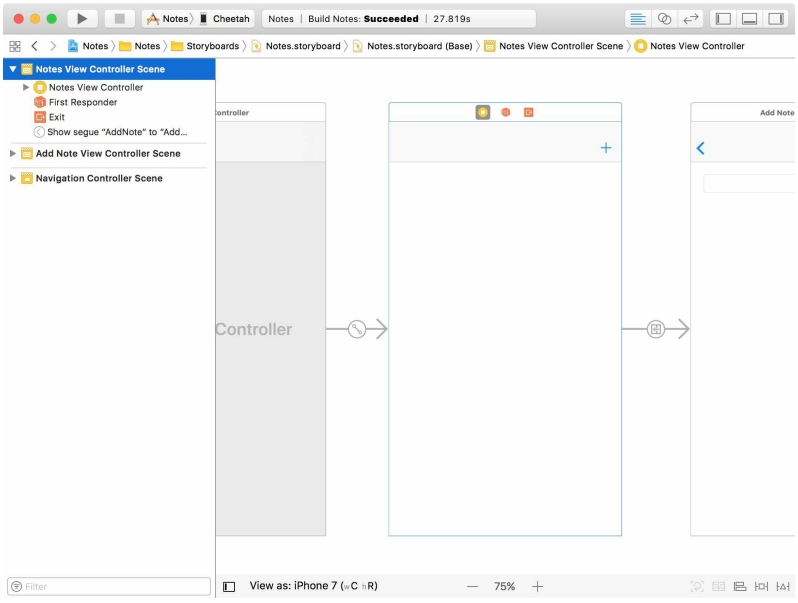
I'm not going to bother you with the implementation of the user interface and the view controllers we need. We're going to focus on the details that relate to the Core Data implementation. Open the starter project of this chapter if you'd like to follow along with me.

The project includes several changes. First, I renamed `ViewController` to `NotesViewController`. This makes more sense since the `NotesViewController` class will display the user's notes. I also renamed **Main.storyboard** to **Notes.storyboard**.



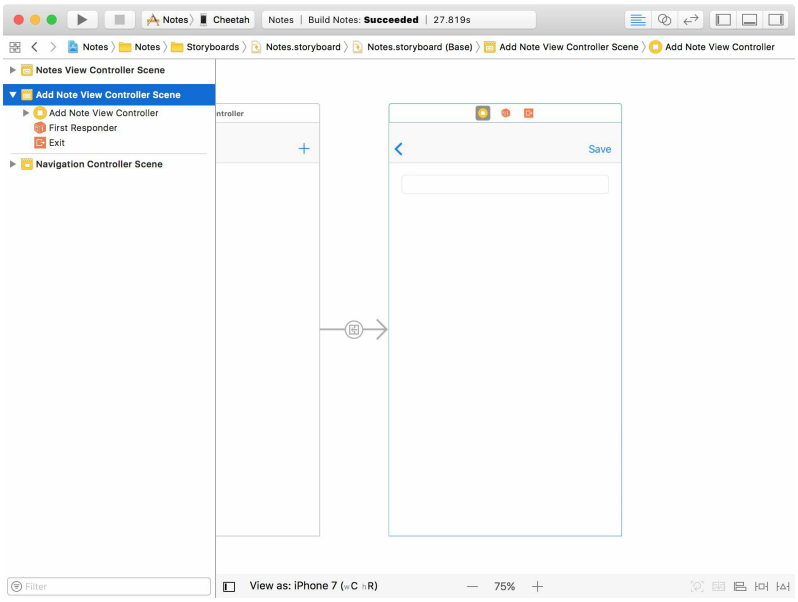
Updating the Project Structure

Second, I embedded the `NotesViewController` class in a navigation controller and added a bar button item in the top right. Tapping the bar button item takes the user to the **Add Note View Controller Scene**.



Notes View Controller Scene

Third, the `AddNoteViewController` class is responsible for adding notes and handling the user's input. It contains a text field for the title of the note and a text view for the note's contents.



Add Note View Controller Scene

With this in place, what changes do we need to make to add the ability to create notes?

Adding Notes

The `AddNoteViewController` class can only create a note if it has access to a managed object context. This means we need to pass it a reference to the Core Data manager's managed object context.

Open **AddNoteViewController.swift**, add an import statement for the **Core Data** framework, and declare a property, `managedObjectContext`, of type `NSManagedObjectContext?`.

AddNoteViewController.swift

```
1  import UIKit
2  import CoreData
3
4  class AddNoteViewController: UIViewController {
5
6      // MARK: - Properties
7
8      @IBOutlet var titleTextField: UITextField!
9      @IBOutlet var contentsTextView: UITextView!
10
11     // MARK: -
12
13     var managedObjectContext: NSManagedObjectContext?
14
15     ...
16
17 }
```

In **ViewController.swift**, we implement the `prepare(for:sender:)` method. When the segue that leads to the add note view controller is about to be

performed, we set the `managedObjectContext` property of the add note view controller.

NotesViewController.swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }

    switch identifier {
    case Segue.AddNote:
        guard let destination = segue.destination as? AddNoteViewCon\
troller else {
            return
        }

        // Configure Destination
        destination.managedObjectContext = coreDataManager.managedOb\
jectContext
    default:
        break
    }
}
```

The next step we need to take is creating and populating a note when the user taps the **Save** button. We create a note in the `save(sender:)` action.

AddNoteViewController.swift

```
@IBAction func save(sender: UIBarButtonItem) {

}
```

We first safely unwrap the value of the `managedObjectContext` property and we also make sure the title text field isn't empty. Remember that the `title` property of the **Note** entity is a required property. It can't be empty.

AddNoteViewController.swift

```
guard let managedObjectContext = managedObjectContext else { return }
guard let title = titleTextField.text, !title.isEmpty else {
    showAlert(with: "Title Missing", and: "Your note doesn't have a \
title.")
    return
}
```

If the title text field *is* empty we show an alert to the user by invoking a helper method, `showAlert(with:and:)`. This method is implemented in an extension for `UIViewController`.

UIViewController.swift

```
import UIKit

extension UIViewController {

    // MARK: - Alerts

    func showAlert(with title: String, and message: String) {
        // Initialize Alert Controller
        let alertController = UIAlertController(title: title, messag\
e: message, preferredStyle: .alert)

        // Configure Alert Controller
        alertController.addAction(UIAlertAction(title: "OK", style: \
.default, handler: nil))

        // Present Alert Controller
        present(alertController, animated: true, completion: nil)
    }

}
```

Using the managed object context, we create a `Note` instance and populate it with the data the user has entered.

AddNoteViewController.swift

```
// Create Note
let note = Note(context: managedObjectContext)

// Configure Note
note.createdAt = Date()
note.updatedAt = Date()
note.title = titleTextField.text
note.contents = contentsTextView.text
```

We pop the add note view controller from the navigation stack to return to the notes view controller.

AddNoteViewController.swift

```
// Pop View Controller
_ = navigationController?.popViewController(animated: true)
```

That's it. That's all it takes to create a note. This is the implementation of the `save(sender:)` method.

AddNoteViewController.swift

```
@IBAction func save(sender: UIBarButtonItem) {
    guard let managedObjectContext = managedObjectContext else { ret\
urn }
    guard let title = titleTextField.text, !title.isEmpty else {
        showAlert(with: "Title Missing", and: "Your note doesn't hav\
e a title.")
```

```
7           return
8       }
9
10      // Create Note
11      let note = Note(context: managedObjectContext)
12
13      // Configure Note
14      note.createdAt = Date()
15      note.updatedAt = Date()
16      note.title = titleTextField.text
17      note.contents = contentsTextView.text
18
19      // Pop View Controller
20      _ = navigationController?.popViewController(animated: true)
21  }
```

Before you run the application, make sure you delete the application first. We want to start with a clean slate. Tap the bar button item, fill out the title text field and the contents text view, and tap **Save**.

Because the notes view controller doesn't display the list of notes yet, we can't verify that everything is working. We'll fix that later.

But we have another problem. If we terminate the application, the note we created is lost because we haven't pushed it to the persistent store. In other words, we haven't saved the changes of the managed object context. We resolve this issue in the next chapter.

13 Don’t Forget to Save

You may be wondering why we didn’t save the note immediately after creating it. That’s a fair question. We could have. But why would we? Why would we push the changes of the managed object context to the persistent store every time something changes in the managed object context? That’s a waste of resources and it may even impact performance, depending on the complexity of the operation.

A common approach is saving the changes of the managed object context when the application is about to enter the background and before it’s terminated. This is very easy to implement.

Revisiting the Core Data Manager

Open **CoreDataManager.swift** and revisit the designated initializer, `init(modelName:)`. We invoke a helper method in the initializer, `setupNotificationHandling()`. I prefer to keep initializers short and concise hence the helper method.

CoreDataManager.swift

```
1 init(modelName: String) {
2     self.modelName = modelName
3 
4     setupNotificationHandling()
5 }
```

In `setupNotificationHandling()`, we add the Core Data manager instance as an observer of two notifications sent by the `UIApplication` singleton:

- `UIApplicationWillTerminate`
- `UIApplicationDidEnterBackground`

CoreDataManager.swift

```
1  private func setupNotificationHandling() {
2      let notificationCenter = NotificationCenter.default
3      notificationCenter.addObserver(self,
4                                     selector: #selector(saveChanges(_\
5  :)),
6                                     name: Notification.Name.UIApplica\
7  tionWillTerminate,
8                                     object: nil)
9 
10     notificationCenter.addObserver(self,
```

```
11                                      selector: #selector(saveChanges(_\
12 :)),
13                                      name: Notification.Name.UIApplica\
14 tionDidEnterBackground,
15                                      object: nil)
16 }
```

When the Core Data manager receives one of these notifications, the `saveChanges(_:)` method is invoked. In this method, we invoke another helper method, `saveChanges()`.

CoreDataManager.swift

```
1 @objc func saveChanges(_ notification: Notification) {
2     saveChanges()
3 }
```

Saving Changes

The save operation takes place in the `saveChanges()` method.

CoreDataManager.swift

```
1 private func saveChanges() {
2
3 }
```

We first ask the managed object context if it has changes we need to push to the persistent store. We do this by asking for the value of its `hasChanges` property. We exit early if no changes need to be pushed to the persistent store.

CoreDataManager.swift

```
1 guard managedObjectContext.hasChanges else { return }
```

If the managed object context *has* changes we need to push, we invoke `save()` on the managed object context in a `do-catch` statement. Remember that `save()` is a throwing method. If the save operation fails, we print the error to the console.

CoreDataManager.swift

```
1 do {
2     try managedObjectContext.save()
3 } catch {
4     print("Unable to Save Managed Object Context")
5     print("\(error), \(error.localizedDescription)")
6 }
```

This is what the implementation of `saveChanges()` looks like.

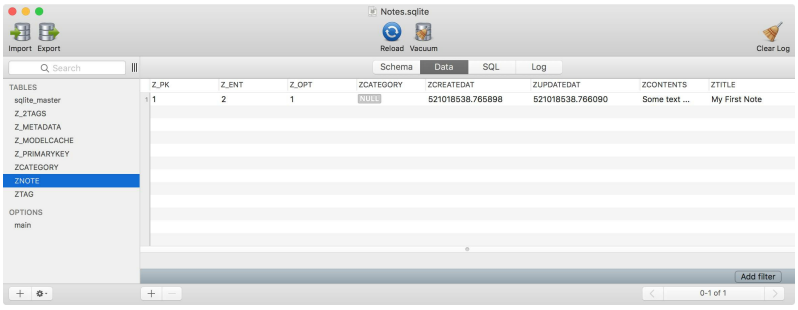
CoreDataManager.swift

```
1  private func saveChanges() {
2      guard managedObjectContext.hasChanges else { return }
3
4      do {
5          try managedObjectContext.save()
6      } catch {
7          print("Unable to Save Managed Object Context")
8          print("\(error), \(error.localizedDescription)")
9      }
10 }
```

Because the save operation takes place when the application isn't in the foreground, it isn't useful to notify the user if the save operation failed. However, that doesn't mean that you can ignore any errors that are thrown when something goes wrong. It's recommended to notify the user at some point that a problem occurred.

Build and Run

Let's see if this works. We start with a clean installation of the application and create a new note. We then push the application to the background and inspect the contents of the persistent store.



Pushing Changes to the Persistent Store

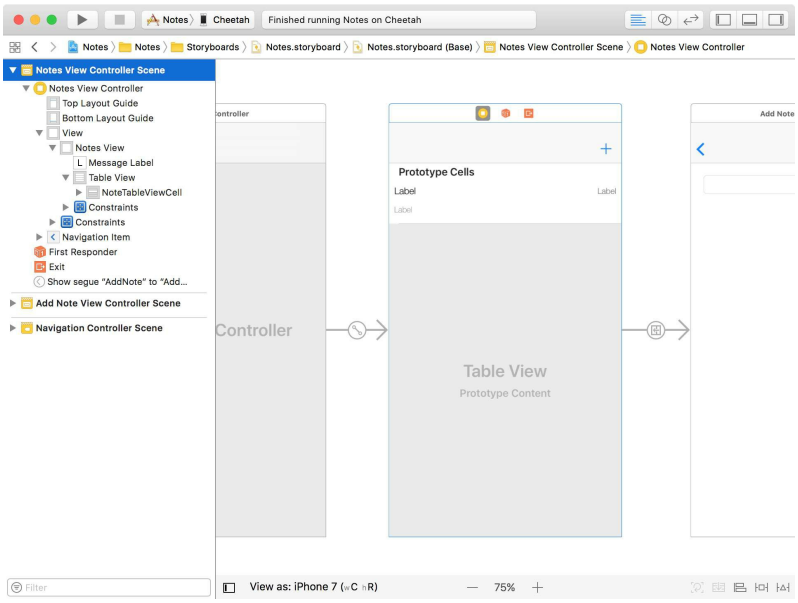
That seems to work fine. In the next chapter, you learn how to fetch the notes from the persistent store and display them in the notes view controller. That's going to be an important and interesting chapter.

14 Fetch Those Notes

In this chapter, we fetch the user's notes from the persistent store and display them in a table view. The notes view controller is in charge of these tasks.

Before We Start

I've already updated the storyboard with a basic user interface. Let me walk you through it. The view of the notes view controller contains a label, for displaying a message to the user, and a table view, for listing the user's notes. The label and the table view are embedded in another view, the notes view. The reason for this becomes clear later in this book. Don't worry about it for now.



Notes View Controller

The table view has one prototype cell of type `NoteTableViewCell`. The `NoteTableViewCell` class defines three outlets, a label for the title of the note, a label that displays the time and date when the note was last updated, and a label for displaying a preview of the note's contents.

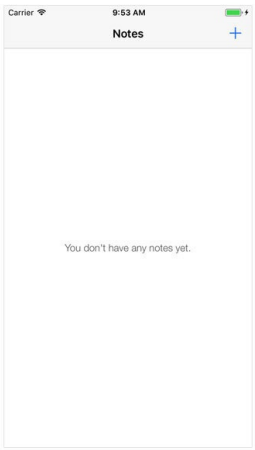
NoteTableViewCell.swift

```
1 import UIKit
2
3 class NoteTableViewCell: UITableViewCell {
4
5     // MARK: - Static Properties
6
7     static let reuseIdentifier = "NoteTableViewCell"
8
```

```
9       // MARK: - Properties
10
11      @IBOutlet var titleLabel: UILabel!
12      @IBOutlet var contentsLabel: UILabel!
13      @IBOutlet var updatedAtLabel: UILabel!
14
15      // MARK: - Initialization
16
17      override func awakeFromNib() {
18          super.awakeFromNib()
19      }
20
21  }
```

As you may have guessed, the notes view controller is the delegate and data source of the table view.

If we run the application, we see a message that tells us we don't have any notes yet despite the fact we successfully created a note in the previous chapter. Let's fix that.



You don't have any notes yet.

Fetching Notes

To display the user's notes, we first need to fetch them from the persistent store. Open **NotesViewController.swift** and add an import statement for the **Core Data** framework.

NotesViewController.swift

```
1 import UIKit
2 import CoreData
```

We declare a property for storing the notes we're going to fetch from the persistent store. We name the property `notes` and it should be of type `[Note]?`. We also define a property observer because we want to update the user interface every time the value of the `notes` property changes. In the property observer, we invoke a helper method, `updateView()`.

NotesViewController.swift

```
private var notes: [Note]? {
    didSet {
        updateView()
    }
}
```

In `viewDidLoad()`, we fetch the notes from the persistent store by invoking another helper method, `fetchNotes()`.

NotesViewController.swift

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

    title = "Notes"

    setupView()

    fetchNotes()
}
```

In the `fetchNotes()` method, we fetch the user's notes from the persistent store.

NotesViewController.swift

```
private func fetchNotes() {

}
```

The first ingredient we need is a fetch request. Whenever you need information from the persistent store, you need a fetch request, an instance of the `NSFetchRequest` class.

NotesViewController.swift

```
// Create Fetch Request
let fetchRequest: NSFetchRequest<Note> = Note.fetchRequest()
```

Notice that we specify the type we expect from the fetch request. The compiler takes care of the nitty-gritty details for us.

We want to sort the notes based on the value of the `updatedAt` property. In other words, we want to show the most recently updated note at the top of the table view. For that to work, we need to tell the fetch request how it should sort the results it receives from the persistent store.

We create a sort descriptor, an instance of the `NSSortDescriptor` class, and set the `sortDescriptors` property of the fetch request. The `sortDescriptors` property is an array, which means we could specify multiple sort descriptors. The sort descriptors are evaluated based on the order in which they appear in the array.

NotesViewController.swift

```
1 // Configure Fetch Request
2 fetchRequest.sortDescriptors = [NSSortDescriptor(key: #keyPath(Note.\
3 updatedAt), ascending: false)]
```

Remember that we never directly access the persistent store. We execute the fetch request using the managed object context of the Core Data manager. We wrap the code for executing the fetch request in a closure, which is the argument of the `performAndWait(_:)` method of the `NSManagedObjectContext` class. What's going on here?

NotesViewController.swift

```
1 // Perform Fetch Request
2 coreDataManager.managedObjectContext.performAndWait {
3
4 }
```

We take a closer look at the reasons for doing this later in this book. What you need to remember for now is that by invoking the fetch request in the closure of the `performAndWait(_:)` method, we access the managed object context on the thread it's associated with. Don't worry if that makes no sense yet. It will click once we discuss threading much later in this book.

What you need to understand now is that the closure of the `performAndWait(_:)` method is executed synchronously hence the **wait** keyword in the method name. It blocks the thread the method is invoked from, the main thread in this example. That isn't an issue, though. Core Data is performant enough that we can trust that this isn't a problem for now. We can optimize this later should we run into performance issues.

Executing a fetch request is a throwing operation, which is why we wrap it in a `do-catch` statement. To execute the fetch request, we invoke `execute()` on the fetch request, a throwing method. We update the `notes` property with the results of the fetch request and reload the table view. If any errors pop up, we print them to the console.

NotesViewController.swift

```
// Perform Fetch Request
coreDataManager.managedObjectContext.performAndWait {
    do {
        // Execute Fetch Request
        let notes = try fetchRequest.execute()

        // Update Notes
        self.notes = notes

        // Reload Table View
        self.tableView.reloadData()

    } catch {
        let fetchError = error as NSError
        print("Unable to Execute Fetch Request")
        print("\(fetchError), \(fetchError.localizedDescription)")
    }
}
```

This is what the `fetchNotes()` method looks like.

NotesViewController.swift

```
private func fetchNotes() {
    // Create Fetch Request
    let fetchRequest: NSFetchRequest<Note> = Note.fetchRequest()

    // Configure Fetch Request
    fetchRequest.sortDescriptors = [NSSortDescriptor(key: #keyPath(N\
ote.updatedAt), ascending: false)]

    // Perform Fetch Request
    coreDataManager.managedObjectContext.performAndWait {
        do {
            // Execute Fetch Request
            let notes = try fetchRequest.execute()

            // Update Notes
            self.notes = notes

            // Reload Table View
            self.tableView.reloadData()

        } catch {
            let fetchError = error as NSError
            print("Unable to Execute Fetch Request")
            print("\(fetchError), \(fetchError.localizedDescription)\
")
        }
    }
}
```

That was probably one of the most complicated sections of this book. Make sure you understand the what and why of the fetch request. Read this chapter again if necessary because it's important that you understand what's going on. Remember that you can ignore the `performAndWait(_:)` method for now, but make sure you understand how to create and execute a fetch request.

Displaying Notes

Before we move on, I want to implement a computed property that tells us if we have notes to display. The implementation of the `hasNotes` property is pretty straightforward.

NotesViewController.swift

```
1 private var hasNotes: Bool {
2     guard let notes = notes else { return false }
3     return notes.count > 0
4 }
```

In the `updateView()` method, we use the value of the `hasNotes` property to update the user interface.

NotesViewController.swift

```
1 private func updateView() {
2     tableView.isHidden = !hasNotes
3     messageLabel.isHidden = hasNotes
4 }
```

Last but not least, we need to update the implementation of the `UITableViewDataSource` protocol. The implementation of `numberOfSections(in:)` is easy. The application returns `1` if it has notes, otherwise it returns `0`.

NotesViewController.swift

```
1 func numberOfSections(in tableView: UITableView) -> Int {
2     return hasNotes ? 1 : 0
3 }
```

The same is true for the implementation of `tableView(_:numberOfRowsInSection:)`. The application returns the number of notes if it has notes to display, otherwise it returns `0`.

NotesViewController.swift

```
1 func tableView(_ tableView: UITableView, numberOfRowsInSection secti\
2 on: Int) -> Int {
3     guard let notes = notes else { return 0 }
4     return notes.count
5 }
```

The implementation of `tableView(_:cellForRowAt:)` is more interesting. We first fetch the note that corresponds with the value of the `indexPath` parameter. We then dequeue a note table view cell and we populate the table view cell with the data of the note.

NotesViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    // Fetch Note
    guard let note = notes?[indexPath.row] else {
        fatalError("Unexpected Index Path")
    }

    // Dequeue Reusable Cell
    guard let cell = tableView.dequeueReusableCell(withIdentifier: N\
oteTableViewCell.reuseIdentifier, for: indexPath) as? NoteTableViewC\
ell else {
        fatalError("Unexpected Index Path")
    }

    // Configure Cell
    cell.titleLabel.text = note.title
    cell.contentsLabel.text = note.contents
    cell.updatedAtLabel.text = updatedAtDateFormatter.string(from: n\
ote.updatedAt)

    return cell
}
```

But it seems that we have a problem. Because we're dealing with Core Data, the type of the `updatedAt` property is `NSDate?`. The date formatter we use to convert the date to a string doesn't like that.

Running Into Type Issues

The solution is simple. We create an extension for the `Note` class and define computed properties for the `updatedAt` and `createdAt` properties, which return a `Date` instance. Create a new group, **Extensions**, in the **Core Data** group and add a file named **Note.swift**. We import **Foundation** and create an extension for the `Note` class. The implementation of the computed properties is straightforward.

Note.swift

```
import Foundation

extension Note {

    var updatedAtAsDate: Date {
        guard let updatedAt = updatedAt else { return Date() }
        return Date(timeIntervalSince1970: updatedAt.timeIntervalSin\
ce1970)
    }

    var createdAtAsDate: Date {
        guard let createdAt = createdAt else { return Date() }
        return Date(timeIntervalSince1970: createdAt.timeIntervalSin\
ce1970)
```

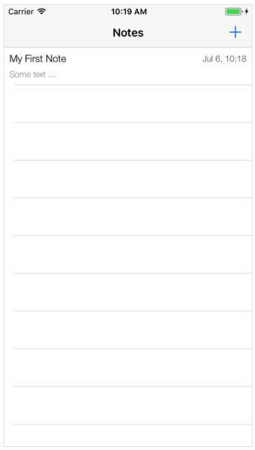
```
    }

}
```

We can now update the implementation of `tableView(_:cellForRowAt:)`.

NotesViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    // Fetch Note
    guard let note = notes?[indexPath.row] else {
        fatalError("Unexpected Index Path")
    }

    // Dequeue Reusable Cell
    guard let cell = tableView.dequeueReusableCell(withIdentifier: N\
oteTableViewCell.reuseIdentifier, for: indexPath) as? NoteTableViewC\
ell else {
        fatalError("Unexpected Index Path")
    }

    // Configure Cell
    cell.titleLabel.text = note.title
    cell.contentsLabel.text = note.contents
    cell.updatedAtLabel.text = updatedAtDateFormatter.string(from: n\
ote.updatedAtAsDate)

    return cell
}
```

And with that change, we're ready to take the application for a spin. You should now see the notes of the user displayed in the table view.



Displaying the User's Notes

In this chapter, you learned how to fetch data from the persistent store. In the next chapter, you learn how to update notes and automatically update the table view.

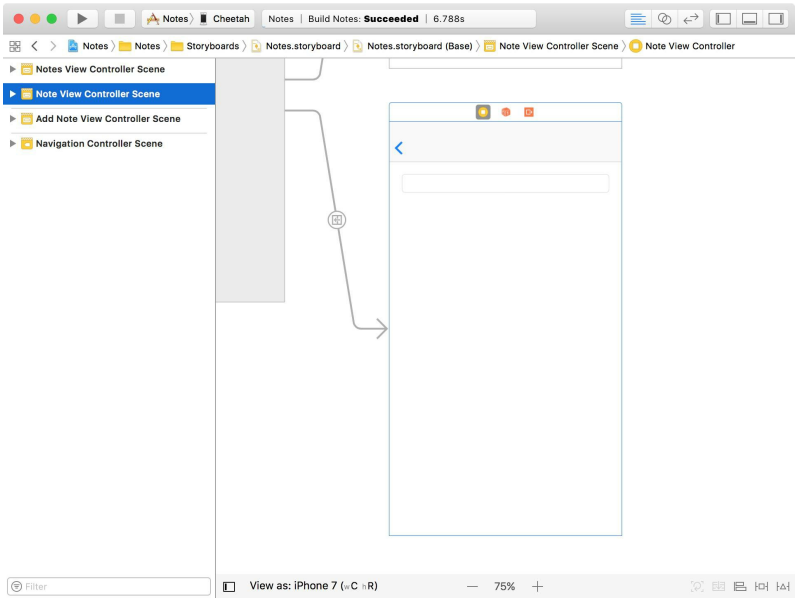
15 Fix That Mistake

The application we're building wouldn't be very useful if it didn't include the ability to edit notes. Would it? If the user taps a note in the notes view controller, they should be able to modify the title and contents of the note.

Before We Start

I've already created the `NoteViewController` class for this purpose. The note view controller is responsible for updating notes. We could have used the `AddNoteViewController` class for this, but I usually create separate view controllers for adding and updating records. That's merely a personal choice.

Unsurprisingly, the user interface of the `NoteViewController` class is very similar to that of the `AddNoteViewController` class. There's a text field for the title of the note and a text view for the contents of the note.



Note View Controller

Passing a Note

The first step we need to take is pass the note the user wants to update to the note view controller. Open **NoteViewController.swift** and declare a property, `note`, of type `Note?`. We could use an implicitly unwrapped optional, but I prefer to play it safe by using an optional. In general, I avoid implicitly unwrapped optionals whenever possible with the exception of outlets.

NoteViewController.swift

```
import UIKit

class NoteViewController: UIViewController {

    // MARK: - Properties

    @IBOutlet var titleTextField: UITextField!
    @IBOutlet var contentsTextView: UITextView!

    // MARK: -

    var note: Note?

    ...

}
```

I hope it's clear *why* we need to use an optional. Every stored property of a class or struct needs to have a valid value by the time the instance is created. This leaves us no option but to use an optional.

Open **NotesViewController.swift**. We pass the note from the notes view controller to the note view controller in the `prepare(for:sender:)` method of the notes view controller. We make sure the destination view controller is an instance of the `NoteViewController` class, fetch the note that corresponds with the currently selected row of the table view, and pass the note to the note view controller.

NotesViewController.swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }

    switch identifier {
    case Segue.AddNote:
        ...
    case Segue.Note:
        guard let destination = segue.destination as? NoteViewContro\
ller else {
            return
        }

        guard let indexPath = tableView.indexPathForSelectedRow, let\
 note = notes?[indexPath.row] else {
            return
        }

        // Configure Destination
        destination.note = note
    default:
        break
    }
}
```

Populating the Note View Controller

It's time to revisit **NoteViewController.swift**. We need to populate the user interface of the note view controller with the contents of the note. In `viewDidLoad()`, we invoke `setupView()`, a helper method.

NoteViewController.swift

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

    title = "Edit Note"

    setupView()
}
```

In `setupView()`, we invoke two other helper methods:

- `setupTitleTextField()`
- `setupContentsTextView()`

NoteViewController.swift

```
private func setupView() {
    setupTitleTextField()
    setupContentsTextView()
}
```

In these helper methods, we set the text field and the text view with the data of the note.

NoteViewController.swift

```
private func setupTitleTextField() {
    // Configure Title Text Field
    titleTextField.text = note?.title
}

private func setupContentsTextView() {
    // Configure Contents Text View
    contentsTextView.text = note?.contents
}
```

Updating a Note

Updating the note is easy. We don't even need a save button. We simply update the note in the `viewWillDisappear(_:)` method of the note view controller. Not having a save button is a very nice feature from a user's perspective. The user has the impression that every change they make is automatically saved. And that's what happens behind the scenes.

NoteViewController.swift

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    // Update Note
    if let title = titleTextField.text, !title.isEmpty {
        note?.title = title
    }

    note?.updatedAt = Date()
    note?.contents = contentsTextView.text
}
```

Notice that we check if the title text field isn't empty before updating the note record. We need to make sure the note has a title because the `title` property is required. We also update the values of the `updatedAt` and `contents` properties.

Updating the Table View

That's it. Run the application to give it a try. Even though we don't see any problems, the table view isn't updated when a note is modified. And the same is true for newly added notes.

There are several options to solve this. We could perform a fetch request every time the notes view controller is the active view controller but that's a waste of resources. There's a much better solution that leverages the Core Data framework. Notifications.

Every managed object context broadcasts several notifications to notify interested objects about itself. These notifications are:

- `NSManagedObjectContextWillSave`
- `NSManagedObjectContextDidSave`
- `NSManagedObjectContextObjectsDidChange`

The names of these notifications are self-explanatory. The notification we're interested in is the `NSManagedObjectContextObjectsDidChange` notification. This notification is sent by the managed object context every time a managed object has changed. Let me show you how this works.

Listening for Notifications

In the `viewDidLoad()` method of the notes view controller, we invoke a helper method, `setupNotificationHandling()`.

NotesViewController.swift

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

```
3
4       title = "Notes"
5
6       setupView()
7       fetchNotes()
8       setupNotificationHandling()
9   }
```

In this helper method, we add the notes view controller as an observer of the `NSManagedObjectContextObjectsDidChange` notification.

NotesViewController.swift

```
1  private func setupNotificationHandling() {
2      let notificationCenter = NotificationCenter.default
3      notificationCenter.addObserver(self,
4                                     selector: #selector(managedObject\
5  ContextObjectsDidChange(_:)),
6                                     name: NSNotification.Name.NSManag\
7  edObjectContextObjectsDidChange,
8                                     object: coreDataManager.managedOb\
9  jectContext)
10 }
```

Notice that the last argument of the method is the managed object context of the Core Data manager. If you're developing an application that uses multiple managed object contexts, you need to make sure you only observe the managed object context the object is interested in. This is very important, not only from a performance perspective, but also in the context of threading. That'll become clear later in this book. Remember for now that you should only observe the managed object context the object is interested in.

The implementation of the `managedObjectContextObjectsDidChange(_:)` method is where the magic happens.

NotesViewController.swift

```
1  // MARK: - Notification Handling
2
3  @objc private func managedObjectContextObjectsDidChange(_ notificati\
4  on: Notification) {
5
6  }
```

The notification object has a property, `userInfo`, a dictionary. This dictionary contains the managed objects that were inserted into the managed object context, deleted from the managed object context, and it also contains he managed objects that were updated. Because we are interested in the contents of the `userInfo` dictionary, we immediately return if `userInfo` is equal to `nil`.

NotesViewController.swift

```
guard let userInfo = notification.userInfo else { return }
```

The `object` property of the notification is the managed object context that sent the notification.

We first declare a helper variable, `notesDidChange`, of type `Bool`. The value of `notesDidChange` tells us whether or not we need to update the user interface of the notes view controller.

NotesViewController.swift

```
// Helpers
var notesDidChange = false
```

Next, we extract the managed objects from the `userInfo` dictionary. The keys we're interested in are:

- `NSInsertedObjectsKey`
- `NSUpdatedObjectsKey`
- `NSDeletedObjectsKey`

NotesViewController.swift

```
if let inserts = userInfo[NSInsertedObjectsKey] as? Set<NSManagedObj\
ect> {

}

if let updates = userInfo[NSUpdatedObjectsKey] as? Set<NSManagedObje\
ct> {

}

if let deletes = userInfo[NSDeletedObjectsKey] as? Set<NSManagedObje\
ct> {

}
```

The value of each of these keys is a set of `NSManagedObject` instances. Let's start with the inserted managed objects. We loop over the inserted managed objects and, if they're `Note` instances, we add them to the array of notes. We also set `notesDidChange` to `true` because we need to update the user interface.

NotesViewController.swift

```
if let inserts = userInfo[NSInsertedObjectsKey] as? Set<NSManagedObj\
ect> {
    for insert in inserts {
```

```
4           if let note = insert as? Note {
5               notes?.append(note)
6               notesDidChange = true
7           }
8       }
9   }
```

We apply the same logic for updated managed objects. The only difference is that we don't add them to the array of notes because they're already part of the array. But we do update the `notesDidChange` variable because we need to update the user interface of the notes view controller.

NotesViewController.swift

```
1 if let updates = userInfo[NSUpdatedObjectsKey] as? Set<NSManagedObje\
2 ct> {
3     for update in updates {
4         if let _ = update as? Note {
5             notesDidChange = true
6         }
7     }
8 }
```

The logic for deleted managed objects is similar to that for inserted managed objects. Instead of inserting the managed object, we remove it from the array of notes.

NotesViewController.swift

```
1  if let deletes = userInfo[NSDeletedObjectsKey] as? Set<NSManagedObje\
2  ct> {
3      for delete in deletes {
4          if let note = delete as? Note {
5              if let index = notes?.index(of: note) {
6                  notes?.remove(at: index)
7                  notesDidChange = true
8              }
9          }
10     }
11 }
```

We obtain the index of the note in the array of notes and use that index to remove it from the array. Again, we set `notesDidChange` to `true` to notify the notes view controller that the user interface needs to be updated.

If `notesDidChange` is set to `true`, we need to perform three additional steps. First, we sort the array of notes based on the value of the `updatedAt` property. Second, we update the table view. And third, we update the view by invoking `updateView()`. This is important for insertions and deletions. We need to show the message label if the last note was deleted and we need to show the table view if the first note was inserted.

NotesViewController.swift

```
if notesDidChange {
    // Sort Notes
    notes?.sort(by: { $0.updatedAtAsDate > $1.updatedAtAsDate })

    // Update Table View
    tableView.reloadData()

    // Update View
    updateView()
}
```

This is the implementation of the `managedObjectContextObjectsDidChange(_:)` method.

NotesViewController.swift

```
// MARK: - Notification Handling

@objc private func managedObjectContextObjectsDidChange(_ notificati\
on: Notification) {
    guard let userInfo = notification.userInfo else { return }

    // Helpers
    var notesDidChange = false

    if let inserts = userInfo[NSInsertedObjectsKey] as? Set<NSManage\
dObject> {
        for insert in inserts {
            if let note = insert as? Note {
                notes?.append(note)
                notesDidChange = true
            }
        }
    }

    if let updates = userInfo[NSUpdatedObjectsKey] as? Set<NSManaged\
Object> {
        for update in updates {
            if let _ = update as? Note {
                notesDidChange = true
            }
        }
    }

    if let deletes = userInfo[NSDeletedObjectsKey] as? Set<NSManaged\
Object> {
        for delete in deletes {
            if let note = delete as? Note {
                if let index = notes?.index(of: note) {
                    notes?.remove(at: index)
                    notesDidChange = true
                }
            }
        }
    }

    if notesDidChange {
        // Sort Notes
```

```
        notes?.sort(by: { $0.updatedAtAsDate > $1.updatedAtAsDate })

        // Update Table View
        tableView.reloadData()

        // Update View
        updateView()
    }
}
```

Run the application again to see the result. Any changes made to a note are immediately visible in the table view. If we have multiple notes, the note that was last updated appears at the top. In the next chapter, we add the ability to delete notes. This is surprisingly easy.

16 To the Trash Can

The application currently supports creating, reading, and updating notes. But it should also be possible to delete notes. The pieces we need to add the ability to delete notes are already present in the notes view controller. In fact, we only need to implement one additional method of the `UITableViewDataSource` protocol to add support for deleting notes.

Deleting a Note

Open **NotesViewController.swift** and revisit the implementation of the `UITableViewDataSource` protocol. The method we need to implement is `tableView(_:commit:forRowAt:)`.

NotesViewController.swift

```
1 func tableView(_ tableView: UITableView, commit editingStyle: UITabl\
2 eViewCellEditingStyle, forRowAt indexPath: IndexPath) {
3
4 }
```

We exit early if the value of `editingStyle` isn't equal to `delete`.

NotesViewController.swift

```
1 guard editingStyle == .delete else { return }
```

We then fetch the note that corresponds with the value of the `indexPath` parameter.

NotesViewController.swift

```
1 // Fetch Note
2 guard let note = notes?[indexPath.row] else { fatalError("Unexpected\
3  Index Path") }
```

To delete the managed object, we pass the note to the `delete(_:)` method of the managed object context to which the note belongs.

NotesViewController.swift

```
1 // Delete Note
2 note.managedObjectContext?.delete(note)
```

The implementation also works if we use the managed object context of the Core Data manager because it's the same managed object context.

NotesViewController.swift

```
1 // Delete Note
2 coreDataManager.managedObjectContext.delete(note)
```

Build and Run

Run the application and delete a note to make sure everything is working as expected.

In the next few chapters, we refactor the notes view controller. Instead of manually keeping track of the notes of the persistent store, we outsource this to an instance of the `NSFetchedResultsController` class, another useful component of the **Core Data** framework.

17 Introducing the Fetched Results Controller

The application now supports creating, reading, updating, and deleting notes. And this works fine. But Core Data has another trick up its sleeve.

On iOS and macOS, data fetched from the persistent store is very often displayed in a table or collection view. Because this is such a common pattern, the Core Data framework includes a class that's specialized in managing the results of a fetch request and providing the data needed to populate a table or collection view. This class is the `NSFetchedResultsController` class.

In this chapter, we refactor the notes view controller. We use a `NSFetchedResultsController` instance to populate the table view of the notes view controller.

Creating a Fetched Results Controller

Open **NotesViewController.swift** and declare a lazy property, `fetchedResultsController`, of type `NSFetchedResultsController`. Notice that we specify the type of objects the fetched results controller will manage. Core Data and Swift work very well together. It makes working with managed objects much easier. Using Core Data with Swift 1 and 2 was much less elegant.

NotesViewController.swift

```
1 private lazy var fetchedResultsController: NSFetchedResultsControlle\
2 r<Note> = {
3
4 }()
```

To create an instance of the `NSFetchedResultsController` class, we need a fetch request. The fetch request is identical to the one we created in the `fetchNotes()` method.

NotesViewController.swift

```
1 // Create Fetch Request
2 let fetchRequest: NSFetchRequest<Note> = Note.fetchRequest()
```

We ask the `Note` class for a `NSFetchRequest<Note>` instance and we configure it by setting its `sortDescriptors` property. We sort the notes based on the value

of the `updatedAt` property like we did earlier in the `fetchNotes()` method.

NotesViewController.swift

```
// Configure Fetch Request
fetchRequest.sortDescriptors = [NSSortDescriptor(key: #keyPath(Note.\
updatedAt), ascending: false)]
```

We then initialize the `NSFetchedResultsController` instance by invoking the designated initializer, `init(fetchRequest:managedObjectContext:sectionNameKeyPath:cacheName:)`. The initializer defines four parameters:

- a fetch request
- a managed object context
- a key path for creating sections
- a cache name for optimizing performance

NotesViewController.swift

```
// Create Fetched Results Controller
let fetchedResultsController = NSFetchedResultsController(fetchReque\
st: fetchRequest,
                                                          managedObj\
ectContext: self.coreDataManager.managedObjectContext,
                                                          sectionNam\
eKeyPath: nil,
                                                          cacheName:\
 nil)
```

The managed object context we pass to the initializer is the managed object context that's used to perform the fetch request. The key path and cache name are not important for this discussion.

Before we return the fetched results controller from the closure, we set its delegate to `self`, the view controller.

NotesViewController.swift

```
// Configure Fetched Results Controller
fetchedResultsController.delegate = self
```

Because the `fetchedResultsController` property is a lazy property, it isn't a problem that we access the `managedObjectContext` property of the Core Data manager. But this means that we need to make sure we access the `fetchedResultsController` property *after* the Core Data stack is fully initialized.

This is the implementation of the `fetchedResultsController` property.

NotesViewController.swift

```
private lazy var fetchedResultsController: NSFetchedResultsControlle\
r<Note> = {
    // Create Fetch Request
    let fetchRequest: NSFetchRequest<Note> = Note.fetchRequest()

    // Configure Fetch Request
    fetchRequest.sortDescriptors = [NSSortDescriptor(key: #keyPath(N\
ote.updatedAt), ascending: false)]

    // Create Fetched Results Controller
    let fetchedResultsController = NSFetchedResultsController(fetchR\
equest: fetchRequest,
                                                              manage\
dObjectContext: self.coreDataManager.managedObjectContext,
                                                              sectio\
nNameKeyPath: nil,
                                                              cacheN\
ame: nil)

    // Configure Fetched Results Controller
    fetchedResultsController.delegate = self

    return fetchedResultsController
}()
```

Performing a Fetch Request

The fetched results controller doesn't perform a fetch if we don't tell it to. Performing a fetch is as simple as invoking `performFetch()` on the fetched results controller. We do this in the `fetchNotes()` method.

NotesViewController.swift

```
private func fetchNotes() {
    do {
        try self.fetchedResultsController.performFetch()
    } catch {
        print("Unable to Perform Fetch Request")
        print("\(error), \(error.localizedDescription)")
    }
}
```

We remove the current implementation of the `fetchNotes()` method and invoke `performFetch()` on the fetched results controller instead. Because the `performFetch()` method is throwing, we wrap it in a `do-catch` statement.

To make sure the user interface is updated after performing the fetch request in `viewDidLoad()`, we invoke `updateView()` at the end of the `viewDidLoad()` method.

NotesViewController.swift

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

    title = "Notes"

    setupView()
    fetchNotes()
    setupNotificationHandling()

    updateView()
}
```

Because the fetched results controller now manages the notes fetched from the persistent store, we can get rid of the `notes` property and ask the fetched results controller for the data we need.

This also means that the implementation of the `hasNotes` property needs some changes. We ask the fetched results controller for the value of its `fetchedObjects` property, the managed objects it fetched from the persistent store.

NotesViewController.swift

```
private var hasNotes: Bool {
    guard let fetchedObjects = fetchedResultsController.fetchedObjec\
ts else { return false }
    return fetchedObjects.count > 0
}
```

If the value of `fetchedObjects` isn't equal to `nil`, we return `true` if the number of managed objects is greater than `0`. As you can see, this isn't rocket science.

Updating the Table View

The most important change is related to the implementation of the `UITableViewDataSource` protocol. We start by optimizing the implementation of the `numberOfSections(in:)` method. The current implementation is fine, but the `NSFetchedResultsController` class offers a better solution.

A fetched results controller is perfectly capable of managing hierarchical data. That's why it's such a good fit for table and collection views. Even though we're not splitting the notes up into sections, we can still ask the fetched results controller for the sections it manages. We return the number of sections using the value of the `sections` property.

NotesViewController.swift

```
func numberOfSections(in tableView: UITableView) -> Int {
    guard let sections = fetchedResultsController.sections else { re\
turn 0 }
```

```
    return sections.count
}
```

In `tableView(_:numberOfRowsInSection:)`, we ask the fetched results controller for the section that corresponds with the value of the `section` parameter. The object that's returned to us conforms to the `NSFetchedResultsSectionInfo` protocol. The `numberOfObjects` property tells us exactly how many managed object the section contains.

NotesViewController.swift

```
func tableView(_ tableView: UITableView, numberOfRowsInSection secti\
on: Int) -> Int {
    guard let section = fetchedResultsController.sections?[section] \
else { return 0 }
    return section.numberOfObjects
}
```

The only change we need to make to the `tableView(_:cellForRowAt:)` method is how we fetch the note that corresponds with the value of the `indexPath` parameter. As I mentioned earlier, the `NSFetchedResultsController` class was designed with table views in mind (collection views were introduced several years later).

To fetch the note, we ask the fetched results controller for the managed object that corresponds with the index path. Again, the fetched results controller knows very well how to handle hierarchical data.

NotesViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    // Dequeue Reusable Cell
    guard let cell = tableView.dequeueReusableCell(withIdentifier: N\
oteTableViewCell.reuseIdentifier, for: indexPath) as? NoteTableViewC\
ell else {
        fatalError("Unexpected Index Path")
    }

    // Fetch Note
    let note = fetchedResultsController.object(at: indexPath)

    ...

    return cell
}
```

The last method we need to update is `tableView(_:commit:forRowAt:)`. We apply the same strategy to fetch the note that needs to be deleted.

NotesViewController.swift

```
func tableView(_ tableView: UITableView, commit editingStyle: UITabl\
eViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    guard editingStyle == .delete else { return }

    // Fetch Note
    let note = fetchedResultsController.object(at: indexPath)

    // Delete Note
    coreDataManager.managedObjectContext.delete(note)
}
```

A Few More Changes

Before we can run the application, we need to make three more changes.

First, we need to update the implementation of `prepare(for:sender:)`. We ask the fetched results controller for the note that corresponds with the currently selected row of the table view.

NotesViewController.swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }

    switch identifier {
    case Segue.AddNote:
        ...
    case Segue.Note:
        guard let destination = segue.destination as? NoteViewContro\
ller else {
            return
        }

        guard let indexPath = tableView.indexPathForSelectedRow else\
 {
            return
        }

        // Fetch Note
        let note = fetchedResultsController.object(at: indexPath)

        // Configure Destination
        destination.note = note
    default:
        break
    }
}
```

Second, we can get rid of `setupNotificationHandling()` and `managedObjectContextObjectsDidChange(_:)`. We no longer need these methods because the fetched results controller observes the managed object context for us. Remove these methods and any references to them.

Third, the view controller is the delegate of the fetched results controller. This means it needs to conform to the `NSFetchedResultsControllerDelegate`

protocol. To satisfy the compiler we create an empty extension for the `NotesViewController` class in which it conforms to the `NSFetchedResultsControllerDelegate` protocol. This isn't a problem because every method of the protocol is optional. The `NSFetchedResultsControllerDelegate` protocol is an Objective-C protocol, which support optional methods.

NotesViewController.swift

```
1 extension NotesViewController: NSFetchedResultsControllerDelegate {
2
3 }
```

Run the application to see the result. Everything seems to work fine … well … more or less. If we create, update, or delete a note, the table view isn't updated. For that to work, we need to implement the `NSFetchedResultsControllerDelegate` protocol. We do that in the next chapter.

18 Exploring the NSFetchedResultsControllerDelegate Protocol

To update the table view, we listened for notifications sent by the managed object context of the Core Data manager. This is a perfectly fine solution. But it can be messy to sift through the managed objects contained in the `userInfo` dictionary of the notification. In a complex Core Data application, the `NSManagedObjectContextObjectsDidChange` notification is sent very frequently. It includes every change of every managed object, even the ones we may not be interested in. We need to make sure we only respond to the changes of the managed objects we *are* interested in.

The `NSFetchedResultsController` class makes this much easier. Well … the `NSFetchedResultsControllerDelegate` protocol does. It exposes four methods that make it very easy to update a table or collection view when it's appropriate.

The most important benefit of using a fetched results controller is that the fetched results controller takes care of observing the managed object context it's tied to and it only notifies its delegate when it's appropriate to update the user interface.

This is a very powerful concept. We hand the fetched results controller a fetch request and the fetched results controller makes sure its delegate is only notified when the results of *that* fetch request change.

Implementing the Protocol

The `NSFetchedResultsControllerDelegate` protocol defines four methods. We need to implement three of those in the extension for the `NotesViewController` class. The first two methods are easy:

- `controllerWillChangeContent(_:)`
- `controllerDidChangeContent(_:)`

The `controllerWillChangeContent(_:)` method is invoked when the fetched results controller starts processing one or more inserts, updates, or deletes. The `controllerDidChangeContent(_:)` method is invoked when the fetched results controller has finished processing the changes.

The implementation of these methods is surprisingly short. In `controllerWillChangeContent(_:)` we inform the table view that updates are on their way.

NotesViewController.swift

```
func controllerWillChangeContent(_ controller: NSFetchedResultsContr\
oller<NSFetchRequestResult>) {
    tableView.beginUpdates()
}
```

And in `controllerDidChangeContent(_:)`, we notify the table view that we won't be sending it any more updates. This is important since multiple updates can occur in a very short time frame. By notifying the table view, we can batch update the table view. This is more efficient and performant.

In `controllerDidChangeContent(_:)`, we also invoke `updateView()`. This is necessary because we need to show the message label when the last note is deleted and we need to show the table view when the first note is inserted.

NotesViewController.swift

```
func controllerDidChangeContent(_ controller: NSFetchedResultsContro\
ller<NSFetchRequestResult>) {
    tableView.endUpdates()

    updateView()
}
```

More interesting is the implementation of the `controller(_:didChange:at:for:newIndexPath:)` method. That's quite a mouthful. As the name of the method implies, this method is invoked every time a managed object is modified. As you can imagine, it's possible that this method is invoked several times for a single change made by the user.

NotesViewController.swift

```
func controller(_ controller: NSFetchedResultsController<NSFetchRequ\
estResult>, didChange anObject: Any, at indexPath: IndexPath?, for t\
ype: NSFetchedResultsChangeType, newIndexPath: IndexPath?) {

}
```

The `type` parameter can have four possible values:

- `insert`
- `delete`

- `update`
- `move`

The `move` type shows once more that the `NSFetchedResultsController` class is a perfect fit for table and collection views.

NotesViewController.swift

```
func controller(_ controller: NSFetchedResultsController<NSFetchRequ\
estResult>, didChange anObject: Any, at indexPath: IndexPath?, for t\
ype: NSFetchedResultsChangeType, newIndexPath: IndexPath?) {
    switch (type) {
    case .insert:

    case .delete:

    case .update:

    case .move:

    }
}
```

Inserts

Let's start with inserts first. The method gives us the destination of the managed object that was inserted. This destination is stored in the `newIndexPath` parameter. We unwrap the value of `newIndexPath` and insert a row at the correct index path. The rest is taken care of by the implementation of the `UITableViewDataSource` protocol.

NotesViewController.swift

```
case .insert:
    if let indexPath = newIndexPath {
        tableView.insertRows(at: [indexPath], with: .fade)
    }
```

Deletes

Deletes are just as easy. The index path of the deleted managed object is stored in the `indexPath` parameter. We safely unwrap the value of `indexPath` and delete the corresponding row from the table view.

NotesViewController.swift

```
case .delete:
    if let indexPath = indexPath {
        tableView.deleteRows(at: [indexPath], with: .fade)
    }
```

Updates

For updates, we don't need to make changes to the table view itself. But we need to fetch the table view cell that corresponds with the updated managed object and update its contents.

For that, we need a helper method, `configure(_:at:)`. We can reuse some of the components of the `tableView(_:cellForRowAt:)` method of the `UITableViewDataSource` protocol. We fetch the managed object that corresponds with the value of `indexPath` and we update the table view cell.

NotesViewController.swift

```
func configure(_ cell: NoteTableViewCell, at indexPath: IndexPath) {
    // Fetch Note
    let note = fetchedResultsController.object(at: indexPath)

    // Configure Cell
    cell.titleLabel.text = note.title
    cell.contentsLabel.text = note.contents
    cell.updatedAtLabel.text = updatedAtDateFormatter.string(from: n\
ote.updatedAtAsDate)
}
```

This also means we can simplify the implementation of `tableView cellForRowAtIndexPath`.

NotesViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    // Dequeue Reusable Cell
    guard let cell = tableView.dequeueReusableCell(withIdentifier: N\
oteTableViewCell.reuseIdentifier, for: indexPath) as? NoteTableViewC\
ell else {
        fatalError("Unexpected Index Path")
    }

    // Configure Cell
    configure(cell, at: indexPath)

    return cell
}
```

We can now use this helper method to update the table view cell that corresponds with the managed object that was updated. We safely unwrap the value of `indexPath`, fetch the table view cell that corresponds with the index path, and use the `configure(_:at:)` method to update the contents of the table view cell.

NotesViewController.swift

```
case .update:
    if let indexPath = indexPath, let cell = tableView.cellForRow(at\
: indexPath) as? NoteTableViewCell {
        configure(cell, at: indexPath)
    }
```

Moves

When a managed object is modified, it can impact the sort order of the managed objects. This isn't easy to implement from scratch. Fortunately, the fetched results controller takes care of this as well through the `move` type.

The implementation is straightforward. The value of the `indexPath` parameter represents the original position of the managed object and the value of the `newIndexPath` parameter represents the new position of the managed object. This means that we need to delete the row at the old position and insert a row at the new position.

NotesViewController.swift

```
case .move:
    if let indexPath = indexPath {
        tableView.deleteRows(at: [indexPath], with: .fade)
    }

    if let newIndexPath = newIndexPath {
        tableView.insertRows(at: [newIndexPath], with: .fade)
    }
```

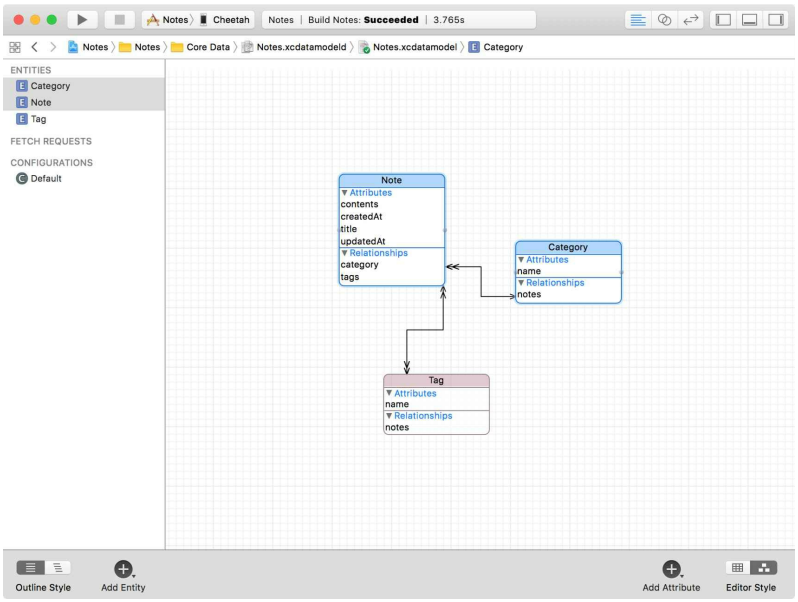
Believe it or not, this is all we need to do to respond to changes using a fetched results controller. Run the application again and make some changes to see the result.

Notice that we now also have animations. This is something we didn't have with the previous implementation because we reloaded the table view with every change.

The `NSFetchedResultsController` class is a great addition to the Core Data framework. Even though it isn't required in a Core Data application, I'm sure you agree that it can be incredibly useful. As of macOS 10.12, the `NSFetchedResultsController` class is also available on the macOS platform.

19 Adding Categories to the Mix

To help users manage their notes, it's helpful to allow them to categorize their notes. Earlier in this book, we added the **Category** entity to the data model. Remember that a note can belong to only one category, but a category can have many notes. In other words, the **Note** and **Category** entities have a **One-To-Many** relationship. The data model editor visualizes this.

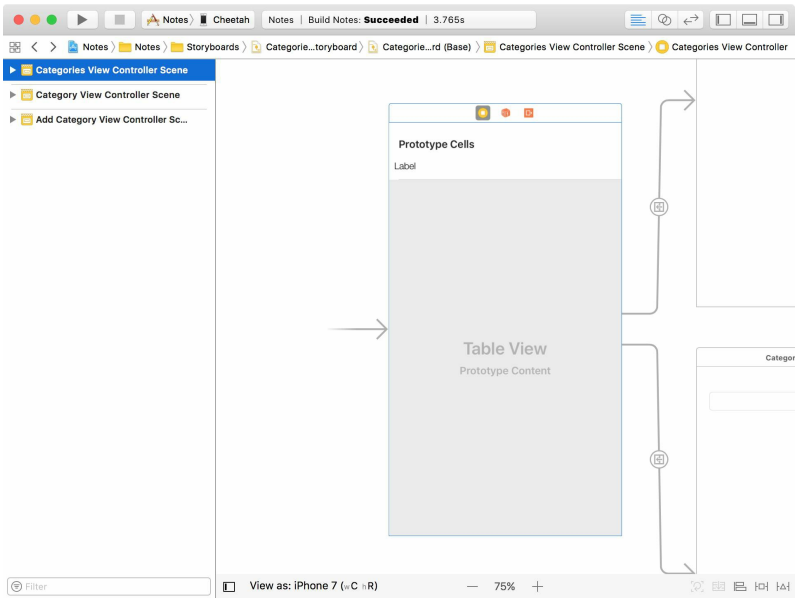


The Note and Category entities have a One-To-Many relationship.

Before We Start

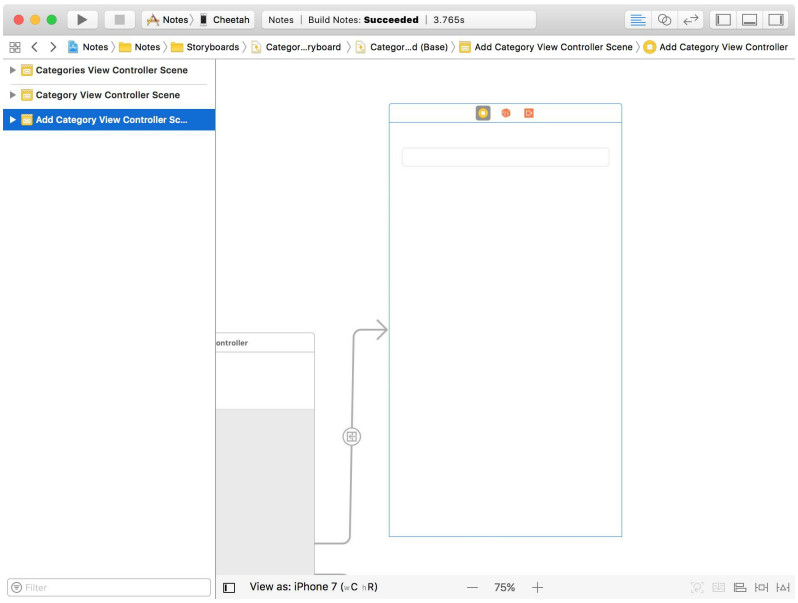
In this chapter, I show you how to assign a category to a note. I've already created view controllers for creating, updating, and deleting categories. You should already know how that works.

The `CategoriesViewController` class displays the categories in a table view. It uses a fetched results controller under the hood.



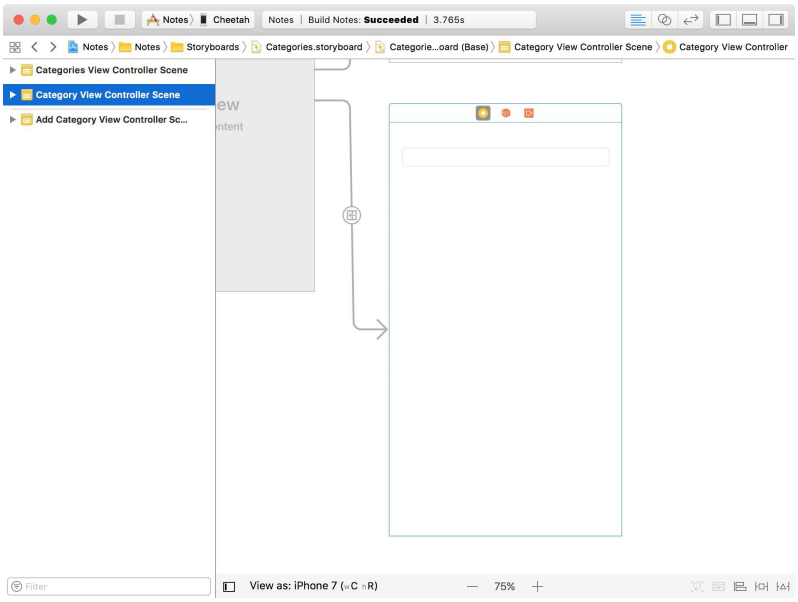
Categories View Controller

The `AddCategoryViewController` class is responsible for creating new categories.



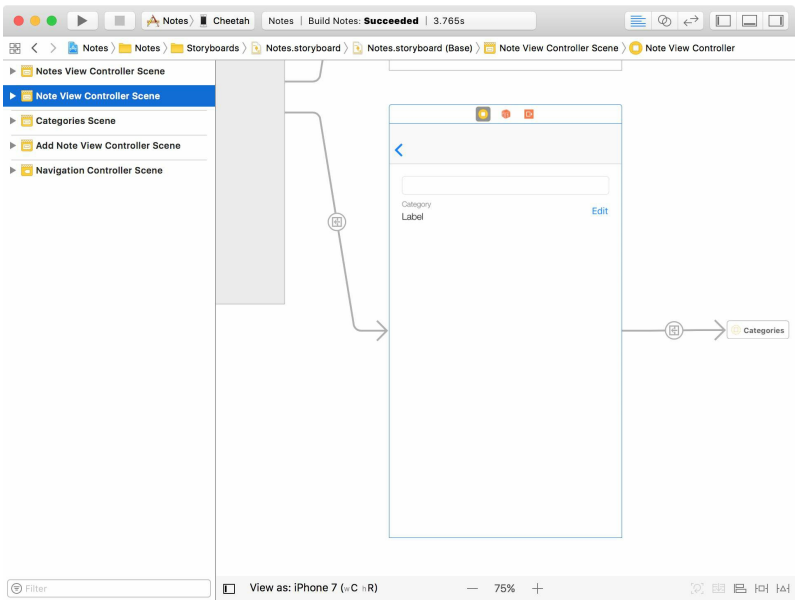
Add Category View Controller

The `CategoryViewController` class is in charge of updating existing categories.



Category View Controller

The scenes for these view controllers are located in the **Categories** storyboard. We use a storyboard reference to navigate from the **Notes** storyboard to the **Categories** storyboard. Storyboard references are a very nice addition to UIKit and make storyboards more manageable and more appealing for larger projects.



Connecting Storyboards With Storyboard References

I also updated the user interface of the note view controller. It now also displays the category to which the note belongs. If a note doesn't have a category, we show that it doesn't belong to a category yet. The **Edit** button on the right takes the user to the categories view controller.

Assigning a Category to a Note

Adding the ability to assign a category to a note is almost identical to updating an attribute of a note. Remember that attributes and relationships are both

properties. Core Data doesn't make much of a distinction from a developer's perspective.

The first step we need to take is passing the `Note` instance to the categories view controller. Open **CategoriesViewController.swift**, remove the `managedObjectContext` property, and declare a property, `note`, of type `Note?`.

CategoriesViewController.swift

```
import UIKit
import CoreData

class CategoriesViewController: UIViewController {

      ...

    // MARK: - Properties

    @IBOutlet var messageLabel: UILabel!
    @IBOutlet var tableView: UITableView!

    // MARK: -

    var note: Note?

    ...

}
```

We no longer need a property for the managed object context because we can access the managed object context through the `note` property. This means we need to make three small changes in the `CategoriesViewController` class.

In the implementation of the `fetchedResultsController` property, we access the managed object context through the `note` property.

CategoriesViewController.swift

```
fileprivate lazy var fetchedResultsController: NSFetchedResultsContr\
oller<Category> = {
    guard let managedObjectContext = self.note?.managedObjectContext\
 else {
        fatalError("No Managed Object Context Found")
    }

    ...
}()
```

And we repeat this change in the `prepare(for:sender:)` method.

CategoriesViewController.swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }
```

```
3
4       switch identifier {
5       case Segue.AddCategory:
6           ...
7
8           // Configure Destination
9           destination.managedObjectContext = note?.managedObjectContext
10      case Segue.Category:
11          ...
12      default:
13          break
14      }
15  }
```

We do the same in the `tableView(_:commit:forRowAt:)` method of the `UITableviewDelegate` protocol.

CategoriesViewController.swift

```
1 func tableView(_ tableView: UITableView, commit editingStyle: UITabl\
2 eViewCellEditingStyle, forRowAt indexPath: IndexPath) {
3       ...
4
5       // Delete Category
6       note?.managedObjectContext?.delete(category)
7   }
```

Before we move on, we need to set the `note` property of the categories view controller in the `prepare(for:sender:)` method of the `NoteViewController` class. Open **NoteViewController.swift** and navigate to the `prepare(for:sender:)` method. Instead of setting the `managedObjectContext` property, which we deleted a minute ago, we set the `note` property. That's it.

NoteViewController.swift

```
1  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
2      guard let identifier = segue.identifier else { return }
3
4      switch identifier {
5      case Segue.Categories:
6          guard let destination = segue.destination as? CategoriesView\
7  Controller else {
8              return
9          }
10
11         // Configure Destination
12         destination.note = note
13     default:
14         break
15     }
16 }
```

Head back to **CategoriesViewController.swift**. To show the user the current category of the note, we highlight it in the table view by changing the text color

of the name label of the category table view cell. In `configure(_:at:)`, we set the text color of the name label based on the value of the note's category.

CategoriesViewController.swift

```
func configure(_ cell: CategoryTableViewCell, at indexPath: IndexPath) {
    // Fetch Note
    let category = fetchedResultsController.object(at: indexPath)

    // Configure Cell
    cell.nameLabel.text = category.name

    if note?.category == category {
        cell.nameLabel.textColor = .bitterSweet
    } else {
        cell.nameLabel.textColor = .black
    }
}
```

What's most interesting to us is what happens when the user taps a category in the table view. We ask the fetched results controller for the category that corresponds with the value of `indexPath`, we set the `category` property of the note, and we pop the categories view controller from the navigation stack.

CategoriesViewController.swift

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)

    // Fetch Category
    let category = fetchedResultsController.object(at: indexPath)

    // Update Note
    note?.category = category

    // Pop View Controller From Navigation Stack
    let _ = navigationController?.popViewController(animated: true)
}
```

As I mentioned earlier, updating the category of a note is as simple as updating the title or contents of a note. But there's a bit of magic that happens behind the scenes.

Remember that the `category` relationship has an inverse relationship, `notes`. The inverse relationship belongs to the **Category** entity. By setting the `category` of the note, the inverse relationship is automatically updated by Core Data. This is something we get for free.

This also means that it doesn't matter which side of the relationship you update. The result is identical. That's important to understand and remember.

Assigning a Note to a Category

Even though the application doesn't have the ability to assign a note to a category, I'd like to show you how you can assign a note to a category. Remember that a category can have many notes. We cannot simply set the `notes` property of the category to add a note to a category. That won't work.

The `notes` property of a category is of type `NSSet?`. You may be wondering how to best add a note to that set. The set isn't mutable. Fortunately, Xcode has you covered.

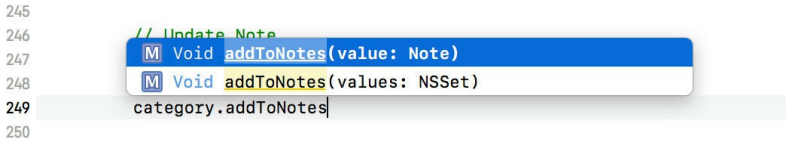
Earlier in the book, I explained that Xcode generates some code for us. It automatically generates an `NSManagedObject` subclass for every entity if you check **Class Definition** in the **Codegen** section of the **Data Model Inspector**.

But Xcode does more than generate a class definition. It also generates convenience methods for adding managed objects to a **To-Many** relationship. The format of these convenience methods is easy to remember. For the `notes` relationship, for example, it is:

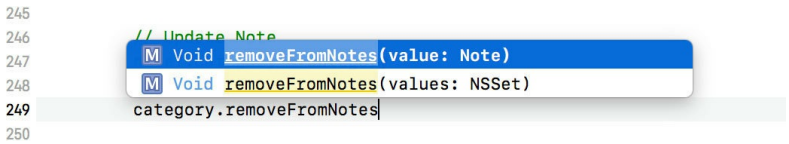
- `addToNotes(_:)` to add a note
- `removeFromNotes(_:)` to remove a note

That's easy enough to remember. Even though it may seem as if Xcode generates two of these convenience methods, it actually generates four.

Take a look at the autocompletion suggestions. We can pass in a note, but we can also pass in a set of managed objects. That's very convenient.



Xcode Generates Convenience Methods to Add Notes



And to Remove Notes

Long story short, we can add a note to a category by passing the note as an argument of the `addToNotes(_:)` method. It's that simple.

And remember that we only need to set one side of the relationship. The other side is automatically updated for us. To make it easy on ourselves, we set the `category` property of the note. Run the application to try it out.

It seems to work, but the note view controller doesn't update the value of the category label. This isn't surprising since we haven't put any code in place that updates the category label when the note of the note view controller is modified.

Updating the Note View Controller

If you've read the previous chapters, you probably know what we need to do. We need to add the note view controller as an observer of the `NSManagedObjectContextObjectsDidChange` notification.

In `viewDidLoad()`, we invoke a helper method, `setupNotificationHandling()`.

NoteViewController.swift

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

    title = "Edit Note"

    setupView()

    setupNotificationHandling()
}
```

In this method, we add the note view controller as an observer of the `NSManagedObjectContextObjectsDidChange` notification. When the note view controller receives such a notification, the `managedObjectContextObjectsDidChange(_:)` method is invoked.

NoteViewController.swift

```
// MARK: - Helper Methods

private func setupNotificationHandling() {
    let notificationCenter = NotificationCenter.default
    notificationCenter.addObserver(self,
                                   selector: #selector(managedObject\
ContextObjectsDidChange(_:)),
                                   name: Notification.Name.NSManaged\
ObjectContextObjectsDidChange,
                                   object: note?.managedObjectContex\
t)
}
```

In this method, we make sure that the `userInfo` dictionary of the notification isn't equal to `nil` and that it contains a value for the `NSUpdatedObjectsKey` key. We then use a fancy line of code to make sure the note of the note view controller is one of the managed objects that was updated. We filter the `updates` set of managed objects and, if the resulting set contains any managed objects, we invoke `updateCategoryLabel()`, another helper method.

NoteViewController.swift

```
// MARK: - Notification Handling

@objc private func managedObjectContextObjectsDidChange(_ notificati\
on: Notification) {
    guard let userInfo = notification.userInfo else { return }
    guard let updates = userInfo[NSUpdatedObjectsKey] as? Set<NSMana\
gedObject> else { return }

    if (updates.filter { return $0 == note }).count > 0 {
        updateCategoryLabel()
    }
}
```

As the name implies, in `updateCategoryLabel()` we update the category label.

NoteViewController.swift

```
private func updateCategoryLabel() {
    // Configure Category Label
    categoryLabel.text = note?.category?.name ?? "No Category"
}
```

You may be wondering why we didn't invoke `setupCategoryLabel()` instead. That's a personal choice. I prefer to keep method names descriptive and I usually only invoke methods related to setup once. Some of these methods have an `update` counterpart that's in charge of updating. This's just a personal choice.

Run the application again and modify the category of a note to make sure everything is working as expected.

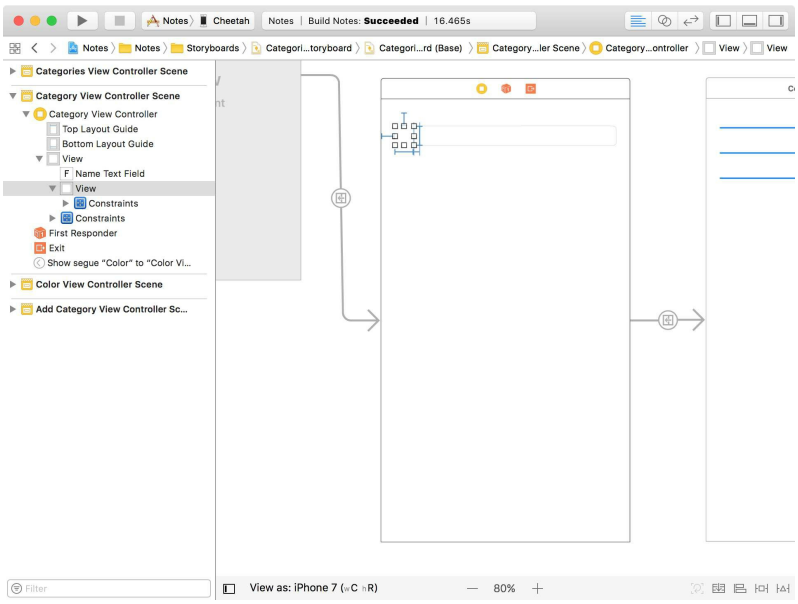
20 Adding a Dash of Color

In this chapter, I'd like to add the ability to assign a color to a category. This makes it easier to visualize which category a note belongs to.

Before We Start

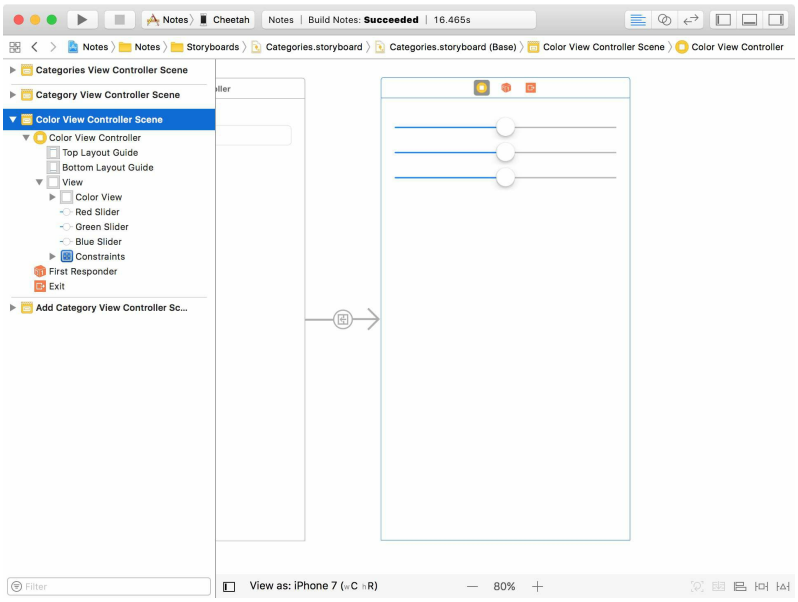
I've already laid the groundwork for this feature. Let me show you what we start with.

The category view controller contains a view for displaying the color of its category. Tapping the color view takes the user to the color view controller, the view controller responsible for picking a color.



Category View Controller

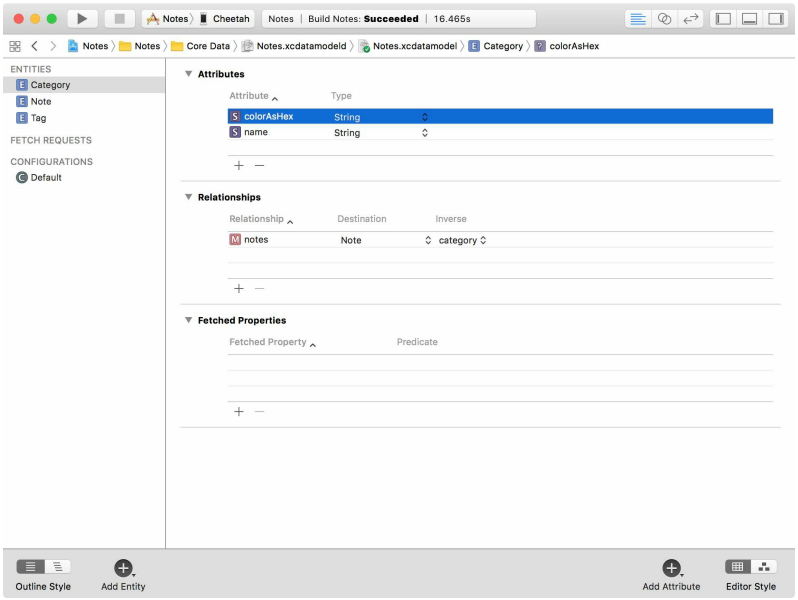
To pick a color, the user needs to adjust three sliders in the color view controller. Because the color view controller is a component that we might want to reuse, it doesn't keep a reference to a category. We use a delegate protocol to notify the category view controller which color the user has picked.



Color View Controller

Updating the Data Model

To associate a color with a category, we need to add an attribute to the **Category** entity in the data model. Open **Notes.xcdatamodeld** and add an attribute. We name the attribute **colorAsHex** because we'll be storing the color of a category as a hex value. The attribute is of type **String**.

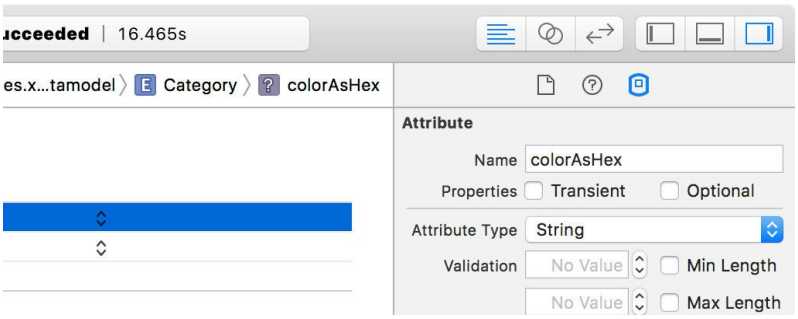


Adding an Attribute to the Category Entity

This is a personal choice. I like this approach because it's easier to read and it results in minimal overhead in terms of converting the value to a `UIColor` instance. Keep in mind that it isn't possible to store `UIColor` instances in the persistent store, we always need to convert it to another value, a **String** in this example.

We could also convert it to binary data. That's another common option. The downside is that it's harder to debug since most developers cannot read binary data … or at least I can't.

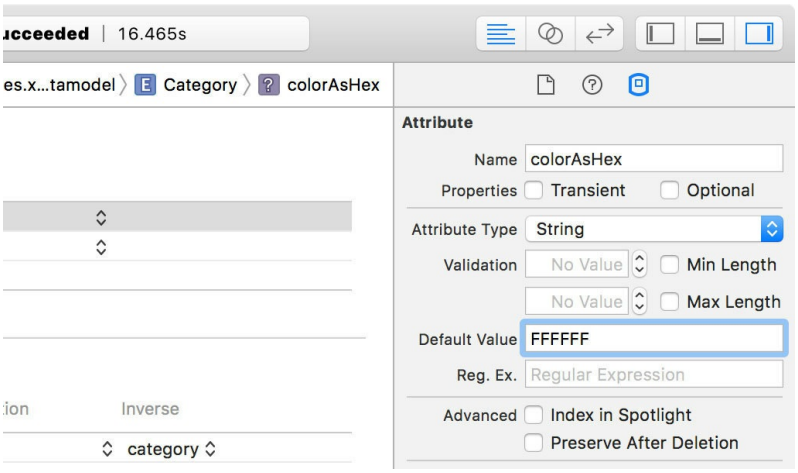
Before we move on, I want to make two changes to the attributes of the `colorAsHex` attribute. With the `colorAsHex` attribute selected, open the **Data Model Inspector** on the right and uncheck the **Optional** checkbox.



Requiring an Attribute

Remember that this makes the attribute required. It means a category cannot be stored in the persistent store if it doesn't have a value for the `colorAsHex` attribute. This is fine because we're going to make another change.

The **Data Model Inspector** also lets us define a default value for an attribute. What does that mean? If we don't explicitly assign a value to the `colorAsHex` attribute, Core Data sets the value of the attribute to the value of the **Default Value** field. Because we're working with hex values, that's very easy to do. In the **Default Value** field, enter the hex value for white, **FFFFFF**. That's another benefit of working with hex values.



Setting a Default Value

Extending UIColor

To make the conversion from and to hex values easier, I created an extension for `UIColor`. You can find the extension in the **Extensions** group in **UIColor.swift**.

UIColor.swift

```
1 import UIKit
2
3 extension UIColor {
```

```
4
5       static let bitterSweet = UIColor(red:0.99, green:0.47, blue:0.44\
6   , alpha:1.0)
7
8   }
9
10  extension UIColor {
11
12      // MARK: - Initialization
13
14      convenience init?(hex: String) {
15          var hexNormalized = hex.trimmingCharacters(in: .whitespacesA\
16  ndNewlines)
17          hexNormalized = hexNormalized.replacingOccurrences(of: "#", \
18  with: "")
19
20          // Helpers
21          var rgb: UInt32 = 0
22          var r: CGFloat = 0.0
23          var g: CGFloat = 0.0
24          var b: CGFloat = 0.0
25          var a: CGFloat = 1.0
26          let length = hexNormalized.characters.count
27
28          // Create Scanner
29          Scanner(string: hexNormalized).scanHexInt32(&rgb)
30
31          if length == 6 {
32              r = CGFloat((rgb & 0xFF0000) >> 16) / 255.0
33              g = CGFloat((rgb & 0x00FF00) >> 8) / 255.0
34              b = CGFloat(rgb & 0x0000FF) / 255.0
35
36          } else if length == 8 {
37              r = CGFloat((rgb & 0xFF000000) >> 24) / 255.0
38              g = CGFloat((rgb & 0x00FF0000) >> 16) / 255.0
39              b = CGFloat((rgb & 0x0000FF00) >> 8) / 255.0
40              a = CGFloat(rgb & 0x000000FF) / 255.0
41
42          } else {
43              return nil
44          }
45
46          self.init(red: r, green: g, blue: b, alpha: a)
47      }
48
49      // MARK: - Convenience Methods
50
51      var toHex: String? {
52          // Extract Components
53          guard let components = cgColor.components, components.count \
54  >= 3 else {
55              return nil
56          }
57
58          // Helpers
59          let r = Float(components[0])
60          let g = Float(components[1])
61          let b = Float(components[2])
62          var a = Float(1.0)
63
64          if components.count >= 4 {
65              a = Float(components[3])
66          }
67
```

```
        // Create Hex String
        let hex = String(format: "%02lX%02lX%02lX%02lX", lroundf(r *\
 255), lroundf(g * 255), lroundf(b * 255), lroundf(a * 255))

        return hex
    }

}
```

Extending Category

We can go one step further and create an extension for the `Category` class to abstract any interactions with hex values from the `Category` class. The goal is that we only deal with `UIColor` instances.

Create a new Swift file in the **Core Data > Extensions** group and name it **Category**. Add an import statement for **UIKit** and create an extension for the `Category` class.

Category.swift

```
import UIKit

extension Category {

}
```

We declare a computed property, `color`, of type `UIColor?`. We define a custom getter and setter for the computed property.

Category.swift

```
import UIKit

extension Category {

    var color: UIColor? {
        get {

        }

        set(newColor) {

        }
    }

}
```

In the getter, we convert the value of the `colorAsHex` property to a `UIColor` instance.

Category.swift

```
get {
    guard let hex = colorAsHex else { return nil }
    return UIColor(hex: hex)
}
```

In the setter, we convert the `UIColor` instance to a hex value and update the value of the `colorAsHex` property. This small improvement will keep the code we write clean and focused.

Category.swift

```
set(newColor) {
    if let newColor = newColor {
        colorAsHex = newColor.toHex
    }
}
```

Updating the Category View Controller

Next, we need to update the `CategoryViewController` class. We need to pass the current color of the category to the color view controller. We do this in the `prepare(for:sender:)` method.

CategoryViewController.swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }

    switch identifier {
    case Segue.Color:
        ...

        // Configure Destination
        destination.delegate = self
        destination.color = category?.color ?? .white
    default:
        break
    }
}
```

We also need to update the delegate method of the `ColorViewControllerDelegate` protocol. In this method, we set the `color` computed property of the category.

CategoryViewController.swift

```
func controller(_ controller: ColorViewController, didPick color: UI\
Color) {
    // Update Category
    category?.color = color

    // Update View
```

```
7       updateColorView()
8   }
```

Last but not least, we need to update the `updateColorView()` method of the category view controller. In this method, we set the background color of the color view with the color of the category.

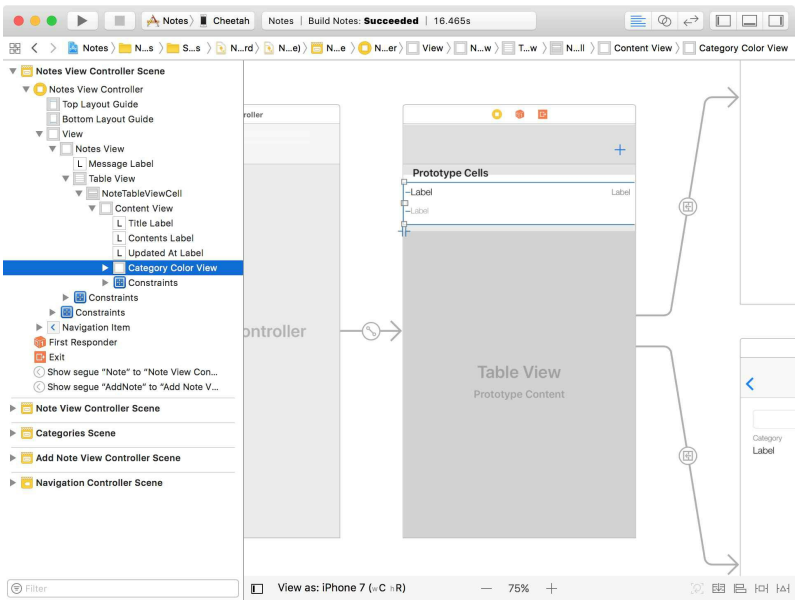
CategoryViewController.swift

```
1   private func updateColorView() {
2       // Configure Color View
3       colorView.backgroundColor = category?.color
4   }
```

I'm sure you agree that the `color` computed property keeps the implementation focused by removing any logic related to value transformations.

Updating the Notes View Controller

Before we run the application, we need to update the user interface of the notes view controller. I already updated the `NoteTableViewCell` class. On the left, it contains a narrow subview that displays the category color. This is a subtle hint for the user, showing them which category the note belongs to.



Displaying the Category of a Note

We update the background color of this subview in the `configure(_:at:)` method, the helper method we created earlier in this book. We safely unwrap the value of the `color` property and update the `backgroundColor` property of the category color view. Even though we default to white if the category doesn't have a color, this shouldn't happen in production.

NotesViewController.swift

```
func configure(_ cell: NoteTableViewCell, at indexPath: IndexPath) {
    ...

    if let color = note.category?.color {
        cell.categoryColorView.backgroundColor = color
    } else {
        cell.categoryColorView.backgroundColor = .white
    }
}
```

A Crash

Run the application to see the new feature in action. Wait. That doesn't look good. The application crashed. The persistent store coordinator wasn't able to add the persistent store.

If we take a closer look at the error message in the console, we see that the persistent store isn't compatible with the data model.

Console

```
CoreData: error: -addPersistentStoreWithType:SQLite configuration:(n\
ull) URL:file:///var/mobile/Containers/Data/Application/40950C64-3D8\
E-45AF-9890-CCAF59444996/Documents/Notes.sqlite options:(null) ... r\
eturned error Error Domain=NSCocoaErrorDomain Code=134100 "The manag\
ed object model version used to open the persistent store is incompa\
tible with the one that was used to create the persistent store." Us\
erInfo={metadata={
    NSPersistenceFrameworkVersion = 832;
    NSStoreModelVersionHashes =     {
        Category = <fa37182c b55c9960 577e91ae b9fc0c14 092dcec2 564\
459a1 19bb513f 45641c4a>;
        Note = <b76dea89 b30116c0 c283030a e66c8678 e411956b fee2991\
0 fbbe70be 034a4d56>;
        Tag = <b740a6fb 4c426dd1 4ea60b33 5a71b968 da756f6f e3482227\
 2cb4a849 ebf7dc73>;
    };
    NSStoreModelVersionHashesVersion = 3;
    NSStoreModelVersionIdentifiers =     (
        ""
    );
    NSStoreType = SQLite;
    NSStoreUUID = "A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3";
    "_NSAutoVacuumLevel" = 2;
}, reason=The model used to open the store is incompatible with the \
one used to create the store} with userInfo dictionary {
    metadata =     {
        NSPersistenceFrameworkVersion = 832;
        NSStoreModelVersionHashes =         {
            Category = <fa37182c b55c9960 577e91ae b9fc0c14 092dcec2\
 564459a1 19bb513f 45641c4a>;
            Note = <b76dea89 b30116c0 c283030a e66c8678 e411956b fee\
29910 fbbe70be 034a4d56>;
            Tag = <b740a6fb 4c426dd1 4ea60b33 5a71b968 da756f6f e348\
2227 2cb4a849 ebf7dc73>;
        };
        NSStoreModelVersionHashesVersion = 3;
        NSStoreModelVersionIdentifiers =         (
```

```
38              ""
39          );
40          NSStoreType = SQLite;
41          NSStoreUUID = "A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3";
42          "_NSAutoVacuumLevel" = 2;
43      };
44      reason = "The model used to open the store is incompatible with \
45  the one used to create the store";
46  }
```

> The managed object model version used to open the persistent store is incompatible with the one that was used to create the persistent store.

Earlier in this book, we ran into the same issue and I told you we would tackle this problem later in the book. Well … it's time to talk about migrations.

21 Data Model Migrations

An application that grows and gains features also gains new requirements. The data model, for example, grows and changes. Core Data handles changes pretty well as long as you play by the rules of the framework.

In this chapter, we take a close look at the cause of the crash we ran into in the previous chapter. We learn how Core Data helps us manage changes of the data model and what pitfalls we absolutely need to avoid.

Finding the Root Cause

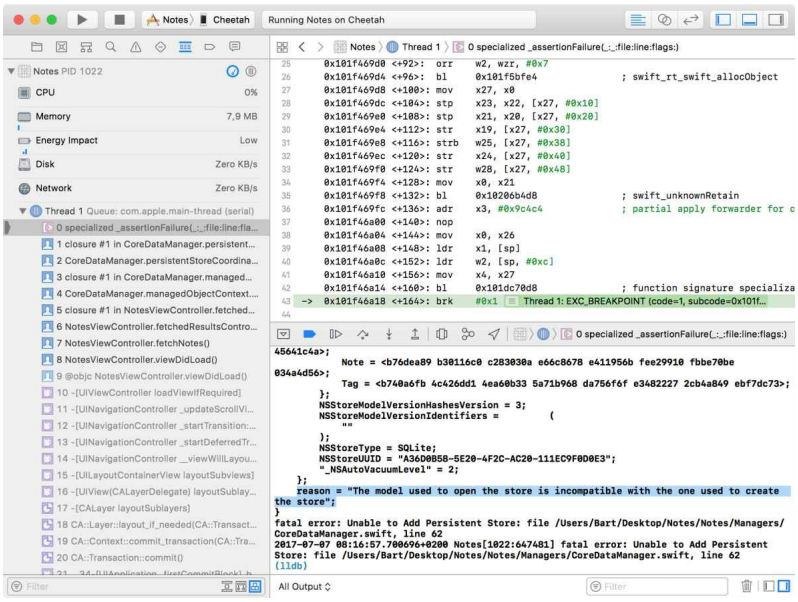
Finding the root cause of the crash is easy. Open **CoreDataManager.swift** and inspect the implementation of the `persistentStoreCoordinator` property. If adding the persistent store to the persistent store coordinator fails, the application throws a fatal error, immediately terminating the application. As I mentioned earlier in this book, you shouldn't throw an error if adding the persistent store fails.

CoreDataManager.swift

```
private lazy var persistentStoreCoordinator: NSPersistentStoreCoordi\
nator = {
    ...

    do {
        // Add Persistent Store
        try persistentStoreCoordinator.addPersistentStore(ofType: NS\
SQLiteStoreType, configurationName: nil, at: persistentStoreURL, opt\
ions: nil)

    } catch {
        fatalError("Unable to Add Persistent Store")
    }

    return persistentStoreCoordinator
}()
```

In this chapter, I want to show you what we need to do to prevent that adding the persistent store to the persistent store coordinator fails. Run the application again and inspect the output in the console. This line tells us what went wrong.

```
reason = "The model used to open the store is incompatible with the \
one used to create the store";
```

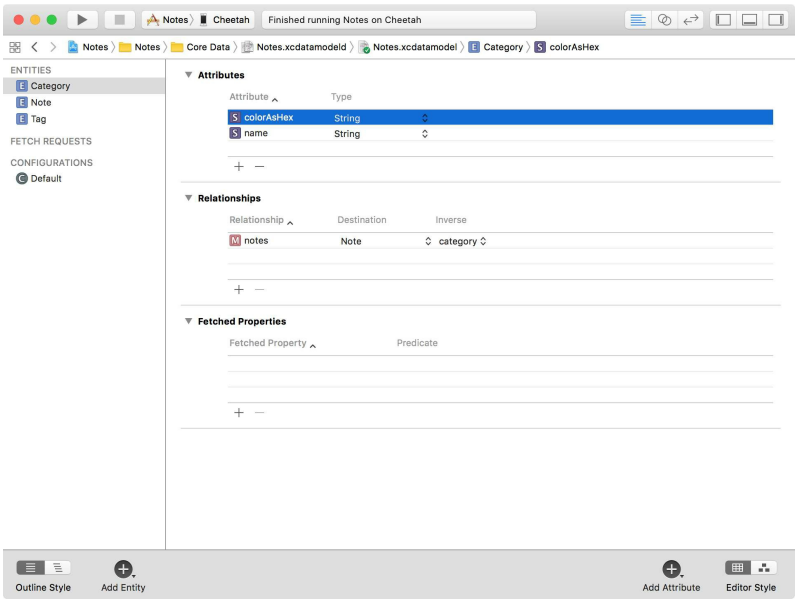


Finding the Root Cause

We're getting closer to the root of the problem. Core Data tells us that the data model isn't compatible with the data model we used to create the persistent store. What does that mean?

Remember that Core Data automatically creates a persistent store if it isn't able to find one to add. To create the persistent store, the framework first inspects the data model. For a SQLite database, for example, Core Data needs to know what the database schema should look like. It fetches this information from the data model.

In the previous chapter, we modified the data model by adding the `colorAsHex` attribute to the `Category` entity. With the new data model in place, we ran the application again … and you know what happened next.



Modifying the Data Model

Before the persistent store coordinator adds a persistent store, it checks if a persistent store already exists. If it finds one, Core Data makes sure the data

model is **compatible** with the persistent store. How this works becomes clear in a moment.

The error message in the console indicates that the data model that was used to create the persistent store isn't identical to the current data model. As a result, Core Data bails out and throws an error.

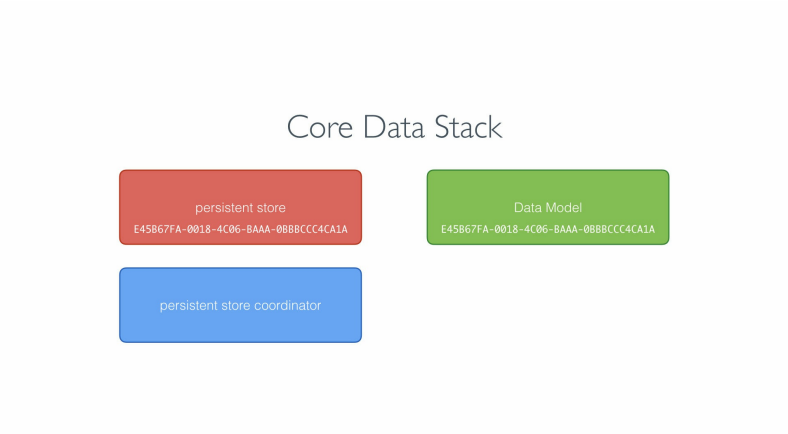
Versioning the Data Model

You should never modify a data model without telling Core Data about the changes you made. Let me repeat that. You should never modify a data model without telling Core Data about the changes you made.

But how *do* you tell Core Data about the changes you made to the data model? The answer is **versioning**.

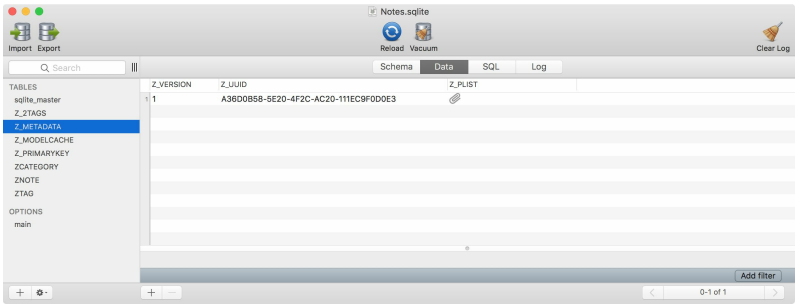
The idea is simple. Core Data tells us that the current data model is not the one that was used to create the persistent store. To solve that problem, we first and foremost leave the data model that was used to create the persistent store untouched. That's one problem solved.

To make changes to the data model, we make a new version of the data model. Each data model version has a unique identifier and Core Data stores this identifier in the persistent store to know what model was used to create the persistent store.



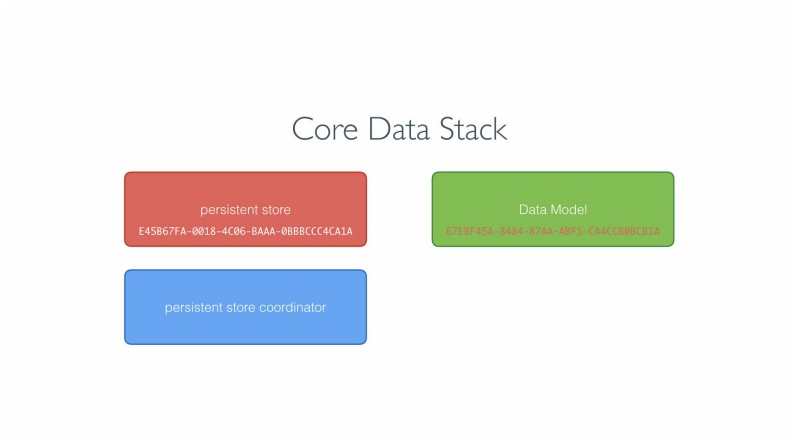
Core Data stores the unique identifier of the data model in the persistent store.

We can verify this by inspecting the persistent store of the application. If we inspect the SQLite database, we see a table named **Z_METADATA**. This table contains the unique identifier of the data model. The unique identifier changes when the data model changes.



Inspecting the Persistent Store

Before the persistent store coordinator adds a persistent store, it compares the unique identifier stored in the persistent store with that of the current data model.



Comparing the Unique Identifier of the Current Data Model

Now that we know what went wrong, we can implement a solution. Fortunately, Core Data makes versioning the data model very easy.

Before You Go

I already mentioned several times that you shouldn't throw a fatal error if adding the persistent store fails in production. It's fine to throw a fatal error if you're developing your application. Once your application is in the hands of users, though, you need to handle the situation more gracefully.

Throwing a fatal error immediately terminates your application, resulting in a bad user experience. Most users don't understand what went wrong and, all too often, they delete the application to resolve the issue. This is ironic since that's the only solution that works if your application doesn't have a solution in place to recover from this scenario.

How you handle failing to add a persistent store depends on your application. To recover from this scenario, you could delete the persistent store and try adding the persistent store again. If that operation fails as well, then you have bigger problems to worry about. Always remember that deleting the persistent store is synonymous to **data loss**. Try to avoid this at all cost.

The first action I usually take is moving the existing persistent store (the one that cannot be added to the persistent store coordinator) to a new location to prevent data loss. This doesn't resolve the issue, but it prevents immediate data loss. You can then safely add a new persistent store to the persistent store coordinator without losing the user's data. This means the user can continue using the application without running into a crash. It can also help debugging the issue if you add a mechanism that enables the user to send you the corrupt persistent store.

The second action is notifying the user about the problem. If the user opens your application and sees it's empty, they think they lost their data. It's important that you inform them about the situation and how to handle it. Explain the problem in easy to understand words. Make sure they don't panic and ask them to get in touch with you to debug the issue.

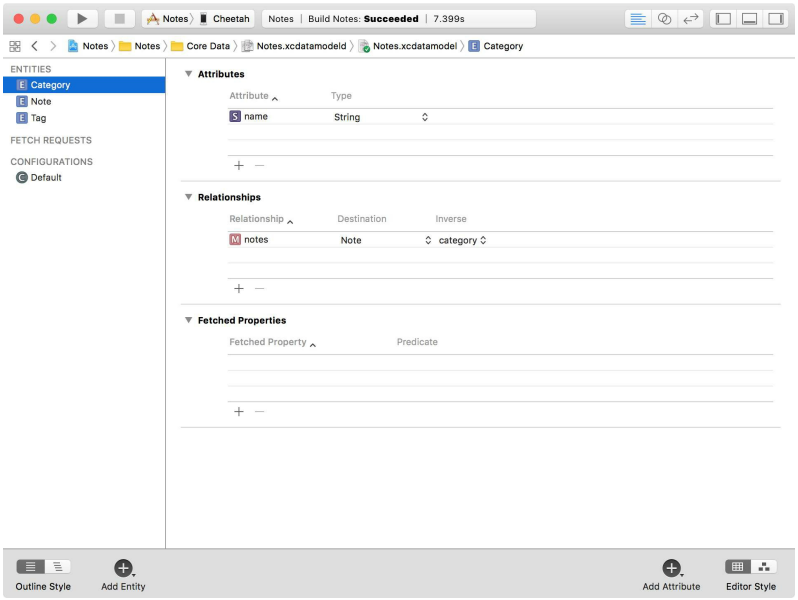
I want to emphasize that there's no *one* solution to this problem. The point I want to drive home is that you need to prepare for this scenario. If this happens, it doesn't necessarily mean you made a mistake. But it *does* mean that it's up to you to solve the problem.

22 Versioning the Data Model

In the previous chapter, we exposed the root cause of the crash we ran into earlier. The solution is versioning the data model.

Restoring the Data Model

Before we create a new version, we need to restore the current data model to its original state. Select **Notes.xcdatamodeld** and remove the **colorAsHex** attribute from the **Category** entity.



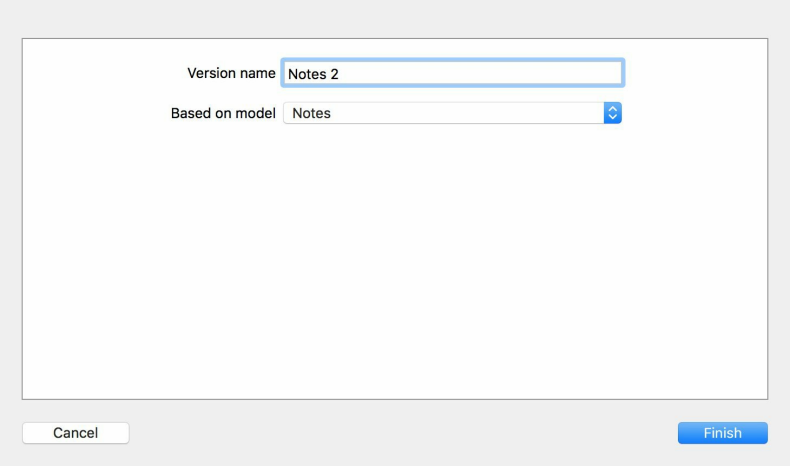
Restoring the Data Model to Its Original State

Because we reverted the data model to its original state, the application should no longer crash.

Don't worry about any errors that pop up. Because we removed the `colorAsHex` attribute, the compiler complains that the `Category` class doesn't have a property named `colorAsHex`. We'll fix that in a minute.

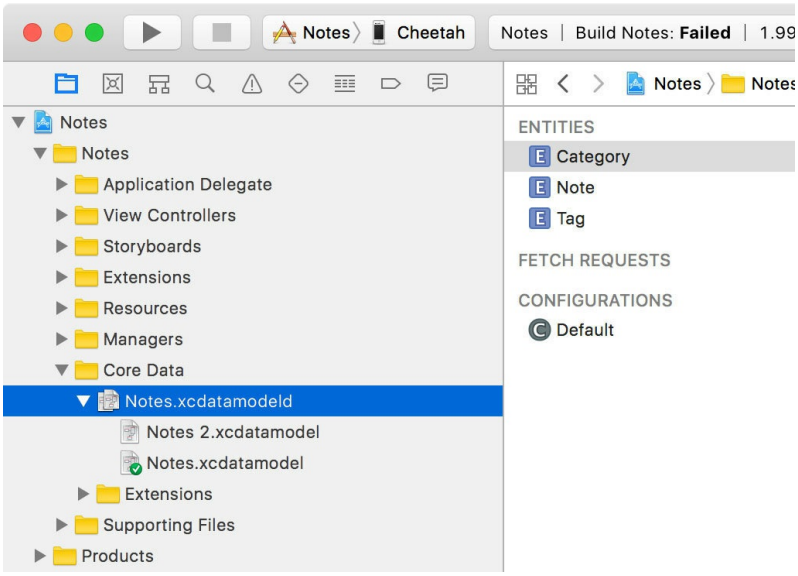
Adding a Data Model Version

It's time to create a new data model version. With the data model selected, choose **Add Model Version…** from Xcode's **Editor** menu. Name the version **Notes 2** and base the data model version on **Notes**. It's the only option available. You should always base a new version of the data model on the previous data model.



Adding a Data Model Version

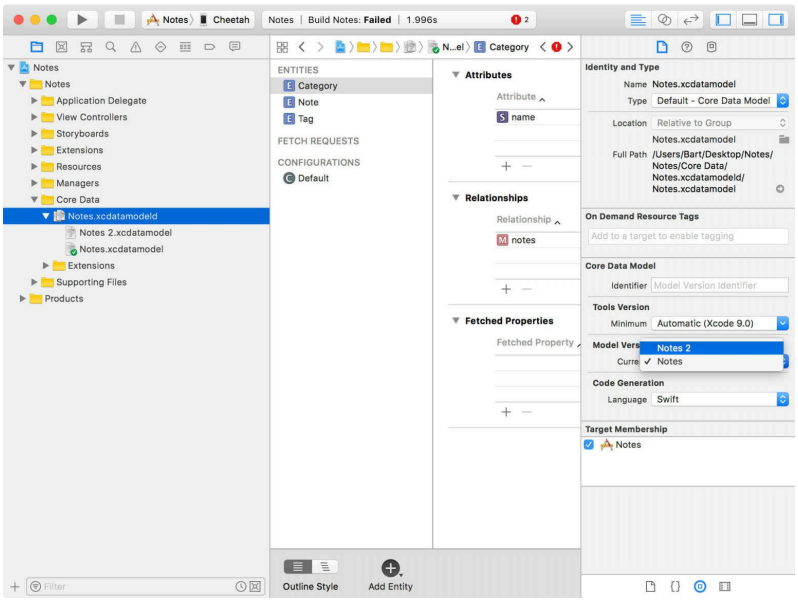
Notice that a small triangle has appeared on the left of the data model in the **Project Navigator**. You can click the triangle to show the list of data model versions.



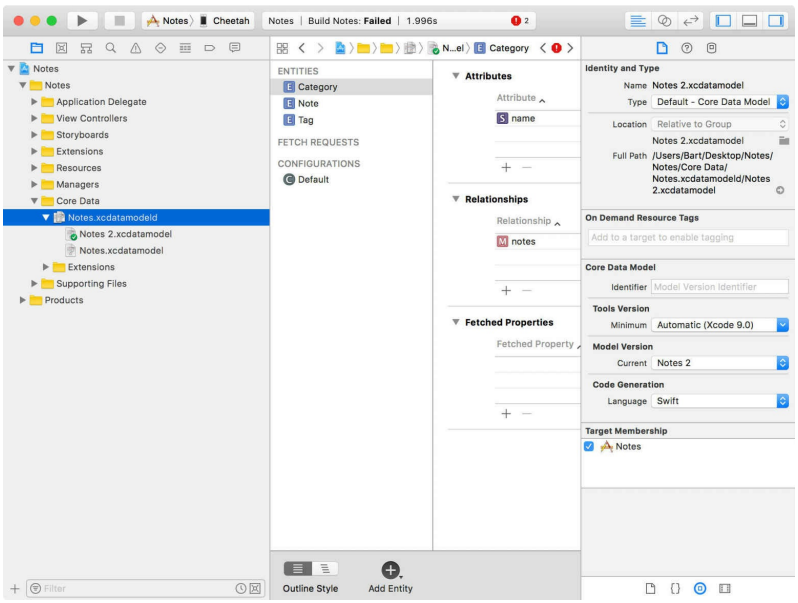
A List of Data Model Versions

You may have noticed that a green checkmark is added to **Notes.xcdatamodel**. This indicates that **Notes.xcdatamodel** is the active data model version. If we were to run the application, Core Data would continue to use the original data model version.

But that's not what we have in mind. Before we make any changes, select **Notes.xcdatamodeld**, not **Notes.xcdatamodel**. Open the **File Inspector** on the right and set **Model Version** to **Notes 2**, the data model version we just added.

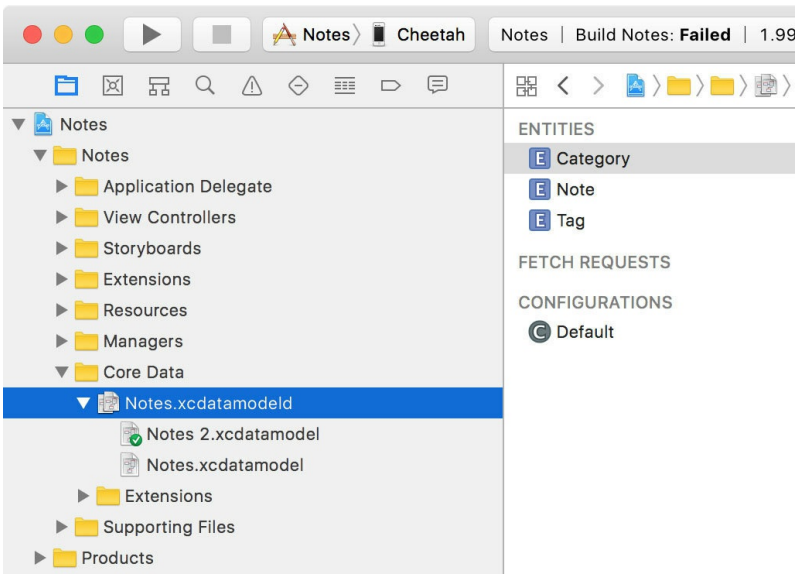


Changing the Active Data Model Version

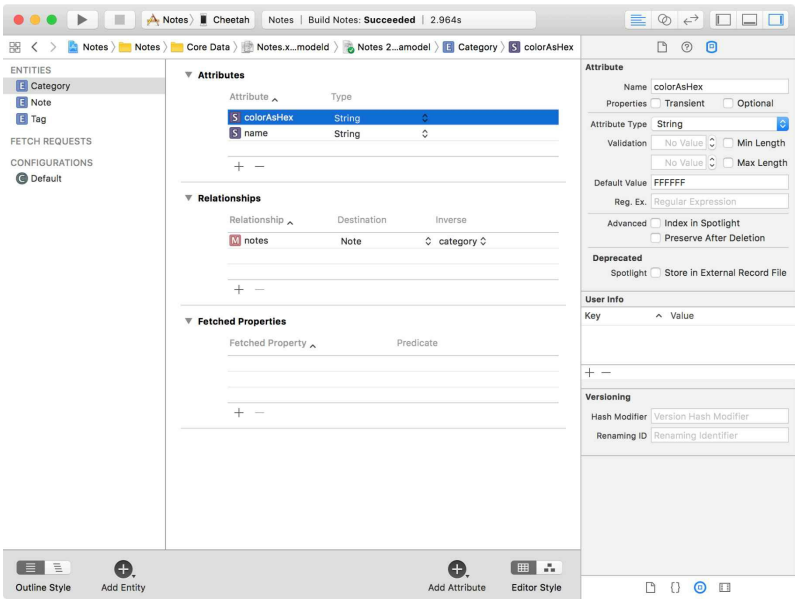


Changing the Active Data Model Version

Notice that the green checkmark has moved from **Notes.xcdatamodel** to **Notes 2.xcdatamodel**.



Because we haven't run the application yet, we can still modify the new data model version without running into compatibility issues. Select **Notes 2.xcdatamodel** and add the **colorAsHex** attribute to the **Category** entity. Don't forget to uncheck the **Optional** checkbox and set **Default Value** to white.



Adding an Attribute to the New Data Model Version

Run the application to see if we solved the incompatibility problem we ran into earlier. Are you still running into a crash? To make changes to the data model, we've added a new data model version. We also marked the new data model version as the active data model version.

What we haven't told Core Data is what it should do if it runs into an incompatibility issue. We need to tell it to perform a migration.

Performing Migrations

I already told you that a persistent store is tied to a particular version of the data model. It keeps a reference to the unique identifier of the data model. If the data model changes, we need to tell Core Data how to migrate the data of the persistent store to the new data model version.

There are two types of migrations:

- lightweight migrations
- heavyweight migrations

Heavyweight migrations are complex and you should try to avoid them whenever possible. Heavyweight migrations migrations are an advanced topic

and they're not covered in this book. Lightweight migrations are much easier because Core Data takes care of the heavy lifting.

To add support for lightweight migrations to the `CoreDataManager` class, we need to make a minor change. Remember that `addPersistentStore(ofType:configurationName:at:options:)` accepts a dictionary of options as its last parameter. To add support for migrations, we pass in a dictionary of options with two keys:

- `NSMigratePersistentStoresAutomaticallyOption`
- `NSInferMappingModelAutomaticallyOption`

CoreDataManager.swift

```
let options = [
    NSMigratePersistentStoresAutomaticallyOption : true,
    NSInferMappingModelAutomaticallyOption : true
]
```

By setting the value of `NSMigratePersistentStoresAutomaticallyOption` to `true`, we instruct Core Data to automatically perform a migration if it detects an incompatibility. That's a good start.

If the value of `NSInferMappingModelAutomaticallyOption` is set to `true`, Core Data attempts to infer the mapping model for the migration based on the data model versions of the data model.

What is a mapping model? A mapping model defines how one version of the data model relates to another version. For lightweight migrations, Core Data can infer the mapping model by inspecting the data model versions. This isn't true for heavyweight migrations and that's what makes heavyweight migrations complex and tedious. For heavyweight migrations, the developer is responsible for creating the mapping model.

With this in mind, we can update the implementation of the `do` clause of the `do-catch` statement in the `CoreDataManager` class. This is what the updated implementation looks like.

CoreDataManager.swift

```
private lazy var persistentStoreCoordinator: NSPersistentStoreCoordi\
nator = {
    ...

    do {
        let options = [
            NSMigratePersistentStoresAutomaticallyOption : true,
            NSInferMappingModelAutomaticallyOption : true
        ]
```

```
10
11          // Add Persistent Store
12          try persistentStoreCoordinator.addPersistentStore(ofType: NS\
13 SQLiteStoreType,
14                                                            configurat\
15 ionName: nil,
16                                                            at: persis\
17 tentStoreURL,
18                                                            options: o\
19 ptions)
20
21      } catch {
22          fatalError("Unable to Add Persistent Store")
23      }
24
25      return persistentStoreCoordinator
26 }()
```

Run the application to see if the solution works. If you don't run into a crash, then Core Data successfully migrated the persistent store to the new data model version.

Keep It Lightweight

Whenever you make a change to a data model, you need to consider the consequences. Lightweight migrations carry little overhead. Heavyweight migrations, however, are a pain. Really. Avoid them if possible.

How do you know if a data model change requires a lightweight or a heavyweight migration? You always need to test the migration to be sure. That said, Core Data is pretty clever and is capable of migrating the persistent store most of the times without your help.

Adding or removing entities, attributes, and relationships are no problem for Core Data. Modifying the names of entities, attributes, and relationship, however, is less trivial for Core Data. If you change the cardinality of a relationship, then you signed up for a wild ride.

Plan, Plan, Plan

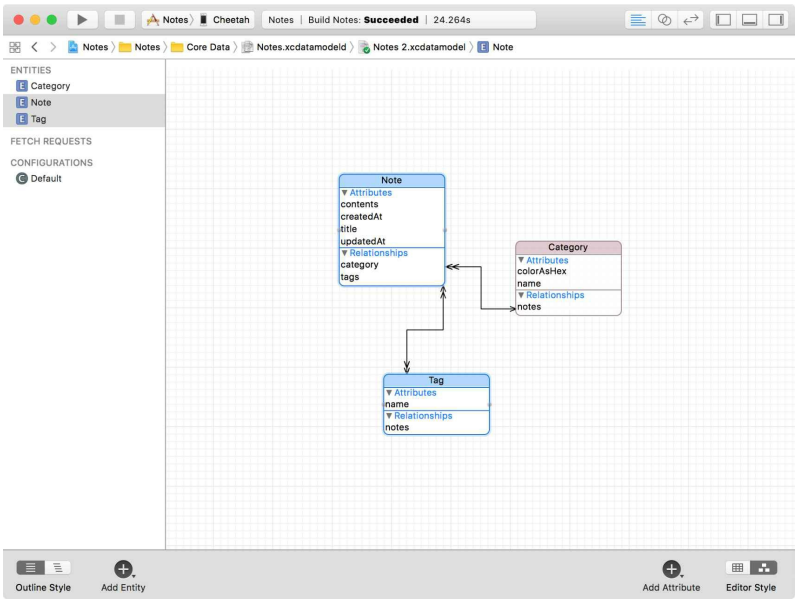
Every form of persistence requires planning. I can't stress enough how important this phase of a project is. If you don't invest time architecting the data model, chances are you run into problems that could have been avoided.

It's fine to make incremental changes to the data model as your application grows, but once your application is in the hands of users you need to make sure they don't lose their data due to a problematic migration. And always test migrations before shipping a new version of your application.

Migrations are an important aspect of Core Data because most applications grow and need to make changes to the data model at some point. Data model changes and migrations aren’t hard, but they require attention and testing.

23 Assigning Tags to a Note

The last feature I want to add is the ability to tag notes. This feature is interesting because of the **Many-To-Many** relationship of the **Note** and **Tag** entities. In this chapter, you learn how to work with such a relationship.

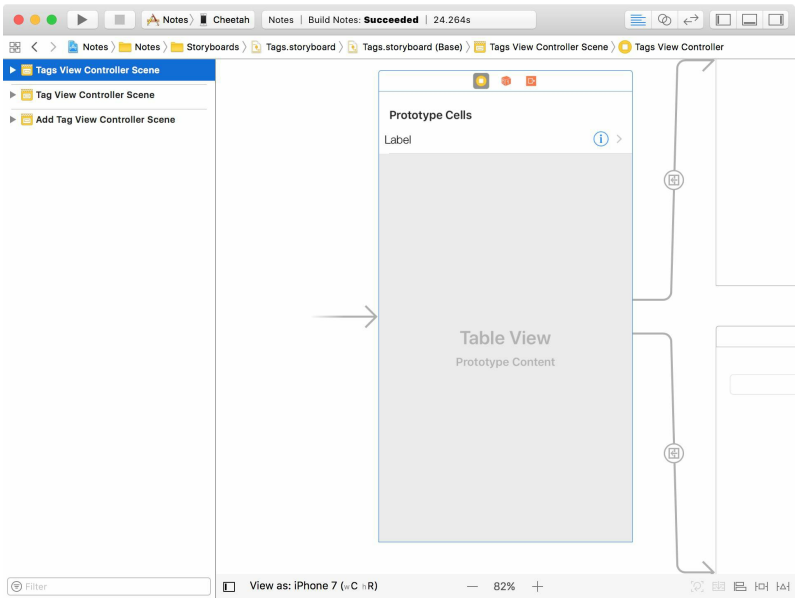


The Note and Tag entities have a Many-To-Many relationship.

Before We Start

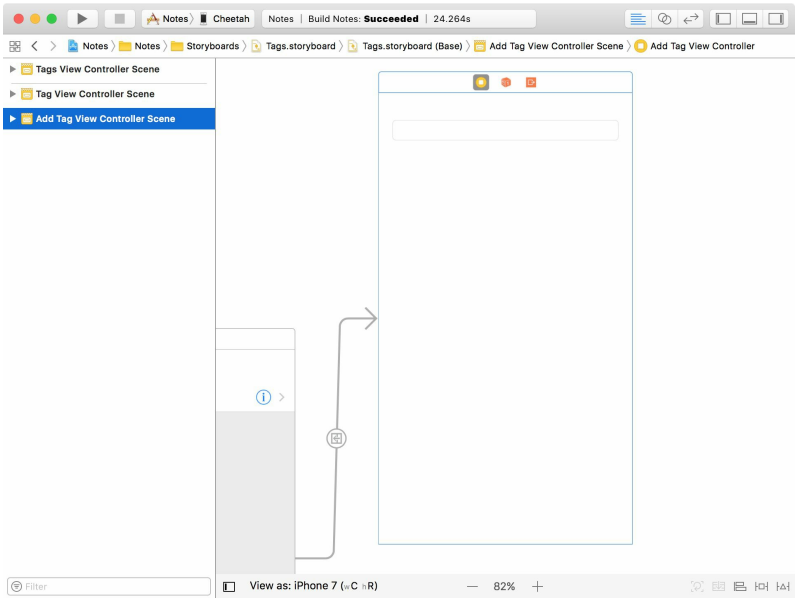
I already created view controllers for creating, updating, and deleting tags. This is very similar to managing notes and categories. This should be familiar by now.

The `TagsViewController` class displays the tags in a table view. It uses a fetched results controller under the hood.



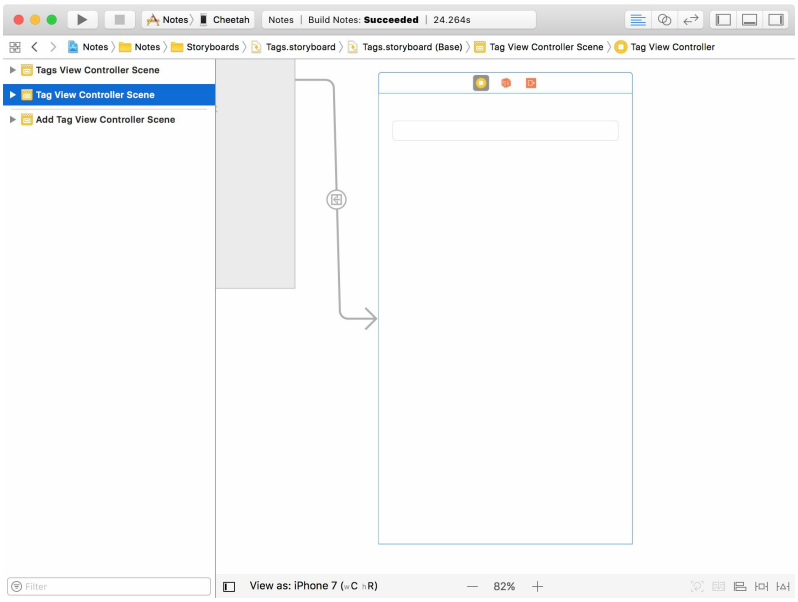
Tags View Controller

The `AddTagViewController` class is responsible for creating new tags.



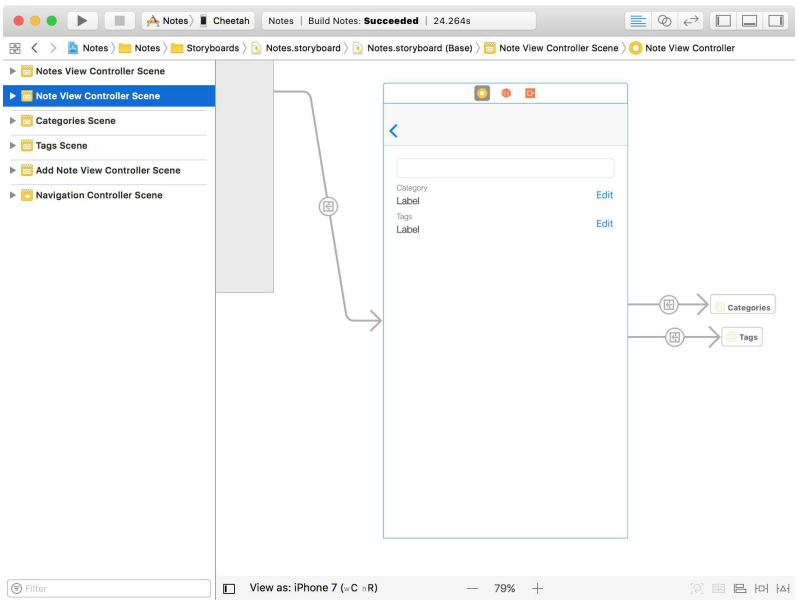
Add Tag View Controller

The `TagViewController` class is in charge of updating existing tags.



Tag View Controller

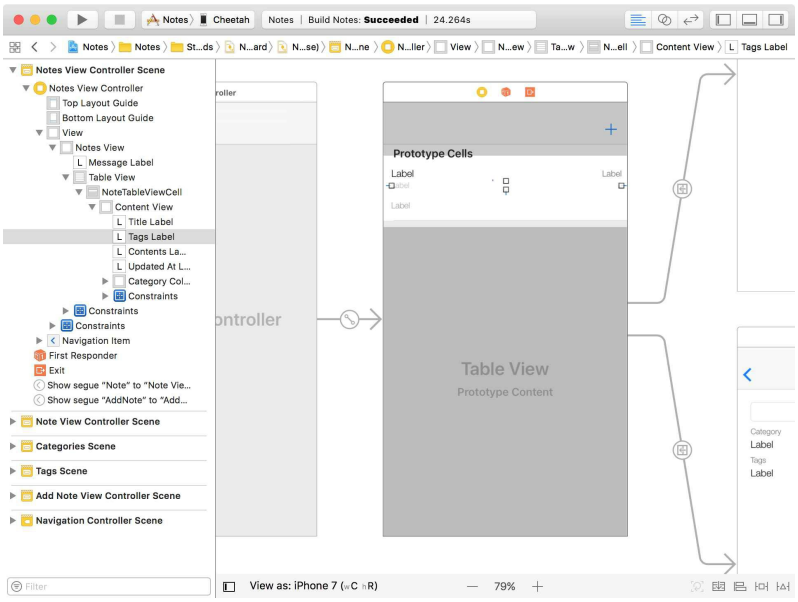
The scenes for these view controllers are located in the **Tags** storyboard. We use another storyboard reference to navigate from the **Notes** storyboard to the **Tags** storyboard.



Connecting Storyboards With Storyboard References

I also updated the user interface of the note view controller. In addition to the category of the note, it displays the tags of the note. If a note doesn't have any tags, we show a message saying that it doesn't have any tags yet. The **Edit** button on the right takes the user to the tags view controller.

We also display the tags of a note in the notes view controller, below the title of the note. To make this work, I added a label to the `NoteTableViewCell` class, below the title label.



Note Table View Cell

Preparing the Note Class

Before we update the notes view controller and the note view controller, I'd like to add two convenience computed properties to the extension for the `Note` class. Remember that we created this extension in **Note.swift** in the **Core Data > Extensions** group. You may be wondering why we need these computed properties.

Note.swift

```
1  import Foundation
2
3  extension Note {
4
5      // MARK: - Dates
6
7      ...
8
9      // MARK: - Tags
10
11     var alphabetizedTags: [Tag]? {
12
13     }
14
15     var alphabetizedTagsAsString: String? {
16
17     }
18
19 }
```

Remember that the items of a **To-Many** relationship are unordered. For example, the type of the `tags` property is a `NSSet?` instance of `NSManagedObject` instances. In the user interface, however, I'd like to show the tags in alphabetical order. That's why we need a few computed properties. We don't want to repeat ourselves every time we need an alphabetical list of the tags of a note.

The first computed property we implement returns an optional alphabetized array of tags. We name the computed property `alphabetizedTags` and its of type `[Tag]?`.

Note.swift

```
var alphabetizedTags: [Tag]? {

}
```

First, we make sure the `tags` property of the note isn't equal to `nil`. We also cast the value of the `tags` property to a set of `Tag` instances.

Note.swift

```
var alphabetizedTags: [Tag]? {
    guard let tags = tags as? Set<Tag> else {
        return nil
    }
}
```

We then sort the set of tags. We need to jump through a few hoops to make this work. Remember that, even though the `name` attribute of the `Tag` entity is required, the `name` property of the `Tag` class is an optional. We discussed the reason for this earlier in this book. The optionality of the `name` property makes the closure of the `sorted(by:)` method a bit verbose. But, thanks to the syntax of the Swift language, the result doesn't look too bad.

Note.swift

```
var alphabetizedTags: [Tag]? {
    guard let tags = tags as? Set<Tag> else {
        return nil
    }

    return tags.sorted(by: {
        guard let tag0 = $0.name else { return true }
        guard let tag1 = $1.name else { return true }
        return tag0 < tag1
    })
}
```

Now that we have access to an alphabetized array of tags, we can turn that array into a string. We implement another computed property, `alphabetizedTagsAsString`, of type `String?`. The tags in the string should be separated by commas.

Note.swift

```
var alphabetizedTagsAsString: String? {

}
```

If a note doesn't have any tags, we return `nil`. We could return a placeholder string, for example, "No Tags", but I prefer to keep implementation details like that out of the model layer. That's the responsibility of the controller or, if you're using MVVM, the view model. If the computed property is equal to `nil`, it's up to the controller or the view model to decide how to respond.

Note.swift

```
var alphabetizedTagsAsString: String? {
    guard let tags = alphabetizedTags, tags.count > 0 else {
        return nil
    }
}
```

In the next step, we convert the array of tags to an array of strings. We use `flatMap(_:)` to extract the name of each tag from the array of tags. The advantage of using `flatMap(_:)` is that it automatically skips any tags that don't have a name. This shouldn't happen, but remember that the `name` property is of type `String?`.

Note.swift

```
var alphabetizedTagsAsString: String? {
    guard let tags = alphabetizedTags, tags.count > 0 else {
        return nil
    }

    let names = tags.flatMap { $0.name }
}
```

We turn the array of strings into a string using the `joined(separator:)` method.

Note.swift

```
var alphabetizedTagsAsString: String? {
    guard let tags = alphabetizedTags, tags.count > 0 else {
        return nil
    }

    let names = tags.flatMap { $0.name }
    return names.joined(separator: ", ")
}
```

Updating the Notes View Controller

We can immediately put the `alphabetizedTagsAsString` computed property to use in the notes view controller and the note view controller.

Open **NotesViewController.swift** and navigate to the `configure(_:at:)` method. In this method, we set the `text` property of the `tagsLabel` to the value of the `alphabetizedTagsAsString` computed property we implemented a moment ago.

NotesViewController.swift

```
func configure(_ cell: NoteTableViewCell, at indexPath: IndexPath) {
    // Fetch Note
    let note = fetchedResultsController.object(at: indexPath)

    // Configure Cell
    cell.titleLabel.text = note.title
    cell.contentsLabel.text = note.contents
    cell.tagsLabel.text = note.alphabetizedTagsAsString ?? "No Tags"
    cell.updatedAtLabel.text = updatedAtDateFormatter.string(from: n\
ote.updatedAtAsDate)

    ...
}
```

We use Swift's nil-coalescing operator to show a default value if a note doesn't have any tags.

Updating the Note View Controller

Open **NoteViewController.swift** and navigate to the `updateTagsLabel()` method. In this method, we set the `text` property of the `tagsLabel` using the same approach we used in the notes view controller.

NoteViewController.swift

```
private func updateTagsLabel() {
    // Configure Tags Label
    tagsLabel.text = note?.alphabetizedTagsAsString ?? "No Tags"
}
```

Updating the Tags View Controller

Most interesting are the changes we need to make to the tags view controller. The first method we update is the `configure(_:at:)` method. We ask the note whether it contains the tag that corresponds with the index path. If it does, we highlight the name of the tag by updating the text color of the name label. If the note doesn't contain the tag, we set the text color of the name label to black. This shows the user which tags are assigned to the current note.

TagsViewController.swift

```
func configure(_ cell: TagTableViewCell, at indexPath: IndexPath) {
    // Fetch Tag
    let tag = fetchedResultsController.object(at: indexPath)

    // Configure Cell
    cell.nameLabel.text = tag.name

    if let containsTag = note?.tags?.contains(tag), containsTag == t\
rue {
        cell.nameLabel.textColor = .bitterSweet
    } else {
        cell.nameLabel.textColor = .black
    }
}
```

The user should be able to add and remove tags by tapping a tag in the table view. To add this ability, we update the implementation of the `tableView(_:didSelectRowAt:)` method of the `UITableViewDelegate` protocol.

We fetch the tag that corresponds with the value of the `indexPath` parameter and ask the note if it contains that tag. If it does, we remove it by invoking `removeFromTags(_:)`, one of the convenience methods that's automatically generated for us by Xcode. If the note doesn't contain the tag, we add it by invoking `addToTags(_:)`, passing in the tag.

TagsViewController.swift

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: I\
ndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)

    // Fetch Tag
    let tag = fetchedResultsController.object(at: indexPath)

    if let containsTag = note?.tags?.contains(tag), containsTag == t\
rue {
        note?.removeFromTags(tag)
    } else {
        note?.addToTags(tag)
    }
}
```

Because the tags view controller is powered by a fetched results controller, the table view is automatically updated for us. The same is true for the notes view controller. The fetched results controller detects the addition or deletion of a tag and automatically updates the table view.

Tweaking the Note View Controller

This isn't true for the note view controller. We need to make a tiny change. Revisit **NoteViewController.swift** and navigate to the `managedObjectContextObjectsDidChange(_:)` method. If the note of the note

view controller was modified, we invoke `updateTagsLabel()`. This is the method we implemented earlier in this chapter.

NoteViewController.swift

```
@objc private func managedObjectContextObjectsDidChange(_ notificati\
on: Notification) {
    guard let userInfo = notification.userInfo else { return }
    guard let updates = userInfo[NSUpdatedObjectsKey] as? Set<NSMana\
gedObject> else { return }

    if (updates.filter { return $0 == note }).count > 0 {
        updateTagsLabel()
        updateCategoryLabel()
    }
}
```

That's it. That's how easy it is to work with a **Many-To-Many** relationship using the Core Data framework. Run the application and create a few tags. Assign a tag to a note to see the result of the changes we made.

24 Working In a Multithreaded Environment

We currently use one managed object context, which we created in the `CoreDataManager` class. In the application, we pass the managed object context to the objects that need it. This works fine, but there will be times when one managed object context won't cut it.

What happens if you access the same managed object context from different threads? What happens if you pass a managed object from a background thread to the main thread? But let's start with the basics.

Concurrency Basics

Before we explore solutions for using Core Data in a multithreaded environment, we need to know how Core Data behaves on multiple threads. The documentation is very clear about this. Core Data expects to be run on a single thread. Even though that thread doesn't *have to* be the main thread, Core Data wasn't designed to be used on different threads.

The Core Data team at Apple isn't naive, though. It knows that a persistence framework needs to be accessible from multiple threads. A single thread may be fine for many applications, but more complex applications need a robust, multithreaded persistence framework.

Before I show you how Core Data can be used in a multithreaded environment, I lay out the basic rules for accessing Core Data in a multithreaded application.

Managed Objects

Instances of the `NSManagedObject` class should never be passed from one thread to another. That's a simple rule you need to respect. If you need to pass a managed object from one thread to another, you use a managed object's `objectID` property.

The `objectID` property is of type `NSManagedObjectID` and uniquely identifies a record in the persistent store. The `NSManagedObjectContext` class knows what to do when you hand it an `NSManagedObjectID` instance. The `NSManagedObjectContext` class defines three methods to work with `NSManagedObjectID` instances:

- `object(with:)`
- `existingObject(with:)`
- `registeredObject(for:)`

Each of these methods accepts an instance of the `NSManagedObjectID` class.

The first method, `object(with:)`, returns a managed object that corresponds with the `NSManagedObjectID` instance that's passed in. If the managed object context doesn't have a managed object for that object identifier, it asks the persistent store coordinator. This method always returns a managed object.

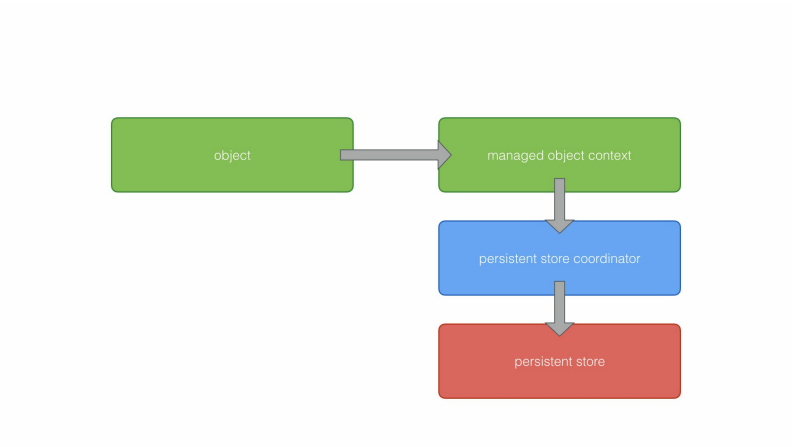
Know that `object(with:)` throws an exception if no record can be found for that object identifier. For example, if the application deleted the record corresponding with the object identifier, Core Data is unable to hand your application the corresponding record. The result is an exception.

The `existingObject(with:)` method behaves similarly. The main difference is that the method throws an error if it can't fetch the managed object corresponding with the object identifier.

The third method, `registeredObject(for:)`, only returns a managed object if the record you're asking for is already registered with the managed object context. In other words, the return value is of type optional `NSManagedObject?`. The managed object context doesn't fetch the corresponding record from the persistent store if it can't find it in the managed object context.

The object identifier of a record is similar, but not identical, to the primary key of a database record. It uniquely identifies the record and enables your application to fetch a particular record regardless of what thread the operation is performed on.

If the application asks a managed object context for a managed object with a particular object identifier, the managed object context first looks if a managed object with that object identifier is registered in the managed object context. If there isn't, the managed object is fetched or returned as a fault. We discuss faults later in this book. Don't worry about it for now.



Fetching a Managed Object

It's important to understand that a managed object context always expects to find a record if you give it an `NSManagedObjectID` instance. That's why `object(with:)` returns an object of type `NSManagedObject`, not `NSManagedObject?`.

Managed Object Context

Creating an `NSManagedObjectContext` instance is a cheap operation. You should **never** share a managed object contexts across threads. This is a hard rule you shouldn't break. The `NSManagedObjectContext` class isn't thread safe. Plain and simple.

Persistent Store Coordinator

Even though the `NSPersistentStoreCoordinator` class isn't thread safe either, the class knows how to lock itself if multiple managed object contexts request access, even if these managed object contexts live and operate on different threads.

It's fine to use a single persistent store coordinator that's accessed by multiple managed object contexts from different threads. This makes Core Data concurrency a little bit easier.

Managing Concurrency

Core Data has come a long way and it used to be a nightmare to use Core Data in a multithreaded environment. You still need to be careful when using Core Data on multiple threads, but it's become easier since iOS 6. Apple added a number of useful APIs to the Core Data framework to make your life as a developer easier.

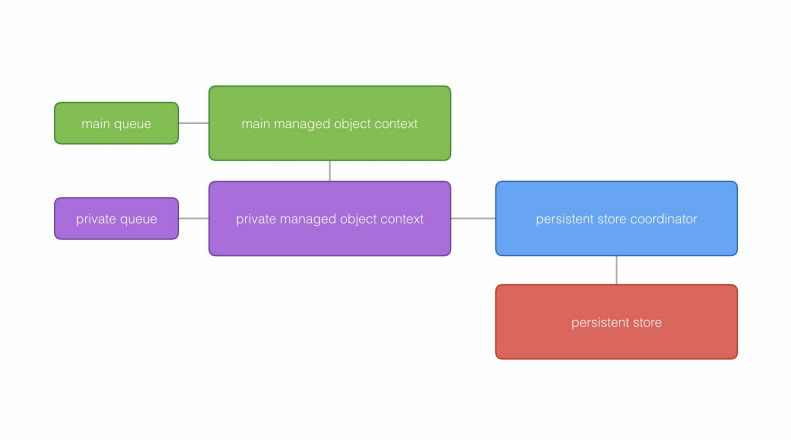
Updating the Core Data Stack

Theory

Before ending this chapter, I want to talk about parent and child managed object contexts, a topic I briefly mentioned in the introduction of this book.

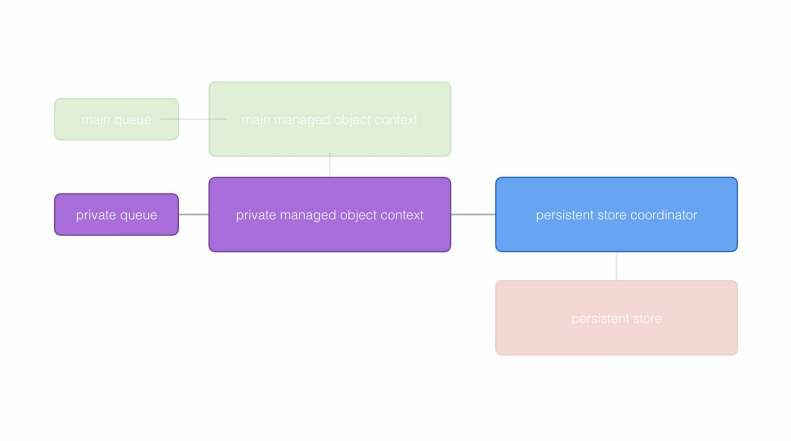
Complex applications that heavily rely on Core Data can run into problems if changes of the managed object context are written to the persistent store on the main thread. Even on modern devices, such operations can result in the main thread being blocked. Because the main thread is also used to update the user interface of your application, the user experiences this as the application freezing up for a moment.

This can be avoided by slightly modifying the Core Data stack of the application. The approach I mostly use looks something like this.



A More Advanced Core Data Stack

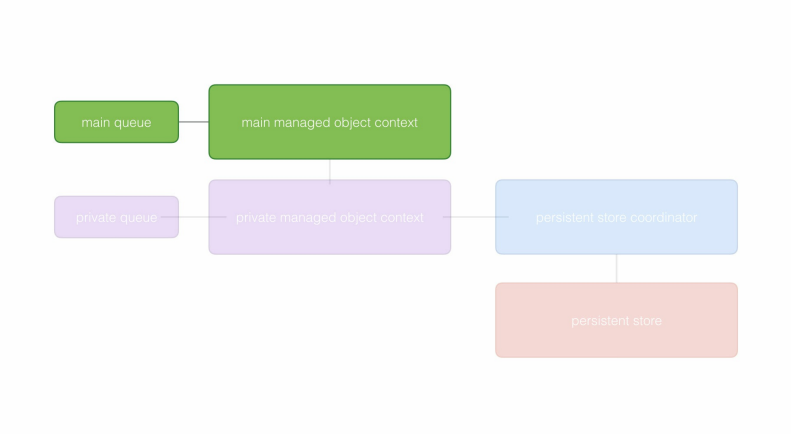
The managed object context linked to the persistent store coordinator isn't associated with the main thread. Instead, it performs its work on a private queue, not on the main queue. When the private managed object context saves its changes, the write operation is performed on that private queue in the background.



The private managed object context performs its operations on a private queue.

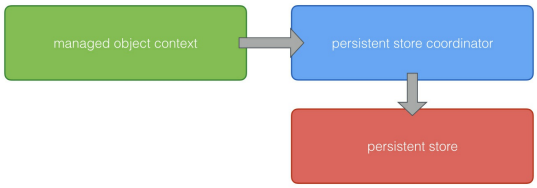
The private managed object context has a child managed object context, which serves as the main managed object context of the application. The

concept of parent-child managed object contexts is key in this scenario.



The private managed object context has a child managed object context.

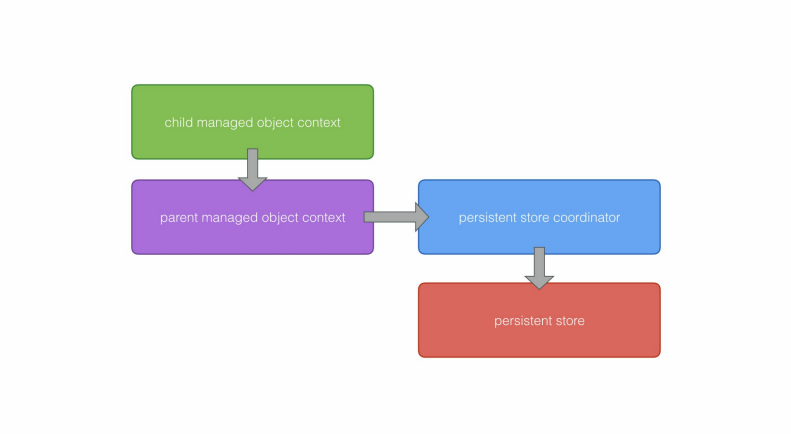
In most scenarios, a managed object context is associated with a persistent store coordinator. When such a managed object context saves its changes, it pushes them to the persistent store coordinator. The persistent store coordinator pushes the changes to the persistent store.



A Managed Object Context Associated With a Persistent Store Coordinator

A **child managed object context** doesn't have a reference to a persistent store coordinator. Instead, it keeps a reference to another managed object context, a **parent managed object context**.

When a child managed object context saves its changes, it pushes them to the parent managed object context. In other words, when a child managed object context saves its changes, the persistent store coordinator is unaware of the save operation. It's only when the parent managed object context performs a save operation that the changes are pushed to the persistent store coordinator and subsequently to the persistent store.



A child managed object context pushes its changes to its parent managed object context.

Because no write operations (no disk I/O) are performed when a child managed object context saves its changes, pushing changes from a child managed object context to its parent is fast and efficient. This also means that the queue on which the operation is performed isn't blocked by a write operation. That's why the main managed object context of the application is the child managed object context of a managed object context that operates in the background on a private queue.

Practice

It's time to put this into practice by updating the `CoreDataManager` class. In the next chapter, we refactor the `CoreDataManager`class to make it more suitable for use in a multithreaded environment.

25 Updating the Core Data Manager for Concurrency

It's time to refactor the `CoreDataManager` class. Let me walk you through the changes we need to make. Don't worry, though, most of the implementation of the `CoreDataManager` class remains unchanged.

Creating a Private Managed Object Context

We start by creating a `privateManagedObjectContext` property of type `NSManagedObjectContext`. It's a private, lazy property.

CoreDataManager.swift

```
1 private lazy var privateManagedObjectContext: NSManagedObjectContext\
2  = {
3
4 }()
```

We initialize a managed object context by invoking `init(concurrencyType:)`. The concurrency type tells Core Data how the managed object context should be managed from a concurrency perspective. What does that mean? The Core Data framework defines three concurrency types:

- `mainQueueConcurrencyType`
- `confinementConcurrencyType`
- `privateQueueConcurrencyType`

The first concurrency type, `mainQueueConcurrencyType`, associates the managed object context with the main queue. This is important if the managed object context is used in conjunction with view controllers or is linked to the application's user interface.

By setting the concurrency type to `privateQueueConcurrencyType`, the managed object context is given a private dispatch queue for performing its operations. The operations performed by the managed object context are not performed on the main thread. That's key.

The `confinementConcurrencyType` concurrency type used to be the default. If you create a managed object context by invoking `init()`, the concurrency type is set to `confinementConcurrencyType`. However, as of iOS 9, the `init()` method of the `NSManagedObjectContext` class is deprecated. A managed object context

should only be created by invoking `init(concurrencyType:)`, passing `mainQueueConcurrencyType` or `privateQueueConcurrencyType` as its argument.

Because we're creating a private managed object context, we pass `privateQueueConcurrencyType` as the argument of `init(concurrencyType:)`.

CoreDataManager.swift

```
// Initialize Managed Object Context
let managedObjectContext = NSManagedObjectContext(concurrencyType: .\
privateQueueConcurrencyType)
```

We set the `persistentStoreCoordinator` property of the private managed object context. This means that a save operation pushes any changes of the managed object context to the persistent store coordinator, which pushes the changes to the persistent store.

CoreDataManager.swift

```
// Configure Managed Object Context
managedObjectContext.persistentStoreCoordinator = self.persistentSto\
reCoordinator
```

We return the managed object context from the closure. This is what the implementation of the `privateManagedObjectContext` property looks like.

CoreDataManager.swift

```
private lazy var privateManagedObjectContext: NSManagedObjectContext\
 = {
    // Initialize Managed Object Context
    let managedObjectContext = NSManagedObjectContext(concurrencyTyp\
e: .privateQueueConcurrencyType)

    // Configure Managed Object Context
    managedObjectContext.persistentStoreCoordinator = self.persisten\
tStoreCoordinator

    return managedObjectContext
}()
```

Updating the Main Managed Object Context

The next step is updating the implementation of the `managedObjectContext` property. First, rename the property to `mainManagedObjectContext` to show that the managed object context is associated with the application's main dispatch queue.

CoreDataManager.swift

```
private(set) lazy var mainManagedObjectContext: NSManagedObjectConte\
xt = {
    ...
}()
```

With what we learned in the previous chapter still fresh in your mind, the change we need to make is easy. The managed object context is created by invoking `init(concurrencyType:)`, passing in `mainQueueConcurrencyType` as its argument. Instead of setting the `persistentStoreCoordinator` property of the managed object context, we set its `parent` property to the private managed object context we created a few moments ago.

CoreDataManager.swift

```
private(set) lazy var mainManagedObjectContext: NSManagedObjectConte\
xt = {
    // Initialize Managed Object Context
    let managedObjectContext = NSManagedObjectContext(concurrencyTyp\
e: .mainQueueConcurrencyType)

    // Configure Managed Object Context
    managedObjectContext.parent = self.privateManagedObjectContext

    return managedObjectContext
}()
```

This means that a save operation pushes changes from the main managed object context to the private managed object context. From a performance point of view, this is more than sufficient for the vast majority of applications that make use of Core Data.

Updating the Save Method

We're not done yet. We also need to update the `saveChanges()` method. It changes pretty dramatically. Before we push any changes from the private managed object context to the persistent store, we need to push the changes from the main managed object context to the private managed object context.

Saving those changes needs to happen on the queue of the managed object context. But how do we know what queue that is? And how can we access that queue?

Fortunately, the Core Data framework can help us with that. To make sure a managed object context is accessed on the queue it's associated with, you use the `perform(_:)` and `performAndWait(_:)` methods.

Both methods accept a closure and the Core Data framework guarantees that the closure is invoked on the queue the managed object context is associated

with. The only difference between both methods is that `performAndWait(_:)` is performed synchronously. As the name implies, it blocks the thread it's called on.

With this in mind, we can continue implementing the `saveChanges()` method. We invoke `performAndWait(_:)` on the main managed object context and, in the closure, we save the main managed object context if it has any changes.

CoreDataManager.swift

```
private func saveChanges() {
    mainManagedObjectContext.performAndWait({
        do {
            if self.mainManagedObjectContext.hasChanges {
                try self.mainManagedObjectContext.save()
            }
        } catch {
            let saveError = error as NSError
            print("Unable to Save Changes of Main Managed Object Con\
text")
            print("\(saveError), \(saveError.localizedDescription)")
        }
    })
}
```

Next, we invoke `perform(_:)` on the private managed object context and in the closure we save the changes of the private managed object context if it has any. This means the private managed object context pushes its changes to the persistent store coordinator.

CoreDataManager.swift

```
private func saveChanges() {
    mainManagedObjectContext.performAndWait({
        do {
            if self.mainManagedObjectContext.hasChanges {
                try self.mainManagedObjectContext.save()
            }
        } catch {
            let saveError = error as NSError
            print("Unable to Save Changes of Main Managed Object Con\
text")
            print("\(saveError), \(saveError.localizedDescription)")
        }
    })

    self.privateManagedObjectContext.perform({
        do {
            if self.privateManagedObjectContext.hasChanges {
                try self.privateManagedObjectContext.save()
            }
        } catch {
            let saveError = error as NSError
            print("Unable to Save Changes of Private Managed Object \
Context")
            print("\(saveError), \(saveError.localizedDescription)")
```

```
        }
    })
}
```

Notice that we first save the changes of the main managed object context. This is important because we need to make sure the private managed object context includes the changes of its child managed object context.

For this reason, we use `performAndWait(_:)` instead of `perform(_:)`. We first want to make sure the changes of the main managed object context are pushed to the private managed object context before pushing the changes of the private managed object context to the persistent store coordinator.

Another Option

There is another option. Take a look at this implementation of the `saveChanges()` method.

CoreDataManager.swift

```
private func saveChanges() {
    mainManagedObjectContext.perform({
        do {
            if self.mainManagedObjectContext.hasChanges {
                try self.mainManagedObjectContext.save()
            }
        } catch {
            let saveError = error as NSError
            print("Unable to Save Changes of Main Managed Object Con\
text")
            print("\(saveError), \(saveError.localizedDescription)")
        }

        self.privateManagedObjectContext.perform({
            do {
                if self.privateManagedObjectContext.hasChanges {
                    try self.privateManagedObjectContext.save()
                }
            } catch {
                let saveError = error as NSError
                print("Unable to Save Changes of Private Managed Obj\
ect Context")
                print("\(saveError), \(saveError.localizedDescriptio\
n)")
            }
        })
    })
}
```

The result is similar. We push the changes of the main managed object context to the private managed object context by invoking `save()` on the main managed object context. But notice that we use `perform()` instead of `performAndWait(_:)`.

After pushing the changes of the main managed object context to its parent, we save the changes of the private managed object context. We do this within the closure we pass to the `perform(_:)` method of the main managed object context. This means that the changes of the main managed object context are saved before those of the private managed object context are pushed to the persistent store coordinator.

When to Save

We currently save the changes of the managed object contexts when it's about to be terminated and when the application is pushed to the background. This is fine, but you need to keep in mind that a crash of your application results in data loss if you adopt this strategy. Any changes that aren't pushed to the persistent store coordinator when the application is suddenly terminated are lost.

You could adopt an alternative approach by saving changes in the private managed object context at regular time intervals or when you know the user isn't actively using your application. This depends on the type of application you're creating. There is no one solution. That's important to understand.

Your knowledge of the Core Data framework has grown quite a bit. Even though concurrency may seem to be an advanced topic, it isn't. If you work with Core Data, then you need to know how Core Data behaves in a multithreaded environment.

26 Using a Better Core Data Stack

Because we updated the `CoreDataManager` class in the previous chapter, we need to make a few changes in the project.

Updating the Notes View Controller

We changed the name of the `managedObjectContext` property to `mainManagedObjectContext`. Open **NotesViewController.swift** and navigate to the `fetchedResultsController` property. Change `managedObjectContext` to `mainManagedObjectContext`.

NotesViewController.swift

```
1  private lazy var fetchedResultsController: NSFetchedResultsControlle\
2  r<Note> = {
3      ...
4
5      // Create Fetched Results Controller
6      let fetchedResultsController = NSFetchedResultsController(fetchR\
7  equest: fetchRequest,
8                                                                manage\
9  dObjectContext: self.coreDataManager.mainManagedObjectContext,
10                                                               sectio\
11 nNameKeyPath: nil,
12                                                               cacheN\
13 ame: nil)
14
15     ...
16 }()
```

We also need to apply this change in the `prepare(for:sender:)` method.

NotesViewController.swift

```
1  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
2      guard let identifier = segue.identifier else { return }
3
4      switch identifier {
5      case Segue.AddNote:
6          ...
7
8          // Configure Destination
9          destination.managedObjectContext = coreDataManager.mainManag\
10 edObjectContext
11     case Segue.Note:
12         ...
13     default:
14         break
15     }
16 }
```

We repeat this change in the `tableView(_:commit:forRowAt:)` method of the `UITableViewDataSource` protocol.

NotesViewController.swift

```
func tableView(_ tableView: UITableView, commit editingStyle: UITabl\
eViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    ...

    // Delete Note
    coreDataManager.mainManagedObjectContext.delete(note)
}
```

That’s it. Even though we could have kept the name of the managed object context unchanged, the name of the property now clearly reflects the nature and purpose of the managed object context. It’s associated with the main queue and it’s this managed object context we need to use for any operations related to the user interface. Because remember that the user interface of an application should always be updated on the main thread.

27 Replacing the Core Data Manager Class

The `CoreDataManager` class is in charge of the Core Data stack of the application. It encapsulates the Core Data stack and only exposes the main managed object context to the rest of the application.

In the previous chapters, we improved the Core Data stack by using a private parent managed object context and a child managed object context that operates on the main queue of the application.

We also use dependency injection to pass the managed object context to the objects that need it. That's a good start. We could make a few other improvements, but that's not the focus of this chapter. This chapter focuses on a brand new addition to the Core Data framework.

Persistent Container

For years, developers have created classes similar to the `CoreDataManager` class because the framework itself didn't provide a similar solution. That, however, has changed very recently. During WWDC 2016, Apple introduced the `NSPersistentContainer` class. This brand new member of the Core Data framework is available as of iOS 10 and macOS 10.12. It looks and behaves very similar to the `CoreDataManager` class we created.

You may be wondering why we didn't use the `NSPersistentContainer` class from the start. There's a very good reason I only tell you about the `NSPersistentContainer` class at this stage of the book. It isn't a good idea to use the `NSPersistentContainer` class without *first* understanding what it can do for you. In other words, you first need to understand how the framework operates before you should use the `NSPersistentContainer` class.

In this chapter, we replace the `CoreDataManager` class with the `NSPersistentContainer` class. The list of changes we need to make is surprisingly small.

Replacing the Core Data Manager

The first change we need to make is replacing the `coreDataManager` property in the notes view controller. We replace it with the `persistentContainer` property. This property is of type `NSPersistentContainer`. The initializer, `init(name:)`, looks very similar to that of the `CoreDataManager` class.

NotesViewController.swift

```
private var persistentContainer = NSPersistentContainer(name: "Notes\
")
```

The name we pass to the initializer is used by the `NSPersistentContainer` class to find the data model of the project. The `NSPersistentContainer` class also defines another initializer that accepts an instance of the `NSManagedObjectModel` class.

Next, we need to replace the references to the `coreDataManager` property in the `NotesViewController` class. The `NSPersistentContainer` class also exposes a managed object context instance that operates on the application's main dispatch queue. This is similar to the `mainManagedObjectContext` property of the `CoreDataManager` class.

The only difference with the `CoreDataManager` class is the name of the property. The main managed object context is accessible through the `viewContext` property. That's another change we need to make.

Navigate to the `fetchedResultsController` property and change `self.coreDataManager.mainManagedObjectContext` to `self.persistentContainer.viewContext`.

NotesViewController.swift

```
private lazy var fetchedResultsController: NSFetchedResultsControlle\
r<Note> = {
    ...

    // Create Fetched Results Controller
    let fetchedResultsController = NSFetchedResultsController(fetchR\
equest: fetchRequest,
                                                              manage\
dObjectContext: self.persistentContainer.viewContext,
                                                              sectio\
nNameKeyPath: nil,
                                                              cacheN\
ame: nil)

    ...
}()
```

We also need to apply this change in the `prepare(for:sender:)` method.

NotesViewController.swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }

    switch identifier {
```

```
    case Segue.AddNote:
        ...

        // Configure Destination
        destination.managedObjectContext = persistentContainer.viewC\
ontext
    case Segue.Note:
        ...
    default:
        break
    }
}
```

We repeat this change in the `tableView(_:commit:forRowAt:)` method of the `UITableViewDataSource` protocol.

NotesViewController.swift

```
func tableView(_ tableView: UITableView, commit editingStyle: UITabl\
eViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    ...

    // Delete Note
    persistentContainer.viewContext.delete(note)
}
```

You can ask a persistent container for a private managed object context by invoking the `newBackgroundContext()` method. But we don't need that in this project.

Adding the Persistent Store

We need to make one more change. In the `CoreDataManager` class, we add the persistent store in the `persistentStoreCoordinator` property. But this operation can take some time. For example, if the application needs to perform one or more migrations, it can take a non-trivial amount of time.

If this is done on the main thread, it can block the main thread. And if this is done during the launch of the application, the operating system can even decide the application takes too long to launch and terminate the application.

That's why some applications crash when you open them after an update. The results can be dramatic. For example, the user could lose their data because of a bad migration or a corrupted persistent store.

It's also important that the application only accesses the Core Data stack *after* it has successfully added the persistent store to the persistent store coordinator.

The solution to these problems is simple. We need to add the persistent store asynchronously on a background queue. And that's an option the `NSPersistentContainer` class offers us. But we need to make a few changes to make this work in our project.

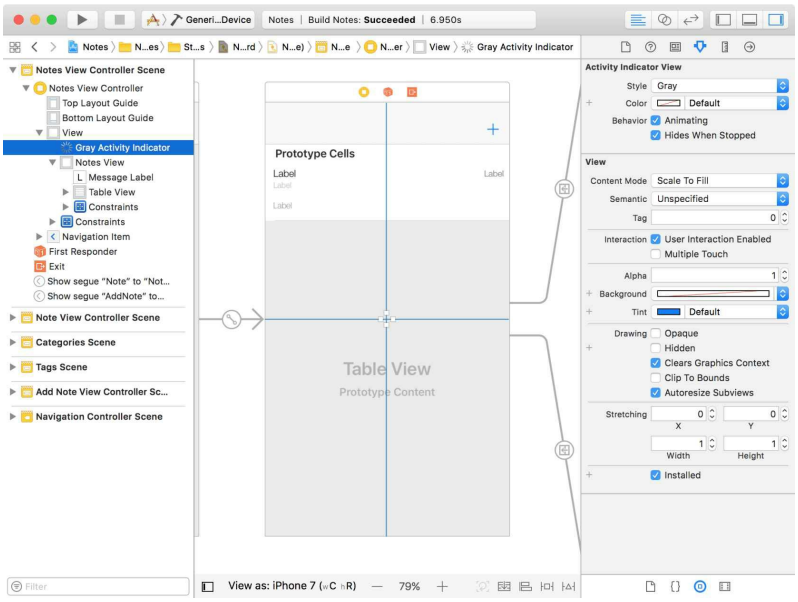
Setting Up the User Interface

We first define an outlet for an activity indicator view. We show the activity indicator view as long as the Core Data stack is being initialized, that is, as long as the persistent store hasn't been added to the persistent store coordinator.

NotesViewController.swift

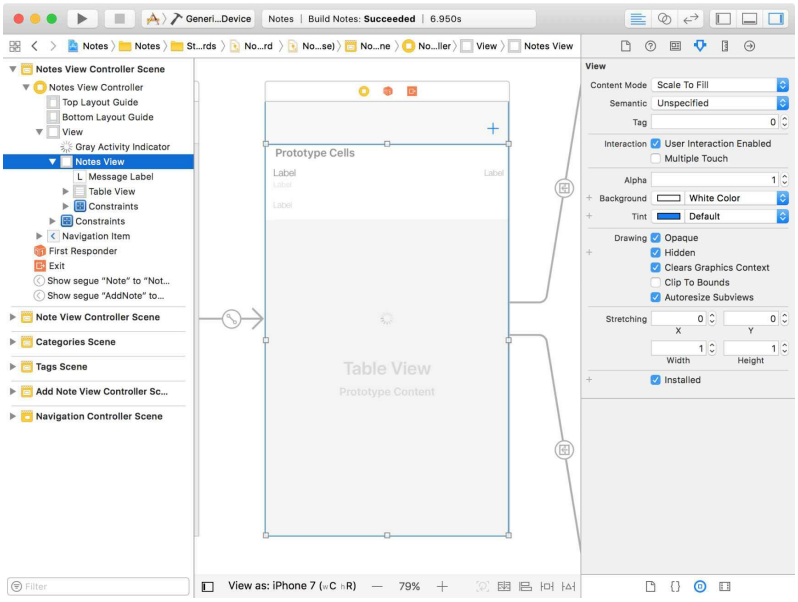
```
1  import UIKit
2  import CoreData
3
4  class NotesViewController: UIViewController {
5
6      ...
7
8      // MARK: - Properties
9
10     @IBOutlet var notesView: UIView!
11     @IBOutlet var messageLabel: UILabel!
12     @IBOutlet var tableView: UITableView!
13     @IBOutlet var activityIndicatorView: UIActivityIndicatorView!
14
15     ...
16
17 }
```

Open **Notes.storyboard** and add an activity indicator view to the notes view controller scene. Make sure you add it to the view of the view controller, not the notes view. Add the necessary constraints, open the **Attributes Inspector** on the right, and check **Animating** and **Hides When Stopped**. Don't forget to connect the activity indicator view with the outlet we created a few moments ago.



Adding an Activity Indicator View

Select the notes view and check **Hidden** to hide it.



Hiding the Notes View

Adding the Persistent Store

Revisit **NotesViewController.swift** and navigate to `viewDidLoad()`. In `viewDidLoad()`, we add the persistent store to the persistent store coordinator by invoking `loadPersistentStores(completionHandler:)`.

NotesViewController.swift

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

    title = "Notes"

    persistentContainer.loadPersistentStores { (persistentStoreDescr\
iption, error) in

    }
```

```
10
11      setupView()
12      fetchNotes()
13      updateView()
14  }
```

The method defines one parameter, a completion handler. The completion handler defines two parameters an `NSPersistentStoreDescription` instance and an optional error. The completion handler is invoked for every persistent store that's added to the persistent store coordinator.

But how does the persistent container know which persistent stores to add? By setting the `persistentStoreDescriptions` property of the persistent container you can specify which persistent stores to add. This property is of type `[NSPersistentStoreDescription]`, an array of `NSPersistentStoreDescription` objects.

An `NSPersistentStoreDescription` object encapsulates the information needed to create a persistent store. It contains information such as the location of the persistent store, the type, and the migration strategy.

If you don't explicitly set the `persistentStoreDescriptions` property, the persistent container tries to find or create a persistent store based on the name of the persistent container and a set of sensible defaults. That should work fine for this project. Remember that we initialized the persistent container with a name of **Notes**.

The completion handler of the `loadPersistentStores(completionHandler:)` method is invoked for each persistent store that's added. This is a bit unfortunate. It means the developer needs to keep track of the state of the Core Data stack. If multiple persistent stores need to be added, you need to make sure you access the Core Data stack when every persistent store is successfully added to the persistent store coordinator.

For a persistent container with one persistent store, though, the setup is simple. If adding the persistent store is successful, the user interface is shown to the user by invoking `setupView()`. We also invoke `fetchNotes()` to fetch the user's notes, and `updateView()`, to update the user interface. Notice that we don't explicitly dispatch the calls to `setupView()`, `fetchNotes()`, and `updateView()` to the main thread. The documentation states that the completion handler of `loadPersistentStores(completionHandler:)` is invoked on the calling thread, the main thread in this example.

NotesViewController.swift

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

```
3
4       title = "Notes"
5
6       persistentContainer.loadPersistentStores { (persistentStoreDescr\
7   iption, error) in
8           if let error = error {
9               print("Unable to Add Persistent Store")
10              print("\(error), \(error.localizedDescription)")
11
12          } else {
13              self.setupView()
14              self.fetchNotes()
15              self.updateView()
16          }
17      }
18  }
```

We need to make a small change to the `setupView()` method. We stop animating the activity indicator view and we show the notes view. That's it.

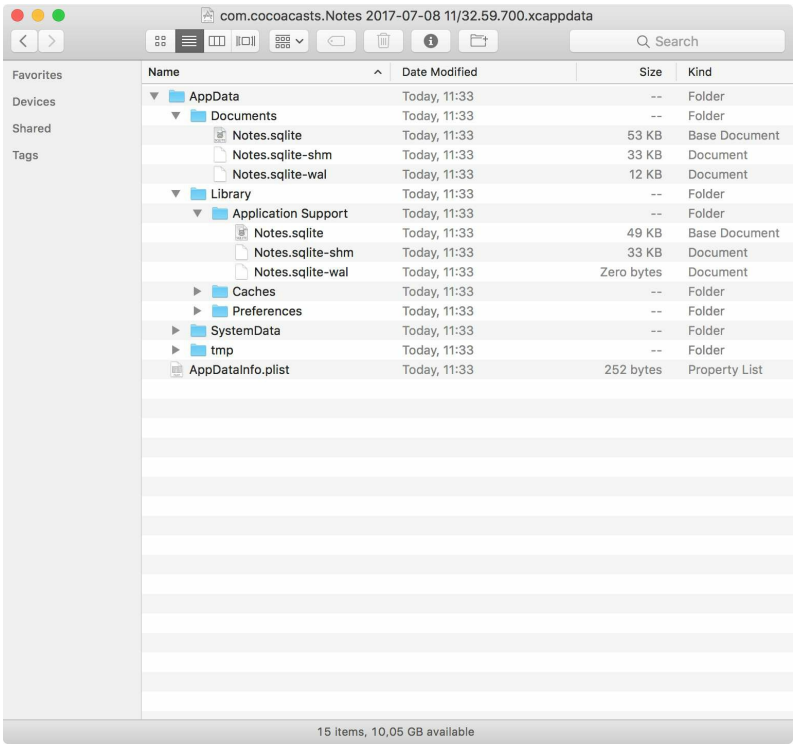
NotesViewController.swift

```
1 private func setupView() {
2       activityIndicatorView.stopAnimating()
3       notesView.isHidden = false
4
5       setupMessageLabel()
6       setupTableView()
7   }
```

Run the application to see the result. Adding the persistent store usually takes very, very little time. This means that you won't see the activity indicator view when you launch the application because it's already hidden when the application is ready to use.

But there is one thing missing. The notes we created earlier. This is easy to explain. The default location of the persistent store when using the `NSPersistentContainer` class isn't the same as the location we used for the persistent store in the `CoreDataManager` class.

By default, the persistent container stores the persistent store in the **Application Support** directory of the **Library** directory. We can verify this by inspecting the application's container.



Locating the Persistent Store

The old persistent store lives in the **Documents** directory whereas the new persistent store is located in the **Application Support** directory of the **Library** directory.

This is fine and we only run into this issue because we replaced the Core Data manager with a `NSPersistentContainer` instance. If you want to modify the default location of the persistent store, you need to subclass the `NSPersistentContainer` class and override the `defaultDirectoryURL()` class method.

Conclusion

It's clear the `NSPersistentContainer` class is a welcome addition to the Core Data framework. It fulfills the needs of many Core Data applications and it offers a modern, easy-to-use API.

As I mentioned earlier, there's one caveat. The downside is that many developers new to Core Data won't bother learning the ins and outs of the framework. As a result, they will inevitably run into problems at some point. By picking up this book, you've avoided that mistake.

28 Understanding Faulting

In this chapter, I'd like to discuss a concept that often confuses developers new to Core Data, **faulting**. Before I explain what faulting is, I want to show it to you.

Make sure the application contains a few notes, a few categories, and a few tags. That's important to illustrate the concept of faulting.

Exploring Faults

Open **NotesViewController.swift** and navigate to the `fetchNotes()` method. Replace the current implementation and create a fetch request for the **Note** entity in a `perform(_:)` closure of the main managed object context.

NotesViewController.swift

```
1 private func fetchNotes() {
2     coreDataManager.mainManagedObjectContext.perform {
3         do {
4             // Create Fetch Request
5             let fetchRequest: NSFetchRequest<Note> = Note.fetchReque\
6 st()
7
8             // Fetch Notes
9             let notes = try fetchRequest.execute()
10
11            if let note = notes.first {
12                print(note)
13            }
14
15        } catch {
16            print(error)
17        }
18    }
19 }
```

We execute the fetch request and print the first note to the console. Run the application to see the result. This is what the output looks like.

```
1 <Note: 0x1c009f4f0> (entity: Note; id: 0xd000000000040000 <x-coredat\
2 a://A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Note/p1> ; data: <fault>)
```

You may not see anything unusual. We fetched a record of the **Note** entity. But notice that the data associated with the record isn't present. Instead we see the word **fault**.

Now that you've seen faulting in action, it's time to explain what it is and why it's so important for Core Data.

What Is a Fault

Core Data is a framework that's incredibly performant thanks to the hard work of the Core Data team at Apple. As you know, Core Data can only operate on records of the persistent store once they're loaded into memory.

This is only possible because Core Data is heavily optimized to keep its memory footprint as low as possible. One of the techniques Core Data uses to accomplish this is **faulting**.

Faulting wasn't invented by the Core Data team at Apple. Several other frameworks use a similar strategy to accomplish similar goals. Ruby on Rails and Ember come to mind.

Even though faulting may look mysterious at first, the idea is simple. Core Data only fetches the data it absolutely needs to satisfy the needs of your application. In the above example, Core Data hasn't fetched the property values of the note yet. Why is that? Because we haven't asked for it. Since we don't access any of the properties of the note, Core Data hasn't bothered fetching the note's property values. Core Data is efficient and performant by being lazy.

The idea of faulting is simple, but the underlying implementation is an advanced bit of programming. Fortunately, we don't have to worry about that. That's the responsibility of the framework. Faulting just works.

Firing a Fault

Let me show you how faulting works with another example. Below the print statement, we safely unwrap the value of the `title` property and print it to the console and we also add another print statement for the note.

NotesViewController.swift

```
1  private func fetchNotes() {
2      coreDataManager.mainManagedObjectContext.perform {
3          do {
4              // Create Fetch Request
5              let fetchRequest: NSFetchRequest<Note> = Note.fetchReque\
6  st()
7
8              // Fetch Notes
9              let notes = try fetchRequest.execute()
10
11             if let note = notes.first {
12                 print(note)
13
```

```
                if let title = note.title {
                    print(title)
                }

                print(note)
            }

        } catch {
            print(error)
        }
    }
}
```

What's happening here. We print the note to the console, ask the value of one of the properties of the note, and print the note again. Why we do this becomes clear when we inspect the results in the console. Run the application. This is what the output looks like. Let's break it down.

```
<Note: 0x1c409b620> (entity: Note; id: 0xd000000000040000 <x-coredat\
a://A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Note/p1> ; data: <fault>)
My First Note
<Note: 0x1c409b620> (entity: Note; id: 0xd000000000040000 <x-coredat\
a://A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Note/p1> ; data: {
    category = "0xd000000000080002 <x-coredata://A36D0B58-5E20-4F2C-\
AC20-111EC9F0D0E3/Category/p2>";
    contents = "Some text ...";
    createdAt = "2017-07-06 07:22:18 +0000";
    tags = "<relationship fault: 0x1c403e900 'tags'>";
    title = "My First Note";
    updatedAt = "2017-07-07 09:15:08 +0000";
})
```

The first print statement shows the fault we discussed earlier. This isn't new.

```
<Note: 0x1c409b620> (entity: Note; id: 0xd000000000040000 <x-coredat\
a://A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Note/p1> ; data: <fault>)
```

Despite this fault, we can access the value of the `title` property and print it to the console. That's interesting.

```
My First Note
```

And this is confirmed by the third print statement in which we print the note again.

```
<Note: 0x1c409b620> (entity: Note; id: 0xd000000000040000 <x-coredat\
a://A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Note/p1> ; data: {
    category = "0xd000000000080002 <x-coredata://A36D0B58-5E20-4F2C-\
AC20-111EC9F0D0E3/Category/p2>";
    contents = "Some text ...";
    createdAt = "2017-07-06 07:22:18 +0000";
    tags = "<relationship fault: 0x1c403e900 'tags'>";
    title = "My First Note";
```

```
    updatedAt = "2017-07-07 09:15:08 +0000";
})
```

Let me explain what's happening under the hood. Core Data gives us what we ask for and exactly that. Nothing more. We first asked the framework for the user's notes and Core Data diligently gave us the list of notes. But, as you can see in the console, it's a list of empty records.

From the moment we ask for the value of a property of one of the records, Core Data jumps into action and fetches the data from the persistent store. This is better known as **firing a fault**. But it doesn't just fetch the value of the `title` property. As you can see in the console, Core Data fetches the values of every property of the note with the exception of relationships.

```
<Note: 0x1c409b620> (entity: Note; id: 0xd000000000040000 <x-coredat\
a://A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Note/p1> ; data: {
    category = "0xd000000000080002 <x-coredata://A36D0B58-5E20-4F2C-\
AC20-111EC9F0D0E3/Category/p2>";
    contents = "Some text ...";
    createdAt = "2017-07-06 07:22:18 +0000";
    tags = "<relationship fault: 0x1c403e900 'tags'>";
    title = "My First Note";
    updatedAt = "2017-07-07 09:15:08 +0000";
})
```

Notice that the value of the `tags` property is missing. Instead, Xcode displays **relationship fault**. This means that the tags of the note haven't been fetched yet.

```
tags = "<relationship fault: 0x1c403e900 'tags'>";
```

And the same applies to the category property. Even though it seems as if Core Data has fetched the data for the category record of the note, it hasn't.

```
category = "0xd000000000080002 <x-coredata://A36D0B58-5E20-4F2C-AC20\
-111EC9F0D0E3/Category/p2>";
```

This becomes clear if we print the value of the category property and run the application again.

NotesViewController.swift

```
private func fetchNotes() {
    coreDataManager.mainManagedObjectContext.perform {
        do {
            // Create Fetch Request
            let fetchRequest: NSFetchRequest<Note> = Note.fetchReque\
st()

            // Fetch Notes
```

```
            let notes = try fetchRequest.execute()

            if let note = notes.first {
                print(note)

                if let title = note.title {
                    print(title)
                }

                print(note)

                if let category = note.category {
                    print(category)
                }
            }

        } catch {
            print(error)
        }
    }
}
```

```
<Category: 0x1c009f3b0> (entity: Category; id: 0xd000000000080002 <x\
-coredata://A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Category/p2> ; data\
: <fault>)
```

I hope it's starting to become clear that Core Data is very lazy … but in a good way. It fetches the minimum amount of data to satisfy the needs of the application.

Faulting and Relationships

Let's print the value of the `tags` property to the console.

NotesViewController.swift

```
private func fetchNotes() {
    coreDataManager.mainManagedObjectContext.perform {
        do {
            // Create Fetch Request
            let fetchRequest: NSFetchRequest<Note> = Note.fetchReque\
st()

            // Fetch Notes
            let notes = try fetchRequest.execute()

            if let note = notes.first {
                if let tags = note.tags as? Set<Tag> {
                    print(tags)

                    for tag in tags {
                        print(tag.name ?? "")
                    }
                }
            }

        } catch {
            print(error)
```

```
23          }
24      }
25  }
```

Core Data hands us a set of objects, but it hasn't actually fetched the tag records itself. The data of the tag records are faults.

```
1 [<Tag: 0x1c009a5e0> (entity: Tag; id: 0xd000000000040004 <x-coredata\
2 ://A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Tag/p1> ; data: <fault>), <T\
3 ag: 0x1c009a540> (entity: Tag; id: 0xd000000000080004 <x-coredata://\
4 A36D0B58-5E20-4F2C-AC20-111EC9F0D0E3/Tag/p2> ; data: <fault>)]
```

The data is fetched the moment we access it. In this example, we ask each tag for the value of its `name` property.

```
1 Monday
2 Family
```

Unable to Fulfill Fault

It's important that you know about and understand Core Data faulting. But the reason for including this chapter in the book is because of a problem many developers working with Core Data run into, firing a fault that cannot be fulfilled by Core Data.

When Core Data tries to fetch data from the persistent store that no longer exists, it tells you it's unable to fulfill the fault. In earlier versions of the framework, Core Data would throw an exception, resulting in a crash of the application.

Fortunately, Core Data has evolved over the years and the framework has become better at handling issues like this. As of iOS 9 and macOS 10.11, the `NSManagedObjectContext` class defines a new property, `shouldDeleteInaccessibleFaults`. This property is set to `true` by default. But let me show you what happens if this property is set to `false`. This is the old behavior of the framework.

Open the `CoreDataManager` class and set the `shouldDeleteInaccessibleFaults` property to `false` for both the main managed object context and the private managed object context.

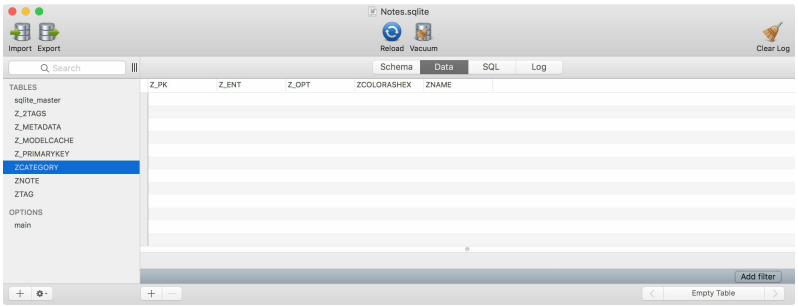
CoreDataManager.swift

```
1 private(set) lazy var mainManagedObjectContext: NSManagedObjectConte\
2 xt = {
3     // Initialize Managed Object Context
4     let managedObjectContext = NSManagedObjectContext(concurrencyTyp\
5 e: .mainQueueConcurrencyType)
6
```

```
    // Configure Managed Object Context
    managedObjectContext.shouldDeleteInaccessibleFaults = false
    managedObjectContext.parent = self.privateManagedObjectContext

    return managedObjectContext
}()

private lazy var privateManagedObjectContext: NSManagedObjectContext\
 = {
    // Initialize Managed Object Context
    let managedObjectContext = NSManagedObjectContext(concurrencyTyp\
e: .privateQueueConcurrencyType)

    // Configure Managed Object Context
    managedObjectContext.shouldDeleteInaccessibleFaults = false
    managedObjectContext.persistentStoreCoordinator = self.persisten\
tStoreCoordinator

    return managedObjectContext
}()
```

Run the application and assign a category to a note. Push the application to the background to save the changes and stop the application.

To show you the problem, I'm going to modify the persistent store. I only do this to show you what could happen in production. We delete the category record from the database.



Deleting the Category of a Note

If we run the application again, we run into an exception. If we inspect the output in the console, we see the reason of the exception. The reason isn't surprising. Core Data is unable to fulfill a fault.

```
Notes[1254:806546] *** Terminating app due to uncaught exception 'NS\
ObjectInaccessibleException', reason: 'CoreData could not fulfill a \
fault for '0xd000000000080002 <x-coredata://A36D0B58-5E20-4F2C-AC20-\
111EC9F0D0E3/Category/p2>''
```

To show the color of the category in the notes view controller, Core Data needs to fetch the category for the note. But the category no longer exists since we removed it from the persistent store, the SQLite database.

Even though we tampered with the database, this problem can also occur in production. It used to drive developers unfamiliar with the framework crazy. As

of iOS 9 and macOS 10.11, Core Data gives developers the option to fail elegantly by deleting any faults that are inaccessible.

29 Where to Go From Here

My hope is that this book has taught you that Core Data isn't as difficult as many developers believe it to be. Core Data isn't complex once you understand how the various pieces of the framework fit together.

Core Data Isn't Scary

It's true that the Core Data stack looks scary when you first encounter it in the wild, but once you understand how the managed object context, the managed object model, and the persistent store coordinator work together, it isn't that difficult.

Many developers make mistakes by not understanding or taking the time to learn the basics. And that's part of the reason some people don't want to work with the framework.

I truly love working with Core Data because I like how the framework is engineered and I know that it's earned its stripes over the more than ten years it's been around. The automatic code generation introduced in Xcode 8 and the nice additions the Swift language brings make the framework even better.

Start Using Core Data

What's the next step for you? Simple. Start using the framework in your projects. Remember to respect the rules we learned in this book, especially the concurrency rules. But don't overcomplicate your application if it isn't necessary.

That said, keep in mind how the users of your application plan to use your application. Take Notes as an example. We've used the application with a handful of notes, but some users may have hundreds or thousands of notes. How will that impact performance? Core Data is a robust persistence framework and it's a great fit for many applications that require a persistence solution.

Testing

Remember to test the migrations of your application. This is an often overlooked problem. Let me illustrate this with an example. Imagine a user restores a backup of their device and installs an older version of your application. After restoring the backup, they update your application to the

latest version, a version with a data model that differs from the old restored version. This means the application needs to perform several migrations. And if the user has many records stored in the persistent store, this can take several seconds if not longer.

Even if this is an edge case, you need to understand that a problem may result in data loss. Depending on the type of application, this can be a nightmare for the user. You need to prevent data loss at all cost. The user trusts you with their data. Don't take that responsibility lightly.

Libraries

I also want to say a few words about third party libraries for Core Data. My advise is to avoid them if possible. Core Data is a first party framework with a great API. Instead of using a third party library, why don't you write a small library yourself with a handful of convenience methods. That goes a long way.

If you know and understand Core Data, then a wrapper around the framework is unnecessary. Give it a try before choosing for more complexity and one more dependency. Don't you agree that the code we wrote in this book isn't complex? And this has nothing to do with the application. Core Data doesn't need to be complicated.

Data Model

In this book, I emphasized how important it is to take the time to create the data model of your application. The data model can and will change over time, but you only get a first try once.

Once the application is in the hands of your users, it's a nightmare to make drastic changes without running into problems. This isn't a Core Data problem. Most persistence solutions face these issues. That's inherent to data persistence.

Take your time to plan ahead. Keep it as simple as possible and only add complexity if necessary.

Continue Learning

The Core Data framework has a lot more to offer than what you learned in this book. It's a very powerful and flexible framework. It's true that most of the more advanced features are used less frequently, but they can sometimes save you a lot of time and frustration.

I encourage you to continue exploring the framework. If you find yourself wrestling with Core Data, then there's probably a better solution to achieve

your goal.

We covered a lot of ground in this book. It’s now time to use what you’ve learned in your projects. If you have any feedback or questions, reach out to me, send me an email (bart@cocoacasts.com), or talk to me on Twitter (@_bartjacobs). I’m curious to hear your feedback and questions.

Table of Contents