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 2017-11-29

Contents

Welcome **1**
Xcode 9 and Swift 4 1
What You'll Learn 1
How to Use This Book 2

1 What Is Core Data **3**
Core Data Manages an Object Graph 3
When to Use Core Data 4
Core Data & SQLite 5
Core Data Goes Much Further 5
Drawbacks 5

2 Building Notes **7**

3 Exploring the Core Data Stack **9**
Managed Object Model 10
Managed Object Context 11
Persistent Store Coordinator 12
How Does Core Data Work 13

4 Creating the Project **19**

5 Setting Up the Core Data Stack **23**
Managed Object Context 29
Managed Object Model 31
Persistent Store Coordinator 32
Adding a Data Model 35

Setting Up the Core Data Stack 39

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.

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].

[1] https://cocoacasts.com/mastering-core-data-with-swift-3/

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.

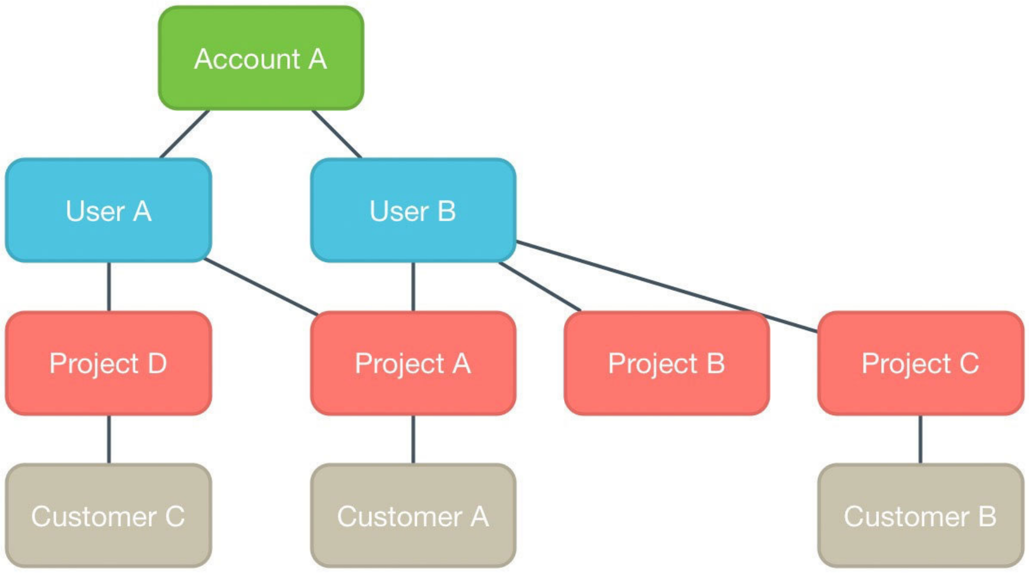
Core Data is the **M** in **MVC**, the model layer of your application. Even though Core Data can persist data to disk, data persistence is actually an optional feature of the framework. Core Data is first and foremost a framework for managing an object graph.

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[2] FMDB[3]. 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.

[2] https://github.com/ccgus
[3] https://github.com/ccgus/fmdb

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.

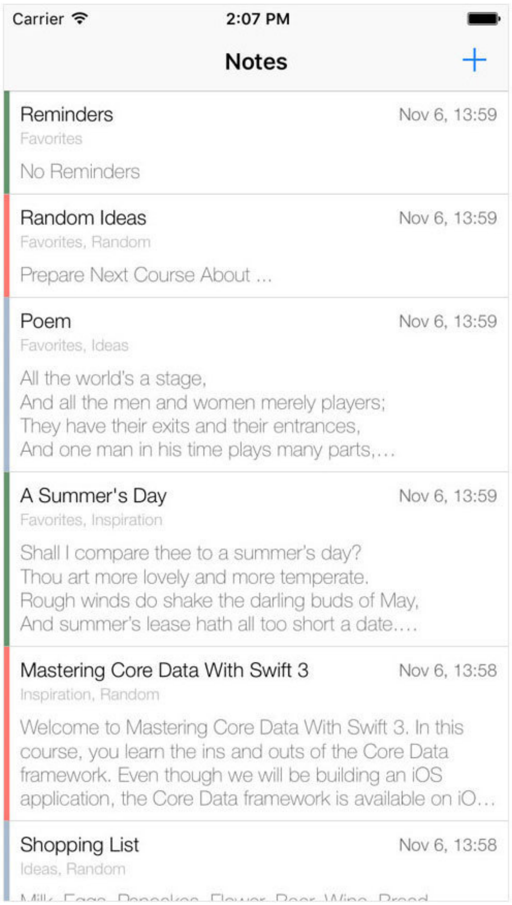
Multithreading

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.



Reminders Nov 6, 13:59
Favorites
No Reminders

Random Ideas Nov 6, 13:59
Favorites, Random
Prepare Next Course About ...

Poem Nov 6, 13:59
Favorites, Ideas
All the world's a stage,
And all the men and women merely players;
They have their exits and their entrances,
And one man in his time plays many parts,...

A Summer's Day Nov 6, 13:59
Favorites, Inspiration
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate.
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date....

Mastering Core Data With Swift 3 Nov 6, 13:58
Inspiration, Random
Welcome to Mastering Core Data With Swift 3. In this course, you learn the ins and outs of the Core Data framework. Even though we will be building an iOS application, the Core Data framework is available on iO...

Shopping List Nov 6, 13:58
Ideas, Random

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.

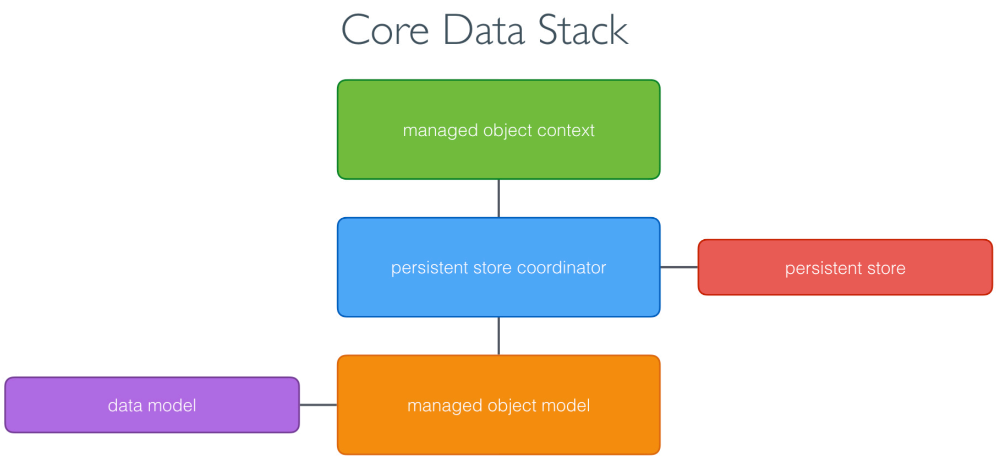
3 Exploring the Core Data Stack

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.

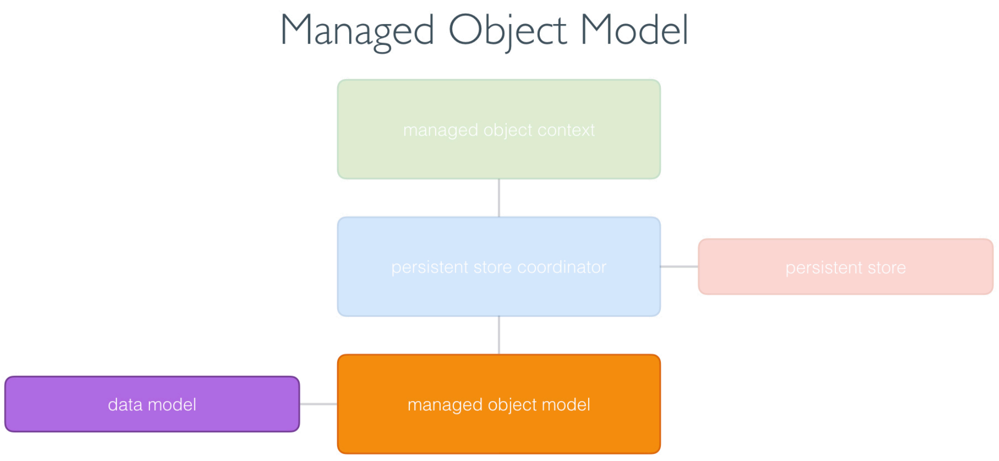


Core Data Stack

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

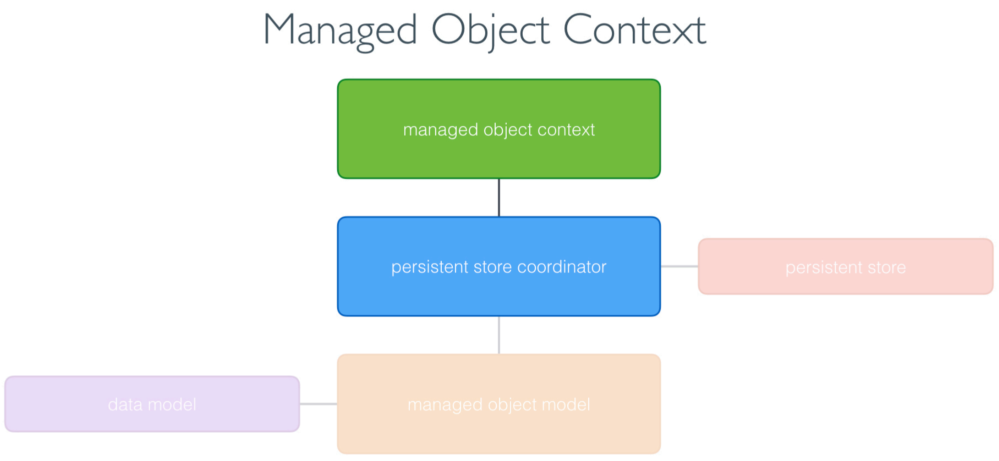
The data model is represented by a file in the application bundle that contains the data schema of the application. The data schema is nothing more than a collection of entities. An entity can have attributes and relationships, which make up the data model of the application.

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.

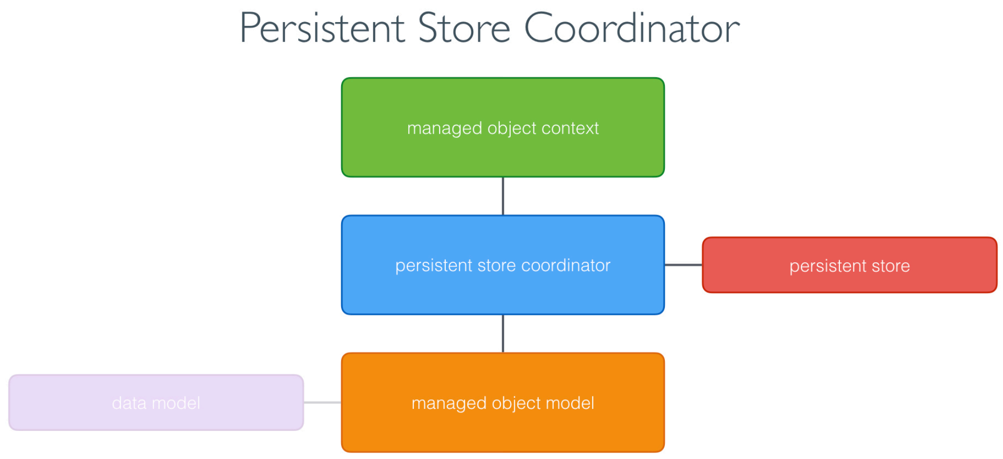


Managed Object Context

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:

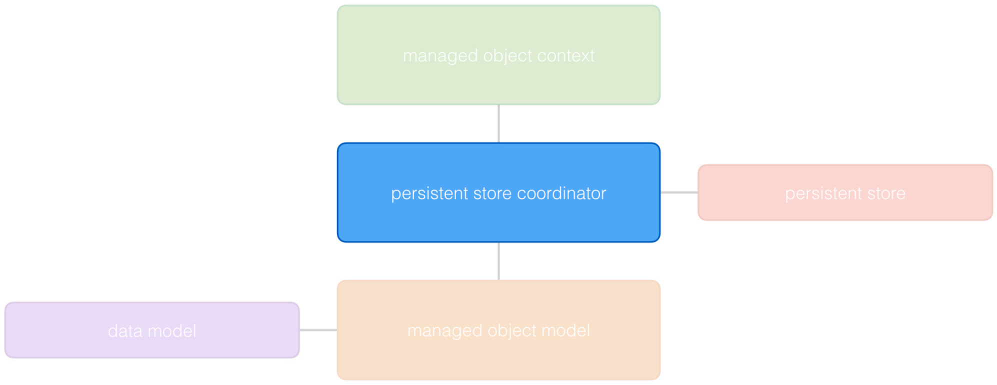
- a SQLite database
- a binary store
- an in-memory store

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.

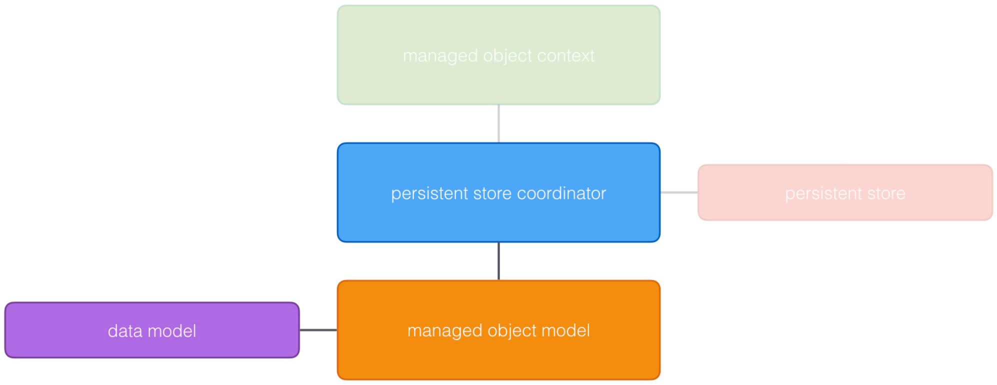
How Does Core Data Work

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.



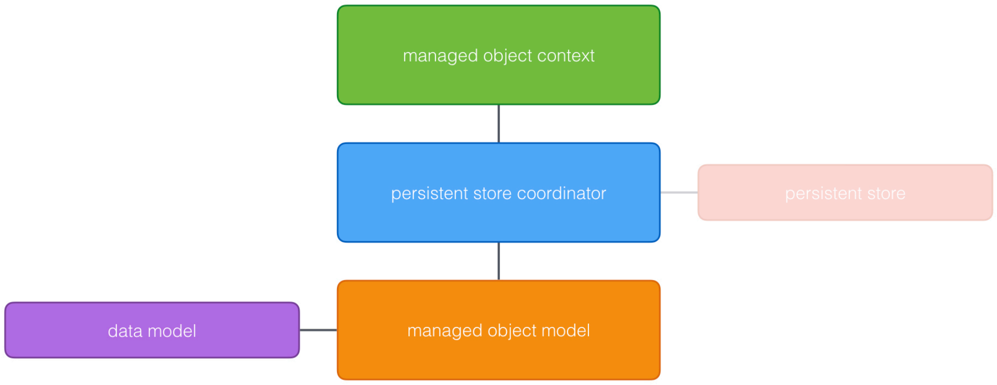
The persistent store coordinator is instantiated first.

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.



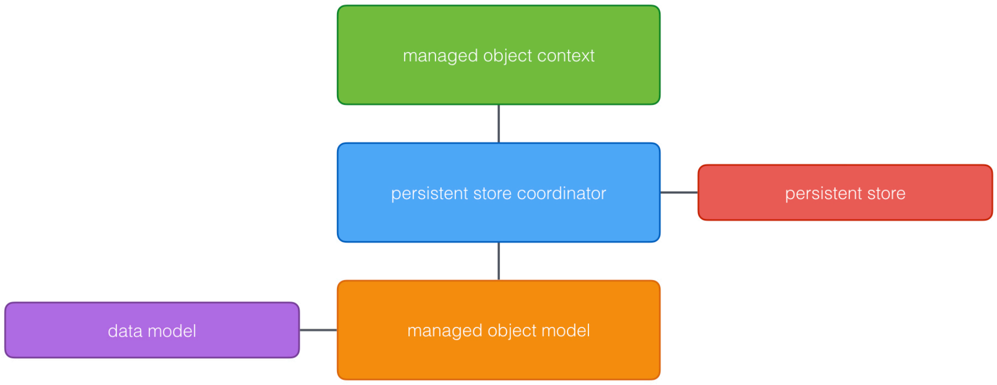
The persistent store coordinator needs a managed object model.

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.

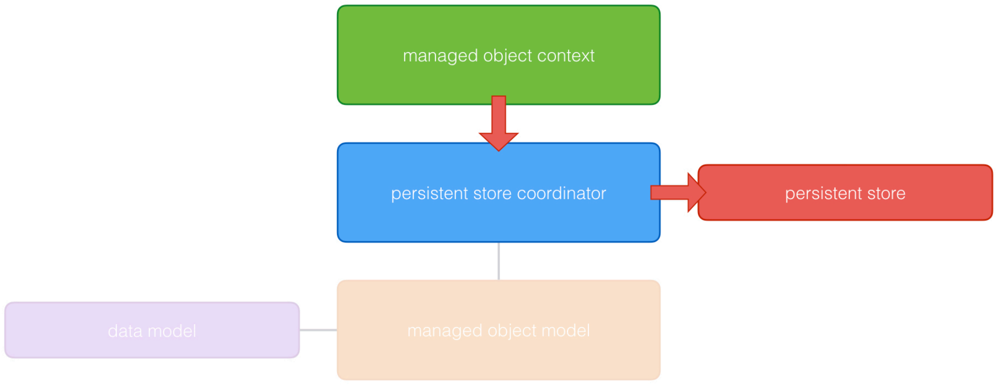
With the Core Data stack set up, the application is ready to use Core Data to interact with the application's persistent store. In most cases, your application interacts with the persistent store coordinator through the managed object context.



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.



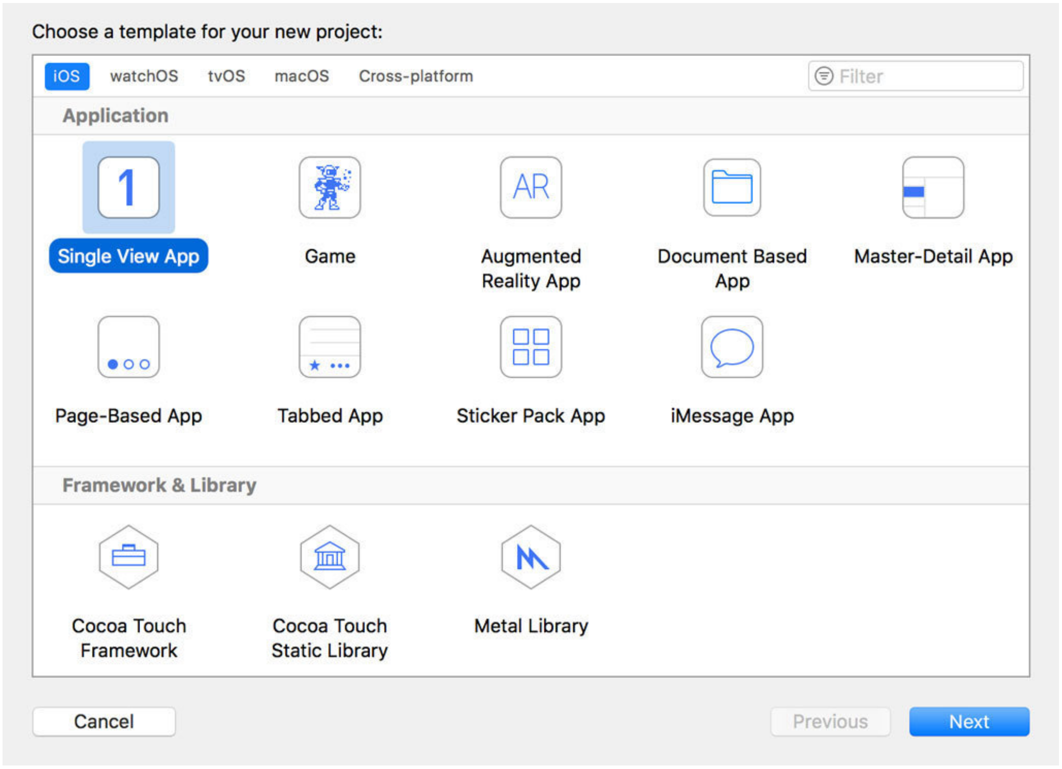
The managed object context pushes changes to the persistent store coordinator, which sends them to the persistent store.

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.

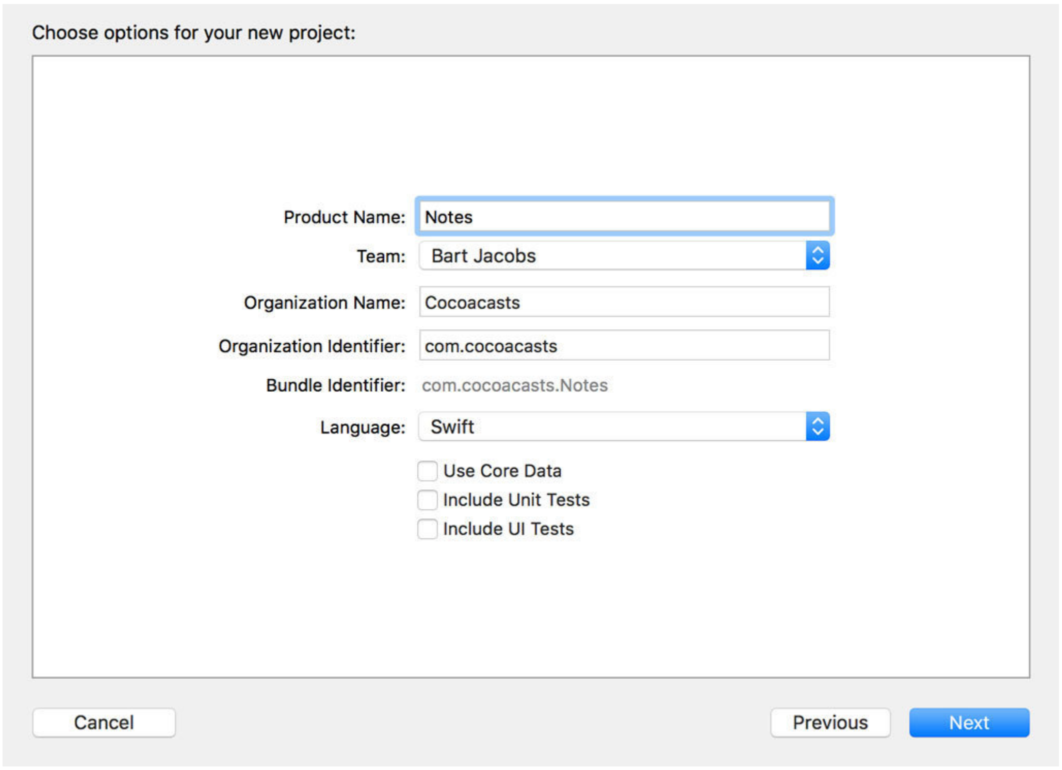
4 Creating the Project

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

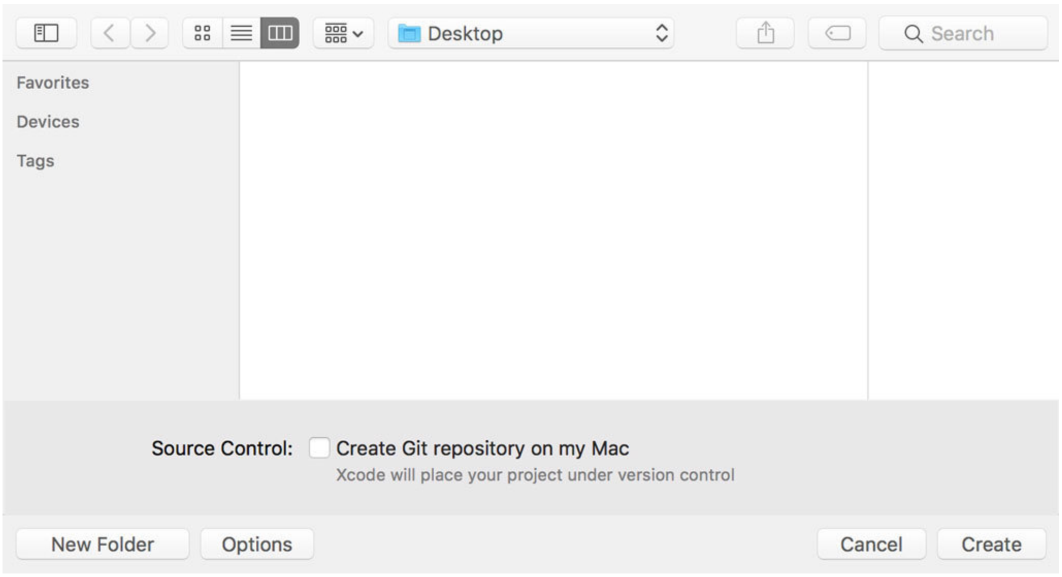
Name the project **Notes**, set **Language** to **Swift**, and, if you're using Xcode 8, set **Devices** to **iPhone**. Make sure **Use Core Data** is unchecked. We're going to start from scratch.


Choose options for your new project:

Product Name: Notes
Team: Bart Jacobs
Organization Name: Cocoacasts
Organization Identifier: com.cocoacasts
Bundle Identifier: com.cocoacasts.Notes
Language: Swift
Use Core Data
Include Unit Tests
Include UI Tests

Configuring the Project

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



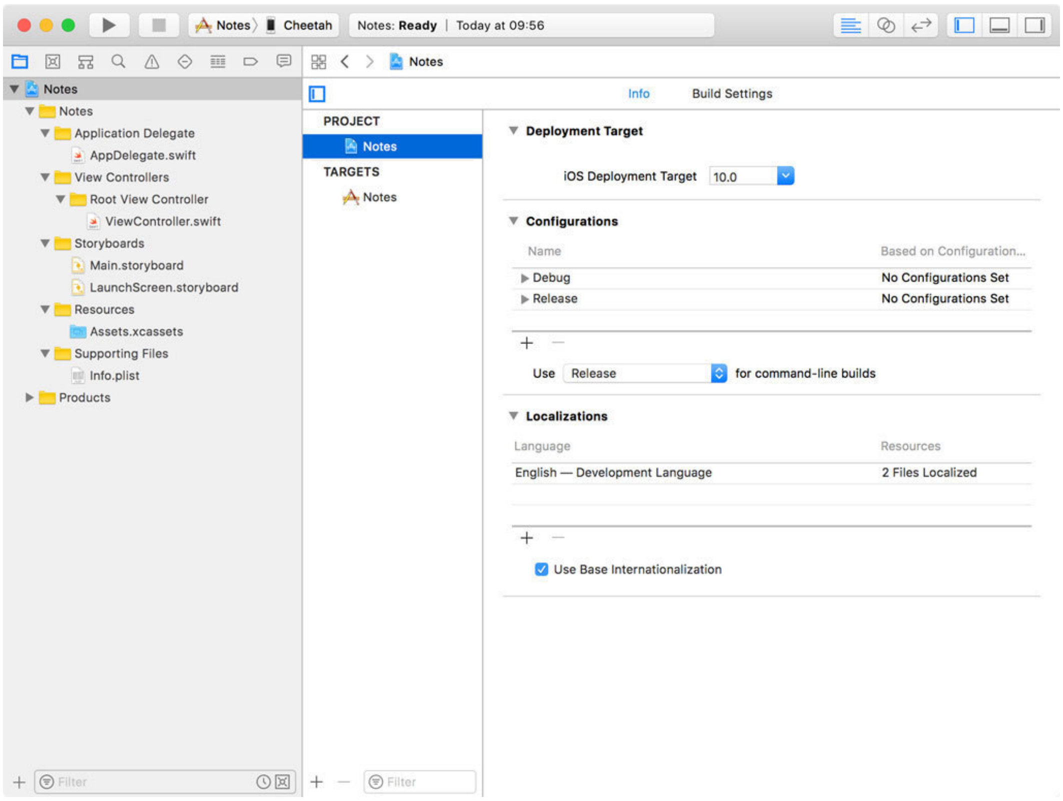
Source Control: Create Git repository on my Mac
Xcode will place your project under version control

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?



Notes
- Notes
  - Application Delegate
    - AppDelegate.swift
  - View Controllers
    - Root View Controller
      - ViewController.swift
  - Storyboards
    - Main.storyboard
    - LaunchScreen.storyboard
  - Resources
    - Assets.xcassets
  - Supporting Files
    - Info.plist
- Products

Deployment Target

iOS Deployment Target 10.0

Configurations

Name	Based on Configuration...
Debug	No Configurations Set
Release	No Configurations Set

Use Release for command-line builds

Localizations

Language	Resources
English — Development Language	2 Files Localized

✓ Use Base Internationalization

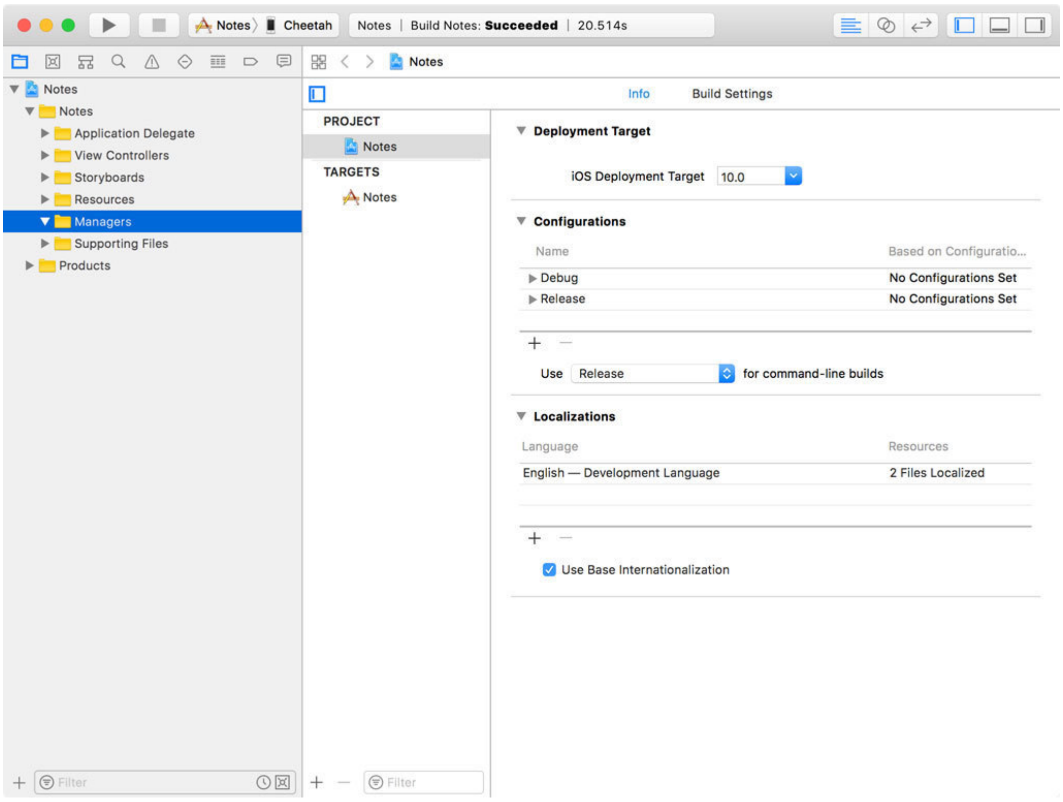
Updating the Project Structure

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**.



Deployment Target

iOS Deployment Target 10.0

Configurations

Name	Based on Configuratio...
Debug	No Configurations Set
Release	No Configurations Set

Use Release for command-line builds

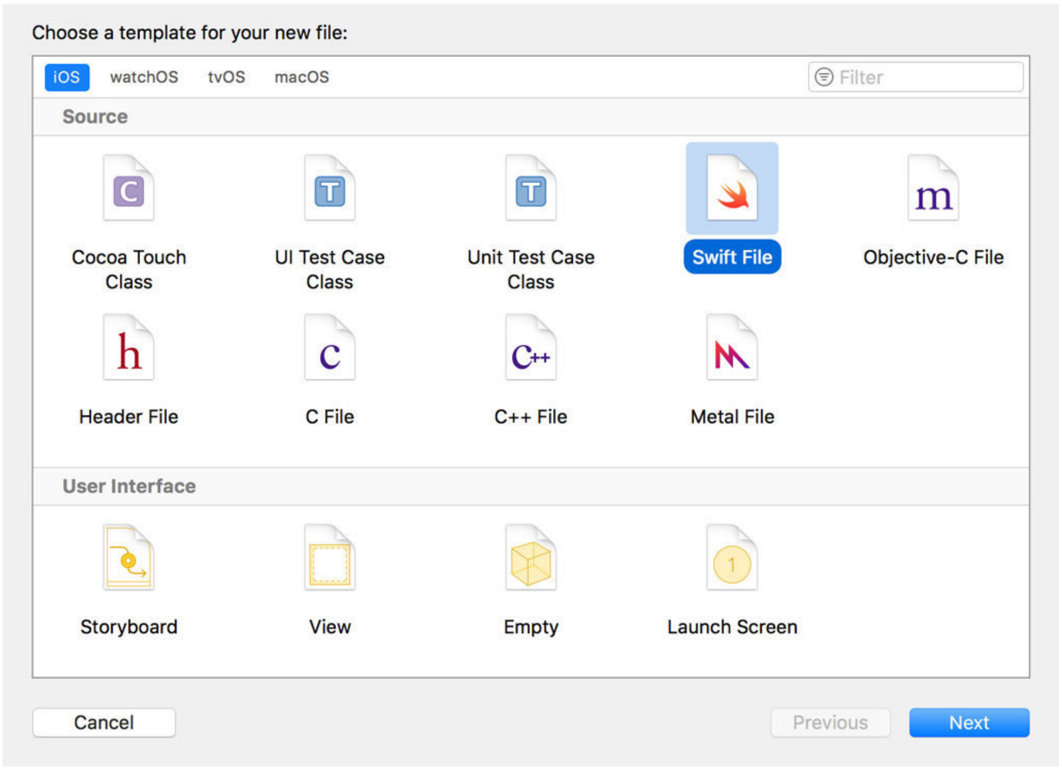
Localizations

Language	Resources
English — Development Language	2 Files Localized

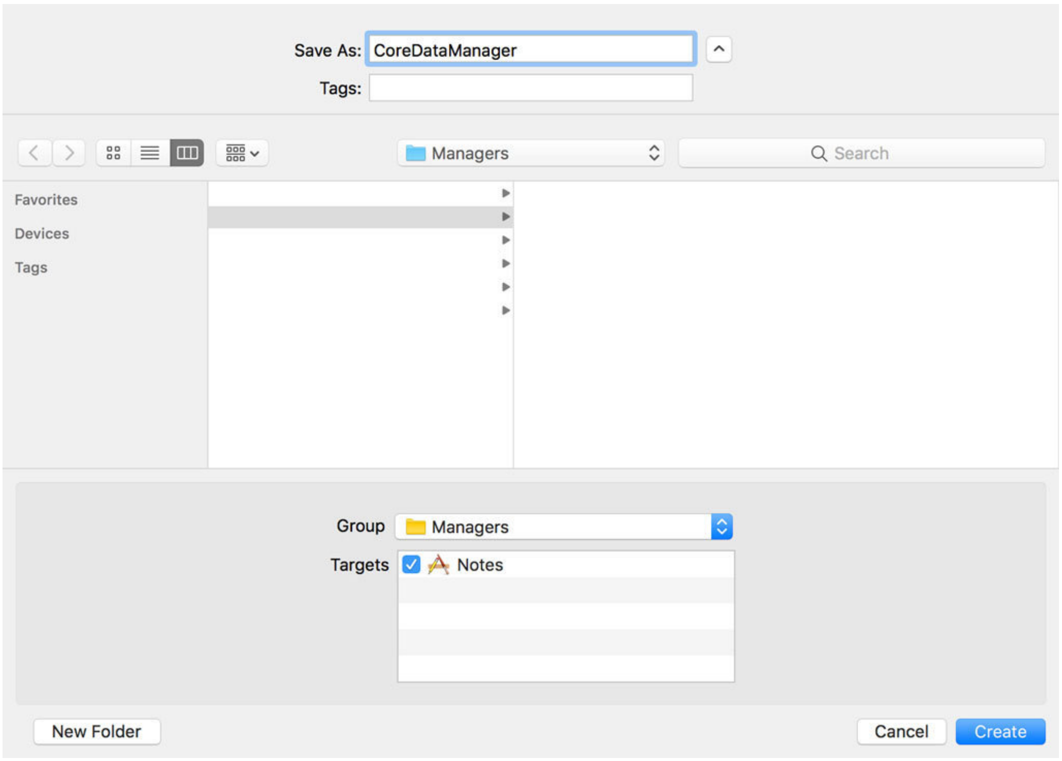
Use Base Internationalization

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



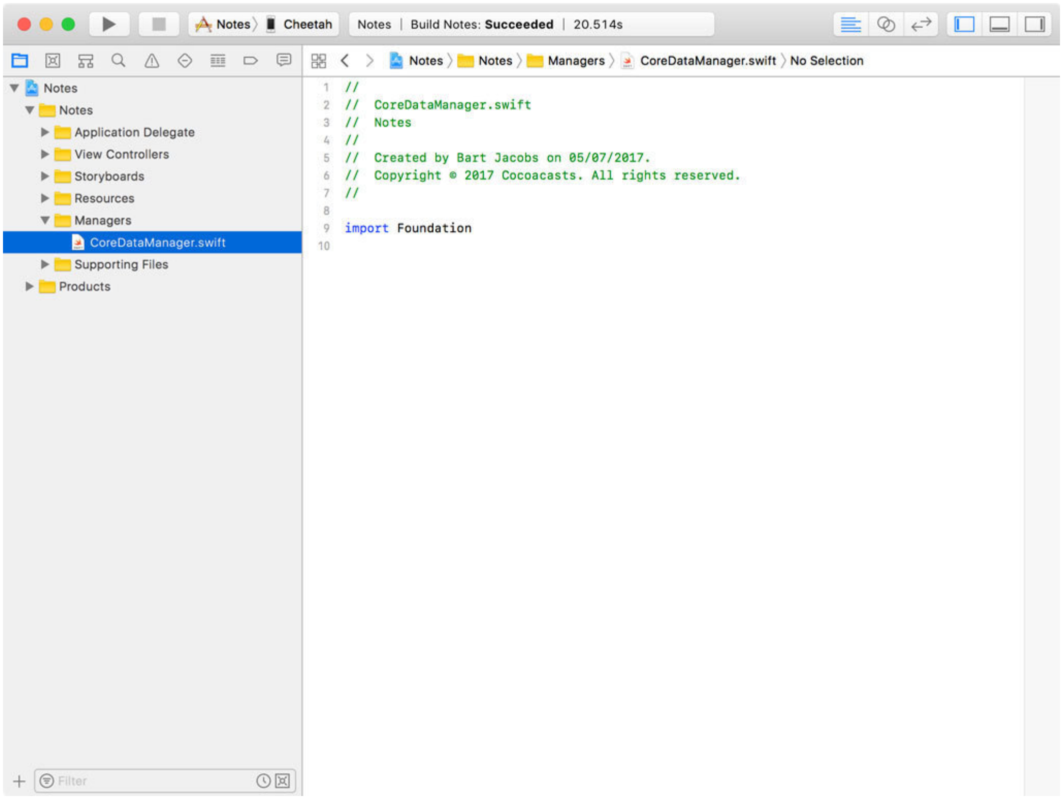
Save As: CoreDataManager

Tags:

Group: Managers

Targets: ✓ Notes

Creating CoreDataManager.swift



```swift
//
//  CoreDataManager.swift
//  Notes
//
//  Created by Bart Jacobs on 05/07/2017.
//  Copyright © 2017 Cocoacasts. All rights reserved.
//

import Foundation
```

Creating CoreDataManager.swift

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

```swift
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)

} 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.

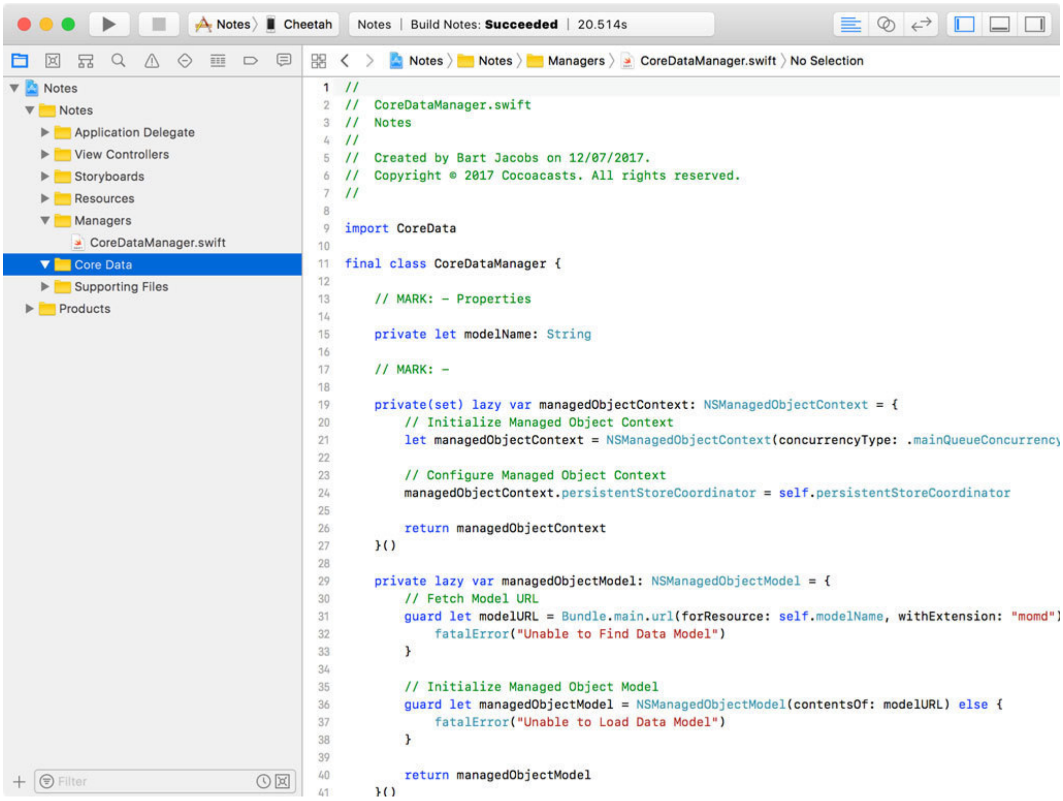
In the `catch` clause, we print the error to the console if the operation failed. We return the persistent store coordinator from the closure.

```
private lazy var persistentStoreCoordinator: NSPersistentStoreCoordi\
nator = {
    // Initialize Persistent Store Coordinator
    let persistentStoreCoordinator = NSPersistentStoreCoordinator(ma\
nagedObjectModel: self.managedObjectModel)

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

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

    // URL Persistent Store
    let persistentStoreURL = documentsDirectoryURL.appendingPathComp\
onent(storeName)

    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
}()
```

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

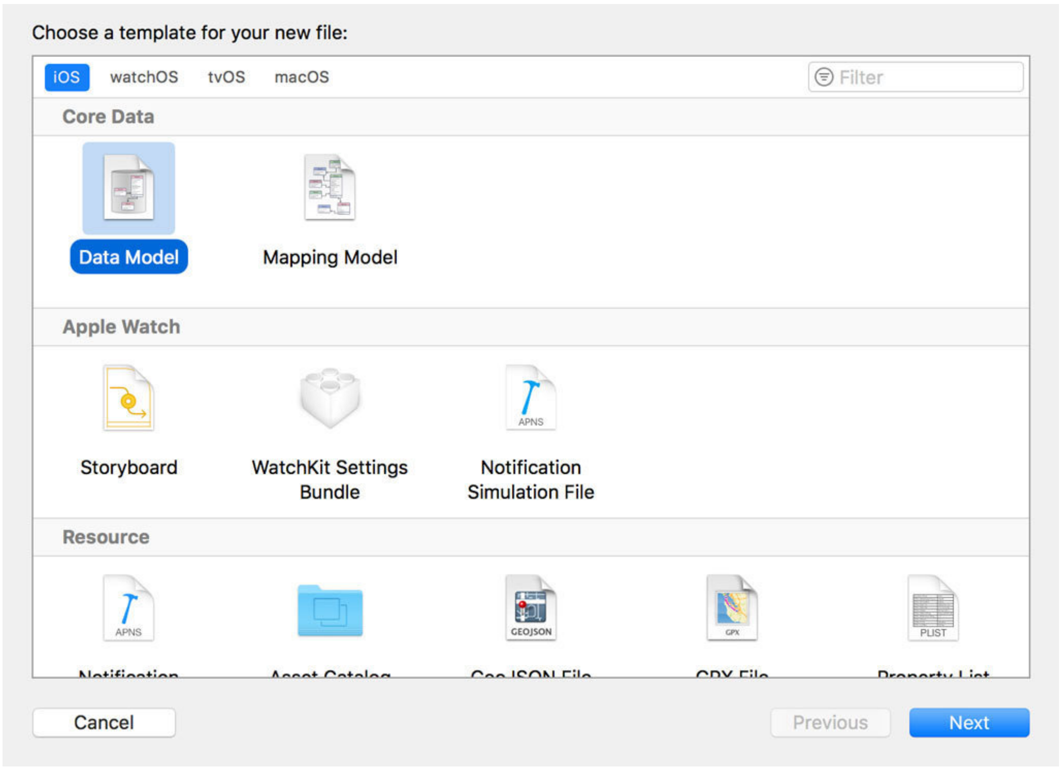
Before we can take the Core Data manager for a spin, we need to add a data model to the project. Create a new group for the data model and name it **Core Data**.



```swift
//
//  CoreDataManager.swift
//  Notes
//
//  Created by Bart Jacobs on 12/07/2017.
//  Copyright © 2017 Cocoacasts. All rights reserved.
//

import CoreData

final class CoreDataManager {

    // MARK: - Properties

    private let modelName: String

    // MARK: -

    private(set) lazy var managedObjectContext: NSManagedObjectContext = {
        // Initialize Managed Object Context
        let managedObjectContext = NSManagedObjectContext(concurrencyType: .mainQueueConcurrenc

        // Configure Managed Object Context
        managedObjectContext.persistentStoreCoordinator = self.persistentStoreCoordinator

        return managedObjectContext
    }()

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

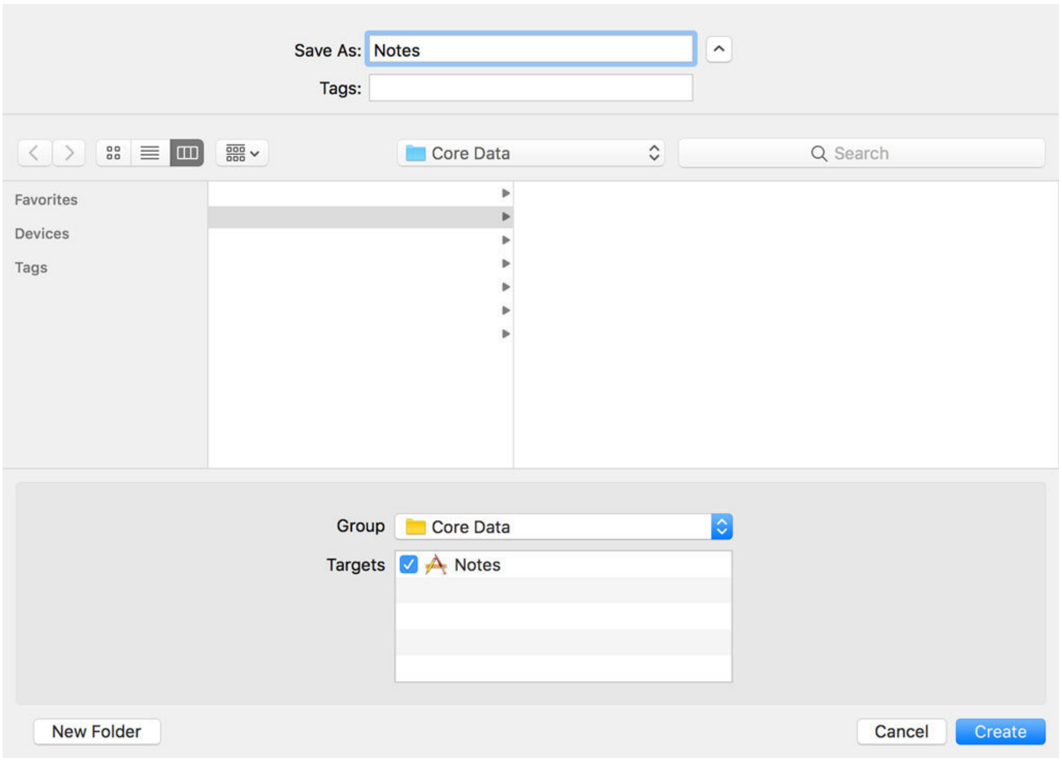
Creating the Core Data Group

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**.

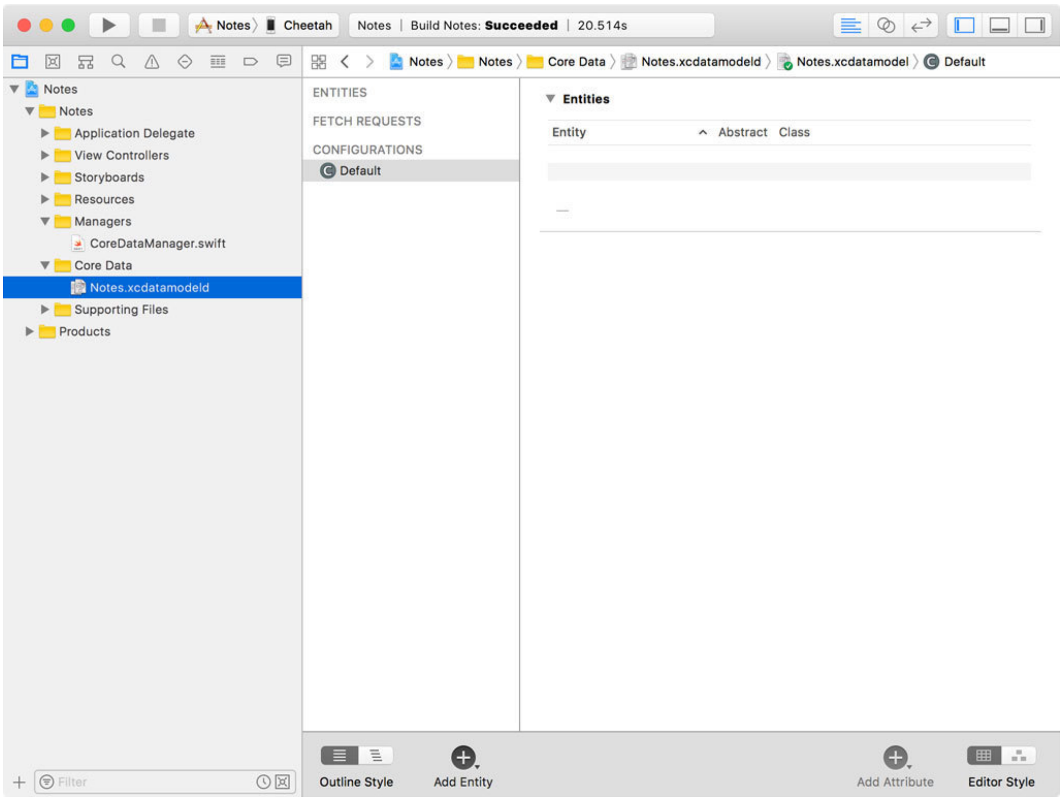

Save As: Notes

Tags:

Group: Core Data

Targets: ✓ Notes

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.

```
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 {
        let coreDataManager = CoreDataManager(modelName: "Notes")
        print(coreDataManager.managedObjectContext)
        return true
    }

}
```

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

```
<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.