

Project 38. GitHub Commits

Overview

Brief: Get on board with Core Data and learn to read, write and query objects using Apple's object graph and persistence framework.

Learn: Core Data, NSFetchRequest, NSSortDescriptor, NSFetchedResultsController, NSDateFormatter.

- Setting up
- Designing a Core Data model
- Adding Core Data to our project: NSPersistentStoreCoordinator
- Creating an NSManagedObject subclass with Xcode
- Loading Core Data objects using NSFetchRequest and NSSortDescriptor
- How to make a Core Data attribute unique using constraints
- Examples of using NSPredicate to filter NSFetchRequest
- Adding Core Data entity relationships: lightweight vs heavyweight migration
- How to delete a Core Data object
- Optimizing Core Data Performance using NSFetchedResultsController
- Wrap up

Setting up

In this project you'll learn how to use Core Data while building an app that fetches GitHub commit data for the Swift project. Core Data is Apple's object graph and persistence framework, which is a fancy way of saying it reads,

writes and queries collections of related objects while also being able to save them to disk.

Core Data is undoubtedly useful, which is why about 500,000 apps in the App Store build on top of it. But it's also rather complicated to learn, which is why I left it so late in this tutorial series – despite my repeated attempts to simplify the topic, this tutorial is still going to be the hardest one in the whole of Hacking with Swift, and indeed I might have missed it out altogether were it not by far the most requested topic from readers!

So, strap in, because you're going to learn a lot: we'll be covering Core Data, which will encompass `NSFetchRequest`, `NSManagedObject`, `NSPredicate`, `NSSortDescriptor`, and `NSFetchedResultsController`. We'll also touch on `NSDateFormatter` for the first time, which is Apple's way of converting `NSDate` objects into human-readable formats.

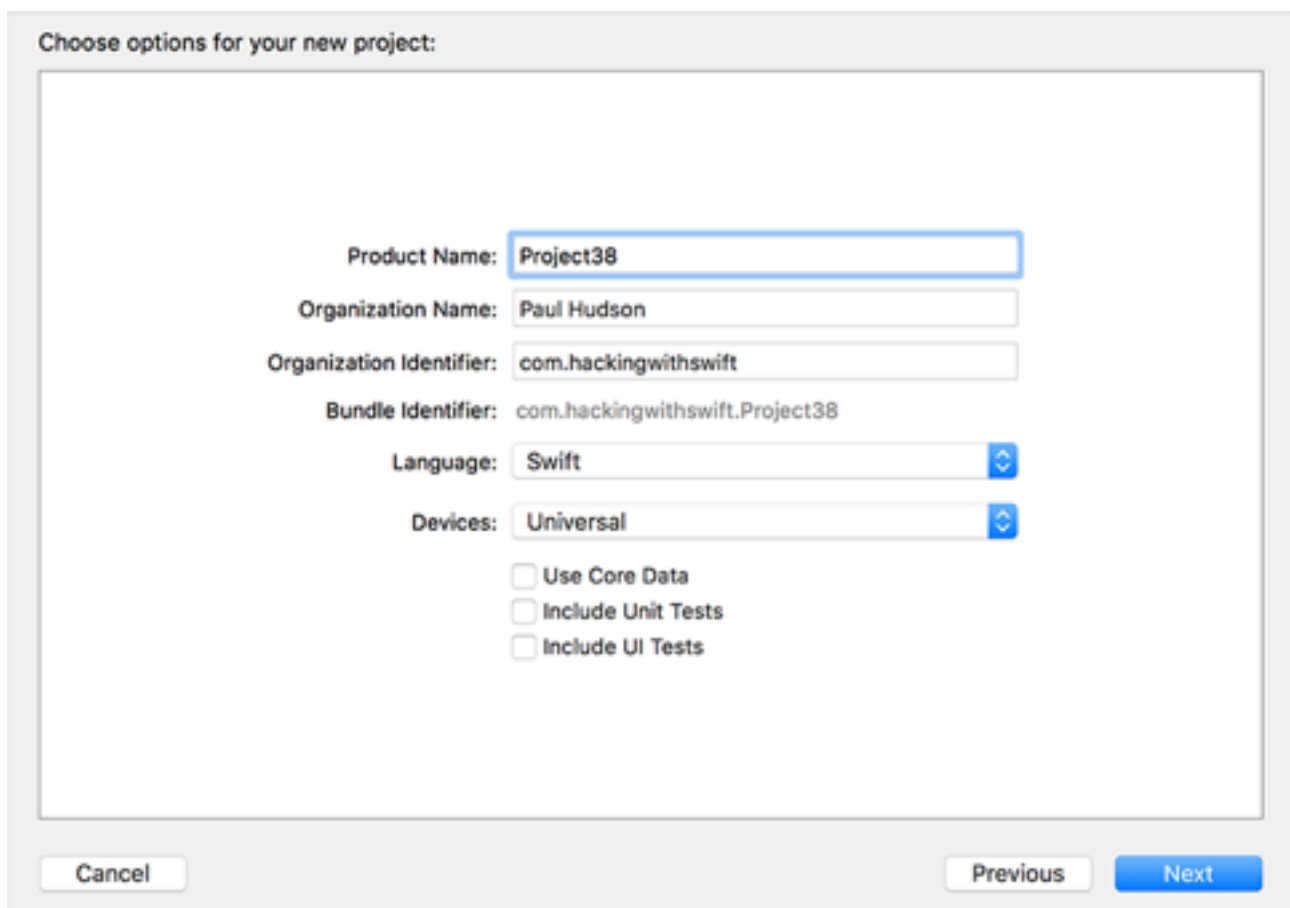
As always, I like to teach new topics while giving you a real-world project to work with, and in our case we're going to be using the GitHub API to fetch information about Apple's open-source Swift project. The GitHub API is simple, fast, and outputs JSON, but most importantly it's public. Be warned, though: you get to make only 60 requests an hour without an API key, so while you're testing your app make sure you don't refresh too often!

If you weren't sure, a “Git commit” is a set of changes a developer made to source code that is stored in a source control repository. For example, if you spot a bug in the Swift compiler and contribute your changes, those changes will form one commit. And before I get emails from random internet pedants, yes I know things are more complicated than that, but it's more than enough of an explanation for the purposes of this tutorial.

Before we start, it's important I reiterate that Core Data can be a bit overwhelming at first. It has a lot of unique terminology, and it needs a lot of boilerplate just to get you up and running. If you find yourself struggling to understand it all, that's perfectly normal – it's not you, it's just Core Data.

Over the next four chapters, we will implement the four pieces of Core Data boilerplate. We're going to use Xcode's built-in Master–Detail Application template, but we're not going to have it generate Core Data code for us – that particular template is confusing enough as it is, and the Core Data variety is practically impenetrable.

So, please go ahead and create a new Master–Detail Application project named Project38. Select Swift for your language, select Universal for your device type, then make sure you uncheck Core Data otherwise the rest of this tutorial will be very confusing indeed.



The image shows the 'Choose options for your new project' dialog box in Xcode. The dialog has a title bar that says 'Choose options for your new project:'. Inside the dialog, there are several input fields and checkboxes. The 'Product Name' field is highlighted with a blue border and contains the text 'Project38'. The 'Organization Name' field contains 'Paul Hudson'. The 'Organization Identifier' field contains 'com.hackingwithswift'. The 'Bundle Identifier' field contains 'com.hackingwithswift.Project38'. The 'Language' dropdown menu is set to 'Swift'. The 'Devices' dropdown menu is set to 'Universal'. Below these fields, there are three unchecked checkboxes: 'Use Core Data', 'Include Unit Tests', and 'Include UI Tests'. At the bottom of the dialog, there are three buttons: 'Cancel', 'Previous', and 'Next'.

Choose options for your new project:

Product Name: Project38

Organization Name: Paul Hudson

Organization Identifier: com.hackingwithswift

Bundle Identifier: com.hackingwithswift.Project38

Language: Swift

Devices: Universal

☐ Use Core Data

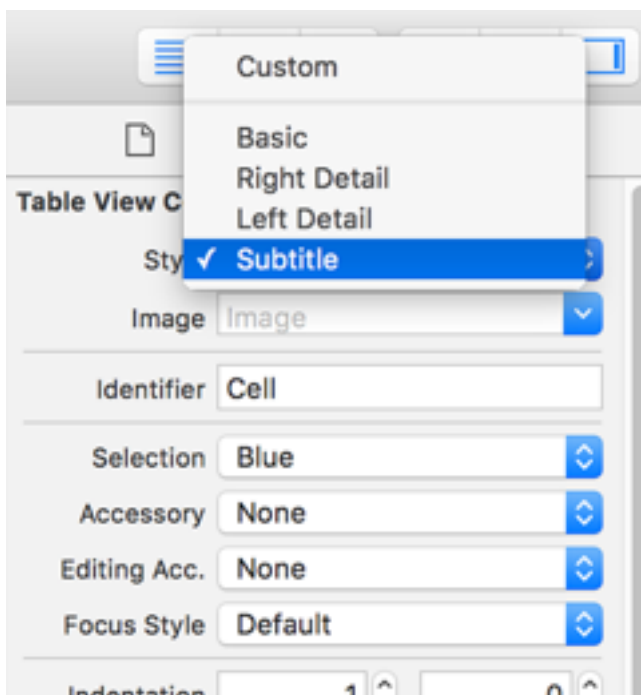
☐ Include Unit Tests

☐ Include UI Tests

Cancel Previous Next

We need to parse the JSON coming from GitHub's API, and the easiest way to do that is with SwiftyJSON. Look it in folder Content.

There are just two more things to do before our project is set up. First, open Main.storyboard in Interface Builder and find the table view controller on the top right of the storyboard. Select its only prototype cell, then open the attributes inspector and change Style from Basic to Subtitle.



Now scroll directly down to find the detail view controller with its single label. Select that label, and change its number of lines property to be 0 - it's not perfect, but it's enough to show everything is working.

Finally, open MasterViewController.swift for editing. Please find and delete the entire insertNewObject() method. Now scroll up to the viewDidLoad() method and delete these two lines of code:

```
let addButton = UIBarButtonItem(barButtonSystemItem: .Add,  
target: self, action: #selector(insertNewObject(_:))  
self.navigationItem.rightBarButtonItem = addButton
```

That's it: the project is cleaned up and ready for Core Data. Remember, there are four steps to implementing Core Data in your app, so let's start with the very first step: designing a Core Data model.

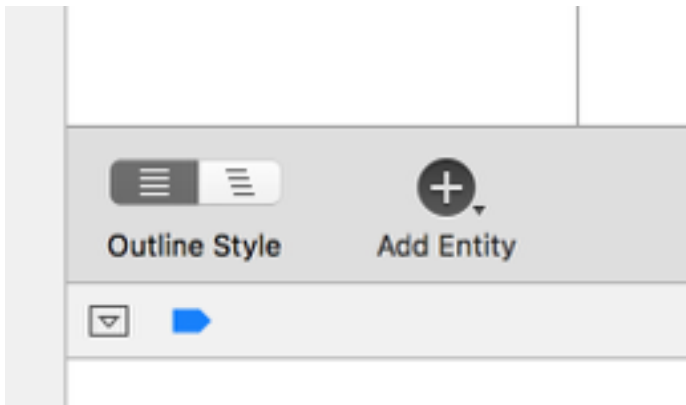
Designing a Core Data model

A data model is a description of the data you want Core Data to store, and is a bit like creating a class in Swift: you define entities (like classes) and give them attributes (like properties). But Core Data takes it a step further by allowing you to describe how its entities relate to other entities, as well as adding rules for validation and uniqueness.

We're going to create a data model for our app that will store a list of all the GitHub commits for the Swift library. Take a look at the raw GitHub JSON now by loading this URL in a web browser: https://api.github.com/repos/apple/swift/commits?per_page=100. You'll see that each commit has a “sha” identifier, committer details, a message describing what changed, and a lot more. In our initial data model, we're going to track the “date”, “message”, “sha”, and “url” fields, but you're welcome to add more if you want to.

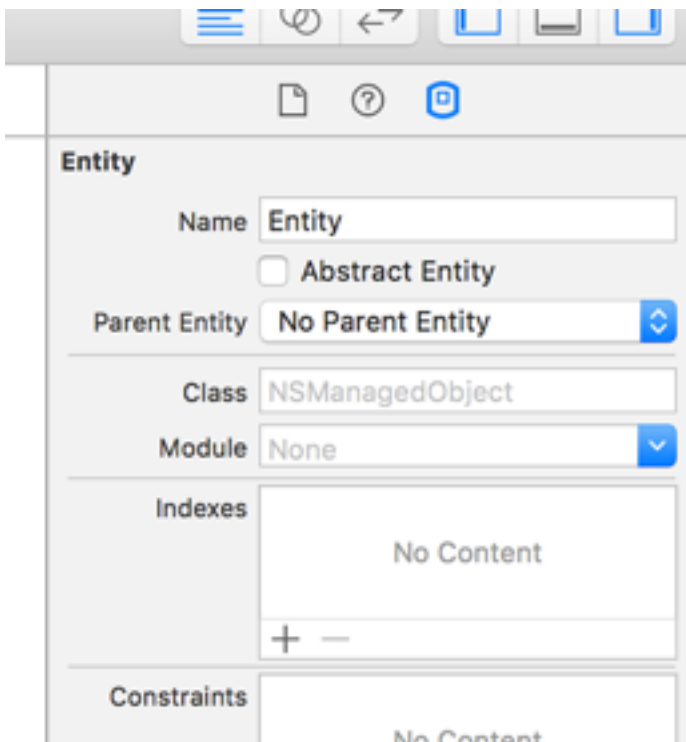
To create a data model, choose File > New > File and select iOS > Core Data > Data Model. Name it Project38, then make sure the “Group” option near the bottom of the screen has a yellow folder to it rather than a blue project icon.

This will create a new file called Project38.xcdatamodeld, and when you select that you'll see a new editing display: the Data Model Editor. At the bottom you'll see a button with the title “Add Entity”: please click that now.



A Core Data “entity” is like a Swift class in that it is just a description of what an object is going to look like. By default, new entities are called “Entity”, but you can change that in the Data Model inspector in the right-hand pane of Xcode – press Alt +

Cmd + 3 if it's not already visible. With your new entity selected, you should see a field named “Name”, so please change “Entity” to be “Commit”.

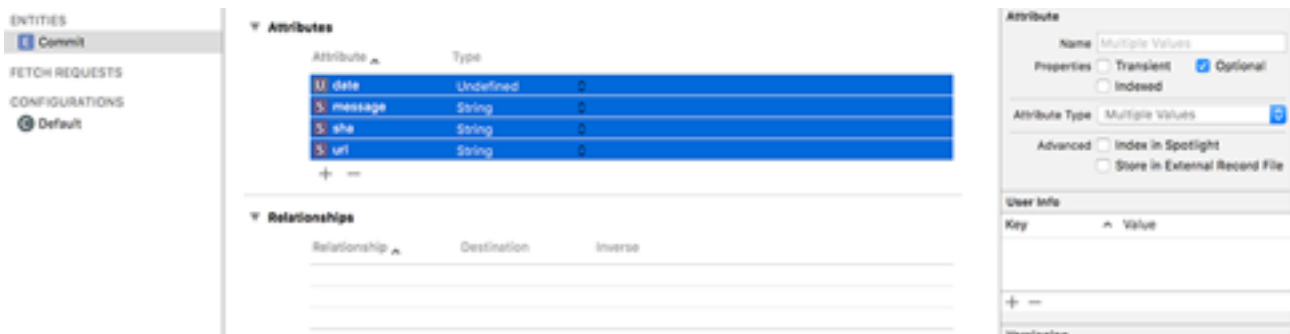


To the right of the Add Entity button is another button, Add Attribute. Click that four times now to add four attributes, then name them “date”, “message”, “sha” and “url”. These attributes are just like properties on a Swift class, including the need to have a data type. You'll see they each have “Undefined” for their type right now, but that's easily changed: set them all to have the String data

type, except for “date”, which should be Date.

The final change we're going to make is to mark each of these four property as non-optional. Click “date” then hold down Shift and click “url” to select all four attributes, then look in the Data Model inspector for the Optional checkbox and deselect it. Note: the Data Model inspector can

be a bit buggy sometimes – if you find it's completely blank, you might need to try selecting one of the other files in your project and/or deselecting then re-selecting your entity to make things work.



Now, you might be forgiven for thinking, “at last! All that time spent mastering Swift optionals is paying off – I know what this checkbox does!”. But I have some bad news for you. Or, more specifically, Core Data has some bad news for you: this `Optional` checkbox has nothing at all to do with Swift optionals, it just determines whether the objects that Core Data stores are required to have a value or not.

That's the first step of Core Data completed: the app now knows what kind of data we want to store. We'll be coming back to add to our model later, but first it's time for step two: adding the base Core Data functionality to our app so we can load the model we just defined and save any changes we make.

Adding Core Data to our project: `NSPersistentStoreCoordinator`

A Core Data model defines what your data should look like, but it doesn't actually store it anywhere. To make our app work, we need to create a persistent store where Core Data can read and write its data, then create what's called a “managed object context” where those objects can be manipulated by us in memory.

In this second step we're going to write code to load the `model` we just defined, load a `persistent store` where saved objects can be stored, and also create a `managed object context` where our objects will live while they are active. Any changes we make to objects won't be saved until we explicitly request it, and it's significantly faster to manipulate objects inside your `managed object context` as much as you need to before saving rather than saving after every change.

When data is saved, it's nearly always written out to an `SQLite` database. There are other options, but take my word for it: almost everyone uses `SQLite`. `SQLite` is a very small, very fast, and very portable database engine, and what `Core Data` does is provide a wrapper around it: when you read, write and query a `managed object context`, `Core Data` translates that into `Structured Query Language (SQL)` for `SQLite` to parse.

If you were wondering, `SQL` is pronounced `Ess Cue Ell`, but many people pronounce it “`sequel`”. The pronunciation of `SQLite` is more varied, but when I met its author I asked him how he pronounces it, so I feel fairly safe that the definitive answer is this: you pronounce `SQLite` as `Ess-Cue-Ell-ite`, as if it were a mineral like `Kryponite` or `Carbonite` depending on your preferred movie.

To get started, open `MasterViewController.swift` and add an import for `Core Data`:

```
import CoreData
```

To set up the basic `Core Data` system, we're going to create a new method called `startCoreData()` that will do the following:

1. Load our data model we just created from the application bundle and create a `NSManagedObjectModel` object from it.
2. Create an `NSPersistentStoreCoordinator` object, which is responsible for reading from and writing to disk.
3. Set up an `NSURL` pointing to the database on disk where our actual saved objects live. This will be an `SQLite` database named `Project38.sqlite`.
4. Load that database into the `NSPersistentStoreCoordinator` so it knows where we want it to save. If it doesn't exist, it will be created automatically
5. Create an `NSManagedObjectContext` and point it at the persistent store coordinator.

Below is all that turned into Swift, along with comments marking the relevant numbers above - please add this new method to the `MasterViewController` class:

```
func startCoreData() {  
    // 1  
    let modelURL =  
    NSBundle.mainBundle().URLForResource("Project38",  
    withExtension: "momd")!  
    let managedObjectModel =  
    NSManagedObjectModel(contentsOfURL: modelURL)!  
  
    // 2  
    let coordinator =  
    NSPersistentStoreCoordinator(managedObjectModel:  
    managedObjectModel)
```

```

        // 3
        let url =
getDocumentsDirectory().URLByAppendingPathComponent("Project3
8.sqlite")

        do {
            // 4
            try
coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
configuration: nil, URL: url, options: nil)

            // 5
            managedObjectContext =
NSManagedObjectContext(concurrencyType: .MainQueueConcurrency
Type)
            managedObjectContext.persistentStoreCoordinator =
coordinator
        } catch {
            print("Failed to initialize the application's saved
data")
            return
        }
    }

// this is our usual helper function to find the user's
documents directory
func getDocumentsDirectory() -> NSURL {
    let urls =
NSFileManager.defaultManager().URLsForDirectory(.DocumentDire
ctory, inDomains: .UserDomainMask)
    return urls[0]
}

```

```
}
```

Note that I've wrapped the call to `addPersistentStoreWithType()` inside a `do/catch` block because it throws an exception when it fails. My project prints a message to the Xcode console, but in production you should of course show some meaningful error to your user.

To make that code work, you'll need to add a property for `managedObjectContext`, which is where we store the context once it is loaded. Add this property to the class now:

```
var managedObjectContext: NSManagedObjectContext!
```

So: the `model` is in your application bundle because you define it while creating your app, the `persistent store` is in your application's documents directory because it gets modified as you use it, and the `managed object context` lives in memory, and is where you manipulate your objects before saving them.

We need to call our new method as soon as our app launches, so add this just before the end of `viewDidLoad()`:

```
startCoreData()
```

The last thing to do is add a way to save the `managed object context` to the `persistent store`, i.e. to make sure the changes we've made in memory are written to disk so they can be used later. To make this work we'll use the `hasChanges` property of the `managed object context` to determine whether a save is needed, and its `save()` method to do the actual writing.

Add this new method just after the previous two:

```
func saveContext() {
    if managedObjectContext.hasChanges {
        do {
            try managedObjectContext.save()
        } catch {
            print("An error occurred while saving: \(error)")
        }
    }
}
```

We'll be calling that whenever we've made changes that should be saved to disk.

At this point, our app has a working `data model` as well as code to load that `model` and prepare a `persistent store` for reading and writing. That means step two is done and we're on to step three: creating objects inside `Core Data` and fetching data from `GitHub`.

Creating an `NSManagedObject` subclass with Xcode

In our app, `Core Data` is responsible for reading data from a `persistent store` (the `SQLite` database) and making it available for us to use as objects.

After changing those objects, we can save them back to the `persistent store`, which is when `Core Data` converts them back from objects to database records.

All this is done using a special data type called `NSManagedObject`. This is a `Core Data` subclass of `NSObject` that uses a unique keyword, `@NSManaged`,

to provide lots of functionality for you. For example, you already saw the `hasChanges` property of a managed object context – that automatically gets set to true when you make changes to your objects, because Core Data tracks when you change properties that are marked `@NSManaged`.

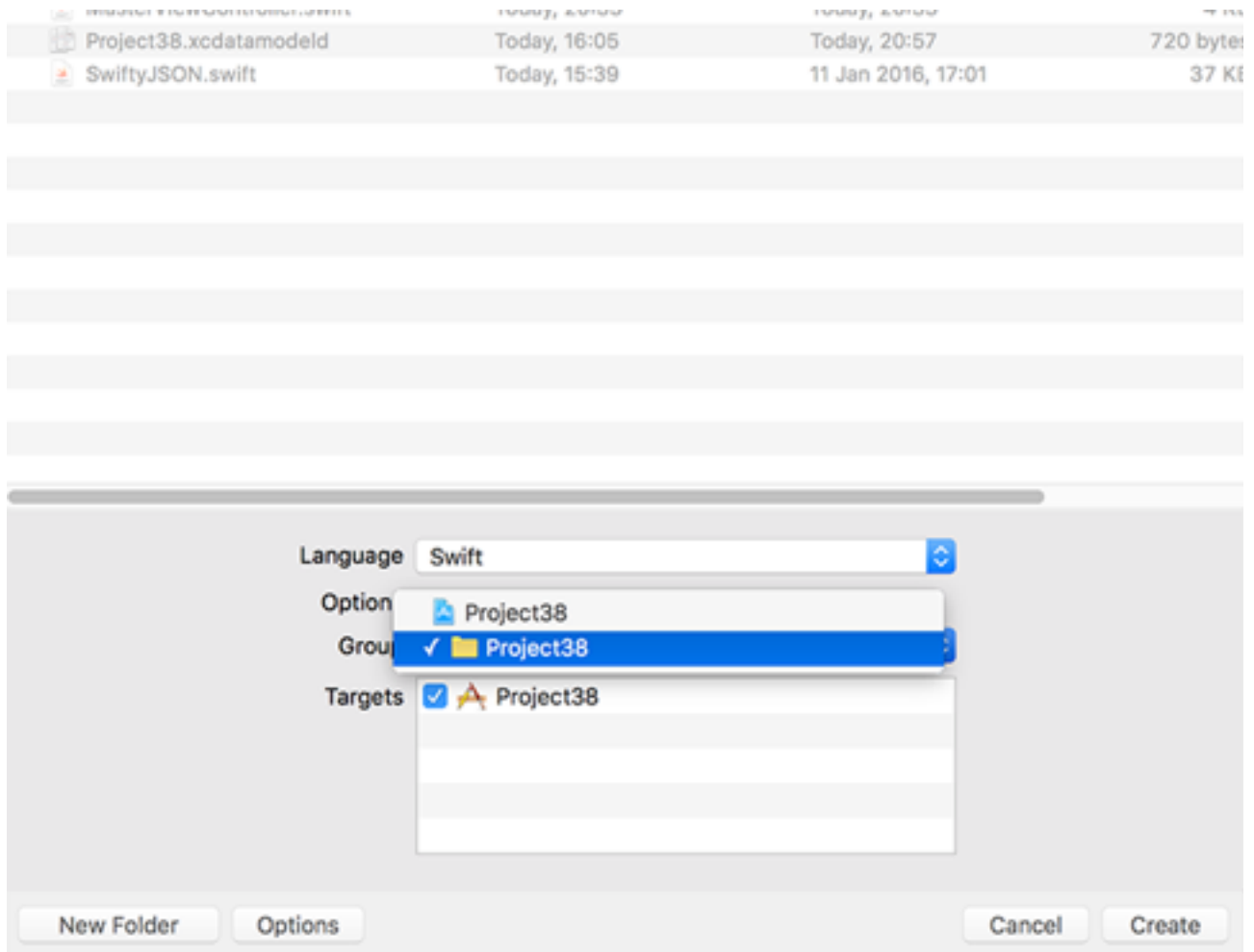
Behind the scenes, `@NSManaged` effectively means “extra code will automatically be provided when the program runs”. It's a bit like functionality injection: when you say “this property is `@NSManaged`” then Core Data will add getters and setters to it when the app runs so that it handles things like change tracking.

If this sounds complicated, relax: Xcode can do quite a bit of work for us. It's not perfect, as you'll see shortly, but it's certainly a head start. So, it's time for step three: creating objects in Core Data so that we can fetch and store data from GitHub.

Select the data model (`Project38.xcdatamodeld`) and choose `Editor > Create NSManagedObject Subclass` from the menu. Make sure your data model is selected then click `Next`. Make sure the `Commit` entity is checked then click `Next` again.



Finally, look next to `Group` and make sure you see a yellow folder next to “`Project 38`” rather than a blue project icon, and click `Create`.



When this process completes, two new files are created: `Commit.swift` and `Commit+CoreDataProperties.swift`. If you examine them you'll see the first one is empty, whereas the second one looks something like this:

```
import Foundation
import CoreData

extension Commit {
    @NSManaged var date: NSDate?
    @NSManaged var message: String?
    @NSManaged var sha: String?
    @NSManaged var url: String?
}
```

First, there's that `@NSManaged` keyword I mentioned to you. Second, notice that the code says `extension Commit` rather than `class Commit`, which is Xcode being clever: `Commit.swift` is an empty class that you can fill with your own functionality, and `Commit+CoreDataProperties.swift` is an extension to that class where Core Data writes its properties. This means if you ever add attributes to the `Commit` entity and regenerate the `NSObject` subclass, Xcode will overwrite only `Commit+CoreDataProperties.swift`, leaving your own changes in `Commit.swift` untouched.

Like I said, this Core Data code generation is a head start, but not perfect: how come all those properties are marked as optional, i.e. `NSDate?` rather than `NSDate`, when we explicitly told Core Data that they were not optional when making the model? Well, remember that Swift's optionals and Core Data's optionals aren't the same, so Xcode's generated code isn't perfect.

Working with optionals when they aren't needed adds an extra layer of annoyance, so I want you go ahead and remove all the question marks from `Commit+CoreDataProperties.swift`. When you're finished, it should look like this:

```
import Foundation
import CoreData

extension Commit {
    @NSManaged var date: NSDate
    @NSManaged var message: String
    @NSManaged var sha: String
    @NSManaged var url: String
}
```

Warning: if you recreate the subclass using the `Create NSManaged Subclass` menu option, these changes will be lost and you will need to remove the optionality again. We'll be doing exactly this later on.

Now that we have `Core Data` objects defined, we can start to write our very first useful `Core Data` code: we can fetch some data from `GitHub` and convert it into our `Commit` objects. To make things easier to follow, I want to split this up into smaller steps: fetching the `JSON`, and converting the `JSON` into `Core Data` objects.

First, fetching the `JSON`. This needs to be a background operation because network requests are slow and we don't want the user interface to freeze up when data is loading. This operation needs to go to the `GitHub` URL, `https://api.github.com/repos/apple/swift/commits?per_page=100` and convert the result into a `SwiftyJSON` object ready for conversion.

To push all this into the background, we're going to use `performSelectorInBackground()` to call `fetchCommits()` – a method we haven't written yet. Put this just before the end of `viewDidLoad()`:

```
performSelectorInBackground(#selector(fetchCommits),
withObject: nil)
```

What the new `fetchCommits()` method will do is very similar to what we did back in `Project 7`: download the `URL` into an `NSData` object then pass it to `SwiftyJSON` to convert into an array of objects. In `Project 10` we extended this to use `dispatch_async()` so that once the `JSON` was ready to be used we did the important work on the main thread.

We're not going to process the JSON just yet, but we can do everything else: download the data, create a `SwiftJSON` object from it, then go back to the main thread to loop over the array of GitHub commits and save the managed object context when we're done. To make things easier to debug, I've added a `print()` statement so you can see how many commits were received from GitHub each time.

Here's our first draft of the `fetchCommits()` method:

```
func fetchCommits() {
    if let data = NSData(contentsOfURL: NSURL(string:
"https://api.github.com/repos/apple/swift/commits?
per_page=100"))! {
        let jsonCommits = JSON(data: data)
        let jsonCommitArray = jsonCommits.arrayValue

        print("Received \(jsonCommitArray.count) new
commits.")

        dispatch_async(dispatch_get_main_queue()) { [unowned
self] in
            for jsonCommit in jsonCommitArray {
                // more code to go here!
            }

            self.saveContext()
        }
    }
}
```

There's nothing too surprising there – in fact right now it won't even do anything, because `saveContext()` will detect no Core Data changes have happened, so the `save()` call won't happen.

The second of our smaller steps is to replace `// more code to go here!` with, well, actual code. Here's the revised version, with a few extra lines either side so you can see where it should go:

```
dispatch_async(dispatch_get_main_queue()) { [unowned self] in
    for jsonCommit in jsonCommitArray {
        // the following three lines are new
        if let commit =
NSEntityDescription.insertNewObjectForEntityForName("Commit",
inManagedObjectContext: self.managedObjectContext) as? Commit
{
            self.configureCommit(commit, usingJSON:
jsonCommit)
        }
    }

    self.saveContext()
}
```

So, there are three new lines of code, of which one is just a closing brace by itself. Of course, it's so short only because I've cheated a bit by calling another method that we haven't written yet, `configureCommit()`, so to make your code build add this for now:

```
func configureCommit(commit: Commit, usingJSON json: JSON) {
}
```

Now let's take a look at the two new lines of code above. First is `NSEntityDescription.insertNewObjectForEntityForName()`, which reads the entity description for `Commit` from the managed object context. This method creates a new object with the type `NSManagedObject`, so we use `if/let` and `as?` `Commit` to safely downcast it to be a `Commit` object. Putting all that together, the code reads “try to create a new `Commit` object in our managed object context, and give it back to me if things went OK”.

If we get a valid `Commit` object back, we pass it onto the `configureCommit()` method, along with the `JSON` data for the matching commit. That `Commit` object is our `NSManagedObject` subclass, so it has all sorts of magic behind the scenes, but to our `Swift` code is just a normal object with properties we can read and write. This would make the `configureCommit()` method straightforward if it were not for dates.

Yes, dates. Not the sweet fruity kind, but the `NSDate` kind. Make sure you have the `GitHub API URL` open in a web browser window so you can see exactly what it returns, and you'll notice that dates are sent back like “2016-01-26T19:46:18Z”. That format is known as `ISO-8601` format, we need to parse that into an `NSDate` in order to put it inside our `Commit` object.

To convert “2016-01-26T19:46:18Z” into an `NSDate` we're going to use a new class called `NSDateFormatter`. This is designed to convert text to `NSDate` and vice versa – all you need to do is describe the textual date format using special syntax. For `ISO-8601` dates, that syntax is “yyyy-MM-dd'T'HH:mm:ss'Z'”: a four-digit year, a two-digit month, a two-digit day, the letter `T`, then hours, minutes, seconds, and the letter `Z`.

We're going to use this date format again later in the tutorial, so add this property to the `MasterViewController` class:

```
let dateFormatISO8601 = "yyyy-MM-dd'T'HH:mm:ss'Z'"
```

Before I show you the new `configureCommit()` method, there's one more thing you need to know: getting an `NSDate` out of a string might fail, for example if the date format doesn't match the one we set. In this case, we'll get `nil` back, which isn't much good for our app, so I'm going to use the `nil` coalescing operator to use a new `NSDate` instance if the date failed to parse.

Here's the new `configureCommit()` method:

```
func configureCommit(commit: Commit, usingJSON json: JSON) {
    commit.sha = json["sha"].stringValue
    commit.message = json["commit"]["message"].stringValue
    commit.url = json["html_url"].stringValue

    let formatter = NSDateFormatter()
    formatter.timeZone = NSTimeZone(name: "UTC")
    formatter.dateFormat = dateFormatISO8601
    commit.date = formatter.dateFromString(json["commit"]
["committer"]["date"].stringValue) ?? NSDate()
}
```

I love how easy `SwiftyJSON` makes JSON parsing! If you've forgotten, it automatically ensures a safe value gets returned even if the data is missing or broken. For example, `json["commit"]["message"].stringValue` will either return the commit message as a string or an empty string, regardless of what the JSON contains. So if `"commit"` or `"message"` don't exist, or if they do

exist but actually contains an integer for some reason, we'll get back an empty string – it makes JSON parsing extremely safe while being easy to read and write.

That completes step three of our Core Data code: we now create lots of objects when we download data from GitHub, and the finishing collection gets saved back to SQLite. That just leaves one final step before we have the full complement of fundamental Core Data code: we need to be able to load and use all those Commit objects we just saved!

Loading Core Data objects using NSFetchRequest and NSSortDescriptor

This is where Core Data starts to become interesting and perhaps – gasp! – even fun. Yes, I know it's taken quite a lot of work to get this far, but I did warn you, remember?

Step four is where we finally get to put to use all three previous steps by showing data to users. After the huge amount of work you've put in, particularly in the previous step, I'm sure you'll be grateful to see everything pay off at last!

Xcode's Master–Detail Application template gave us this array to use in the table view:

```
var objects = [AnyObject]()
```

In our project, we know we're using Commit objects rather than AnyObject, so please change that line to this:

```
var objects = [Commit]()
```

As soon as you do that, your code will break in two places because the template expects `NSDate` objects to be in that array. Don't worry, this is easily fixed. Start by going to `prepareForSegue()` and finding this line:

```
let object = objects[indexPath.row] as! NSDate
```

To fix that line, just remove the `as! NSDate` part, like this:

```
let object = objects[indexPath.row]
```

Now scroll down to find `cellForRowAtIndexPath`, and in particular these two lines:

```
let object = objects[indexPath.row] as! NSDate
cell.textLabel!.text = object.description
```

Change them to this:

```
let object = objects[indexPath.row]
cell.textLabel!.text = object.message
cell.detailTextLabel!.text = object.date.description
```

That will show a snippet of each commit message in the table view cell titles, and a basic formatted date in the subtitles.

With that change made, we need to write one new method in order to make our entire app spring into life. But before we jump into the code, you need to learn about one of the most important classes in `Core Data`: `NSFetchRequest`. This is the class that performs a query on your data, and returns a list of objects that match.

We're going to use `NSFetchRequest` in a really basic form for now, then add more functionality later. In this first version, we're going to ask it to give us an array of all `Commit` objects that we have created, sorted by date descending so that the newest commits come first.

The way fetch requests work is very simple: you tell it what entity you want to query, give it an array of data describing how you want the sorting to happen, then pass it to `executeFetchRequest()` on your managed object context. If the fetch request worked then you'll get back an array of objects matching the query; if not, an exception will be thrown that you need to catch.

The sorting is done through a special data type called `NSSortDescriptor`, which is a trivial wrapper around the name of what you want to sort (in our case "date"), then a boolean setting whether the sort should be ascending (oldest first for dates) or descending (newest first). You pass an array of these, so you can say "sort by date descending, then by message ascending", for example.

OK, time for some code, and I hope you'll be pleasantly surprised by how easy it is:

```
func loadSavedData() {
    let fetch = NSFetchRequest(entityName: "Commit")
    let sort = NSSortDescriptor(key: "date", ascending:
false)
    fetch.sortDescriptors = [sort]

    do {
        if let commits = try
managedObjectContext.executeFetchRequest(fetch) as? [Commit]
        {
            print("Got \(commits.count) commits")
        }
    }
}
```

```

        objects = commits
        tableView.reloadData()
    }
} catch {
    print("Fetch failed")
}
}

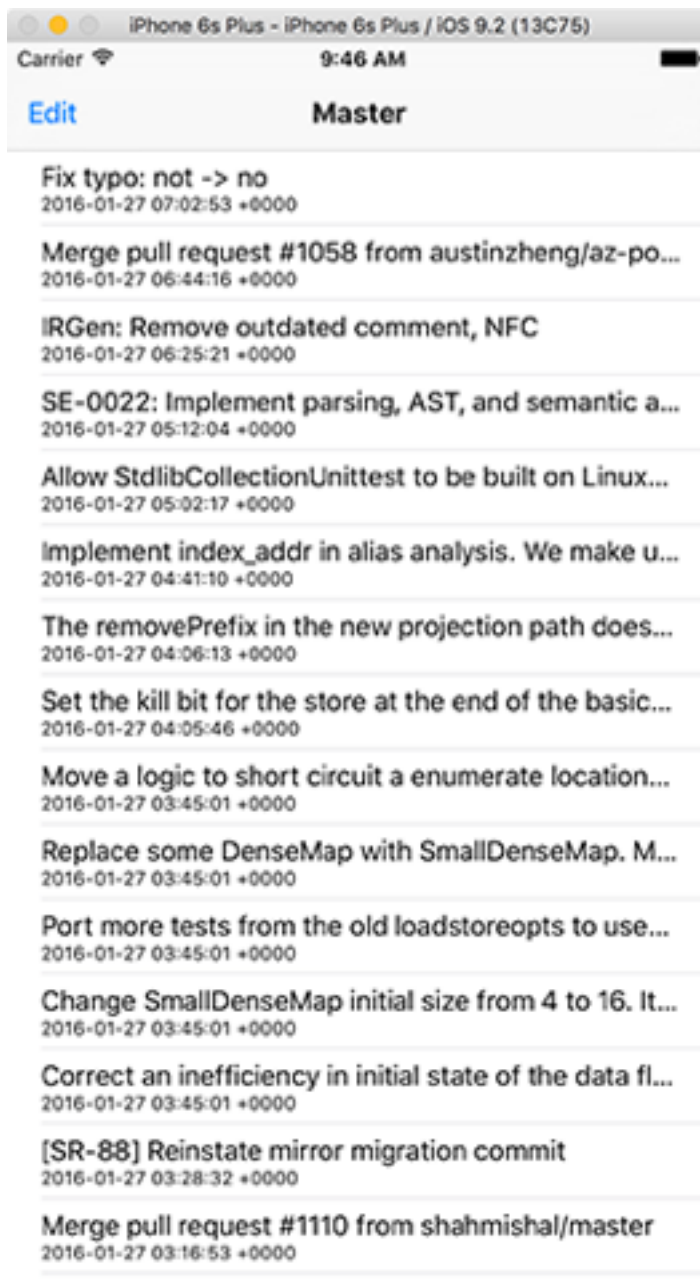
```

So, that creates the `NSFetchRequest`, gives it a sort descriptor to arrange the newest commits first, then uses the `executeFetchRequest()` method to fetch the actual objects. That method returns an array of `NSManagedObjects`, but the code safely downcasts that to be `Commit`. Once that's done, it's just a matter of assigning the resulting array to the `objects` property used in the Xcode template and calling `reloadData()` on the table – everything else is handled for us by the Xcode template.

To make the app work, we need to call this new `loadSavedData()` method in two places. First, add a call just after `startCoreData()` in the `viewDidLoad()` method. Second, add a call to the `fetchCommits()` method, just after where we have `self.saveContext()`. You will, of course, need to use `self.fetchCommits()` in that instance.

Again, I've written a simple `print()` statement when errors occur, but in your own production apps you will need to show something useful to your user.

Good news: that completes all four basic bootstrapping steps for Core Data. We have defined our model, loaded the data store and managed object context, fetched some example data and saved it, and loaded the resulting objects. You should now be able to run your project and see it all working!



Getting a crash? If your code crashes in the `cellForRowAtIndexPath` method, make sure you changed the table view's cell type to be `Subtitle` rather than `Basic`.

How to make a Core Data attribute unique using constraints

After such a huge amount of work getting Core Data up and running, you'll probably run your app a few times to enjoy it all working. But it's not perfect, I'm afraid: first, you'll see GitHub commits get duplicated each time the app runs,

and second you'll notice that tapping on a commit shows the `Detail` screen with not much helpful information.

We'll be fixing that second problem later, but for now let's focus on the first problem: duplicate `GitHub` commits. In fact, you probably have triplicate or quadruplicate by now, because each time you run the app the same commits are fetched and added to `Core Data`, so you end up with the same data being repeated time and time again.

No one wants repeated data, so we're going to fix this problem. And for once I'm pleased to say that `Core Data` makes this very easy thanks to a simple technology called “unique constraints”. All we need to do is find some data that is guaranteed to uniquely identify a commit, tell `Core Data` that is a unique identifier, and it will make sure objects with that same value don't get repeated.

Even better, `Core Data` can intelligently merge updates to objects in situations where this is possible. It's not going to happen with us, but imagine a situation where a commit author could retrospectively change their commit message from “I fixed a bug with Swift” to “I fixed a bug with Swift”. As long as the unique identifier didn't change, `Core Data` could recognize this was an update on the original commit, and merge the change intelligently.

Inserting data uniquely used to be harder before `iOS 9`, because you needed to run an `NSFetchRequest` to see if an object matching your identifier existed, then create it or update it as needed. As of `iOS 9`, this is done automatically: you just need to tell `Core Data` what your unique attribute is.

In this app, we have the perfect unique attribute just waiting to be used: every commit has a “sha” attribute that is a long string of letters and numbers that

identify that commit uniquely. SHA stands for “secure hash algorithm”, and it's used in many places to generate unique identifiers from content.

A “hash” is a little bit like one-way, truncated encryption: one piece of input like “Hello world” will always generate the same hash, but if you change it to be “Hello World” – just capitalising a single letter – you get a completely different hash. It's “truncated” because no matter how much content you give it as input, the “sha” will always be 40 letters. It's “one way” because you can't somehow reverse the hash to discover the original content, which is where hashes are different to encryption: an encrypted message can be decrypted to its original content, whereas a hashed message cannot be “dehashed” back to its original.

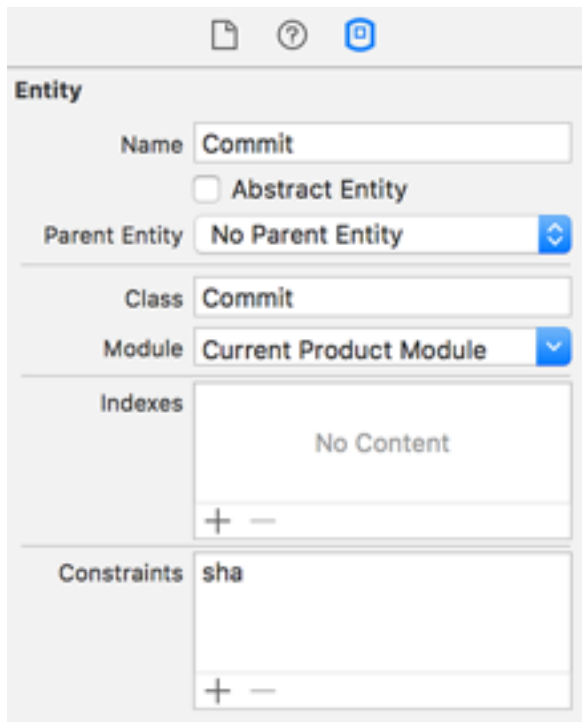
Hashes are frequently used as a checksum to verify that a file or data is correct: if you download a 10GB file and want to be sure it's exactly what the sender created, you can just compare your hash with theirs. Because hashes are truncated to a specific size, it is technically possible for two pieces of very different content to generate the same hash, known as a “collision”, but this is extremely rare.

Enough theory. Please go ahead and run your app a few times to make sure there are a good number of duplicates so you can see the problem in action. We added some `print()` statements in there for debugging purposes, so you'll see a message like this:

```
Got 500 commits
Received 100 new commits.
Got 600 commits
```

Select the data model (`Project38.xcdatamodeld`) and make sure the `Commit` entity is selected rather than one of its attributes. If you look in the `Data`

Model inspector you'll see a field marked "Constraints" – click the + button at the bottom of that field. A new row will appear saying "comma,separated,properties". Click on that, hit Enter to make it editable, then type "sha" and hit Enter again to save the changes.



Now for the important part, because in my testing Xcode didn't behave quite as it ought to. First, go to the the iOS simulator, then choose the Simulator menu and choose Reset Content And Settings. Now quit Xcode, and relaunch it. Re-open your project, hold down the Alt key, and go to Product > Clean Build Folder from the menu and click Clean when prompted.

What you just did was completely reset the state of your project and the iOS Simulator. The reason this is required is two-fold: first, because you just made an important change to your model, which is generally a bad idea unless you know what you're doing; second, because Xcode frequently fails to register changes to unique constraints unless you explicitly save, so by asking you to restart Xcode and clean your project I can feel fairly confident that everyone will be in a working state.

Before you run your project again, I want you to make one tiny code change. In your `startCoreData()` method, add this line to where the managed object context is configured:

```
managedObjectContext.mergePolicy =  
NSMergeByPropertyObjectTrumpMergePolicy
```

It should look like this:

```
managedObjectContext =  
NSManagedObjectContext(concurrencyType: .MainQueueConcurrency  
Type)  
managedObjectContext.persistentStoreCoordinator = coordinator  
managedObjectContext.mergePolicy =  
NSMergeByPropertyObjectTrumpMergePolicy
```

This instructs Core Data to allow updates to objects: if an object exists in its data store with message A, and an object with the same unique constraint (“sha” attribute) exists in memory with message B, the in-memory version “trumps” (overwrites) the data store version.

Go ahead and run your project a few times now and you'll see this message in the Xcode log:

```
Got 100 commits  
Received 100 new commits.  
Got 100 commits
```

As you can see, 100 commits were loaded from the persistent store, 100 “new” commits were pulled in from GitHub, and after Core Data resolved unique attributes there were still only 100 commits in the persistent store. Perfect! If you run your project again after a few hours, the numbers will start to go up slowly as new commits appear on GitHub — Swift is a live project, after all!

Note: in a couple of chapters I'll be introducing you to something called `NSFetchedResultsController`. Using attribute constraints can cause problems with `NSFetchedResultsController`, but in this tutorial we're always doing a full save and load of our objects because it's an easy way to avoid problems later. Don't worry about it for now – I'll mention it again at the appropriate time.

Examples of using `NSPredicate` to filter `NSFetchRequest`

Predicates are one of the most powerful features of Core Data, but they are actually useful in lots of other places too so if you master them here you'll learn a whole new skill that can be used elsewhere. For example, if you already completed Project 33 you'll have seen how predicates let us find iCloud objects by reference.

Put simply, a predicate is a filter: you specify the criteria you want to match, and Core Data will ensure that only matching objects get returned. The best way to learn about predicates is by example, so I've created three examples below that demonstrate various different filters. We'll be adding a fourth one in the next chapter once you've learned a bit more.

First, add this new property to the `MasterViewController` class:

```
var commitPredicate: NSPredicate?
```

I've made that an optional `NSPredicate` because that's exactly what our fetch request takes: either a valid predicate that specifies a filter, or nil to mean “no filter”.

Find your `loadSavedData()` method and add this line just below where the `sortDescriptors` property is set:

```
fetch.predicate = commitPredicate
```

With that property in place, all we need to do is set it to whatever predicate we want before calling `loadSavedData()` again to refresh the list of objects. The easiest way to do this is by adding a new method called `changeFilter()`, which we'll use to show an action sheet for the user to choose from. First we need to add a button to the navigation bar that will call this method, so put this code into `viewDidLoad()`:

```
navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Filter", style: .Plain, target: self, action:
#selector(changeFilter))
```

And here's an initial version of that new method for you to add to your view controller:

```
func changeFilter() {
    let ac = UIAlertController(title: "Filter commits...",
message: nil, preferredStyle: .ActionSheet)

    // 1
    // 2
    // 3
    // 4

    ac.addAction(UIAlertAction(title: "Cancel",
style: .Cancel, handler: nil))
```

```
        presentViewController(ac, animated: true, completion:
nil)
    }
```

We'll be replacing the four comments one by one as you learn about predicates.

Let's start with something easy: matching an exact string. If we wanted to find commits with the message “I fixed a bug in Swift” – the kind of commit message that is frowned upon because it's not very descriptive! – you would write a predicate like this:

```
commitPredicate = NSPredicate(format: "message == 'I fixed a
bug in Swift'")
```

That means “make sure the message attribute is equal to this exact string”. Typing an exact string like that is OK because you know what you're doing, but please don't ever use string interpolation to inject user values into a predicate. If you want to filter using a variable, use this syntax instead:

```
let filter = "I fixed a bug in Swift"
commitPredicate = NSPredicate(format: "message == %@",
filter)
```

The `%@` will be instantly recognizable to anyone who has used Objective-C before, and it means “place the contents of a variable here, whatever data type it is”. In our case, the value of `filter` will go in there, and will do so safely regardless of its value.

Like I said, “I fixed a bug in Swift” isn't the kind of commit message you'll see in your data, so `==` isn't really a helpful operator for our app. So let's

write a real predicate that will be useful: put this in place of the `// 1` comment in the `changeFilter()` method:

```
ac.addAction(UIAlertAction(title: "Show only fixes",
style: .Default, handler: { [unowned self] _ in
    self.commitPredicate = NSPredicate(format: "message
CONTAINS[c] 'fix'")
    self.loadSavedData()
}))
```

The `CONTAINS[c]` part is an operator, just like `==`, except it's much more useful for our app. The `CONTAINS` part will ensure this predicate matches only objects that contain a string somewhere in their message – in our case, that's the text “fix”. The `[c]` part is predicate-speak for “case-insensitive”, which means it will match “FIX”, “Fix”, “fix” and so on. Note that we need to use `self.` twice inside the closure to make capturing explicit.

Another useful string operator is `BEGINSWITH`, which works just like `CONTAINS` except the matching text must be at the start of a string. To make this second example more exciting, I'm also going to introduce the `NOT` keyword, which flips the match around: this action below will match only objects that don't begin with 'Merge pull request'. Put this in place of the `// 2` comment:

```
ac.addAction(UIAlertAction(title: "Ignore Pull Requests",
style: .Default, handler: { [unowned self] _ in
    self.commitPredicate = NSPredicate(format: "NOT message
BEGINSWITH 'Merge pull request'")
    self.loadSavedData()
}))
```

For a third and final predicate, let's try filtering on the “date” attribute. This is the `NSDate` data type, and `Core Data` is smart enough to let us compare that date to any other date inside a predicate. In this example, which should go in place of the `// 3` comment, we're going to request only commits that took place 43,200 seconds ago, which is equivalent to half a day:

```
ac.addAction(UIAlertAction(title: "Show only recent",
style: .Default, handler: { [unowned self] _ in
    let twelveHoursAgo =
NSDate().dateByAddingTimeInterval(-43200)
    self.commitPredicate = NSPredicate(format: "date > %@",
twelveHoursAgo)
    self.loadSavedData()
})))
```

Again, the magic `%@` will work with `Core Data` to ensure the `NSDate` we created is used correctly in the query.

For the final comment, `// 4`, we're just going to set `commitPredicate` to be `nil` so that all commits are shown again:

```
ac.addAction(UIAlertAction(title: "Show all commits",
style: .Default, handler: { [unowned self] _ in
    self.commitPredicate = nil
    self.loadSavedData()
})))
```

That's it! `NSPredicate` uses syntax that is new to you so you might find it a bit daunting at first, but it really isn't very hard once you have a few examples to work from, and it does offer a huge amount of power to your apps.

Adding Core Data entity relationships: lightweight vs heavyweight migration

It's time to take your Core Data skills up a notch: we're going to add a second entity called `Author`, and link that entity to our existing `Commit` entity. This will allow us to attach an author to every commit, but also to find all commits that belong to a specific author.

Open the data model (`Project38.xcdatamodeld`) for editing, then click the `Add Entity` button. Name the entity `Author`, then give it two attributes: `name` and `email`. Please make both strings, and make sure both are not marked as optional. This time we're also going to make one further change: select the `name` attribute and check the box marked `Indexed`.

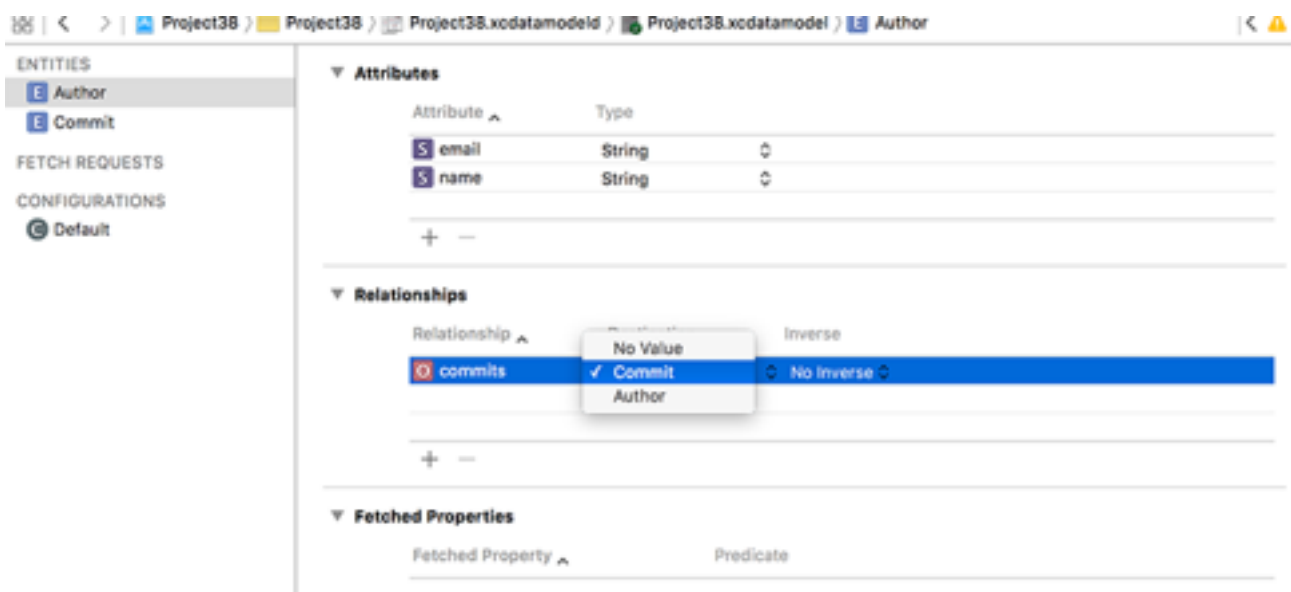
The screenshot shows the Xcode Core Data model editor interface. At the top, there are icons for a list, a link, a double-headed arrow, and three window icons. Below these are icons for a document, a question mark, and a Core Data icon. The main panel is titled 'Attribute' and contains the following fields and controls:

- Name:** A text field containing the value 'name'.
- Properties:** A section with three checkboxes:
 - ☐ Transient
 - ☐ Optional
 - ☒ Indexed
- Attribute Type:** A dropdown menu showing 'String' with a blue arrow icon on the right.
- Validation:** Two rows of controls:
 - First row: A text field with 'No Value', a spinner icon, and a checkbox for 'Min Length'.
 - Second row: A text field with 'No Value', a spinner icon, and a checkbox for 'Max Length'.
- Default Value:** A text field containing 'Default Value'.
- Reg. Ex.:** A text field containing 'Regular Expression'.
- Advanced:** A section at the bottom with a checkbox for 'Index in Spotlight'.

An indexed attribute is one that is optimized for fast searching. There is a cost to creating and maintaining each index, which means you need to choose carefully which attributes should be index. But when you find a particular fetch request is happening slowly, chances are it's because you need to index an attribute.

We want every `Author` to have a list of commits that belong to them, and every `Commit` to have the `Author` that created it. In `Core Data`, this is represented using `relationships`, which are a bit like calculated attributes except `Core Data` adds extra functionality to handle the situation when part of a relationship gets deleted.

With the `Author` entity selected, click the `+` button under the `Relationships` section – it's just below the `Attributes` section. Name the new relationship “`commits`” and choose “`commit`” for its destination. In the `Data Model` inspector, change `Type` to be “`To Many`”, which tells `Core Data` that each author has many `Commits` attached to it.



Now choose the `Commit` entity we created earlier and add a relationship named “`author`”. Choose `Author` for the destination then change “`No Inverse`” to be “`commits`”. In the `Data Model` inspector, change `Type` to be “`To One`”, because each commit has exactly one author).

That's it for our model changes, but if you run the app now it will either do nothing or crash, both of which are bad. If your app does nothing, it means Xcode didn't spot the model change, so please repeat the procedure from before: quit Xcode, relaunch, then hold down `Alt` and choose `Product > Clean Build Folder`. If your app crashed – great! Well, not great, but it was at least supposed to do that.

We just changed our data model, and Core Data doesn't know how to handle that – it considers any variation in its data model an unwelcome surprise, so either we need to tell Core Data how to handle the changed model or we need to tell it to figure out the differences itself.

These two options are called “heavyweight migrations” and “lightweight migrations”. The latter is usually preferable, and is what we'll be doing here, but it's only possible when your changes are small enough that Core Data can perform the conversion correctly. We added a new “authors” relationship, so if we tell Core Data to perform a lightweight migration it will simply set that value to be empty.

To tell Core Data we want to perform a lightweight migration, we need to set two properties when configuring our persistent store coordinator in the `startCoreData()` method. First, `NSMigratePersistentStoresAutomaticallyOption` needs to be `true`, which tells Core Data to upgrade its SQLite database when the model changes. Second, `NSInferMappingModelAutomaticallyOption` also needs to be `true`, which tells Core Data to figure out the differences when the model changes, and apply sensible defaults if possible.

To use these two options, find this line in the `startCoreData()` method:

```
try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,  
configuration: nil, URL: url, options: nil)
```

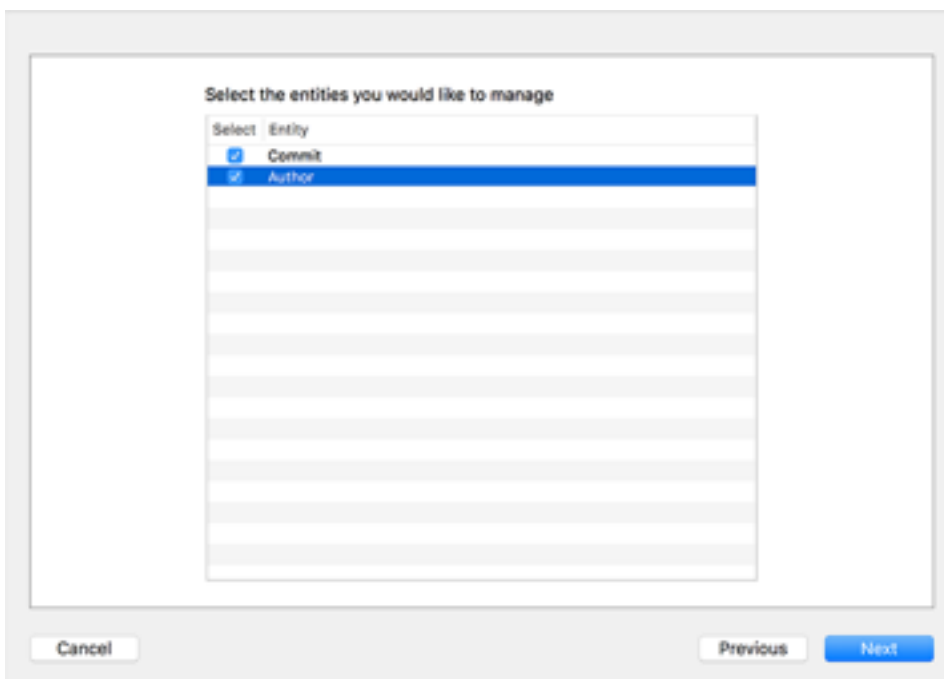
Now replace it with this:

```
try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
configuration: nil, URL: url, options:
[NSMigratePersistentStoresAutomaticallyOption: true,
NSInferMappingModelAutomaticallyOption: true])
```

That will stop the app from crashing, which is always a good thing. But it doesn't actually use our new `Author` entity, and to do that we first need to do something rather tedious: we need to re-use the `NSManagedObject` generator, which, if you remember, also means having to remove property optionals by hand.

So, go back to the data model, and choose Editor > Create NSManagedObject Subclass again. This time I want you to choose both

Author and Commit, but don't forget to change Group from the blue project icon to the yellow folder icon – Xcode does love to keep resetting that particular option.



Once the files are generated you'll now have four files: two each for `Author` and `Commit`. Go ahead and remove optionality from the properties in `Author+CoreDataProperties.swift` and `Commit+CoreDataProperties.swift`. While you're there, watch out for a strange corner case that might sometimes happen: in my testing sometimes `Commit` sometimes was created with `@NSManaged var author: NSManagedObject?`. If this happens to you, either recreate the classes once more or just change it to `@NSManaged var author: Author?` – then make it non optional, of course.

In order to attach authors to commits, I want to show you how to look for a specific named author, or create it if they don't exist already. Remember, we made the “name” attribute indexed, which makes it lightning fast for search. This needs to set up and execute a new `NSFetchRequest` (using an `== NSPredicate` to match the name), then use the result if there is one. If no matching author is found, we'll use `insertNewObjectForEntityForName()` to create and configure a new author, and use that instead.

Put this new code just before the end of the `configureCommit()` method:

```
var commitAuthor: Author!

// see if this author exists already
let authorFetchRequest = NSFetchRequest(entityName: "Author")
authorFetchRequest.predicate = NSPredicate(format: "name == %@", json["commit"]["committer"]["name"].stringValue)

if let authors = try?
managedObjectContext.executeFetchRequest(authorFetchRequest)
as! [Author] {
    if authors.count > 0 {
```

```

        // we have this author already
        commitAuthor = authors[0]
    }
}

if commitAuthor == nil {
    // we didn't find a saved author – create a new one!
    if let author =
NSEntityDescription.insertNewObjectForEntityForName("Author",
inManagedObjectContext: managedObjectContext) as? Author {
        author.name = json["commit"]["committer"]
["name"].stringValue
        author.email = json["commit"]["committer"]
["email"].stringValue
        commitAuthor = author
    }
}

// use the author, either saved or new
commit.author = commitAuthor

```

You'll note that I used `try?` for `executeFetchRequest()` this time, because we don't really care if the request failed: it will still fall through and get caught by the `if commitAuthor == nil` check later on.

To show that this worked, change your `cellForRowAtIndexPath` method so that the detail text label contains the author name as well as the commit date, like this:

```

cell.detailTextLabel!.text = "By \(object.author.name) on \(
object.date.description)"

```


You should be able to run the app now and see the author name appear after a moment, as Core Data merges the new data with the old. If you hit problems, don't be afraid to use “Reset Content and Settings” again in the simulator – that usually cures most Core Data hiccups.

We can also show that the inverse relationship works. Open `DetailViewController.swift` and change the `detailItem` property from `AnyObject?` to `Commit?`. Now change its `configureView()` method to this:

```
func configureView() {
    // Update the user interface for the detail item.
    if let detail = self.detailItem {
        if let label = self.detailDescriptionLabel {
            label.text = detail.message
            navigationItem.rightBarButtonItem =
UIBarButtonItem(title: "Commit 1/\
(detail.author.commits.count)", style: .Plain, target: self,
action: #selector(showAuthorCommits))
        }
    }
}
```

That will make a tappable button in the top-right corner showing how many other commits we have stored from this author. We haven't written a `showAuthorCommits()` method yet, but don't worry: that will be your homework later on!

Now that every commit has an author attached to it, I want to add one last filter to our `changeFilter()` method to show you just how clever `NSPredicate` is. Add this just before the “Show all commits” action:

```
ac.addAction(UIAlertAction(title: "Show only Durian commits",
style: .Default, handler: { [unowned self] _ in
    self.commitPredicate = NSPredicate(format: "author.name
== 'Joe Groff'")
    self.loadSavedData()
}))
```

There are three things that bear explaining in that code:

- By using `author.name` the predicate will perform two steps: it will find the “author” relation for our commit, then look up the “name” attribute of the matching object.
- Joe is one of Apple's Swift engineers. Although it's fairly likely you'll see commits by him, it can't be guaranteed – I'm pretty sure that Apple give him a couple of days vacation each year. Maybe.
- Durian is a fruit that's very popular in south-east Asia, particularly Malaysia, Singapore and Thailand. Although most locals are big fans, the majority of foreigners find that it really, really stinks, so I'm sure there's some psychological reason why Joe Groff chose it for his website: duriansoftware.com.

Run your app now and the new filter should work. Remember, it might not return any objects, depending on just how many commits Joe has done recently. No pressure, Joe! In those changes, I also modified the detail view controller so that it shows the commit message in full, or at least as full as it can given the limited space – this app is starting to come together!

How to delete a Core Data object

The Xcode template we've been using comes with support for deleting objects, but it only removes them from the objects array directly – nothing is actually changed in our Core Data set up. So if you “delete” a commit, it will be there when you relaunch the app rather than gone for good.

Fortunately, this is really easy to change: our managed object context has a `deleteObject()` method that will delete any object regardless of its type or location in the object graph. Once an object has been deleted from the context, we can then call `saveContext()` to write that change back to the persistent store so that the change is permanent.

All this is easy to do by adding three new lines of code to the `commitEditingStyle` method. Here's the new method:

```
override func tableView(tableView: UITableView,
commitEditingStyle editingStyle: UITableViewCellEditingStyle,
forRowAtIndexPath indexPath: NSIndexPath) {
    if editingStyle == .Delete {
        let commit = objects[indexPath.row]
        managedObjectContext.deleteObject(commit)
        objects.removeAtIndex(indexPath.row)
        tableView.deleteRowsAtIndexPaths([indexPath],
withRowAnimation: .Fade)

        saveContext()
    } else if editingStyle == .Insert {
```

```
        // Create a new instance of the appropriate class,  
insert it into the array, and add a new row to the table  
view.  
    }  
}
```

So, it 1) pulls out the `Commit` object that the user selected to delete, 2) removes it from the `managed object context`, 3) removes it from the `objects array`, 4) deletes it from the table view, then 5) saves the context. Lines 3 and 4 were there in the original template, so it's only lines 1, 2, and 5 that are new. Remember: you must call `saveContext()` whenever you want your changes to persist.

Try running the app now, and either swiping to delete rows or using the `Edit` button in the top left. As you'll see you can delete as many commits as you want, and everything seems to work great. Now try running the app once again, and you'll get a nasty shock: the deleted commits reappear! What's going on?

Well, if you think about it, the app is doing exactly what we told it to do: every time it runs it re-fetches the list of commits from `GitHub`, and merges it with the commits in its data store. This means any commits we try to delete just get redownloaded again – they really are being deleted, but then they get recreated as soon as the app is relaunched.

This problem is not a hard one to fix, and it gives me a chance to show you another part of `NSFetchRequest`: the `fetchLimit` property. This tells `Core Data` how many items you want it to return. What we're going to do is find the newest commit in our data store, then use the date from that to ask `GitHub` to provide only newer commits.

First, go to the `fetchCommits()` method and modify the start of it to this:

```

func fetchCommits() {
    let newestCommitDate = getNewestCommitDate()

    if let data = NSData(contentsOfURL: NSURL(string:
"https://api.github.com/repos/apple/swift/commits?
per_page=100&since=\(newestCommitDate)")!) {
        let jsonCommits = JSON(data: data)

```

We'll be adding the `getNewestCommitDate()` method shortly, but what it will return is a date formatted as an ISO-8601 string. This date will be set to one second after our most recent commit, and we can send that to the GitHub API using its “since” parameter to receive back only newer commits.

Here is the `getNewestCommitDate()` method – only three pieces of it are new, and I'll explain them momentarily.

```

func getNewestCommitDate() -> String {
    let formatter = NSDateFormatter()
    formatter.timeZone = NSTimeZone(name: "UTC")
    formatter.dateFormat = dateFormatISO8601

    let newestCommitFetchRequest = NSFetchRequest(entityName:
"Commit")
    let sort = NSSortDescriptor(key: "date", ascending:
false)
    newestCommitFetchRequest.sortDescriptors = [sort]
    newestCommitFetchRequest.fetchLimit = 1

    if let commits = try?
managedObjectContext.executeFetchRequest(newestCommitFetchReq
uest) as! [Commit] {

```

```

        if commits.count > 0 {
            return
formatter.stringFromDate(commits[0].date.dateByAddingTimeInterval(1))
        }
    }

    return
formatter.stringFromDate(NSDate(timeIntervalSince1970: 0))
}

```

The first of the new pieces of code is the `fetchLimit` property for the fetch request. As you might imagine, it's always more efficient to fetch as few objects as needed, so if you can set a fetch limit you should do so. Second, the `stringFromDate()` method is the inverse of the `dateFromString()` method we used when parsing the commit JSON. We use the same date format that was defined earlier, because GitHub's "since" parameter is specified in an identical way. Finally, `dateByAddingTimeInterval()` is used to add one second to the time from the previous commit, otherwise GitHub will return the newest commit again.

If no valid date is found, the method returns a date from the 1st of January 1970, which will reproduce the same behavior we had before introducing this date change.

This solution is a good start, but it has a small flaw – see if you can spot it! If not, don't worry: I'll be setting it as homework for you. Regardless, it gave me the chance to show you the `fetchLimit` property, and you know how much I love squeezing new knowledge in...

Optimizing Core Data Performance using NSFetchedResultsController

You've already seen how Core Data takes a huge amount of work away from you, which is great because it means you can focus on writing the interesting parts of your app rather than data management. But, while our current project certainly works, it's not going to scale well. To find out why open the `Commit.swift` file and modify it to this:

```
class Commit: NSManagedObject {
    override init(entity: NSEntityDescription,
insertIntoManagedObjectContext context:
NSManagedObjectContext?) {
        super.init(entity: entity,
insertIntoManagedObjectContext: context)
        print("Init called!")
    }
}
```

When you run the program now you'll see "Init called!" in the Xcode log at least a hundred times - once for every `Commit` object that gets pulled out in our `loadSavedData()` method. So what if there are 1000 objects? Or 10,000? Clearly it's inefficient to create a new object for everything in our object graph just to load the app, particularly when our table view can only show a handful at a time.

Core Data has a brilliant solution to this problem, and it's called `NSFetchedResultsController`. It takes over our existing `NSFetchRequest` to load data, replaces our objects array with its own storage, and even works to ensure the user interface stays in sync with changes to the data by controlling the way objects are inserted and deleted.

No tutorial on Core Data would be complete without teaching `NSFetchedResultsController`, so that's the last thing we'll be doing in this project. I left it until the end because, although it's very clever and certainly very efficient, `NSFetchedResultsController` is entirely optional: if you're happy with the project as it is, you're welcome to skip over this last chapter.

First, add a new property for the fetched results controller:

```
var fetchedResultsController: NSFetchedResultsController!
```

We now need to rewrite our `loadSavedData()` method so that the existing `NSFetchRequest` is wrapped inside a `NSFetchedResultsController`. We want to create that fetched results controller only once, but retain the ability to change the predicate when the method is called again.

Before I show you the code, there are three new things to learn. First, we're going to be using the `fetchBatchSize` property of our fetch request so that only 20 objects are loaded at a time. Second, we'll be setting the master view controller as the delegate for the fetched results controller – you'll see why soon. Third, we need to use the `performFetch()` method on our fetched results controller to make it load its data.

Here's the revised `loadSavedData()` method:

```
func loadSavedData() {
    if fetchedResultsController == nil {
        let fetch = NSFetchRequest(entityName: "Commit")
        let sort = NSSortDescriptor(key: "date", ascending:
false)
        fetch.sortDescriptors = [sort]
```



```

        fetch.fetchBatchSize = 20

        fetchedResultsController =
NSFetchedResultsController(fetchRequest: fetch,
managedObjectContext: managedObjectContext,
sectionNameKeyPath: nil, cacheName: nil)
        fetchedResultsController.delegate = self
    }

    fetchedResultsController.fetchRequest.predicate =
commitPredicate

    do {
        try fetchedResultsController.performFetch()
        tableView.reloadData()
    } catch {
        print("Fetch failed")
    }
}

```

Because we're setting delegate, you'll also need to make `MasterViewController` conform to the `NSFetchedResultsControllerDelegate` protocol, like this:

```

class MasterViewController: UITableViewController,
NSFetchedResultsControllerDelegate {

```

That was the easy part. When you use `NSFetchedResultsController`, you need to use it everywhere: that means it tells you how many sections and rows you have, it keeps track of all the objects, and it is the single source of truth when it comes to inserting or deleting objects.

You can get an idea of what work needs to be done by deleting the `objects` property: we don't need it any more, because the fetched results controller stores our results. Immediately you'll see five errors appear wherever that property was being touched, and we need to rewrite all those instances to use the fetched results controller.

First, the `prepareForSegue()` method. This line will be flagged as an error:

```
let object = objects[indexPath.row]
```

Please replace it with this:

```
let object =  
fetchedResultsController.objectAtIndex(indexPath) as!  
Commit
```

You'll note that the fetched results controller uses index paths (i.e., sections as well as rows) rather than just a flat array. We also need to typecast its contents, because it returns `AnyObject` by default.

Second, replace the `numberOfSectionsInTableView()` and `numberOfRowsInSection` methods with these two new implementations:

```
override func numberOfSectionsInTableView(tableView:  
UITableView) -> Int {  
    return fetchedResultsController.sections?.count ?? 0  
}
```

```
override func tableView(tableView: UITableView,  
numberOfRowsInSection section: Int) -> Int {
```

```
        let sectionInfo = fetchedResultsController.sections!  
[section]  
        return sectionInfo.numberOfObjects  
    }  
}
```

As you can see, we can read the sections array, each of which contains an array of `NSFetchedResultsController` objects describing the items in that section. For now, we're just going to use that for the number of objects in the section.

Third, find this line inside the `cellForRowAtIndexPath` method:

```
let object = objects[indexPath.row]
```

And replace it with this instead:

```
let object =  
self.fetchedResultsController.objectAtIndex(indexPath)  
as! Commit
```

The final two errors are in the `commitEditingStyle` method, where deleting items happens. And this is where things get more complicated: we can't just delete items from the fetched results controller, and neither can we use `deleteRowsAtIndexPaths()` on the table view. Instead, Core Data is much more clever: we just delete the object from the managed object context directly.

You see, when we created our `NSFetchedResultsController`, we hooked it up to our existing managed object context, and we also made our current view controller its delegate. So when the managed object context detects an

object being deleted, it will inform our fetched results controller, which will in turn automatically notify our view controller if needed.

So, to delete objects using fetched results controllers you need to rewrite the `commitEditingStyle` method to this:

```
override func tableView(tableView: UITableView,
commitEditingStyle editingStyle: UITableViewCellEditingStyle,
forRowAtIndexPath indexPath: NSIndexPath) {
    if editingStyle == .Delete {
        let commit =
        fetchedResultsController.objectAtIndexPath(indexPath) as!
        Commit
        managedObjectContext.deleteObject(commit)
        saveContext()
    } else if editingStyle == .Insert {
        // Create a new instance of the appropriate class,
        insert it into the array, and add a new row to the table
        view.
    }
}
```

As you can see, that code pulls the object to delete out from the fetched results controller, deletes it, then saves the changes – we no longer touch the table view here.

That being said, we do need to add one new method that gets called by the fetched results controller when an object changes. We'll get told the index path of the object that got changed, and all we need to do is pass that on to the `deleteRowsAtIndexPaths()` method of our table view:

```

func controller(controller: NSFetchedResultsController,
didChangeObject anObject: AnyObject, atIndexPath indexPath:
NSIndexPath?, forChangeType type: NSFetchedResultsControllerChangeType,
newIndexPath: NSIndexPath?) {
    switch type {
    case .Delete:
        tableView.deleteRowsAtIndexPaths([indexPath!],
withRowAnimation: .Automatic)

    default:
        break
    }
}

```

Now, you might wonder why this approach is an improvement – haven't we just basically written the same code only in a different place? Well, no. That new delegate method we just wrote could be called from anywhere: if we delete an object in any other way, for example in the detail view, that method will now automatically get called and the table will update. In short, it means our data is driving our user interface, rather than our user interface trying to control our data.

Previously, I said “Using attribute constraints can cause problems with NSFetchedResultsController, but in this tutorial we're always doing a full save and load of our objects because it's an easy way to avoid problems later”. It's time for me to explain the problem: attribute constraints are only enforced as unique when a save happens, which means if you're inserting data then an NSFetchedResultsController may contain duplicates until a save takes place. This won't happen for us because I've made the project perform a save before a load to make things easier, but it's something you need to watch out for in your own code.

If you run the app now, you'll see "Init called!" appears far less frequently because the fetched results controller lazy loads its data – a significant performance optimization.

Before we're done with `NSFetchedResultsController`, I want to show you one more piece of its magic. You've seen how it has sections as well as rows, right? Well, try changing its constructor in `loadSavedData()` to be this:

```
fetchedResultsController =  
NSFetchedResultsController(fetchRequest: fetch,  
managedObjectContext: managedObjectContext,  
sectionNameKeyPath: "author.name", cacheName: nil)
```

The only change there is that I've provided a value for `sectionNameKeyPath` rather than `nil`. Now try adding this new method to `MasterViewController`:

```
override func tableView(tableView: UITableView,  
titleForHeaderInSection section: Int) -> String? {  
    return fetchedResultsController.sections![section].name  
}
```

Edit	Master	Filter	↕	Detail	Commit 1/8
	Janek Spaderna				
	Move logic into one place by mergi...				
	By Janek Spaderna on 2016-01-25 19:43:23 +0000				
	Jordan Rose				
	[ClangImporter] Wait until a bridgi...				
	By Jordan Rose on 2016-01-27 22:55:16 +0000				
	[docs] LibraryEvolution: Public me...				
	By Jordan Rose on 2016-01-27 01:45:30 +0000				
	[docs] LibraryEvolution: To remov...				
	By Jordan Rose on 2016-01-27 01:45:29 +0000				
	Merge pull request #987 from ddu...				
	By Jordan Rose on 2016-01-26 05:01:12 +0000				
	[docs] LibraryEvolution: Eliminate t...				
	By Jordan Rose on 2016-01-26 00:43:19 +0000				
	[docs] LibraryEvolution: Move "Ver...				
	By Jordan Rose on 2016-01-26 00:43:19 +0000				
	[docs] LibraryEvolution: Weaken "f				

If you run the app now, you'll see the table view has sections as well as rows, with each section marked out with the author of those commits. So, `NSFetchedResultsController` is not only faster, but it even adds powerful functionality with one line of code – what's not to like?

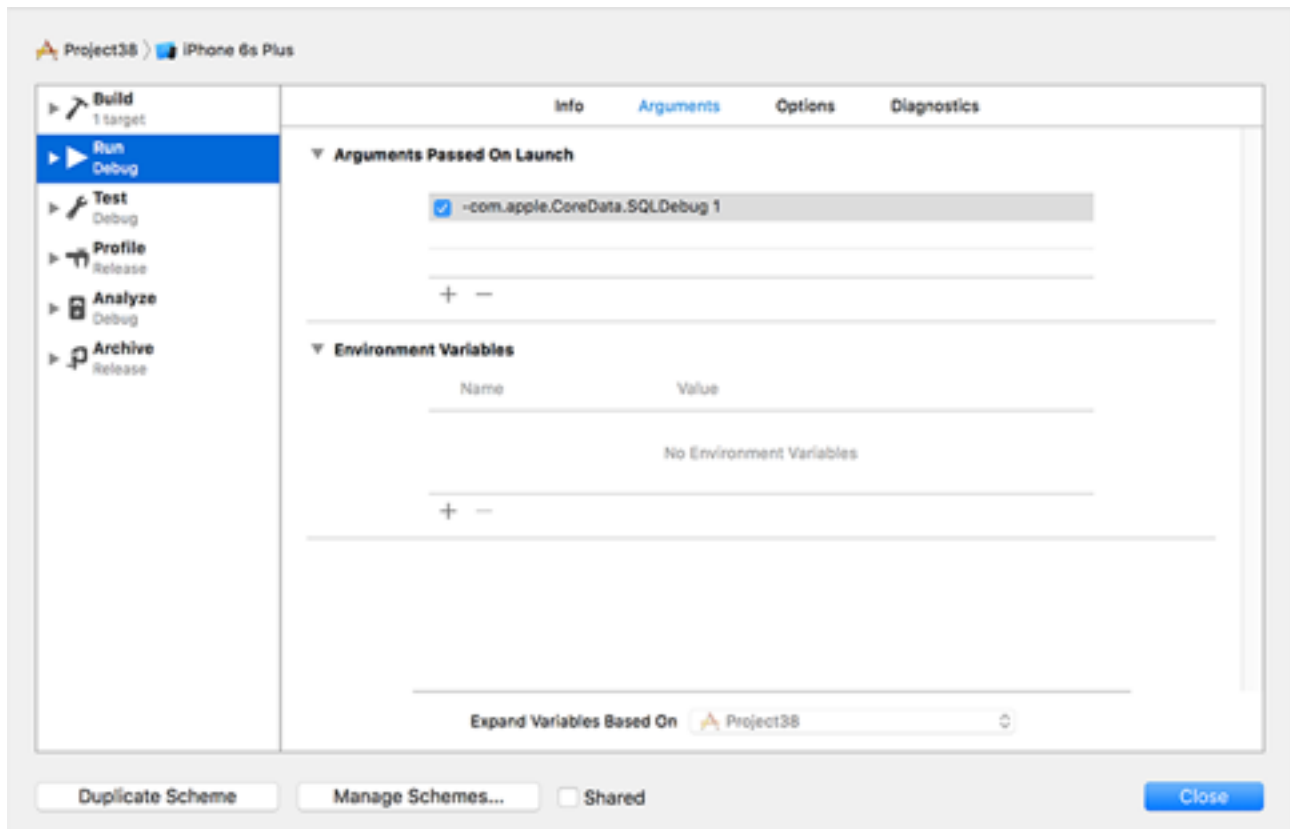
Wrap up

Core Data is complicated. It really is. So if you've made it this far, you deserve a pat on the back because you've learned a huge amount in this one project – good job! You've learned about model design, relationships, predicates, sort descriptors, persistent stores, managed object contexts, fetch requests, fetched results controllers, indexes, hashes and more, and I hope you're pleased with the final project.

But I'm afraid I have some bad news: even with everything you've learned, there's still a lot more to go if you really want to master Core Data. Migrating data models, multiple managed object contexts, delete rules, and thread safety should be top of that list, but at the very least I hope I've managed to give you a firm foundation on the technology – and perhaps even get you a bit excited about what it can do for you!

At this point, you know enough about Core Data that you should be able to start a new project with the Master–Detail Application template, but this time enabling the Core Data option for the template. You'll see all the code Apple generates for you, and hopefully you'll see how it's similar to our own code in this project.

Before you're done, I have two small tips for you. First, go to the Product menu and choose Scheme > Edit Scheme. In the window that appears, choose your Run target and select the Arguments tab. Now click + and enter the text `-com.apple.CoreData.SQLDebug 1`. Once that's done, running your app will print debug SQL into the Xcode log pane, allowing you to see what Core Data is up to behind the scenes.



Warning: when this option is enabled, you will see scary language like “fault fulfilled from database”. No, there isn't really a fault in your code – it's Core Data's way of saying that the objects it lazy loaded need to be loaded for real, so it's going back to the database to read them. You'll get into this more when you explore Core Data further, but for now relax: it's just a poor choice of phrasing from Apple.

Second, if you make changes to your model, watch out for the message “The model used to open the store is incompatible with the one

used to create the store". If you intend to turn this into a production app, you need to check your managed object context after `startCoreData()` has run to see whether everything was started correctly, and handle things gracefully on failure. While you're testing, don't be afraid to reset the iOS simulator as often as you need to in order to clean out old models.