CommitsApp

Practice in Swift with Xcode (tutorial #4)

Description: Get started with Core Data by building an app to fetch and store GitHub commits for Swift.

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.

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 `ISO8601DateFormatter` for the first time, which is one of Apple's ways to convert `Date` objects to and from strings.

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.

Before we start, it's important we reiterate that Core Data can be a bit overwhelming at first. It has a lot of unique terminology, so 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 practice, we will implement the four pieces of Core Data boilerplate. We're going to use Xcode's built-in Single View Application template, but we're not ***not*** going to have it generate Core Data code for us because we want you to understand it from the ground up.

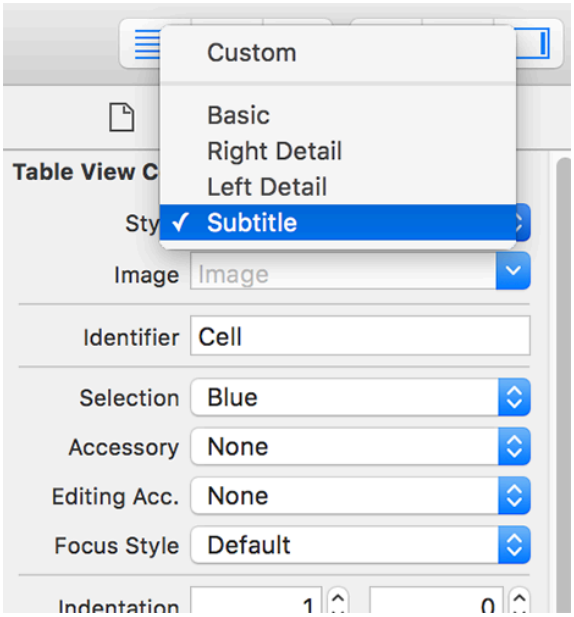
So, please go ahead and create a new Single View Application project named CommitsApp. Select Swift for your language, select Universal for your device type, then make sure you *uncheck* Core Data otherwise the rest of this practice will be very confusing indeed.

We need to parse the JSON coming from GitHub's API, and the easiest way to do that is with SwiftyJSON. If you haven't already downloaded the content for this project, please get it from GitHub now. You'll see SwiftyJSON is there, so please drag that into your project now.

We're going to use a table view controller rather than a regular view controller, so we need to do the same conversion job we've done several times before. So:

1. Open ViewController.swift and make it inherit from `UITableViewController`.
2. Open Main.storyboard and delete the view controller that's there right now.

3. Set its class to be “ViewController”.
4. Make it the initial view controller for the storyboard.
5. Embed it inside a navigation controller.
6. Select its prototype cell, give it the style Subtitle, the identifier “Commit”, and a disclosure indicator for its accessory.



We also need a detail view controller, but it doesn’t need to do much – it’s just there to show that everything works correctly. So, drag out a new view controller from the object library, and give it the storyboard ID “Detail”. Drag a label out onto it so that it fill the view controller, then set Auto Layout rules so that it always stays edge to edge. Center its text, then give it 0 for its number of lines property.

The main table view controller class and interface are done for now, but we need to create a new class to handle the detail view controller. So, go to File > New > File and choose iOS > Source > Cocoa Touch Class. Name it “DetailViewController”, make it a subclass of “UIViewController”, then click Finish and Create.

Back in Main.storyboard, change the class of the detail view controller to be “DetailViewController”, then switch to the assistant editor and create an outlet for the label called “detailLabel”.

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.

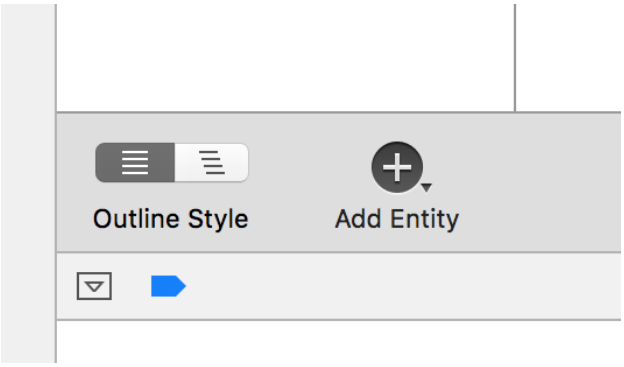
Designing a Core Data model

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.

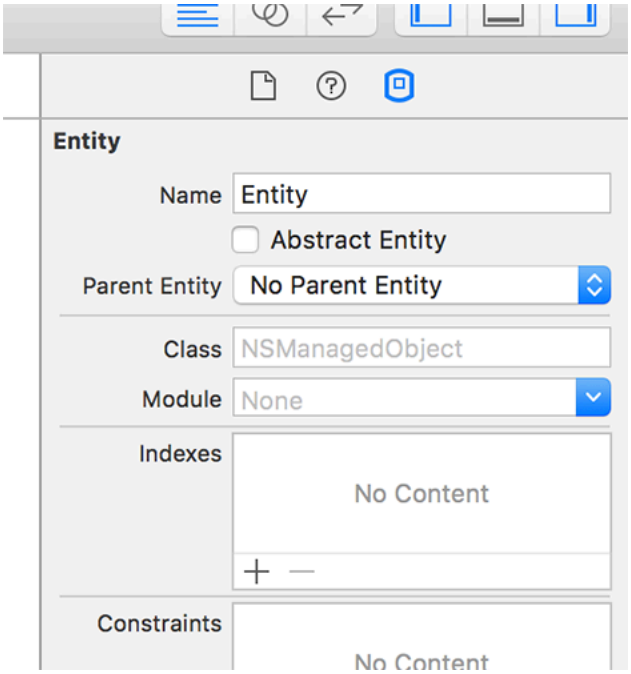
We're going to create a data model for our app that will store a list of all the GitHub commits for the Swift library. Take a look at the raw GitHub JSON now by loading this URL in a web browser: https://api.github.com/repos/apple/swift/commits?per_page=100. You'll see that each commit has a "sha" identifier, 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 CommitsApp, 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 CommitsApp.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.



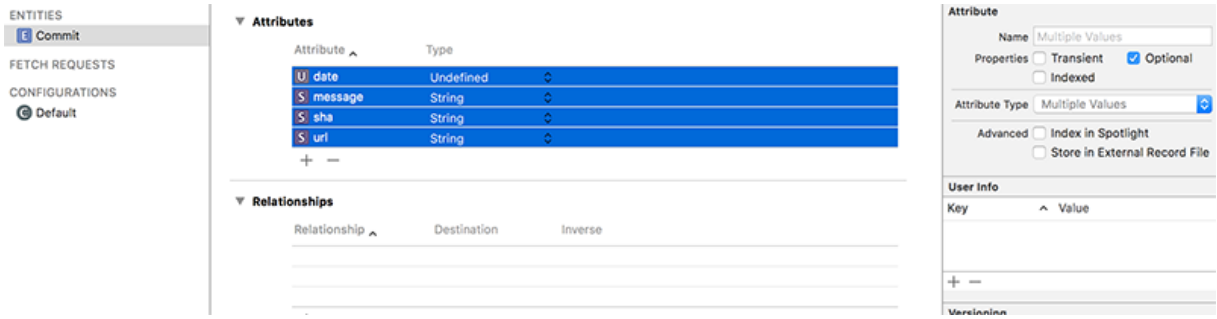
A Core Data "entity" is like a Swift class in that it is just a description of what an object is going to look like. By default, new entities are called "Entity", but you can change that in the Data Model inspector in the right-hand pane of Xcode – press Alt+Cmd+3 if it's not already visible. With your 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.



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.

Warning: When you make *any* changes to the Core Data editor in Xcode, you should press Cmd+S to save your changes. At the time of writing – and indeed for some time now – Xcode has not saved Core Data model changes when you build your app, so if you don't save the changes yourself you'll find they haven't been applied and you'll spend hours investigating ghost bugs.

Adding Core Data to our project: NSPersistentContainer

A Core Data model defines what your data should look like, but it doesn't actually store the real data anywhere. To make our app work, we need to load that model, create a real working database from it, load that database, then prepare what's called a "managed object context" – an environment where we can create, read, update, and delete Core Data objects entirely in memory, before writing back to the database in one lump.

This all used to be a massive amount of work, to the point where it would put people off Core Data for life. But from iOS 10 onwards, Apple rolled all this work up into a single new class called `NSPersistentContainer`. This has removed almost all the tedium from setting up Core Data, and you can now get up and running in just a few lines of code.

So, 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 – all using the new `NSPersistentContainer` class. Once it finishes its work, we'll have a managed object context ready to work with, and any changes we make to Core Data objects won't be saved until we explicitly request it. It is

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.

Unless you plan to get into more advanced usage, you don't need to know anything about SQLite to use Core Data.

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

```
import CoreData
```

We're going to create the `NSPersistentContainer` as a property, so we can load it once and share it elsewhere in our app. So, add this property now:

```
var container: NSPersistentContainer!
```

To set up the basic Core Data system, we need to write code 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 `URL` pointing to the database on disk where our actual saved objects live. This will be an SQLite database named CommitsApp.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.

Beautifully, brilliantly, all five of those steps are exactly what `NSPersistentContainer` does for us. So what used to be 15 to 20 lines of code is now summed up in just six – add this to `viewDidLoad()` now:

```
container = NSPersistentContainer(name: "CommitsApp")

container.loadPersistentStores { storeDescription, error in
    if let error = error {
        print("Unresolved error \(error)")
    }
}
```

The first line creates the persistent container, and must be given the name of the Core Data model file we created earlier: "CommitsApp". The next line calls the `loadPersistentStores()` method, which loads the saved database if it exists, or creates it otherwise. If any errors come back here you'll know something has gone fatally wrong, but if it succeeds then you can be guaranteed the data has loaded and you're ready to continue.

There's one small thing we do still need to do ourselves, and that's to write a small method to save any changes from memory back to the database on disk. The persistent container gives us a property called `viewContext`, which is a managed object context: an environment where we can manipulate Core Data objects entirely in RAM.

Once you've finished your changes and want to write them permanently – i.e., save them to disk – you need to call the `save()` method on the `viewContext` property. However, this should

only be done if there are any changes since the last save – there's no point doing unnecessary work. So, before calling `save()` you should read the `hasChanges` property. We're going to wrap this all up in a single method called `saveContext()` – add this new method just after `viewDidLoad()`:

```
func saveContext() {
    if container.viewContext.hasChanges {
        do {
            try container.viewContext.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 it into a managed object context 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 its properties. 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.

Let's look at it briefly now: open CommitsApp.xcdatamodeld, select the Commit entity again, then look in the data model inspector for the "Codegen" option. Change it to "Class Definition", press Cmd+S to save the change, then press Cmd+B to have Xcode build the project.

What just changed might look small, but it's remarkably smart. Open ViewController.swift and add this code at the end of `viewDidLoad()`:

```
let commit = Commit()
commit.message = "Woo"
commit.url = "http://www.example.com"
```

Can you figure out what Xcode has done for us? The Codegen value is short for "code generation" – when you pressed Cmd+B to build your project, Xcode converted the Commit Core Data entity into a Commit Swift class. You can't see it in the project – it's dynamically generated when the Swift code is being built – but it's there for you to use, as you just saw. You get access to its attributes as properties that you can read and write, and any changes you make will get written back to the database when you call our `saveContext()` method.

However, this feature is imperfect, at least right now – although that might change at any point in the future as Apple updates Xcode. First, try adding this line below the previous three:

```
commit.date = Date()
```

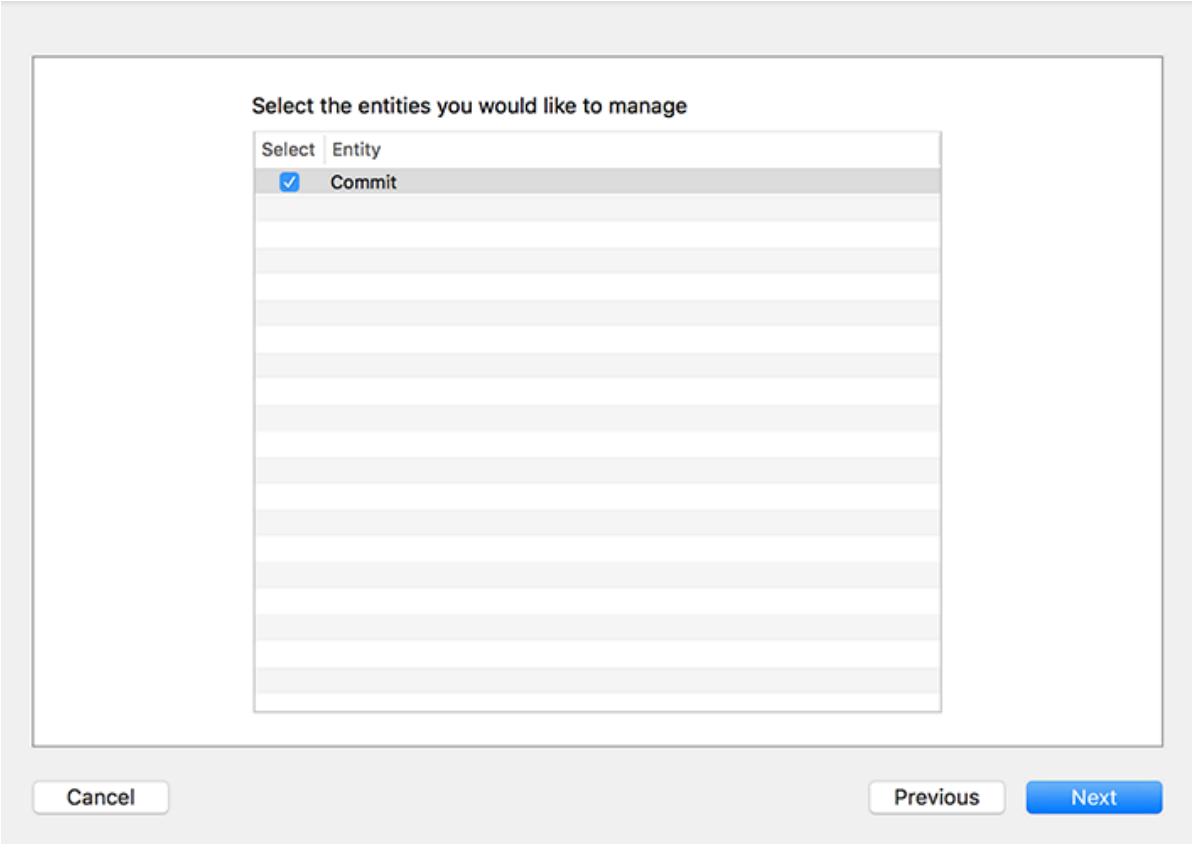
That means "set the `date` property to the current date." But it won't work – right now Xcode generates its classes using the old Swift 2.2 `NSDate` class rather than the shiny new Swift 3 `Date` struct. This means you need to use `NSDate` in your own Swift code, or add conversions everywhere, neither of which are pleasant.

Xcode's auto-generated class also has one more annoyance, and you'll see it if you try using code completion to view its properties: all four of the properties it made for us are optional, so `name` is a `String?`, `date` is an `NSDate?` and so on. Yes, even though we marked all the attributes as non-optional in the Core Data editor, that just means they need to have values by the time they get saved – Xcode will quite happily let them be nil at other times.

Sometimes that's OK, but usually it's not. So, let's put the codegen feature to one side for now – go back to the Core Data editor, change Codegen back to "Manual/None", then press Cmd+S to save and Cmd+B to rebuild your app. You'll get compiler errors now because the `Commit` class no longer exists, but that's OK.

You've seen codegen, which is the first way Xcode can help us create objects in Core Data. The *second* way is to create our own custom `NSManagedObject` subclass, which right now is the preferred way forward because it lets us take the dynamically generated class and customize it.

Still inside the Core Data editor, go to the Editor menu and choose Create NSManagedObject Subclass. 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 "CommitsApp" rather than a blue project icon, and click Create.

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

```
import Foundation
import CoreData

extension Commit {
    @nonobjc public class func fetchRequest() -> NSFetchRequest<Commit> {
        return NSFetchRequest<Commit>(entityName: "Commit");
    }

    @NSManaged public var date: NSDate?
    @NSManaged public var message: String?
    @NSManaged public var sha: String?
    @NSManaged public var url: String?
}
```

First, there's that `@NSManaged` keyword we mentioned to you. Second, notice that the code says `extension Commit` rather than `class Commit`, which is Xcode being clever: Commit+CoreDataClass.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 `NSManagedObject` subclass, Xcode will overwrite only Commit+CoreDataProperties.swift, leaving your own changes in Commit.swift untouched.

There's one more thing in there, which is a bit of syntactic sugar. It's this bit:

```
@nonobjc public class func fetchRequest() -> NSFetchRequest<Commit> {
    return NSFetchRequest<Commit>(entityName: "Commit");
}
```

We'll come on fetch requests later on, but to satisfy your curiosity that line means we can write this:

```
Commit.fetchRequest()
```

Rather than this:

```
NSFetchRequest<Commit>(entityName: "Commit");
```

It's syntactic sugar – it's a piece of syntax that makes your code nicer.

This Core Data code generation is a head start, but not perfect: it has given us exactly what Xcode was dynamically generating before. Now, though, it's flattened to real Swift code, which means we can change it. You can see for yourself how Xcode is using `NSDate` for the `date` property, and how it's made everything optional.

Now that we have real Swift code to work with, we can go ahead and make changes. However, we should remind you that 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, so prepare yourself!

Working with optionals when they aren't needed adds an extra layer of annoyance, so we want you go ahead and remove all the question marks from Commit+CoreDataProperties.swift. We also want you to change the old `NSDate` data type to the shiny new `Date` data type.

I'd also like you to change the `fetchRequest()` syntactic sugar because it has an annoying flaw right now: it uses the same name as a different method that comes from `NSManagedObject`, and Xcode can't tell which one you mean. So, we prefer to rename it to `createFetchRequest()` to avoid the ambiguity. So, change the method to this:

```
@nonobjc public class func createFetchRequest() -> NSFetchRequest<Commit> {
    return NSFetchRequest<Commit>(entityName: "Commit");
}
```

When you're finished with these changes, it should look like this:

```
import Foundation
import CoreData

extension Commit {
    @nonobjc public class func createFetchRequest() -> NSFetchRequest<Commit>
{
        return NSFetchRequest<Commit>(entityName: "Commit");
    }

    @NSManaged public var date: Date
    @NSManaged public var message: String
    @NSManaged public var sha: String
    @NSManaged public var url: String
}
```

If you build your code now, it should work again because we have a `Commit` class again. Before we continue, delete the testing code we had in `viewDidLoad()` – it's time for real Core Data work!

Time for some useful code

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, we 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 `performSelector(inBackground:)` to call `fetchCommits()` – a method we haven't written yet. Put this just before the end of `viewDidLoad()`:

```
performSelector(inBackground: #selector(fetchCommits), with: nil)
```

What the new `fetchCommits()` method will do is very similar to what we did back in project 7: download the URL into an `Data` object then pass it to SwiftyJSON to convert into an array of objects. In project 10 we extended this to use GCD's `async()` method 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 SwiftyJSON 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 = try? Data(contentsOf: URL(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.")

        DispatchQueue.main.async { [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:

```
DispatchQueue.main.async { [unowned self] in
    for jsonCommit in jsonCommitArray {
        // the following three lines are new
        let commit = Commit(context: self.container.viewContext)
        self.configure(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, `configure(commit:)`, so to make your code build add this for now:

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

Now let's take a look at the two new lines of code above. First is `Commit(context: self.container.viewContext)`, which creates a `Commit` object inside the managed object context given to us by the `NSPersistentContainer` we created. This means its data will get saved back to the SQLite database when we call `saveContext()`.

Once we have a new `Commit` object, we pass it onto the `configure(commit:)` 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 `configure(commit:)` method straightforward if it were not for dates.

Yes, dates. Not the sweet fruity kind, but the `Date` 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 `Date` in order to put it inside our `Commit` object.

To convert "2016-01-26T19:46:18Z" into a `Date` we're going to use a new class called `ISO8601DateFormatter`. This is designed to convert `Date` objects to and from strings like "2016-01-26T19:46:18Z".

Before we show you the new `configure(commit:)` method, there's one more thing you need to know: getting an `Date` out of a string might fail, for example if the string isn't in ISO-8601 format. In this case, we'll get `nil` back, which isn't much good for our app, so we're going to use the nil coalescing operator to use a new `Date` instance if the date failed to parse.

Here's the new `configure(commit:)` method:

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

    let formatter = ISO8601DateFormatter()
    commit.date = formatter.date(from: json["commit"]["committer"]
["date"].stringValue) ?? Date()
}
```

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.

Step four is where we finally get to put to use all three previous steps by showing data to users.

In our project, we know we're using `Commit` objects to represent individual GitHub commits, so we need to store those objects in an array property. Add this to the `ViewController` class now:

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

We now need to write the usual table view methods, `numberOfRowsInSection` and `cellForRowAt`. The former will just return the size of the `commits` array, and the latter will place each commit's message and date into the cell's `textLabel` and `detailTextLabel`. There are lots of ways to convert dates to and from strings – you already saw `ISO8601DateFormatter`, for example – but here we're just going to use the simplest: every `Date` object has a `description` property that converts it to a human-readable string.

For a change, we're also going to write a third method that reports how many sections are in the table view. This returns 1 by default, and our new method will also return 1, but this will become useful later on.

Add these three methods now:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
    return commits.count
}

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Commit", for:
indexPath)

    let commit = commits[indexPath.row]
    cell.textLabel!.text = commit.message
    cell.detailTextLabel!.text = commit.date.description

    return cell
}
```

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 create one from the `NSManagedObject` subclass you're using for your entity, then pass it to managed object context's `fetch()` method. 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.

Time for some code:

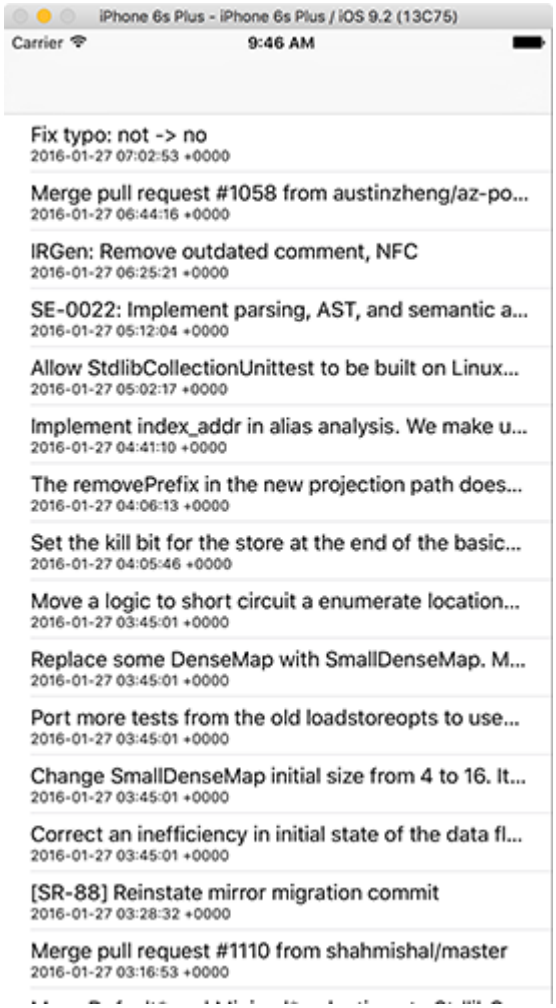
```
func loadSavedData() {
    let request = Commit.createFetchRequest()
    let sort = NSSortDescriptor(key: "date", ascending: false)
    request.sortDescriptors = [sort]

    do {
        commits = try container.viewContext.fetch(request)
        print("Got \(commits.count) 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 `fetch()` method to fetch the actual objects. That method returns an array of all `Commit` objects that exist in the data store. Once that's done, it's just a matter of calling `reloadData()` on the table to have the data appear.

To make the app work, we need to call this new `loadSavedData()` method in two places. First, add a call at the end of the `viewDidLoad()` method. Second, add a call in the `fechCommits()` 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!



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: first, you'll see GitHub commits get duplicated each time the app runs, and second you'll notice that tapping on a commit doesn't do anything.

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 we say that Core Data makes this trivial 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 fxed 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.

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.

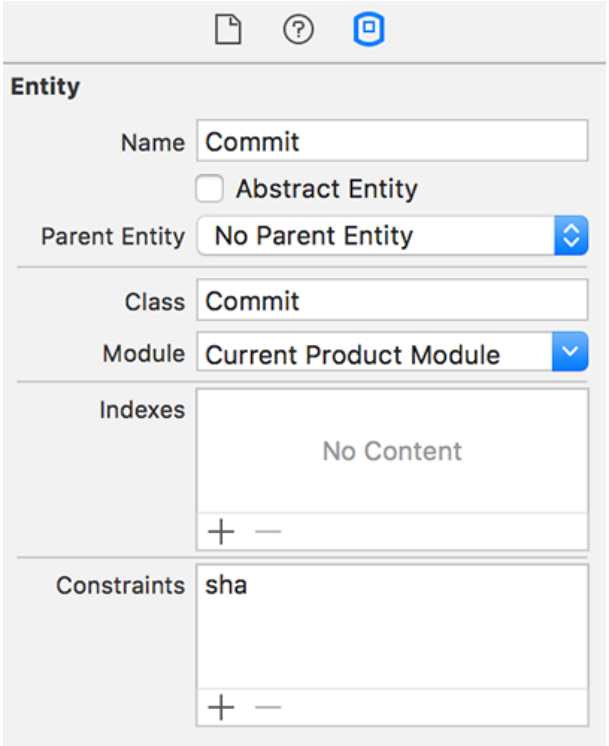
A "hash" is a little bit like like one-way, truncated encryption: one piece of input like "Hello world" will always generate the same hash, but if you change it to be "Hello World" – just capitalizing a single letter – you get a completely different hash. It's "truncated" because no matter how much content 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 (CommitsApp.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. Make sure you press Cmd+S to save your changes!



Now for the important part: go to the the iOS simulator, then choose the Simulator menu and choose Reset Content And Settings. What you just did was completely reset the state of the iOS Simulator. The reason this is required is because you just made an important change to your model, which is generally a bad idea unless you know what you're doing.

Before you run your project again, we want you to make one tiny code change. In your `viewDidLoad()` method, modify the `loadPersistentStores()` method call to this:

```
container.loadPersistentStores { storeDescription, error in
    self.container.viewContext.mergePolicy =
NSMergeByPropertyObjectTrumpMergePolicy

    if let error = error {
        print("Unresolved error \(error)")
    }
}
```

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: Using attribute constraints can cause problems with `NSFetchedResultsController`, but in this practice we're always doing a full save and load of our objects because it's an easy way to avoid problems later.

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 practice once you've learned a bit more.

First, add this new property to the `ViewController` 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:

```
request.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))
    present(ac, animated: true)
}
```

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 we 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)
{ [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, we're 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)
{ [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 `Date` 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)
{ [unowned self] _ in
    let twelveHoursAgo = Date().addingTimeInterval(-43200)
    self.commitPredicate = NSPredicate(format: "date > %@", twelveHoursAgo as
NSDate)
    self.loadSavedData()
})
```

As you can see, we've hit the `NSDate` vs `Date` problem again: Core Data wants to work with the old type, so we typecast using `as`. Once that's done, the magic `%@` will work with Core Data to ensure the `NSDate` 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:

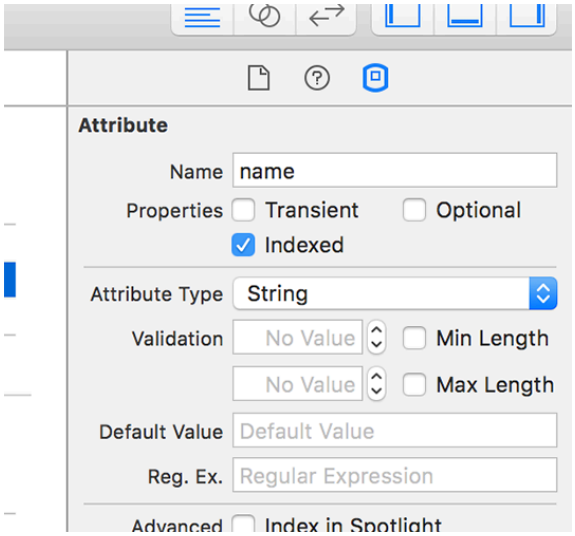
```
ac.addAction(UIAlertAction(title: "Show all commits", style: .default)
{ [unowned self] _ in
    self.commitPredicate = nil
    self.loadSavedData()
})
```

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.

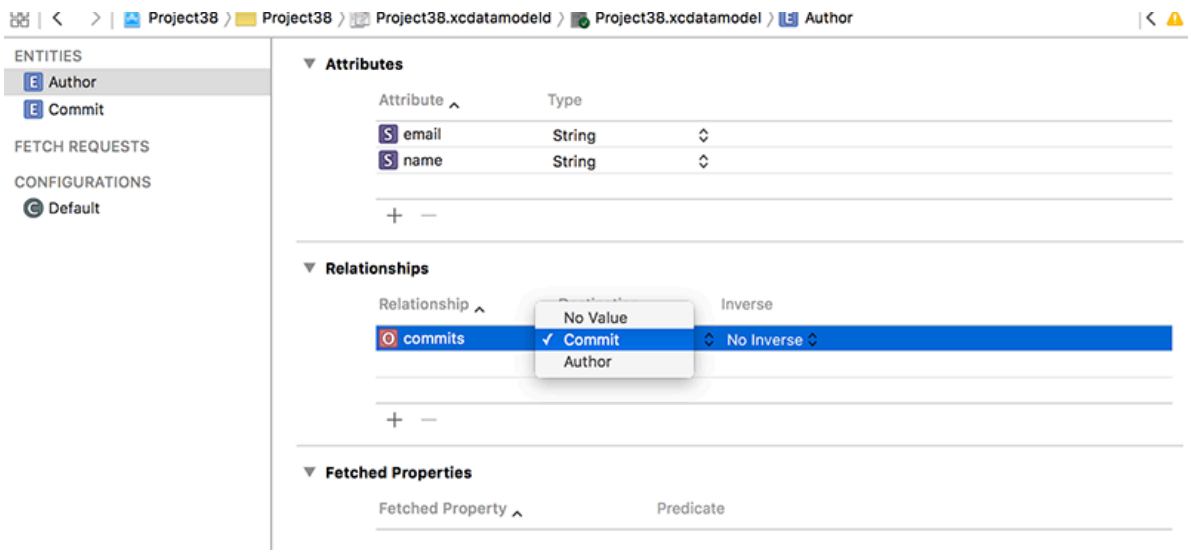
Open the data model (CommitsApp.xcdatamodeld) for editing, then click the Add Entity button. Name the entity Author, then give it two attributes: "name" and "email". Please make both strings, and make sure both are not marked as optional. This time we're also going to make one further change: select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



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 properties 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, so press Cmd+S to save then Cmd+R now to build and run the app. What you'll see is… well, exactly what you saw before: the same list of commits. What changed?

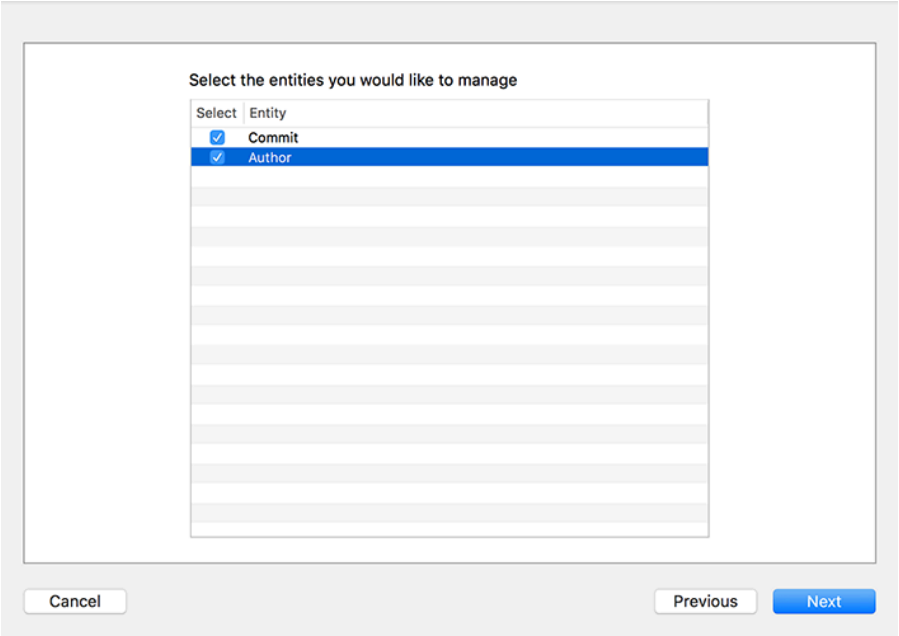
You've already seen how `NSPersistentContainer` does a huge amount of set up work on your behalf. Well, it's also doing something remarkably clever here too because we just changed our data model. By default Core Data doesn't know how to handle that – it considers any variation in its data model an unwelcome surprise, so 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.

The magic of `NSPersistentContainer` is that it automatically configures Core Data to perform a lightweight migration if it's needed and if it's possible – that is, if the changes are small enough to be figured out by the system. So, as long as your changes are strictly additive, `NSPersistentContainer` will take care of all the work. Awesome, right?

Of course, all this cleverness doesn't actually use our new Author entity. 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 re-add our custom changes such as removing optionality from its properties.

So, go back to the data model, and choose Editor > Create NSManagedObject Subclass again. This time we 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. We need to make a few changes to clean them up for use, starting with Commit+CoreDataProperties.swift:

1. Remove optionality from all five properties.
2. Change `NSDate` to `Date`.
3. Change `fetchRequest()` to `createFetchRequest()`.

Now in Author+CoreDataProperties.swift:

1. Remove optionality from all three properties.
2. Change `fetchRequest()` to `createFetchRequest()`.

Notice that Author+CoreDataProperties.swift includes some extra methods for adding and removing commits.

In order to attach authors to commits, we 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 create and configure a new author, and use that instead.

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

```
var commitAuthor: Author!

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

if let authors = try? container.viewContext.fetch(authorRequest) {
    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!
```

```
    let author = Author(context: container.viewContext)
    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 we used `try?` for `fetch()` 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 `cellForRowAt` method so that the detail text label contains the author name as well as the commit date, like this:

```
cell.detailTextLabel!.text = "By \(commit.author.name) on \
(commit.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.

Broadly speaking you don't want to make these kinds of model changes while you're still learning Core Data, so once you've verified that it works we would suggest you use "Reset Content and Settings" again in the simulator to make sure you have a clean foundation again.

We can also show that the inverse relationship works, so it's tie to make the detail view controller do something. Open DetailViewController.swift and give it this property:

```
var detailItem: Commit?
```

Now change its `viewDidLoad()` method to this:

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

    if let detail = self.detailItem {
        detailLabel.text = detail.message
        // navigationItem.rightBarButtonItem = UIBarButtonItem(title: "Commit
1/\(detail.author.commits.count)", style: .plain, target: self, action:
#selector(showAuthorCommits))
    }
}
```

I commented out one of the lines 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, we 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) { [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.

- 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 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, we 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.

To test out that change, we need to write the `didSelectRowAt` method so that it loads a detail view controller from the storyboard, assigns it the selected commit, then pushes it onto the navigation stack. Add this method to `ViewController`:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    if let vc = storyboard?.instantiateViewController(withIdentifier:
"Detail") as? DetailViewController {
        vc.detailItem = commits[indexPath.row]
        navigationController?.pushViewController(vc, animated: true)
    }
}
```

You should be able to run the app now.

***