

iOS 8

Day by Day

A review of iOS 8 for developers,
in 39 bite-size chunks.

by Sam Davies

iOS 8 Day by Day

Sam Davies

©2014 - 2015 Scott Logic Ltd

Also By Sam Davies

[iOS7 Day by Day](#)

[Bitesize Android KitKat](#)

for Pui

Contents

Preface	i
About this book	ii
What you'll get from this book	ii
What you won't get from this book	ii
How to read this book	iii
Conventions	iii
Project Code	iii
Day 1 :: Swift for Blaggers	1
Initialization	1
Mutability	2
Strong Typing and AnyObject	5
Protocol Conformance	6
Enums	7
Conclusion	8
Day 2 :: Sharing Extension	9
Creating a share extension	9
Validating user input	13
Uploading from within an extension	14
Conclusion	20
Day 3 :: UIVisualEffects	22
Blurring with UIVisualEffectView	22
Improving legibility with vibrancy	24
Performance Concerns	26
Conclusion	26
Day 4 :: Custom Fonts in Interface Builder	27
Font availability within Interface Builder	27
Declaring fonts for use within your app	29
Conclusion	30
Day 5 :: Auto-sizing table view cells	31

CONTENTS

Using the ‘stock’ table view cells	31
Creating custom table view cells	33
Conclusion	35
Day 6 :: Profiling Unit Tests	36
Measuring Test Performance	36
Moving average example	37
Improving the moving average	39
Conclusion	40
Day 7 :: Adaptive Layout and UITraitCollection	42
Adaptive Layout	42
Xcode Assistance	46
Conclusion	48
Day 8 :: Today Extension	49
Creating a widget	49
Sharing code with the parent app	51
Sharing a cache with the parent app	52
Navigating back to the parent app	54
Conclusion	56
Day 9 :: Designated Initializers	57
Creating objects	57
Usage in objective-C	60
Conclusion	60
Day 10 :: Xcode 6 Playgrounds	61
Interactive Coding & Timelines	62
Custom QuickLook	63
Custom View Development	66
Conclusion	69
Day 11 :: Asynchronous Testing	70
Testing an Asynchronous Method	70
Multiple Expectations	72
Key-Value Observation Expectation	73
Conclusion	74
Day 12 :: HealthKit	76
Data Structure Overview	76
Permissions	78
Writing Data	81
Reading Data	83
Conclusion	86

CONTENTS

Day 13 :: CoreImage Detectors	87
Detecting Rectangles	87
Detecting QR Codes	90
Conclusion	92
Day 14 :: Rotation Deprecation	93
Auto Layout to the rescue	93
Customizing rotation behavior	95
Conclusion	97
Day 15 :: NSFormatter	98
Temporal Formatters	98
Physical Quantity Formatters	100
Conclusion	104
Day 16 :: Navigation Bar Hiding	105
Navigation Controller Updates	105
Sample app	106
Conclusion	109
Day 17 :: Live Rendering in Interface Builder	110
@IBDesignable	111
@IBInspectable	112
Debugging Views	114
Conclusion	114
Day 18 :: UISplitViewController	116
Adaptive View Controller Hierarchy	116
Overriding Default Behavior	121
Advanced Features	122
Conclusion	124
Day 19 :: CoreImage Kernels	125
Filters and Kernels	125
Custom Kernel Types	126
General Kernels	130
Conclusion	133
Day 20 :: Photos Framework	134
Photo Library Outline	135
Querying For Models	135
Requesting Assets	135
Performing Model Updates	137
Registering for Update Notifications	138
Conclusion	140

CONTENTS

Day 21 :: Alerts and Popovers	141
Pop Overs	141
Alerts	145
ActionSheets	147
Conclusion	150
Day 22 :: Linking to Settings App	151
Linking to the Settings Page	151
Conclusion	154
Day 23 :: Photo Extension	155
Creating a Photo Extension	155
Starting Interactive Editing	157
Discard Changes?	159
Finalizing the Edit	160
Resumable Editing	162
Conclusion	164
Day 24 :: Presentation Controllers	165
The role of the Presentation Controller	165
Creating a custom Presentation Controller	166
Using the custom Presentation Controller	168
Adaptive UI with Presentation Controllers	172
Custom Presentation Animation	173
Conclusion	174
Day 25 :: Notification Actions	176
Requesting Permission	176
Registering Actions	179
Firing Actions	182
Handling Actions	183
Foreground Notifications	185
Conclusion	186
Day 26 :: AVKit	187
Using AVKit to play a video	187
Integration with Photos Framework	189
AVFoundation Pipeline	191
Conclusion	192
Day 27 :: Launch Images	194
Scaling the existing approach	194
Creating a launch screen XIB	196
Restrictions on Launch Screen XIBs	198

CONTENTS

Conclusion	198
Day 28 :: Document Picker	199
Conceptual Overview	199
Document Menu	200
Document Picker	202
Use on Simulator	204
Conclusion	205
Day 29 :: Safari Action Extension	206
Creating an Action Extension	206
Extracting Content from a Web Page	208
Interacting with JavaScript	211
Conclusion	214
Day 30 :: App Previews	215
Recording a Video	215
Video Considerations	217
Conclusion	218
Day 31 :: Using Touch ID to Secure the Keychain	219
Secure Enclave	220
Access Control Lists	221
Implementation	222
Conclusion	225
Day 32 :: Layout Margins	226
Layout Margins in Interface Builder	226
Layout Margins in Code	227
Preserving Superview Layout Margins	230
Conclusion	232
Day 33 :: CloudKit	233
High-level CloudKit Concepts	233
Enabling CloudKit	236
Creating Records	237
Querying For Records	243
Modifying Records	246
CloudKit Dashboard	248
Summary of other Features	250
Conclusion	251
Day 34 :: CoreLocation Authorization	253
Refresher on CoreLocation	254
New Methods on CoreLocationManager	256

CONTENTS

Providing Usage Strings	257
Conclusion	259
Day 35 :: CoreMotion	260
Motion Activity Data	260
Pedometer Data	263
Altimeter Data	265
Conclusion	267
Day 36 :: Location Notifications	268
Requesting Authorization	268
Creating Notifications	271
Responding to Notifications	272
Conclusion	273
Day 37 :: Autosizing Collection View Cells	274
Enabling Sizing in a Flow Layout	274
Autosizing via Interface Builder	277
Mechanics of Autosizing	278
Conclusion	278
Day 38 :: Handoff	279
Handoff Logistics	280
Preparing an App for Handoff	280
Resuming an Activity	282
Conclusion	286
Day 39 :: WatchKit	287
What can (and can't) I do on a watch?	287
Getting Started	288
Code sharing	299
Conclusion	300
Appendix	301
The road ahead	301
Useful resources	301

Preface

Apple's WorldWide Developer Conference (WWDC) of 2014 was held, as ever, in San Francisco in June. It was the first event that introduced a formal ticket lottery system, the previous year having sold out via the "internet connection lottery" in just two minutes. There were high hopes and expectations from the event - what had Apple been working on? What new coolness would be unleashed upon the developer world?

The headline-grabbing news was the introduction of Swift: a new language that represents the future of programming Apple devices. Understandably, the world got incredibly excited about this, and a plethora of experts in programming language design suddenly appeared out of nowhere. This spawned many blog posts, tutorials, books, talks and other content to allow developers to get a handle on Swift.

This response was great. However, I think it overshadowed the other announcements that formed iOS 8 (and OSX Yosemite). iOS 8 introduced many thousands of new APIs, with some fantastic new functionality, but got significantly less attention than it deserved since it wasn't the new, shiny thing in the room. iOS8 affects developers before they need to worry about Swift. There are changes and deprecations that require updates in existing apps, and there are new features that apps should take advantage of to stay fresh and relevant to their users.

The other issue that accompanies new iOS releases, is that learning about the new features is a time-consuming exercise. You could sit and watch the WWDC videos, but each is an hour long, and video is a terrible way to learn things. The excellent [ASCII WWDC¹](#) goes some way to easing this pain, with the text-transcripts, but it's still not the easiest approach. You could trawl through the documentation and the header files associated with the new and updated frameworks, but this also isn't an especially easy task. It's designed much more for solving problems, than about feature discovery.

This book started out life as a series of 39 blog posts, imaginatively titled "iOS8: Day-by-Day". The motivation behind the blog series was to cover the many great new features in iOS 8, at a "feature-discovery" level. Each post introduced a feature new to iOS 8, before running through the details you need to get started. Since every post has an accompanying project, it wasn't limited to regurgitating the documentation, but included experiences from actually using the new frameworks - common pitfalls, and areas not covered by the documentation.

This book represents the collection of these 39 articles in a handy book format. It was a lot of fun to write, and I hope you find some of the chapters useful to you.

sam

[@iwantmyrealname²](#)

¹<http://asciwwdc.com>

²<https://twitter.com/iwantmyrealname>

About this book

iOS8: Day-by-Day is a book born out of a collection of articles posted on the ShinobiControls blog at shinobicontrols.com/iOS8DayByDay³. As such, each chapter is entirely self-contained, and can be read completely independently. It is not aimed at people who are new to iOS or programming. There are great resources for this. Go and read them first, and then come back to this to learn about all the great new things you can do in the latest version of iOS. See you soon!

What you'll get from this book

- Short articles highlighting the key features of iOS 8, and how to use them.
- A full project with source code demonstrating how to use each feature. Every chapter has an app or playground that features the new technology. This allows you to see it “in the wild”, not just “on paper”, as it is in the documentation.
- “Lessons learnt” from creating the app. Often these parts can be more valuable than the tech outline itself. This includes problems encountered and their associated workarounds.
- A great understanding of how to update your existing apps to ensure compatibility with iOS 8.
- Hours of fun. Well, maybe...

What you won't get from this book

- An introduction into how to develop your first iOS app. There are loads of great resources for this.
- An introduction or exploration of finer points in Swift. Chapter 1 will cover some of the headline issues, but the rest is left to the reader.
- A deep-dive into every new feature/framework in iOS 8. This book is designed to be enough to make you aware of what features are available, and how to get started with them. Once you've got to that point, you'll be at home with the documentation.
- Recantation of the documentation. The book is very much designed to be supplementary to the material provided by Apple.

³<http://shinobicontrols.com/iOS8DayByDay>

How to read this book

As mentioned, this book is a collection of articles originally posted on a blog. Therefore each chapter stands on its own, and there is little importance placed in their order. You can pick and choose your way through just the chapters that interest you - treating the book entirely as a reference.

However, since the content makes up a comprehensive review of the new technologies introduced in iOS 8, reading the book in a more traditional, linear manner is perfectly valid. This is likely to result in you gaining a better understanding of iOS 8 as a whole, but at the cost of learning about things you will probably never use.

Conventions

This book is not a tutorial on Swift. In fact, the only time that Swift itself is discussed is in chapter 1 (*Swift for Blaggers*). However, nearly all the code is written in Swift. This means that you'll need to have a reasonable understanding of Swift in order to follow the code. Don't be put off by this if you're not well-versed in Swift - it's not nearly as difficult as you might expect.

All the code is written in such a way that it doesn't rely on any Swift-only features, so translation back into Objective-C should be a painless process.

Some of the chapters are relevant to OSX as well as iOS, but wherever possible, the focus is very much on iOS. There is one exception to this (in chapter 10 *Playgrounds*), but this is noted within the chapter.

The code might not turn out to be "best practice" in Swift, but was written well before any best practices had been established. However, it should all be pretty good code, and easy to follow.

Some error handling and the suchlike might have been skipped over in the interests of brevity and keeping 'on-message'. This is an arguably sloppy practice, but error handling outside of a well-defined application context is not always especially clear. It's left as an exercise to the reader.

Project Code

All of the code for the projects that accompany this book is open source - and available on github at github.com/ShinobiControls/ios8-day-by-day⁴. You can go there to download, clone or fork all 39 chapters' worth of code.

If you find any problems then please do create a pull-request so that the code is kept up to date and working for future readers. If there are significant changes then the book will be updated as well.

The code is licensed under the Apache 2.0 license.

⁴<https://github.com/ShinobiControls/ios8-day-by-day>

Day 1 :: Swift for Blaggers



It won't have gone unnoticed that at WWDC this year, in addition to announcing iOS 8, they also introduced a new programming language in the form of Swift. This is quite a different language from objective-C in that it is strongly-typed and includes some features common to more modern languages.

In the interests of embracing anything and everything that's new and shiny, this book will exclusively use Swift. There is a wealth of information out there about how to learn Swift, and how to interact with the Cocoa libraries - in fact you can't go wrong with starting out by reading through the official books:

- [The Swift Programming Language⁵](#)
- [Using Swift with Cocoa and Objective-C⁶](#)



You should also check out the official [swift blog⁷](#), and some of the other [resources⁸](#) made available by Apple.

Since there is so much good info out there about how to use Swift, this chapter is not going to attempt to cover any of that. Instead, it's going to run through some of the important gotchas and potential pain points when using Swift for the first time - especially when relating to the system frameworks.

There is an Xcode 6 playground which accompanies this chapter - including short samples for each of the sections. You can get hold of it at [github.com/ShinobiControls/iOS8-day-by-day⁹](#).

Initialization

Swift formalizes the concepts surround initialization of objects somewhat - including designated -vs- convenience initializers, and sets a very specific order of the operations to be called within the initialization phases of an object. In the coming weeks, there will be an article as part of this series which will go into detail about how initialization works in Swift, and how this affects any objective-C that you write - so look out for this.

There is one other fairly major difference in initialization between Swift and objective-C, and that is return values and initialization failure. In objective-C an initializer looks a lot like this:

⁵<https://itunes.apple.com/us/book/the-swift-programming-language/id881256329>

⁶<https://itunes.apple.com/us/book/using-swift-cocoa-objective/id888894773>

⁷<https://developer.apple.com/swift/blog/>

⁸<https://developer.apple.com/swift/resources/>

⁹<https://github.com/ShinobiControls/iOS8-day-by-day>

```

1 - (instancetype)init {
2     self = [super init];
3     if (self) {
4         // Do some stuff
5     }
6     return self;
7 }
```

Whereas in Swift:

```

1 init {
2     variableA = 10
3     ...
4     super.init()
5 }
```

Notice that in objective-C the initializer is responsible for ‘creating’ and then returning `self`, but there is no `return` statement in the Swift equivalent. This means that there is actually no way in which you can return a `nil` object, which is a pattern commonly used to indicate an initialization failure in objC.

This is apparently likely to change in an upcoming release of the language, but for now the only workaround is to use class methods which return optional types:

```

1 class MyClass {
2     class func myFactoryMethod() -> MyClass? {
3         ...
4     }
5 }
```

Interestingly, factory methods on objective-C APIs are converted into initializers in Swift, so this approach is not preferred. However, until language support arrives, it’s the only option for initializers which have the potential to fail.

Mutability

The concept of (im)mutability is not new to Cocoa developers - we’ve been used to using `NSArray` and its mutable counterpart `NSMutableArray` where appropriate, and even understand that we should always prefer the immutable version wherever possible. Swift takes this concept to the next level, and bakes immutability into the language as a fundamental concept.

The `let` keyword defines an immutable variable, which means that you can’t change what it represents. For example:

```

1 let a = MyClass()
2 a = MySecondClass() // Not allowed

```

This means that you can't redefine something specified with the `let` keyword. Depending on the type of the object referred to, it might itself be immutable too. If it is a value type (such as a struct) then it will also be immutable. If it is a reference type, such as a class then it will be mutable.

To see this in action, consider the following struct:

```

1 struct MyStruct {
2     let t = 12
3     var u: String
4 }

```

If you define a variable `struct1` with the `var` keyword then you get the following behavior:

```

1 var struct1 = MyStruct(t: 15, u: "Hello")
2 struct1.t = 13 // Error: t is an immutable property
3 struct1.u = "GoodBye"
4 struct1 = MyStruct(t: 10, u: "You")

```

You can mutate the `u` property, since this is defined with `var`, and you can redefine the `struct1` variable itself, again because this is defined with `var`. You can't mutate the `t` property, since this is defined with `let`. Now take a look what happens when you define an instance of a struct using `let`:

```

1 let struct2 = MyStruct(t: 12, u: "World")
2 struct2.u = "Planet" // Error: struct2 is immutable
3 struct2 = MyStruct(t: 10, u: "Defeat") // Error: struct2 is an immutable ref

```

Here, not only are you unable to mutate the `struct2` reference itself, but you are also unable to mutate the struct itself (i.e. the `u` property). This is because a struct is a **value type**.

The behavior is subtly different with a class:

```

1 class MyClass {
2     let t = 12
3     var u: String
4
5     init(t: Int, u: String) {
6         self.t = t
7         self.u = u
8     }
9 }
```

Defining a variable using `var` gives behavior you might be used to from objective-C:

```

1 var class1 = MyClass(t: 15, u: "Hello")
2 class1.t = 13 // Error: t is an immutable property
3 class1.u = "GoodBye"
4 class1 = MyClass(t: 10, u: "You")
```

You can mutate both the reference itself, and any properties defined using `var`, but you are unable to mutate any properties defined with `let`. Compare this to the behavior when the instance is defined with `let`:

```

1 let class2 = MyClass(t: 12, u: "World")
2 class2.u = "Planet" // No error
3 class2 = MyClass(t: 11, u: "Geoid") Error: class2 is an immutable reference
```

Here you are unable to mutate the reference itself, but you **can** still mutate any properties defined with `var` within the class. This is because a class is a **reference type**.

This behavior is fairly easy to understand, and is well-explained in the language reference books. There is potential for confusion when looking at Swift collection types though.

An `NSArray` is a reference type. That is to say that when you create an instance of `NSArray`, you create an object and your variable is a pointer to the location of the array itself in memory - hence the asterisk in the objective-C definition. If you take a look back over what you've learnt about the semantics of reference and value types with respect to `let` and `var` then you can probably work out how they would behave. In fact, if you want a mutable version of an `NSArray` you have to use a different class - in the shape of `NSMutableArray`.

Swift arrays aren't like this - they are value types instead of reference types. This means that they behave like a struct, not a class. Therefore, the `let` or `var` keyword not only specifies whether or not the variable can be redefined, but also whether or not the created array is mutable.

An array defined with `var` can both be reassigned, and mutated:

```

1 var array1 = [1,2,3,4]
2 array1.append(5)           // [1,2,3,4,5]
3 array1[0] = 27            // [27,2,3,4,5]
4 array1 = [3,2]             // [3,2]

```

But an array defined with `let` can be neither:

```

1 let array2 = [4,3,2,1]
2 array2.append(0) // Error: array2 is immutable
3 array2[2] = 36  // Error: array2 is immutable
4 array2 = [5,6]   // Error: cannot reassign an immutable reference

```

This is an area with a huge potential for confusion. Not only does it completely change the way we think about mutability for collections, but it also mixes up two previously distinct concepts. There is potential that this might be changed in a future release of the language - so keep an eye on the language definition.

A corollary of this is that since arrays are value types, they are passed by copy. `NSArray` instances are always passed by reference - so a method which takes an `NSArray` pointer will point to exactly the same chunk of memory. If you pass a Swift array into a method, it will receive a copy of that array. Depending on the type of the objects stored in that array this could either be a deep, or a shallow copy. Be aware of this whilst writing your code!

Strong Typing and AnyObject

Strong typing is seen as a great feature of Swift - it can allow for safer code, since what in objective-C would have been runtime exceptions can now be caught at compile time.

This is great, but as you're working with the objective-C system frameworks you'll notice a lot of this `AnyObject` type. This is the Swift equivalent of objective-C's `id`. In many respects, `AnyObject` feels rather un-Swift-like. It allows you to call `any` methods it can find on it, but these will result in a run-time exception. In fact, it behaves *almost* exactly the same as `id` in objective-C. The difference is that properties and methods which take no arguments will return `nil` if that method/property doesn't exist on the `AnyObject`:

```

1 let myString: AnyObject = "hello"
2 myString.cornerRadius // Returns nil

```

In order to work in a more Swift-like way with the Cocoa APIs, you'll see the following pattern a lot:

```

1 func someFunc(parameter: AnyObject!) -> AnyObject! {
2     if let castedParameter = parameter as? NSString {
3         // Now I know I have a string :)
4         ...
5     }
6 }
```

If you know that you've definitely been passed a string, you don't necessarily need to guard around the cast:

```
1 let castedParameter = parameter as NSString
```

A top-tip is to realize that casting arrays is really easy too. All arrays that you'll receive from a Cocoa framework will be of the type `[AnyObject]`, since `NSArray` doesn't support generics. However, in nearly every case not only are all the elements of the same type, but they are of a known type. You can cast an entire array in both the conditional and unconditional ways expressed above, with the following syntax:

```

1 func someArrayFunc(parameter: [AnyObject]!) {
2     let newArray = parameter as [String]
3     // Do something with your strings :)
4 }
```

Protocol Conformance

Protocols are well-understood in Swift - defined as follows:

```

1 protocol MyProtocol {
2     func myProtocolMethod() -> Bool
3 }
```

One of the things you often want to do is test whether an object conforms to a specified protocol, which you could do as follows:

```

1 if let class1AsMyProtocol = class1 as? MyProtocol {
2     // We're in
3 }
```

However, this will have an error, because in order to check conformance of a protocol that protocol must be an objective-C protocol - and annotated with `@objc`:

```
1 @objc protocol MyNewProtocol {
2     func myProtocolMethod() -> Bool
3 }
4
5 if let class1AsMyNewProtocol = class1 as? MyNewProtocol {
6     // We're in
7 }
```

This can actually be more effort than you'd expect, since in order that a protocol be labeled as `@objc`, all of its properties and method return types must also be understood in the objective-C world. This means that you might end up annotating loads of classes you thought you only cared about in Swift with `@objc`.

Enums

Enums in Swift have become super-charged. Not only can an enum now have associated values (which needn't be of the same type), but also contain functions too.

```
1 enum MyEnum {
2     case FirstType
3     case IntType (Int)
4     case StringType (String)
5     case TupleType (Int, String)
6
7     func prettyFormat() -> String {
8         switch self {
9             case .FirstType:
10                 return "No params"
11             case .IntType(let value), .StringType(let value):
12                 return "One param: \(value)"
13             case .TupleType(let v1, let v2):
14                 return "Some params: \(v1), \(v2)"
15             default:
16                 return "Nothing to see here"
17         }
18     }
19 }
```

This is really powerful - use it as follows:

```
1 var enum1 = MyEnum.FirstType
2 enum1.prettyFormat() // "No params"
3 enum1 = .TupleType(12, "Hello")
4 enum1.prettyFormat() // "Some params: 12, Hello"
```

It'll take a little practice to see where you can get some benefit out of the power of these, but just as an indication of what you can achieve - the optionals system within Swift is built out of enumerations.

Conclusion

Swift is really powerful - and it's going to take us a while to establish best practice and work out what idioms and patterns we can now use that weren't possible within the constraints of objective-C. This article has outlined some of the common areas of confusion when moving from objective-C to Swift, but don't let this put you off. All of the projects associated with this book are written using Swift, and on the most-part are really simple to understand.

There is a playground which contains some of the samples mentioned in this chapter - it's part of the Github repo which accompanies the book. You can get it at github.com/ShinobiControls/iOS8-day-by-day¹⁰.

¹⁰<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 2 :: Sharing Extension



A huge new feature in iOS 8 is the introduction of support for extensions. These are areas in which developers can enhance the operating system, including both 3rd-party apps and Apple-provided apps, with their own features and functionality. There are 6 extensions applicable to iOS:

- Today screen widgets
- Share extensions
- Actions
- Photo editing
- Storage providers
- Custom keyboards

We'll cover some of these in later chapters of iOS 8 Day-by-Day, but today's article is focused on sharing extensions.

Sharing extensions give you, the developer of an app, to show an icon on the common share-sheet, and then to handle the sharing of the content the user has requested. This means that you can supplement the list of possibilities present in the OS (e.g. Twitter, Flickr, Sina Weibo).

A word of warning: this topic is not an easy one. From the mere nature of an extension it is pretty complex. The article will take you through some of the most common use cases, but be aware that you can pretty much build your own visual appearance. Apple has some excellent resources in this area, so it's worth having a read through them if you get stuck with anything.

The sample app for today's chapter is called "ShareAlike", and demonstrates building a sharing extension which allows sharing an image and some text. The code is available on Github at [github.com/ShinobiControls/iOS8-day-by-day¹¹](https://github.com/ShinobiControls/iOS8-day-by-day).

Creating a share extension

Extensions are compiled binaries which have to be contained within an app - it's not possible to add an extension to the system without an accompanying app. Xcode has a template for adding an extension to an existing app - it'll set up a new module within your project and provide you with the required files to get started.

¹¹<https://github.com/ShinobiControls/iOS8-day-by-day>

The major part of a sharing extension is its visual appearance, and as such you're provided with a subclass of `SLComposeServiceViewController` and a storyboard. The default appearance of `SLComposeServiceViewController` gives you a lot of sensible behavior (including a character count, image display, text entry, post and cancel buttons) and fits in with the iOS UI. In this example we're going to stick with this default behavior.

In addition to the standard `UIViewController` methods, `SLComposeServiceViewController` has some methods and properties associated with the life-cycle of a share-sheet composition view:

- `presentationAnimationDidFinish()` this provides a great hook to perform any heavy-lifting tasks. We'll use this to extract the image to share.
- `contentText` a string which represents what the user has typed into the composer
- `didSelectPost()` called when the **Post** button is tapped. The point to kick off the upload task.
- `didSelectCancel()` called when the **Cancel** button is tapped.
- `isContentValid()` called every time the content in the compose view changes.
- `charactersRemaining` is a number which appears on the compose sheet. When negative its appearance becomes red

Simply adding an extension to your app's project will allow it to be selected when a user requests to share something. We'll take a look at implementing some useful functionality soon, but first, let's learn a little about how to build, run and debug.

Building, running and debugging



In production versions of Xcode 6, the issues mentioned in this section have been resolved.
The section is kept in the app for informational purposes only.

In theory, you should be able to select the scheme associated with the extension and hit run. You'll then be asked which host app you'd like to debug it in, and you'll be away. This experience can be more than a little flakey. The only apps it is possible to choose (as of Xcode 6b3) are your own development apps - not system apps. This seems a little strange - the perfect app for testing sharing would be the photos app.

However, the following approach seems to be reliable:

1. Either use your extension's host app, or add a new app to the same project which has easily available content for sharing. In the sample project, the **ShareAlike** host app has an image and a share button which will trigger a standard UI share sheet:

```
1  @IBAction func handleShareSampleTapped(sender: AnyObject) {
2      shareContent(sharingText: "Highland Cow", sharingImage: sharingContentImageView.image)
3  }
4
5
6  // Utility methods
7  func shareContent(#sharingText: String?, sharingImage: UIImage?) {
8      var itemsToShare = [AnyObject]()
9
10     if let text = sharingText {
11         itemsToShare.append(text)
12     }
13     if let image = sharingImage {
14         itemsToShare.append(image)
15     }
16
17     let activityViewController = UIActivityViewController(activityItems: itemsToShare,
18     applicationActivities: nil)
19     presentViewController(activityViewController, animated: true, completion: nil)
20 }
```

Carrier 4:46 PM

ShareAlike

[Share Sample](#)



ShareAlike

2. Develop your extension
3. To debug and or test, run the scheme associated with your app. Then when you hit share then the debugger will automatically attach to the process associated with the extension.

4. You can use the sharing extension from other apps on the simulator, but you won't have debugging access. It's good to get it to a working state and then test it within the Photos app.

Specifying what can be shared

In the same way that the settings associated with an app are contained within an **Info.plist** file, there is an equivalent file for the extension module. One of the things you can control is the name of the extension which appears under the icon in the share sheet.

The name is defined by the **Bundle display name** (`CFBundleDisplayName`):

Localization	Native	Development	...	String	en
				Bundle display name	ShareAlike
				Executable file	<code>NSString stringWithFormat:@"%@.app", \${EXECUTABLE_NAME}</code>

Bundle Display Name

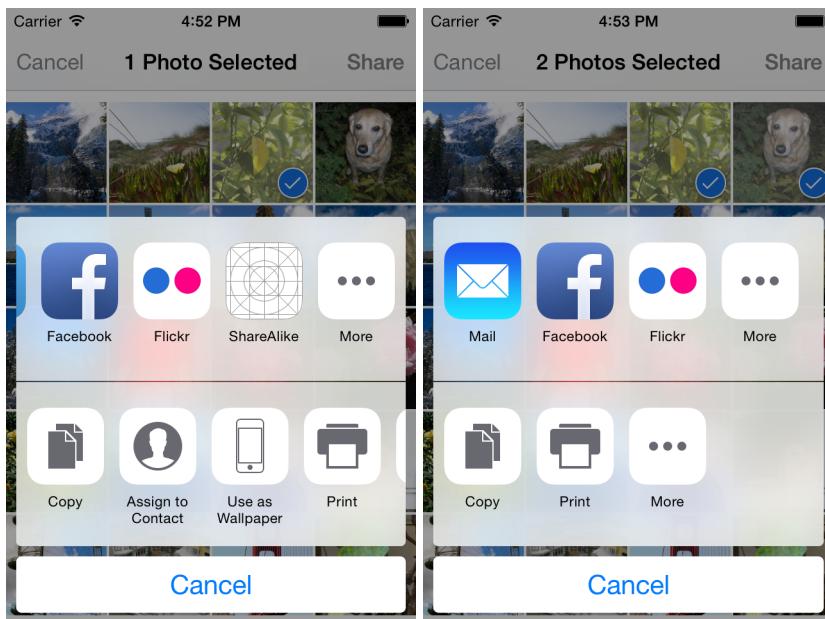
You can also use the **Info.plist** file to define under which circumstances your share extension is applicable - e.g. can it handle videos? There is a very powerful string-predicate language you can use here, but for 99% of cases, using a dictionary will suffice.

Inside the `NSExtension` dictionary, there is a `NSExtensionAttributes` dictionary, one of the values of which is `NSExtensionActivationRule`. This can be a boolean, a string or a dictionary. The following shows a dictionary with settings which will enable a single image to be shared, disables videos, files and URLs

▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(3 items)
▼ NSExtensionActivationRule	Dictionary	(6 items)
NSExtensionActivationSupportsFileWithMaxCount	Number	0
NSExtensionActivationSupportsImageWithMaxCount	Number	1
NSExtensionActivationSupportsText	Boolean	YES
NSExtensionActivationSupportsVideoWithMaxCount	Number	0
NSExtensionActivationSupportsWebPageWithMaxCount	Number	0
NSExtensionActivationSupportsWebURLWithMaxCount	Number	0
NSExtensionPointName	String	com.apple.share-services
NSExtensionPointVersion	String	1.0

Activation Rules

The different dictionary keys are pretty self-explanatory, and the result can be seen here; selecting one image to share shows **ShareAlike** as an option, whereas two images does not:



Validating user input

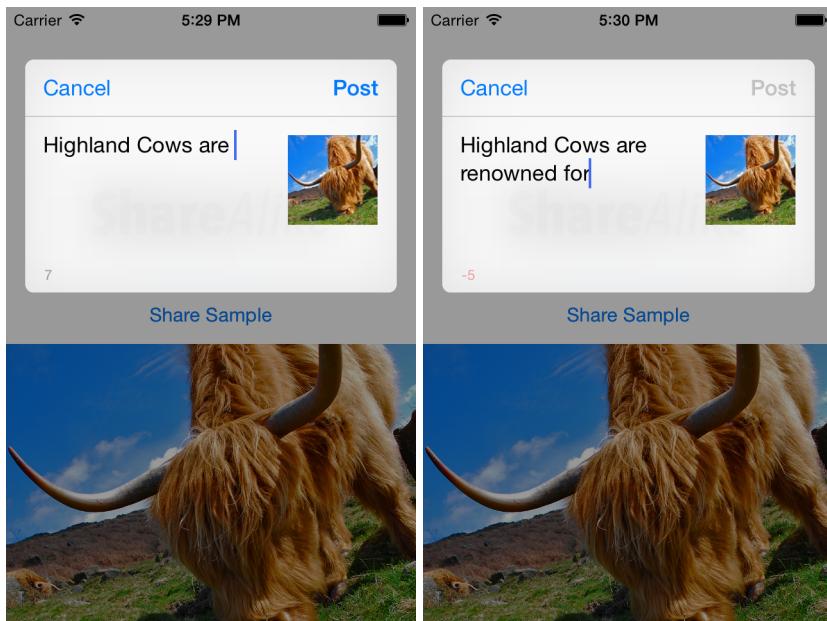
Now that you've got an understanding of how to create extensions and control them, let's take a look at implementing some custom behavior. First up is how you can validate input provided by the user. One of the most common things you might want to do is to limit the number of characters in text string that the user enters, and `SLComposeServiceViewController` has create support for this.

As was mentioned in the in the overview of sharing extensions, there is a method called `isContentValid() -> Bool` which will get called every time that the content of the text field changes . Returning true states that you're happy with the current content, and therefore the **Post** button should be enabled. The button will be disabled if you return false. The following implementation will perform a simple character limit:

```

1 let sc_maxCharactersAllowed = 25
2
3 override func isContentValid() -> Bool {
4     if let currentMessage = contentText {
5         let currentMessageLength = countElements(currentMessage)
6         charactersRemaining = sc_maxCharactersAllowed - currentMessageLength
7
8         if Int(charactersRemaining) < 0 {
9             return false
10        }
11    }
12    return true
13 }
```

The `contentText` property is a `String` which contains the current content of the compose view. Here you're finding it's length using `countElements()` (note that you can't ask a `String` for its length). This approach is always $O(N)$. `charactersRemaining` is a numeric property, which is represented on the UI. You then determine whether you have reached the limit or not, and enable/disable the `Post` button appropriately.



Uploading from within an extension

So far you've learnt about how to create an extension, how to configure it and how to control some of the user experience, but the main purpose of a share extension is to upload the content to some kind of network end-point. Let's take a look at that.

Extensions are meant to be much more lightweight than full-blown apps, so you need to keep resource usage down. You can think of an extension as being invited by the host app to perform some simple operation - it would be rude to suddenly attempt to steal the limited system resources from your host. Therefore all uploading needs to happen as a background process (helpfully introduced in `NSURLSession` in iOS7). You're probably thinking that this'll be easy - you can just refer to the relevant chapter in [iOS 7 Day-by-Day¹²](#) and you'll be done. Well, that's not quite true.

There are a couple of complications with this. First of all, it's not trivial to extract the content (i.e. image) you've been asked to share, and secondly, an extension doesn't get provided any writable disc access. This might seem a little strange - why would an extension *need* disc access? Well, it's all to do with the background network process - at the point it is called, it will cache the data on disc and then start the upload in the background. In order that this can happen, you need to create

¹²<https://leanpub.com/ios7daybyday>

a shared container within your host app, and allow the extension to use it for caching. We'll take a look at how to do this in a minute, but first, let's take a look at how to get hold of the image.

Extracting an image to upload

There is a property on `SLComposeServiceViewController` called `extensionContext`, and this contains all the data that's associated with this instance of the extension, including an array of `NSInputItem` objects, called `inputItems`. An `NSInputItem` has a collection of attachments, each of type `NSItemProvider`. Each of these attachments represents an item of media - such as an image, a video, a file or a link.

```
1 func imageFromExtensionItem(extensionItem: NSExtensionItem,
2                               callback: (image: UIImage?) -> Void) {
3     for attachment in extensionItem.attachments as [NSItemProvider] {
4         ...
5     }
6 }
```

This is a function which is going to extract a `UIImage` from an extension item. Notice that this function does not have a return type, but instead returns the result via a callback closure.

In order to determine whether an attachment contains a media of a particular type, you need to use the `hasItemConformingToTypeIdentifier()` method.

```
1 if(attachment.hasItemConformingToTypeIdentifier(kUTTypeImage as NSString)) {
2     ...
3 }
```

The type identifier is a string and is part of the framework. In order to get it imported, you'll need to import the `MobileCoreServices` module at the top of your Swift file:

```
1 import MobileCoreServices
```

There are many type identifiers defined, from incredibly general, to very specific, including the following:

- `kUTTypeImage`
- `kUTTypeMovie`
- `kUTTypeAudio`
- `kUTTypeSpreadsheet`
- etc

Now that you're sure you have an attachment which includes an image, you need to extract it. Since this could be an expensive process, it should be performed on a background queue to ensure that the UI doesn't hang. This uses the `loadItemForTypeIdentifier()` method, which takes the same type as before, and a closure to which the result will be delivered:

```

1 // Marshal on to a background thread
2 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)) {
3     attachment.loadItemForTypeIdentifier(kUTTypeImage as NSString, options: nil) {
4         (imageProvider, error) -> Void in
5         ...
6     }
7 }
```

If you're using objective-C then you can use the types of the block parameters to coerce the result into the type you expect (i.e. into `UIImage`), but this isn't possible using Swift. Therefore, the `imageProvider` variable will be of type `NSSecureCoding`. You can then cast this to `NSData`, and create an image from this:

```

1 var image: UIImage? = nil
2 if let e = error {
3     println("Item loading error: \(e.localizedDescription)")
4 }
5 image = imageProvider as? UIImage
6 dispatch_async(dispatch_get_main_queue()) {
7     callback(image: image)
8 }
```

Notice that at the end of this, there is a `callback()` with the created `image` object.

This process of extracting the image from the attachments can be kicked off whilst the user is entering their text - after the view has finished animating into view. There is a perfect method to for this:

```

1 var attachedImage: UIImage?
2
3 override func presentationAnimationDidFinish() {
4     // Only interested in the first item
5     let extensionItem = extensionContext?.inputItems[0] as NSExtensionItem
6     // Extract an image (if one exists)
7     imageFromExtensionItem(extensionItem) {
8         image in
9         if image {
10             dispatch_async(dispatch_get_main_queue()) {
11                 self.attachedImage = image
12             }
13         }
14     }
15 }
```

This method asks for the image to be extracted, and in the callback it saves it off into the instance property `attachedImage`.

Performing a background upload

Once the user has completed their entry, and clicks the **Post** button, then the extension should upload the content to some web service somewhere. For the purposes of this example, the URL of the endpoint is contained within a property on the view controller:

```
1 let sc_uploadURL = "http://requestb.in/oha28noh"
```



This is a URL for the Request Bin service, which gives you a temporary URL to allow you to test network operations. The above URL (and the one in the sample code) won't work for you, but if you visit [requestb.in¹³](http://requestb.in) then you can get hold of your own URL for testing.

As mentioned previously, it's important that extensions put very little strain on the limited system resources. Therefore, at the point the **Post** button is tapped, there is no time to perform a synchronous, foreground network operation. Luckily, `NSURLSession` provides a simple API for creating background network operations, and that's what you'll need here.

The method which gets called when the user taps post is `didSelectPost()`, and in its simplest form it must look like this:

```
1 override func didSelectPost() {
2     // Perform upload
3     ...
4
5     // Inform the host that we're done, so it un-blocks its UI.
6     extensionContext?.completeRequestReturningItems(nil, completionHandler: nil)
7 }
```

Setting up an `NSURLSession` is pretty standard:

¹³[http://requestb.in/](http://requestb.in)

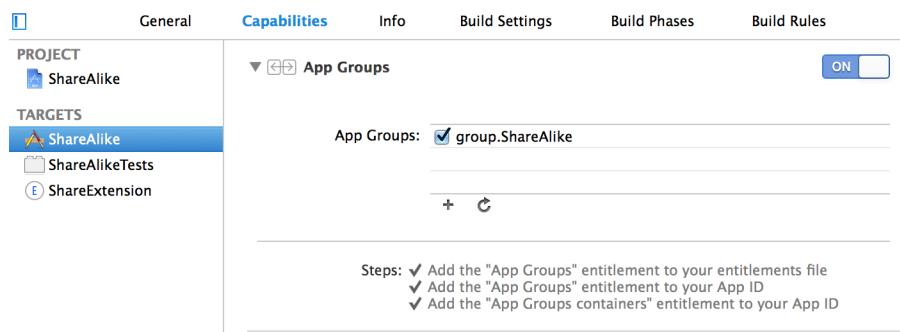
```

1 let configName = "com.shinobicontrols.ShareAlike.BackgroundSessionConfig"
2 let sessionConfig = NSURLSessionConfiguration.
3         backgroundSessionConfigurationWithIdentifier(configName)
4 // Extensions aren't allowed their own cache disk space - share with application
5 sessionConfig.sharedContainerIdentifier = "group.ShareAlike"
6 let session = NSURLSession(configuration: sessionConfig)

```

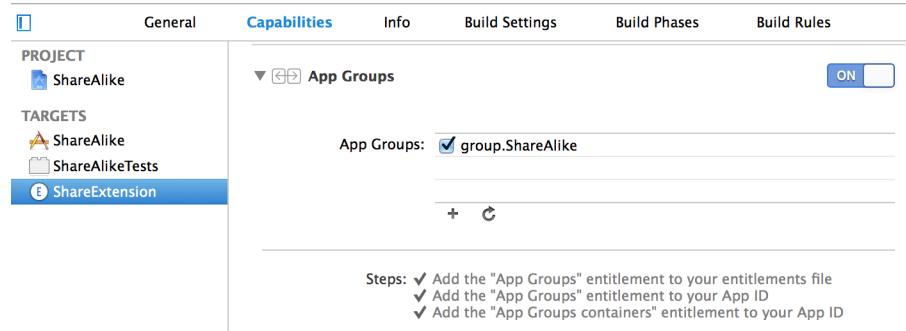
The important part to note of the above code segment is the line which sets the `sharedContainerIdentifier` on the session configuration. This specifies the name of the container that `NSURLSession` can use as a cache (since extensions don't have their own writable disc access). This container needs to be set up as part of the host application (i.e. `ShareAlike` in this demo), and can be done through Xcode:

1. Go to the capabilities tab of the app's target
2. Enable App Groups
3. Create a new app group, entitled something appropriate. It must start with group.. In the demo the group is called `group.ShareAlike`.
4. Let Xcode go through the process of creating this group for you.



App group for host app

Then you need to go to the extension's target, and follow the same process. Note that you won't need to create a new app group, but instead select the one that you created for your host application.



App group for extension

These app groups are registered against your developer ID, and the signing process ensures that only your apps are able to access these shared containers.

Xcode will have created an entitlements file for each of your projects, and this will contain the name of the shared container it has access to.

Now that you've got your session set up correctly, you need to create a URL request to perform:

```
1 // Prepare the URL Request
2 let request = urlRequestWithImage(attachedImage, text: contentText)
```

This calls a method which constructs a URL request which uses HTTP POST to send some JSON, which includes the string content, and some metadata properties about the image:

```
1 func urlRequestWithImage(image: UIImage?, text: String) -> NSURLRequest? {
2     let url = NSURL.URLWithString(sc_uploadURL)
3     let request = NSMutableURLRequest(URL: url)
4     request.addValue("application/json", forHTTPHeaderField: "Content-Type")
5     request.addValue("application/json", forHTTPHeaderField: "Accept")
6     request.HTTPMethod = "POST"
7
8     var jsonObject = NSMutableDictionary()
9     jsonObject["text"] = text
10    if let image = image {
11        jsonObject["image_details"] = extractDetailsFromImage(image)
12    }
13
14    // Create the JSON payload
15    var jsonError: NSError?
16    let jsonData = NSJSONSerialization.dataWithJSONObject(jsonObject,
17                                                          options: nil, error: &jsonError)
18    if jsonData {
19        request.HTTPBody = jsonData
20    } else {
21        if let error = jsonError {
22            println("JSON Error: \(error.localizedDescription)")
23        }
24    }
25
26    return request
27 }
```

This method doesn't actually create a request which uploads the image, although it could be adapted to do so. Instead, it extracts some details about the image using the following method:

```

1 func extractDetailsFromImage(image: UIImage) -> NSDictionary {
2     var resultDict = NSMutableDictionary()
3     resultDict["height"] = image.size.height
4     resultDict["width"] = image.size.width
5     resultDict["orientation"] = image.imageOrientation.toRaw()
6     resultDict["scale"] = image.scale
7     resultDict["description"] = image.description
8     return resultDict.copy() as NSDictionary
9 }
```

Finally, you can ask the session to create a task associated with the request you've built, and then call `resume()` on it to kick it off in the background:

```

1 // Create the task, and kick it off
2 let task = session.dataTaskWithRequest(request!)
3 task.resume()
```

If you run through this process now, with your own `requestb.in` URL in place, then you can expect to see results like this:

FORM/POST PARAMETERS	HEADERS
<code>None</code>	<code>Content-Length: 141</code> <code>User-Agent: com.shinobicontrols.ShareAlike.ShareExtension/1</code> <code>CFNetwork/703.2 Darwin/13.2.0</code> <code>Accept-Language: en-us</code> <code>Accept-Encoding: gzip, deflate</code> <code>Connection: close</code> <code>Accept: application/json</code> <code>Host: requestb.in</code> <code>Content-Type: application/json</code> <code>X-Request-Id: 1f2fa395-743a-4ed6-87e0-1eb2ca37f6ca</code>
RAW BODY	
<pre>{"image_details": {"scale": 1, "width": 1000, "orientation": 0, "description": "<UIImage: 0x7d2788f0>", "height": 1000}, "text": "Highland Cow says hi!"}</pre>	

Request Bin Success

Conclusion

Sharing extensions are just one of the extensions available to developers in iOS 8, and represents Apple opening up the operating system in a way they've been asked to do for a while. Interestingly, it's done in a way that has prioritized security and privacy, arguably at a small cost of customizability.

Building sharing extensions is far from trivial, and along the route there are many things that can trip you up. However, if it's applicable to you and your app then it's definitely worth investing the time in. The API is good, and the gain to your users could be huge.

The code for this project is available on github at github.com/ShinobiControls/iOS8-day-by-day¹⁴. In order to get the uploading working, you will need to ensure that you've got a shared container configured correctly, although the project file should do most of that for you.

¹⁴<https://github.com/ShinobiControls/iOS8-day-by-day>

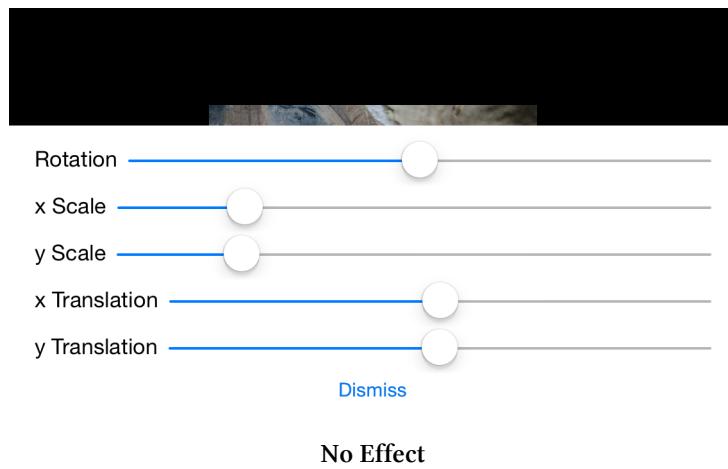
Day 3 :: UIVisualEffects



As part of the iOS 7 refresh the concept of visual depth within an app was introduced. This include some stunning visual effects involving live-filtering content in the background of the current view. This can be seen all over iOS 7 - from the navigation bar blurring the content as it scrolls underneath it, to the background of control center.

This type of filter is computationally expensive, and in iOS 7, developers weren't provided with an API to the underlying hardware-accelerate functionality, so we all built our own versions. Most of these were on the CPU, and therefore often weren't performant enough to cope with live-views. In iOS 8 there's a new option available in the form of `UIVisualEffects`. Apple has made their underlying implementation publicly accessible, so now you too can get cool blurring-style effects in your apps.

The app which accompanies this project displays an image, and has an overlay view with controls to vary the transform applied to the image. Before using the UI visual effects it looks like this:



The source code for the app is available on github at [github.com/ShinobiControls/iOS8-day-by-day¹⁵](https://github.com/ShinobiControls/iOS8-day-by-day). Feel free to send any pull requests or issues you find :)

Blurring with `UIVisualEffectView`

The architecture associated with the visual effects includes 2 classes:

- `UIVisualEffect` Specifies which kind of visual effect you want to apply. Choices are currently **blur** or **vibrancy**.

¹⁵<https://github.com/ShinobiControls/iOS8-day-by-day>

- `UIVisualEffectView` The view which actually performs the effect. You add your overlaid content to the `contentView` property and add the visual effect view itself to your view hierarchy.

In order to construct a `UIVisualEffectView` you need to have created a `UIVisualEffect`. To create a blur effect, use the `UIBlurEffect` subclass:

```
1 // Create the blur effect
2 let blurEffect = UIBlurEffect(style: .Light)
```

You create a blur effect with a particular style, of which there are 3 choices:

- `ExtraLight` Blurs and results in a lighter hue than the background
- `Light` Similar hue to that of the background
- `Dark` Darker appearance than the background

Since `ExtraLight` and `Dark` involve changing the hue as well as blurring, they involve an additional render pass, and are therefore marginally more expensive.

Once you've created a visual effect, then you can create a view to enable it to be rendered on screen:

```
1 let blurEffectView = UIVisualEffectView(effect: blurEffect)
```

The reason that you've created this blur view is so that you can use it as a background for some content you wish to display. Importantly, you do **not** add the content as a subview of the effect view itself, but rather its `contentView` property:

```
1 blurEffectView.contentView.addSubview(contentView)
```

Then, you just have to add the `blurEffectView` to your view hierarchy and you're done.

```
1 containerView.addSubview(blurEffectView)
```

A real-world example

Well, that's the theory. In practice, you have to size and position your view appropriately. Which isn't too difficult in itself. But I like to design my view in a storyboard, using auto layout. When you remove a view from the hierarchy (as you have to do in this case) it loses any layout constraints associated with its superview, therefore you need to re-create some. In most cases you want the content to be the same size as the visual effects view, which in turn should be the same size as its container.

```

1 // Prepare autolayout
2 contentView.setTranslatesAutoresizingMaskIntoConstraints(false)
3 blurEffectView.setTranslatesAutoresizingMaskIntoConstraints(false)
4 applyEqualSizeConstraints(blurEffectView.contentView, v2: contentView)
5 applyEqualSizeConstraints(containerView, v2: blurEffectView)

```

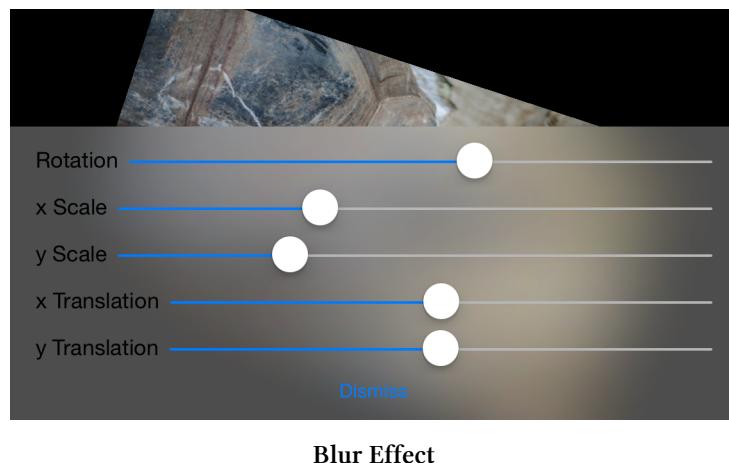
Where the `applyEqualSizeConstraints` is as follows:

```

1 func applyEqualSizeConstraints(v1: UIView, v2: UIView, includeTop: Bool) {
2     v1.addConstraint(NSLayoutConstraint(item: v1, attribute: .Left,
3                                         relatedBy: .Equal, toItem: v2, attribute: .Left,
4                                         multiplier: 1, constant: 0))
5     v1.addConstraint(NSLayoutConstraint(item: v1, attribute: .Right,
6                                         relatedBy: .Equal, toItem: v2, attribute: .Right,
7                                         multiplier: 1, constant: 0))
8     v1.addConstraint(NSLayoutConstraint(item: v1, attribute: .Bottom,
9                                         relatedBy: .Equal, toItem: v2, attribute: .Bottom,
10                                        multiplier: 1, constant: 0))
11    v1.addConstraint(NSLayoutConstraint(item: v1, attribute: .Top,
12                                         relatedBy: .Equal, toItem: v2, attribute: .Top,
13                                         multiplier: 1, constant: 0))
14 }

```

In the accompanying demo app, adding this functionality allows the layout of the controls to be in a storyboard. It results in an overlaid image controls view that looks like this:



Improving legibility with vibrancy

When this topic was introduced you may have noticed that in addition to the blur effect, there is also a **vibrancy** effect. Vibrancy is used in combination with a blur effect, and it applies a recoloring

effect to the content itself (as opposed to the blur effect which only affects the background). This effect not only looks great, it also ensures that the content is always legible.

Since the vibrancy needs to appear different for each of the different blur effect types, it's necessary to initialize it with a blur effect. This means that it's not possible to apply a vibrancy effect to a view which doesn't also have a blur effect applied to it.

In exactly the same way you did with the blur effect, create an effect object, and then create a view which uses this:

```
1 // Create the vibrancy effect - to be added to the blur
2 let vibrancyEffect = UIVibrancyEffect(forBlurEffect: blurEffect)
3 let vibrancyEffectView = UIVisualEffectView(effect: vibrancyEffect)
```

You add your content as a subview to the vibrancy effect view, and that in turn to the blur effect view:

```
1 vibrancyEffectView.contentView.addSubview(contentView)
2 blurEffectView.contentView.addSubview(vibrancyEffectView)
```

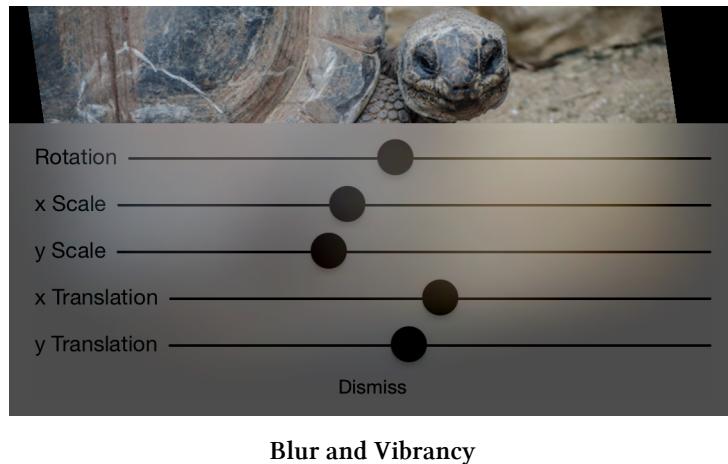
Then add the blurEffectView to your view hierarchy to display it:

```
1 containerView.addSubview(blurEffectView)
```

You can use the same auto layout dance as before to ensure that the view sizes are all correct:

```
1 // Prepare auto layout
2 contentView.setTranslatesAutoresizingMaskIntoConstraints(false)
3 blurEffectView.setTranslatesAutoresizingMaskIntoConstraints(false)
4 vibrancyEffectView.setTranslatesAutoresizingMaskIntoConstraints(false)
5
6 applyEqualSizeConstraints(vibrancyEffectView.contentView, v2: contentView)
7 applyEqualSizeConstraints(blurEffectView.contentView, v2: vibrancyEffectView)
8 applyEqualSizeConstraints(containerView, v2: blurEffectView)
```

This will result in this really rather cool effect:



Blur and Vibrancy

Performance Concerns

Creating these effects is hard work - involving multiple render passes. Therefore it's quite possible that on older devices using these with complex backgrounds can result in framedropping - which appears to users as choppiness. Even if the device can cope with it, the process will be quite power hungry. Therefore, it's important to use carefully, and ensure that you are fully aware of what usage entails.

In addition, Apple makes some recommendations on how to best use the visual effects view:

- Don't use transparency in the background of the visual effects view.
- Don't use masking in any superviews.
- To snapshot when you're using a visual effects view, you need to capture the entire window or screen in order for the effect to be visible.

Conclusion

This is a really powerful feature of iOS 8 - and will really allow developers to easily create apps which sit really well within the depth-model introduced in iOS 7. It was a little disappointing that this ability wasn't available in iOS 7, so it's good to have it ready to go in iOS 8.

Go and give it a try, but remember not to go wild with it - keep concerns of performance and energy usage at the forefront of your mind.

Don't forget that the code which accompanies this book is all available on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day¹⁶. Feel free to send any pull requests or issues you find, or gimme a shout on twitter - [@iwantmyrealname](https://twitter.com/iwantmyrealname)¹⁷.

¹⁶<https://github.com/ShinobiControls/iOS8-day-by-day>

¹⁷<https://twitter.com/iwantmyrealname>

Day 4 :: Custom Fonts in Interface Builder



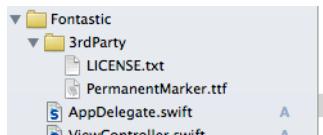
It is not unreasonable to want to use custom fonts in your apps, and it has been possible to do so for a while on iOS. However, it has always been troublesome designing the UI in Interface Builder, since you have to build your interface with the system font. This resulted in building and running your app repeatedly to get the desired result.

Well, Xcode 6 finally changes this! Now, any custom fonts present in your app's bundle will be available in Interface Builder, and will render as they would look in your running. This will be a noticeable jump in productivity, and you'll learn how easy it is to use in today's day-by-day article.

There isn't really any source *code* associated with today's post, but there is a project which demonstrates the functionality. The code is available on github at github.com/ShinobiControls/iOS8-day-by-day¹⁸.

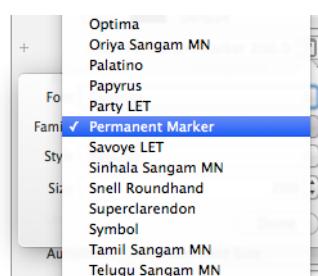
Font availability within Interface Builder

Interface Builder is able to pick up any fonts that are part of your app's bundle, and the easiest way to ensure this is to drag your ttf or otf file from the finder into the app:



Font Installation

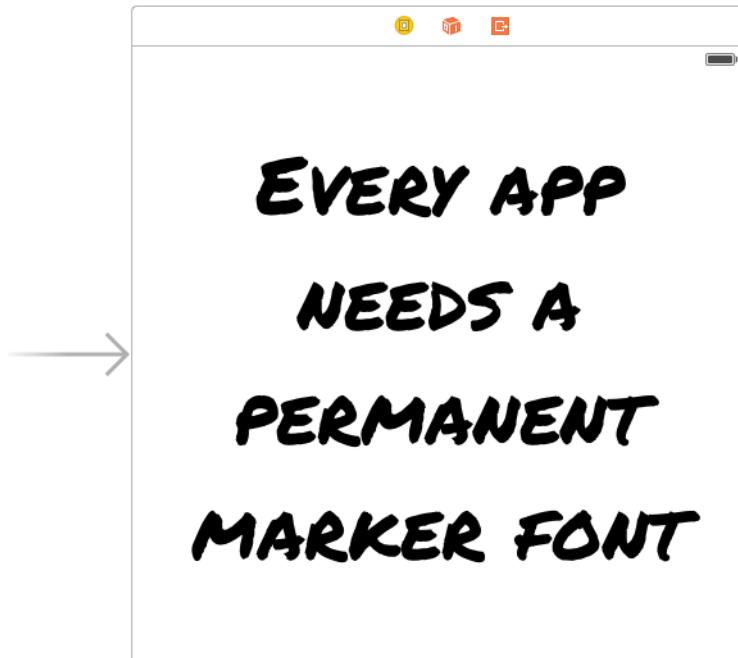
Then, you'll be able to select this font using the attributes inspector in Xcode, when editing a **xib** or **storyboard** file:



Font Selection

¹⁸<https://github.com/ShinobiControls/iOS8-day-by-day>

This will then update the appearance of the view appropriately in Interface Builder:



Font Appearing in IB

This is great, but if you run your app up now, then you'll very quickly realise that it's not a completely happy story:

Carrier 1:50 PM

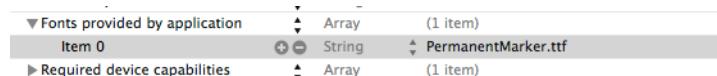
Every
app
needs a
permane
nt marker
font

Font not working

Declaring fonts for use within your app

The reason that your font isn't being picked up at runtime, is because although Interface Builder is able to pick up any fonts in the bundle, you still have to declare them in the plist (as before) in order to use them.

This is as simple as opening up the app's **Info.plist** file, either directly from the project navigator, or using the **Info** tab of the target's settings page, and adding the font:



Adding font to PList

You can run your app up again now, and this time you'll see it working!



EVERY APP
NEEDS A
PERMANENT
MARKER
FONT

Font working

Conclusion

Today's was quite a short chapter, but it represents a fix in Interface Builder which has been niggling at developers for a long time. I think that the little bits and pieces like this that Apple has been working on within the developer toolset demonstrates their commitment to making the process much more friendly.

As ever, the demo project is available on github at github.com/ShinobiControls/iOS8-day-by-day¹⁹.

¹⁹<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 5 :: Auto-sizing table view cells



In iOS 7 developers were introduced to the concept of dynamic type - which allows the user to change the font size used within apps, via the settings panel. This is really powerful, and can drastically improve the user experience for users with varying qualities of eye-sight.

However, there was a huge problem associated with this - in the form of table views. Although the type size can change, it was up to the developer to define the height of cells in the table view. This would mean either pre-calculating the values, or calculating them on the fly, neither of which was easy. Surely there must be a better way?

Well, in iOS 8 it is finally possible to have table view cells which can autosize themselves. In this project we'll take a look at how easy it is to implement, with a demo associated with dynamic type and with custom cells.

As with all articles in this series, the sample project is available at github.com/ShinobiControls/iOS8-day-by-day²⁰.

Using the 'stock' table view cells

When you first create a table view, chances are you've just used the stock table view cells - of which there are 4 styles (basic, left detail, right detail, and subtitle). In iOS 8, the labels in the base `UITableViewCell` are pre-configured for dynamic type. This means that they adapt to the text size specified in the device settings panel.

Since these cells are also laid out using autolayout, they will autosize to fit the differing sizes of text.

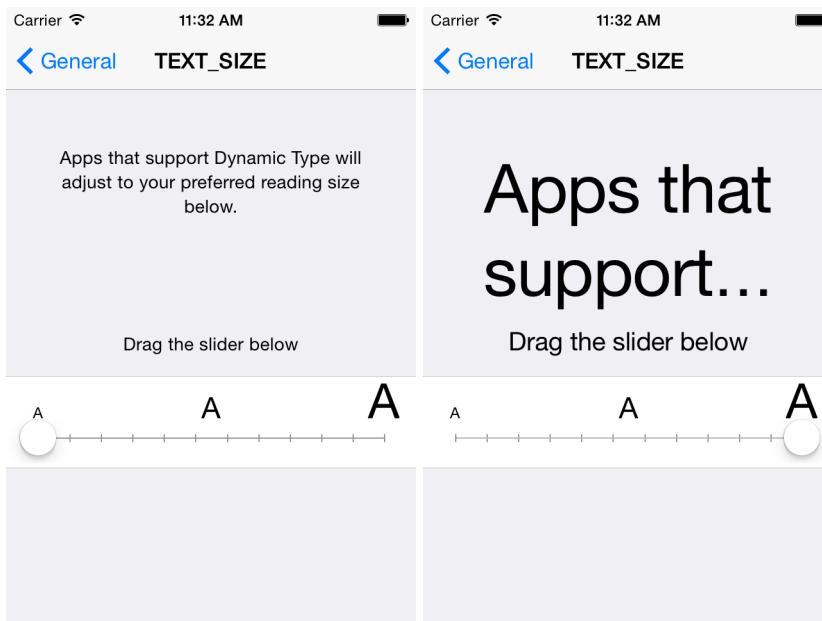
This actually means that you get auto-sizing table view cells **for free** if you just use the stock cells. To see this in action run up the accompanying **MagicTable** app:

²⁰<https://github.com/ShinobiControls/iOS8-day-by-day>

Autosizing for Free!	
Cell 0	0,0
Cell 1	0,1
Cell 2	0,2
Cell 3	0,3
Cell 4	0,4
Cell 5	0,5
Cell 6	0,6
Cell 7	0,7
Cell 8	0,8

Stock Cells Small

If you then use the settings to change the text size (in the same way as iOS7):



And then return to the **MagicTable** app then you'll be able to see the effect:



Stock Cells Large

This is all well, and good, but more often than not you'll want to create your own custom table cells. In order to do this, you need to understand a little bit more about how the auto sizing actually works - let's take a look at that next.

Creating custom table view cells

Traditionally setting the row height for a table view cell would be done on the table - using the `rowHeight` property. In order to vary the row size on a per-row basis, you would use the table view delegate method `estimatedHeightForRowAtIndexPath:` to return a different height for each row.

The problem for this approach is that you either need to know the row height at compile time, or calculate the row height for each row at run-time. The delegate method is called when the tableview first appears, which means that you need to calculate the cell height *before* the cells are created. In order to do this you can end up writing layout code twice - once for size calculations and once for display. This process can take a long time to perform - and involves entirely up-front calculations.

iOS 7 introduced `estimatedRowHeight`, which transformed the row-height requests into lazy calculations - only requesting the height for a row once it is about to be displayed on the screen. However, it still required you to calculate the row height yourself.

In iOS 8, you can still use these approaches, however, cells can now be responsible for their own sizing - via autolayout. This is both great from an ease-of-use perspective and also from a software design angle. A cell is responsible for its own layout, so it makes sense that it should also be responsible for determining its own height.

It's actually pretty easy to get auto-cell height working for a custom cell. The most important part is that your constraints properly define the height of the cell.

You must provide an estimated height for the rows, and if you provide an actual height (either via the property on the table, or via the delegate) then this will override any calculated cell size. In order to specify that you haven't set a cell height, use the `UITableViewAutomaticDimension` constant:

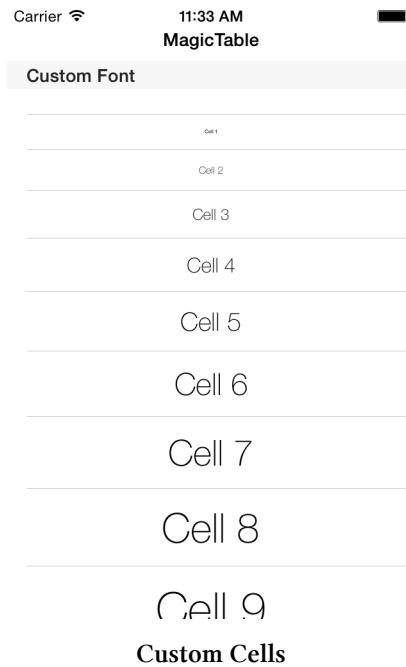
```
1 tableView.rowHeight = UITableViewAutomaticDimension
```

The constraints need to relate to the `contentView` within a `UITableViewCell`, and can be set up in code or in IB. The accompanying project sets up constraints in IB:



Creating Constraints

In order to demo the different heights, the datasource for the table in `MagicTable`, changes the font height of the custom label depending on the label:



Conclusion

Auto-sizing table view cells is something that developers have longed for, and it's great news that iOS 8 introduces this functionality. In many cases, since you should already be using auto-layout, you'll just get this functionality for free - just a matter of not specifying cell heights. It's definitely worth the time to go and ensure that your existing tableviews support this behavior.

As with all articles in this series, the sample project is available to download or clone from the Shinobi github at [github.com/ShinobiControls/iOS8-day-by-day²¹](https://github.com/ShinobiControls/iOS8-day-by-day).

²¹<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 6 :: Profiling Unit Tests



Unit tests are widely accepted as incredibly useful tools to aid writing better software. They allow you to establish the behavior of your code before putting it in front of end-users, and to ensure that everything works as expected in conditions which aren't necessarily default.

In Xcode 5 **OCUnit** was replaced with **XCUnit**, but there was very little noticeable difference to end users. In Xcode 6, **XCUnit** has had a few features added to it, one of which is the ability to measure the performance of a piece of code. This is really helpful to find sticking points in your app, and then iteratively improve the performance.

The app which accompanies this article demonstrates how you can use the new performance testing features of **XCUnit** to improve the speed of a simple algorithm. In this instance it takes a naive implementation of a moving average, measures the performance, before creating a new implementation which has better performance.

The code is available on github at [github.com/ShinobiControls/iOS8-day-by-day²²](https://github.com/ShinobiControls/iOS8-day-by-day), so grab it. Any questions then hit me up on twitter [@iwantmyrealname²³](https://twitter.com/iwantmyrealname).

Measuring Test Performance

The concept behind the new performance measuring functionality in **XCTest** is you can specify a block which the test runner will run repeatedly, measuring the time it takes to complete each time.

Each time you run your test suite, this performance measurement will be repeated, and a comparison to the last time will be displayed. This allows you to spot regressions in performance easily, and to iterate on improving performance.

Implementing this is really simple - **XCTestCase** now has method called `measureBlock` on it. This takes a void block/closure and times it:

²²<https://github.com/ShinobiControls/iOS8-day-by-day>

²³<https://twitter.com/iwantmyrealname>

```
1 func testSampleTestPerformance() {
2     self.measureBlock() {
3         // Code under test
4         methodCallUnderTest()
5     }
6 }
```

This might become a little clearer with a specific example.

Moving average example

Moving average is a common problem in computing, and is basically an implementation of a finite impulse response filter with unit weights. There are many ways to implement it, but the simplest is as follows:

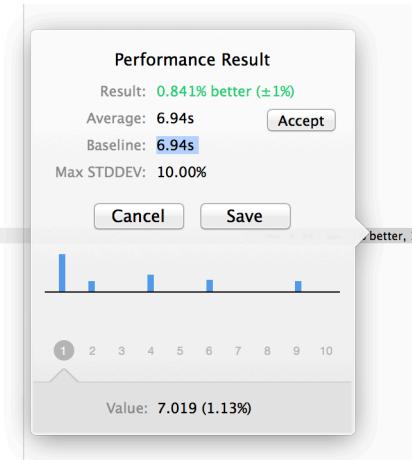
```
1 func calculateMovingAverage(data: Double[]) -> Double[] {
2     // Create an array to store the result
3     var result = Double[]()
4
5     // Now perform the calculation
6     for i in 0...(data.count - windowSize) {
7         let slice = data[i...(i+windowSize)]
8         let partialSum = slice.reduce(0) { $0 + $1 }
9         result.append(Double(partialSum) / Double(windowSize))
10    }
11    return result
12 }
```

The moving average is the mean of a window which moves along the input array - which is exactly what is being implemented here. This method is part of the `NaiveMovingAverageCalculator` class, which we can then put under test.

```
1 class MovingAverageTests: XCTestCase {
2     let calculatorCreator : () -> MovingAverageCalculator = {
3         return NaiveMovingAverageCalculator()
4     }
5     var calculator: MovingAverageCalculator = NaiveMovingAverageCalculator()
6
7     override func setUp() {
8         super.setUp()
9         // Reset the calculator
10        calculator = calculatorCreator()
11    }
12
13    func testMovingAveragePerformance() {
14        // This is an example of a performance test case.
15        calculator.windowSize = 1000
16        self.measureBlock() {
17            // Put the code you want to measure the time of here.
18            let randomArray = self.RandomDouble(10000)
19            let result = self.calculator.calculateMovingAverage(randomArray)
20            XCTAssertEqual(result.count, 9000)
21        }
22    }
23
24    func RandomDouble(length: Int) -> Array<Double> {
25        srand48(time(nil))
26        var result = Double[J]()
27        for i in 1..
```

There is one test method in the above code sample (`testMovingAveragePerformance()`), which includes the key `measureBlock()` method. Inside this block we are using the `RandomDouble()` utility method to create an array of random doubles of length 10000, before requesting the calculator to generate the moving average.

When you run this test suite then that test will be run 10 times. Once completed you can click the tag on the right hand side to see how it performed:



Initial Result

You can see a column chart of the time for each test run - giving you an indication of the variance. You can also see the mean time and its standard deviation. Setting a baseline saves this result, and then subsequent test runs will be compared to this. In fact, in the above picture, the baseline was set and you can see that a subsequent test run had only a tiny difference in performance, as you would expect.

Improving the moving average

Having decided that 6 seconds is an incredibly long time to perform a moving average, you can go ahead and attempt to improve upon it. The result of this is the `BetterMovingAverageCalculator` class, which has the following `calculateMovingAverage()` method:

```

1 func calculateMovingAverage(data: Double[]) -> Double[] {
2     var result = Double[]()
3
4     var currentSum = data[0..

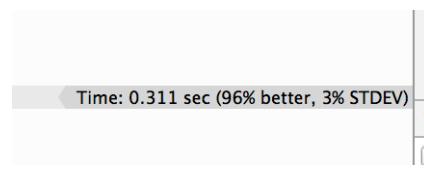
```

This approach is a common improvement to a moving average - keeping a running sum means that the same sum operation is not repeated for every output value.

To see how this fairs in comparison to the original implementation, you can just update the test class as follows:

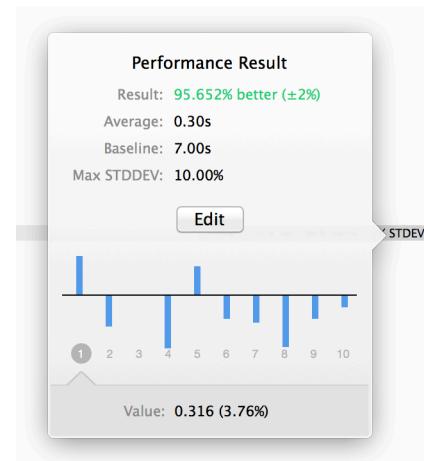
```
1 let calculatorCreator : () -> MovingAverageCalculator = {
2     return BetterMovingAverageCalculator()
3 }
4 var calculator: MovingAverageCalculator = BetterMovingAverageCalculator()
```

Now running the test suite again will cause the following balloon to appear:



Small improvement

If you click on it, you can reveal more result data:



Large improvement

That's quite a huge difference - well over 95%. It'd be great if it was that easy to optimize all code.

If you're happy with this new code, then you can reset the baseline to the new calculator, and you'll have changed your benchmark. The results can be checked in to source control and shared with your team, so you're all working to the same baselines.

Conclusion

There is a popular adage in software engineering, that you shouldn't optimize prematurely, so you probably shouldn't be throwing this new functionality all over your tests.

However, there are a couple of cases that this new tools really caters for:

- Ensuring that performance regressions are noticed, caught and dealt with or accepted.
- Making optimization a lot easier when the time comes.

It can be used in conjunction with a more fine-grained profiler, such as that provided by instruments to really assist with improving the performance of your app.

The code for this moving average project is available on github at github.com/ShinobiControls/iOS8-day-by-day²⁴. Feel free to fork it, and write a more optimal moving average calculator :)

²⁴<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 7 :: Adaptive Layout and UITraitCollection



Adaptive UI is probably the most important concept to be introduced in iOS 8. It might not seem like it at first, but abstracting the UI away from dependencies on things like device and orientation, allows closer concentration on the design itself, and less on the mechanics of implementation.

The most significant aspect to adaptive UI in iOS 8 is adaptive layout - that is layout which can automatically change depending on particular characteristics of the container.

This leads on to having universal storyboards - i.e. a single storyboard which can be used on multiple devices. This might well make storyboards an even more attractive proposition.

Today's post will give a quick introduction to adaptive layout in iOS 8. The accompanying project demonstrates a universal storyboard, with varying layouts. It's available as part of the github repo at [github.com/shinobicontrols/iOS8-day-by-day²⁵](https://github.com/shinobicontrols/iOS8-day-by-day).

Adaptive Layout

iOS 8 introduces the concept of **size classes** - which are used to categorize the amount of space in a particular dimension. There are 2 possible classes - **Regular** and **Compact**. Every view has particular size classes associated with them via a **TraitCollection**. By default these are defined by the device, but they can be overridden. A view can then customize it's view according to its size classes. The classes and their use on different devices are outlined below:

	Vertical Size Class	Horizontal Size Class
iPad Portrait	Regular	Regular
iPad Landscape	Regular	Regular
iPhone Portrait	Regular	Compact
iPhone Landscape	Compact	Compact

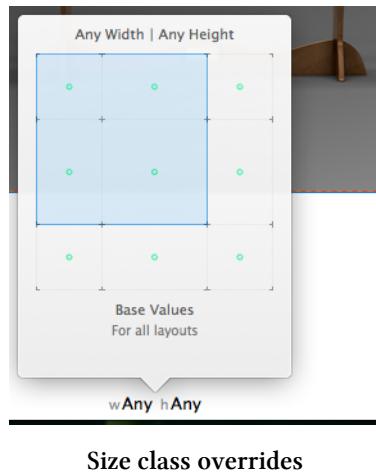
The key thing to consider here is that when designing layouts, you don't *have* to specify a size class - i.e. you might create a generic layout, and then override a few bits of the layout for the compact vertical size class.

Because of these size classes you are able to use just one universal storyboard for both iPad and iPhone layouts - removing an entire category of errors caused by mistakes in copying between the

²⁵<https://github.com/ShinobiControls/iOS8-day-by-day>

two storyboards.

Interface builder in Xcode 6 makes this process really rather easy to follow, with the new trait override selector. This can be found in the bar at the bottom of the view and allows you to specify exactly which size class combination your design is currently for.



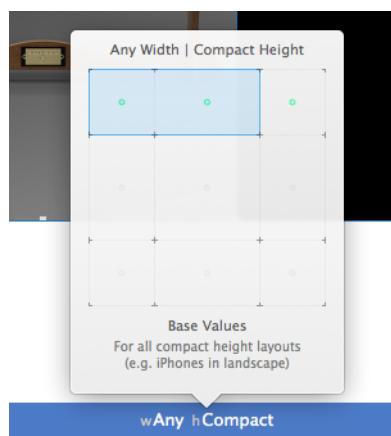
Size class overrides

The process for creating adaptive layouts should be as follows:

1. Start by creating the generic layout. This should be the base layout - the one from which the other layouts inherit - they are specializations of this layout - not completely independent.
2. Implement your layout specializations in the *least* constricting override class.

The key to this is thinking less in terms of devices, and more in terms of the layout itself.

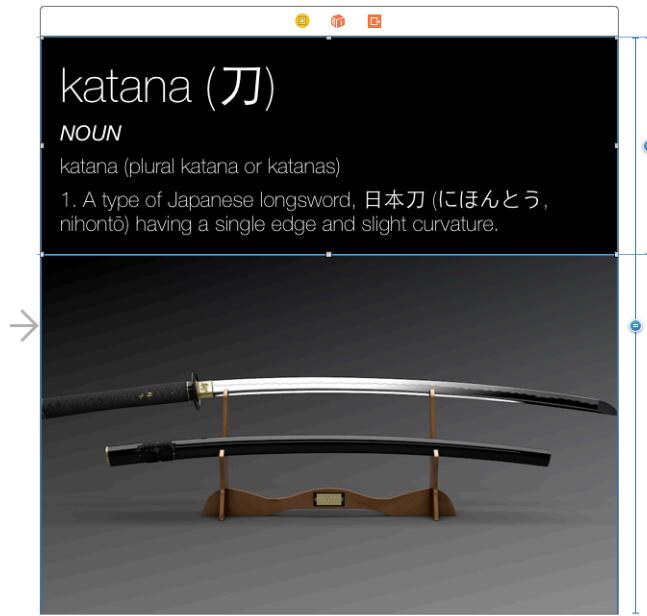
When you specify a size class override in IB then you'll see the bottom bar turn blue:



Blue bottom bar

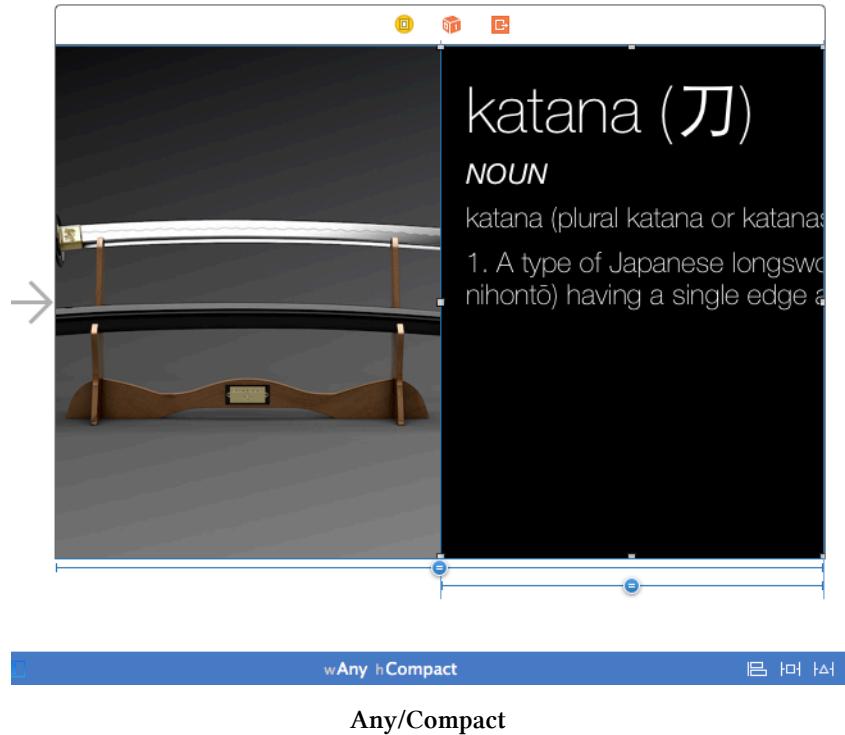
You are then able to add or remove layout constraints, and even (un)install views to configure a specific layout for this size class.

For example, the following is a screenshot from the accompanying project, in the ‘base values’ mode:



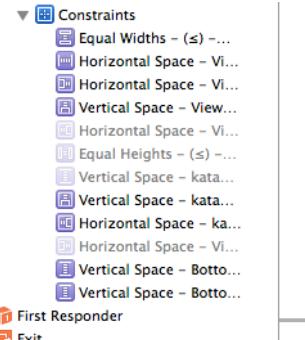
Any/Any

and here for horizontally compact classes:



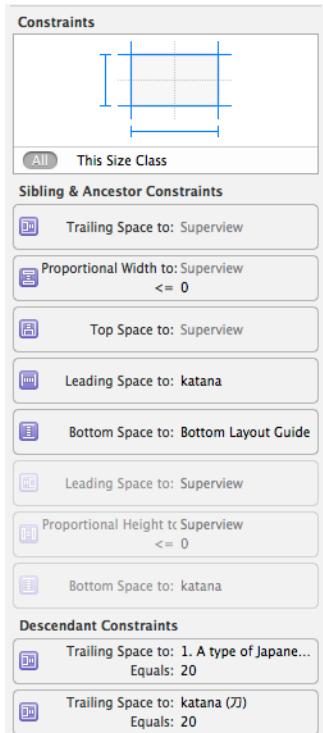
You can see that the layout has changed, with the picture moving to the left rather than being at the bottom. You should also spot that the sizes of the views on the storyboard don't relate to the devices. This is to encourage you to move away from thinking of pixel-perfect device appearance, and more towards the concept of generalized layout.

Removing layout constraints for particular size class overrides is known as uninstalling them - and can be done by pressing delete whilst the constraint is selected. The view of constraints in the document navigator shows both the installed and uninstalled constraints:



Constraints

You can view all the constraints which affect a particular view in the size inspector, with an option to see all constraints, or just those for the current size class:



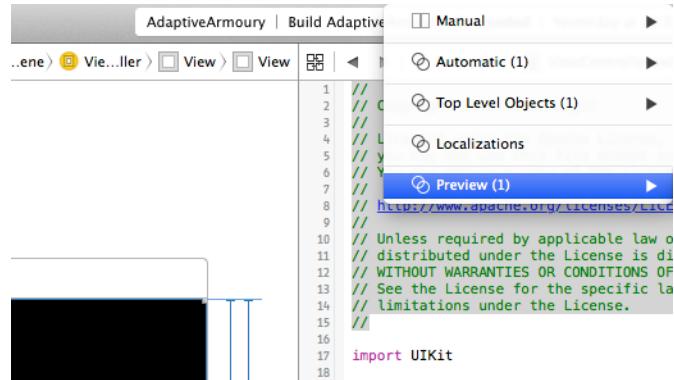
Size Inspectors

That's pretty much all you actually need to know in order to get started with adaptive layout. The key thing to remember is to think in terms of generalized layout, and not specifics for different devices.

Xcode Assistance

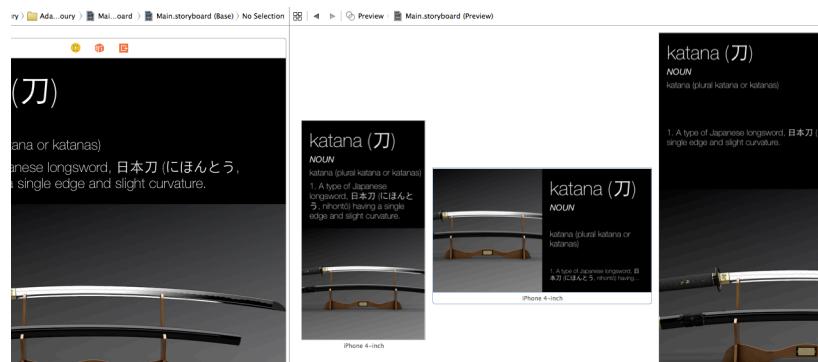
This approach can seem a little daunting at first, but Xcode 6 has some helpful features to ease your transition to the new way of thinking. The first is a new assistant view which will show you what your layout looks like on different devices, without having to run the project up on a simulator.

You access the preview by opening an assistant editor, and then selecting the preview from the jump bar:



Opening the preview editor

Once there you can add preview for different devices, and rotate them to your chosen orientation:



Preview editor

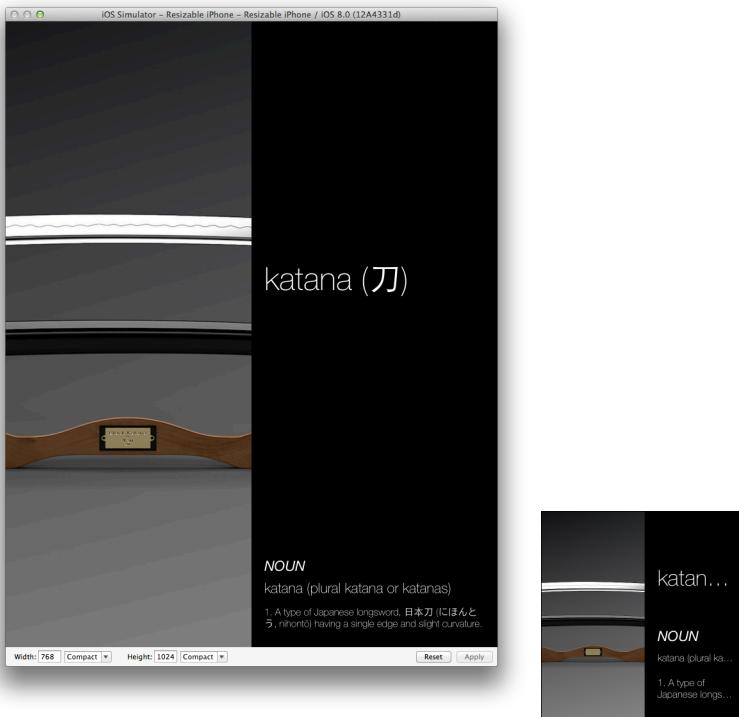
The views in the preview will update as you change the layouts in the storyboard.

The other new addition to your toolkit is resizable simulators - these are simply simulators for which you can specify both point sizes and size classes. These are accessible from the list of simulators in the same way as the standard device simulators:



Starting Resizable Simulator

You can then see how your design works at a multitude of different resolutions and size classes. The following show a simulator running at different resolutions, but with both horizontal and vertical size classes set to 'compact'.



Conclusion

The technologies behind adaptive layout aren't actually that complicated. It involves you rethinking how your design works. If you haven't already embraced the world of autolayout, then now is the time to get onboard. The introduction of adaptive layout should be a strong indication that in the future we'll be designing for more device configurations than the three we're currently used to. Attempting to scale up the existing device specific approach will not be achievable - therefore implying that autolayout is going to become a requirement.

You should definitely spend some time becoming acquainted with IB in Xcode 6 - it offers a lot of improvements over previous versions.

The project which accompanies today's post demonstrates setting up a simple adaptive layout project using storyboards - and you can grab it at [github.com/ShinobiControls/iOS8-day-by-day²⁶](https://github.com/ShinobiControls/iOS8-day-by-day).

²⁶<https://github.com/ShinobiControls/iOS8-day-by-day>

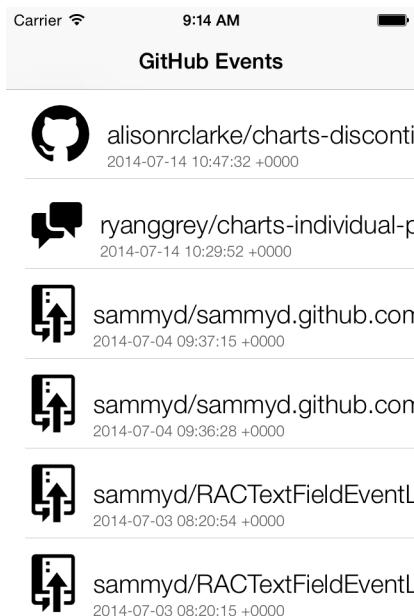
Day 8 :: Today Extension

8 08

Way back in chapter two we took a look at the new sharing extension - which is just one of six new extension points on iOS. Today, it's the chance of the today extension, or widget.

Widgets allow you to add content to the today screen on a device. Until iOS 8, this area has been sacred - with only system apps being allowed to display anything there. This new extension point will allow you to bring small amounts of relevant info to your users, in an easily accessible fashion.

The project which accompanies this project is based around the Github user public event feed. The app itself shows the most recent events, and the today widget shows just the latest event. Throughout this post you'll learn how to create a today extension, how to share code with the app, how to share cached data with the app and how to communicate from the widget to the app.



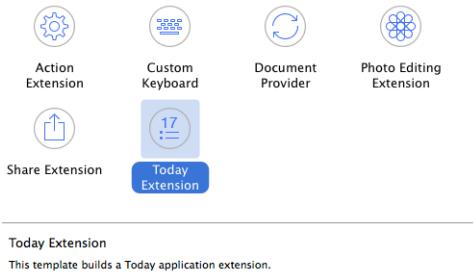
The code for this project is available on Github at [github.com/ShinobiControls/iOS8-day-by-day²⁷](https://github.com/ShinobiControls/iOS8-day-by-day).

Creating a widget

Like all other extensions, widgets have to be distributed as part of a host app and therefore Xcode provides an extension to add a today extension target to an existing project. This sets you up with

²⁷<https://github.com/ShinobiControls/iOS8-day-by-day>

a good foundation for a widget.



Today extension template

At its heart, a widget is just a view controller which gets displayed within the context of the today overlay. The Xcode template includes a view controller, so you can kick off by implementing the same things that you normally would within that.

The Xcode template also includes a storyboard, already wired in, containing a simple “Hello World” label. If your layout permits, using a storyboard will allow you to create a layout which works well for a widget. It’s certainly worth using autolayout when designing your layout, as then it will cope well with different devices.

Part of the difficulty using a storyboard for this design is that in order to work well with the visual effects used on the today screen, your view should be transparent. Therefore you can be building a view without really being able to see it.

By default, a widget has a wide left margin - which will not be part of your view controller’s view. In order to alter this, a new protocol has been introduced - `NCWidgetProviding`. This contains methods which allow you to customize both the behavior and the appearance of the widget. One of the methods on this protocol is `widgetMarginInsetsForProposedMarginInsetss()` which passed you the default margin insets, and allows you to return your own version. In the `GitHubToday` sample project, the following override is used:

```

1 func widgetMarginInsetsForProposedMarginInsetss(defaultMarginInsets: UIEdgeInsets)
2                                     -> UIEdgeInsets {
3     let newInsets = UIEdgeInsets(top: defaultMarginInsets.top,
4                                 left: defaultMarginInsets.left-30,
5                                 bottom: defaultMarginInsets.bottom,
6                                 right: defaultMarginInsets.right)
7     return newInsets
8 }
```

This extends the widget 30 points to the left. The design uses this space for an icon which represents the type of event:

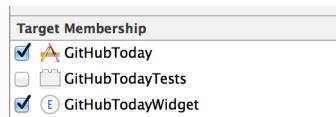


The other method on `NCWidgetProviding` is called by the system to ask whether there are any updates available for the widget. This allows you to discover whether there are any updates available, update the layout if necessary, and to let the system know the result. The method is `widgetPerformUpdateWithCompletionHandler(completionHandler:)` and you will see a sample implementation of it from `GitHubToday` later in the article.

First of all, you need to address a couple of issues, the first being how to share code between the parent app and the widget.

Sharing code with the parent app

`GitHubToday` makes a network request to the Github API, and then parses the resulting JSON to extract the content it needs to display. This process would effectively be repeated by the widget as well as the app itself - but copying the same code between two projects is incredibly inefficient. One option would be to add the source files to both targets:



Code in two targets

This approach will definitely work, but will result in the same functionality being created in two binaries. Luckily, there's a better way. iOS 8 introduces the concept of a dynamically library, and a widget can use the same library as the host app. Therefore the best approach is to create a dynamic framework, and put all the common code in there.

This article is not primarily about dynamic frameworks, so won't go in to detail about how to create or use them, but once you've created them, you can move any shared code into it. For example, in `GitHubToday`, the entire model layer and networking implementation is all packaged into a dynamic framework. This includes the `GitHubEvent` class, along with `GitHubDataProvider`, which is used to make the network request itself.

Being able to share code between the app and the widget is really helpful, but widgets need to be super responsive - setting off a network operation each time the today screen comes in to view is not going to give the fast, snappy user experience you desire. In the next section you'll learn how you can improve upon this by creating a cache which can be shared between the app and the widget.

Sharing a cache with the parent app

Since a widget is an extension it's not allowed access to its own disc space, but it can use a shared container. If you remember back to day 2 of this series, you learnt how to create a shared container which both an extension and its host app can write to. There it was only used as a cache for a `NSURLSession` background task, but you can also use it as a shared cache, between the app and the widget.

You can obviously store files in this container, so you could create an SQLite database, or use CoreData, but the simplest approach here is to use `NSUserDefaults` as a key-value store.

Since the only data that the GitHubToday widget ever needs is the latest event, then you can store a `GitHubEvent` object in an `NSUserDefaults` file, which can live in the shared container. When the widget first opens then it can populate itself from this cached event, and then update the cache (and the view) when the network operation completes.

The following class (which exists in the common dynamic framework) demonstrates the behavior of a simple cache:

```
1 let mostRecentEventCacheKey = "GitHubToday.mostRecentEvent"
2 public class GitHubEventCache {
3     private var userDefaults: UserDefaults
4
5     public init(userDefaults: UserDefaults) {
6         self.userDefaults = userDefaults
7     }
8
9     public var mostRecentEvent: GitHubEvent? {
10        get {
11            if let data = userDefaults.object(forKey:mostRecentEventCacheKey) as? NSData {
12                if let event = NSKeyedUnarchiver.unarchiveObjectWithData(data)
13                    as? GitHubEvent {
14                    return event
15                }
16            }
17            return nil
18        }
19        set {
20            if let event = newValue {
21                let data = NSKeyedArchiver.archivedDataWithRootObject(event)
22                userDefaults.setObject(data, forKey: mostRecentEventCacheKey)
23            } else {
24                userDefaults.removeObjectForKey(mostRecentEventCacheKey)
25            }
26        }
27    }
28}
```

```
26     }
27     }
28
29 }
```

If defines a property `mostRecentEvent`, which is pulled from the `UserDefaults` if it exists. It also ensures that the event is pushed back there when it is updated. It's important that `GitHubEvent` implements the `NSCoding` protocol, so that the keyed archiver knows how to archive and unarchive it.

The `GitHubEventCache` requires an `NSUserDefaults` object to read and write to. In order that this can be shared between the app and the extension, this must be created in the shared container using the `NSUserDefaults(suiteName:)` initializer:

```
1 let mostRecentEventCache = GitHubEventCache(userDefaults: UserDefaults(suiteNa\
2 me: "group.GitHubToday"))
```

Once you have created a cache in the widget's view controller then you can use it to populate the view in `viewDidLoad()`:

```
1 let mostRecentEventCache = GitHubEventCache(userDefaults:
2                                     UserDefaults(suiteName: "group.GitHubToday"))
3 var currentEvent: GitHubEvent? {
4     didSet {
5         dispatch_async(dispatch_get_main_queue()) {
6             if let event = self.currentEvent {
7                 self.typeLabel.text = event.eventType.icon
8                 self.repoNameLabel.text = event.repoName
9             } else {
10                 self.typeLabel.text = ""
11                 self.repoNameLabel.text = ""
12             }
13         }
14     }
15 }
16
17 override func viewDidLoad() {
18     super.viewDidLoad()
19     currentEvent = mostRecentEventCache.mostRecentEvent
20 }
```

The system will call the `widgetPerformUpdateWithCompletionHandler()` method once the widget had been displayed, and at this point you can kick off a network request to ensure that the latest data is displayed:

```
1 func widgetPerformUpdateWithCompletionHandler(completionHandler:  
2                                     ((NCUpdateResult) -> Void)!) {  
3     dataProvider.getEvents("sammyd", callback: {  
4         events in  
5         let newestEvent = events[0]  
6         if newestEvent != self.currentEvent {  
7             self.currentEvent = newestEvent  
8             self.mostRecentEventCache.mostRecentEvent = newestEvent  
9             completionHandler(.NewData)  
10        } else {  
11            completionHandler(.NoData)  
12        }  
13    })  
14}  
15 }
```

Note that here if the latest event you receive from the web service is different to the one you are currently displaying then you can update the view and then tell the system that you have new data, by calling `completionHandler(.NewData)`. If you don't receive new data then you can instead call `completionHandler(.NoData)`.

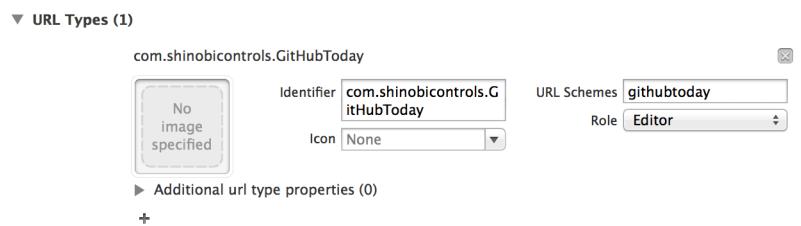
Using this caching approach means that you can display the latest data when appropriate, but also maintain the responsiveness and user experience required of a today widget.

Obviously this implementation of a cache is incredibly simple - you can build as much complexity as you want in to the cache. Note that in the main app, when a network request is completed, the latest result is pushed into the cache, ensuring that the widget will always start by displaying the most recently downloaded data, whether it be from the app or the widget itself.

Navigating back to the parent app

The standard user story for a widget is that a user would look at the summary and if they want more info, then they can tap the appropriate part of the widget. In order to achieve this, then you can utilize the existing iOS URL functionality.

You can define a URL scheme in the **Info** section of the app's target:



Defining URL Scheme

As is standard when defining a URL scheme for an app, you also need to implement the appropriate method in your app delegate:

```

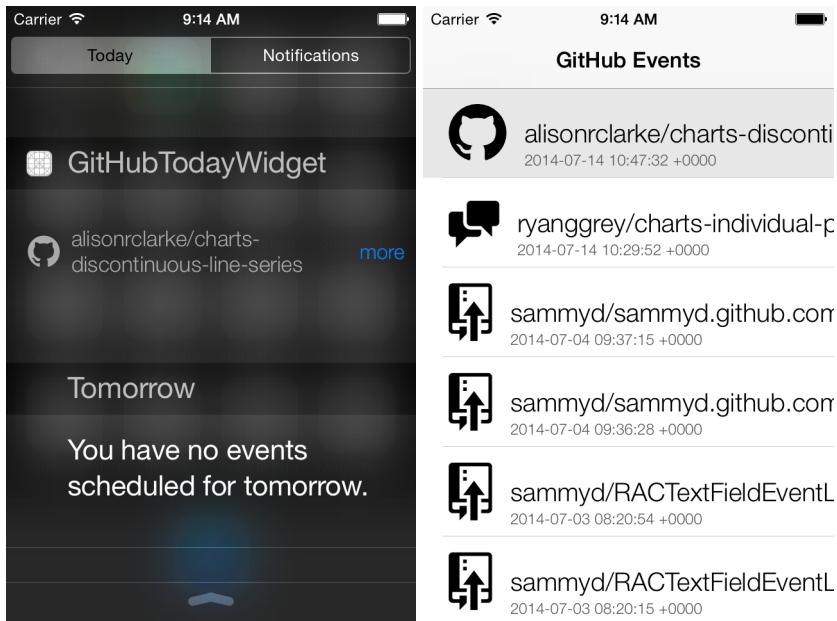
1 func application(application: UIApplication!, openURL url: NSURL!,
2                   sourceApplication: String!, annotation: AnyObject!) -> Bool {
3     if let navCtlr = window?.rootViewController as? UINavigationController {
4         if let tableCtlr = navCtlr.topViewController as? UITableView {
5             if let eventId = url.lastPathComponent.toInt() {
6                 tableCtlr.scrollToAndHighlightEvent(eventId)
7             }
8         }
9     }
10    return true
11 }
```

This here will attempt to scroll to and highlight the cell in the table view which represents the event with the given ID.

Now that you've got a URL scheme set up, you can use the `openURL()` method on the `extensionContext` to link from the widget to the relevant row in the app:

```

1 @IBAction func handleMoreButtonTapped(sender: AnyObject) {
2     let url = NSURL(scheme: "githubtoday", host: nil, path: "/\$(currentEvent?.id)")
3     extensionContext?.openURL(url, completionHandler: nil)
4 }
```



Conclusion

All the extensions are a really cool new feature of iOS 8 - they represent the beginning of Apple opening up the operating system for devs. The today extension could be hugely powerful - offering the chance to really improve the user experience. However, it's important to use it wisely. If there's a massive influx of mediocre widgets, then users might get annoyed enough not to trust any. Since they affect a core area of their device usage, widgets need to be good citizens.

The code for this app and widget is available on github at github.com/ShinobiControls/iOS8-day-by-day²⁸. Feel free to grab it, try it and break it. Do let me know how you get on - I'm really interested to see what widgets I'm going to be adding to my today screen in the coming months - I'm [@iwantmyrealname](https://twitter.com/iwantmyrealname)²⁹

²⁸<https://github.com/ShinobiControls/iOS8-day-by-day>

²⁹<https://twitter.com/iwantmyrealname>

Day 9 :: Designated Initializers



The concept of designated initializers is not new to the Cocoa world - they exist in objective-C, but somewhat informally. Swift formalizes class initialization, both in terms of what the different initialization methods should do, and the order in which things should be done.

In order to ease interoperability with objective-C, there is also a little more formalization there too.

In this post you'll learn a bit about how initialization works in Swift, with an explanation of what designated initializers are, and how they relate to convenience initializers.

The accompanying code for this project is part of an Xcode playground - and is available in the repo on github at github.com/ShinobiControls/iOS8-day-by-day³⁰. You should be able to just open it up and see the live-run results.

Creating objects

Initializers are used to instantiate an instance of a class or a struct. There are two types of initializer:

- **Designated** This is responsible for preparing all the properties, and calling the superclass' initializer, allowing it to do the same.
- **Convenience** Don't have to directly prepare and instance state, but must call a designated initializer. Cannot call an initializer of the superclass.

For example:

```
1 class Person {  
2     var name: String  
3     var age: Int?  
4     var consideredDangerous = false  
5  
6     init(name: String, age: Int?) {  
7         self.name = name  
8         self.age = age  
9     }  
10 }
```

³⁰<https://github.com/ShinobiControls/iOS8-day-by-day>

This is a class which has 3 properties. `init(name:, age:)` is a designated initializer, and therefore it is responsible for ensuring that all properties have been correctly initialized. In the example above that actually only means that it must set a value for the `name` property. This is because optionals will be set to `nil` by default, and `consideredDangerous` is set inline.

Since the `age` is optional, we might want to offer another initializer which just accepts a name. One option would be to make another designated initializer:

```
1 init(name: String) {  
2     self.name = name  
3 }
```

This is perfectly acceptable, but it is actually repeating the same code as in the previous designated initializer. Should you decide to store uppercase names, then you'd have to change both initializers. This is where the concept of initializer chaining comes in - which is the pattern established with convenience initializers.

Convenience Initializers

Convenience initializers are denoted by the use of the `convenience` keyword, and they **must** call a designated initializer:

```
1 convenience init(name: String) {  
2     self.init(name: name, age: nil)  
3 }
```

This method now has the desired behavior - it only requires a name, and delegates the actual initialization to a designated initializer. Convenience initializers **cannot** call the super class, but **must** call a designated initializer in the same class.

Subclassing

When you create a subclass then in addition to the aforementioned rules associated with designated initializers, you are also required to call a designated initializer of the superclass.

Look at the following example - a ninja is clearly a person, but they have a collection of weapons, defined by the accompanying enum:

```
1 class Ninja: Person {  
2     var weapons: [Weapon]?  
3 }  
4  
5 enum Weapon {  
6     case Katana, Tsuba, Shuriken, Kusarigama, Fukiya  
7 }
```

The `Ninja` class definition doesn't currently have an initializer, and since the only additional property is an optional (and thus defaults to nil) it isn't a requirement to have one. However, it'd be nice to add one which allows setting the `weapons` array at initialization time:

```
1 init(name: String, age: Int?, weapons: [Weapon]?) {  
2     self.weapons = weapons  
3  
4     super.init(name: name, age: age)  
5  
6     self.consideredDangerous = true  
7 }
```

This demonstrates the rules of how a designated initializer must be formed in Swift:

1. Any properties on the subclass must be initialized correctly. Here it's not strictly necessary since `weapons` is an optional type.
2. Once the current object's properties are all initialized, there **must** be a call to a designated initializer on the superclass.
3. You can then update any properties inherited from the superclass.

This order is very important, and changing it will result in compiler errors. Note that the order is different to the ordering used in subclass initializers in objective-C, where the call to the superclass is the first instruction.

You can add convenience initializers to subclasses as well, but they must call a designated initializer of the same class. They **cannot** call an initializer of a superclass:

```
1 convenience init(name: String) {  
2     self.init(name: name, age: nil, weapons: nil)  
3 }
```

This results in you being able to create ninjas in 2 ways:

```
1 let tina = Ninja(name: "tina", age: 23, weapons: [.Fukiya, .Tsuba])
2 let trevor = Ninja(name: "trevor")
```

Usage in objective-C

Objective-C has the notion of designated initializers - but in an informal context. In order to enable full interoperability between objective-C and Swift, there is a macro with which you can annotate your objective-C initializers: `NS_DESIGNATED_INITIALIZER`:

```
1 - (instancetype)init NS_DESIGNATED_INITIALIZER;
```

By using this, then all other initializers in your class will be interpreted as being convenience initializers. The same rules apply with objective-C initializers as their Swift counterparts.

Conclusion

The designated initializer pattern has existed in the Cocoa world for a long time, but Swift formalizes it somewhat. You'll need to fully understand it and what is required of you as a developer in order to create your own classes and subclasses in Swift.

It's also definitely worth adopting the new macro in any new objective-C that you write, particularly if you want it to be interoperable with Swift code.

The code which accompanies this post is in the form of an Xcode 6 playground - and demonstrates how the different patterns work. It is part of the day-by-day repo on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day³¹.

³¹<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 10 :: Xcode 6 Playgrounds



Playgrounds are a completely new concept in Xcode 6 - bringing the world of a completely interactive code development environment with the powerful new language that is Swift. Using a playground feels very much like a powered-up REPL - combining the persistence of a source code file with the immediate response of a REPL.

Playgrounds are really easy to get started with - in fact when you open Xcode 6 there is an option to create a new playground. You can choose either an iOS playground or an OSX playground, and they only support writing in Swift.

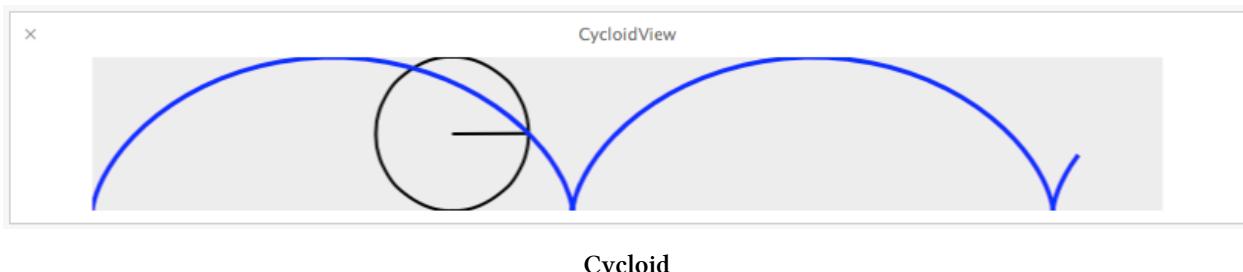
Today's article won't cover many of the basics of playgrounds - it's not difficult to get started, but will instead cover some of the more advanced features available through the `XCPlayground` framework.

The accompanying project is itself a playground, and is available in the ShinobiControls github repo at github.com/ShinobiControls/iOS8-day-by-day³². You'll notice that there are two projects within this repo - one for iOS, one for OSX.



Note: At the time of publication, not all the features are available for iOS Xcode 6.2b3. Therefore, the OSX playground is provided for reference. The code is very similar - with like-for-like exchanges where possible. The code snippets in this article will refer to the iOS version.

The aim of the playground project is to create a view which can draw a cycloid. A cycloid is the name of the curve which is traced by a point on the edge of a wheel as it rotates. This development process is very representative of what you could use a playground for. You can read more about cycloids on [Wikipedia](http://en.wikipedia.org/wiki/Cycloid)³³, and the following image (also from Wikipedia) gives you an idea of what you're going to try and create:



³²<https://github.com/ShinobiControls/iOS8-day-by-day>

³³<http://en.wikipedia.org/wiki/Cycloid>

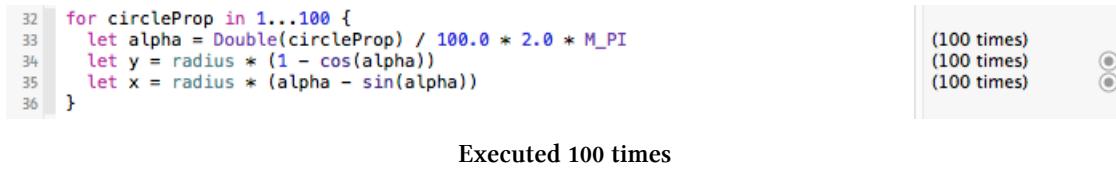
Interactive Coding & Timelines

As has already been mentioned, playgrounds allow for instantaneous feedback from code as you write it. This makes it ideal for algorithm development. In the centroid example, we can use high-school geometry to calculate the x and y positions of the point on the rim of the wheel for a given angle:

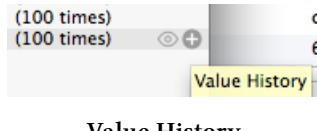
```

1 let radius = 10.0
2 for circleProp in 1...100 {
3     let alpha = Double(circleProp) / 100.0 * 2.0 * M_PI
4     let y = radius * (1 - cos(alpha))
5     let x = radius * (alpha - sin(alpha))
6 }
```

This code loops through 100 samples of the angle of rotation of the circle and calculates the location of the point for each of them. If you look at this code in the playground you'll see that over on the right-hand side it tells you that each line was executed 100 times (as you would expect):

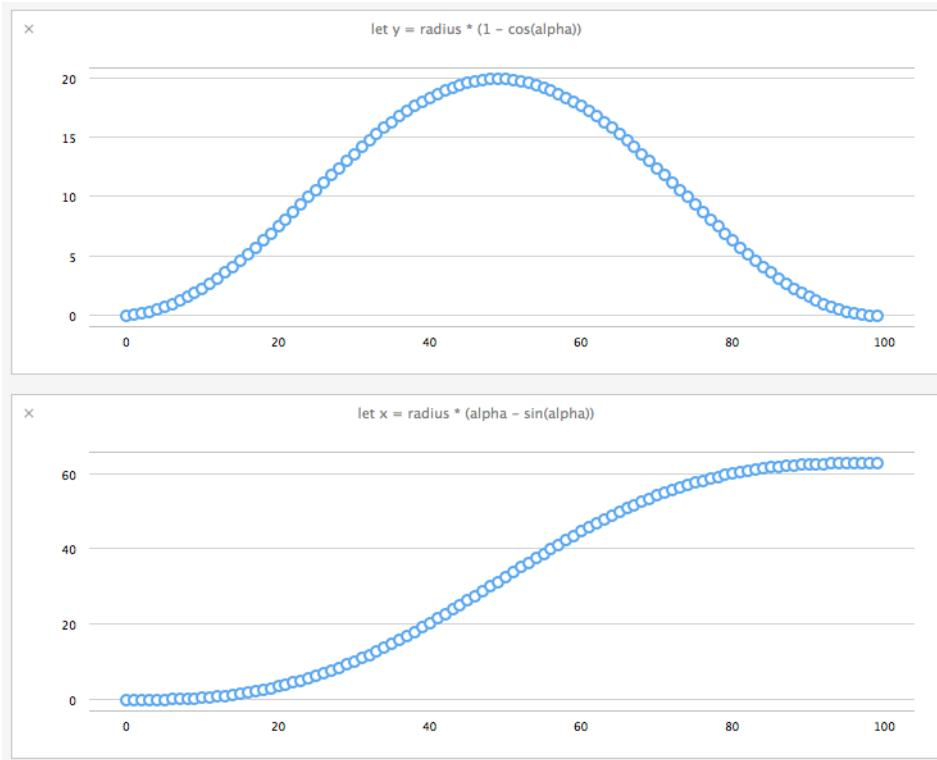


That in itself is not particularly ground-breaking, but if you mouse-over the quicklook gutter then you'll see a couple of icons appearing - a quicklook eye and a plus for **Value History**:



Value History

If you click this button then the assistant editor will open with a timeline view, containing a graph of these values over time:



Timeline for x and y

The timeline will automatically add reassessments to variables of the same name. If you want to specify that a particular value should be added to a given timeline chart you can use the `XCPCaptureValue(identifier:, value:)` method. This is part of the `XCPlayground` framework, so you'll need to import that:

```
1 import XCPlayground
```

Any time that you call `XCPCaptureValue()`, it'll be added to the timeline representation for the given string identifier. i.e. it'll be added to the end of a chart if one exists for the given name (and the type can be plotted). If the identifier has not yet been seen then a new view will appear in the timeline assistant editor and will be populated appropriately for the given type.

You can add other types to the timeline as well, including views, bezier paths, colors, strings, arrays, dictionaries etc. In fact pretty much anything which has an associated quick look. However, how do you add something which doesn't have a quick look representation? In the next section you'll learn how to do this.

Custom QuickLook

Now that you're happy with the algorithm itself, you'll want to build a class to contain the functionality. You can do this within a playground as well, and they offer some additional tools for making the experience great.

With the cycloid example, the initializer requires a radius and number of rotations:

```
1 init(radius: Double, numberRotations: Double = 2.5) {  
2     self.radius = radius  
3     self.numberRotations = numberRotations  
4     super.init()  
5 }
```

And the previously developed algorithm is used in the `generateDatapoint()` method:

```
1 func generateDatapoint(angle: Double) -> CGPoint {  
2     let y = radius * (1 - cos(angle))  
3     let x = radius * (angle - sin(angle))  
4     return CGPoint(x: CGFloat(x), y: CGFloat(y))  
5 }
```

So to actually create the points which make up the cycloid path you create the following method:

```
1 func pointsForCycloid(numberSamples: UInt) -> [CGPoint] {  
2     var dataPoints = [CGPoint]()  
3     for sampleIndex in 0..4         let angle = Double(sampleIndex) / Double(numberSamples)  
5             * 2.0 * M_PI * numberRotations  
6         dataPoints.append(generateDatapoint(angle))  
7     }  
8     return dataPoints  
9 }
```

OK, so you think that you've got this class working correctly, but how can you check? Well, if your class inherits from `NSObject` then you can implement the `debugQuickLookObject()` method to return something that the playground can visualize. The objects you can return include bezier paths, strings, colors and images.

Since you're dealing with `CGPoint` objects here then a bezier path would be ideal. The following method will create a bezier path from using the result of the `pointsForCycloid()` method:

```

1 func bezierPath(numberSamples: UInt) -> UIBezierPath {
2     let bezierPath = UIBezierPath()
3     bezierPath.moveToPoint(CGPoint.zeroPoint)
4     for point in pointsForCycloid(numberSamples) {
5         bezierPath.addLineToPoint(point)
6     }
7     return bezierPath
8 }
```

This means that the `debugQuickLookObject()` method is as simple as:

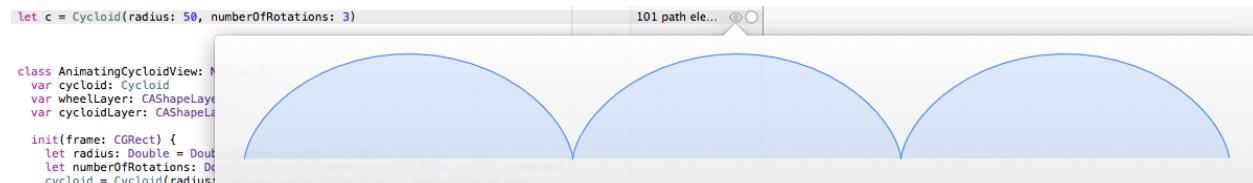
```

1 func debugQuickLookObject() -> AnyObject? {
2     return bezierPath(100)
3 }
```

To see this in action, simple instantiate one of these Cycloid objects:

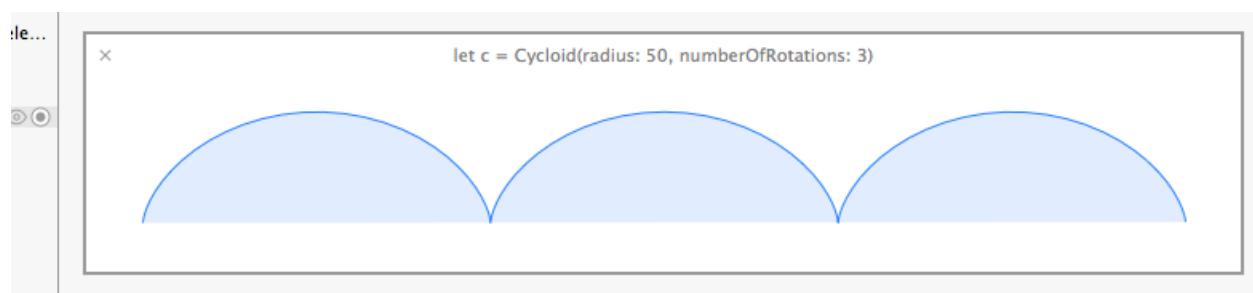
```
1 let c = Cycloid(radius: 50, numberOfRotations: 3)
```

You can either click the quicklook icon to see the result in a popover:



Quicklook Cycloid

Or, like you did with the values of `x` and `y`, you can add it to the timeline so that it will stay there and get updated each time the playground re-evaluates :



Timeline Cycloid

This is starting to look pretty cool now - you've got a class which will create the cycloid path for you. In the next section you're going to see how to use this to create a view which animated a wheel drawing this path.

Custom View Development

Creating custom views can be a really slow process in Xcode - although this has improved with the ability to do live rendering in interface builder. Even so, you are often required to build your project and run it up on a simulator before you can see what it actually looks like. This makes for a slow iterative loop - speeding this up could offer massive gains in productivity. Even just the ability to quicklook a view in playgrounds really can help here, but that's just scratching the surface of what's possible.



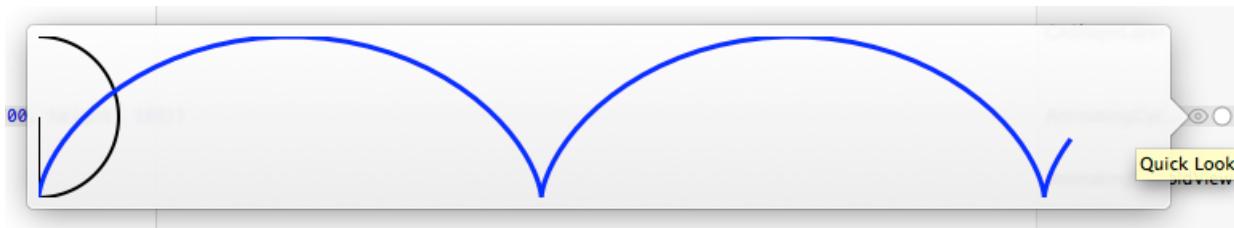
Note: Remember throughout this section that at the time of publication, animation of `UIView` in playgrounds was not supported. Therefore, in order to see it in action, use the OSX version instead. It is very similar code, with a few platform differences (including coordinate system differences).

The sample playground includes a `UIView` subclass called `AnimatingCycloidView`. This uses the previously created `Cycloid` class to determine the path of the appropriate cycloid and draws it on screen using a `CAShapeLayer`.

```
1 func createCycloidLayer() -> CAShapeLayer {
2     let layer = CAShapeLayer()
3     layer.bounds = self.bounds
4     layer.position = CGPoint(x: self.bounds.width / 2.0,
5                               y: self.bounds.height / 2.0)
6     layer.path = self.cycloid.bezierPath(100).CGPath
7     layer.fillColor = UIColor.clearColor().CGColor
8     layer.strokeColor = UIColor.blueColor().CGColor
9     layer.lineWidth = 3.0
10
11    return layer
12 }
```

A similar method is used to create the wheel. Since this is a `UIView` you get a quicklook for free:

```
1 let view = AnimatingCycloidView(frame: CGRectMake(x: 0, y: 0, width: 700, height: 10\
2 0))
```

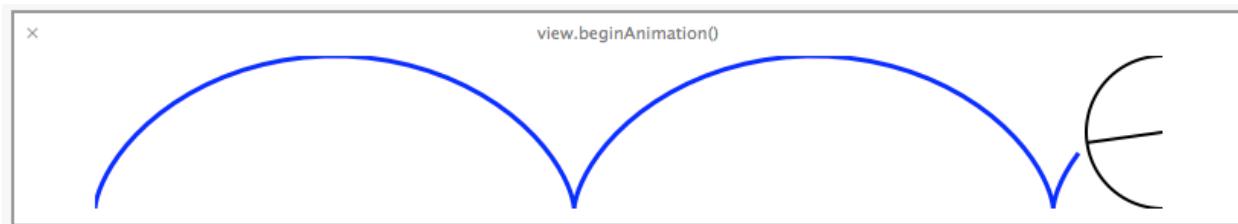


Quicklook Cycloid View

However, you want to be able to animate the wheel to trace the cycloid path. The animation method is fairly standard, and just uses `CABasicAnimation` to rotate and translate the wheel simultaneously:

```
1 func beginAnimation() {
2     self.wheelLayer.setValue(-2 * M_PI * self.cycloid.numberOfRotations,
3                               forKeyPath: "transform.rotation.z")
4     self.wheelLayer.setValue(self.bounds.width, forKeyPath: "position.x")
5     self.cycloidLayer.strokeEnd = 1.0
6
7     CATransaction.begin()
8     CATransaction.setAnimationDuration(6.0)
9
10    let animation = CABasicAnimation(keyPath: "transform.rotation.z")
11    animation.fromValue = 0
12    animation.toValue    = -2 * M_PI * cycloid.numberOfRotations
13
14    let translation = CABasicAnimation(keyPath: "position.x")
15    translation.fromValue = 0
16    translation.toValue   = self.bounds.width
17
18    let animationGroup = CAAnimationGroup()
19    animationGroup.animations = [animation, translation]
20    animationGroup.timingFunction = CAMediaTimingFunction(name:
21                                            kCAMediaTimingFunctionLinear)
22    animationGroup.removedOnCompletion = false
23
24    wheelLayer.addAnimation(animationGroup, forKey: "wheelSpin")
25
26    CATransaction.commit()
27 }
```

Try adding the `view` to the timeline with the **Value History** button and then watching what happens. You'll see the cycloid view in the timeline, but only in its completed state:

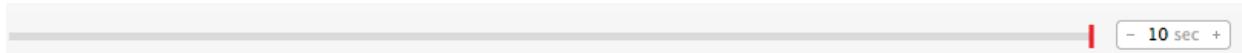


Cycloid after animation

It would be really helpful if you could see the animation happening. Well, the **XCPlayground** has a function which can help you out - in the form of `XCPShowView()`. This takes an identifier and a view object:

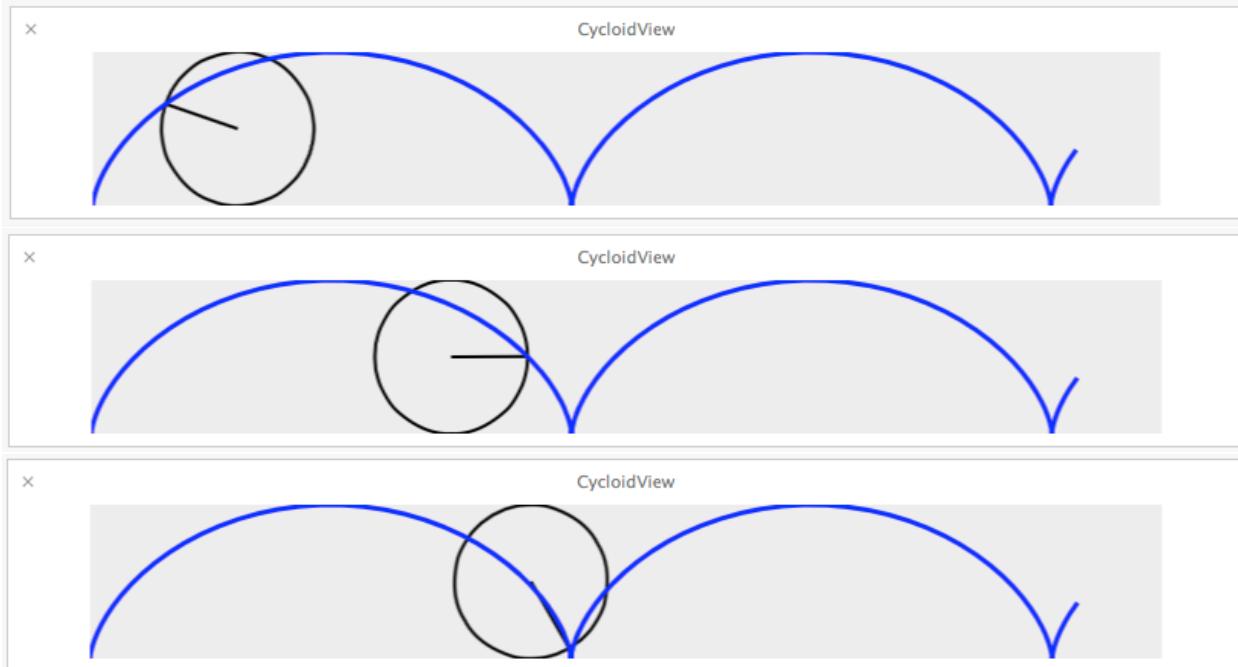
```
1 XCPShowView("CycloidView", view)
```

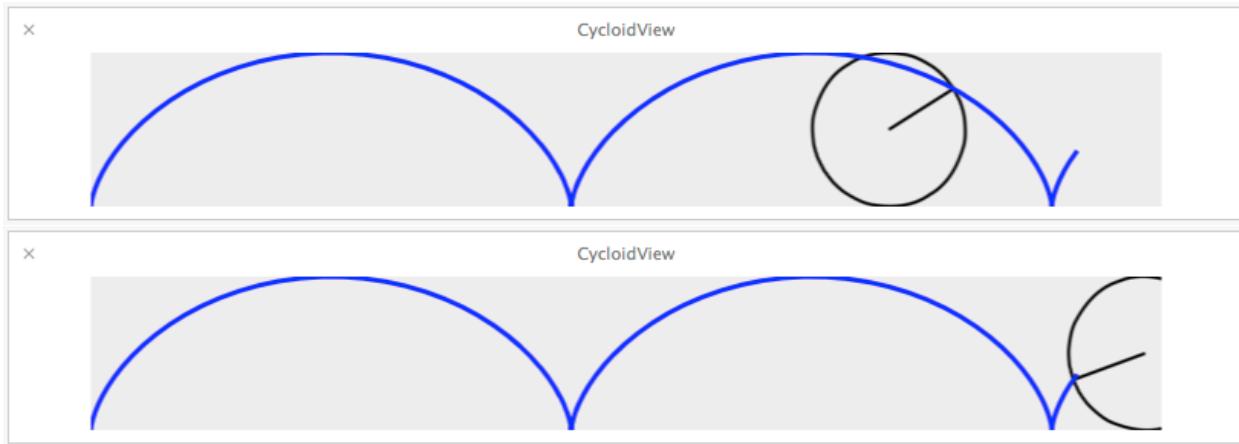
This function automatically adds a view to the timeline, and then delays the end of execution within the timeline, until the timeout value specified in the lower right hand corner of the timeline:



Timeline Timeout

You will now see the wheel animating across the view. You can also then use the slider at the bottom of the timeline to track backwards and forwards in time. The frames of the view animation are captured so you can see exactly how the animation is behaving.





The `XCPShowView()` method is using a method called `XCPSetExecutionShouldContinueIndefinitely` under the hood. This method allows the playground process to continue until either the timeline timeout is reached, or the playground source has changed - at which point the code will be re-run.

Conclusion

Playgrounds offer the potential to be really very useful. They are certainly great for learning Swift - but in that respect they don't offer a lot more than the REPL does. Once you add the full force of the Cocoa and CocoaTouch APIs then there really is a lot to play with. Even in the short sample project, it was possible to go from building an algorithm from simple mathematical principles through to designing a class to encompass that functionality, right up to building a view containing fairly complex animations. At no point do you have to consider any of the mechanics of interacting with an app, which is of huge importance when iterating quickly.

I encourage you to jump in and muck around in a playground if you haven't already. Even without some of the more advanced topics covered here today, they offer huge value.

Don't forget that the source code which accompanies today's article is on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day³⁴](https://github.com/ShinobiControls/iOS8-day-by-day).

³⁴<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 11 :: Asynchronous Testing

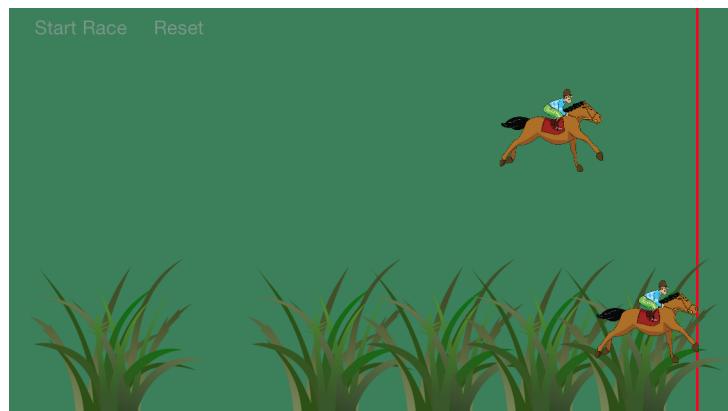
8 11

This book has already taken a look at one of the new features available in XCTest, the testing framework in Xcode, in the shape of profiling unit tests - you can check it out in chapter 6. This isn't the only addition to the framework though - and today's topic will look at one of the other new features in the form of asynchronous testing.

There are many things which can be formulated as asynchronous tasks in your iOS app - rather than returning a result immediately will instead call a block/closure you provide when the result is ready. This is generally good practice, since you can write functions which won't block the current queue whilst they're waiting for some kind of response. However, this can make unit testing far more difficult - rather than calling a function and then checking that the result is correct, you need to somehow check the result of a callback during the unit test.

In the past, there have been add-on testing frameworks which provide this functionality, but in Xcode 6, you can use XCTest directly.

The project which accompanies this chapter is called **HorseRace**, and is a very simple game which animates two horses across the screen. Note, the app itself is pretty much irrelevant - but its API has asynchronous components that can be tested. The code for this project is available on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day³⁵](https://github.com/ShinobiControls/iOS8-day-by-day).



Horse Race

Testing an Asynchronous Method

Asynchronous methods can represent all kinds of things - including background processing, UI events and network events. It's not recommended to unit test for all these things (unit testing the

³⁵<https://github.com/ShinobiControls/iOS8-day-by-day>

network isn't really in the spirit of unit testing), but the process is the same for all.

The **HorseRace** sample project includes a **TwoHorseRaceController** class which is responsible for running the race itself. Note that since unit tests in Swift exist in a separate module, both the types to be tested, and the methods within **must** be marked **public**. Private and internal types and methods are not visible outside the module and therefore not available to your test module.

The **TwoHorseRaceController** includes the following sample async method:

```

1 public func someKindOfAsyncMethod(completionHandler: () -> ()) {
2     dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), {
3         sleep(3)
4         dispatch_async(dispatch_get_main_queue(), {
5             completionHandler()
6         })
7     })

```

This method is a very simple async method, which doesn't actually do anything. It accepts a closure with the signature `() -> ()` as the `completionHandler`. It then sleeps for three seconds on a background queue, before calling the completion handler closure. Note that the `someKindOfAsyncMethod()` will return almost instantaneously, but the callback won't occur until the sleep timer has completed.

Writing a test for this method to check that the `completionHandler` closure is called is actually really simple. The core of asynchronous tests in `XCTest` is the `XCTTestExpectation` class. You can create multiple instances of these, and when each one of them has been completed you call the `fulfill()` method on it. At the end of the test you set up a timer which will wait either for the timer to run out or all the expectations to be fulfilled - whichever occurs first. This determines the pass/fail state of the test.

The following is a demonstration of creating a test for the aforementioned async method:

```

1 func testBasicAsyncMethod() {
2     // Check that we get called back as expected
3     let expectation = expectationWithDescription("Async Method")
4
5     raceController.someKindOfAsyncMethod({
6         expectation.fulfill()
7     })
8
9     waitForExpectationsWithTimeout(5, handler: nil)
10 }

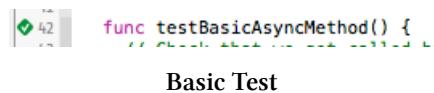
```

- Use `expectationWithDescription()` on `XCTestCase` to create an `XCTTestExpectation` This expectation must be fulfilled in order for the test to pass.
- `raceController` is a property on the test class which is populated in the `setUp()` method:

```
1 override func setUp() {
2     super.setUp()
3     // Put setup code here. This method is called before the invocation of
4     // each test method in the class.
5
6     // Get hold of the view controller
7     let window = UIApplication.sharedApplication().delegate?.window!
8     viewController = window.rootViewController as? ViewController
9     raceController = viewController.raceController
10 }
```

- The test method is invoked, with a closure that simply fulfills the expectation.
- `waitForExpectationsWithTimeout()` is used to tell the test runner how long it should wait for the expectations to be fulfilled. It's important to have a timeout so that your tests don't hang when they aren't performing as expected.

The asynchronous tests are run in exactly the same was as other unit tests in Xcode. You'll notice that there will be a pause when it hits this test, whilst it waits the three seconds for the expectation to be fulfilled. If you set the timeout to below three seconds then the test will fail - since the expectation wasn't fulfilled before the timeout occurred. Note that if all the expectations are fulfilled before the timeout then the test runner will continue on to the next test - it doesn't wait until the timeout for every test.



Multiple Expectations

The last section used an expectation to await a callback - but there are occasions where you want multiple callbacks - each with a different argument. Take, for example, the `startRace(maxDuration:, horseCrossedLineCallback:)` method, which has a closure of the form `((Horse) -> Void)?` for its second argument. This closure will be called each time a horse crosses the line, completing the race. In this very simple game, every horse is guaranteed to cross the line exactly once, and this closure will not be invoked for any other reason. The following test method will assure this:

```
1 func testRaceCallbacks() {
2     // The horse race controller should callback each time a horse completes
3     // the race.
4     let h1Expectation = expectationWithDescription("Horse 1 should complete")
5     let h2Expectation = expectationWithDescription("Horse 2 should complete")
6
7     raceController.startRace(3, horseCrossedLineCallback: {
8         (horse: Horse) in
9         switch horse.horseView {
10            case self.viewController.horse1:
11                h1Expectation.fulfill()
12            case self.viewController.horse2:
13                h2Expectation.fulfill()
14            default:
15                XCTFail("Completion called with unknown horse")
16        }
17    })
18
19    waitForExpectationsWithTimeout(5, handler: nil)
20 }
```

1. Since this is a two-horse race, you need to expectations. Each one represents the event of a horse crossing the line.
2. In the callback, you switch on which horse has crossed the line - and then fulfill the appropriate expectation.
3. If the callback is invoked with a different horse then the test will fail - via use of the `XCTFail()` function.
4. `waitForExpectationsWithTimeout()` is used in the same was as before. Since there are now two expectations defined within the scope of the test function, they will both have to be fulfilled for the test to pass.

Key-Value Observation Expectation

One of the common places that you might want to use asynchronous testing is to assert that a property on an object should have changed. In CocoaTouch notifications about property changes are provided by the Key-Value Observation (KVO) architecture. You could quite easily implement a KVO pattern with the new async tools that you've learnt about already, however, it's quite involved. Since KVO changes are quite a common async operation, the XCTest framework provides a convenience method for creating an expectation which uses KVO. The following test demonstrates it in action:

```

1 func testResetButtonEnabledOnceRaceComplete() {
2     let expectation = keyValueObservingExpectationForObject(
3             viewController.resetButton, keyPath: "enabled",
4             expectedValue: true)
5
6     // Simulate tapping the start race button
7     viewController.handleStartRaceButton(viewController.startRaceButton)
8
9     // Wait for the test to run
10    waitForExpectationsWithTimeout(5, handler: nil)
11 }

```

This test is ensuring that the reset button becomes enabled once the race has finished: 1. The `keyValueObservingExpectationForObject(keyPath:, expectedValue:)` method is used to create an expectation which will wait for the given key path to be updated, and that it should change to the specified value. 2. The race is then kicked off, by simulating a tap on the `startRaceButton`. 3. Again, `waitForExpectationsWithTimeout(handler:)` is used to set a timeout for the test.

Conclusion

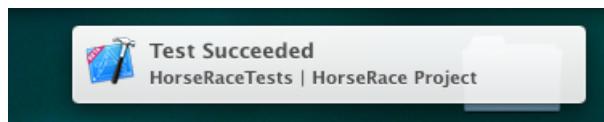
You can run all three of these tests by running the horse race app and seeing them in action. Importantly, these tests demonstrate how you can test parts of the user interaction which would be asynchronous - i.e. async testing doesn't just work for long running background processing and network operations.

```

21 class HorseRaceTests: XCTestCase {
22

```

All Tests Green



All Tests Passed

If you're not already starting to use asynchronous APIs in your apps then it might be time to start considering them - certainly they have suddenly become far easier to test than in previous versions of XCode.

It's great to see that Apple has been concentrating on improving the testing tools available within the IDE. It shows a commitment to modern software development and writing high-quality code.

The code for the **HorseRace** app is available at github.com/ShinobiControls/iOS8-day-by-day³⁶.

³⁶<https://github.com/ShinobiControls/iOS8-day-by-day>

Grab it and take a look at how the tests work. Whatever you do, don't rate it as the recommended way to build a game like this - the game is very much secondary to the tests here :)

Day 12 :: HealthKit

8 12

One of the more consumer-oriented features introduced in iOS 8 is that of the Health app. It was featured in the keynote at WWDC and has received a fair amount of hype since that point. In fact some of the world's biggest names in healthcare have already put their names behind it as being a huge step forward in modern healthcare.

The underlying technology behind the health app is HealthKit, which is essentially a structured datastore designed specifically for health data. Not only is there a schema for health data, but extra concerns such as access permissions, querying and unit conversion. HealthKit allows app developers to interact with this datastore - to add and query data points and calculate statistics.

In today's article you'll get a whistle-stop tour of some of the features of HealthKit, and see how easy it is to create an app which can save data points and query the datastore. The accompanying app is called **BodyTemple** and is available in the github repo at github.com/ShinobiControls/iOS8-day-by-day³⁷.

Data Structure Overview

The structure of data in HealthKit represents an ontology of pretty much everything related to personal well-being. This includes obvious measurements such as body mass, but also extends to more specialized readings such as blood oxygen saturation and dietary zinc. The underlying database is a set of time-series - consisting of independent time-indexed samples.

In terms of code these samples are represented by the `HKSample` class, which itself is a subclass of the immutable `HKObject` class. `HKObject` is the fundamental type used in HealthKit - and includes properties such as `UUID`, `source` and `metadata`. Since these objects are immutable, they are created using initializers on the `HKQuantitySample` and `HKCategorySample` subclasses. These are fairly self explanatory in their distinction between representing a categorical data type and a quantifiable type (such as body mass). They each have some kind of type property to specify what the value represents (`categoryType` and `quantityType` respectively). The following demonstrates creating a quantity sample for body mass:

³⁷<https://github.com/ShinobiControls/iOS8-day-by-day>

```

1 let weightType = HKObjectType
2             .quantityTypeForIdentifier(HKQuantityTypeIdentifierBodyMass)
3 let weightValue = HKQuantity(unit: HKUnit(fromString: "kg"),
4                             doubleValue: massNumber)
5 let metadata = [ HKMetadataKeyWasUserEntered : true ]
6 let sample = HKQuantitySample(type: weightType, quantity: weightValue,
7                               startDate: datePicker.date, endDate: datePicker.date,
8                               metadata: metadata)

```

- `HKObjectType` is a class which represents the aforementioned type property. It has methods for creating types such as `quantityTypeForIdentifier()` and `categoryTypeForIdentifier()`, which take a framework-defined string constant to specify exactly which type it represents (e.g. `HKQuantityTypeIdentifierBodyMass`).
- A quantity sample obviously has an associated value, and this is provided by the `HKQuantity` class. This requires a double value and an `HKUnit` object which represents the units the double was measured in.
- You can provide a metadata dictionary including details such as the measurement device - here the `HKMetadataKeyWasUserEntered` key shows that the data has been manually entered.
- Finally the `HKQuantitySample` is constructed using the type, quantity and metadata objects you've created, along with a start and end date.

The `HKUnit` class is an incredibly powerful concept, in that it will cause HealthKit to convert between units (provided they represent the same physical quantity) implicitly. This goes both for simple units (such as those representing mass, volume and temperature) and for compound units (such as those representing density). The powerful string parsing of the `HKUnit(fromString:)` initializer is an excellent way to create these objects.

To see the selection of different values that can be recorded, check out the `HKTypeIdentifiers` file. The following represents a small selection:

- Body mass
- Body fat percentage
- Height
- Step count
- Blood glucose
- Blood alcohol content
- Dietary energy consumed
- Body temperature
- Inhaler usage

In addition to quantity and categorical time-series types, you can also store characteristic types, that is those which are incredibly unlikely (or don't) to change over time, for example date of birth, sex and blood type. You would expect that a user would set these in the health app and then they are available to you when reading data from the store.

As well as personal physical measurements, you can also store details of workout sessions through the `HKWorkout` subclass of `HKSample`. This has properties for recording the type of workout, the duration, the energy burned and the distance. Since `HKWorkout` is another subclass of `HKObject` it is also immutable, so these properties are all populated at initialization time. The energy burned and distance are again examples of `HKQuantity` objects, and the activity type is an enum (`HKWorkoutActivityType`) which includes over 50 types of exercise, including:

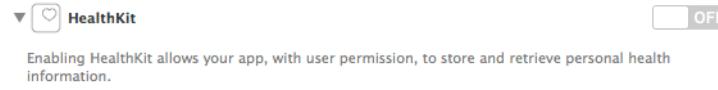
- Archery
- CrossTraining
- Dance
- Fishing
- Hockey
- SkatingSports
- SnowSports
- StairClimbing
- Swimming
- Yoga

You'll notice that through out this run through of the data model, you've been dealing solely with `HKObject` subclasses. This is very much like how CoreData uses `NSManagedObject`, and allows the interface to the data store to be consistent, irrespective of whether you're storing a workout or a blood pressure reading. We'll take a look at this interface soon, but first there's a rather important issue to take a look at - that of permissions.

Permissions

By its nature, any data associated with health is extremely personal, and therefore incredibly sensitive. Whilst having this central store for all health data on a device is great - the security risk could be huge. For example, an ad network knowing the user's body mass, and when they last ate is a huge concern for privacy. It can also be a lot more subtle than that - there is huge correlation between users recording blood sugar levels, and being diabetic - so even knowing that the data *exist* is a leak of personal data.

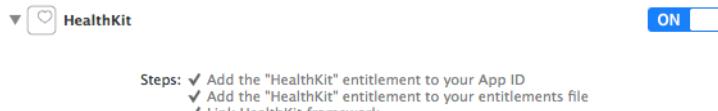
Before your app can use HealthKit, you need to enable it in the **Capabilities** pane of the project settings. This requires that you sign in to your Apple developer account, and then acquires the appropriate entitlements for your app.



Turning on HealthKit will...

- Add the "HealthKit" entitlement to your App ID
- Add the "HealthKit" entitlement to your entitlements file
- Link HealthKit.framework

HealthKit Capabilities



HealthKit Capabilities Enabled

In order to protect users' privacy, HealthKit includes a very granular permissions system, including a super-simple UI. In order for an app to read from or write to HealthKit it has to ask the user for specific permissions. The permissions are based on the types introduced in the last section - e.g. HKQuantityType or HKCharacteristicType. All access to the data is performed through a HKHealthStore object, and it is this that you request permissions from. The following method demonstrates requesting access to the data store:

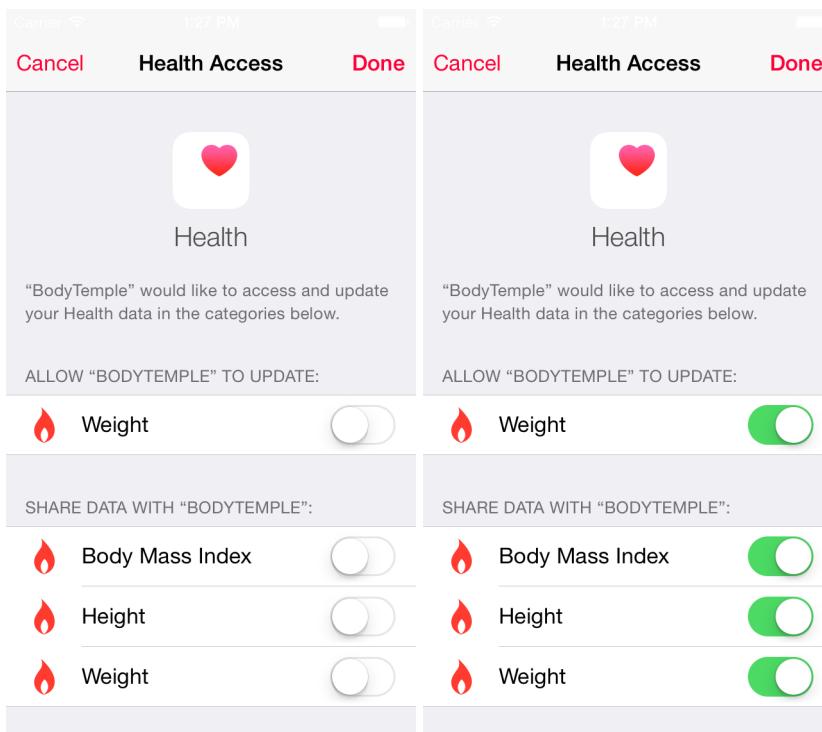
```

1 private func requestAuthorisationForHealthStore() {
2     let dataTypesToWrite = [
3         HKQuantityType.quantityTypeForIdentifier(HKQuantityTypeIdentifierBodyMass)
4     ]
5     let dataTypesToRead = [
6         HKQuantityType.quantityTypeForIdentifier(HKQuantityTypeIdentifierBodyMass),
7         HKQuantityType.quantityTypeForIdentifier(HKQuantityTypeIdentifierHeight),
8         HKQuantityType
9             .quantityTypeForIdentifier(HKQuantityTypeIdentifierBodyMassIndex),
10        HKCharacteristicType
11            .characteristicTypeForIdentifier(HKCharacteristicTypeIdentifierDateOfBirth)
12    ]
13
14    self.healthStore?.requestAuthorizationToShareTypes(
15        NSSet(array: dataTypesToWrite),
16        readTypes: NSSet(array: dataTypesToRead), completion: {
17            (success, error) in
18                if success {
19                    println("User completed authorization request.")
20                } else {
21                    println("The user canceled the authorization request. \(error)")
22                }
23        })
24 }
```

```
23     })
24 }
```

The method `requestAuthorizationToShareTypes(_:, readTypes:, completion:)` accepts `NSSet` objects containing the quantities that you require to share (write) and read. The completion closure is called once the procedure is complete.

When this method is run, then the store checks to see whether it has already asked the user for this configuration of permissions. If it has then it will call the completion block immediately - irrespective of whether permission was granted or not. If it hasn't yet asked the user about this set of permissions then it'll display a modal request page like this:



Note that you cannot discover whether you have permissions to read a given type, and that if you attempt to write a type for which you do not have authorization then you'll get an error back describing the problem.

```
Saving weight
Error: Error Domain=com.apple.healthkit Code=5 "Authorization is not
determined" UserInfo=0x7f8e68d8bb10 {NSLocalizedDescription=Authorization is
not determined}
```

You should only use one `HKHealthStore` throughout your app - this way meaning that you can consolidate your permission requests to one point. In `BodyTemple` this happens in the `TabBarController`, which since it is the top-level view controller owns and propagates the health store to its child view controllers, and requests the permissions required for use of the app.

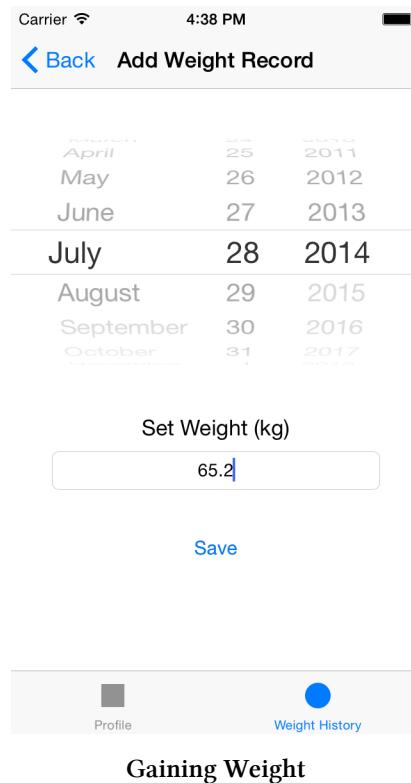
Writing Data

You've already seen the `HKHealthStore` object in the context of requesting appropriate permissions, and this same object is used to both read and write to the data store. Saving data is actually really simple - and this is primarily because of the apparent complexity in the data model. Since every single thing that can be stored in HealthKit is a descendant of `HKObject` then saving is as simple as:

```
1 func saveSampleToHealthStore(sample: HKObject) {
2     println("Saving weight")
3     self.healthStore?.saveObject(sample, withCompletion: {
4         (success, error) in
5         if success {
6             println("Weight saved successfully ⚡")
7         } else {
8             println("Error: \(error)")
9         }
10    })
11 }
```

The `saveObject(_:, withCompletion:)` will attempt to save any `HKObject` to the datastore, and since it is an asynchronous method it has a callback for when it is complete. It is at this point that you would discover that your app doesn't have permission to write (as in the screenshot in the previous section).

In **BodyTemple** hitting save on the following screen will create a `HKQuantitySample` object which is then saved on the screen dismissal:



Once this has completed you can see that it has worked in the Health app:

The image contains two side-by-side screenshots of the Health app. The left screenshot shows the 'Weight' section with a daily average of 168.43 lb and a red 'Show All Data' button. The right screenshot shows the 'All Recorded Data' section with a single entry for Aug 1 at 168.432 lb. Both screenshots show a red 'Show All Data' button at the bottom.

Reading Data

There are several different approaches for requesting data from HealthKit, depending primarily on the type of data you want to retrieve, and whether you want HealthKit to perform any kind of processing on the data.

In the simplest case you might want to retrieve some characteristic data - such as the date of birth of the user:

```
1 func requestAgeAndUpdate() {
2     var error: NSError?
3     let dob = self.healthStore?.dateOfBirthWithError(&error)
4
5     if error {
6         println("There was an error requesting the date of birth: \(error)")
7         return
8     }
9
10    // Calculate the age
11    let now = NSDate()
12    let age = NSCalendar.currentCalendar().components(.YearCalendarUnit,
13                                              fromDate: dob, toDate: now, options: .WrapComponents)
14
15    self.ageLabel.text = "\(age.year)"
16 }
```

There are methods directly on the `HKHealthStore` to get hold of these. The above `requestAgeAndUpdate()` method populates a label with the user's age:



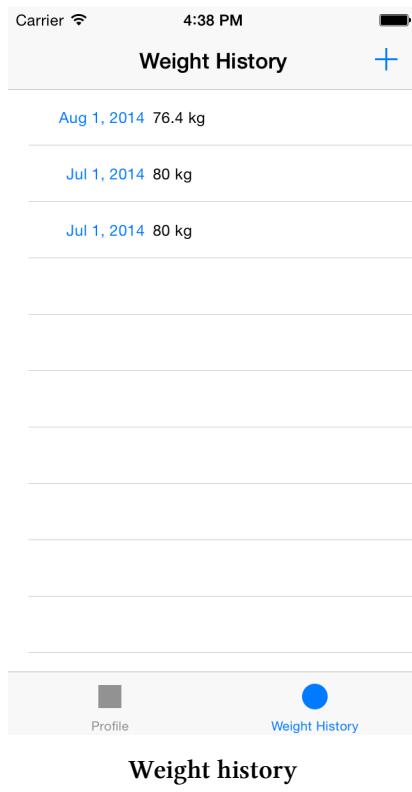
To request sampled data you need to create an `HKQuery`, which itself requires a sample type and predicate, upon which to filter. The following method demonstrates how to get a list of body mass samples for the last 2 months:

```
1 func performQueryForWeightSamples() {
2     let endDate = NSDate()
3     let startDate = NSCalendar.currentCalendar()
4         .dateByAddingUnit(.CalendarUnitMonth, value: -2,
5                         toDate: endDate, options: nil)
6
7     let weightSampleType = HKSampleType
8         .quantityTypeForIdentifier(HKQuantityTypeIdentifierBodyMass)
9     let predicate = HKQuery.predicateForSamplesWithStartDate(startDate,
10      endDate: endDate, options: .None)
11
12    let query = HKSampleQuery(sampleType: weightSampleType, predicate: predicate,
13      limit: 0, sortDescriptors: nil, resultsHandler: {
14        (query, results, error) in
15        if !results {
16            println("There was an error running the query: \(error)")
17        }
18        dispatch_async(dispatch_get_main_queue()) {
```

```

19         self.weightSamples = results as [HKQuantitySample]
20         self.tableView.reloadData()
21     }
22   })
23   self.healthStore?.executeQuery(query)
24 }
```

Notice that the query is again asynchronous, and so you must provide a closure to handle the delivery of the results. Note that this will occur on a background queue and therefore you need to make sure that you marshal any UI updates back to the main queue.



This represents a one-off query, but you might also want to perform a long-running query, which will notify you each time new samples appear in the database, by repeatedly calling the result handler closure. This can be useful if you are working with values which have a high sample rate, and are being created by a different app. This is exactly the functionality provided by `HKObserverQuery`, which looks very similar to the `HKSampleQuery` you saw above. Since it is a long-running query it must be canceled by calling the `stopQuery(_)` method on `HKHealthStore`.

Another query type worth mentioning is the anchored query, represented by `HKAnchoredObjectQuery`, which will provide you a pointer to the most recent result returned. You can then re-run the query at a later stage and only get newer results returned. This means that you don't have to repeatedly process the result set to determine whether you have already seen each sample.

The final class worth mentioning with respect to HealthKit is `HKStatistics`, which, as the name suggests, allows you to perform basic statistical treatments of the dataset - i.e. finding the sum/mean. It's important here to be aware of the fact that some measurements are cumulative (that is to say the sum makes sense, e.g. energy intake), whereas some are discrete - where min and max might apply. These concepts are well-defined in HealthKit determined by the `HKQuantityAggregationStyle` enumeration.

You can also generate a collection of statistics (such as total energy consumed per day for the last week) using the `HKStatisticsCollectionQuery`. This is an expansion on a statistics query, but with the concept of time intervals and an anchor date.

Conclusion

On the surface of it the health app seems like a pretty cool idea for iOS 8, but once you get down into HealthKit then you realize that this could be really quite powerful. In creating HealthKit Apple hasn't attempted to create one healthcare app to rule them all, but instead has created a framework through which the important aspects of healthcare monitoring can interoperate. There is only one canonical set of body mass readings for a person, so why do they currently have to be entered for each app? It's also a lot clearer to the end user exactly what healthcare data the app has access to - which is another easy win.

From the point of view of developers, having an easy-to-use pre-existing datastore is huge news in itself. It might not offer the perfect schema for your purposes, but the power of being able to share data entered from other apps is huge.

Admittedly HealthKit is only going affect developers of apps in this space, but it is certainly worth taking a look at it. It gives you an idea of how Apple's APIs are modernizing as the ecosystem moves forward.

The code for the accompanying app is available on github at the usual place on the ShinobiControls github: github.com/ShinobiControls/iOS8-day-by-day³⁸.

³⁸<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 13 :: CoreImage Detectors

8 13

CoreImage never tends to get top-billing when looking at new features in iOS, and this isn't terribly fair. Over the past few years CoreImage has seen some huge advances in performance and functionality, and iOS 8 is no exception. In fact there are some really huge changes, including the ability to create custom filter kernels for the first time. But that's not the focus of today's post - instead we're going to concentrate on new feature detection algorithms.

Last year a face detector was added to **CoreImage** - you can read about how to use it in chapter 18 of [iOS 7 Day-by-Day³⁹](#). At the same time a QR code detector was added to **AVFoundation** - see chapter 16 of the [same book⁴⁰](#), but it's actually quite hard to use **AVFoundation** in the general image case. Well, the advances in **CoreImage** in iOS 8 improve the situation somewhat.

The **CIDetector** class was introduced in iOS 7, but only contained a facial feature detector. iOS 8 augments this with rectangle and QR code detection, which are the subject of today's article.

The accompanying project is an app which takes a live camera view and runs the frames through **CoreImage**, detecting either rectangles or QR codes. The code is written in such a way that it is obvious how you can adapt it to work on images you've grabbed from sources other than the camera - making it particularly versatile. As usual, the source code is available in the iOS8-day-by-day repo on github at [github.com/ShinobiControls/iOS8-day-by-day⁴¹](#).



Note that since the app uses a camera the app has to be run on a physical device.

Detecting Rectangles

Detecting rectangles in images is often one of the first parts of a computer vision algorithm - whether it be automatic business card interpretation, or road sign processing. Although rectangle detection sounds like it should be a really simple process, as with many problems in computer vision, it's far harder than you might expect. So it's great that Apple have implemented an efficient algorithm as part of the CoreImage detectors.

The main class associated with CoreImage detectors is the aptly-named **CIDetector**. The same class is used for all the different types of detectors, and is instantiated with the **CIDetector(ofType: ,**

³⁹<https://leanpub.com/iOS7DayByDay>

⁴⁰<https://leanpub.com/iOS7DayByDay>

⁴¹<https://github.com/ShinobiControls/iOS8-day-by-day>

`context:, options:)` initializer. The type argument is a string, which for a rectangle detector is `CIDetectorTypeRectangle`. The options argument is a dictionary of settings associated with this detector. The following method created a `CIDetector` to be used for detecting rectangles:

```

1 func prepareRectangleDetector() -> CIDetector {
2     let options = [CIDetectorAccuracy: CIDetectorAccuracyHigh,
3                     CIDetectorAspectRatio: 1.0]
4     return CIDetector(ofType: CIDetectorTypeRectangle,
5                        context: nil, options: options)
6 }
```

You can see that the options here are specifying the accuracy as high, with the `CIDetectorAccuracy` key, and the `CIDetectorAspectRatio` key is used to specify that you're searching for squares. This aspect ratio doesn't mean that you're only looking for squares, but it will be used in the ranking of possible rectangles to determine which is the most likely candidate. For example, if you know that you're going to use the detector for business cards then setting the aspect ratio to `2.0` will likely yield better results.

Once you've created a `CIDetector` it's actually really simple to use. The method `featuresInImage()` takes a `CIImage` and then returns an array of `CIFeature` objects (well, a subclass of) which represent the detected objects. In the case of a rectangle detector, the `CIFeature` subclass is `CIRectangleFeature`, which has `CGPoint` properties for each of the four corners of the detected rectangle.

The following method demonstrates how you can use the detector to find a rectangle in a supplied `CIImage`:

```

1 func performRectangleDetection(image: CIImage) -> CIImage? {
2     var resultImage: CIImage?
3     if let detector = detector {
4         // Get the detections
5         let features = detector.featuresInImage(image)
6         for feature in features as [CIRectangleFeature] {
7             resultImage = drawHighlightOverlayForPoints(image,
8                  topLeft: feature.topLeft, topRight: feature.topRight,
9                  bottomLeft: feature.bottomLeft, bottomRight: feature.bottomRight)
10        }
11    }
12    return resultImage
13 }
```

This unwraps the option `detector`, before using the `featuresInImage()` method to perform the detection itself. At the time of writing, the rectangle detector will only ever detect one rectangle in

an image, so the `features` array will have either exactly one or zero `CIRectangleFeature` objects in it.

This method returns a new `CIIImage`, which contains a red patch overlaid on the source image over the position of the detected rectangle. This uses the utility method `drawHighlightOverlayForPoints()` method:

```
1 func drawHighlightOverlayForPoints(image: CIIImage, topLeft: CGPoint,
2                                     topRight: CGPoint, bottomLeft: CGPoint,
3                                     bottomRight: CGPoint) -> CIIImage {
4     var overlay = CIIImage(color: CIColor(red: 1.0, green: 0, blue: 0, alpha: 0.5))
5     overlay = overlay.imageByCroppingToRect(image.extent())
6     overlay = overlay.imageByApplyingFilter("CIPerspectiveTransformWithExtent",
7                                             withInputParameters: [
8         "inputExtent": CIVector(CGRect: image.extent()),
9         "inputTopLeft": CIVector(CGPoint: topLeft),
10        "inputTopRight": CIVector(CGPoint: topRight),
11        "inputBottomLeft": CIVector(CGPoint: bottomLeft),
12        "inputBottomRight": CIVector(CGPoint: bottomRight)
13    ])
14    return overlay.imageByCompositingOverImage(image)
15 }
```

This method creates a colored image, and then uses the perspective transform filter to map it to the points provided. It then creates a new `CIIImage` by overlaying this colored image with the source image.

In the `LiveDetection` sample project, the `performRectangleDetection()` method is used as part in the `CoreImageVideoFilter` class to run the detector on each of the frames received from the camera, before rendering it on screen. This class is a little more involved than you might expect it to be, and it isn't within the scope of this article to go in to much detail, however an overview might be helpful.

1. An `AVFoundation` pipeline is created which uses the camera as input, and provides a pixel buffer of each frame to a delegate method.
2. In this delegate method a `CIIImage` is created from this pixel buffer.
3. The provided filter function (which takes an input `CIIImage` and returns a new `CIIImage`, much like a map function) is passed the current image.
4. The image is cropped to match the view size it is to be rendered in.
5. The `CIIImage` is then rendered on the OpenGL ES surface using a pre-created `CIRenderContext`.
6. Rinse and repeat for each frame received from the camera.

If you need to do live video processing using `CoreImage` then it might be worth taking a look at this class in greater detail, but if you don't then the `performRectangleDetection()` method just takes a `CIIImage`, which you can create using one of the many constructors.

Running this app up will kick off the rectangle detection right away, and you'll see results like the following on the screen:



Rectangle Detector

The performance is pretty good for real-time use - certainly the demo app copes well on an iPad 3.

Detecting QR Codes

Once you've understood rectangle detection, then you'll find that QR code detection is very similar. You again need to create a `CIDetector` instance, this time with the detector type string being `CIDetectorTypeQRCode`:

```
1 func prepareQRCodeDetector() -> CIDetector {
2     let options = [CIDetectorAccuracy: CIDetectorAccuracyHigh]
3     return CIDetector(ofType: CIDetectorTypeQRCode, context: nil, options: options)
4 }
```

Exactly the same procedure is used to actually perform the detection - calling `featuresInImage()` and providing a `CIImage`. This time it will return an array of `CIQRCodeFeature` objects - each of which has the same corner points, with the addition of `messageString`.

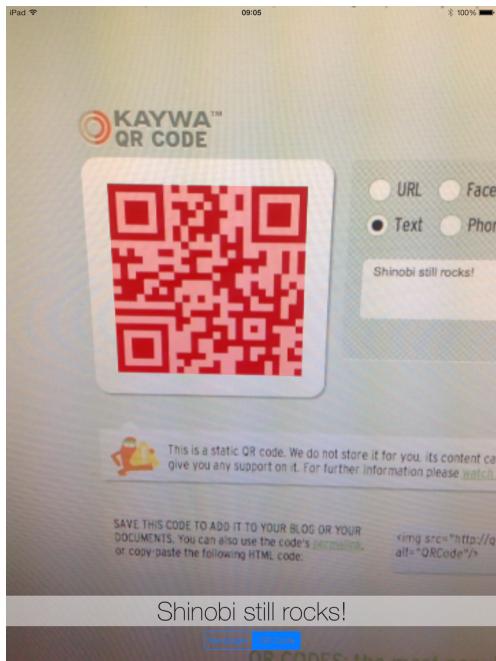
The following method will again highlight the location of the barcode within the provided image, but also return the decoded string:

```
1 func performQRCodeDetection(image: CIImage) -> (outImage: CIImage?,  
2                                                 decode: String) {  
3     var resultImage: CIImage?  
4     var decode = ""  
5     if let detector = detector {  
6         let features = detector.featuresInImage(image)  
7         for feature in features as [CIQRCodeFeature] {  
8             resultImage = drawHighlightOverlayForPoints(image,  
9                 topLeft: feature.topLeft, topRight: feature.topRight,  
10                bottomLeft: feature.bottomLeft, bottomRight: feature.bottomRight)  
11            decode = feature.messageString  
12        }  
13    }  
14    return (resultImage, decode)  
15 }
```

In the **LiveDetection** app this method is used in the same way as the rectangle detection method, but with the addition that the text of a label is updated with the current decoded value:

```
1 detector = prepareQRCodeDetector()  
2 videoFilter.applyFilter = {  
3     image in  
4     let found = self.performQRCodeDetection(image)  
5     dispatch_async(dispatch_get_main_queue()) {  
6         if found.decode != "" {  
7             self.qrDecodeLabel.text = found.decode  
8         }  
9     }  
10    return found.outImage  
11 }
```

If you run the app up, select QR code, and then point the camera at a QR code you can see the result:



QR Code Detector

Conclusion

I think that CoreImage is really cool, and offers loads of functionality that developers just aren't aware of, but maybe that's just me and my previous life in the world of computer vision. These new detectors are just part of what's available, and are a much easier way of performing these detections than using **AVFoundation**. Although you might not be able to use it directly in your apps it's a fun thing to have a play with - and you never know when it might be useful.

As ever, the code for the **LiveDetection** demo app is available to fork, clone or download from the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁴².

⁴²<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 14 :: Rotation Deprecation



It might be scary to discover that as part of embracing adaptive UI in iOS 8, the view controller rotation methods you've come to love have all been deprecated. In fact the new approach to rotation is far simpler and much more coherent.

In today's article you'll learn about how iOS 8 deals with device rotation, and what you need to do to get your apps up to scratch.

The accompanying app is a simple app called `RotateToDeprecate`, which demonstrates how you can build a layout which rotates elegantly and then further customize the rotation behavior. You can find the source code for the app in the iOS 8 Day-by-Day github repo at [github.com/ShinobiControls/iOS8-day-by-day⁴³](https://github.com/ShinobiControls/iOS8-day-by-day).

Auto Layout to the rescue

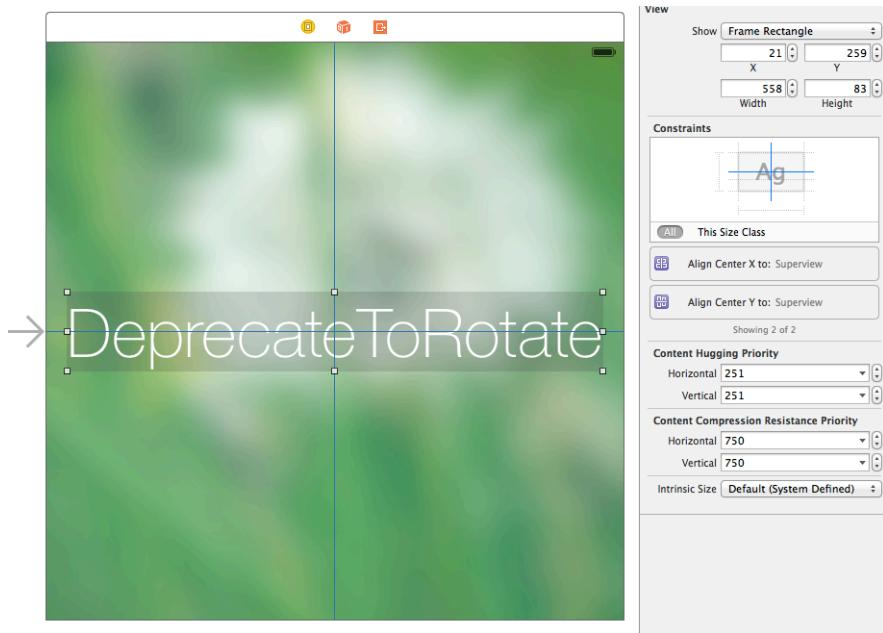
The underlying principle for rotation in iOS 8 is that from the point of view of the content, a rotation is simply a bounds change. Nothing more complicated than that.

If you think about it this makes perfect sense: the window needs to know that the device has been rotated so that it too can rotate to match the new orientation. It also needs to resize since there are currently no iOS devices with square screens. Now, from the point of view of the content of the window - i.e. your view controller hierarchy, you just need to respect the change of size of your parent - the fact that it is a rotation is not particularly relevant.

Therefore in the simplest case, dealing with rotation can be covered by having a suitably adaptive layout - which leads you to using auto layout, storyboards and size classes. If a rotation causes a size class change then your view controller will automatically transition from one layout to the new one. Note that if the rotation causes a trait collection change then your views will all get calls to `traitCollectionDidChange()` and your view controllers calls to `willTransitionToTraitCollection()`.

For example, in the `RotateToDeprecate` app, the constraints on the large label are set up simply to ensure that it is centered both vertically and horizontally in its superview:

⁴³<https://github.com/ShinobiControls/iOS8-day-by-day>



Label Constraints

These constraints will be re-evaluated on rotation (or indeed on any superview bounds change) to ensure that they are always valid. This means that the label will be centered irrespective of orientation:



iPhone Preview

This is great news - and in most cases this is probably all you need to do. No longer do you manually have to relayout your content in one of the rotation methods. However, what if you do want to have some custom behavior? Read on...

Customizing rotation behavior

You've already seen one of the ways that you can customize behavior of some rotations - via the `willTransitionToTraitCollection()` method (which is on the `UIContentContainer` protocol, which is itself adopted by `UIViewController`). This method will be called *whenever* the trait collection for a container changes, which is true of rotations on an iPhone. It provides you with a transition coordinator which allows you to hook in to the transition animation.

However, this won't be true for all rotations - for example, iPads have the same size classes irrespective of their orientation, and therefore won't be transitioning between trait collections on rotation. Helpfully there is another method on the `UIContentContainer` protocol in the shape of `viewWillTransitionToSize(_:, coordinator:)`, which will be called whenever the container is transitioning to a new size. You can override this method to customize the rotation functionality.

The second argument to this method is a `UIViewControllerTransitionCoordinator`, which you'll no doubt remember from the forest of protocols associated with custom view controller transitions. In iOS 8 their functionality has been extended to supporting transitioning within a view controller as well as between two view controllers.

If you wish to perform some custom animations you can use the `animateAlongsideTransition(_:, completion:)` method which allows you to hook into the system animation itself.

For example, in `DeprecateToRotate`, there are 2 possible background images - one which should be associated with a **tall** orientation, and one with **wide**. The following method performs the switch between the background images:

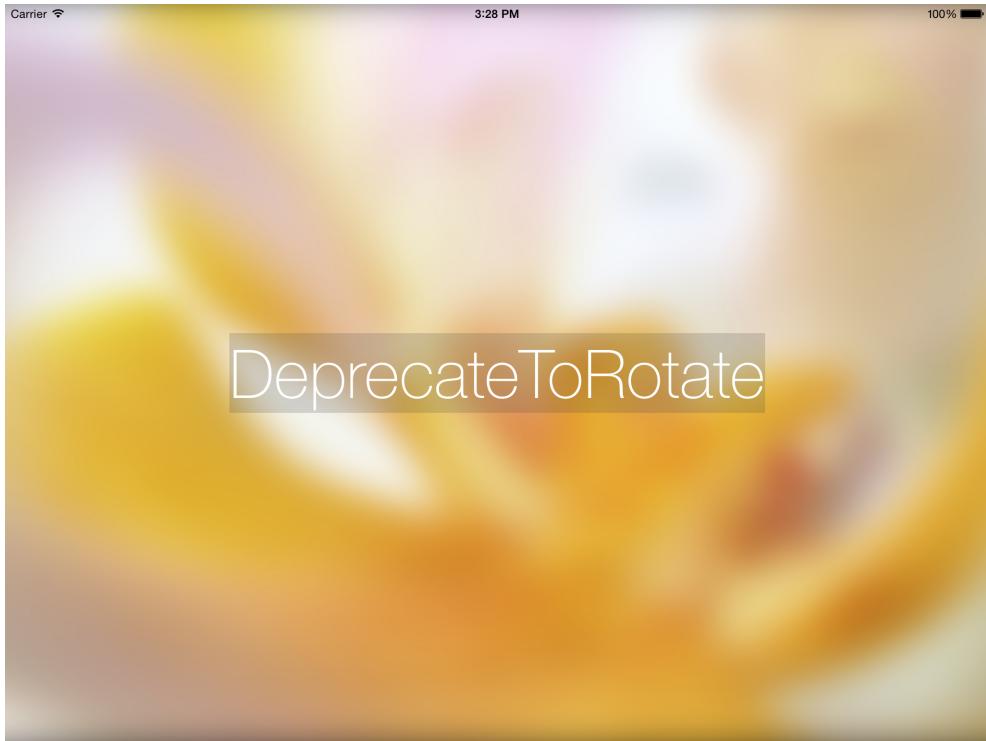
```
1 override func viewWillTransitionToSize(size: CGSize,
2                                         withTransitionCoordinator coordinator:
3                                         UIViewControllerTransitionCoordinator) {
4     let transitionToWide = size.width > size.height
5     let image = UIImage(named: transitionToWide ? "bg_wide" : "bg_tall")
6
7     coordinator.animateAlongsideTransition({
8         context in
9         // Create a transition and match the context's duration
10        let transition = CATransition()
11        transition.duration = context.transitionDuration()
12
13        // Make it fade
14        transition.timingFunction = CAMediaTimingFunction(name:
15                                              kCAMediaTimingFunctionEaseInEaseOut)
16        transition.type = kCATransitionFade
17        self.bgImageView.layer.addAnimation(transition, forKey: "Fade")
18    }
```

```
19 // Set the new image
20 self.bgImageView.image = image
21 }, completion: nil)
22 }
```

1. First we determine whether the transition is moving to a **wide** configuration, by looking at the provided `size` argument.
2. This is used to choose which background image should be shown.
3. The `animateAlongsideTransition()` method is used to animate from the current image to the new one.
4. A `CATransition` is used to get a nice fade effect. Note that you are provided with a `UIViewControllerTransitionCoordinatorContext`, from which you can find the duration of the animation.



Portrait iPad



Landscape iPad

You'll note that throughout this there has been no mention of the actual rotation - which for 90% of cases is fantastic. However, there might be occasions that your animation requires knowledge of the rotation - for example complex rotation effects for parts of your content. In these instances you can use the `targetTransform()` method on the transition coordinator context to obtain the `CGAffineTransform` which represents the rotation. This will either be a +90, -90 or 180 rotation or the identity transform.

Conclusion

The thought of having to fix all your custom rotation code might seem daunting at first, but in actual fact, this new rotation behavior is far more logical. It enables you to not even think about rotation in 90% of the cases. You can consider your content, and how it should be displayed in different layouts - irrespective of not only the orientation but the device itself.

The sample project which accompanies today's post is available on the ShinobiControls github page at github.com/ShinobiControls/iOS8-day-by-day⁴⁴.

⁴⁴<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 15 :: NSFormatter



Many apps that we built represent data flow in some form or another. However, since apps are primarily visual it's often important to display this data to the user. This requires converting numbers, dates etc to strings which the user will understand. This is more complicated than it might sound - since different locales will have different formatting conventions, different names and even different calendars.

Cocoa has traditionally been very strong in this area - with the `NSFormatter` abstract class - and more specifically the `NSNumberFormatter` and `NSDateFormatter` concrete implementations. You'll almost certainly have come across them in your app development - e.g. converting an `NSDate` object to a string for display in an `UILabel`.

iOS 8 adds to this family of `NSFormatter` concrete implementations with some new date/time formatters and some physical quantity formatters. This article will take a quick tour of the new formatters, and how to use them.

The sample code which accompanies this post is in the form of a Xcode 6 playground, and is available in the iOS 8 Day-by-Day git repo on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁴⁵](https://github.com/ShinobiControls/iOS8-day-by-day).

Temporal Formatters

Joining the hugely popular `NSDateFormatter` are two new formatters - `NSDateComponentsFormatter` and `NSDateIntervalFormatter`.

NSDateComponentsFormatter

`NSDateFormatter` is useful for creating a string which represents a specific point in time (e.g. "12th June 2010"), but it can't be used for specifying some kind of temporal duration (e.g. "12 days 13 hours and 12 minutes"). These durations are still very much in need of a formatter, since they need to be carefully localized. Enter `NSDateComponentsFormatter`, which can create a suitably localized string for an instance of `NSDateComponents`.

⁴⁵<https://github.com/ShinobiControls/iOS8-day-by-day>

```

1 let dateComponentsFormatter = NSDateComponentsFormatter()
2 let components = NSDateComponents()
3 components.hour    = 2
4 components.minute = 45

```

`NSDateComponentsFormatter` has a `unitsStyle` property which can be any one of `.Positional`, `.Abbreviated`, `.Short`, `.Full`, or `.SpellOut`:

The following snippet demos the different unit styles:

```

1 let dcfStyles: [NSDateComponentsFormatterUnitsStyle] =
2   [.Positional, .Abbreviated, .Short, .Full, .SpellOut]
3 for style in dcfStyles {
4   dateComponentsFormatter.unitsStyle = style
5   dateComponentsFormatter.stringFromDateComponents(components)
6 }

```

Units Style	Result
<code>.Positional</code>	2:45
<code>.Abbreviated</code>	2h 45m
<code>.Short</code>	2 hrs, 45 mins
<code>.Full</code>	2 hours, 45 minutes
<code>.SpellOut</code>	two hours, forty-five minutes

By default these string are localized to the default locale, but you can override this by providing a specific `NSCalendar` instance:

```

1 let calendar = NSCalendar.currentCalendar()
2 calendar.locale = NSLocale(localeIdentifier: "th-TH")
3 dateComponentsFormatter.calendar = calendar
4 dateComponentsFormatter.stringFromDateComponents(components)
5 // => ၂၁၁ ၂၀၁၅၁၁ ၁၁၁ ၂၀၁၅၁၁၁၁၁ ၁၁၁၁

```

You can also specify that “approximation” and “remaining” phrases be added to the output string:

```

1 dateComponentsFormatter.includesApproximationPhrase = true
2 dateComponentsFormatter.stringFromDateComponents(components)
3 // => About 2 hrs, 45 mins
4 dateComponentsFormatter.includesTimeRemainingPhrase = true
5 dateComponentsFormatter.stringFromDateComponents(components)
6 // => About 2 hrs, 45 mins remaining

```

NSDateIntervalFormatter

`NSDateComponentsFormatter` is really useful for creating duration strings, but if you want to specify a specific time interval, then `NSDateIntervalFormatter` is your friend. It takes two `NSDate` objects, and again has a selection of different time and date styles, which can be set independently for the time and the date:

```

1 let dateIntervalFormatter = NSDateIntervalFormatter()
2 let now = NSDate()
3 let longTimeAgo = NSDate(timeIntervalSince1970: 0.0)
4 let difStyles: [NSDateIntervalFormatterStyle] =
5   [.NoStyle, .ShortStyle, .MediumStyle, .LongStyle, .FullStyle]
6 for style in difStyles {
7   dateIntervalFormatter.dateStyle = style
8   dateIntervalFormatter.timeStyle = style
9   dateIntervalFormatter.stringFromDate(longTimeAgo, toDate: now)
10 }

```

Style	Result
.NoStyle	
.ShortStyle	1/1/70, 1:00 AM - 7/30/14, 9:32 AM
.MediumStyle	Jan 1, 1970, 1:00:00 AM - Jul 30, 2014, 9:32:35 AM
.LongStyle	January 1, 1970, 1:00:00 AM GMT+1 - July 30, 2014, 9:32:35 AM GMT+1
.FullStyle	Thursday, January 1, 1970, 1:00:00 AM GMT+01:00 - Wednesday, July 30, 2014, 9:32:35 AM British Summer Time

Physical Quantity Formatters

A completely new addition to the `NSFormatter` family in iOS are the physical quantity formatters, which have primarily been added in support of HealthKit. They all contain the ability to format numbers with a selection of relevant units.

NSLengthFormatter

`NSLengthFormatter` is used to format numbers as lengths, and includes localizable units such as metres, inches, yards and miles. The `stringFromMeters()` method will convert from a value in meters to the unit appropriate for the current locale, whereas the `stringValue()` method takes a units value, allowing you to specify what the number represents, and should be output:

```

1 let lengthFormatter = NSLengthFormatter()
2 lengthFormatter.stringFromMeters(1.65)
3 // => 1.804 yd
4 let lfUnits: [NSLengthFormatterUnit] =
5   [.Millimeter, .Centimeter, .Meter, .Kilometer, .Inch, .Foot, .Yard, .Mile]
6 for unit in lfUnits {
7   lengthFormatter.stringValue(15.2, unit: unit)
8   lengthFormatter.unitStringFromValue(10.3, unit: unit)
9 }
```

`NSLengthFormatterUnit` allows you to specify which unit type should be used:

Unit	Result
.Millimeter	15.2 mm
.Centimeter	15.2 cm
.Meter	15.2 m
.Kilometer	15.2 km
.Inch	15.2 in
.Foot	15.2 ft
.Yard	15.2 yd
.Mile	15.2 mi

You can also specify unit styles as well, which allow you to control the length of the units themselves:

```

1 let unitStyles: [NSFormattingUnitStyle] = [.Short, .Medium, .Long]
2 for style in unitStyles {
3   lengthFormatter.unitStyle = style
4   lengthFormatter.stringValue(1.65, unit: .Meter)
5 }
```

Style	Result
.Short	1.65m
.Medium	1.65 m
.Long	1.65 meters

In some locales, the units used for measuring a person's height are different from those used to measure other lengths - e.g. human height is rarely measured in yards. `NSLengthFormatter` has a boolean `forPersonHeightUse` property which controls this behavior.

If you only require units, as opposed to the number and units then the `unitsStringFromValue(, unit:)` method allows will do that for you. It respects the current `unitStyle`, and uses the provided `value` to determine whether or not the unit should be plural.

NSMassFormatter

In many respects, the `NSMassFormatter` is very similar to the `NSLengthFormatter`, but representing the physical quantity of mass instead of distance. It too can convert from the standard unit (kilograms) to the unit used in the appropriate locale, and also has a selection of units available for manual use:

```

1 let massFormatter = NSMassFormatter()
2 massFormatter.stringFromKilograms(56.4)
3 // => 124.08 lb
4 let mfUnits: [NSMassFormatterUnit] =
5   [.Gram, .Kilogram, .Ounce, .Pound, .Stone]
6 for unit in mfUnits {
7   massFormatter.stringFromValue(165.2, unit: unit)
8 }
```

Unit	Result
.Gram	165.2 g
.Kilogram	165.2 kg
.Ounce	165.2 oz
.Pound	165.2 lb
.Stone	165.2 st

And again, you can specify the style of the units:

```

1 massFormatter.stringFromKilograms(76.2)
2 for style in unitStyles {
3   massFormatter.unitStyle = style
4   massFormatter.stringFromValue(34.2, unit: .Kilogram)
5 }
```

Style	Result
.Short	34.2kg
.Medium	34.2 kg
.Long	34.2 kilograms

Again, some locales tend to use different (some may say strange) units to measure the mass of a human, and therefore there is a `forPersonMassUse` property which will switch the formatter into ‘human’ mode.

NSEnergyFormatter

The final new formatter is for formatting energy, and its existence is driven by the requirement to measure the energy content of food in HealthKit. It shares many of the same features of the previous two physical quantity formatters.

The `stringFromJoules()` method will convert to the unit it thinks is most appropriate for the locale, but once again you can specify units with `stringValue(unit:)`:

```

1 let energyFormatter = NSEnergyFormatter()
2 energyFormatter.stringFromJoules(42.5)
3 // => 10.158 cal
4 let efUnits: [NSEnergyFormatterUnit] =
5   [.Joule, .Kilojoule, .Calorie, .Kilocalorie]
6 for unit in efUnits {
7   energyFormatter.stringValue(54.2, unit: unit)
8 }
```

Unit	Result
.Joule	54.2 J
.Kilojoule	54.2 kJ
.Calorie	54.2 cal
.Kilocalorie	54.2 kcal

In the same way as before, you can specify a selection of unit styles:

```

1 for style in unitStyles {
2   energyFormatter.unitStyle = style
3   energyFormatter.stringValue(5.6, unit: .Kilojoule)
4 }
```

Style	Result
.Short	5.6kJ
.Medium	5.6 kJ
.Long	5.6 kilojoules

For some completely unimaginable reason, when referring to the energy content of food, some locales tend to invent a new unit, helpfully called “calories”, which is identical to a traditional “kilocalorie”. Luckily, before I go off on a rant, `NSEnergyFormatter` has your back - with the `forFoodEnergyUse` property:

```
1 energyFormatter.forFoodEnergyUse = true
2 energyFormatter.stringFromJoules(4200)
3 // => 1.004 Cal
```

Conclusion

Formatting values is hard enough in the locale you use every day, but attempting to get it right across all locales is near-enough impossible. In the past numbers and dates have been possible through the existing `NSFormatter` subclasses, but iOS 8 adds some great new formatters to ease this pain.

This new functionality isn’t something that you’re likely to be able to rush out and use immediately to implement some cool new feature, but it’s one of those things that is incredibly useful when you need it. The important part is to remember that these formatters exist, so that next time you are creating an app which needs to represent a distance, or a duration, you know that you can have localization for free.

The code for this post is all available in a playground which you can get from the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁴⁶.

⁴⁶<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 16 :: Navigation Bar Hiding



Today's post is quite a short one - a snippet if you will, but that makes it no less interesting or useful. Back in iOS 7, Safari introduced the ability to show and hide the navigation and tool bars by tapping or swiping. This really allowed the content to shine through - reducing the proportion of the screen dedicated to less-useful chrome at appropriate times.

In iOS 8 the navigation controller has been updated with some additional properties which offer this functionality out-the-box. In today's post you'll learn about how to use it and how you can configure it to behave exactly as you want.

There is a small accompanying project which is available in the iOS 8 Day-by-Day repo on github at [github.com/ShinobiControls/iOS8-day-by-day⁴⁷](https://github.com/ShinobiControls/iOS8-day-by-day).

Navigation Controller Updates

Since the bars usually present in an iOS app are part of a navigation controller the auto-hiding functionality is made available through additions to the `UINavigationController` API.

There are four different "modes" of bar hiding available - hiding on tap, hiding on swipe, hiding based on vertical size class and hiding on keyboard appearance.

Hiding on tap

The simplest implementation is to have the bars disappear when you tap on the view, and this is performed by setting the `hidesBarsOnTap` property to `true`. You can also get hold of the gesture recognizer via the `barHideOnTapGestureRecognizer` property. This means that you can use the gesture delegate methods to use your own gesture recognizers in concert with this 'global' tap recognizer.

Tapping is all well and good, but if you want to provide some UI that the user should be able to interact with via tapping then it won't be clear whether tapping with hide/reveal the bars, or interact with the content. In this scenario then the swipe gesture might be more appropriate.

⁴⁷<https://github.com/ShinobiControls/iOS8-day-by-day>

Hiding on swipe

In the same way that tapping can show/hide the bars, there is also a built in swipe gesture recognizer to perform exactly the same task. In this instance rather than toggling, a swipe up will hide the bars and a swipe down will reveal them again.

You can enable this behavior using the `hidesBarsOnSwipe` boolean property, and again you can get hold of the gesture recognizer via the `barHideOnSwipeGestureRecognizer` property.

Hiding when vertically compact

Clearly a fairly common use case would be to hide the bars when there is a restriction on vertical space. In the new world of adaptive layout this corresponds to having a **compact** vertical size class. This functionality is again provided as a boolean property - `hidesBarsWhenVerticallyCompact`. Setting this to true will cause the bars to disappear when the container becomes vertically compact. Note that you can use this together with the tap and swipe properties to get the bars to re-appear in this mode.

Hiding on keyboard appearance

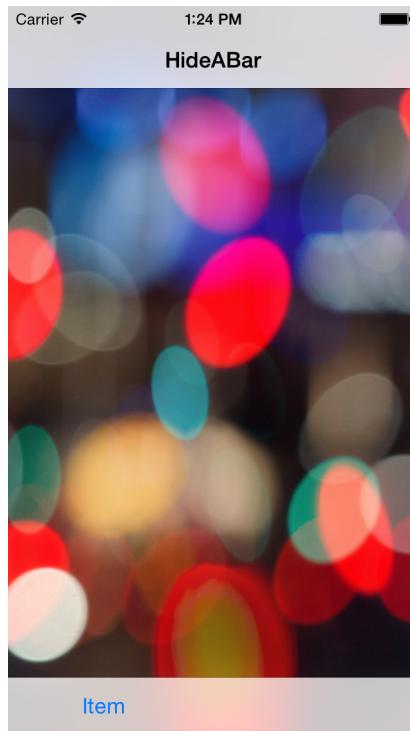
The final mode is controlled by the `hidesBarsWhenKeyboardAppears` boolean property. Setting this to true will cause the bars to disappear when the keyboard animates on screen. The user can bring the bars back simply by tapping in the content area.

Sample app

The sample app which accompanies this article is incredibly simple. It shows a navigation controller which has the tap, swipe and compact height properties set to true:

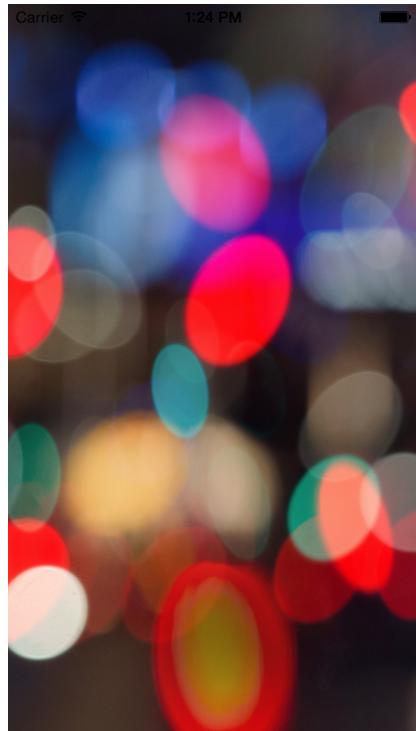
```
1 navigationController?.hidesBarsOnSwipe = true
2 navigationController?.hidesBarsOnTap = true
3 navigationController?.hidesBarsWhenVerticallyCompact = true
```

The app will start with the bars visible by default:



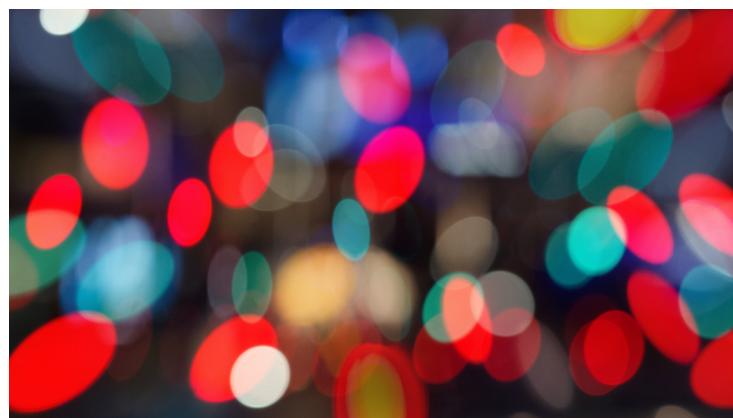
Bars visible

Tapping, or swiping upwards will hide these bars:



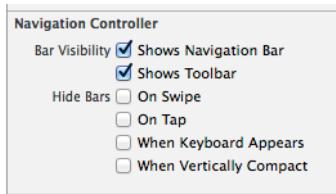
Bars hidden

If you rotate the iPhone then it'll transition to a compact vertical size class and so the bars will hide:



Vertically compact

As well as using the properties in code, the **Attributes Inspector** inside interface builder have some new check boxes to support this new behavior:



IB Check boxes

Conclusion

Short and sweet - this is nice new functionality that's incredibly simple to use. It has well-understood UX and really adopts the 'defer to content' paradigm that was introduced in iOS 7.

You can grab today's sample project from the iOS 8 day-by-day github repo on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁴⁸.

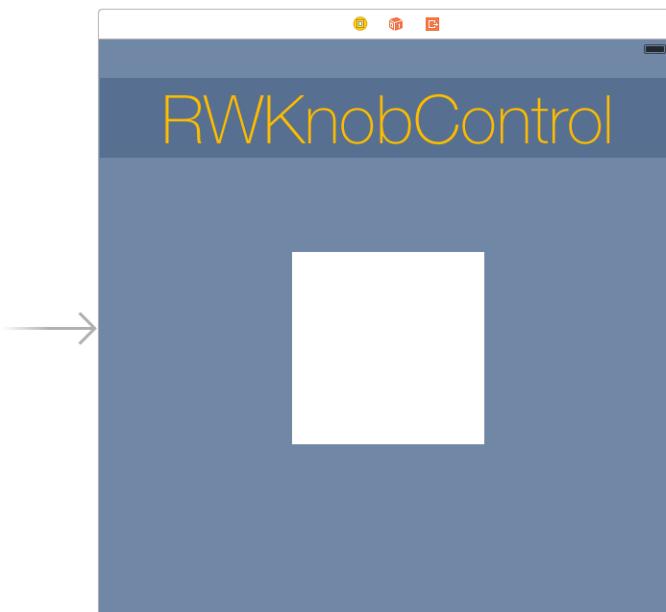
⁴⁸<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 17 :: Live Rendering in Interface Builder



Custom views are a great way of both splitting up your code as a nod towards software design best practices, but also allowing the creation of reusable components. I've talked about this concept a lot in terms of building custom UI Controls - in fact I've given a fair few [talks⁴⁹](#) on the subject.

One of the limitations in the iOS world with this technique is that you instantly lose the ability to work with Interface Builder (IB) in Xcode. Well, you can continue to use it, but you have to work with a large set of blank rectangles:



Storyboard before

However, this has all changed with Xcode 6, which introduces the concept of **Live Views**, which are views which appear in IB as they will in the running app. The exciting news is that they are super easy to use - and you can find out how today.

The sample project which accompanies this project contains the code for the **RWKnobControl** I created for a tutorial on building custom controls over on [RayWenderlich.com⁵⁰](http://RayWenderlich.com). The code has been

⁴⁹<http://iwantmyreal.name/blog/2013/09/18/the-art-of-custom-ui-controls/>

⁵⁰<http://www.raywenderlich.com/56885/custom-control-for-ios-tutorial-a-reusable-knob>

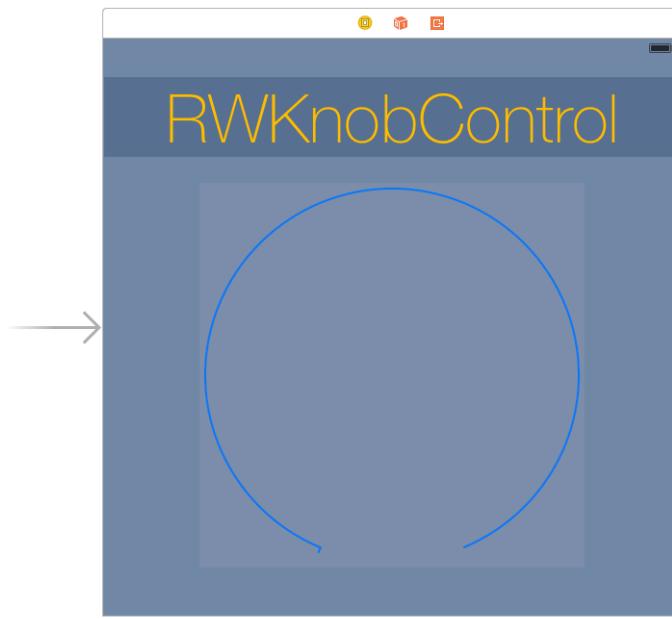
translated into Swift, but other than that remains the same. You can get hold of this project in the iOS 8 day-by-day repo on github at [github.com/ShinobiControls/iOS8-day-by-day⁵¹](https://github.com/ShinobiControls/iOS8-day-by-day).

@IBDesignable

The simplicity associated with this new functionality is truly beautiful. Given that you have a `UIView` subclass, then enabling the live-rendering in IB is as simple as annotating the class with the `@IBDesignable` annotation:

```
1 @IBDesignable
2 class KnobControl : UIControl {
3     ...
4 }
```

Now, the view in the storyboard will automatically update to render the live view:



Default Storyboard

And that's it! The objective-C equivalent is `IB_DESIGNABLE`.

You can override the `prepareForInterfaceBuilder()` method if you need to provide any custom setup for the IB view. This will get called only when rendering inside interface builder.

⁵¹<https://github.com/ShinobiControls/iOS8-day-by-day>

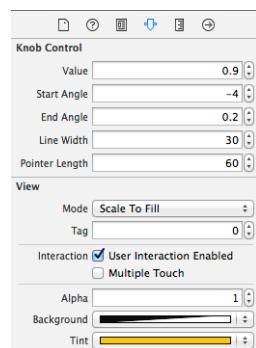
@IBInspectable

It's all very well and good being able to see the view in interface builder, but in the knob control example you've just seen it doesn't look too good. The knob control has a whole set of properties that are used to configure its appearance. If you annotate these properties with `@IBInspectable` (`IB_INSPECTABLE` in objective-C) then IB will provide you with config fields in the attributes inspector to configure them:

```

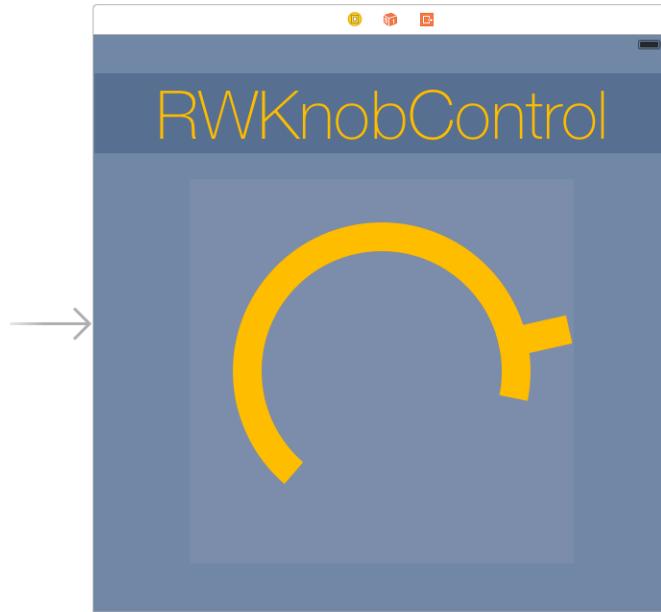
1  @IBInspectable
2  var value:CGFloat {
3      get { return self._primitiveValue }
4      set { self.setValue(newValue, animated: false) }
5  }
6  @IBInspectable
7  var startAngle:CGFloat {
8      get { return self.knobRenderer.startAngle }
9      set { self.knobRenderer.startAngle = newValue }
10 }
11 ...

```



Attributes Inspector

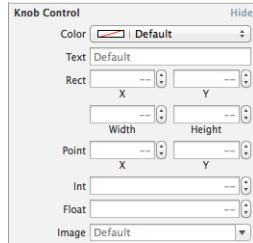
As you change these values then the view in the storyboard itself will update to reflect the new settings:



Updated Visualization

You can edit any types that you are used to using within IB:

```
1 @IBInspectable
2 var color: UIColor = UIColor.whiteColor()
3
4 @IBInspectable
5 var text: String = ""
6
7 @IBInspectable
8 var rect: CGRect = CGRect.zeroRect
9
10 @IBInspectable
11 var point: CGPoint = CGPoint.zeroPoint
12
13 @IBInspectable
14 var int: Int = 0
15
16 @IBInspectable
17 var float: CGFloat = 0.0
18
19 @IBInspectable
20 var image: UIImage = UIImage()
```

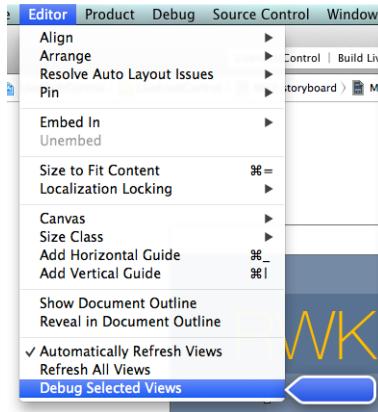


Inspectable Types

Debugging Views

So you've managed to get a view which appears inside IB, but maybe it's not behaving in quite the way you want it. In days gone by you would have thrown in some breakpoints to your view code, and then run it up in the simulator to debug it. Obviously you can continue to use this methodology, but the new live views functionality also offers the ability to debug views right in Interface Builder.

Set up your break points in the usual way, and then select the view in the storyboard. Use the **Editor > Debug Selected Views** menu option to kick off the debug process.



Debug selected view

The debug process works in exactly the same way as it does on a device or in the simulator - so once you've started debugging then changing a value in the attributes inspector will run exactly the same code you'd expect - and hence hit any appropriately placed breakpoints.

Conclusion

This new functionality is really great for anybody who uses custom views and quite likes using IB for laying out their view controllers. It's so simple to integrate into your views that there isn't really any excuse not to do it for views which are fairly simple. Obviously there are many custom views which are far more complex (and require table views and the suchlike), however the hooks in to the IB render process might still allow big gains over the iterative render process that you're used to.

The code for this project is again available in the github repo at github.com/ShinobiControls/iOS8-day-by-day⁵².

⁵²<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 18 :: UISplitViewController



If you've used Xcode's master-detail template in the past then you'll be aware of the rather confusing code that it creates. This is because it uses two completely different view controller hierarchies depending on the device it's running on. The split view controller works on the iPad, but not the iPhone, so a navigation controller is used instead. This means that the code is littered with idiom checks and repetition.

The fundamental principle is sound - you can't use the same view controller hierarchy on all devices. However, there is no reason that this shouldn't be abstracted away from the developer into the framework itself. This is exactly what iOS 8 does - with the introduction of adaptive view controller hierarchies.

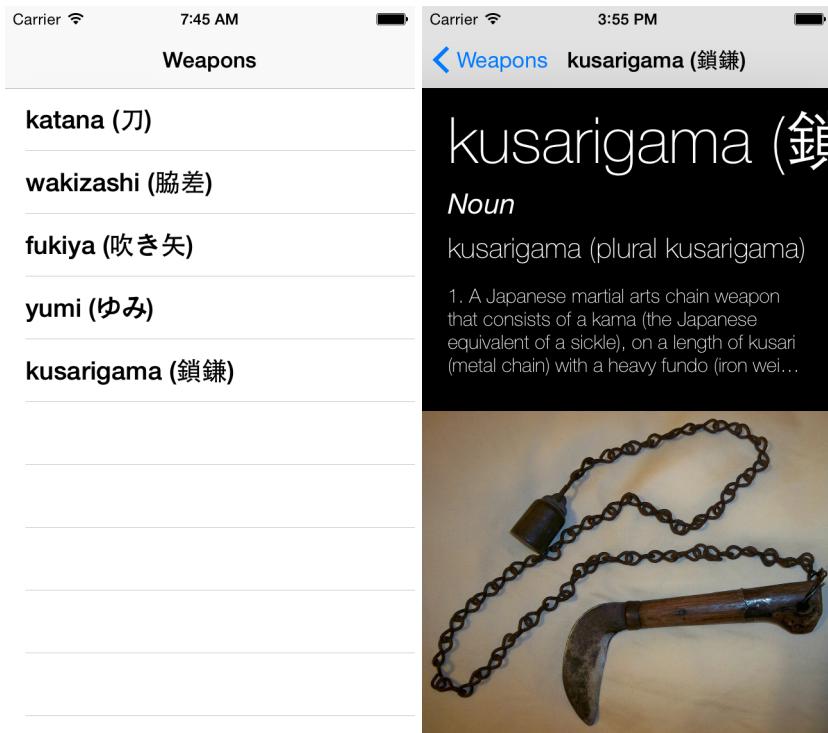
In the master-detail scenario this means that a `UISplitViewController` can now be used on all devices. It retains the same appearance on an iPad as in iOS 7, but on an iPhone it appears as a navigation controller.

In today's article you're going to learn more about what this means for your code, and how you can override the default behavior. The accompanying project is based on the master-detail project template in Xcode 6, so uses the new split view controller. You can download the code from the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁵³](https://github.com/ShinobiControls/iOS8-day-by-day).

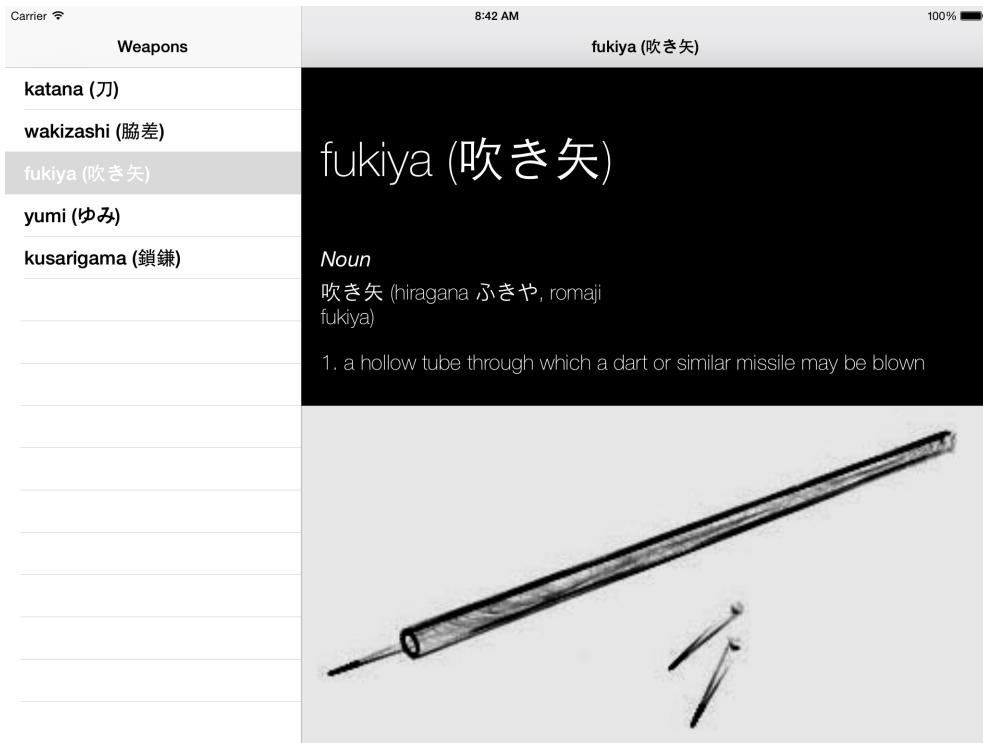
Adaptive View Controller Hierarchy

In the newly adaptive `UISplitViewController`, the view controller hierarchy is determined by its horizontal size class. When `Compact` then the split view will appear **Collapsed** - in a navigation controller. The master view controller will be displayed first, and selecting one of the rows in the table will push the detail view controller onto the navigation stack.

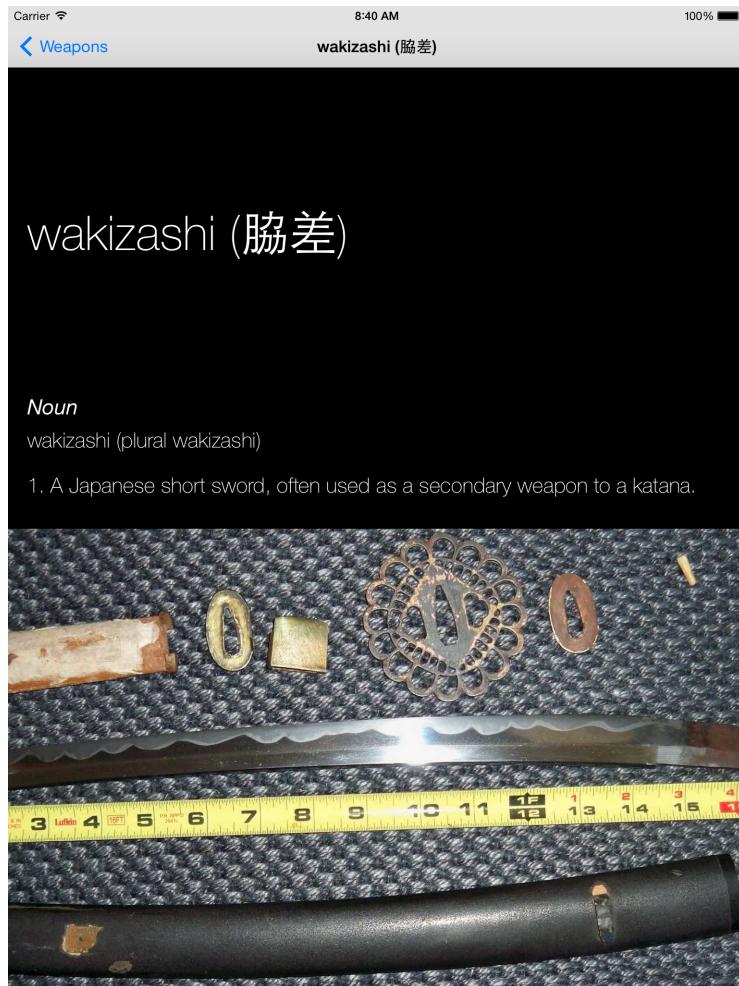
⁵³<https://github.com/ShinobiControls/iOS8-day-by-day>

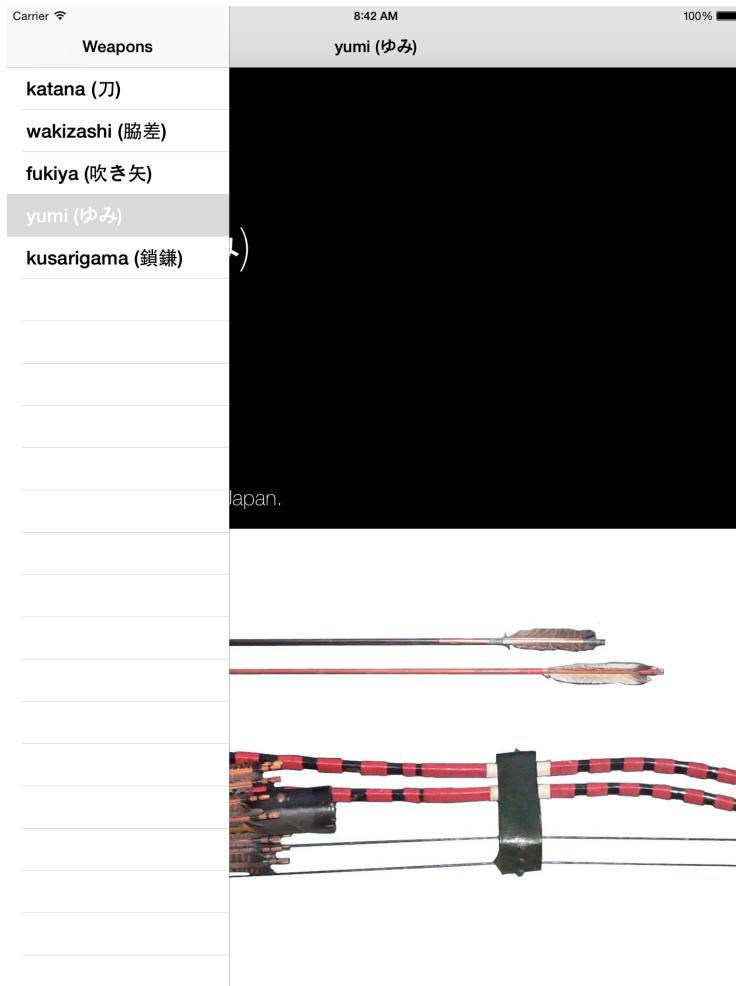


This is in contrast to the Regular horizontal class, which expands to display the master and detail view controllers simultaneously. This can be in a selection of configurations - known as display modes. This defaults in the landscape having a permanently visible primary column:



And in portrait an overlaid view controller which can be toggled using the left button item in the navigation bar:





Importantly, the new split view controller requires no differences in code to represent the two different view controller hierarchies. Providing the detail view controller with the appropriate model object is now performed in the `prepareForSegue(segue:, sender:)` method:

```
1 override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
2     if segue.identifier == "showDetail" {
3         let indexPath = self.tableView.indexPathForSelectedRow()
4         if let weapon = weaponProvider?.weapons[indexPath.row] {
5             let destNavVC = segue.destinationViewController as UINavigationController
6             let controller = destNavVC.topViewController as DetailViewController
7             controller.weapon = weapon
8             controller.navigationItem.leftBarButtonItem =
9                     self.splitViewController.displayModeButtonItem()
10            controller.navigationItem.leftItemsSupplementBackButton = true
11        }
12    }
```

```
13 }
```

This raises an interesting problem - if you want to present a new view controller in the secondary pane in code then you need to know whether the split view is currently appearing as a navigation controller, or an expanded split view. To address this, two new methods have been added to `UIViewController` in the form of `showViewController()` and `showDetailViewController()`. These methods have different behavior depending on the hierarchy the view controller finds itself in. The ancestors are interrogated in turn until one is found which provides an implementation. If either reaches the root view controller then the provided view controller will be displayed using `presentViewController()`.

The following table details the behaviors exhibited in different scenarios:

	<code>show</code>	<code>showDetail</code>
Navigation Controller	push	-
Expanded Split View	-	redirect to a show on secondary
Collapsed Split View	-	redirect to a show on primary
Vanilla view controller	-	-
Root	present	present

This might look complicated, but the end result is that using the show and show detail methods will not only guarantee that the supplied view controller will be displayed, but that it will make sense for the current context.

Overriding Default Behavior

Since the appearance is determined by size class you can easily show an expanded split view on an iPhone by overriding the trait collection. This is done by creating a container view controller and providing a trait collection override.

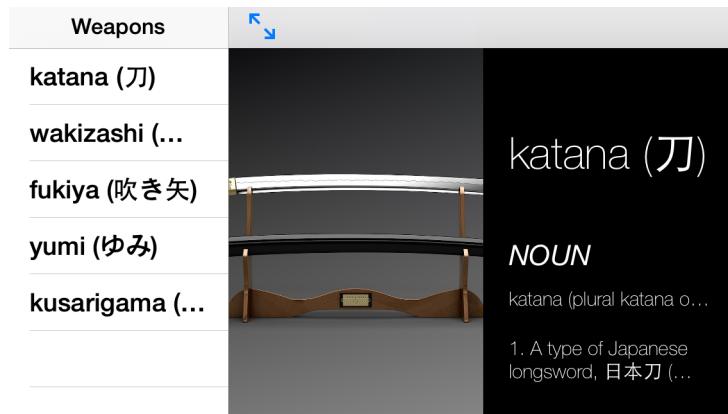
The sample application overrides the horizontal size class based on the width of the current view:

```
1 class TraitOverrideViewController: UIViewController,
2                                     UISplitViewControllerDelegate {
3
4     override func viewDidLoad() {
5         super.viewDidLoad()
6         performTraitCollectionOverrideForSize(view.bounds.size)
7     }
8
9     override func viewWillTransitionToSize(size: CGSize,
10                                         withTransitionCoordinator coordinator:
11                                         UIViewControllerTransitionCoordinator) {
```

```

12     super.viewWillTransitionToSize(size, withTransitionCoordinator: coordinator)
13     performTraitCollectionOverrideForSize(size)
14 }
15
16 private func performTraitCollectionOverrideForSize(size: CGSize) {
17     var overrideTraitCollection: UITraitCollection? = nil
18     if size.width > 320 {
19         overrideTraitCollection = UITraitCollection(horizontalSizeClass: .Regular)
20     }
21     for vc in self.childViewControllers as [UIViewController] {
22         setOverrideTraitCollection(overrideTraitCollection,
23                                     forChildViewController: vc)
24     }
25 }
26 }
```

`performTraitCollectionOverrideForSize()` takes a `CGSize`, and if the width is larger than 320pts forces all the child view controllers to have a horizontally regular size class. This method needs to be called in two places - once when the view first loads, and then whenever the view controller changes size, the latter using the new `viewWillTransitionToSize(_:, withTransitionCoordinator:)` method.



UISplitViewController on an iPhone

Advanced Features

The split view delegate has new methods as well - which you can use to control the behavior associated with expanding and collapsing split views. This occurs when the split view transitions from being expanded to being collapsed - e.g. on the newly updated iPhone version of **NinjaWeapons**.

You can specify whether the detail view controller should be popped onto the master's navigation stack when the split view is collapsing, and conversely whether the top view controller should be

used as the detail view when expanding . The default behavior is great for most use cases, but if you have a more complex hierarchical data structure, then these methods are invaluable.

The delegate methods even allow you to specify completely new view controllers in these cases - so you can build a completely custom split view.

`UISplitViewController` also provides a toolbar button which allows cycling between the different display modes when the split view is expanded. This button adapts to the current circumstance - so it will show/hide an overlaid master VC, or toggle between always visible and hidden otherwise. This button is accessible via the `displayModeButtonItem()` method.

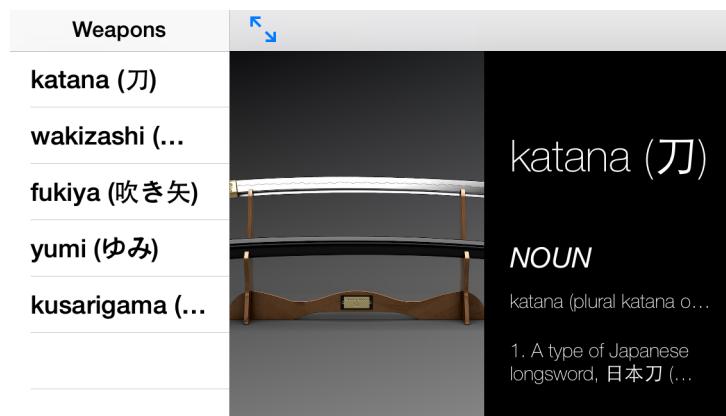
The width of the split is also configurable via the following properties on `UISplitViewController`:

- `preferredPrimaryColumnWidthFraction`
- `minimumPrimaryColumnWidth`
- `maximumPrimaryColumnWidth`

The following code is used in `NinjaWeapons` to configure both the split width and set up the display mode button:

```

1 private func configureSplitVC() {
2     // Set up split view delegate
3     let splitVC = self.childViewControllers[0] as UISplitViewController
4     splitVC.delegate = self
5     splitVC.preferredPrimaryColumnWidthFraction = 0.3
6     let navVC = splitVC.childViewControllers.last as UINavigationController
7     navVC.topViewController.navigationItem.leftBarButtonItem =
8             splitVC.displayModeButtonItem()
9 }
```



UISplitViewController on an iPhone

Conclusion

The new and improved `UISplitViewController` is great - removing both idiom checks and repeated code. Its out-the-box functionality is suitable for many use cases, but is easily extendable to cover more esoteric requirements.

If you want to dive head-first into the new `UISplitViewController` in loads more detail then I've written an entire chapter on the topic in the Ray Wenderlich iOS 8 By Tutorials book. It comes highly recommended (by me, as one of the authors) and you can grab your copy from here.

The **NinjaWeapons** project is available on github, as usual. It's at github.com/ShinobiControls/iOS8-day-by-day⁵⁴.

⁵⁴<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 19 :: CoreImage Kernels

8 19

CoreImage regularly pops up in the Day-by-Day series, but is underused in the wild, often because it is misunderstood. It provides highly optimized image processing algorithms, in the form of filters and detectors, which can easily be chained together in a light-weight processing pipeline.

One of the restrictions with CoreImage on iOS devices has always been that you are restricted to using the provided filter kernels. This has meant that it is incredibly difficult to use the GPU to perform any bespoke image processing. However, iOS 8 changes this - with the introduction of custom kernels in CoreImage.

This post gives a brief introduction into the new custom kernel functionality - with a couple of examples of creating your own filters. Custom kernels can be quite complex - they are written in the GLSL language, and represent image processing algorithms. This article will not attempt to describe either of these in any detail, but instead discuss the mechanics of creating the kernels and filters themselves.

The accompanying project is called **FilterBuilder** and demonstrates building two different types of CoreImage kernels - one using a color kernel, and one a general kernel. The source code for this project is available in the repo on github at [github.com/ShinobiControls/iOS8-day-by-day⁵⁵](http://github.com/ShinobiControls/iOS8-day-by-day).

Filters and Kernels

In image processing a filter can be thought of as an algorithm which takes a number of inputs, including an image, and produces an output image. A filter is a wrapper around a kernel - which is the algorithm which is applied in turn to every single pixel in the image.

The kernel is the core of the filter, and since it will be run for every single pixel in the image, it needs to be very efficient. The GPU is very good at performing the kinds of calculations you require for an image filter, so the code for the kernel is written in the language which will run directly on the graphics hardware: GLSL. If you've ever done any work in OpenGL ES then you'll recognize this as the language you've used. In order to preserve platform independence, then the code is provided as a string and compiled on the device, at run time. This can make writing the code itself quite hard work, since you don't get the features you've come to expect from an IDE.

A filter in CoreImage is represented by the `CIFilter` class. These can be chained together, and provided an input `CIImage`. The output will in turn be another `CIImage`. One of the great things about this architecture is that it is incredibly light-weight, and is lazy evaluated. Creating a filter

⁵⁵<http://github.com/ShinobiControls/iOS8-day-by-day>

and adding it to the processing chain will not involve any calculation until the point at which the output image is required.

iOS comes with 115 built-in image filters available for you to use in your image processing pipelines, but iOS 8 allows you to create your own filters from custom kernels. These are represented by the `CIColorKernel` class, which contains a string of the kernel method itself, written in GLSL.

In the next section you'll learn about the different types of custom kernel, and see how to create one.

Custom Kernel Types

Image processing is an area of programming where algorithm optimization often pays off. This is primarily due to the problem class - you want an incredibly tight inner loop, since the same kernel will be run for every single pixel in an image. Therefore CoreImage provides several different subclasses of `CIColorKernel` which are highly optimized for a particular class of problem:

- `CIColorKernel` is optimized for the scenario where each output pixel depends only on the pixel in the same position on the input image. This means that you can't read pixels from elsewhere in the input. This is ideal for color transformations, as the name suggests.
- `CIWarpKernel` doesn't change the value of input pixel, but allows you to choose the location of the output pixel in the input image. This is equivalent to defining a vector field across the input image, and hence has the effect of warping the input image to create the output.
- `CIKernel`. The superclass allows you to create a general kernel, for which you calculate the output pixel value from anywhere within the input image. The kernel routine itself has access to the entire image.

There are some common concepts across all the different kernel types. They are all initialized with the same constructor:

```
1 let kernelString = ""  
2 let kernel = CIColorKernel(string: kernelString)
```

The `kernelString` is the string of GLSL which represents the kernel algorithm itself. Each of the `CIKernel` subclasses has a different way of actually invoking the kernel, but they all include an `arguments` parameter, which is of type `[AnyObject]`. This represents the objects which should be passed into the kernel function at runtime, in the order specified by the kernel function signature.

All of the kernels have an `extent` parameter, which is a `CGRect`. This, also known as the domain of the function, specifies the rectangle over which the kernel should operate. This is required because `CIImage` can have infinite extent, and there are times when the domain is non-obvious (e.g. compositing images). Often, this will be the same as the extent of the input image.

Color Kernels

A color kernel takes each pixel in an image, and uses it to calculate the value for the output pixel. Therefore, the (minimum) input for the kernel function is a single pixel value, and it needs to return a single pixel. You can provide additional arguments in an array as described above.

In the sample project the chromakey example uses a `CIColorKernel`. The principle behind chromkeying (commonly known as ‘green-screen’) is that areas of a picture which appear green are replaced with a different image. In the `ChromaKeyKernel` the pixels are set to transparent. The user can provide the color and a threshold. The following function creates the kernel:

```

1 private func createKernel() -> CIColorKernel {
2     let kernelString =
3         "kernel vec4 chromaKey( __sample s, __color c, float threshold ) { \n" +
4             "    vec4 diff = s.rgb - c;\n" +
5             "    float distance = length( diff );\n" +
6             "    float alpha = compare( distance - threshold, 0.0, 1.0 );\n" +
7             "    return vec4( s.rgb, alpha );\n" +
8         "}"
9     return CIColorKernel(string: kernelString)
10 }
```

Some key points:

- The return type is `vec4`, which is a vector of floats of length 4 - representing RGBA.
- `diff` is the element-wise difference between the supplied color and the current pixel value
- `distance` is the Euclidean length of the difference vector
- The `compare` function is used to determine whether this pixel is close enough to the active color or not. It creates an alpha value of `0.0` or `1.0`
- The returned value has the same RGB values as the input pixel, but takes the calculated `alpha` value.

To use this kernel, it must be wrapped in a custom `CIFilter` subclass. `CIFilter` has an `outputImage() -> CIImage?` method which must be overridden, and will be called when the processing chain requires the filter to be executed.

The following is the implementation of `outputImage()` for `ChromaKeyFilter`:

```
1 func outputImage() -> CIImage? {
2     if let inputImage = inputImage {
3         let dod = inputImage.extent()
4         if let kernel = kernel {
5             var args = [inputImage as AnyObject, activeColor as AnyObject,
6                         threshold as AnyObject]
6             return kernel.applyWithExtent(dod, arguments: args)
7         }
8     }
9 }
10 return nil
11 }
```

This uses the method `applyWithExtent(_:, arguments)` on `CIColorKernel` to return the result. The arguments array is created to match the input requirements for the aforementioned kernel function. As previously discussed, the extent here is just the extent of the input image.

This filter can then just be used in the normal way:

```
1 let inputImage = UIImage(named: "chroma_key")
2 filter.inputImage = CIImage(image: inputImage)
3 filter.activeColor = CIColor(red: 0, green: 1, blue: 0)
4 filter.threshold = CGFloat(thresholdSlider.value)
5 let outputImage = filter.outputImage
6 outputImageView.image = UIImage(CIImage: outputImage)
```

This has the following results:



Warp Kernels

Warp kernels don't allow you to change anything about the pixel value itself, but instead allow you to define which point in the source image it should copy. This means that a warp kernel will return a `vec2`, representing the coordinates of the input image that should be sampled for the current output pixel.

The following represents a no-op warp kernel:

```
1 kernel vec2 passThrough () {
2     return destCoord();
3 }
```

The major difference between using warp kernels and color kernels is the addition to the ROI callback in the apply method:

```
1 func applyWithExtent(extent: CGRect, roiCallback callback: CIKernelROICallback!,
2                     inputImage image: CIImage!, arguments args: [AnyObject]!) -> CIImage!
```

`CIKernelROICallback` is a closure which will provide a `CGRect` in the output image, and require the equivalent rect in the input image. ROI stands for Region of Interest, and it is used by the filter to determine which part of the input image needs to be loaded into the graphics hardware to allow processing of a particular region of the output image. This is in part due to the fact that the architecture of CoreImage has been updated with iOS 8 (and OSX 10.10) to support processing images much larger than the graphics memory can cope with. Slicing the image into sections allows efficient use of both the graphics memory and the processing power of the GPU.

General Kernels

The final kernel type is a superclass of both `CIColorKernel` and `CIWarpKernel`, and is known as a general kernel. Whereas both of the subclasses had specific use cases, the general kernel has no such restrictions. For example, a huge class of image filters are based on convolution, which neither the color kernel nor the warp kernel are able to implement. A convolution kernel (such as Gaussian blur) requires the values of not only the current pixel, but those surrounding to calculate the value of the output pixel.

As such, a general kernel will output a `vec4`, representing the current pixel value. You can provide multiple images as input, and they will each be represented as a `sampler`, from which you can request samples, at different locations.

In the sample project, there is a general kernel which implements the Sobel filter. This filter is used for finding edges, and their direction, and is the result of a couple of convolutions with 3×3 matrices. The following shows the kernel implementation:

```

1 private func createKernel() -> CIKernel {
2     let kernelString =
3         "kernel vec4 sobel (sampler image) {\n" +
4             "    mat3 sobel_x = mat3( -1, -2, -1, 0, 0, 0, 1, 2, 1 );\n" +
5             "    mat3 sobel_y = mat3( 1, 0, -1, 2, 0, -2, 1, 0, -1 );\n" +
6             "    float s_x = 0.0;\n" +
7             "    float s_y = 0.0;\n" +
8             "    vec2 dc = destCoord();\n" +
9             "    for (int i=-1; i <= 1; i++) {\n" +
10                "        for (int j=-1; j <= 1; j++) {\n" +
11                    "            vec4 currentSample = sample(image,\n" +
12                        "                            samplerTransform(image, dc + vec2(i,j)));\n" +
13                    "            s_x += sobel_x[j+1][i+1] * currentSample.g;\n" +
14                    "            s_y += sobel_y[j+1][i+1] * currentSample.g;\n" +
15                }\n" +
16            }\n" +
17            "            return vec4(s_x, s_y, 0.0, 1.0);\n" +
18        }"
19     return CIKernel(string: kernelString)
20 }

```

This kernel function requires just the image, and then it will calculate the convolution of the current point with 2 matrices. One of these will find the vertical edges, and one the horizontal. The return pixel has the vertical magnitude in the green channel, and the horizontal in the red.

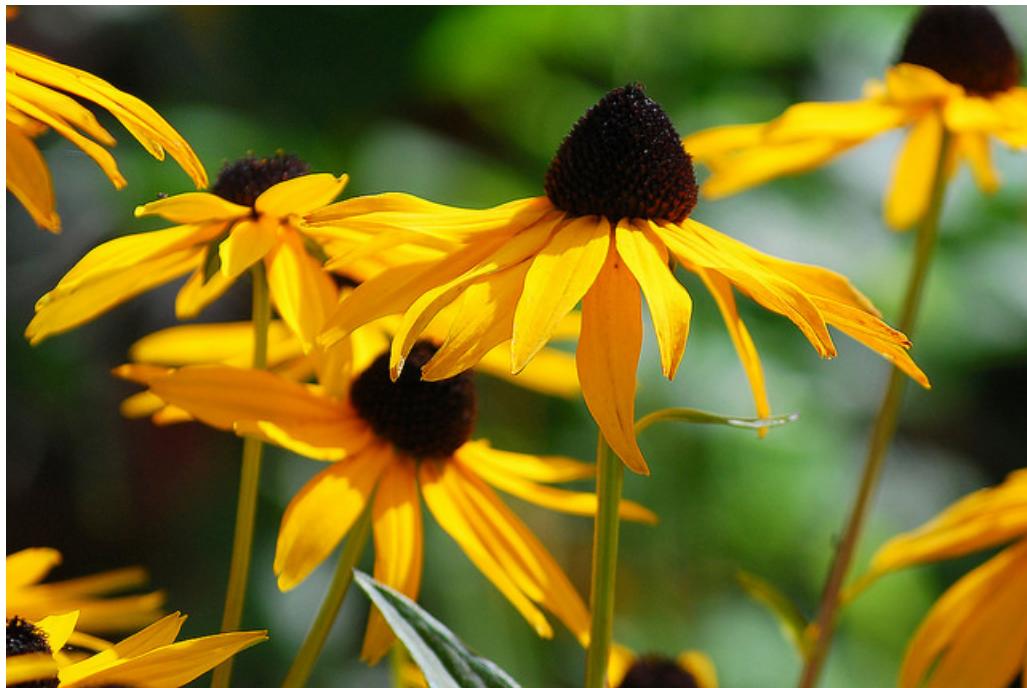
General kernels require an ROI callback in the same way that warp kernels do. In this instance, a given output rect will require a rectangle in the input image which is centered at the same point, but is expanded by one pixel in every direction. The following method on SobelFilter demonstrates how this works:

```

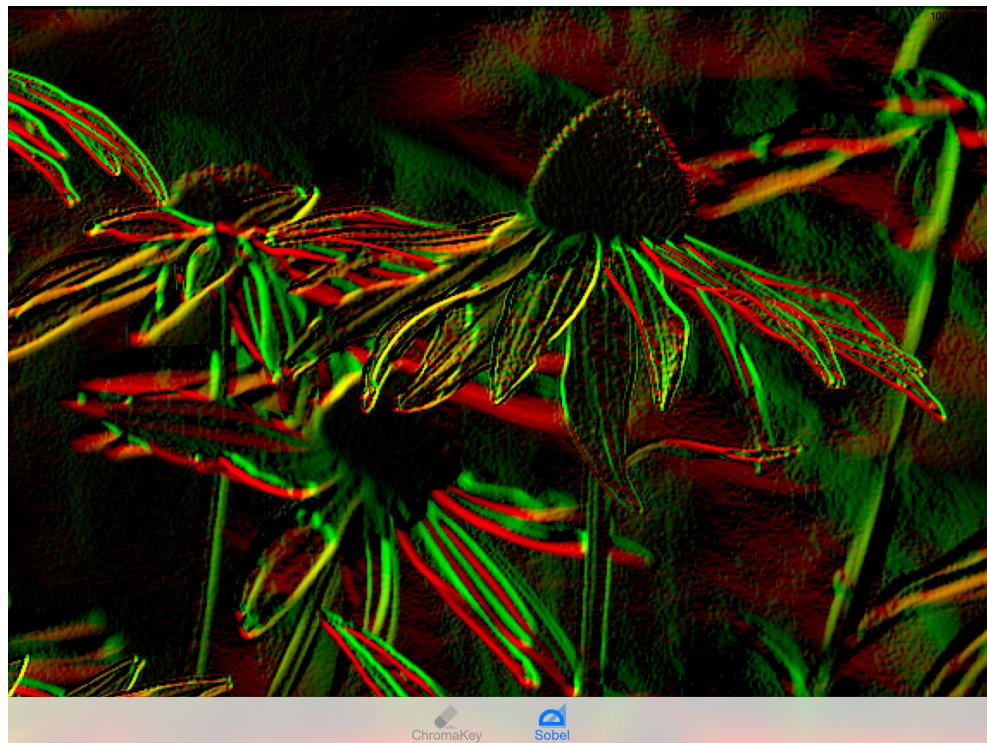
1 func outputImage() -> CIImage? {
2     if let inputImage = inputImage {
3         let dod = inputImage.extent()
4         if let kernel = kernel {
5             let args = [inputImage as AnyObject]
6             let dod = inputImage.extent().rectByInsetting(dx: -1, dy: -1)
7             return kernel.applyWithExtent(dod, roiCallback: {
8                 (index, rect) in
9                 return rect.rectByInsetting(dx: -1, dy: -1)
10            }, arguments: args)
11        }
12    }
13    return nil
14 }

```

Wiring this up in exactly the same way you did with the color kernel will result in the following input and output:



Flowers



Sobel Output

Conclusion

CoreImage is great - and often overlooked. This new functionality offers an immense amount of power - and is a really easy way to utilize the graphics hardware on mobile devices. However, before attempting to craft your own kernels, make sure that you've taken a look through the list of 115 off-the-shelf options. Loads of the common image processing requirements are either already in there or can be created by forming a pipeline of them.

If you do end up building your own filter kernels then pay close attention to optimizing the kernel itself. This is the inner loop of something which will run millions of times to process a camera image. It needs to be as fast as possible. It's worth learning about GLSL, specifically the CoreImage dialect.

The source code for **FilterBuilder** is available on github at

⁵⁶<http://github.com/ShinobiControls/iOS8-day-by-day>

⁵⁷<https://twitter.com/iwantmyrealname>

Day 20 :: Photos Framework



Providing access to the images in the system photo gallery is a fairly standard requirement for many apps - from social, through to photo editing. iOS has provided this ability through many releases, but it has never been particularly easy. In iOS 8, a new **PhotoKit** framework has been introduced, which provides complete access to the user's photo gallery - from the models which make up the library data structure, through to the automatic retrieval of images from within iCloud.

This post is going to take a look at some of the basics of using the new framework - from how the assets datastore is structured, accessed and updated through to how you can create a caching image manager.

The accompanying app is called **StarGallery** and is a simple implementation of a gallery backed by the photo library. The app is able to update the 'favorite' status of the photos, and demonstrates how to use the in-built image cache. As ever, the source code is available on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁵⁸](https://github.com/ShinobiControls/iOS8-day-by-day).



Star Gallery

⁵⁸<https://github.com/ShinobiControls/iOS8-day-by-day>

Photo Library Outline

The library is comprised of model objects - the fundamental one being `PHAsset`, which represents a single media asset. This could be a photo or a video, and has properties including `pixelWidth`, `pixelHeight`, `creationDate`, `modificationDate` and `favorite`. All PhotoKit model objects are immutable, so all of these properties are read-only.

These `PHAsset` objects are collected together into `PHAssetCollection` objects, which are ordered collections of assets. These represent albums, smart albums and moments and have properties such as `type`, `title`, `startDate` and `endDate`.

Folders and years of moments are formed from `PHCollectionList`, which is again an ordered collection.

The combination of these three classes allows the entire structure of the photo library to be modeled. However, none of these give you access to anything other than asset metadata. `PHImageManager` is used to request the image associated with a given `PHAsset`. This handles any network requests, rescaling and can even do caching. It can provide a placeholder image whilst the required quality is requested from the network.

Querying For Models

Each of the model classes has class methods associated with it to query the photo library - for example `PHAsset` has `fetchAssetsWithMediaType()`. All the model fetch methods are synchronous, and return an instance of `PHFetchResult`. The following will create a fetch result containing all the images:

```
1 let images = PHAsset.fetchAssetsWithMediaType(.Image, options: nil)
```

`PHFetchResult` has an API which looks very like `NSArray`, so you can iterate through the results, access results by index, and count the total number of items in the fetch result. Importantly, the items are lazy-loaded from the store in batches - i.e. the results aren't all loaded into memory at call-time, but are instead loaded when required. This reduces the memory footprint on something that could be a result set.

In the accompanying project, `StarGallery`, there is a collection view which displays all the images in the photo library. The asset models are loaded using the code snippet above in `viewDidLoad()`. As mentioned before, these objects are just the models which represent the assets in the photo library - they don't include the image itself. In the next section you'll learn how to request the image.

Requesting Assets

If you're going to use the photo library, then it's likely that you'll want to get hold of the images themselves - not just the asset models. PhotoKit makes this really easy, via the `PHImageManager` class.

Once you've created an instance, then you can request an image with the `requestImageForAsset()` method, as follows:

```
1 self.imageManager?.requestImageForAsset(imageAsset!,  
2                                         targetSize: CGSize(width: 320, height: 320),  
3                                         contentMode: .AspectFill, options: nil) {  
4     image, info in  
5     self.photoImageView.image = image  
6 }
```

In addition to providing the `PFAsset` which represents the image, you also need to provide a size and a content mode. This combination means that the image manager will rescale the image appropriately and return it to you. This method is asynchronous - with the image returned to the closure above. Since this is usually going to be used to update the UI, the framework will invoke this closure on the main queue. The closure might well be called more than once - if the required image quality isn't immediately available then a placeholder may well be provided until the request can be fulfilled (e.g. returned from the network).

In addition the base `PHImageManager` class, PhotoKit also includes a `PHCachingImageManager`, which caches images, allowing preheating which results in better scrolling performance. In order to use this you need to be able to tell the image manager which assets it should start caching (i.e. load up) and which ones it no longer needs to cache. Note that the cache works on specific image sizes - so you can cache the same asset multiple times (at different sizes).

The following method is part of an `ImageCacheController` class in `StarGallery`:

```
1 func updateVisibleCells(visibleCells: [NSIndexPath]) {  
2     let updatedCache = NSMutableIndexSet()  
3     for path in visibleCells {  
4         updatedCache.addIndex(path.item)  
5     }  
6     let minCache = max(0, updatedCache.firstIndex - cachePreheatSize)  
7     let maxCache = min(images.count - 1, updatedCache.lastIndex + cachePreheatSize)  
8     updatedCache.addIndexesInRange(NSMakeRange(minCache, maxCache - minCache + 1))  
9  
10    // Which indices can be chuck?  
11    self.cachedIndices.enumerateIndexesUsingBlock {  
12        index, _ in  
13        if !updatedCache.containsIndex(index) {  
14            let asset: AnyObject! = self.images[index]  
15            self.imageCache.stopCachingImagesForAssets([asset],  
16                                              targetSize: self.targetSize, contentMode: self.contentMode, options: nil)  
17            println("Stopping caching image \(index)")  
18        }  
19    }  
20 }
```

```
18     }
19 }
20
21 // And which are new?
22 updatedCache.enumerateIndexesUsingBlock {
23     index, _ in
24     if !self.cachedIndices.containsIndex(index) {
25         let asset: AnyObject! = self.images[index]
26         self.imageCache.startCachingImagesForAssets([asset],
27             targetSize: self.targetSize, contentMode: self.contentMode, options: nil)
28         println("Starting caching image \(index)")
29     }
30 }
31 cachedIndices = NSIndexSet(indexSet: updatedCache)
32 }
```

As the user scrolls, this method is called with an array of `NSIndexPath` objects for the visible cells. This method determines whether the cache needs updating, based on the visible cells, and the `cachePreheatSize`. It then uses the `stopCachingImagesForAssets()` and `startCachingImagesForAssets()` methods to update the caching image manager.

Note that obtaining the image from a caching image manager is exactly the same as the base `PHImageManager` - and it will obtain the image even if it doesn't exist in the cache.

Performing Model Updates

Having immutable model objects is great for thread-safety, but it does mean that updating the photo library is made a little more complex. You can't just update some settings on the model objects and then request the framework to persist them. Instead, you must create a change request, and then get the library to perform it.

`PHPhotoLibrary` has a `performChanges()` method, which takes a closure containing the required change requests. Each of the model types has an associated change request class - for example `PHAssetChangeRequest`, and they are instantiated using a model object. Once you have the change request object, you can mutate it to match the change you wish to perform. Since this is created within the `performChanges()` closure, the framework will then perform this change asynchronously.

In the accompanying app, there is a star button which, when pressed, will toggle the 'favorite' status of an asset. The following is the method which gets called:

```
1 @IBAction func handleStarButtonPressed(sender: AnyObject) {
2     PHPhotoLibrary.sharedPhotoLibrary().performChanges({
3         let changeRequest = PHAssetChangeRequest(forAsset: self.imageAsset)
4         changeRequest.favorite = !self.imageAsset!.favorite
5     }, completionHandler: nil)
6 }
```

Here, the change request is created using `PHAssetChangeRequest(forAsset:)` with the image asset associated with this cell, before toggling the value of the `favorite` property.

In this instance the `completionHandler` closure is `nil`, but you can get a callback with the status of the change request.

Registering for Update Notifications

Now that you've managed to update the asset in the photo library, you might be wondering about how to update your UI. You could use the completion block in the `performChanges()` method, but there is a more generic approach.

You can subscribe to change notifications on a photo library, meaning that you will be informed whenever it changes - irrespective of whether the change comes from your app, or elsewhere (e.g. the Photos app).

Registering for updates is as simple as:

```
1 PHPhotoLibrary.sharedPhotoLibrary().registerChangeObserver(self)
```

In order for this to work, your class must adopt the `PHPhotoLibraryChangeObserver` protocol, which includes the `photoLibraryDidChange(changeInstance: PHChange!)` method.

The `PHChange` class encapsulates the library update, and you can request either to get the change details for a specific object, or a `PHFetchRequest`. In `StarGallery`, the `PHFetchResultChangeDetails` are requested for the fetch request displayed in the collection view:

```
1 func photoLibraryDidChange(changeInstance: PHChange!) {
2     let changeDetails = changeInstance.changeDetailsForFetchResult(images)
3     self.images = changeDetails.fetchResultAfterChanges
4     dispatch_async(dispatch_get_main_queue()) {
5         // Loop through the visible cell indices
6         let indexPaths = self.collectionView?.indexPathsForVisibleItems()
7         for indexPath in indexPaths as [NSIndexPath]{
8             if changeDetails.changedIndexes.containsIndex(indexPath.item) {
9                 let cell = self.collectionView?
```

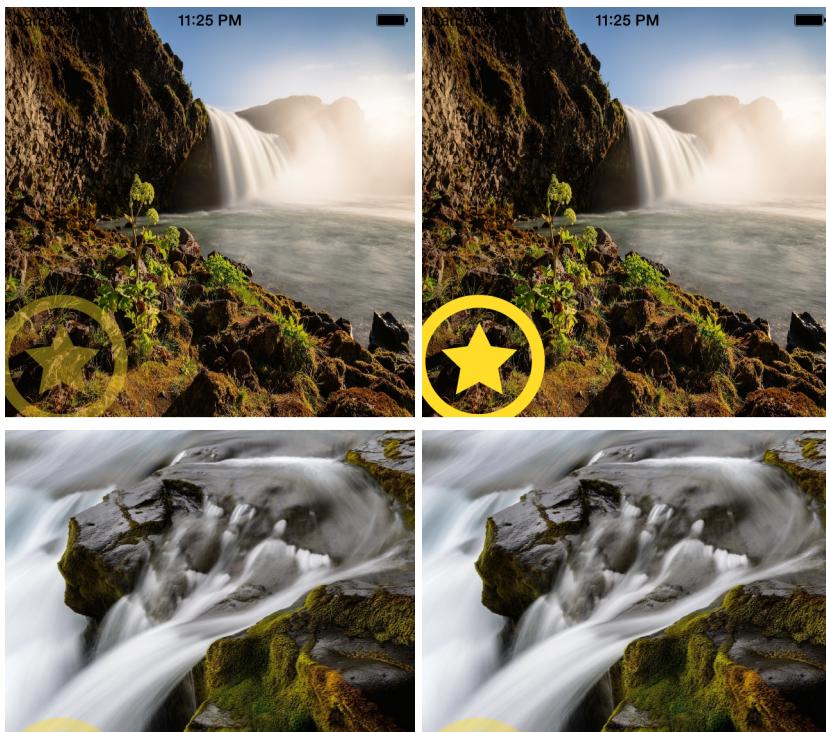
```
10         .cellForItemAtIndexPath(indexPath) as PhotosCollectionViewCell
11     cell.imageAsset = changeDetails
12         .fetchResultAfterChanges[indexPath.item] as? PHAsset
13     }
14 }
15 }
16 }
```

This implementation then determines which of the visible cells are affected by this change, and provides them with the updated model object, which will update the appearance appropriately.

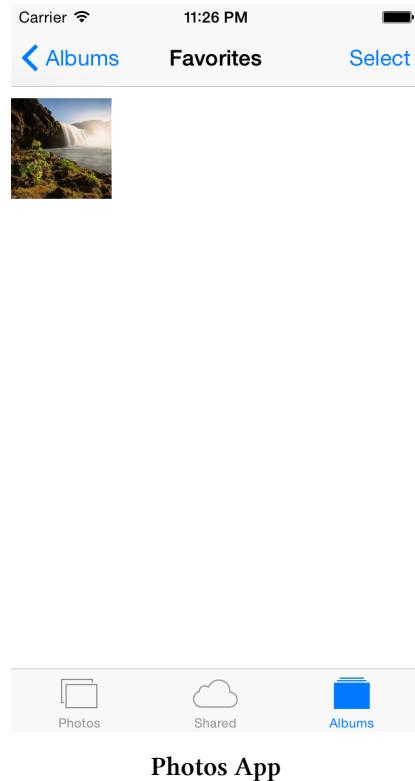


Note: In addition to the `fetchResultAfterChanges` and `changedIndexes`, the `PHFetchResultChanges` class also includes details of inserted, deleted and moved objects.

This now completes the ‘favoriting’ functionality of the `StarGallery` app - tapping the *star* icon will toggle the favorite status:



You can check that the photo library underlying data store is being updated by checking the favorites album in the Photos app:



Conclusion

This new framework makes working with images on the device both much easier for the developer and a more seamless experience for the user. In this article you've seen how to access the data models, the images and to perform simple model updates. There is even more depth to the framework, which allows you to create new assets, and to perform asset content edits as well.

As ever, the source code for today's sample photo app **StarGallery** is available on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁵⁹.

⁵⁹<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 21 :: Alerts and Popovers



Presenting new views and view controllers to users on iOS has evolved a lot since the original iPhoneOS. This has resulted in a pretty inconsistent state in iOS 7 - with some presentation methods being device restrictive, and others creating views in second UIWindow objects.

iOS 8 introduces the concept of a presentation controller, in the form of `UIPresentationController`. This is used to control the process of presenting new view controllers in every scenario. This means that the way you use alerts, popovers and action sheets has changed, to make a much more coherent and self-consistent system.

This post won't go into depth on presentation controllers (that may well appear in a later article), but instead focuses on what you need to know to update your apps to the new iOS 8 way of presenting popovers, alerts and action sheets.

The sample app which accompanies this project is available at github.com/ShinobiControls/iOS8-day-by-day⁶⁰.

Pop Overs

Popovers are now completely adaptive - which means that rather than having separate code paths for iPhone and iPad, they can be used on every device. A new class `UIPopoverPresentationController` has been introduced, and this controls the presentation of a view controller in a popover style. You don't create one of these directly, but instead one is created for you by UIKit when the `modalPresentationStyle` property on `UIViewController` is set to `.Popover`.

```
1 let popoverVC = storyboard?  
2     .instantiateViewControllerWithIdentifier("codePopover") as UIViewController  
3 popoverVC.modalPresentationStyle = .Popover
```

You can then get hold of the popover presentation controller from the `popoverPresentationController` property of `UIViewController`:

```
1 let popoverController = popoverVC.popoverPresentationController
```

This has settings that you'll recognize from `UIPopoverController` which you can use to configure the popover:

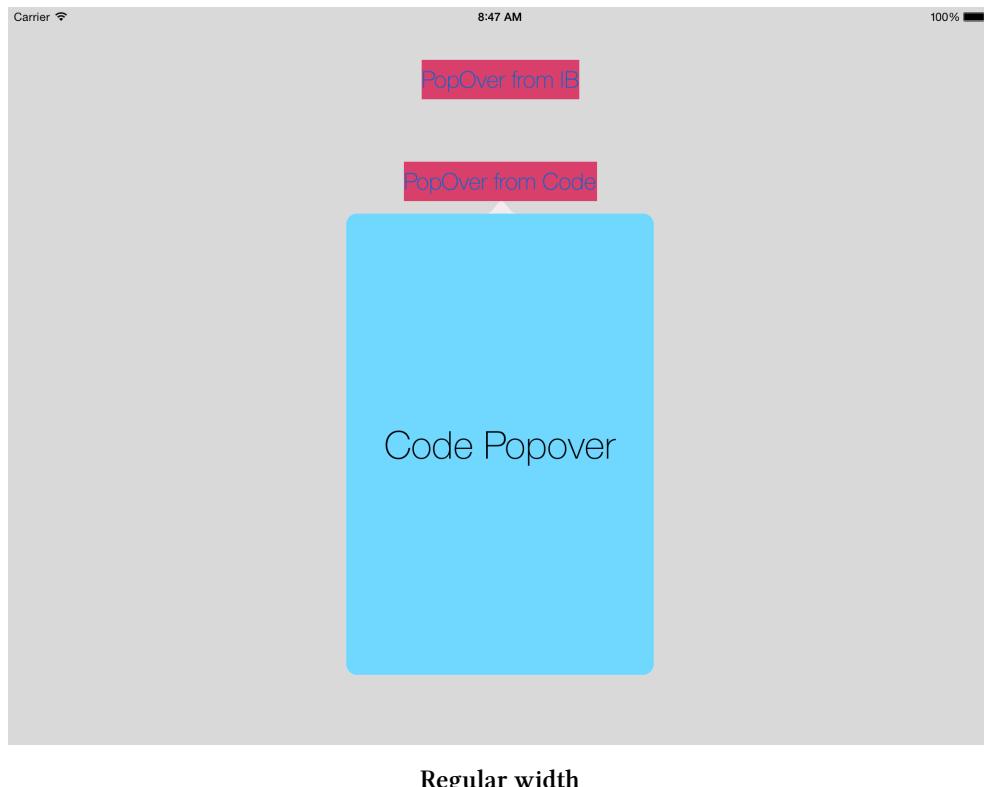
⁶⁰<https://github.com/ShinobiControls/iOS8-day-by-day>

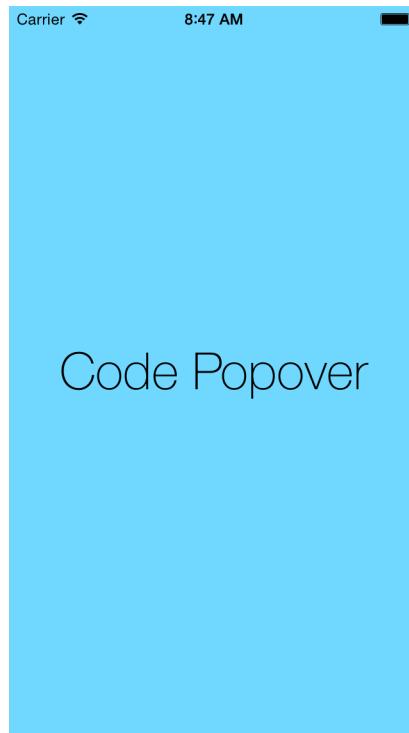
```
1 popoverController.sourceView = sender  
2 popoverController.sourceRect = sender.bounds  
3 popoverController.permittedArrowDirections = .Any
```

Then, presenting the popover is as simple as calling `presentViewController()`:

```
1 presentViewController(popoverVC, animated: true, completion: nil)
```

The popover presentation controller is inherently adaptive - a regular horizontal size class will show a traditional popover, but a compact will (by default) present using a full-screen modal presentation.





Compact Width

You can configure exactly how the adapted view controller appears (i.e. for compact width) using a `UIPopoverPresentationDelegate`. This has two methods - one for specifying the modal presentation style, and the other for returning a custom view controller.

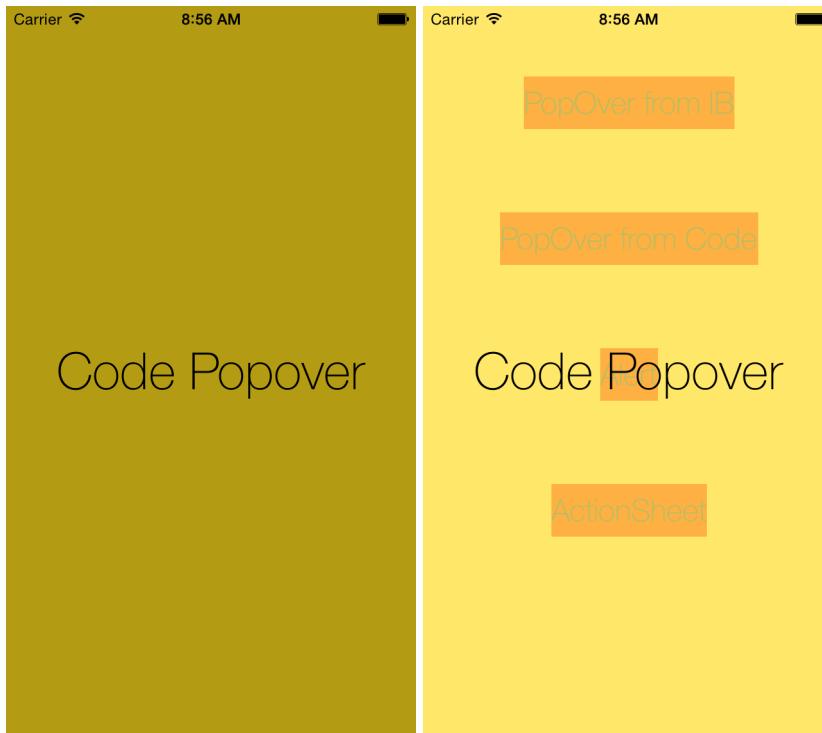
You set the delegate as you'd expect:

```
1 popoverController.delegate = self
```

The adaptive presentation style can be either `.FullScreen` or `.OverFullScreen`, the difference being that fullscreen will remove the presenting view controller's view, whereas over-fullscreen won't. You can set it with the following delegate method:

```
1 func adaptivePresentationStyleForPresentationController
2           (controller: UIPresentationController!) -> UIModalPresentationStyle {
3     return .FullScreen
4 }
```

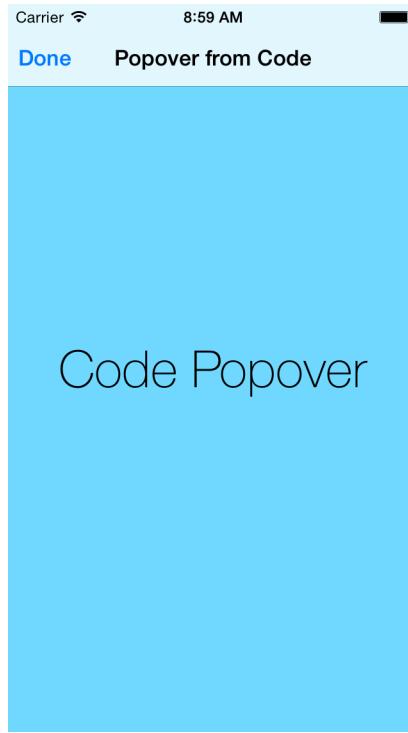
You can see the difference by setting the background color to semi-transparent, as shown below:



The other delegate method allows you to return a completely custom view controller for the adaptive display. For example, the following will put the popover view controller inside a navigation controller:

```
1 func presentationController(controller: UIPresentationController!,  
2     viewControllerForAdaptivePresentationStyle style: UIModalPresentationStyle)  
3     -> UIViewController! {  
4     return UINavigationController(rootViewController:  
5                                     controller.presentedViewController)  
6 }
```

And will result in something that looks like this:



Popover in Nav Controller

Note that none of this has changed the appearance for regular width - that will still show as a standard popover controller.

Alerts

In the past, alerts were actually a subclass of `UIView`, which were then displayed in a new `UIWindow`. This caused all kinds of issues with rotation and really doesn't fit the adaptive rotation-agnostic world of iOS 8. Since alerts are just another way to present content, they have been brought in-line with the rest of UIKit, in that it is just a `UIViewController` that is displayed using the `presentViewController()` method.

The class you need is `UIAlertController`, and it is instantiated with a title, message and preferred style:

```
1 let alert = UIAlertController(title: "Alert",
2                               message: "Using the alert controller",
3                               preferredStyle: .Alert)
```

There are two options for the `preferredStyle`, and this represents the difference between alerts and action sheets.

The API has been modernized to use closures instead of delegate callbacks, so you can add buttons as actions:

```
1 alert.addAction(UIAlertAction(title: "Cancel",
2                             style: .Cancel,
3                             handler: dismissHandler))
```

Here, `dismissHandler` is defined as follows:

```
1 let dismissHandler = {
2     (action: UIAlertAction!) in
3     self.dismissViewControllerAnimated(true, completion: nil)
4 }
```

There are three different styles for `UIAlertAction`: `.Cancel`, `.Default` and `.Destructive`.

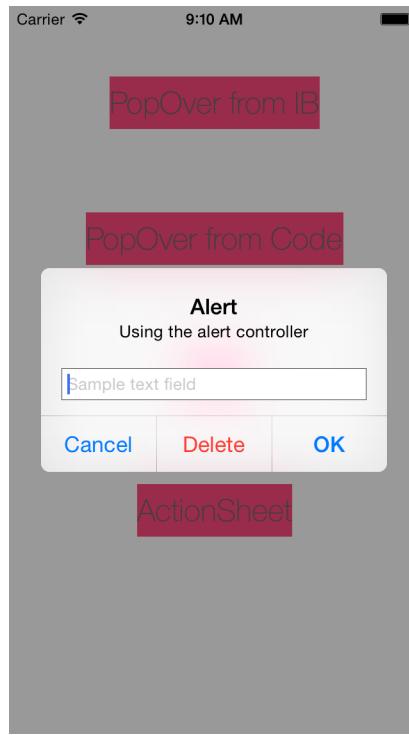
You can also add text fields to alerts with `addTextFieldWithConfigurationHandler`, which requires a closure which takes a `UITextField` and configures it appropriately:

```
1 alert.addTextFieldWithConfigurationHandler { textField in
2     textField.placeholder = "Sample text field"
3 }
```

Since `UIAlertController` is a subclass of `UIViewController`, you can then present it as you would any other:

```
1 presentViewController(alert, animated: true, completion: nil)
```

This will look like this:



Alert Controller

ActionSheets

Action sheets are actually a different style of a `UIAlertController`:

```
1 let actionSheet = UIAlertController(title: "Action Sheet",
2                                     message: "Using the alert controller",
3                                     preferredStyle: .ActionSheet)
```

You can't display text fields in an action sheet, but actions are created in exactly the same way as for an alert:

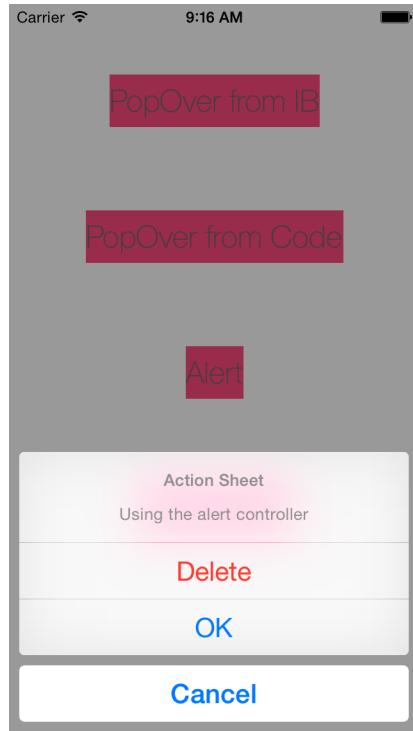
```
1 let dismissHandler = {
2     (action: UIAlertAction!) in
3     self.dismissViewControllerAnimated(true, completion: nil)
4 }
5 actionSheet.addAction(UIAlertAction(title: "Cancel", style: .Cancel,
6                                     handler: dismissHandler))
7 actionSheet.addAction(UIAlertAction(title: "Delete", style: .Destructive,
8                                     handler: dismissHandler))
9 actionSheet.addAction(UIAlertAction(title: "OK", style: .Default,
10                           handler: dismissHandler))
```

Action sheets are adaptive, and when in a regular horizontal size class will appear as a popover controller. You can configure the popover controller by grabbing hold of the popover presentation controller, exactly as you did for popover controllers:

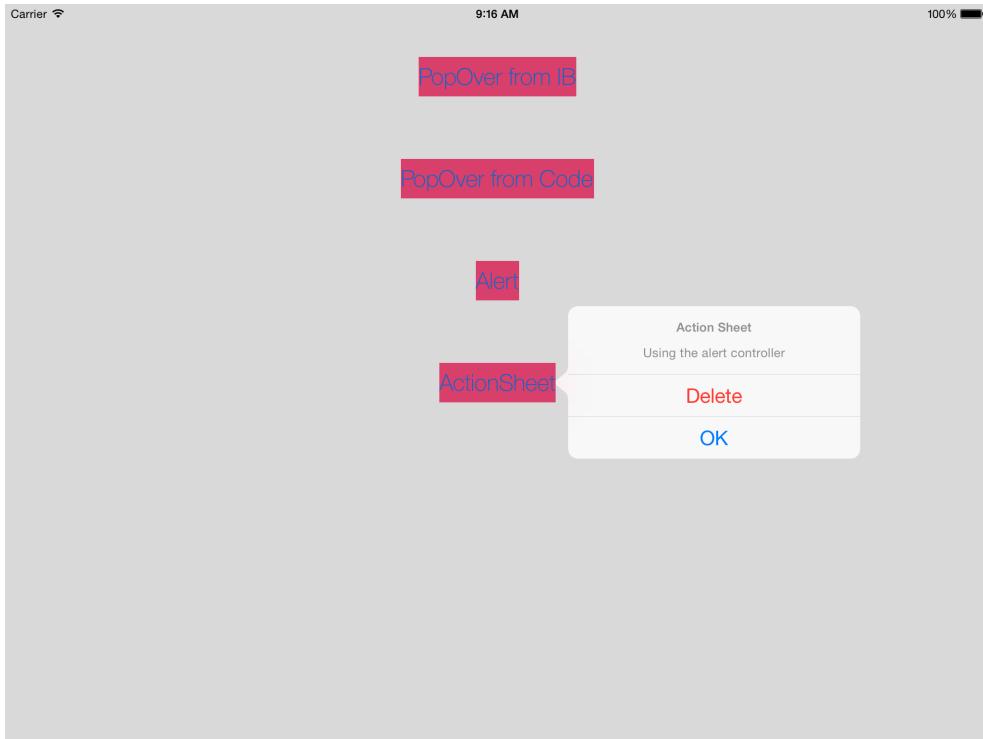
```
1 if let presentationController = actionSheet.popoverPresentationController {
2     presentationController.sourceView = sender
3     presentationController.sourceRect = sender.bounds
4 }
```

And since an action sheet is again a UIViewController subclass, you present it in the same way:

```
1 presentViewController(actionSheet, animated: true, completion: nil)
```



ActionSheet Compact Width



ActionSheet Regular Width

Conclusion

It's great when Apple take a look back at existing parts of the framework and consolidate the way in which it works. This section of UIKit had evolved over many years, and was in need of modernization. The quest for adaptivity and becoming device agnostic has made it an ideal time to rethink the way they work. It's an area that you should make sure you are aware of - it's a lot more understandable than it used to be.

As ever, all the code for the app which accompanies today's article is available on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁶¹.

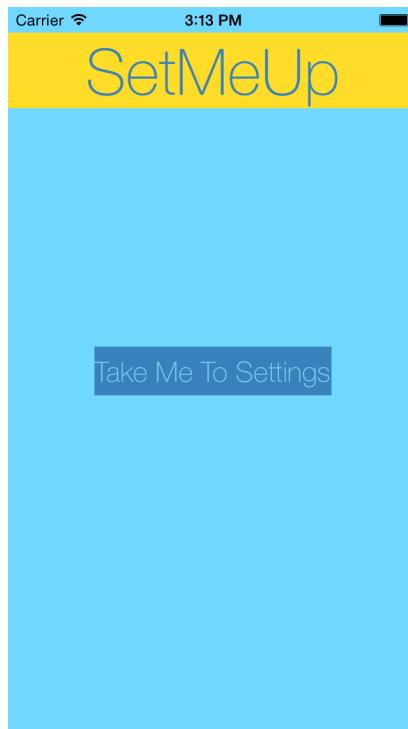
⁶¹<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 22 :: Linking to Settings App



Today's chapter is a short one, but it is definitely worth knowing. iOS provides a really nice way to handle app settings through `NSUserDefaults`, and you've long been able to integrate your app into the iOS settings app via a Settings bundle. However, in order for your user to edit settings they have to make their own way to the settings app. iOS 8 introduces a way to change this.

The app which accompanies today's post is a simple one - a single page app with a button which is linked up to the settings app. There is only one setting, which represents the title label in the app. You can get hold of the source code for the app on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁶².



App Screenshot

Linking to the Settings Page

The sample app includes a title label, whose text is defined from a setting inside the user defaults:

⁶²<https://github.com/ShinobiControls/iOS8-day-by-day>

```

1 let userDefaults = NSUserDefaults.standardUserDefaults()
2 titleLabel.text = userDefaults.stringForKey("AppTitle")

```

In order to edit this default, a settings bundle has been defined, which includes a text field to edit this property:

Key	Type	Value
iPhone Settings Schema	Dictionary	(2 items)
Preference Items	Array	(2 items)
Item 0 (Group - App Settings)	Dictionary	(2 items)
Title	String	App Settings
Type	String	Group
Item 1 (Text Field - Title)	Dictionary	(8 items)
Autocapitalization Style	String	None
Autocorrection Style	String	No Autocorrection
Default Value	String	SetMeUp
Text Field Is Secure	Boolean	NO
Identifier	String	AppTitle
Keyboard Type	String	Alphabet
Title	String	Title
Type	String	Text Field
Strings Filename	String	Root

Settings Bundle

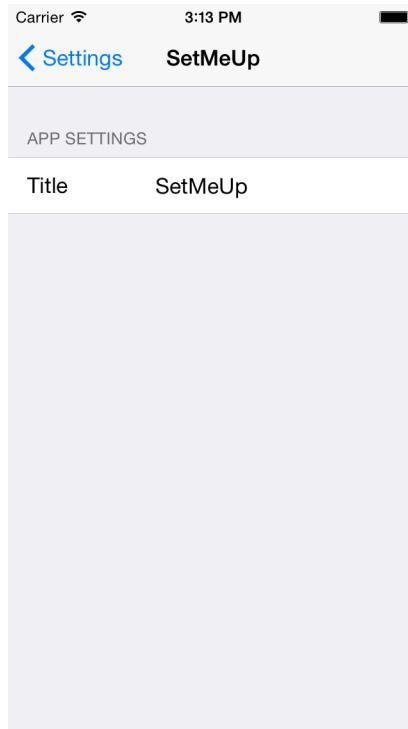
New to iOS 8, there's a new string constant URL which, when provided to `openURL()` on `UIApplication` will take the user to the current app's page within the settings app. The **Take Me To Settings** button has the following handler:

```

1 @IBAction func handleTakeMeButtonPressed(sender: AnyObject) {
2     let settingsUrl = NSURL.URLWithString(UIApplicationOpenSettingsURLString)
3     UIApplication.sharedApplication().openURL(settingsUrl)
4 }

```

The important new thing here is the `UIApplicationOpenSettingsURLString` string constant. Tapping it will switch apps to the settings app, with the correct page opened:



Settings

Since it's really easy to navigate to the settings panel, it's likely the user will go there and then back to your app. If the app doesn't respond to the new settings then it'll be incredibly confusing. Luckily, there is a notification to let you know that some settings have been updated - in the form of `NSUserDefaultsDidChangeNotification`.

You can register for this notification in the standard way:

```
1 override func viewDidLoad() {  
2     super.viewDidLoad()  
3     // Do any additional setup after loading the view, typically from a nib.  
4     NSNotificationCenter.defaultCenter().addObserver(self,  
5             selector: "defaultsChanged",  
6             name: NSUserDefaultsDidChangeNotification,  
7             object: nil)  
8     configureAppearance()  
9 }
```

Where the definitions of `defaultsChanged()` and `configureAppearance()` are as follows:

```
1 func defaultsChanged() {
2     configureAppearance()
3 }
4
5 private func configureAppearance() {
6     let userDefaults = UserDefaults.standardUserDefaults()
7     titleLabel.text = userDefaults.stringForKey("AppTitle")
8 }
```

Don't forget that with `NSNotificationCenter` you must stop observing:

```
1 deinit {
2     NSNotificationCenter.defaultCenter().removeObserver(self)
3 }
```

This will mean that whenever the user updates any settings then the appearance will get updated appropriately.



There is a bug in the iOS 8 betas (last checked beta 5), which means that the settings in the settings app are not persisted to `NSUserDefaults`. Therefore this app doesn't behave as expected. This will be fixed very soon.

Conclusion

As promised - today's chapter was pretty short. If you're already using a settings bundle then this makes life super easy for your users. If you've decided to implement your own settings UI because it's difficult to get to the settings app, then it might be time to check out the settings bundle.

The code for today's project is available on github at github.com/ShinobiControls/iOS8-day-by-day⁶³.

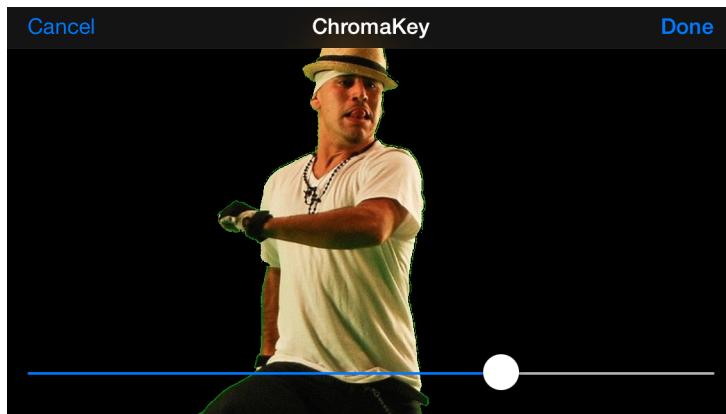
⁶³<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 23 :: Photo Extension



iOS 8 Day-by-Day has looked at a couple of the new extensions in iOS 8 already - the share extension and the today extension. In today's article you're going to learn the basics behind the photo extension.

Extension points are ways that you can write your own code and functionality which enhance the OS in more far-reaching ways than just your app. In the case of photo extensions, you can create filters and more complex editing functionality that will be available to the Photos app.



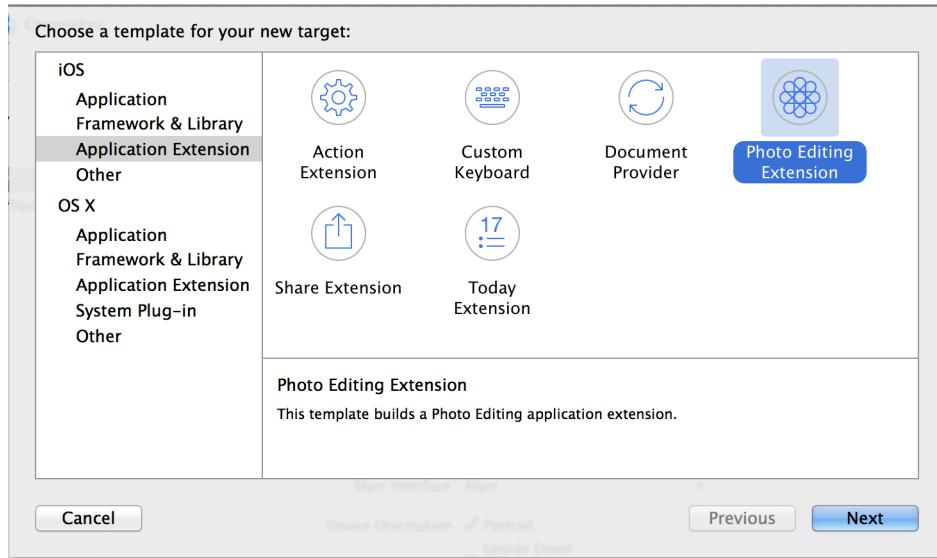
Edit in Progress

The sample code which accompanies today's article builds a photo extension from the chromakey core image filter that was used in day 19. You can get the source code from the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁶⁴.

Creating a Photo Extension

In the same way as for the other extension types, the easiest way to create a photos extension is to use the template provided by Xcode 6:

⁶⁴<https://github.com/ShinobiControls/iOS8-day-by-day>



Extension Template

This creates a new target, and within it a subclass of `UIViewController`, which adopts the `PHContentEditingController` protocol. This adds some lifecycle methods which are specific to photo extensions.

You can probably guess from the fact that this is a view controller, that a photo extension has UI which is provided by you. In fact, the view controller is presented with a navigation bar containing the name of the photo extension. The rest is up to you. Since it's just a subclass of `UIViewController` you can use all the usual techniques, including creating the design in a storyboard.

In the ChromaKey extension which accompanies this article, a simple design including a slider is used to display the live result from the filter, and to configure the threshold. This layout is created in the storyboard.

The project also contains the custom `CIFilter` class from day 19 - and the `PhotoEditingViewController` contains a reference to one:

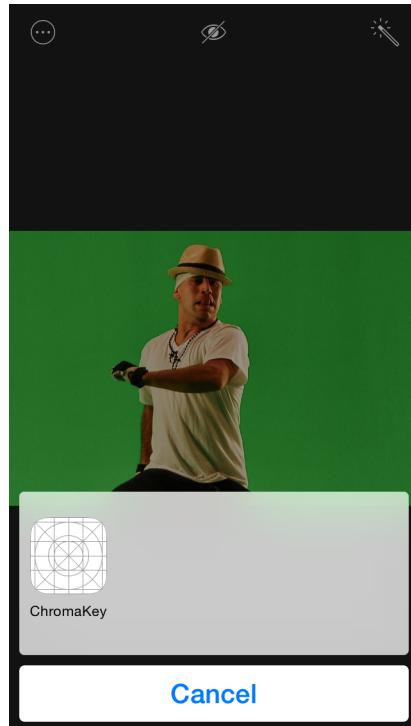
```

1 class PhotoEditingViewController: UIViewController, PHContentEditingController {
2
3     let filter = ChromaKeyFilter()
4     ...
5 }
```

You'll use this filter both during the interactive editing phase, and also when rendering the final image. Note that this doesn't necessarily represent a great use-case here: the ChromaKey filter relies on you creating an image which includes transparency which doesn't fit with rendering as JPEG. The concepts introduced here are all sound, and it wouldn't be too difficult to develop this extension into something more useful.

Starting Interactive Editing

One of the methods defined on `PHContentEditingController` signifies the beginning of the photo editing process - `startContentEditingWithInput(contentEditingInput:, placeholderImage:)`. This is called when the user selects your extension from the list in the edit menu:



Choosing Extension

At this point here it is your responsibility to grab hold of the image, provide it to your filter and enable the interactive controls which allow the user to edit the photo.

```
1 func startContentEditingWithInput(contentEditingInput: PHContentEditingController?,  
2                                     placeholderImage: UIImage) {  
3     input = contentEditingInput  
4     filter.inputImage = CIImage(image: input?.displaySizeImage)  
5     thresholdSlider.value = filter.threshold  
6     updateOutputImage()  
7 }
```

You are provided a `PHContentEditingController` object, which includes info about the media you're editing. Importantly, there are several different ways you can get hold of an input image:

- The supplied `placeholderImage`. This is a `UIImage` of what the output of your filter provided

last time (if you're using resumable editing). Use this if it takes a long time to set up your filter process.

- The `displaySizeImage` property on `contentEditingInput`. This is a `UIImage` which has been appropriately scaled to the current screen size. This should be used for interactive editing, since it will be less processor intensive than using the full-sized image.
- The `fullSizeImageURL` property on `contentEditingInput`. This provides you access to the full-sized image, so that you can apply the final edit to the original image. You should only use this during the edit finalization stages.

In the `ChromaKey` example above, a the `displaySizeImage` is pushed into `CoreImage` and then provided to the filter as the input image. The slider is set to the appropriate value and then `updateOutputImage()` is called, which invokes the filter:

```
1 private func updateOutputImage() {  
2     filter.threshold = thresholdSlider.value  
3     glRenderer?.renderImage(filter.outputImage)  
4 }
```

This method uses a custom `GLRenderer` class to display the image in a `GLKView` on the screen. This ensures that the filtering and display all takes place on the GPU, and the image isn't being passed back and forth through the CPU. The key method of the `GLRenderer` class is `renderImage()`, which is very similar to the code used in the `CoreImage` live detection code from day 13:

```
1 func renderImage(image: CIImage) {  
2     glView.bindDrawable()  
3     if glView.context != EAGLContext.currentContext() {  
4         EAGLContext.setCurrentContext(glView.context)  
5     }  
6  
7     // Calculate the position and size of the image within the GLView  
8     // This code is equivalent to UIViewContentModeScaleAspectFit  
9     let imageSize = image.extent().size  
10    var drawFrame = CGRectMake(0, 0, CGFloat(glView.drawableWidth),  
11                               CGFloat(glView.drawableHeight))  
12    let imageAR = imageSize.width / imageSize.height  
13    let viewAR = drawFrame.width / drawFrame.height  
14    if imageAR > viewAR {  
15        drawFrame.origin.y += (drawFrame.height - drawFrame.width / imageAR) / 2.0  
16        drawFrame.size.height = drawFrame.width / imageAR  
17    } else {  
18        drawFrame.origin.x += (drawFrame.width - drawFrame.height * imageAR) / 2.0  
19    }  
20}
```

```

19     drawFrame.size.width = drawFrame.height * imageAR
20 }
21
22 // clear eagl view to black
23 glClearColor(0.0, 0.0, 0.0, 1.0);
24 glClear(0x00004000)
25
26 // set the blend mode to "source over" so that CI will use that
27 glEnable(0x0BE2);
28 glBlendFunc(1, 0x0303);
29
30 renderContext.drawImage(image, inRect: drawFrame, fromRect: image.extent())
31
32 glView.display()
33 }
```

In order to make the editing interactive, the threshold slider is wired up such that as the user changes the value, the filter is updated, and the image re-rendered:

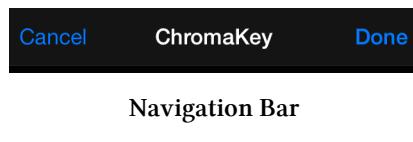
```

1 @IBAction func handleThresholdSliderChanged(sender: UISlider) {
2     updateOutputImage()
3 }
```

That's it for getting the photo extension up and running. Now you need to cope with the next phases of the lifecycle - first up: when the user cancels the edit.

Discard Changes?

When the user sees your extension, part of the framework-provided UI is a navigation bar including **Cancel** and **Done** buttons:



When the user presses the cancel button then you'll get a call to the `cancelContentEditing()` method which will allow you to perform any tidying up of temporary files and the suchlike before the extension disappears.

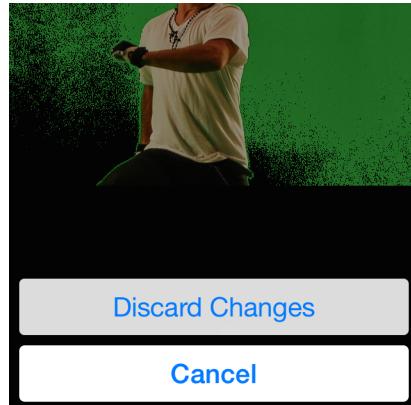
You also get a chance to tell the framework that you'd like a “are you sure?” style dialog to be thrown up before the user dismisses your extension. This is via the `shouldShowCancellationConfirmation` property:

```
1 var shouldShowCancelConfirmation: Bool {  
2     return includesChanges  
3 }
```

In ChromaKey, there is an `includesChanges` boolean property, which defaults to `false`, but is updated to `true` as soon as the user interacts with the slider:

```
1 @IBAction func handleThresholdSliderChanged(sender: UISlider) {  
2     updateOutputImage()  
3     includesChanges = true  
4 }
```

This means that if the user just starts the extension and immediately cancels it then they won't see a question dialog, but as soon as they change any values then it'll prompt them to ask whether they'd like to discard their changes or not:



Discard Changes

Now you've looked at the cancel button, it's time to turn your attention to the more important **Done** button.

Finalizing the Edit

Once the user has completed choosing their settings for the edit, using the interactive editing functionality you've provided for them, they'll tap the **Done** button to save those changes. From your point of view, at this stage you'll want to apply the filter the user has configured to the full size image and then provide this back to the Photos framework for storage in the library.

`PHContentEditingController` specifies a method which will be called when the edit is completed - `- finishContentEditingWithCompletionHandler()`. At this point you should disable any UI, and then apply the filter as it is currently configured to the full-sized image that is provided on the `PHContentEditingControllerInput` object.

The following shows the implementation in **ChromaKey**:

```
1 func finishContentEditingWithCompletionHandler(completionHandler:  
2                                     ((PHContentEditingOutput!) -> Void)!) {  
3     // Update UI to reflect that editing has finished and output is being rendered.  
4     thresholdSlider.enabled = false  
5  
6     // Render and provide output on a background queue.  
7     dispatch_async(dispatch_get_global_queue(  
8                     CLong(DISPATCH_QUEUE_PRIORITY_DEFAULT), 0)) {  
9         // Create editing output from the editing input.  
10        let output = PHContentEditingOutput(contentEditingInput: self.input)  
11  
12        // Write the JPEG Data  
13        let fullSizeImage = CIImage(contentsOfURL: self.input?.fullSizeImageURL)  
14        UIGraphicsBeginImageContext(fullSizeImage.extent().size);  
15        self.filter.inputImage = fullSizeImage  
16        UIImage(CIImage: self.filter.outputImage.drawInRect(fullSizeImage.extent()))  
17        let outputImage = UIGraphicsGetImageFromCurrentImageContext()  
18        let jpegData = UIImageJPEGRepresentation(outputImage, 1.0)  
19        UIGraphicsEndImageContext()  
20  
21        jpegData.writeToURL(output.renderedContentURL, atomically: true)  
22  
23        // Call completion handler to commit edit to Photos.  
24        completionHandler?(output)  
25    }  
26 }
```

The rendering of the larger image all takes place on a background queue, ensuring that you aren't blocking the UI.

Within the rendering routine, the first operation is creating a `PHContentEditingOutput` object from the aforementioned `PHContentEditingInput` object. This has two properties on it - one of which you'll learn about in the *Resumable Editing* section. The property you're interested in here is the `renderedContentURL`. The Photos framework expects that your editing extension will take the image it provided at the `fullSizeImageURL` location (on the `input` object), apply your editing process, and then write the result to the `renderedContentURL`.

The next chunk of code performs just that - pulling the image from the `CoreImage` context into a `CoreGraphics` context, so that a JPEG representation can be created.

Once the output image has been written, then you need to tell the framework that you're done, by calling the supplied `completionHandler()`, passing in the `output` object that you created.

The Photos framework will then update both its data store, and the underlying asset as well. Note that, your edit will not overwrite the original asset - instead you are providing a new version, which

the user can revert at any time. Which leads on rather nicely to the concept of resumable editing.

Resumable Editing

As you make edits to an image, you are never replacing the original image in the Photos library. Instead you provide an edited version of the file, and some data which details exactly how the image was edited. This takes the form of a `PHAdjustmentData` object, which contains three important properties: `formatIdentifier` and `formatVersion` are strings which specify the plugin which performed the edit, and `data` is an `NSData` blob which is used by the editor to save the settings associated with the edit.

When a user requests editing with a specified extension, then if the image has been previously edited, the system will call the `canHandleAdjustmentData()` method on the `PHContentEditingController` object. At this point you can take a look at the adjustment data object and determine whether or not your extension understands it:

```
1 let formatIdentifier = "com.shinobicontrols.chromakey"
2 let formatVersion    = "1.0"
3 func canHandleAdjustmentData(adjustmentData: PHAdjustmentData?) -> Bool {
4     return adjustmentData?.formatIdentifier == formatIdentifier &&
5         adjustmentData?.formatVersion == formatVersion
6 }
```

In the ChromaKey project, the format identifier and version are specified as variables, and the extension will only return true if the edit was made with exactly the same version.

The framework then goes on to call the `startContentEditingWithInput()` method, but the content it provides differs according to the return value of `canHandleAdjustmentData()`. If the extension has said that it understands the adjustment data from the previous edit, then you'll be given the original image, and then have to re-create the filter's settings from before. This allows the user to update their latest edit.

However, if your extension doesn't understand the previous edit, then the framework will provide a pre-rendered image, and your filter will be starting from scratch - effectively layering your edit on top of the previous one.

In the `startContentEditingWithInput()` method, the following lines are added to import the filter settings from the adjustment data:

```
1 if let adjustmentData = contentEditingInput?.adjustmentData {
2     filter.importFilterParameters(adjustmentData.data)
3 }
```

Note that this just calls the following method on `ChromaKeyFilter` to attempt to extract the filter settings from the provided `NSData` blob:

```

1 func importFilterParameters(data: NSData?) {
2     if let data = data {
3         if let dataDict = NSKeyedUnarchiver
4             .unarchiveObjectWithData(data) as? [String : AnyObject] {
5             activeColor = dataDict["color"] as? CIColor ?? activeColor
6             threshold   = dataDict["threshold"] as? NSNumber ?? threshold
7         }
8     }
9 }
```

The `activeColor` and `threshold` properties are preset with defaults, so that if this fails at any point, the filter will resort to its default values:

```

1 var activeColor = CIColor(red: 0.0, green: 1.0, blue: 0.0)
2 var threshold: Float = 0.7
```

In order for the extension to be able to resume an existing edit, then when an edit is completed, it needs to write this `adjustmentData` back to the Photos framework. The `PHContentEditingOutput` object has an `adjustmentData` property which you can populate in your implementation of the completion method, `finishContentEditingWithCompletionHandler()`:

```

1 let newAdjustmentData = PHAdjustmentData(formatIdentifier: self.formatIdentifier,
2                                         formatVersion: self.formatVersion,
3                                         data: self.filter.encodeFilterParameters())
4 output.adjustmentData = newAdjustmentData
```

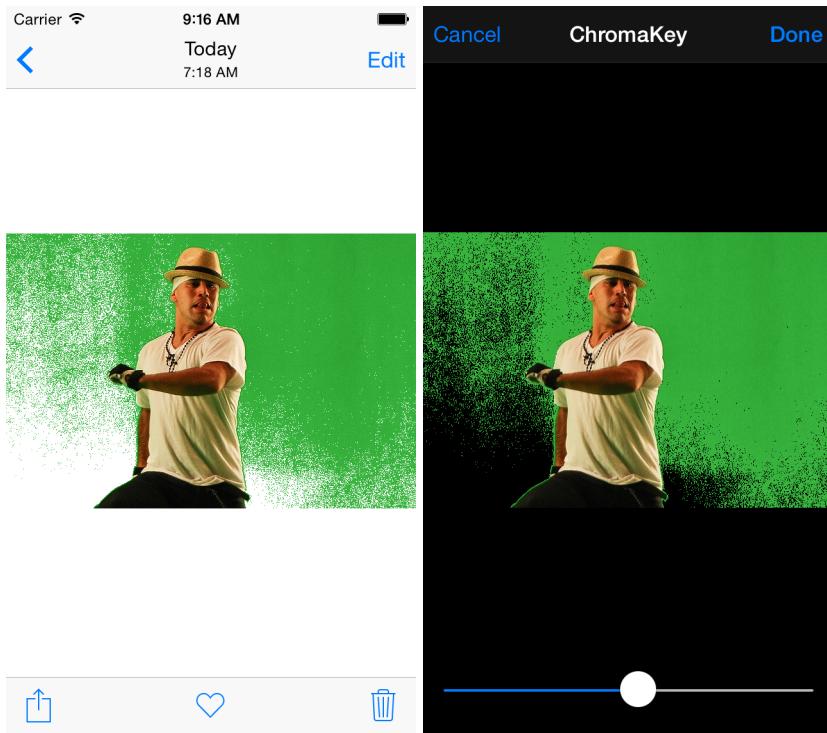
This uses another utility method on `ChromaKeyFilter`:

```

1 func encodeFilterParameters() -> NSData {
2     var dataDict = [String : AnyObject]()
3     dataDict["activeColor"] = activeColor
4     dataDict["threshold"] = threshold
5     return NSKeyedArchiver.archivedDataWithRootObject(dataDict)
6 }
```

Note that `newAdjustmentData` is created using the format identifier and version that are used as a check in `canHandleAdjustmentData()`. This is important, since it is these that all extensions will check against to determine whether they are going to be able to interpret a previous edit, or whether they need to perform their filter on top of a pre-rendered image.

Now, when you run this extension, set a threshold, save the edit and then re-edit it, you'll see that the threshold slider persists between the different edits:



Conclusion

Photo extensions are a quite specialized - unless your app is in the business of providing custom image manipulation algorithms. That doesn't make them any less cool - in fact the fact that Apple has opened up this kind of functionality is really rather exciting for the platform.

In addition to all the easy-access retro-filters that we can all use to take hipster photos of our food, I think it'll be really exciting to see what other ideas people come up with. For example, you could use a photo extension to implement a steganography algorithm right into a filter, and then use it from the photos app. Pretty cool stuff.

The source code for today's algorithm is available, as ever, on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁶⁵.

⁶⁵<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 24 :: Presentation Controllers



Back in chapter 21 you saw how UIKit has been updated to give a more consistent approach to showing alerts, popovers and action sheets. The technology underlying this uses presentation controllers, which are new to iOS 8. It's possible to provide your own custom presentation controllers for use in view controller transitions and presentations.

In today's article you'll learn about what exactly presentation controllers are for, what they can do and how to create your own. The project which accompanies today's post is called **BouncyPresent** and shows how to use a custom presentation controller to create a customized modal view controller presentation. It uses some of the view controller transitioning animation concepts introduced in iOS 7 alongside the new presentation controller concepts to create an easy-to-understand, reusable presentation component.

As ever, the source code is available to clone, download or fork on the ShinobiControls github, at [github.com/ShinobiControls/iOS8-day-by-day⁶⁶](https://github.com/ShinobiControls/iOS8-day-by-day).

The role of the Presentation Controller

Whenever a view controller appears on the screen of an iOS device, it is said to be presented. The improvements to alerts and the suchlike discussed in day 21 included consolidating all the different techniques into using view controllers, so that they too become view controller presentations. In order to control how a view controller is presented, iOS 8 introduces `UIPresentationController`.

UIKit uses a presentation controller to manage the presentation of view controllers from the moment they appear, through their lifetime, until they are dismissed. The common modal presentation styles (such as popover) have built-in presentation controllers, which exhibit the desired behavior, but if you wish to provide your own then you must set the `modalPresentationStyle` to `.Custom` (further implementation detail will be discussed later).

The presentation controller is responsible for positioning the view controller which is being presented, and for adding any extra views to the view hierarchy as appropriate. For example, in the sample app today, the presentation controller will ensure that the presented view controller is inset from the container, and a view which appears to dim the background will be added behind the presented view controller.

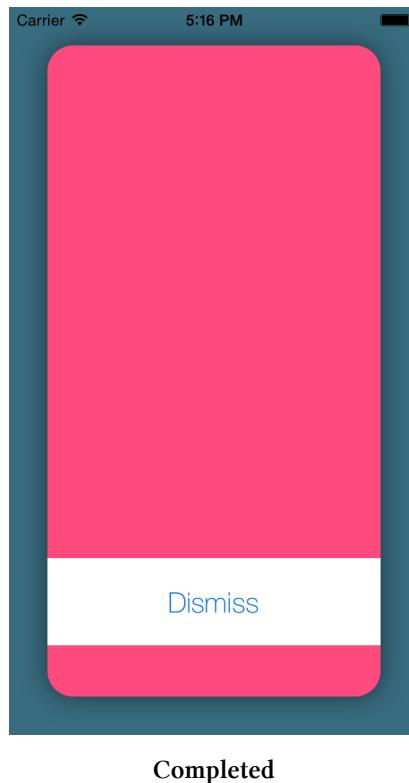
On the surface of it, presentation controllers don't allow you to do anything that you couldn't do before using the view controller transitioning system in iOS 7, but the responsibilities are much

⁶⁶<https://www.shinobicontrols.com/iOS8-day-by-day>

clearer. An animation controller is responsible for the animation and the display of the view controller's content, and the presentation controller is responsible for the appearance of everything else. In the past, view controllers were responsible for views which weren't within their own view hierarchy, which was confusing at best. As you'll see, presentation controllers control the wider view hierarchy, and have hooks which allow them to animate alongside the existing animations.

Creating a custom Presentation Controller

Custom presentation controllers inherit from `UIPresentationController`, which provides various methods which hook in to the lifecycle of a view controller presentation. In the associated sample project, `OverlayPresentationController` is a custom presentation controller, which ensures that the presented view controller is inset from the container, and the background content is suitably dimmed:



Completed

The dimming view is a property on the `OverlayPresentationController`, so that a reference is maintained:

```
1 class OverlayPresentationController: UIPresentationController {  
2     let dimmingView = UIView()  
3     ...  
4 }
```

The view is configured more fully during the initialization of the presentation controller. Note that `init(presentedViewController:, presentingViewController:)` is the designated initializer of `UIPresentationController`, so it's imperative that the super method is called:

```
1 override init(presentedViewController: UIViewController!,  
2                 presentingViewController: UIViewController!) {  
3     super.init(presentedViewController: presentedViewController,  
4                 presentingViewController: presentingViewController)  
5     dimmingView.backgroundColor = UIColor(white: 0.0, alpha: 0.5)  
6 }
```

This dimming view should be added to the view hierarchy when the view controller is presented, and removed when it is dismissed. The `presentationTransitionWillBegin()` method is called when the presentation is starting:

```
1 override func presentationTransitionWillBegin() {  
2     dimmingView.frame = containerView.bounds  
3     dimmingView.alpha = 0.0  
4     containerView.insertSubview(dimmingView, atIndex: 0)  
5  
6     presentedViewController.transitionCoordinator().animateAlongsideTransition({  
7         context in  
8             self.dimmingView.alpha = 1.0  
9     }, completion: nil)  
10 }
```

Here the dimming view is sized and positioned appropriately, and set as fully transparent. It is also added at the back of the container view's view hierarchy, i.e. behind the presented view controller's view.

Then, in order to fade the dimming view in, you can grab hold of the transition coordinator from the presented view controller and use the `animateAlongsideTransition()` method to specify animations which will occur as part of the view controller's transition animation. Here, the alpha of the view is updated so that it appears.

The opposite approach is used in the `dismissTransitionWillBegin()` method to remove the dimming view:

```
1 override func dismissalTransitionWillBegin() {
2     presentedViewController.transitionCoordinator().animateAlongsideTransition({
3         context in
4             self.dimmingView.alpha = 0.0
5     }, completion: {
6         context in
7             self.dimmingView.removeFromSuperview()
8     })
9 }
```

If you run the app now, then you won't actually see any difference in your presentation. This is for two reasons:

1. UIKit isn't aware that it should be using your custom presentation controller.
2. The presented view still has its default, full-screen size. You won't see the dimming view until the presented view is smaller.

You can fix the latter of these with the `frameOfPresentedViewInContainerView()` method on your presentation controller:

```
1 override func frameOfPresentedViewInContainerView() -> CGRect {
2     return containerView.bounds.rectByInsetting(dx: 30, dy: 30)
3 }
```

This ensures that the presented view will have a frame inset from the container view by 30 points on each edge.

Using the custom Presentation Controller

The `UIViewControllerTransitioningDelegate` protocol was introduced in iOS 7 powers customizable view controller transitions. In iOS 8, this has been extended with a new method for providing a presentation controller. You need to create a transitioning delegate object which adopts this protocol and returns the presentation controller that you've created:

```

1 class OverlayTransitioningDelegate : NSObject,
2                               UIViewControllerTransitioningDelegate {
3     func presentationControllerForPresentedViewController(
4         presented: UIViewController!,
5         presentingViewController presenting: UIViewController!,
6         sourceViewController source: UIViewController!) -
7         > UIPresentationController! {
8
9     return OverlayPresentationController(presentedViewController: presented,
10                                         presentingViewController: presenting)
11 }
12 }
```

Note that UIKit will maintain a strong reference to the presentation controller for the duration of the presentation, so you don't need to.

There are two main ways of initiating a view controller presentation, through a segue in a storyboard, or entirely through code. You can customize the presentation for both of these.

Presenting from code involves creating the view controller and then using `presentViewController(_:, animated:, completion:)` to present it:

```

1 @IBAction func handleBouncyPresentPressed(sender: AnyObject) {
2     let overlayVC = storyboard.instantiateViewControllerWithIdentifier(
3         "overlayViewController") as UIViewController
4     prepareOverlayVC(overlayVC)
5     presentViewController(overlayVC, animated: true, completion: nil)
6 }
```

Importantly, the `prepareOverlayVC(_)` utility method does the necessary customization:

```

1 private func prepareOverlayVC(overlayVC: UIViewController) {
2     overlayVC.transitioningDelegate = overlayTransitioningDelegate
3     overlayVC.modalPresentationStyle = .Custom
4 }
```

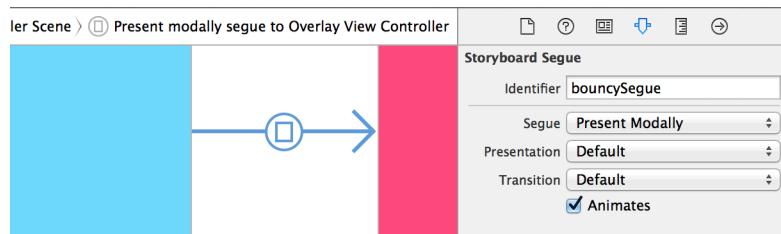
Here, the `modalPresentationStyle` is set to `.Custom`, which means that UIKit will query the transitioning delegate for a presentation controller, rather than use one of its internal implementations. The `transitioningDelegate` property is also set. Note that since this is a delegate object, UIKit will not maintain a reference, so you need to keep a transitioning delegate around:

```
1 let overlayTransitioningDelegate = OverlayTransitioningDelegate()
```

You can use this same utility method for customizing the presentation of a view controller via a segue:

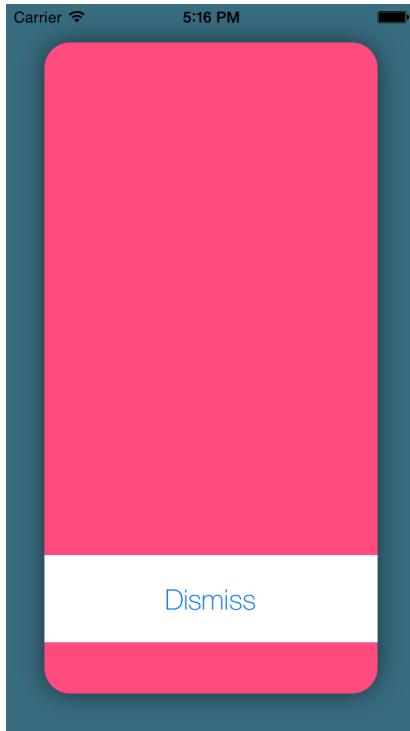
```
1 override func prepareForSegue(segue: UIStoryboardSegue!, sender: AnyObject!) {
2     if segue.identifier == "bouncySegue" {
3         let overlayVC = segue.destinationViewController as UIViewController
4         prepareOverlayVC(overlayVC)
5     }
6 }
```

The segue is appropriately named in the storyboard, and then the `prepareForSegue()` method will configure the presented view controller when appropriate:



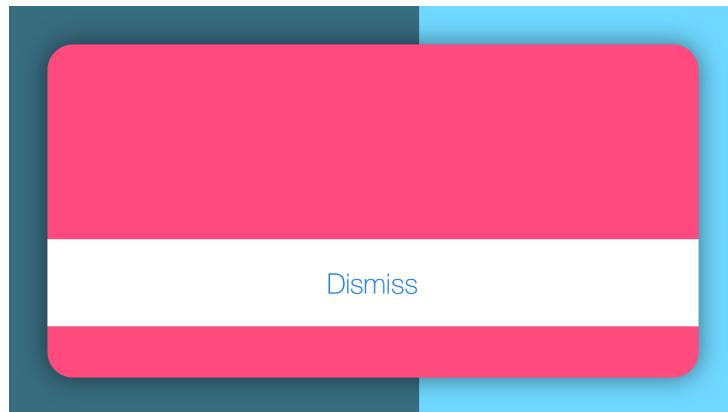
Segue naming

You can now run the app up, and you'll see that you have successfully managed to customize the presentation of your overlay view controller:



Completed Product

However, if you rotate the device, then you'll notice that your view isn't adapting to the bounds change as you might expect:



Not Adaptive

Let's take a look at ways to fix that.

Adaptive UI with Presentation Controllers

One of the core facilitators of adaptive layout in UIKit is the `UIContentContainer` protocol, which you've already seen (back on [day 7⁶⁷](#)) `UIViewController` adopts. This is responsible for two levels of layout adaptivity - coarse grained changes through trait collections changing (i.e. size classes), and more fine-grained changes via view sizes changing.

Since presentation controllers can be thought of as the root 'controller' for a presented view controller, they too adopt the `UIContentContainer` protocol, so once again you get the opportunity to adapt the presentation according to the trait collection and size.

You also get an even more fine-grained layer in the form of `containerViewWillLayoutSubviews()` and `containerViewDidLayoutSubviews()`. These are equivalent to their `UIViewController` counterparts `viewWillLayoutSubviews()` and `viewDidLayoutSubviews()`, and you can use them to precisely control the appearance whenever a layout pass occurs.

There are several possible approaches to fixing the dimming view resizing problem you saw at the end of the last section:

- Create the dimming view with autolayout constraints that will pin it to the edges of the content view.
- Override `viewWillTransitionToSize(size:, withTransitionCoordinator:)` in the presentation controller and update the dimming view appropriately.
- Override `containerViewWillLayoutSubviews()` and again, update the dimming view appropriately.

Since we all know how to use autolayout, and [day 14⁶⁸](#) introduced the size transition method on `UIContentContainer`, in the accompanying sample project, the last option is used:

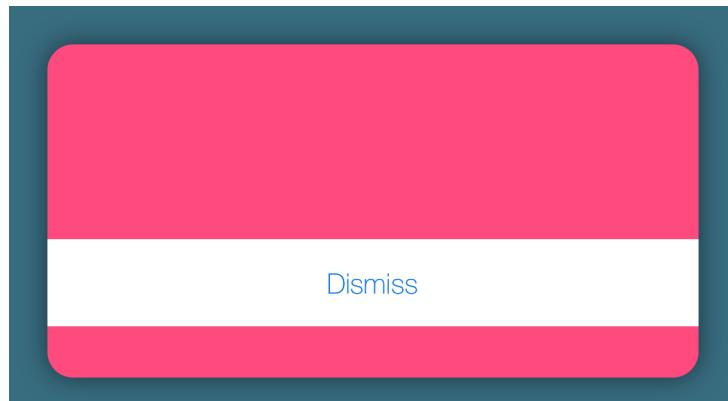
```
1 override func containerViewWillLayoutSubviews() {  
2     dimmingView.frame = containerView.bounds  
3     presentedView().frame = frameOfPresentedViewInContainerView()  
4 }
```

Here, the frames of both the dimming view and the presented view are updated to ensure they are correctly sized and positioned.

Run the app up again now and you'll see that the rotation behavior is as expected:

⁶⁷<http://www.shinobicontrols.com/blog/posts/2014/07/28/ios8-day-by-day-day-7-adaptive-layout-and-utraitcollection/>

⁶⁸<http://www.shinobicontrols.com/blog/posts/2014/08/06/ios8-day-by-day-day-14-rotation-deprecation/>



Landscape Correct

Custom Presentation Animation

The focus of today's chapter isn't creating custom animations for view controller transitions. This functionality was introduced back in iOS 7 (in fact chapter 10 of [iOS 7 Day-by-Day](#)⁶⁹ covers the architecture in some detail), but it is worth seeing how this interacts with the new presentation controllers.

Custom animations are the realm of the `UIViewControllerAnimatedTransitioning` protocol, and as such require just two methods to be implemented:

```
1 class BouncyViewControllerAnimator : NSObject,
2                               UIViewControllerAnimatedTransitioning {
3
4     func transitionDuration(transitionContext:
5                             UIViewControllerContextTransitioning) -> NSTimeInterval {
6         return 0.8
7     }
8
9     func animateTransition(transitionContext:
10                            UIViewControllerContextTransitioning) {
11         if let presentedView = transitionContext
12             .viewForKey(UIViewTransitionContextToViewKey) {
13             let centre = presentedView.center
14             presentedView.center = CGPointMake(centre.x,
15                                               -presentedView.bounds.size.height)
16             transitionContext.containerView().addSubview(presentedView)
17 }
```

⁶⁹<https://leanpub.com/iOS7DayByDay>

```
18     UIView.animateWithDuration(self.transitionDuration(transitionContext),
19         delay: 0, usingSpringWithDamping: 0.6, initialSpringVelocity: 10.0,
20         options: nil, animations: {
21             presentedView.center = centre
22         }, completion: {
23             _ in
24             transitionContext.completeTransition(true)
25         })
26     }
27 }
28 }
```

You'll notice here that `animateTransition()` is only concerned with the presented view - obtaining it, positioning it, adding it to the view hierarchy and animating it. In the past, the animator would also be trying to animate other aspects of the view hierarchy (such as the dimming view), but with the introduction of the presentation controller, it has much more well-defined responsibilities.

The other thing you need to remember to do is to update the transitioning delegate so that UIKit knows there is a custom animator it should be using:

```
1 func animationControllerForPresentedController(presented: UIViewController!, pre\
2 sentingController presenting: UIViewController!, sourceController source: UIView\
3 Controller!) -> UIViewControllerAnimatedTransitioning! {
4     return BouncyViewControllerAnimator()
5 }
```

And then you're done. You can run the app up again now and you'll see that in place of the rather dull ease-in-out animation curve, you've now got a more interesting bounce.

Conclusion

On the surface of it you might not think that presentation controllers offer you much more functionality than you had in iOS 7. In some respects you'd be correct. However, the much better software design with the separation of concerns and single responsibility goes a long way to understanding the protocol soup that is view controller presentations.

Presentation controllers are also imperative in the adoption of adaptivity throughout UIKit. To get great adaptivity you need not only to be able to adapt layouts, but also view controller hierarchies. This would have been incredibly difficult using just view controller hierarchies. Presentation controllers which respond to the adaptive concepts of `UIContentContainer` give you, the developer, great power to completely change the appearance of an app, including the view controller hierarchy, in a device and context dependent manner.

I think that it's going to take a while before we fully understand the best practices and usefulness of presentation controllers, but that's true of the entire adaptive approach to app design. In the long run I think it'll result in easier to understand, more maintainable apps, which work on a huge variety of devices.

Don't forget that the code for today's article is available at github.com/ShinobiControls/iOS8-day-by-day⁷⁰.

⁷⁰<https://www.shinobicontrols.com/iOS8-day-by-day>

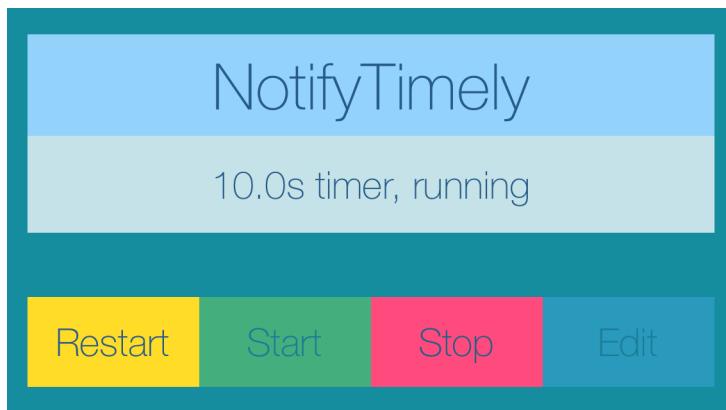
Day 25 :: Notification Actions



Local and remote notifications have been an integral part of iOS for many years now - allowing apps to respond to events even whilst they aren't running. Notifications are either silent or visual, with visual notifications taking the form of alerts or banners which the user can interact with. The interaction is very limited though - either dismissing the notification or opening the app.

iOS 8 introduces new notification actions - which allow you to hook into the notification system and provide a set of customized actions that the user will see and can execute alongside the notification.

This chapter digs into the new notification actions by creating a simple timer app based on local notifications. Exactly the same principles apply for providing custom actions for remote notifications too - just using push notifications as a trigger. The sample application is called **NotifyTimely** and the source code is available on github at github.com/ShinobiControls/iOS8-day-by-day⁷¹.



Timer App

Requesting Permission

Before getting started with the exciting new notification actions, it's worth making note of some other changes to the UIKit notification system - in particular the way you request permission from the user to present user notifications.

In the past you had to request permission from the user for some aspects of remote notifications, but not for local notifications. And the type of remote notification (silent versus user) was mixed in with the request.

⁷¹<https://github.com/ShinobiControls/iOS8-day-by-day>

In iOS 8 these two orthogonal concepts have been split out: an app has to register both for its interest in receiving remote notifications, and also for the specific types of user notifications it would like to utilize (for both remote and local use).

Remote Notifications

Registering for remote notifications is now super-simple - there is a method on `UIApplication` which will perform the operation for you:

```
1 UIApplication.sharedApplication().registerForRemoteNotifications()
```

The user won't actually be prompted at this point - new apps will be enabled by default, however a user can disable remote notifications through the settings app. You can check the status using the `isRegisteredForRemoteNotifications()` method.

You need to do this to enable remote notifications of both the silent and user variety, and if this is enabled then silent notifications will work as expected. However, in order for user notifications to appear, then you need to register these as well.

User Notifications

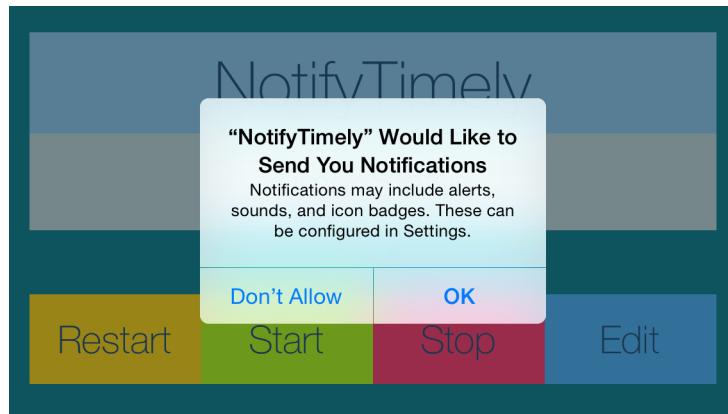
User notifications are alerts, icon badges and sounds which are used to gain the user's attention. The source for these can be either remote or local notifications and, since they can be quite disruptive, require the user's permission.

In iOS 8, this is done using the `registerUserNotificationSettings()` method on `UIApplication`, providing a `UIUserNotificationSettings` object to specify what permissions you require. In the simplest form you can use the following code to register for sounds and alerts:

```
1 let requestedTypes = UIUserNotificationType.Alert | .Sound
2 let settingsRequest = UIUserNotificationSettings(forTypes: requestedTypes,
3                                                 categories: nil)
4 UIApplication.sharedApplication()
5         .registerUserNotificationSettings(settingsRequest)
```

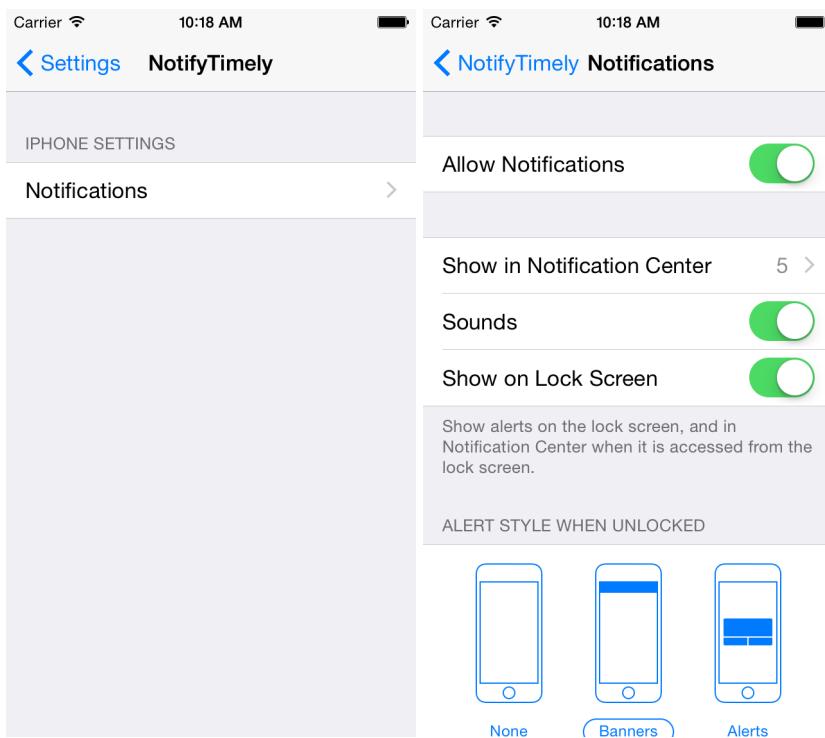
`UIUserNotificationType` can have values `.Badge`, `.Sound`, `.Alert` or `.None`, and the `categories` parameter is related to custom actions - you'll learn about it in the next section.

When the `registerUserNotificationSettings()` method is first run, the user will see an alert which asks for permission to use notifications:



Requesting Permission

They won't see the alert again, but can update their choices in the app's page in the settings app:



Once the user has responded to the user notification alert, UIKit will call the `application(_:, didRegisterUserNotificationSettings:)` on the app delegate, with the results of the request. Alternatively, the `currentUserNotificationSettings()` method on `UIApplication` will provide you with the most up-to-date settings. You can use this to alter your app's behavior appropriately - preventing the creation of notifications which will never appear.

That completes this aside about notification permissions - now on to the more interesting topic of notification actions.

Registering Actions

On first glance, the workflow for providing custom actions for notifications might seem a little complicated, but it really isn't. Since user notifications could be fired both by remote and local notifications, it's not possible to completely define a notification and its actions within the notification itself - remote notifications need to be as small as possible.

Therefore the different notifications are pre-defined. Well, the *category* of notification is pre-defined - obviously the content is dependent on the notification itself.

A `UIUserNotificationCategory` defines a set of actions which will accompany a notification of that type. For example a "new email" notification would want actions such as reply and delete, whereas a "timer fired" notification would need restart and edit.

The actions themselves are formed from `UIUserNotificationAction` objects, more often created using the `UIMutableUserNotificationAction` counterpart. For example, the action associated with restarting the timer:

```
1 let restartAction = UIMutableUserNotificationAction()
2 restartAction.identifier = restartTimerActionString
3 restartAction.destructive = false
4 restartAction.title = "Restart"
5 restartAction.activationMode = .Background
6 restartAction.authenticationRequired = false
```

The properties: - `identifier` is a string which you'll use to determine which button the user pressed when handling the notification action. - `destructive` if true will present the button in a red "danger" color - `title` the text that appears on the button - `activationMode` determines whether or not the app needs to be launched in the background or the foreground. If you need further UI to complete the action then the app must be launched in the foreground, but if you just need a few seconds of CPU time to complete processing the action, then this can be performed in the background. - `authenticationRequired` if the user needs to enter their PIN or TouchID in order to carry out the action. Note that if you have a background action which does not require authentication then the device will remain locked. Therefore, you need to prepare for the level of access to the keychain and filesystem that you'll be provided.

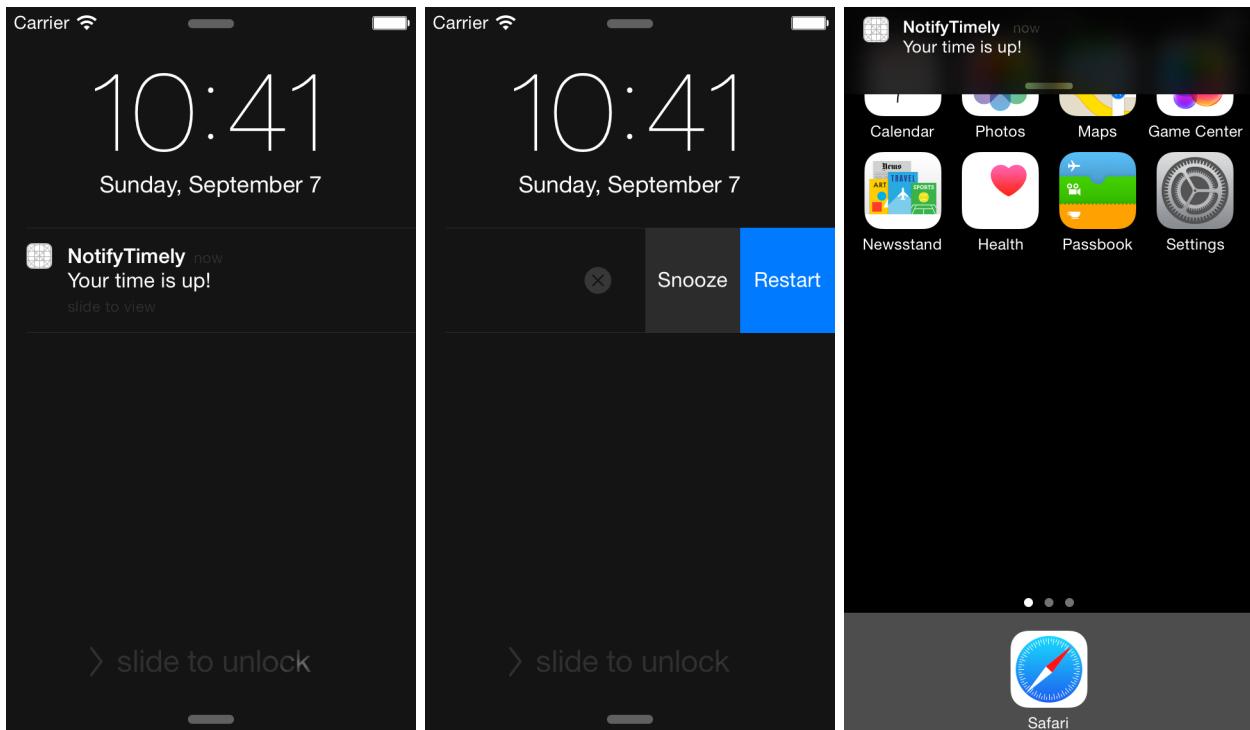
Once you've created the actions that you want to use, you have to collect them together in a `UIUserNotificationCategory`, via its mutable sister:

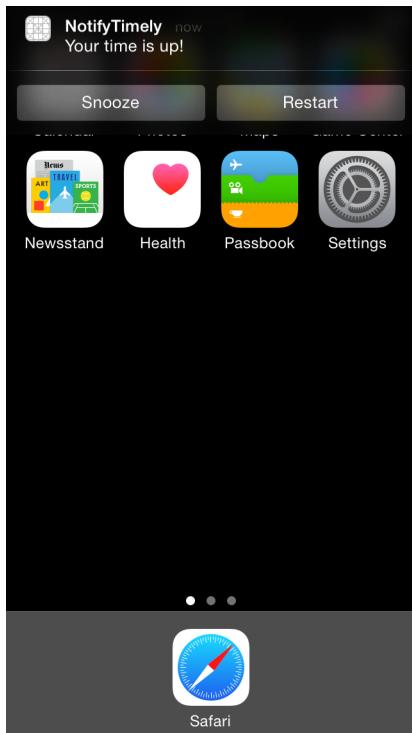
```
1 let category = UIUserNotificationCategory()  
2 category.identifier = timerFiredCategoryString  
3 category.setActions([restartAction, snoozeAction], forContext: .Minimal)  
4 category.setActions([restartAction, snoozeAction, editAction],  
5 forContext: .Default)
```

The `identifier` is again a string, and is this time used to specify which set of actions the system should present to a user when a notification fires. You'll see how to use this in local notifications in the next section, but for remote notifications, a new `category` key has been added to the JSON spec for push messages, and the value should be one of the notification categories you've registered for your app.

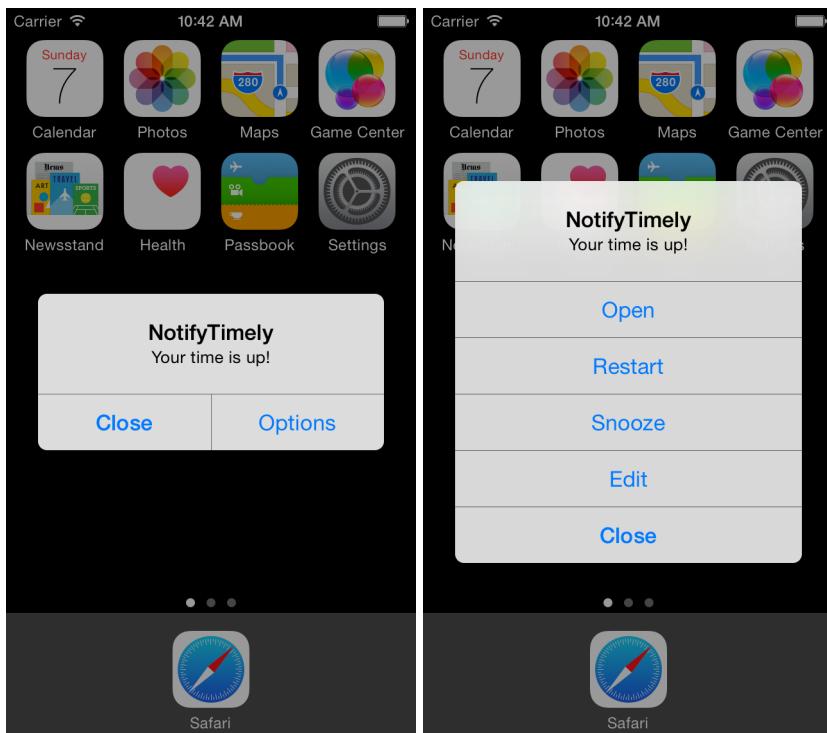
Call `setActions(_:, forContext:)` to add the actions to the `UIUserNotificationCategory`. The first parameter is just an array of the actions you wish to add, and the second parameter is of type `UIUserNotificationActionContext`, which can be either `.Minimal` or `.Default`. This context parameter is used which actions to show in different scenarios:

- `.Minimal` - when a notification appears on the lock screen or notification center and the user drags to the left, or when the notification is a banner and the user drags down. These all have a maximum of two actions due to limited space.





- Default - when a notification appears as an alert and the user selects the Options button. The alert view will expand to a sheet, where a maximum of four actions can be provided to supplement the system Open and Close actions.



Once you've created all the user alert categories your app will need, you must ensure that you register them with the system, by updating your registration code as follows:

```
1 let requestedTypes = UIUserNotificationType.Alert | .Sound
2 let categories = NSSet(object: timerFiredNotificationCategory())
3 let settingsRequest = UIUserNotificationSettings(forTypes: requestedTypes,
4                                                 categories: categories)
5 UIApplication.sharedApplication()
6                     .registerUserNotificationSettings(settingsRequest)
```

This uses the `categories` argument of `UIUserNotificationSettings()`, but providing an `NSSet` of the different categories your app will require.

Now you've done that, your app is ready to display these notifications with custom actions, so let's see how to do that with local notifications.

Firing Actions

To support these new user notification categories, `UILocalNotification` has a new `category` property in iOS 8. This is a string, and its value should either be `nil`, or match the identifier of one of the categories registered with the app.

In the timer app, a new local notification is created with a specific offset time, using the following utility method:

```
1 private func createTimer(fireOffset: Float) -> UILocalNotification {
2     let notification = UILocalNotification()
3     notification.category = timerFiredCategoryString
4     notification.fireDate = NSDate(timeIntervalSinceNow:
5                                     NSTimeInterval(fireOffset))
6     notification.alertBody = "Your time is up!"
7     return notification
8 }
```

Note that here the `timerFiredCategoryString` is part of a set of global constant strings created:

```
1 let restartTimerActionString = "RestartTimer"
2 let editTimerActionString = "EditTimer"
3 let snoozeTimerActionString = "SnoozeTimer"
4 let timerFiredCategoryString = "TimerFiredCategory"
```

This notification is then scheduled in the standard way:

```
1 let timer = createTimer(fireOffset)
2 UIApplication.sharedApplication().scheduleLocalNotification(timer)
```

Now, provided the app isn't in the foreground, when the notification fires, the system will present context dependent UI to the user, as shown in the screenshots in the previous section.

This is great, but tapping on the buttons doesn't actually do anything, yet.

Handling Actions

All the previous application delegate call-backs associated with notifications still exist and work, but two new ones have been added:

- `application(_ handleActionWithIdentifier:, forRemoteNotification:, completionHandler:)`
- `application(_ handleActionWithIdentifier:, forLocalNotification:, completionHandler:)`

Clearly the only difference between these two methods is the original source of the notification - all of the other parameters are the same.

The action identifier argument is a string, that corresponds to the `identifier` property on the previously-created `UIUserNotificationAction`. This is used to determine which button the user actually pressed.

The `completionHandler` argument is a closure which you need to call once you've finished processing the action. It's great that you're provided this - it means that you can perform asynchronous operations without the system killing your process when you're in the background.

The following is the implementation of the handler for local notification in the accompanying timer app:

```
1 func application(application: UIApplication,
2                   handleActionWithIdentifier identifier: String?,
3                   forLocalNotification notification: UILocalNotification,
4                   completionHandler: () -> Void) {
5   // Pass the action name onto the manager
6   timerNotificationManager.handleActionWithIdentifier(identifier)
7   completionHandler()
8 }
```

In this instance, the action is handled by the timer manager in the following method:

```
1 func handleActionWithIdentifier(identifier: String?) {
2     timerRunning = false
3     if let identifier = identifier {
4         switch identifier {
5             case restartTimerActionString:
6                 restartTimer()
7             case snoozeTimerActionString:
8                 scheduleTimerWithOffset(snoozeDuration)
9             case editTimerActionString:
10                delegate?.presentEditOptions()
11            default:
12                println("Unrecognised Identifier")
13        }
14    }
15 }
```

Note that if your action was specified to be `.Foreground`, then the handle action method will be called after the app has relaunched into the foreground. This means in the case of the timer app, that the `delegate` property on the timer manager will have been set, and therefore the following code will run:

```
1 func presentEditOptions() {
2     performSegueWithIdentifier("configureTimer", sender: self)
3 }
```

Resulting in the edit UI being presented - right from tapping a button in a notification:



If the actions are specified to be `.Background`, then the UI will obviously not be loaded, and you'll get a few seconds to complete whatever tasks you need.

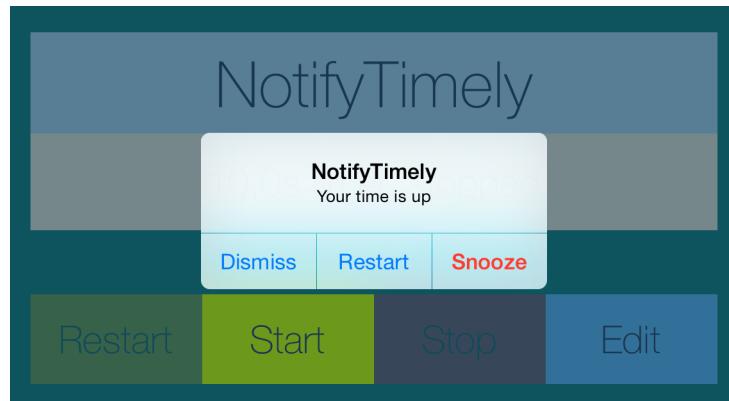
Foreground Notifications

As before, UIKit will only present the notification UI if the app is not currently active (i.e. in the foreground). Although this hasn't changed, it's still worth remembering how to create your own alert which appears when the foreground app receives a notification:

```
1 func application(application: UIApplication,
2 didReceiveLocalNotification notification: UILocalNotification) {
3     // Pass the "firing" event onto the notification manager
4     timerNotificationManager.timerFired()
5     if application.applicationState == .Active {
6         let alert = UIAlertController(title: "NotifyTimely",
7                                         message: "Your time is up",
8                                         preferredStyle: .Alert)
9     // Handler for each of the actions
10    let actionAndDismiss = {
11        (action: String?) -> ((UIAlertAction!) -> ()) in
12        return {
13            _ in
14            self.timerNotificationManager.handleActionWithIdentifier(action)
15            self.window?.rootViewController?
16                .dismissViewControllerAnimated(true, completion: nil)
17        }
18    }
19
20    alert.addAction(UIAlertAction(title: "Dismiss", style: .Cancel,
21                                 handler: actionAndDismiss(nil)))
22    alert.addAction(UIAlertAction(title: "Restart", style: .Default,
23                                 handler: actionAndDismiss(restartTimerActionString)))
24    alert.addAction(UIAlertAction(title: "Snooze", style: .Destructive,
25                                 handler: actionAndDismiss(snoozeTimerActionString)))
26    window?.rootViewController?.presentViewController(alert,
27                                                    animated: true, completion: nil)
28}
29}
```

This method checks to see that the app state is `.Active`, before creating a new `UIAlertController` to present to the user. The actions associated with the alert actually just trigger exactly the same handler methods as the notification alert handler does, so actually the amount of duplicated code is kept to a minimum.

This produces an alert like this:



Foreground Alert

Conclusion

The most important part of today's post is that you absolutely **must** update your existing apps. Otherwise, come iOS 8, users will no longer see the notifications you'd expect them to. The API for registering for notifications is pretty simple, and separating out the arrival vector from the user notification permissions makes it easier for both developers and users to understand.

That's not the most fun part though. Being able to specify custom actions for notifications could be really quite powerful. In some cases you'll be able to have entire apps which provide incredibly powerful functionality whilst never actually being launched into the foreground. Even though you only get two or four 'slots' in which to put your custom actions, there is a lot you can do with that, and I'm quite excited to see what the apps whose notifications I read every day decide to do with them.

As ever, all the code for today's sample project **NotifyTimely** is available on the ShinobiControls github page at github.com/ShinobiControls/iOS8-day-by-day⁷².

⁷²<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 26 :: AVKit



If you want to play a video in iOS then you've traditionally been presented with two choices. If you want simple playback with framework provided UI then you could use `MPMoviePlayerViewController`, or if you require fine-grain control of the underlying `AVFoundation` pipeline then you could use an `AVPlayer`, rendered through an `AVPlayerLayer`. The problem with these approaches is that the former doesn't give you much control over the playback process, whereas the latter requires that you create your own UI.

AVKit is new to iOS 8 and pulls both scenarios into a common pipeline. `AVPlayerViewController` effectively deprecates `MPMoviePlayerViewController`, and sits on top of `AVFoundation`. It provides contextual video playback UI that matches that of the OS, and plays any `AVPlayer` object from `AVFoundation`.

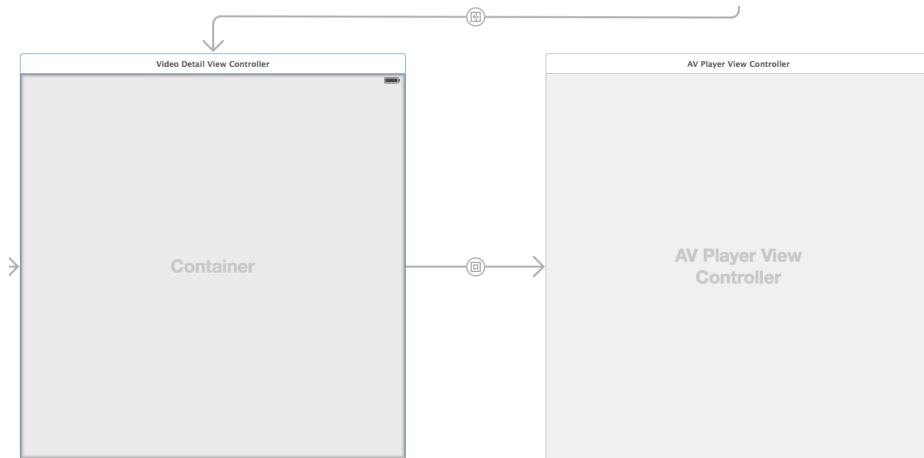
In today's chapter you'll discover how easy it is to integrate AVKit into your app, and see some of the cool new things you can do. The app which accompanies the article is a simple video player, which uses the Photos framework to find videos in the user's library, and then plays them using AVKit. You can download the source code from the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁷³](https://github.com/ShinobiControls/iOS8-day-by-day). If you're using the simulator then you can add some videos to the library by dragging them in from the finder, and then selecting "save" from the sharing menu.

Using AVKit to play a video

AVKit is a very simple framework - consisting of just one class on iOS - `AVPlayerViewController`. This is a subclass of `UIViewController` with a few additional properties associated with video playback. The most important of these is the `player` property, which is of type `AVPlayer`. This is a class from `AVFoundation` and represents the control of playback. In the past if you wanted to use `AVPlayer`, you'd use the `AVPlayerLayer` subclass of `CALayer` to visualize the video stream, and create your own UI for video controls. `AVPlayerViewController` handles both of these things for you, simply by setting the `player` property to an `AVPlayer` object.

`AVPlayerViewController` is fully integrated into Interface Builder - so you can just drag one onto the storyboard from the object library:

⁷³<https://github.com/ShinobiControls/iOS8-day-by-day>



Storyboard Integration

In this example, the player is contained by the `VideoDetailViewController`, which sets up the `player` property appropriately.



Note: If you do reference AVKit within the storyboard, then you may well have to manually link against the library yourself - like off've the olden days. You can do this from the project settings page, in the *Linked Frameworks and Libraries* panel.

There are two ways you can create an `AVPlayer` - either with a URL or a `PlayerItem`. The URL approach matches the `MPMoviePlayerViewController` use case - where the location of a video file, either local or remote, is provided. AVFoundation then takes care of grabbing the content, buffering it and then playing it back.

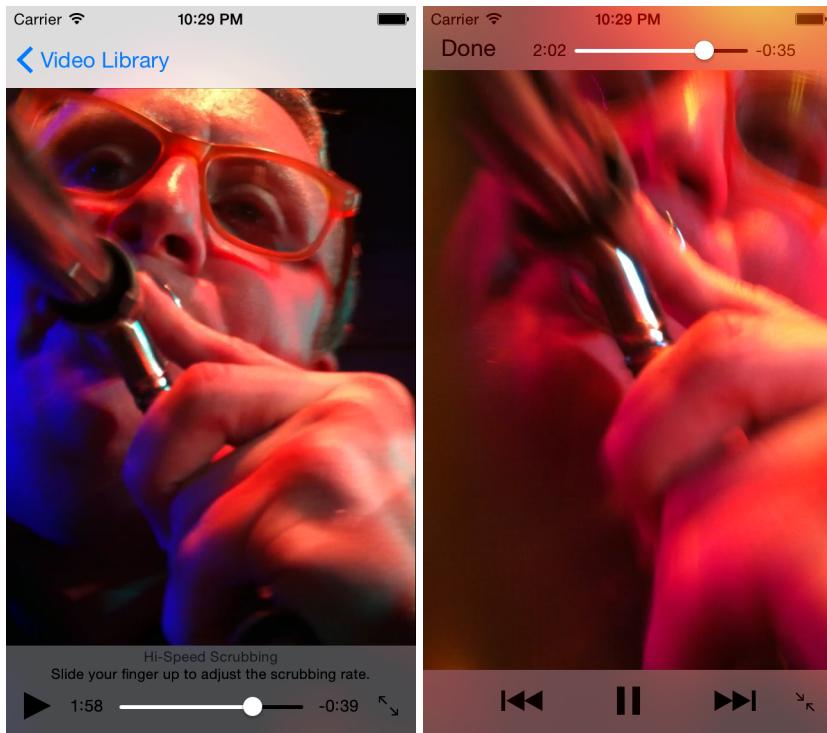
```

1  if let avpVC = self.childViewControllers.first as? AVPlayerViewController {
2      dispatch_async(dispatch_get_main_queue()) {
3          let url = NSURL(string: "/path/to/my/video")
4          avpVC.player = AVPlayer(URL: url)
5      }
6  }

```

The player-item approach is a link into the AVFoundation pipeline, and allows you to specify assets and create AVFoundation compositions for playback.

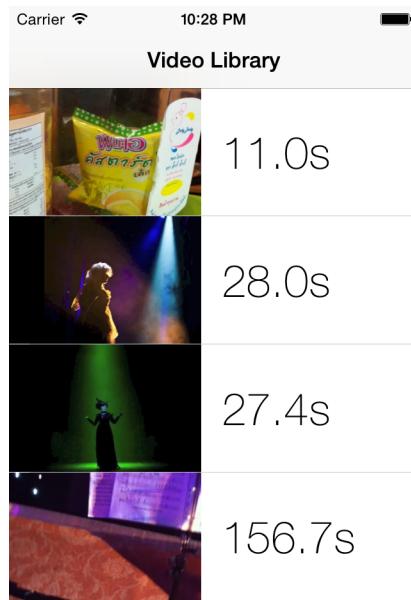
Setting the `player` property is all that you need to do to get video playback with fully adaptive and context aware playback controls:



Integration with Photos Framework

Back in chapter 20 we had a quick run-down of the new Photos framework, and how it can make access to the photo library super-easy on iOS 8. Well, despite its name, the Photos framework also includes access to all of the videos in the user's library, and with that comes the ability to use AVKit for easy playback.

If you take a look at the accompanying VCR app, you'll see it is a master- detail app, which shows a list of all videos, with thumbnails in the master, and then plays the video back when the user taps on a specific cell.



List View

Grabbing all the videos from the library using the Photos framework is simple:

```
1  videos = PHAsset.fetchAssetsWithMediaType(.Video, options: nil)
```

And getting thumbnails for the table cells is similarly easy:

```
1  self.imageManager?.requestImageForAsset(videoAsset,
2      targetSize: CGSize(width: 150, height: 150),
3      contentMode: .AspectFill, options: nil) {
4      image, info in
5      self.thumbnailImageView.image = image
6  }
```

The `PHImageManager` class has several methods for requesting video content - one of which will return a `AVPlayerItem` - exactly what's needed to create the `AVPlayer` needed by `AVKit`. It's an asynchronous API, since the `PHAsset` you're requesting the `AVPlayerItem` for might well be remote. Here, the `videoAsset` is a property of type `PHAsset`, provided by the master view controller during the segue to the detail:

```
1  imageManager?.requestPlayerItemForVideo(videoAsset, options: nil,
2      resultHandler: {
3          playerItem, info in
4              self.player = AVPlayer(playerItem: playerItem)
5      })
```

The `player` property has a `didSet` closure, which will provide the new `AVPlayer` to the appropriate child view controller:

```
1 var player: AVPlayer? {
2     didSet {
3         if let avpVC = self.childViewControllers.first as? AVPlayerViewController {
4             dispatch_async(dispatch_get_main_queue()) {
5                 avpVC.player = self.player
6             }
7         }
8     }
9 }
```

Note that since setting the `player` property on the `AVPlayerViewController` is going to be performing lots of UI operations, it's imperative that it is called on the main queue - hence the `dispatch_async()` call.

You can see this in action in the **VCR** app which accompanies the project. It has a relatively small amount of code to create a simple video browser and playback tool.

AVFoundation Pipeline

As mentioned at the top of this article, one of the great features of the new `AVPlayerViewController` is that it sits on top of, and allows access to, the underlying AVFoundation pipeline. This means that you can easily visualize the output of the complex compositing and audio-mixing effects that you apply to your input videos.

As a simple demonstration of this, the **VCR** app uses `AVQueuePlayer` to preface the playback of every video in the library with a surprisingly irritating countdown timer.

```

1 func configureView() {
2     if let videoAsset = videoAsset {
3         imageManager?.requestPlayerItemForVideo(videoAsset, options: nil,
4             completionHandler: {
5                 playerItem, info in
6                 self.player = self.createPlayerByPrefixingItem(playerItem)
7             })
8         }
9     }

```

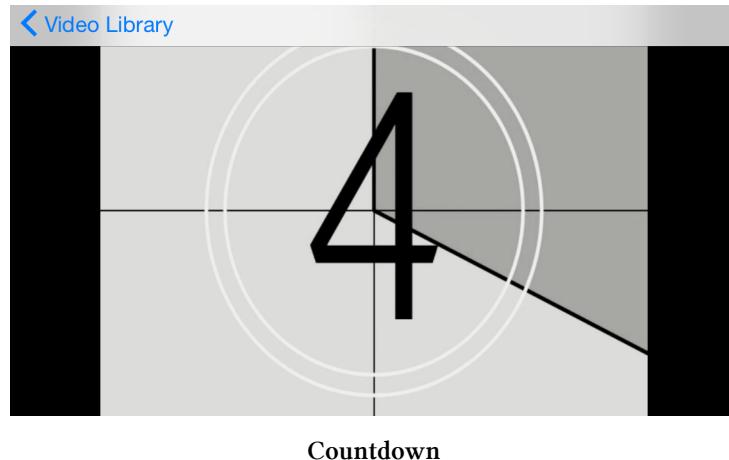
Here, the `createPlayerByPrefixingItem()` method is used to create an `AVPlayer` which includes the requested item, and also the `countdown_new.mov` file, located within the app's bundle:

```

1 private func createPlayerByPrefixingItem(playerItem: AVPlayerItem) -> AVPlayer {
2     let countdown = AVPlayerItem(URL: NSBundle.mainBundle()
3         .URLForResource("countdown_new", withExtension: "mov"))
4     return AVQueuePlayer(items: [countdown, playerItem])
5 }

```

This means that when the user selects a video for playback, they'll first have to sit through this delight:



Obviously this is somewhat of a stupid example, but it shows that you have access to the complete AVFoundation pipeline, as well as the convenience of system-provided playback UI.

Conclusion

If you're using `MPMoviePlayerViewController` then you don't need to worry about it being deprecated - it's still around. However, more than likely, it'll be a really easy operation to switch to the new `AVPlayerViewController` - much of the API is replicated (well, from `MPMoviePlayerController`).

If you've had to implement your own UI on top of an `AVPlayerLayer` then transitioning to `AVPlayerViewController` is likely to be a little more difficult. However, it does reduce the area of code that you're responsible for as iOS upgrades in future. It also ensures that a common appearance for playing videos is used across both the system apps and yours.

Combined with the ease of plugging the new Photos framework into AVKit, it's certainly worth taking a look at. If you're implementing new video playback functionality in an app then AVKit is definitely the place to start - and will serve you well 90% of the time.

As mentioned in the introduction, the accompanying app demos building a simple video browser and player for all the videos in the user's library. The source code is available on github at [github.com/ShinobiControls/iOS8-day-by-day⁷⁴](https://github.com/ShinobiControls/iOS8-day-by-day).

⁷⁴<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 27 :: Launch Images



One of the new features in iOS 8 and Xcode 6 that has maybe slipped under the radar a little is an update in the ways you can specify launch screens. Traditionally, these have been images which are displayed immediately on app launch, before the live app UI is ready to go. With the introduction of the new screen sizes of the iPhone 6 and iPhone 6 Plus, the lack of scalability of the image-based approach is starting to show. Luckily, you can now provide a storyboard or XIB file which will be used as the launch screen, across all flavors of device.

Today's article will take a look at how (and why) to upgrade an app which was built using Xcode 5 to use the new XIB-based approach. As such there is an accompanying app that demonstrates the usage - available on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁷⁵.

Scaling the existing approach

Launch screens are meant to be subtle - their sole purpose being to give the impression that the app is loading super-fast. Therefore, in Apple's Human Interface Guidelines, it suggests that launch screens should actually be identical to the first screen of the app, minus any dynamic content such as text.

In iOS 7 you would provide the content for the launch screen as images in an asset library - requiring one image per device that it would run on. Taking into account the fact that iPads can launch apps in both portrait and landscape orientations, and the existence of both retina and non-retina screens, this means that you need to create a total of 6 different launch images:

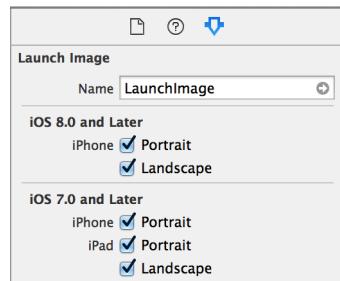


iOS 7 Launch Images

⁷⁵<https://github.com/ShinobiControls/iOS8-day-by-day>

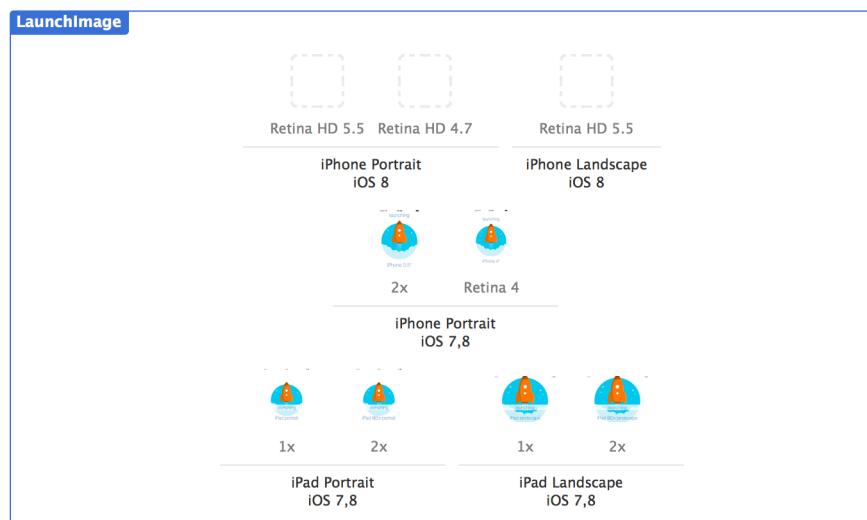
The addition of iPhone 6 and iPhone 6 Plus actually means that you have to produce three more launch images - since the iPhone 6 Plus can actually launch apps in landscape.

If you open an iOS 7 app in Xcode 6 and head on over to the **LaunchImage** asset in the asset catalog, then you might expect that there would be some new empty spaces to fill, but this isn't the case. You first need to enable the new iOS8- only types, which can be done using the attributes inspector:



Upgrading the asset library

Having checked the two boxes associated with iOS 8 iPhone orientations, the asset will be updated:



iOS 8 Launch Images



Top Tip: Until you either populate these new image cells, *or* provide a XIB/Storyboard launch screen, then your app will run in scaled mode on iPhone 6 and 6 Plus. It's therefore *really* important to add this when upgrading your old apps if you want to properly support the new devices.

Although this approach is perfectly viable, there are a couple of issues:

- It's a pain having to generate 9 different launch images for your app

- The images are obviously part of your app bundle, and therefore get distributed with your app. That's *all* of the images. One device will need at most two of the images, which means that your app includes at least seven images that it will *never use*.

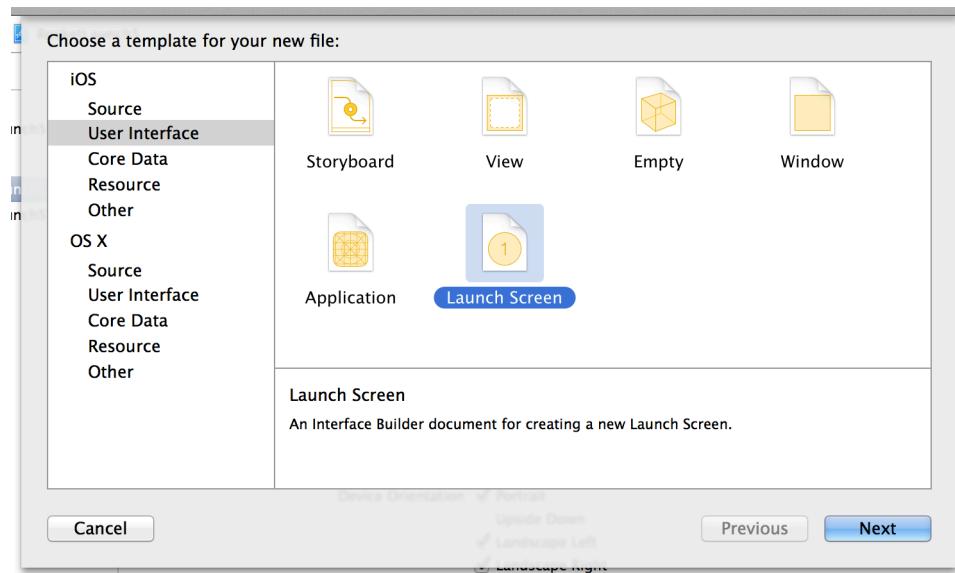
Surely there's a better way?

Creating a launch screen XIB

As Apple's Human Interface Guide suggests that your launch screen should probably look like the first screen that a user sees, then wouldn't it be great if you could create the launch screen in the same way as you create your UI? Well, with iOS 8 you can - by providing a launch screen as a XIB.

When you create a new iOS project in Xcode 6 then it'll automatically create a XIB launch screen file and set up the project correctly to use it, but upgrading an existing app is pretty easy as well.

There's a new template for a launch screen file, which you can use by creating a new file in your project:



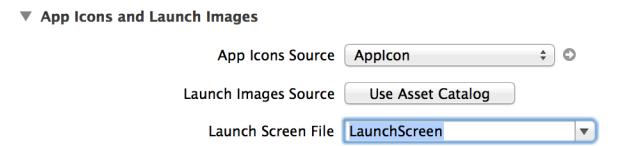
Creating Launch Screen

This will create a default screen containing the name of your project, and some copyright information:



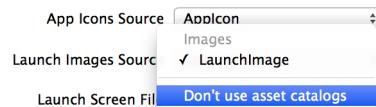
Default Launch Screen

If you run the app up now, then you'll see that your app is still using the old files - you need to tell the project what to use as a launch screen. You can do this in the **App Icons and Launch Images** panel of your app's settings.



Switching to XIB

You need to select the XIB you created from the **Launch Screen File** drop down menu, and update the **Launch Images Source** to **Don't Use Asset Catalogs**:



Don't use asset catalogs

Once you've done this you can build and run your app and notice that the new launch screen is being used in place of the old images, which you can delete.



Note: It seems that the new launch images don't always work on simulators, so be sure to check them out on a device.

Restrictions on Launch Screen XIBs

The launch screen XIBs don't support all the features you're used to from IB, so don't get too carried away. The XIB is meant to be very quick to render, and the (presumably) cached on the device - therefore there are some restrictions on what UI elements you can use.

You should construct the UI from **UIImageView** and **UILabel** objects - definitely not **UIWebView**. There can be no custom code associated with a launch XIB, so outlets and custom classes won't work.

You can (and absolutely should) use Auto Layout to layout your UI. Your XIB is also fully size-class compliant, which is how you should be adapting the launch screen to the different devices it will run on.

For more information about adopting adaptive layout in iOS 8 you should check out day 7 of iOS 8 Day-by-Day.

Conclusion

This is a welcome improvement to iOS and Xcode, making the development and release process that little bit easier, and also delivering app bundle size improvements.

However, the biggest takeaway from today's article should be that you need to either provide new launch images or make the change to XIB launch screens if you properly want to support the new iPhone 6 and 6 Plus. This isn't particularly obvious - since your app will run on the new devices, but it runs in scaled mode.

Today's app is a little bit different because it was actually created with Xcode 5 and then upgraded to include the new launch XIBs in Xcode 6. As such, the git history might be of some use in looking how this was achieved. It's available in the iOS 8 Day-by-Day git repo on github at github.com/ShinobiControls/iOS8-day-by-day⁷⁶.

⁷⁶<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 28 :: Document Picker



It has always been hard work to get content from one app to another on iOS. Although possible, it is not generally a well-understood workflow, and therefore results in users deciding that it is not possible.

This is set to change in iOS 8 with the introduction of the unified document picker view controller. This allows you to present UI from within your app which allows the user to select documents which are present within iCloud drive, on iCloud but managed by another app, and also from 3rd party document providers.

This chapter is a very brief introduction into what you can do and how to do it. It isn't an introduction to working with documents on iOS. Or an introduction to using CloudKit. Or an introduction to building your own document providers. If you want those then shout at me and I might write them. There is a basic sample app which accompanies the post - as ever available on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁷⁷.

Conceptual Overview

UIKit provides a view controller, which allows a user to select a file path. This is communicated back to you via a single delegate method. It's actually rather simple. However, understanding what's actually possible is a little more complex.

The document picker enables four possible modes:

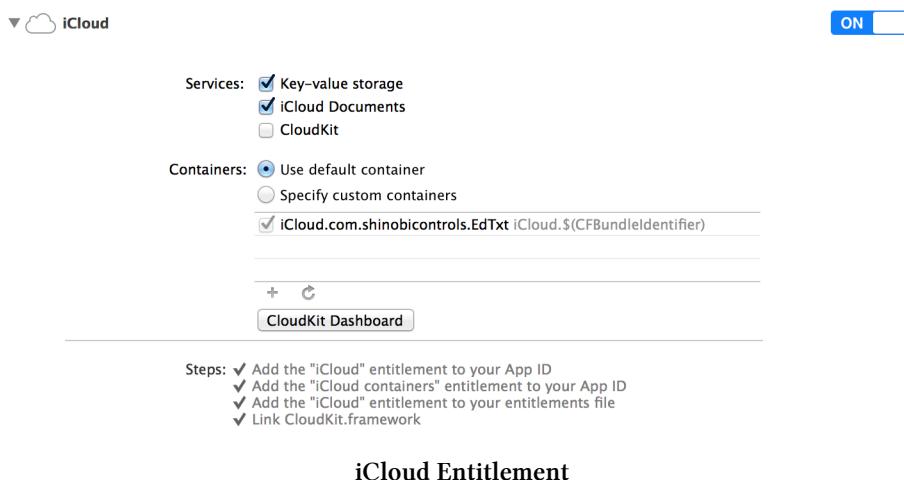
- **Import.** This creates a copy of the selected file inside your app's own sandbox. It's the equivalent to a user selecting the file in a different app and then exporting it to your app via the **Open in...** dialog.
- **Export.** This allows you to publish a copy of your local file to a different app's sandbox. The user selects where they want to export to and then the system performs the copy. You will be unable to access the URL of the resultant file.
- **Open.** Accesses a file within a different sandbox - i.e. editing it in situ. This requires a more complicated interaction pattern (using security- scoped URLs, and file coordination), since you can no longer be sure that you are the sole user of a file.
- **Move.** The equivalent to an export and open with a delete. It moves a file from your sandbox to a different place, and then provides you with a security- scoped URL to access it.

⁷⁷<https://github.com/iOS8-day-by-day>

This pretty much covers everything you might need to do in your app. It's very important that you don't expect the user to decide between these 4 options. Your workflow should be well-defined enough that these options are presented at contextually relevant times. For example, your app might not need to use the open and move modes - import and export might be good enough.



Note: Since the document picker and menu use iCloud by default, you'll need to add the iCloud entitlement to your app:



iCloud Entitlement

Document Menu

There are two new view controllers that have been added to UIKit to provide this functionality:

- `UIDocumentMenuViewController` displays the different document data sources available on the device. By default this only includes iCloud.
- `UIDocumentPickerViewController` is the actual file picker UI. In iCloud this presents the different containers, directories and files available.

As you'd expect, the menu view controller results in selecting a picker view controller.

A document menu view controller is created in two different ways; when opening or importing, you specify a list of document types you're interested in. When moving or exporting, you specify the URL of the file you'd like to work with.

For example, to import a file:

```
1 let importMenu = UIDocumentMenuViewController(documentTypes:  
2                                         [kUTTypeText as NSString], inMode: .Import)
```

This says that you're interested in text files, and that you'd like to import the result.

In order to get the resultant picker, you need to set and implement a delegate:

```
1 importMenu.delegate = self
```

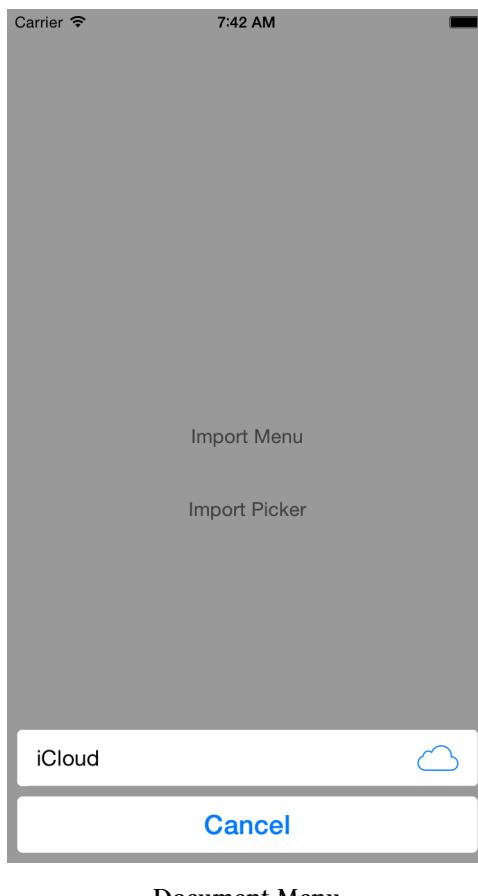
You need to adopt the `UIDocumentMenuDelegate` protocol, which has two methods, one of which covers cancellation, the other is required - `documentMenu(_:, didPickDocumentPicker:)`.

This will return the document picker object that the user selected. You can then choose to present this (see later).

Once you've set the delegate you are free to present the new view controller:

```
1 presentViewControllerAnimated(importMenu, animated: true, completion: nil)
```

This will something that resembles an action sheet:

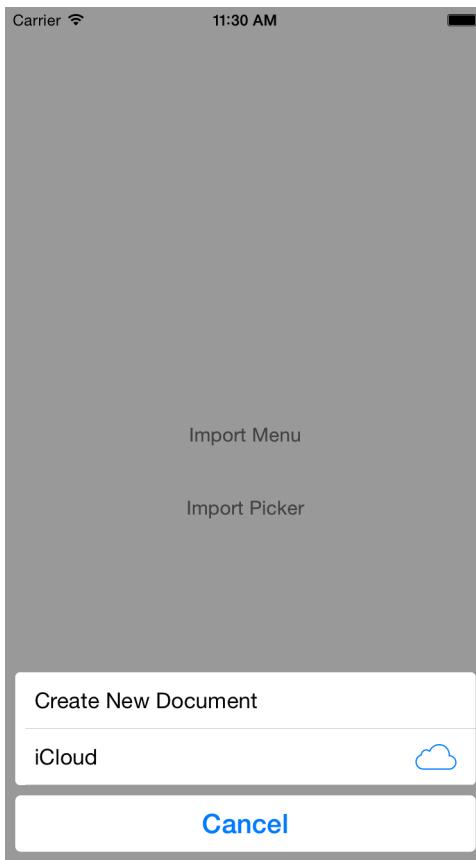


Document Menu

You are also able to add your own custom actions to this sheet via `addOptionWithTitle(_:, image:, order:, handler:)`. For example, in addition to importing a document, you might want a user to be able to create a new document:

```
1 importMenu.addOptionWithTitle("Create New Document", image: nil, order: .First,
2                               handler: { println("New Doc Requested") })
```

Which will produce the following:



Document menu with new

Document Picker

If you've implemented the document menu, then when the delegate callback gets called, it will return you a `UIDocumentPickerViewController` object. Alternatively you can create your own using either a URL (for exporting and moving) or document types (for opening or importing):

```
1 let documentPicker = UIDocumentPickerViewController(documentTypes:  
2 [kUTTypeText as NSString], inMode: .Import)
```

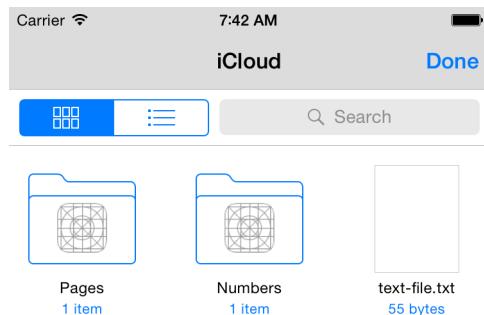
Either way, once you have a picker, you again need to set its delegate and then present it:

```
1 documentPicker.delegate = self  
2 presentViewControllerAnimated(documentPicker, animated: true, completion: nil)
```

The delegate protocol, `UIDocumentPickerDelegate` has 2 methods - one for user cancellation, one for returning the URL of the resultant file. This has different properties depending on the mode used:

```
1 func documentPicker(controller: UIDocumentPickerViewController,  
2 didPickDocumentAtURL url: NSURL) {  
3     // Do something  
4     println(url)  
5 }
```

Presenting the view controller (for the iCloud document picker) will result in a view like this:



iCloud Picker

Notice that you can see the sandboxes associated with different applications, as well as files which are present in iCloud Drive. Although you can see the different files, for import/open you'll only be able to select the files which match the types you specified in your UTType array.

Use on Simulator

It's helpful when developing to use a simulator instead of a physical device, which can cause some issues.

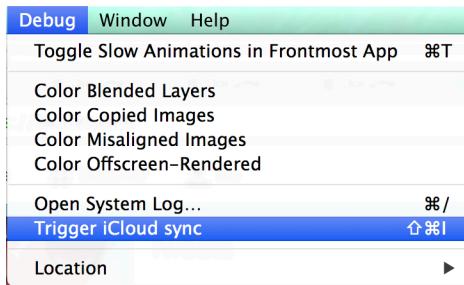
Before enabling iCloud Drive on the simulator (or device), you'll see the following when attempting to use the iCloud document picker:



Enable iCloud

You can enable iCloud on a simulator in the same way as you can on a device. However, if you want to use a test AppleID (highly recommended) then you *must* enable it for iCloud by logging in to iCloud on a device or your mac. Otherwise, logging in on a simulator will result in a password authentication error.

Once you've done this, you'll be able to open an iCloud document picker, but it will probably appear empty. You need to trigger an iCloud sync, which you can do from the **Debug** menu on the simulator:



Trigger iCloud Sync

Conclusion

If your app is based around documents, then the ability to select documents consistently across the system is huge. Even more so once you combine it with custom document providers that we can expect from people like Dropbox and Google Drive.

The API for the document picker view controller is fairly simple - certainly compared to working with `NSFileCoordinator` and security-scoped URLs. If it's relevant to you I urge you to go ahead and implement it.

The sample app for today's post is pretty simple, and is available on the iOS 8 Day-by-Day github repo at github.com/ShinobiControls/iOS8-day-by-day⁷⁸.

⁷⁸<https://github.com/iOS8-day-by-day>

Day 29 :: Safari Action Extension

8 29

This book has covered three of the new extension points so far (sharing, today & photo) and today sees the turn of the action extension.

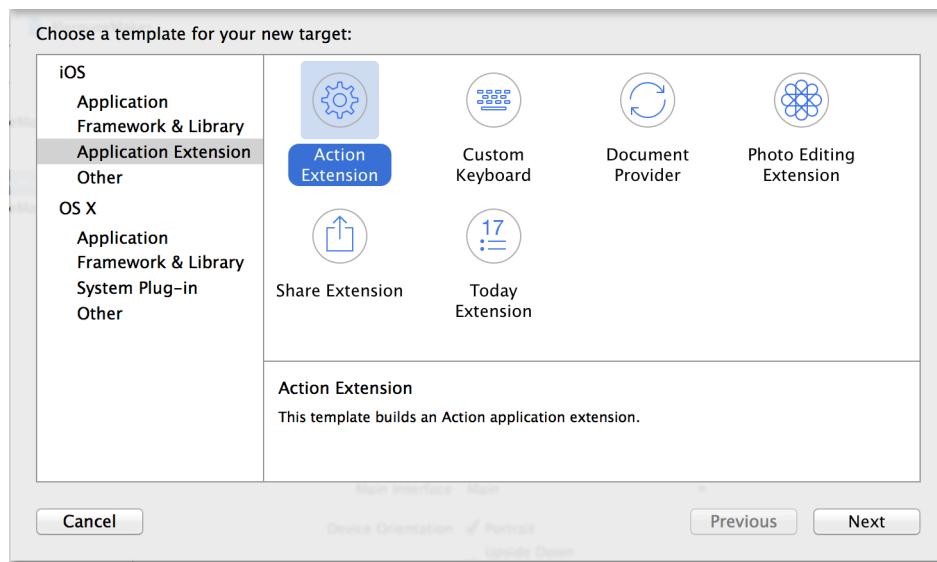
The action extension is quite similar to the sharing extension - in fact it appears on the same 'share-sheet' popover UI. The difference is very much with the intent: the sharing extension is used for exactly what its name suggests - extracting content from an app for the purpose of sharing it, either directly or via a social network.

In contrast, the action extension is intended to perform fairly light-weight transformations of the content - either providing their own UI or returning updated content to the requesting app.

Today's article will demonstrate how to create an action extension, specifically for manipulating the content of web pages. The source code for the app, **MarqueeMaker** is available on github at [github.com/ShinobiControls/iOS8-day-by-day⁷⁹](https://github.com/ShinobiControls/iOS8-day-by-day).

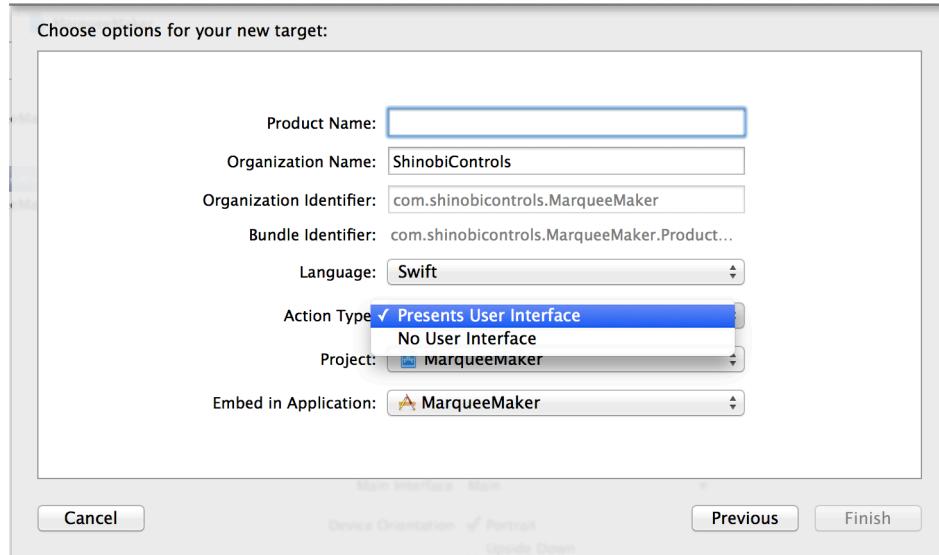
Creating an Action Extension

Creating an action extension is best performed in the same way that you create any of the extensions - via the templates in Xcode. Since all extensions require that they are bundled with a host app, you need to create an app first, and then add the extension to it:



⁷⁹<https://github.com/ShinobiControls/iOS8-day-by-day>

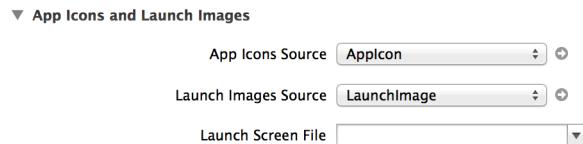
Action extensions can either have UI or not. If your extension doesn't require any input from the user in order to run, then you can choose not to have any UI. The accompanying app does have some UI - allowing the user to specify which HTML tags they'd like to manipulate:



Interface

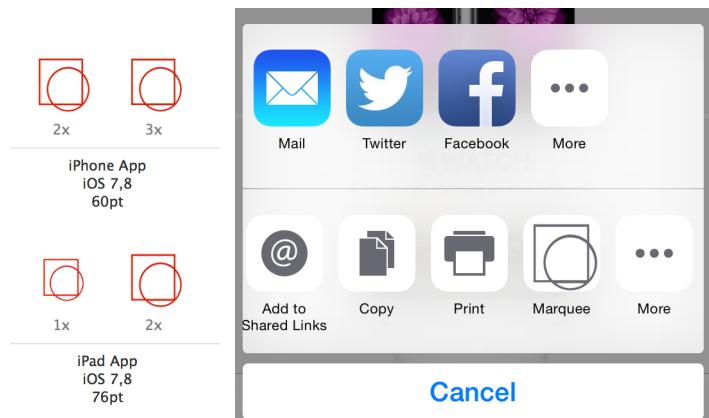
Once you've done that then you'll be presented with a template that includes a good starting point for building your extension. If you've selected to include a user interface, then you'll be provided a view controller and storyboard in which to build it. As ever, it's highly recommended to use adaptive layout to create your interface.

You'll want to create an icon image to represent your app in the share sheet. This icon is actually just the extension icon - which you can provide using an asset catalog - ensuring that it is set correctly in the project settings:



Asset Catalog

Note that the action sheet uses template images - i.e. your icon image should be composed of a single color and transparent. The non-transparent pixels will be transformed to match the other icons in the sheet:



Extracting Content from a Web Page

When you create a new extension from a template, it will appear in every single action sheet by default. This means that it won't perform any checking of the content types to determine whether or not it is appropriate. If you leave it like this and attempt to submit it for app store review then it will be rejected - you need to specify under what circumstances your extension should appear.

This is controlled via the extension's `Info.plist` file - in the `NSEExtension` dictionary. This contains a `NSEExtensionAttributes` dictionary that contains various properties which determine the behavior of the extension. One of these is `NSEExtensionActivationRule` - and when you create the extension this will be set to be a string: `TRUEPREDICATE`. This means that your extension will always appear.

You have a couple of options for this property - the simplest being turning it into a dictionary (the alternative is to write a custom predicate), with keys which specify different media types. Since `MarqueeMaker` operates on web pages, only `NSEExtensionActivationSupportsWebPageWithMaxCount` is used, with a value of 1 - since the extension only works for a single web page at a time:

▼ NSEExtension	Dictionary	(3 items)
▼ NSEExtensionAttributes	Dictionary	(2 items)
▼ NSEExtensionActivationRule	Dictionary	(1 item)
NSEExtensionActivationSupportsWebPageWithMaxCount	Number	1
Activation Rule		

Now, when you load up a web page in mobile Safari, and pull up the action sheet then you'll see your extension appearing.

Running a JS Preprocessor

Since you're using a web page here, extensions have a cool feature which allows you to specify some JavaScript that Safari will run before invoking your extension. From this you can provide content extracted from the web page to the extension itself - which is pretty cool!

The javascript that you want running must conform to some requirements:

- There must be a global object called `ExtensionPreprocessingJS`
- This object must have a `run` method, which takes a single argument

The following JS code demonstrates how you can create a simple javascript object which will provide the URL of the current page to the extension:

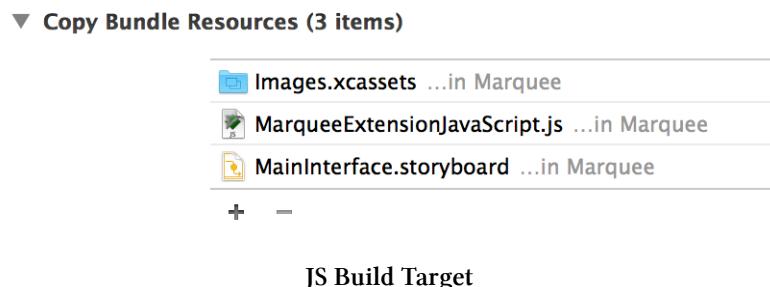
```

1 var MarqueeMakerExtension = function() {};
2
3 MarqueeMakerExtension.prototype = {
4   run: function(arguments) {
5     arguments.completionFunction({ "baseURI" : document.baseURI });
6   }
7 }
8
9 var ExtensionPreprocessingJS = new MarqueeMakerExtension;

```

Note that you provide content over the JS->UIKit bridge by calling the `completionFunction` method on the argument to the `run` function, providing a dictionary of the things you want to access from within your Swift (or objC) code.

This JS code should be placed in its own file, which is added to the extension's target:



Accessing the JS-provided content from Swift

To tell your extension that you've provided a JS preprocessor file, you need to head back to the `Info.plist` and add a new key to the `NSExtensionAttributes` dictionary. The value associated with the `NSExtensionJavaScriptPreprocessingFile` key should be set to the name of the JS file **without** the `.js` extension:

▼ NSExtensionAttributes	Dictionary	(2 items)
▼ NSExtensionActivationRule	Dictionary	(1 item)
NSExtensionActivationSupportsWebPage...	Number	1
NSExtensionJavaScriptPreprocessingFile	String	MarqueeExtensionJavaScript
Preprocessing		

To access the bundled up content, you use exactly the same approach you did with the sharing extension - via `NSItemProvider`. The `NSExtensionContext` has a collection of `NSExtensionItem` objects via its `inputItems` property, which each have `NSItemProvider` objects on the `attachments` property:

```
1 override func viewDidLoad() {
2     super.viewDidLoad()
3
4     for item: AnyObject in self.extensionContext!.inputItems {
5         let inputItem = item as NSExtensionItem
6         for provider: AnyObject in inputItem.attachments! {
7             let itemProvider = provider as NSItemProvider
8             if itemProvider
9                 .hasItemConformingToTypeIdentifier(kUTTypePropertyList as NSString) {
10                 // You HAVE to call loadItemForTypeIdentifier to get the JS injected
11                 itemProvider.loadItemForTypeIdentifier(kUTTypePropertyList as NSString,
12                                                 options: nil, completionHandler: {
13                     (list, error) in
14                     if let results = list as? NSDictionary {
15                         NSOperationQueue.mainQueue().addOperationWithBlock {
16                             // We don't actually care about this...
17                             println(results)
18                         }
19                     }
20                 })
21             }
22         }
23     }
24 }
```

You can interrogate each item provider to see whether it contains items of a particular type - in this case the content provided from the JS preprocessor will be of type `kUTTypePropertyList`. You can load it using the `loadItemForTypeIdentifier(_:, options: _, completionHandler:)` method.



NOTE: It's actually very important to call this `loadItemForTypeIdentifier` method. If you don't then the javascript preprocessor won't be executed. Here, in `MarqueeMaker` you're not actually interested in the results, but it's necessary to call it to ensure that the javascript is invoked, providing the functionality for the next section.

When run, the above code will result in logged output:

```
1 2014-09-23 08:40:55 MobileSafari[13553:212239] Unknown activity items supplied: (
2     "http://www.apple.com/",
3     "<WBUPrintPageRenderer: 0x7fa893faccf0>",
4     "<UIPrintInfo: 0x7fa896063790>"
5 )
```

You can run any type of javascript you like here - so you can extract all the images from a web page, or query the DOM in any clever ways you can think of. The possibilities are endless.

It's worth noting that debugging an extension which includes a JS preprocessor can be quite hard work. However, it is possible to get a handle on what your JS is doing by using the iOS safari debugger - which is part of the desktop version of Safari - in the hidden **Developer** menu.

Interacting with JavaScript

The javascript preprocessor is available to both action and sharing extensions - providing the content extraction functionality for web pages.

Action extensions are also able to run code *after* the extension has completed. This enables you to update the displayed web page with content you've generated using your extension.

The hook for this is pretty simple - in addition to the `run` method on the JS preprocessor object, there is also a `finalize` method, which again takes a single argument.

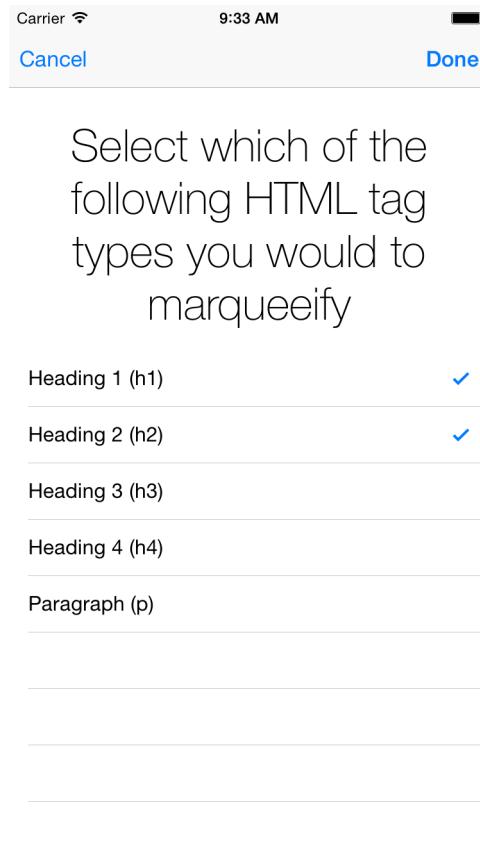
In MarqueeMaker, the following shows the completed preprocessor object:

```
1 MarqueeMakerExtension.prototype = {
2     run: function(arguments) {
3         arguments.completionFunction({ "baseURI" : document.baseURI });
4     },
5
6     finalize: function(arguments) {
7         marqueeWrapper(arguments["marqueeTagName"]);
8     }
9 }
```

Here, `marqueeWrapper` is a method which will take an array of HTML tag names, and wrap their content with `<marquee>` tags. For example, if the tag array is `["h1"]` then `<h1>Hello</h1>` would be replaced with `<h1><marquee>Hello</marquee></h1>`. If you're interested in the JS code that does this then check out the accompanying app.

The argument provided to the `finalize` function is just a JS dictionary, which you can generate in your Swift code.

The UI for the MarqueeMaker app allows a user to select which of the HTML tags they'd like to add the <marquee> tag to:



Tag Selection

When the user hits the **Done** button, the following method is executed:

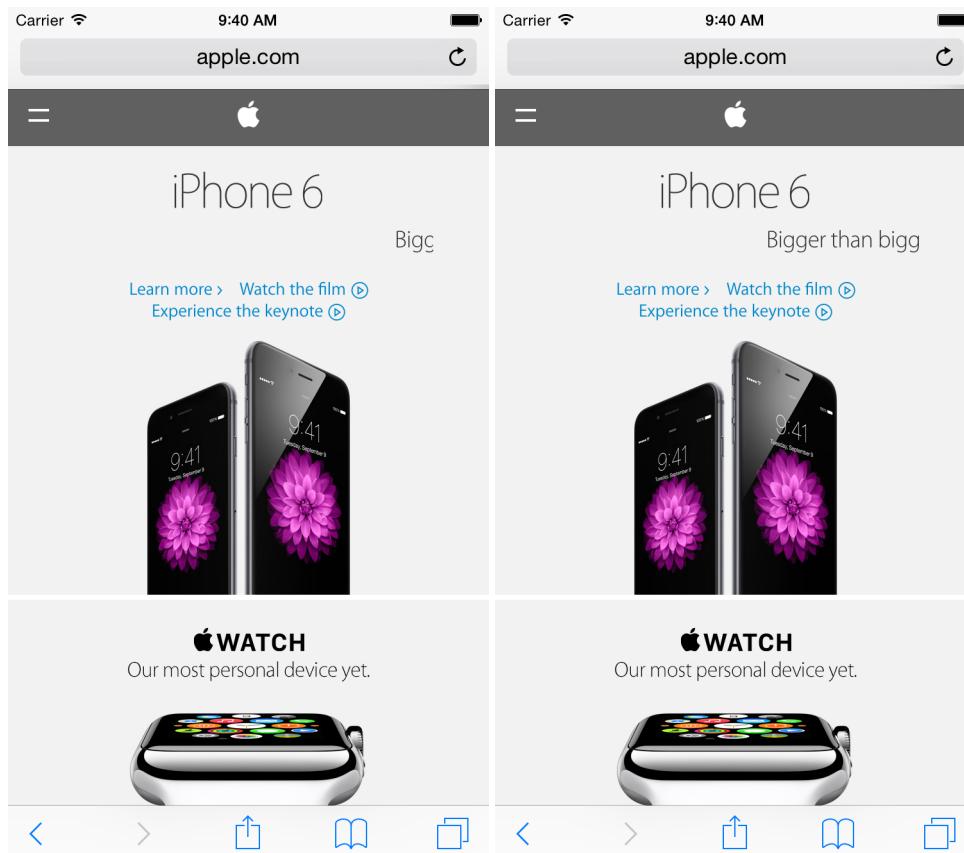
```
1 @IBAction func done() {
2     // Find out which tags need marqueeifying
3     let marqueeTagNames = tagList.filter{ $0.status }.map{ $0.tag }
4
5     // Parcel them up in an NSExtensionItem
6     let extensionItem = NSExtensionItem()
7     let jsDict = [ NSExtensionJavaScriptFinalizeArgumentKey :
8                     [ "marqueeTagNames" : marqueeTagNames ] ]
9     extensionItem.attachments = [
10         NSItemProvider(item: jsDict, typeIdentifier: kUTTypePropertyList as NSString)
11     ]
12
13     // Send them back to the javascript processor
14     self.extensionContext!.completeRequestReturningItems([extensionItem],
```

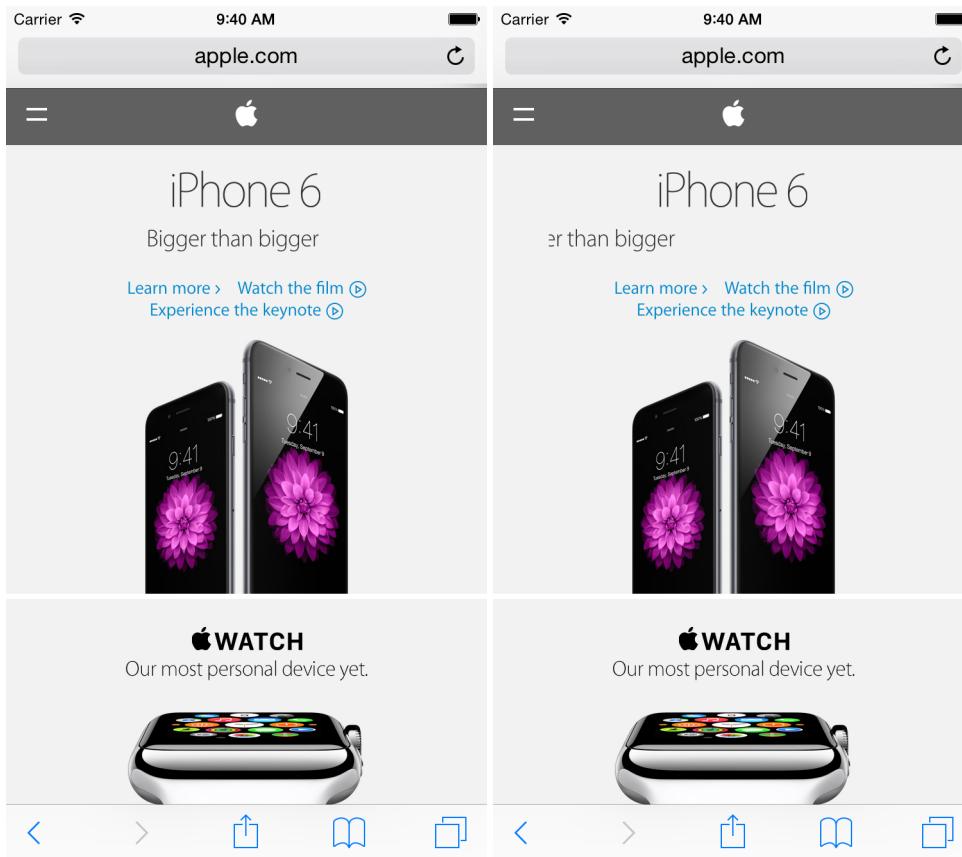
```
15                                         completionHandler: nil)  
16 }
```

This method performs the following:

1. `marqueeTagNames` determines which of the HTML tags the user selected from the table view, and is an array of strings - e.g. `["h1", "h2"]`.
2. Providing data back to the JS preprocessor is done via `NSExtensionItem` objects. You need a dictionary with a key of `NSExtensionJavaScriptFinalizeArgumentKey` and the value of the data you want to send back. The value should be something that can be easily converted to JSON (i.e. array/dictionary).
3. You then provide this dictionary as an attachment to the `NSExtensionItem`, with type `kUTTypePropertyList` - the same type you used to extract data *sent from* the preprocessor.
4. Finally, you call `completeRequestReturningItems(_ completionHandler:)` on the extension context to send the extension item back to the preprocessor.

In the case of **MarqueeMaker** the UI of the extension is used to determine *which* of the HTML tags should be wrapped in `<marquee>` tags. The javascript preprocessor actually performs the DOM manipulation in order to effect the change.





Conclusion

Action extensions are pretty cool - you can do some interesting things with them, but they're aren't really that significantly different to sharing extensions. The really interesting part of today's post is the ability to write javascript which will interact with the web page displayed - both pre and post extension invocation.

Admittedly, this marquee extension is nothing more than an irritation, but the possibilities here are huge - in the WWDC keynote, a translation engine was demoed using action extensions, and already we're seeing innovative uses by password managers.

As ever, the code for today's post is available to clone, download or fork on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁸⁰.

⁸⁰<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 30 :: App Previews

8 30

The screenshots that adorn your app's page in the App Store have always been incredibly important in advertising your app - giving a flavor of what a user can expect and attempting to get them to push the all-important **Buy** button. Back when iOS 7 was first announced I wrote a blog post about how given the new, flatter UI design, screenshots just weren't going to cut it any more ([iOS7: What does it mean for developers?](#)⁸¹), and it seems that Apple agree. Starting with iOS 8, you can now upload a short video to demo your app, to be displayed alongside the screenshots in the App Store.

In today's article you'll learn how to create these videos, and take a look at some of the recommendations from Apple regarding their content. Since it's still early days in the video approval process, there is little empirical evidence as to how strictly the rules are applied - with time the community will learn both what works, and what Apple is happy to approve.

In a change to the standard format, today's post doesn't have a sample project. It does describe how to get started with recording video from your device - with a sample demoing the **MarqueeMaker** action extension from day 29.

Recording a Video

In the past, when wanting to record video from a device you'd have to resort to using AirPlay and a 3rd-party app on OSX. AirPlay has low resolution, and a high compression ratio - allowing it to work over wireless networks. This means that a recording from a device looks shabby at best.

New to iOS 8 and OSX Yosemite, it's now possible to record directly from the device onto your computer. In fact, when you plug in an iOS 8 device (via a lightning cable) it appears as a new camera.

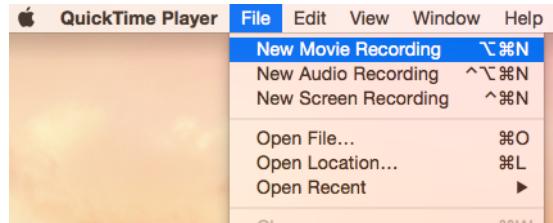
To record a video, you need to start out by opening QuickTime. Note, that by default, QuickTime doesn't open a window when you start it - but you can see that it has started in the dock:



QuickTime Dock

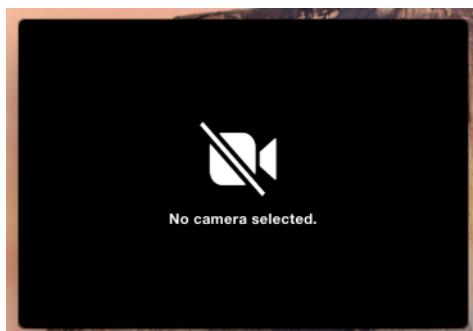
⁸¹<http://www.shinobicontrols.com/blog/posts/2013/07/01/ios7-what-does-it-mean-for-developers>

You start a new recording via the **File** menu:



New Recording

This will open a new window and tell you that you need to select a camera:



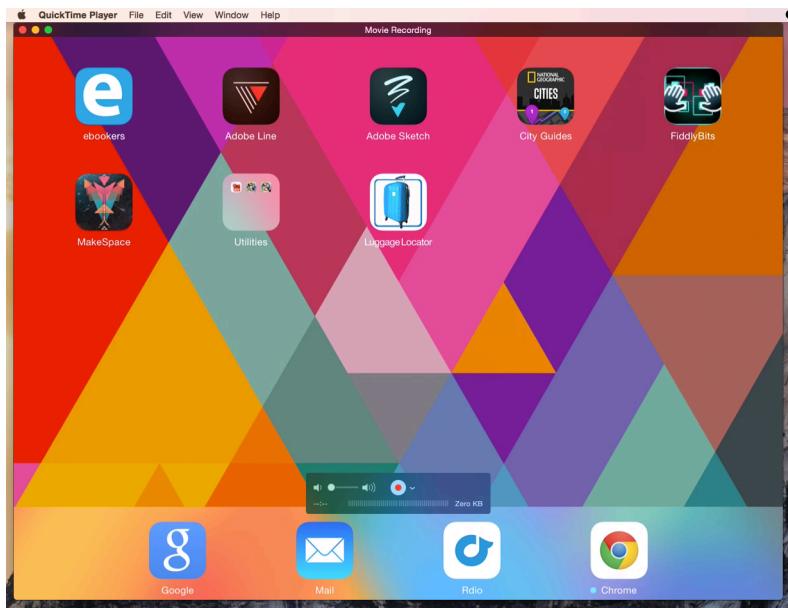
No Camera

If your iOS 8 device is connected via a lightning cable then you'll be able to select it as a camera from the source menu:



Select Camera

QuickTime will then resize to mirror your device's current screen, at which point you can start recording. Notice also that the status bar has changed from the regular one to the standard one - displaying the "Apple Time" of 09:41, removing your service provider and filling your battery:



Ready to Record

This will allow you to create a simple video, that at the very least will allow you to record the source material to cut together to build your complete demo video.

Video Considerations

Although this sorts the technicalities of actually getting video from a device into a recording, it's only the tip of the iceberg that is creating a great video.

Apple has created a page full of info about how to create [App Previews⁸²](#), which goes some way to helping work out what to do. Some of the most important points are summarized below:

- Videos should be 15 to 30 seconds in length
- The video will appear before the screenshots
- iTunes Connect allows you to choose a poster image. This is the image that is displayed before the video starts playing. Note that you can't change this without uploading a new binary to iTunes Connect, so choose wisely. Remember that this poster frame is taking the place of the first screenshot - so make sure it is as powerful as the screenshot it is replacing.
- You can submit device-specific videos. This means iPad, 4-inch iPhone, 4.7- inch iPhone and 5.5-inch iPhone can all have their own videos.
- Apple is keen on well-produced videos. this involves careful planning, storyboarding to create a well-formed story. It should demo the top 5 features of your app.
- Prepare appropriate demo data.

⁸²<https://developer.apple.com/app-store/app-previews/>

- **Previews (like screenshots) are for all audiences.** This means the content needs to be suitable for the 4+ age-group.
- **Videos are not localized.** This is good in one respect - you don't need to create hundreds of versions of the same video. However, if you're marketing in multiple markets, you need to ensure that your video is understandable irrespective of the user's native language. You also can't include any pricing details, since these will vary by territory.
- **Soundtracks are encouraged.** Make sure you have the rights for everything you want to use in your video.
- **In-app purchases must be disclosed.** If your demo shows features only available via in-app purchase, you have to notify users of this. In a language- agnostic manner. Good luck with that.
- **Videos should only feature on-device footage.** You should not have video of somebody using the app, or anything similar. In this respect it should be very much like the screenshots you're used to producing.

Apple has helpfully produced a PDF guide of how to create app previews using Final Cut Pro⁸³, which gives you some indication of the level of quality that Apple is expecting.

Conclusion

There are a couple of different things in today's article - the ability to record video from a device, and app previews. I think that the ability to record video is super-useful - and is something that was never really very easy to achieve in the past.

From a user's perspective, app previews are a definite win - for exactly the reasons I outlined in the aforementioned article last year. Getting a 'feel' for an app via static screenshots has become increasingly difficult, so app preview videos will make life easier.

However, for developers, app previews represent yet another thing that needs to be done as part of the release process. Due to their nature, they are likely to benefit large shops - who have the skills to create this kind of creative. I think that small developer-only teams will find creating compelling videos quite hard work. Remember, huge amounts of money are spent on creating TV advertising, which are also 30s videos. Some of the restrictions that Apple have laid out will help level the playing field between those who have money to burn and those who don't.

⁸³<https://developer.apple.com/app-store/app-previews/final-cut-pro/Creating-App-Previews-with-Final-Cut-Pro-X.pdf>

Day 31 :: Using Touch ID to Secure the Keychain

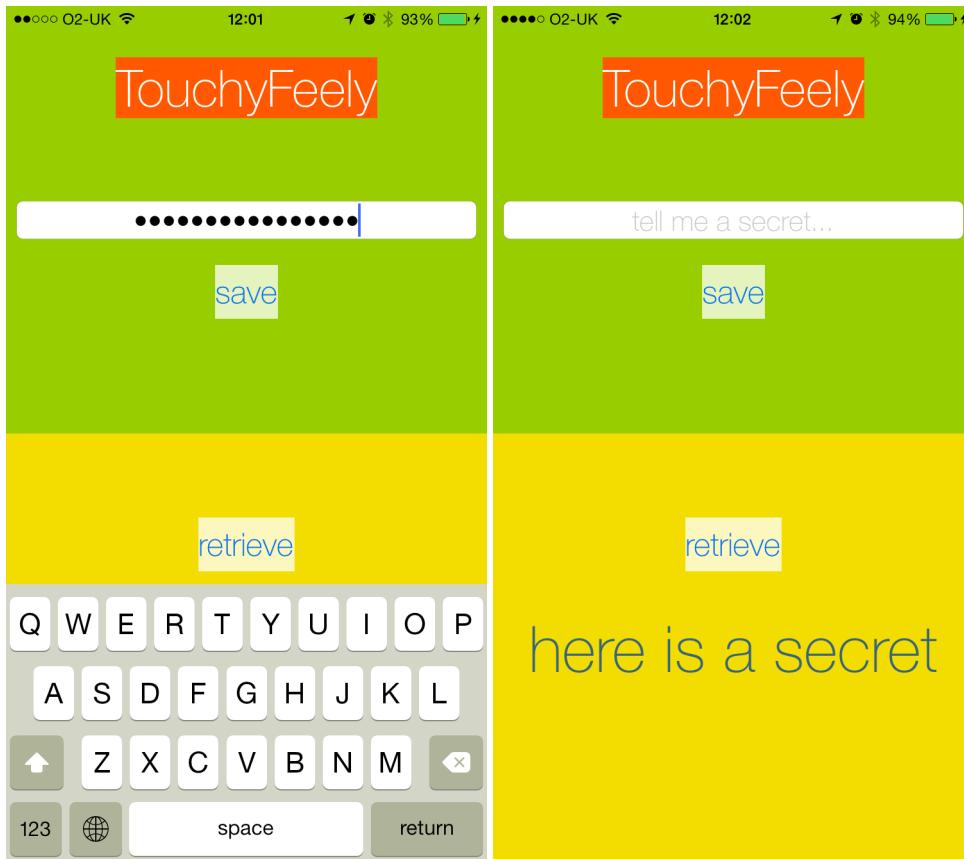


The iPhone 5s introduced the world to Touch ID - the ability to authenticate with your device using just a finger print. In iOS 7 use of this was limited to unlocking the device, and purchases against your Apple ID. Developers were desperate to get hold of an API to allow them to use this secure and convenient method of authentication in their own apps. In iOS 8 this is now possible.

There are two classes of problem that you can solve using Touch ID in iOS 8 - authentication to access content of the Keychain, and confirming user presence for your own app's usage. Today's article takes a look at how to use Touch ID in tandem with the Keychain. If you want to know more about using Touch ID to perform user authentication then take a look at LocalAuthentication in the documentation.

The sample app which accompanies today's post is a very simple app which uses the Keychain to save a secret and requires Touch ID authentication in order to retrieve it. The code is available on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁸⁴](https://github.com/ShinobiControls/iOS8-day-by-day).

⁸⁴<https://github.com/ShinobiControls/iOS8-day-by-day>



Secure Enclave

Before getting coding, it's worth understanding a little background regarding the way the Keychain works on iOS devices. Devices with Touch ID have a hardware 'device' called a secure enclave. This is responsible for the cryptographic operations on the device including encryption of the Keychain, and managing the Touch ID fingerprints. This means that the OS never has access to the fingerprints themselves - but instead defers authentication to the secure enclave.

When you store something in the Keychain, it gets encrypted. When you request it back from the Keychain, then the encrypted data gets passed to the secure enclave which performs the decryption. Since the secure enclave is also responsible for authenticating Touch ID, if the Keychain item requires Touch ID authentication then it will only decrypt the token if the Touch ID is successful.

Although it's not entirely necessary to know this, it's important to realize that the APIs made available to developers allowing them to leverage Touch ID don't provide access to the finger prints, or indeed the UI which accompanies the request for authentication. In fact, in the case of securing the Keychain, if you already have experience of using the Keychain on iOS, then there is very little to learn.

Access Control Lists

When you create items on the Keychain you set various attributes, which specify both what the item represents, and the access level that it should have. For example, the following specifies that you are creating a password entry, for the given service and user name:

```
1 private func save(token: String) {
2     if let data = token.dataUsingEncoding(NSUTF8StringEncoding,
3                                     allowLossyConversion: false) {
4
5         // Rather than update, just delete and continue
6         delete()
7
8     let keyChainQuery = [
9         kSecClass          : kSecClassGenericPassword,
10        kSecAttrService   : serviceIdentifier,
11        kSecAttrAccount   : accountName,
12        kSecValueData      : data
13    ]
14
15    SecItemAdd(keyChainQuery, nil)
16 }
17 }
```

This actually works out quite nicely in Swift - the dictionary is bridged to its CoreFoundation counterpart, as are the strings and NSData object.

iOS 8 adds the concept of Access Control Lists (ACL) to the Keychain. These are used to specify the *accessibility* and *authentication* associated with the keychain item to which they are applied:

- **Accessibility** represents when a particular item should be readable - for example when the device is unlocked, or limiting it to this device only (i.e. not shared via iCloud Keychain). This behaves in exactly the same way that the kSecAttrAccessible attribute does.
- **Authentication** is a new concept, and is represented by a policy. The policy specifies what authentication requirements have to be satisfied before the system will decrypt the Keychain item.

Currently, the only authentication policy is “user presence” - i.e. requiring confirmation that the user is actually there. This will behave differently according to the device configuration:

- **Device without passcode set.** The request to retrieve the keychain item will always fail - since you can never guarantee that the correct user is present.

- **Device with pass code set.** The user will be presented with UI to enter the device passcode. If they enter it correctly then the item will be decrypted.
- **Device with Touch ID.** User is prompted to unlock the item using Touch ID. They can use their passcode as a fall-back.

This behavior sounds great - but how can you use it? Read on...

Implementation

There has been a lot of theory today, and not much doing. It turns out that the implementation is really quite simple. There is an additional attribute to add to your Keychain - in the form of `kSecAttrAccessControl`. The value associated with this is of type `SecAccessControl`, and can be created using the `SecAccessControlCreateWithFlags()` function.

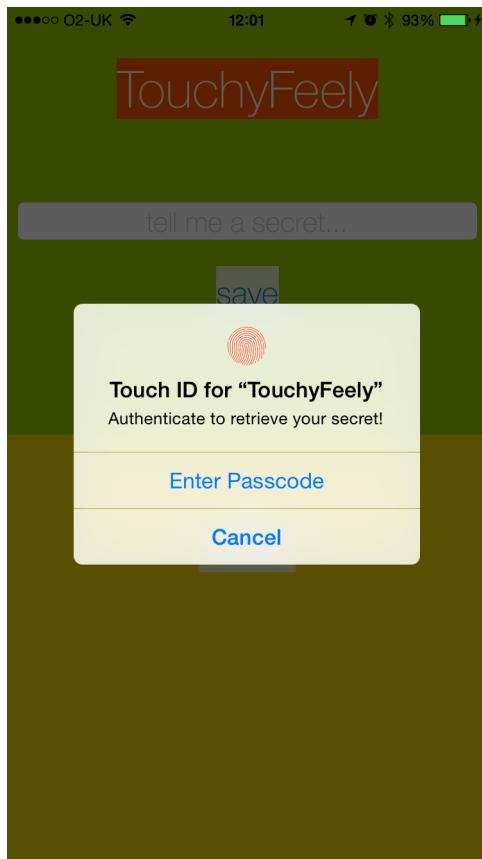
The following demonstrates how to update the previous `save()` method to enforce the **user presence** authentication policy:

```
1 private func save(token: String) {
2     if let data = token.dataUsingEncoding(NSUTF8StringEncoding,
3                             allowLossyConversion: false) {
4
5         // Rather than update, just delete and continue
6         delete()
7
8         // Create the appropriate access controls
9         let accessControl = SecAccessControlCreateWithFlags(kCFAlocatorDefault,
10                 kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
11                 .UserPresence, nil)
12
13        let keyChainQuery = [
14            kSecClass          : kSecClassGenericPassword,
15            kSecAttrService    : serviceIdentifier,
16            kSecAttrAccount    : accountName,
17            kSecValueData      : data,
18            kSecAttrAccessControl : accessControl.takeUnretainedValue()
19        ]
20
21        SecItemAdd(keyChainQuery, nil)
22    }
23 }
```



Note: The access control object specifies both the accessibility and the authentication policy (one helpfully using a string, the other an enum). This is then provided to the key chain query with the `kSecAttrAccessControl` key. Due to the CoreFoundation bridging, the `SecAccessControlCreateWithFlags()` actually has a return type `Unmanaged<SecAccessControl>`, so here you need to call `takeUnretainedValue()`.

Now, when you request secret retrieval on a Touch ID device, you'll be presented with the following UI:

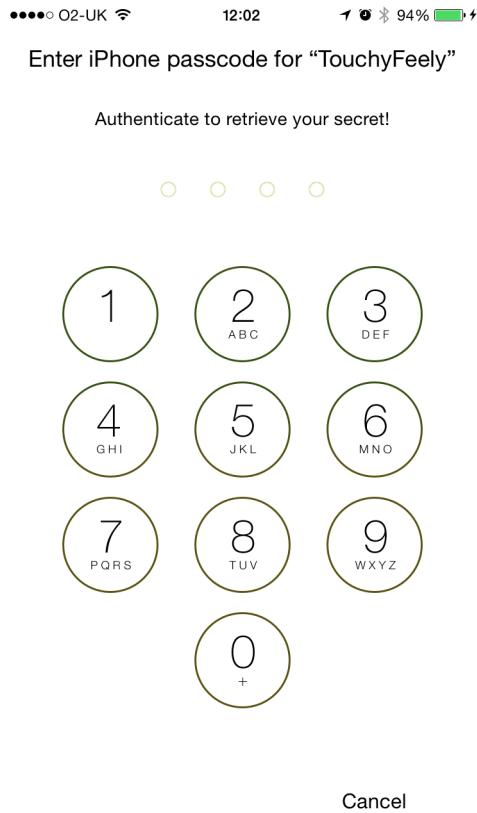


Touch ID UI

You can specify the sub-heading within this dialog using the `kSecUseOperationPrompt` key in the retrieval query:

```
1 private func load() -> String? {
2     let keyChainQuery = [
3         kSecClass           : kSecClassGenericPassword,
4         kSecAttrService      : serviceIdentifier,
5         kSecAttrAccount       : accountName,
6         kSecReturnData        : true,
7         kSecMatchLimit        : kSecMatchLimitOne,
8         kSecUseOperationPrompt : "Authenticate to retrieve your secret!"
9     ]
10
11    var extractedData: Unmanaged<AnyObject>? = nil
12
13    let status = SecItemCopyMatching(keyChainQuery, &extractedData)
14
15    let opaque = extractedData?.toOpaque()
16    var contentsOfKeychain: String?
17
18    if let opaque = opaque {
19        let retrievedData = Unmanaged<NSData>.fromOpaque(opaque)
20                           .takeUnretainedValue()
21        // Convert the data retrieved from the keychain into a string
22        contentsOfKeychain = NSString(data: retrievedData,
23                                       encoding: NSUTF8StringEncoding)
24    } else {
25        println("Nothing was retrieved from the keychain. Status code \(status)")
26    }
27
28    return contentsOfKeychain
29 }
```

The fallback for a TouchID device when using the Keychain uses the passcode, and has the same appearance as when used on a device without Touch ID:



If the device doesn't have passcode lock enabled then there will be an error in attempting to retrieve this keychain item.

Conclusion

Touch ID is really cool - having only owned a phone that has it for a week I have no idea how I used to cope without it. The introduction of APIs that give access to Touch ID for 3rd party app developers is great. The Keychain API is particularly easy to use - it was far more fiddly working out how to use the Keychain in Swift than it was to add Touch ID support.

The code for today's app, **TouchyFeely** demos how to use the Keychain in this way, and is available on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁸⁵](https://github.com/ShinobiControls/iOS8-day-by-day).

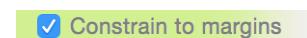
If you are interested in establishing user presence when you don't require the Keychain then you can use the LocalAuthentication APIs.

⁸⁵<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 32 :: Layout Margins



If you've been using the Pin menu in Xcode 6 then you might have noticed the addition of a slightly mystical option - **Constrain to margins**:



Constrain to Margins

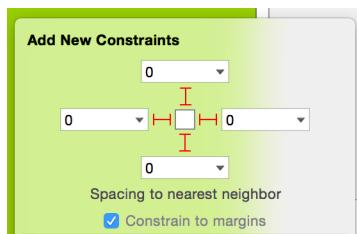
If you've left this option ticked then you'll have an idea what it does, although at first the behavior can seem a little confusing.

Today's post digs down into what layout margins are, what they represent to the Auto Layout system, how to use them in IB and how to use them in code. There is an accompanying project which demos these concepts - but be warned - the majority of it uses the storyboard. You can grab it on the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁸⁶.

Layout Margins in Interface Builder

Layout margins represent padding around the interior of a `UIView` that the layout system can use when laying out subviews - to ensure that a gap is left between the edge of a view and a subview. In this respect it is very much like the padding property associated with blocks in CSS.

You've already seen one of the places that margins appear in interface builder - in the pin menu. When creating new auto layout constraints, you can choose them to be relative to the margins of a view instead of the bounds:



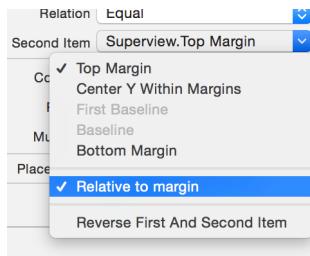
Adding Margin Constraints

By default, a `UIView` has layout margins of 8 points on each side - but you can change this using the `layoutMargins` property. This takes a `UIEdgeInsets` object - which just comprises 4 `CGFloat` values - one for each of top, bottom, left and right.

These margins are only used if you're using Auto Layout - and represent an additional attribute you can specify when creating an `NSLayoutConstraint`. In IB, when you look at the size inspector

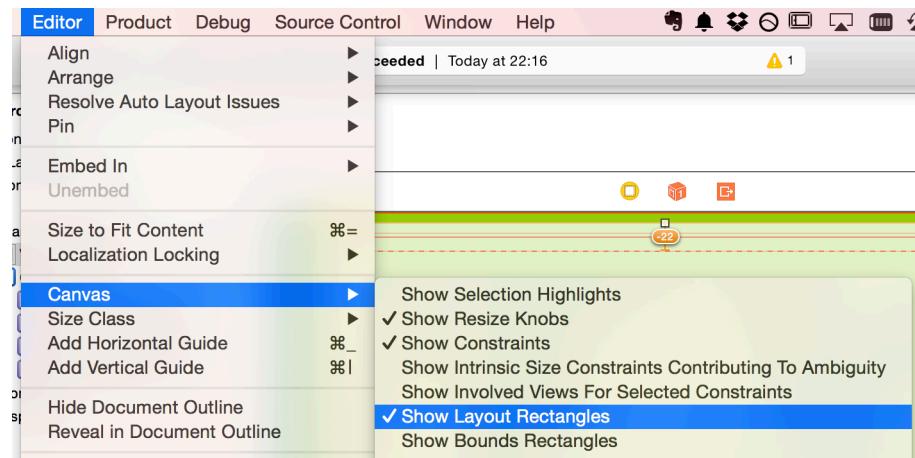
⁸⁶<https://github.com/ShinobiControls/iOS8-day-by-day>

for a constraint then you can see that it's possible to choose whether a constraint should be margin relative or not:



Size Inspector

Changing this is likely to update the appearance of a layout in IB - but it is really helpful to be able to see the layout margins whilst building your layout. You can get IB to display the margins with **Editor > Canvas > Show Layout Rectangles**:



Show Layout Rectangles

Layout Margins in Code

When you select that an auto layout constraint should be margin relative in IB, then you are actually updating the layout attributes associated with that constraint. An auto layout constraint describes a relationship between two view objects, and each of these has an associated attribute that specifies what aspect of the view the layout engine should be using during the layout process. For example, there are attributes for left edge, right edge etc. In order to support layout margins, the `NSLayoutAttribute` enum has been updated in iOS 8 to include the following margin-related cases:

```
1 enum NSLayoutAttribute : Int {  
2     ...  
3     case LeftMargin  
4     case RightMargin  
5     case TopMargin  
6     case BottomMargin  
7     case LeadingMargin  
8     case TrailingMargin  
9     case CenterXWithinMargins  
10    case CenterYWithinMargins  
11    ...  
12 }
```

You can alter the layout margins themselves using the `layoutMargins` property - which just takes a `UIEdgeInsets` object.

Since IB doesn't currently have the UI to alter the layout margins, you could build your own `UIView` subclass which allows the user to alter them in IB:

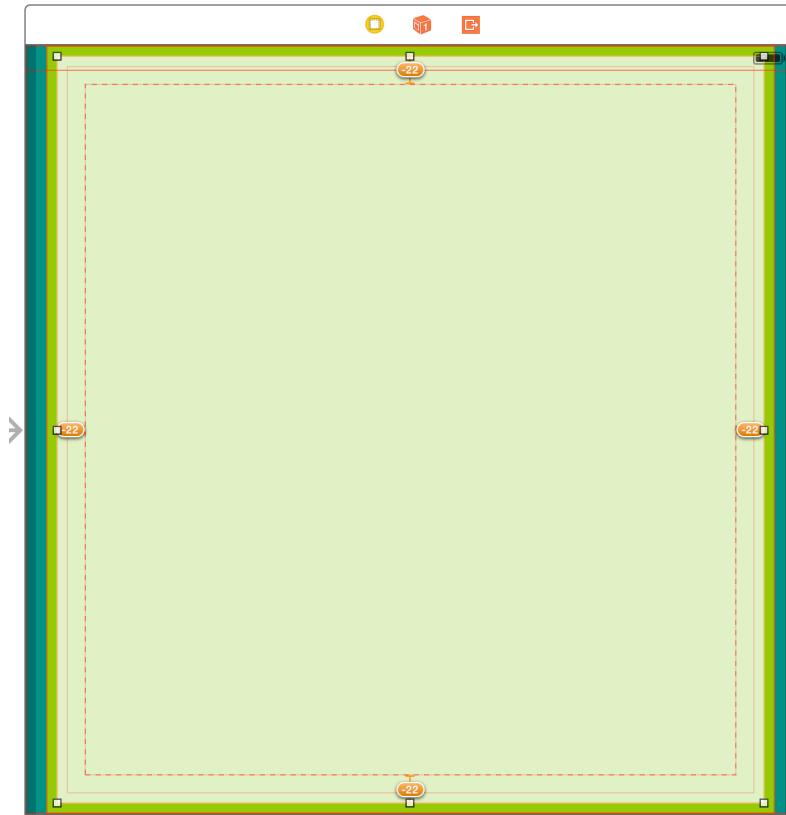
```
1 @IBDesignable  
2 class MarginConfigView: UIView {  
3     @IBInspectable  
4     var margin: CGFloat = 16 {  
5         didSet {  
6             self.layoutMargins = UIEdgeInsets(top: margin, left: margin,  
7                                              bottom: margin, right: margin)  
8         }  
9     }  
10 }
```

The above class has a common margin size for all four sides, and allows you to alter it in IB:



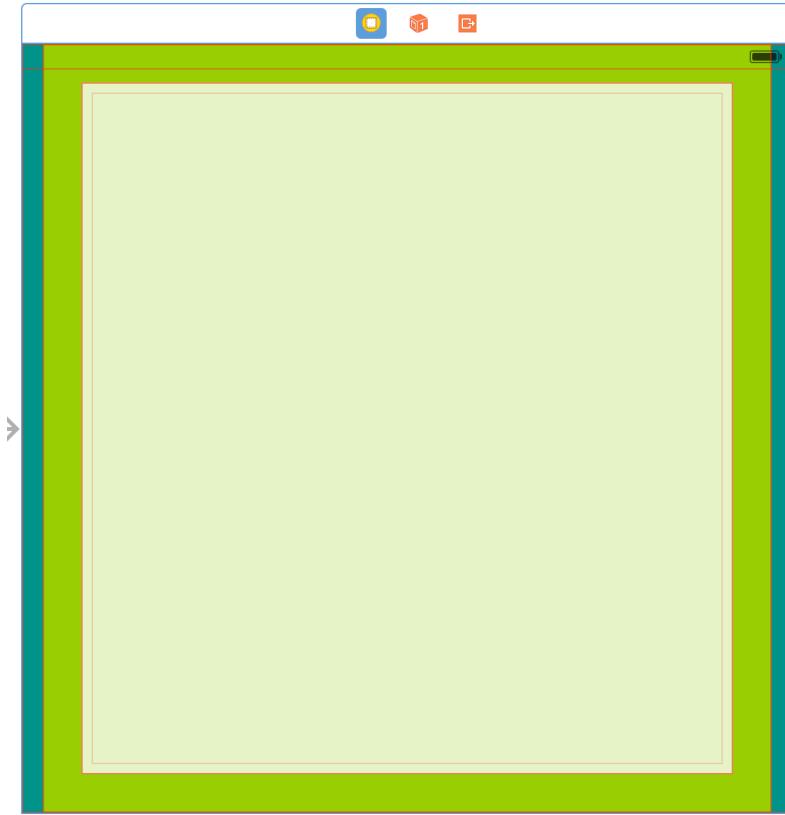
Setting Margin

Setting this value will update the canvas to demonstrate that there are now unsatisfied constraints:



Unsatisfied Constraints

Note that you can see the updated margin positions. You can of course get IB to update its layout to satisfy the constraints:

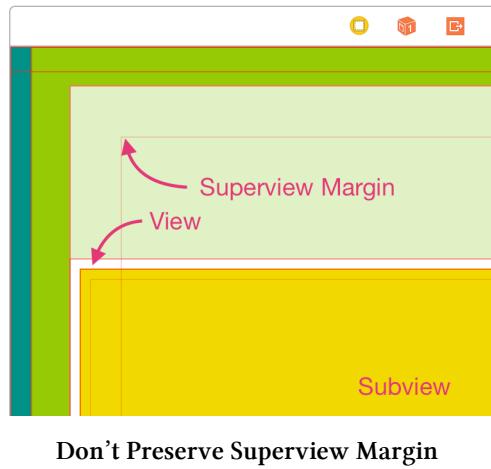


Satisfied Constraints

If you are performing manual layout, then you might also wish to override `layoutMarginsDidChange()`, which will be called whenever the margins are changed - allowing you to update your layout appropriately.

PreservingSuperviewLayoutMargins

There is one remaining margin-related property on `UIView` - `preservesSuperviewLayoutMargins`. This is a `Bool` with a default value of `false`. It describes the very specific situation where the layout of a subview is relative to the a layout margin, but this layout margin is *outside* the margin of the superview. This is quite complicated to explain, since it involves a hierarchy of three `UIViews`. The following diagram represents this scenario:



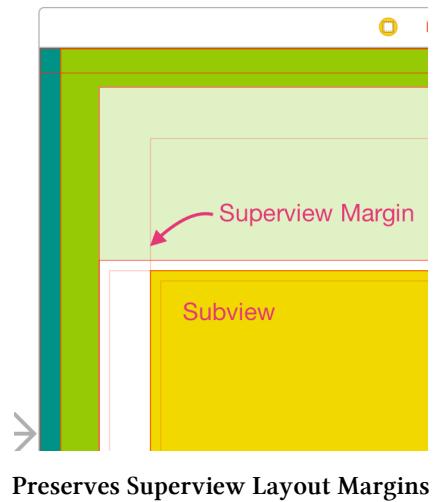
Don't Preserve Superview Margin

Here, the yellow view is positioned relative to the margin of its (white) superview. However, the white view is positioned *outside* the margin of its (light green) superview. This has the result that the yellow view is positioned relative to a margin, but is ignoring the margin of the green view.

By setting `preservesSuperviewLayoutMargins` to true this behavior changes. The following code is part of the white view:

```
1 override init(frame: CGRect) {  
2     super.init(frame: frame)  
3     preservesSuperviewLayoutMargins = true  
4 }  
5  
6 required init(coder aDecoder: NSCoder) {  
7     super.init(coder: aDecoder)  
8     preservesSuperviewLayoutMargins = true  
9 }
```

Now, the positioning of the yellow view will respect the layout margins of the light green view:



Conclusion

Today's article took a look at something that is pretty common in most other layout systems, but has been noticeably lacking in iOS. It's actually far more powerful and understandable than layout guides - probably due to there being far less mystery shrouding how it operates. It does appear to lack much documentation from Apple though, and so can be quite confusing at first.

The **Marginal** app which demos these concepts is available on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁸⁷](https://github.com/ShinobiControls/iOS8-day-by-day).

⁸⁷<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 33 :: CloudKit



Apple introduced iCloud to the world a couple of years ago, and since then has been introducing new ways for developers to utilize it. First was iCloud CoreData, which is meant to be a completely seamless way of persisting and syncing a object graph across devices. This suffered from all kinds of issues, and iOS 8 introduces something which pretends to be much less in the form of CloudKit.

CloudKit offers very little magic in terms of data-syncing, and has no elements of local persistence. It is simply a remote database - a transport mechanism for storing data remotely. Having said that, you do get a huge amount of functionality from the API - including user management, huge storage and bandwidth capacities, security and privacy.

Today's article is going to take a look at some of the features of CloudKit in reasonable detail. The framework is really big, so not every feature will be covered. There is a sample app to accompany the chapter - called **CloudNotes**. This is a simple note taking app which uses CloudKit as a persistence layer. Although this app demonstrates CloudKit fairly well, it should not be taken as 'best practice' for creating a data-driven app in this way. For example, there is no facility for offline resilience. You can grab the source code of the app from the repo on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁸⁸](https://github.com/ShinobiControls/iOS8-day-by-day).

High-level CloudKit Concepts

Before jumping in to look at some code, there are some architectural concepts associated with CloudKit that you should get your head around.

Authentication

Since CloudKit is build on iCloud, there isn't actually any authentication to worry about - provided a user is logged into iCloud on the device, and they permit your app to use iCloud then you're done! Your user would then be set up with appropriate iCloud containers, and databases.

The developer doesn't get access to a list of users - which is great for privacy. However, it does raise an interesting issue of discovery; loads of apps expect the user to interact with their friends - which would be very difficult if there's no way of discovering other users.

This is built in to CloudKit - a user can specify on a per-app basis whether they want to allow users of that app to search for them by email address. As a developer you can either implement this an individual search, or a full address book search.

⁸⁸<https://github.com/ShinobiControls/iOS8-day-by-day>

A user can opt-in or out of allowing an app to use the email address lookup functionality, or indeed using iCloud altogether in the Settings app under **iCloud Drive**.

Containers

In the same way that the local storage available to your device is accessible through a container, so too is your CloudKit allocation. This is the top-level object in the world of CloudKit, and by default each app has its own, independent container. It is possible that two apps signed by the same developer share the same container - permitting sharing between iOS and OSX, as well as across multiple apps on the same platform.

Databases

The next level down the CloudKit tree is the databases that reside inside the container. From the perspective of the app, each container has exactly two databases: one *public* and one *private*. The public database is shared between *all* users of this container - everybody can access the data, and by default everybody can write to it.

The private database is, as you might expect, completely private to the current user. This means that only the logged-in user has access to this data - they can't choose to share part of it, nor can the developer take a look at a user's private database. This is an extremely powerful feature - offering top-notch privacy straight out of the box.

The API is very clear about which database you are interacting with - all operations are performed on a database, and you use either the `privateCloudDatabase` or `publicCloudDatabase` properties on your `CKContainer` container object.

Record Zones

Unlike in traditional relational databases, you can pop your records straight into a database. You don't have to create a schema and associated tables - you are free to implicitly build your schema as you go.

However, there is one more level between a database and records, and that is the concept of a record zone. These just represent a collection of records - allowing you to partition your data appropriately. Every database is created with a default zone, but you are free to create your own custom zones as and when you wish.

Zones offer some additional functionality to record partitioning, namely:

- **Per-zone Notifications** Notifications are built into CloudKit - translating into push notifications whenever something changes (see later). You can set up a subscription to get notifications whenever something within a zone changes.

- **Atomic Commits** This is essentially transactions - ensuring that all records within a specified operation either complete or don't change at all. This is crucial to ensuring the integrity of your data.
- **Delta Changes** When an app arrives back on the network after being away it can send a 'last-known-change' token up to iCloud, and it will then be sent all the changes since that point, and a new token. This is invaluable for ensuring that your local persistence store stays in sync with that on iCloud.

It is worth noting that records cannot change zones - instead they would have to be recreated in a different zone, and that it's also not possible to have a relationship between records in different zones.

Records

As in many datastores, the lowest level of object in CloudKit is that of the record. This is represented by the `CKRecord` class, and forms the basis of every object type you want to store.

It is key-value coding compliant, and doesn't require you to specify the fields up-front (much like `NSDictionary`). The field names should be strings, and the values can be any of the following:

- `NSString`
- `NSNumber`
- `NSData`
- `NSDate`
- `CLLocation`
- `CKReference` - a reference to another `CKRecord`
- `CKAsset` - a binary blob of data uploaded to iCloud
- Arrays of the above

Records are created within a zone, and have a specified type - represented by a string. All objects of the same type will share the same set of attributes - much in the same way as you're used to in relational databases.

Records all have unique IDs, represented by the `CKRecordID` class, which combines a zone ID with a record name. You can specify the record name part of this ID when you create a record using the constructor that takes a name. This is useful if you already have a unique ID for your object, and don't want to manage two separate IDs. If you don't provide a record name, one will be randomly generated for you.

Subscriptions

Subscriptions allow you to ask iCloud to notify you of changes to the database as they occur. You've already seen a mention of record zone subscriptions, which will notify you when anything changes within a zone. It's also possible to create query subscriptions - where you specify a query (via a predicate), and the request that your app be notified if the results to the query should change.

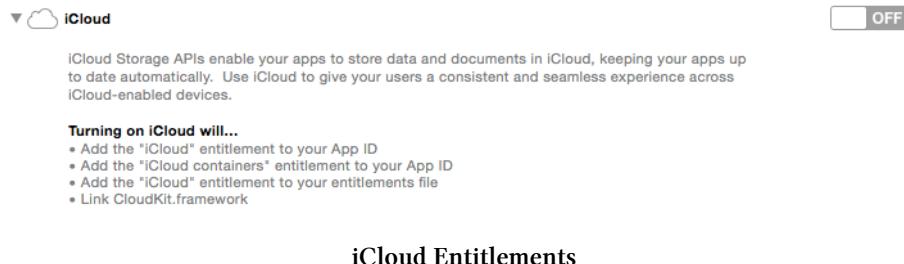
Subscription notifications are delivered as push messages via APNS. There are some caveats around their use - namely once a notification has been received, it's important that you check for changes (since push notifications get superseded).

Subscriptions offer a great alternative to polling iCloud to wait for changes. This results in reduced power consumption - the holy grail of app optimization requirements.

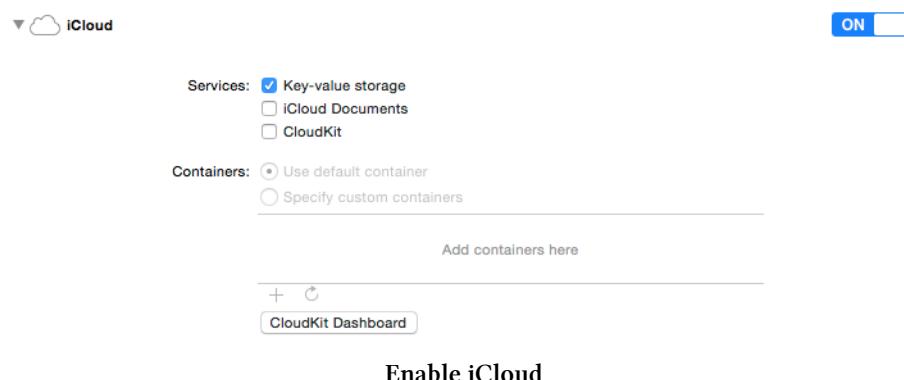
Enabling CloudKit

CloudKit needs to be enabled on a per-app basis, and associated with a particular developer ID. This involves adding the appropriate entitlements to your app ID, and linking against the CloudKit framework.

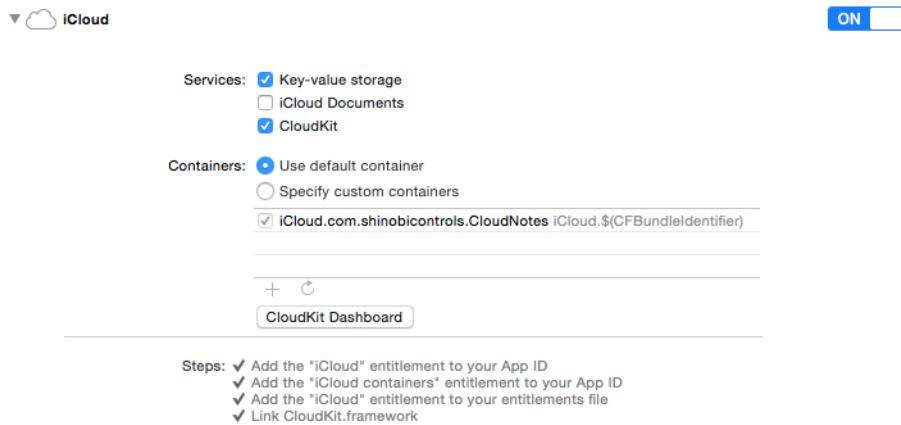
Xcode has you covered in this area - via the **Capabilities** tab of your project settings file:



Flicking the switch will enable iCloud for your app:



Note that the default settings just enable the iCloud Key-Value store, and that you need to check the **CloudKit** checkbox to link against the correct framework and configure the entitlements appropriately:



Enable CloudKit

At this point you can create custom containers, should you wish to, or you can stick with the default container created for you.

Notice that there is also a button labeled **CloudKit Dashboard** that will send your browser to the web dashboard associated with this container. You can read more about the dashboard later in this article.

Now that you've enabled CloudKit, you can go ahead and start creating pushing some data into it.

Creating Records

The accompanying sample app is a note pad, with the following protocol representing the fields contained by a note:

```

1 protocol Note {
2     var id: String? { get }
3     var title: String { get set }
4     var content: String? { get set }
5     var location: CLLocation? { get set }
6     var createdAt: NSDate { get }
7     var lastModifiedAt: NSDate { get }
8 }
```

In order to persist an object with these properties in CloudKit, you have to represent it as a CKRecord. As previously mentioned, if you don't specify a name then CloudKit will generate a unique name for

your record automatically, and as you might expect `createdAt` and `lastModifiedAt` are similarly auto-managed. This leaves you with three properties which need representing in a `CKRecord`.

A `CKRecord` behaves very much like an `NSDictionary`, in that you create fields by assigning values to keys. For example, `CloudNotes` implements the custom properties from the `Note` protocol as follows:

```
1 class CloudKitNote: Note {
2     let record: CKRecord
3
4     ...
5
6     var title: String {
7         get {
8             return record.objectForKey("title") as String
9         }
10    set {
11        record.setObject(newValue, forKey: "title")
12    }
13 }
14
15 var content: String? {
16     get {
17         return record.objectForKey("content") as? String
18     }
19     set {
20         record.setObject(newValue, forKey: "content")
21     }
22 }
23
24 var location: CLLocation? {
25     get {
26         return record.objectForKey("location") as? CLLocation
27     }
28     set {
29         record.setObject(newValue, forKey: "location")
30     }
31 }
32
33 ...
34 }
```

`CloudKitNote` contains a `CKRecord` object, and the data for the properties is accessed via `objectForKey()` and `setObject(_, forKey:)`.

There accessors for the non-custom properties just proxy through to the relevant properties on CKRecord:

```
1 var id: String? {
2     return record.recordID.recordName
3 }
4
5 var createdAt: NSDate {
6     return record.creationDate
7 }
8
9 var lastModifiedAt: NSDate {
10    return record.modificationDate
11 }
```

In this design, a CloudKitNote is either constructed with a CKRecord which has been returned from a CloudKit API, or from another Note:

```
1 init(record: CKRecord) {
2     self.record = record
3 }
4
5 init(note: Note) {
6     record = CKRecord(recordType: "Note")
7     title = note.title
8     content = note.content
9     location = note.location
10 }
```

Note that when creating a new CKRecord you have to specify *at least* the recordType. This is a string, and represents a set of objects which share common attributes - similar in concept to a table in a relational database.

Now that you have created an appropriate CKRecord, you need to tell CloudKit to save it. CloudKit actually has two distinct APIs, the so-called convenience API and the NSOperation API. As you might expect from the naming, the convenience API is a little easier to use, but at the cost of being less configurable. This article will use both APIs to show you a flavor of the two options.

There is a convenience API method on CKDatabase that allows you to save a record, in the form of `saveRecord(_, completionHandler:)`. In order to use this you need to get hold of a reference to a CKDatabase object.

Remember that a CloudKit app has one or more containers - and each of these has access to two databases. If you are just using the default container, then the `defaultContainer()` class method on

CKContainer will return you a reference. A CKContainer object then has two database properties: privateCloudDatabase and publicCloudDatabase. Since CloudNotes is currently only supporting private notes, then it uses the privateCloudDatabase to construct a custom CloudKitNoteManager object:

```
1 let noteManager = CloudKitNoteManager(database: CKContainer.defaultContainer()
2                                         .privateCloudDatabase)
```

CloudKitNoteManager is a helper class which implements the following protocol, to encompass all the different persistence methods that the app needs:

```
1 protocol NoteManager {
2     func createNote(note: Note, callback: ((success: Bool, note: Note?) -> ())?) {
3         func getSummaryOfNotes(callback: ([Note]) -> ())
4         func getNote(noteID: String, callback: (Note) -> ())
5         func updateNote(note: Note, callback: ((success: Bool) -> ())?) {
6             func deleteNote(note: Note, callback: ((success: Bool) -> ())?) {
7         }
8     }
9 }
```

Designing your app in this way (using the Note and NoteManager protocols) will make it a lot easier to add a local persistence layer, or switch out CloudKit for an alternative should you decide to.

The implementation of createNote(note:, callback:) in CloudKitNoteManager looks like this:

```
1 func createNote(note: Note, callback: ((success: Bool, note: Note?) -> ())?) {
2     let ckNote = CloudKitNote(note: note)
3     database.saveRecord(ckNote.record) { (record, error) in
4         if error != nil {
5             println("There was an error: \(error)")
6             callback?(success: false, note: nil)
7         } else {
8             println("Record saved successfully")
9             callback?(success: true, note: ckNote)
10        }
11    }
12 }
```

Notice that first we construct a CloudKitNote object from the supplied Note. This allows you to use any object that conforms to the Note protocol (in fact, in CloudNotes, this will be of type PrototypeNote, which is a POSO [I think I just invented that acronym (C)SD]).

Once you have a CKRecord then you can call saveRecord(_, completionHandler:) on your CKDatabase object. The completion handler is a closure which includes a Bool to indicate success and

an `NSError` object. It is vitally important that you implement this completion handler, and actually inspect the error.

I'll say that again. You can't just ignore the error like you usually do. CloudKit **will fail**. For perfectly legitimate reasons. If you don't handle the error then your app **will lose data**.

There are a total of 28 CloudKit-specific error codes, including things such as `NetworkUnavailable`, `NotAuthenticated`, `LimitExceeded` and `ServerRejectedRequest`. When you build an app around you need to *at least* investigate and handle the errors associated with network issues. Your users are guaranteed to try to use your app without network access. How you handle this is the difference between having users and not having users.

Note that despite mentioning how important errors are, CloudNotes doesn't really handle that. Writing good error code is a pain, and is left to an exercise for the reader ;-)

The other important thing that's worth mentioning is that the completion handler is not likely to be called back on the main thread. Therefore, ensure you marshal any UI code back onto the main queue.

The following shows the `createNote(_:, callback:)` method in use in the `MasterViewController`, in a delegate method which creates a note:

```
1 func completedEditingNote(note: Note) {
2     dismissViewControllerAnimated(true, completion: nil)
3     showOverlay(true)
4     noteManager.createNote(note) {
5         (success, newNote) in
6         self.showOverlay(false)
7         if let newNote = newNote {
8             let newCollection = self.noteCollection + [newNote]
9             self.noteCollection = newCollection
10        }
11    }
12 }
```

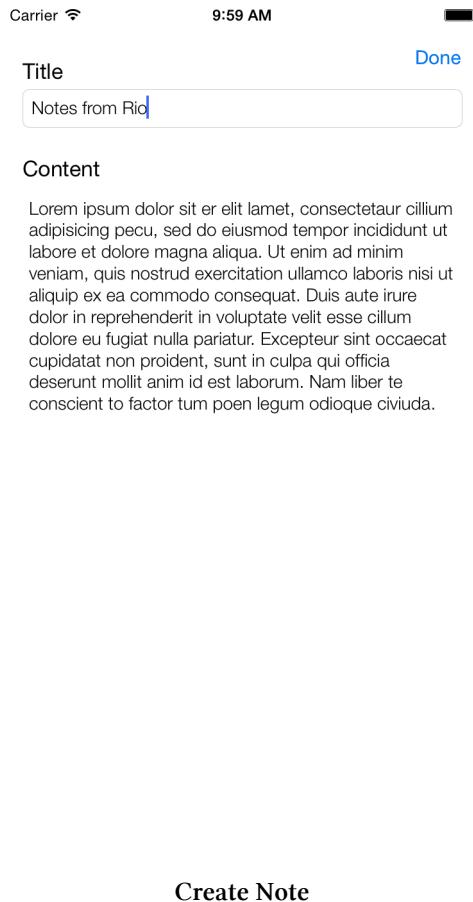
There are two points which involve updating the UI, hiding the "Loading" overlay (terrible UX, I know):

```
1 private func showOverlay(show: Bool) {
2     dispatch_async(dispatch_get_main_queue()) {
3         UIView.animateWithDuration(0.5) {
4             self.loadingOverlay.alpha = show ? 1.0 : 0.0
5         }
6     }
7 }
```

And reloading the table's data when the noteCollection is updated:

```
1 var noteCollection: [Note] = [Note]()
2 didSet {
3     dispatch_async(dispatch_get_main_queue()) {
4         self.tableView.reloadData()
5     }
6 }
7 }
```

Notice that both of these marshal back to the main queue for UI updates.



Querying For Records

There are two methods on the `NoteManager` protocol which represent queries for records - `getSummaryOfNotes(callback:)` and `getNote(noteID: , callback:)`. The latter of these uses the convenience API, and the implementation is as follows:

```
1 func getNote(noteID: String, callback: (Note) -> ()) {
2     let recordID = CKRecordID(recordName: noteID)
3     database.fetchRecordWithID(recordID) {
4         (record, error) in
5         if error != nil {
6             println("There was an error: \(error)")
7         } else {
8             let note = CloudKitNote(record: record)
9             callback(note)
10    }
```

```
11     }
12 }
```

This uses the `CKDatabase` method `fetchRecordWithID`. As previously mentioned, a `CKRecordID` is a unique identifier for a record within a container - combining a zone ID with a record name. Since CloudNotes only uses the default zone, a `CKRecordID` can be constructed solely from the `noteID`.

`fetchRecordWithID` is another asynchronous method, and so takes a completion handler closure which returns a `CKRecord` object, and an `NSError`. It's again really quite important to handle the error appropriately.

Here, a `CloudKitNote` object is created from the returned `CKRecord`, and returned via the supplied callback. Note again that this will not be called on the main thread, so and UI updates will need marshaling to the main thread.

There is also a convenience method on `CKDatabase` which allows you to run more generalized queries - `performQuery(_:, inZoneWithID:completionHandler:)`. A query is of the type `CKQuery`, which combines a record type with a predicate and sort descriptors. The predicate is of type `NSPredicate`, and can therefore be incredibly expressive. There are some things that `NSPredicate` can express that are not supported by `CKQuery` - and you can find details of exactly what is support in the `CKQuery` documentation. It is possible to include queries based on location - which is really helpful - removing any of the difficult spherical mathematics from your own code.

The one thing that this convenience method doesn't allow you to do is restrict which fields you want returned. `CKRecord` objects can represent *partial* records - that is to say, not containing all the properties that exist on the original object. This can be really helpful if you just want to display a summary of records, each of which contains a huge number of properties.

In CloudNotes, the table of all notes only needs the creation date and title of the notes, and therefore it would be good to not have to request the content and location (this is somewhat of a contrived example - the content and location properties of the note are pretty small).

In order to request partial records, you have to drop the convenience API and use the operation-based API instead.

NSOperation-based API

The convenience API is a wrapper around a much more powerful API - which is based around `NSOperation`. `CKOperation` is an abstract base class that represents all the different operations that are performed on a CloudKit database. There are concrete subclasses of this class, each of which represents a specific type of operation. For example:

- `CKQueryOperation`
- `CKModifyRecordsOperation`
- `CKFetchSubscriptionsOperation`

- ...

You'll need to use the operation API whenever you want to perform slightly more complex tasks with the database. Since these objects all inherit from `NSOperation`, they are invoked by adding them to an `NSOperationQueue`. You can either provide your own, or use the default one associated with the database. This affords you a huge amount of control when specifying dependencies between operations, and priorities.

In `CloudNotes`, a `CKQueryOperation` is used in the implementation of `getSummaryOfNotes(callback:)`:

```
1 func getSummaryOfNotes(callback: (notes: [Note]) -> ()) {
2     let query = CKQuery(recordType: "Note", predicate: NSPredicate(value: true))
3     let queryOperation = CKQueryOperation(query: query)
4     queryOperation.desiredKeys = ["title"]
5     var records = [Note]()
6     queryOperation.recordFetchedBlock = {
7         record in records.append(CloudKitNote(record: record))
8     }
9     queryOperation.queryCompletionBlock = { _ in callback(notes: records) }
10
11    database.addOperation(queryOperation)
12 }
```

Firstly, a `CKQuery` is constructed, which specifies the correct `recordType` of `Note`, and since the notes don't need filtering, the predicate is just the `true` predicate - i.e. a filter that always returns `true`.

You can then use this `CKQuery` object to instantiate a `CKQueryOperation`. Amongst other properties (such as `resultsLimit`), this has a property named `desiredKeys`. This is an array of strings, that allows you to specify which of the keys you want to retrieve - creating partial records. Since this method is used to populate the table view, on `title` is required.

`recordFetchedBlock` and `queryCompletionBlock` allow you to provide closures which will be called after each record arrives and after the query completes respectively. These are used to construct an array of notes, and then passing them back via the supplied callback block.

Finally, once you've created the operation, then you need to add it to an operation queue so that it gets invoked. Here it's being added to the default operation queue associated with the database.

The combination of these two methods allows `CloudNotes` to retrieve the records from CloudKit, both for display in the table, and then (with a more rich representation) in the detail view.

I fancy a trip to Russia 2014-10-14 08:58:54...

Live from Hawaii 2014-10-14 08:59:17 +0000

Notes from Rio 2014-10-14 08:59:56 +0000

hello from London

Placeholder text (lorem ipsum)

Map of the United Kingdom and surrounding countries, with a red dot on London.

Modifying Records

The final two operations to look at in the basic CRUD CloudNotes app are update and delete. Both of these use the `CKModifyRecordsOperation` class to update the database. This has a constructor which takes an array of `recordsToSave`, and an array of record IDs to delete.

It also has `perRecordCompletionBlock` and `modifyRecordsCompletionBlock` closures to get feedback on the process.

The `updateNote(note:, callback:)` method is implemented as follows:

```

1 func updateNote(note: CloudKitNote, callback:((success: Bool) -> ())?) {
2     let updateOperation = CKModifyRecordsOperation(recordsToSave: [note],
3                                                 recordIDsToDelete: nil)
4     updateOperation.perRecordCompletionBlock = { record, error in
5         if error != nil {
6             // Really important to handle this here
7             println("Unable to modify record: \(record). Error: \(error)")
8         }
9     }
10    updateOperation.modifyRecordsCompletionBlock = { saved, _, error in
11        if error != nil {
12            if error.code == CKErrorCode.PartialFailure.toRaw() {
13                println("There was a problem completing the operation. The following " +
14                    "records had problems: \(error.userInfo?[CKPartialErrorsByItemIDKey])")
15            }
16            callback?(success: false)
17        } else {
18            callback?(success: true)
19        }
20    }
21    database.addOperation(updateOperation)
22 }
```

Again it's important to handle errors. In this instance, the error passed to the `modifyRecordCompletionBlock` might contain a code of `PartialFailure`, which indicates that some of the modifications weren't successful. In this case, you can get hold of the records involved with the `CKPartialErrorsByItemIDKey` entry on the `userInfo` dictionary.

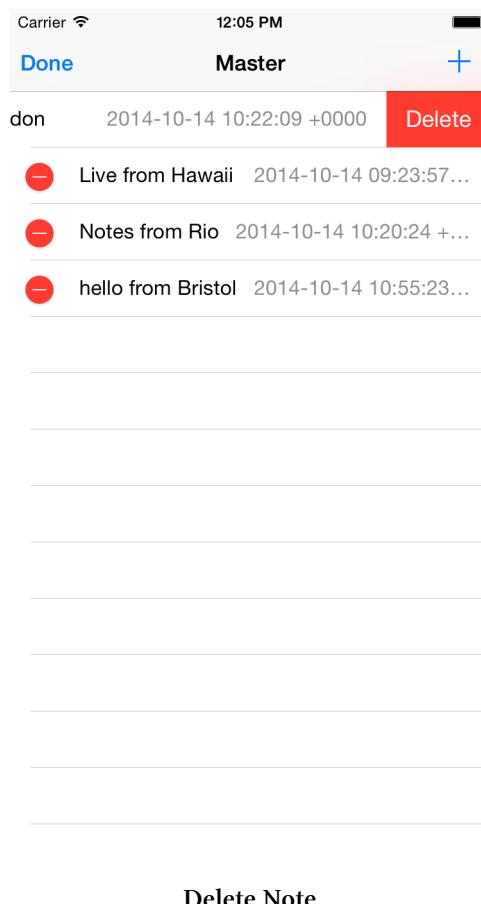
The `deleteNote(note:, callback:)` method is almost identical in its implementation:

```

1 func deleteNote(note: Note, callback: ((success: Bool) -> ())?) {
2     let deleteOperation = CKModifyRecordsOperation(recordsToSave: nil,
3                                                 recordIDsToDelete: [CKRecordID(recordName: note.id)])
4     deleteOperation.perRecordCompletionBlock = { record, error in
5         if error != nil {
6             println("Unable to delete record: \(record). Error: \(error)")
7         }
8     }
9     deleteOperation.modifyRecordsCompletionBlock = { _, deleted, error in
10        if error != nil {
11            if error.code == CKErrorCode.PartialFailure.toRaw() {
12                println("There was a problem completing the operation. The following " +
13                    "records had problems: \(error.userInfo?[CKPartialErrorsByItemIDKey])")
14            }
15        }
16    }
17 }
```

```
14      }
15      callback?(success: false)
16  }
17  callback?(success: true)
18 }
19 database.addOperation(deleteOperation)
20 }
```

In this case, a CKRecordID is created for the specified note, and then this is added to the recordIDsToDelete array.



CloudKit Dashboard

You've now been building your datastore, and can grab things out of it, but it's very much a black box. It would be really helpful to be able to see what's going on inside your app's container. Well, Apple thought of this, and built the CloudKit Dashboard:

The screenshot shows the 'Record Types' section of the CloudKit dashboard. A 'Note' record type is selected, which has three attributes: 'content' (String, indexed), 'title' (String, indexed), and 'location' (Location, indexed). The sidebar includes sections for Schema (Record Types, Security Roles, Subscription Types), PUBLIC DATA (User Records, Default Zone), PRIVATE DATA (Default Zone), and ADMIN (Deployment). The environment is set to DEVELOPMENT.

CloudKit Dashboard

You can access the dashboard via [icloud.developer.apple.com⁸⁹](https://icloud.developer.apple.com), or via the link in the iCloud capabilities tab.

As you build up record types you can see them and their attributes. It's also possible to edit, remove and add new attributes to the types you've created, as well as specifying which should be indexed, searchable and sortable.

The security roles pane allows you to specify roles, and associated permissions on a per-record-type basis. This allows you to build a structure to your public database whereby you have varying levels of protection:

The screenshot shows the 'Security Roles' section of the CloudKit dashboard. An 'Admin' role is selected, which has permissions for the 'Note' record type, specifically 'Create', 'Read', and 'Write'. The sidebar includes sections for Schema (Record Types, Security Roles, Subscription Types), PUBLIC DATA (User Records, Default Zone), PRIVATE DATA (Default Zone), and ADMIN (Deployment). The environment is set to DEVELOPMENT.

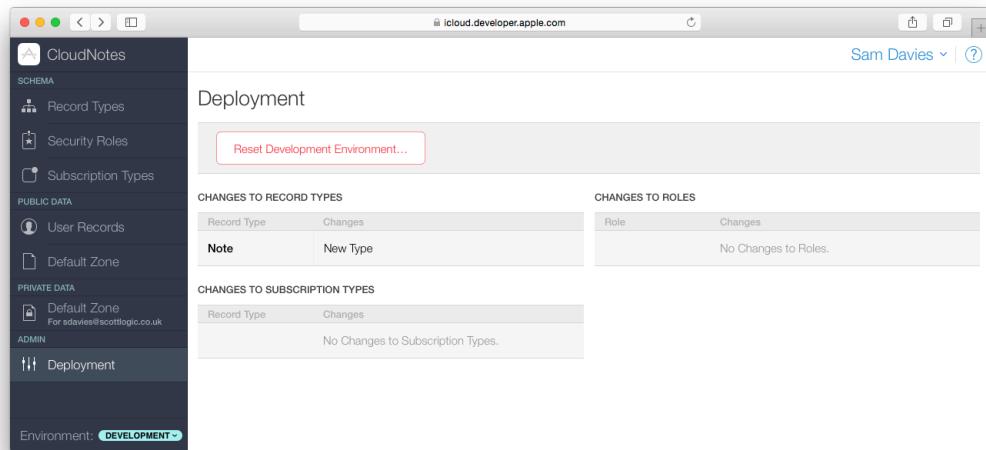
Security Roles

⁸⁹<https://icloud.developer.apple.com>

Subscription Types allows you to see and edit the subscriptions that you've created from within your app, and you can also see the data that exists within the different zones. Note that although you can see all the public data (as you might expect), you can only see the private data associated with your developer account (i.e. the account you're logged in with).

Because of this, it's really handy to have a different developer ID than your personal version. You can even log in to an iCloud ID on a simulator, although in order for it to be activated for iCloud, you **must** log in to it on a device before hand. This is a pain.

The final panel of interest is **Deployment**. This allows you to push changes (of structure etc) from your development database to production, and also to completely reset the development database. This functionality is invaluable, and if you're finding that all your CloudKit requests are getting internal server errors, then I'd suggest giving this big red button a try:



Deployment

Summary of other Features

CloudKit is huge - and today's post has only really covered some of the more basic functionality. Here's a summary of some of the other features you'd expect from a datastore, and a brief description of how they apply in the CloudKit world:

- **Change Notifications** Subscriptions allow you to build a query and then ask iCloud to send you push messages whenever the results of the query are changed. Or you can create a record zone subscription which will send a push notification whenever data changes.
- **Relationships** CKReference allows you to specify that two records are related to each other. The documentation highly recommends that in one-to-many situations the reference goes backwards. i.e. a child has one parent rather than a parent having multiple children.

- **Blob fields** If you have large chunks of data (such as images) that you wish to associate with a record then you can use the `CKAsset` class to upload, retrieve and reference this data. Note that in iOS 8.1 it will only be possible for an asset to belong to a single record.
- **Transactions** In a custom zone it's possible to specify that a particular set of `CKOperations` should be an 'atomic commit'. That is to say that either all operations will succeed, or none of them will be committed. This is really useful to maintain data integrity.
- **Cascading Deletes** By default, if you delete a record which has a reference to another record, then the child record will be deleted as well. It's worth noting that if you have multiple references to the same record, then the first delete will win - i.e. if you don't want the child deleted you need to change the default behavior.
- **Save Rules** In any database with multiple points of asynchronous access, managing conflicts is a hugely complex task. By default, CloudKit uses a very conservative rule, placing the responsibility for resolving these completely in your hands. If you wish to then force overwrite the contents of a record in iCloud then you must change the save rule to represent this.
- **Production** Whilst developing your app, the schema is being built up as you add records and structure. At the point that you want to push your app into the wide world, you don't want the schema changing again. Nor do you want your database full of dirty dev data. CloudKit allows you to switch to a production database through the CloudKit Dashboard. At this stage you are also able to specify which attributes should be indexed and searchable. This allows the database engine to optimize appropriately.

Conclusion

Phew - that was a long one! As you have probably realized, CloudKit is enormous. It is incredibly versatile, and appears to be pretty well thought-out. It has loads of advantages - huge quotas, simplicity, versatility, functionality, user management, user trust, privacy, security etc. I do have some concerns about potential disadvantages as well though.

As you might expect from Apple, the tooling of the first iteration just isn't quite there. Things don't always work as expected - the dashboard throws up frequent errors, I spent a long time trying to diagnose "internal server error", and simulator support is flaky at best. However, I'd expect all of these things to be fixed over the next year.

Far more concerning is the lock-in to Apple. It might be great to prototype something with CloudKit, but can you definitely say that you don't want to build a custom web dashboard to your data at a later stage? Or maybe an Android app? It'd be great if Apple were to open up a REST API or something, to at least allow the possibility of expansion in the future, but it's not in their interest to do so.

CloudKit is extremely powerful, and relatively easy to get started with. It's certainly worth having a play around with it, but be aware that it's not a magic bullet. You still need to cope with all the difficulties associated with a remote system going offline when the user moves out of range - local persistence and conflict resolution. And you should also consider whether or not this is a long-term solution, or just something for prototyping.

As ever, the project associated with this article is available on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁹⁰](https://github.com/ShinobiControls/iOS8-day-by-day). Please remember that this project was created to demo CloudKit, and almost certainly has problems if you were to try and use it as a real note-taking app. Feel free to take it and fix that - I'll happily take a pull request.

⁹⁰<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 34 :: CoreLocation Authorization

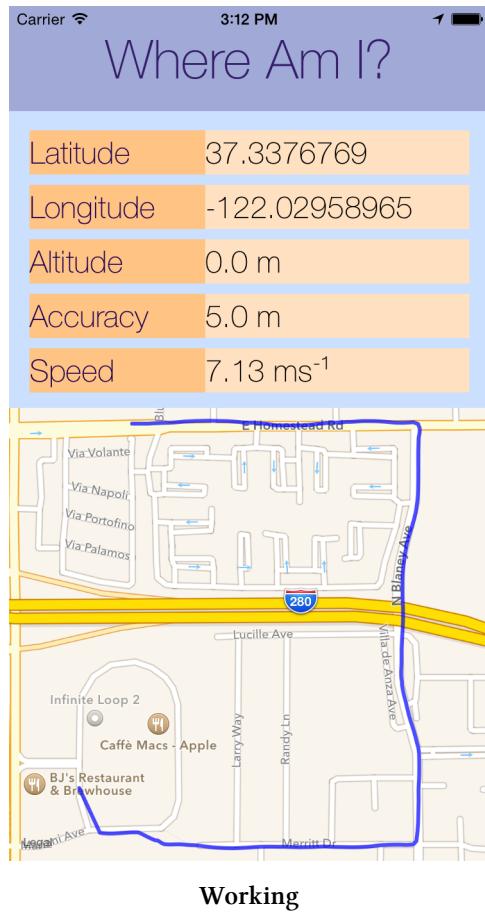


CoreLocation is the framework which gives you access to the geo-location functionality of devices within iOS. You'd use it if you wanted to find the user's currently location. The functionality provided by CoreLocation is obviously very sensitive - it's important for users to be able to manage their privacy. In iOS7 and before, CoreLocation provided this functionality automatically, but this is no longer the case.

In today's article you'll discover the different options for privacy with CoreLocation introduced in iOS 8, and how you need to update your apps to use them.

The app which accompanies today's post is a simple route tracking app called **Where Am I?**. It registers for updates from CoreLocation, displays them on screen and plots the route on a map. You can get hold of the source code for **Where Am I?** on the ShinobiControls github at [github.com/ShinobiControls/iOS8-day-by-day⁹¹](https://github.com/ShinobiControls/iOS8-day-by-day).

⁹¹<https://github.com/ShinobiControls/iOS8-day-by-day>



Working

Refresher on CoreLocation

The model for finding the current location using CoreLocation is via a manager object that pushes out updates via a delegate. `CLLocationManager` can do more than this, but in the simplest example you might create one like this:

```
1 let locationManager = CLLocationManager()
```

And then configure it as follows:

```
1 locationManager.delegate = self
2 locationManager.desiredAccuracy = 20
3 locationManager.startUpdatingLocation()
```

You need to set a delegate, and then tell it that it should start work.

The delegate adopts the `CLLocationDelegate` protocol, and in the simple case you can just implement the `locationManager(_, didUpdateLocations:)` method:

```
1 func locationManager(manager: CLLocationManager!,  
2                     didUpdateLocations locations: [AnyObject]!) {  
3     if let location = locations.first as? CLLocation {  
4         // Update the fields as expected:  
5         latValLabel.text = "\(location.coordinate.latitude)"  
6         longValLabel.text = "\(location.coordinate.longitude)"  
7         altValLabel.text = "\(location.altitude) m"  
8         accValLabel.text = "\(location.horizontalAccuracy) m"  
9         spdValLabel.text = "\(location.speed) ms⁻¹"  
10        // Re-center the map  
11        mapView.centerCoordinate = location.coordinate  
12        // And update the track on the map  
13        historicalPoints.append(location.coordinate)  
14        updateMapWithPoints(historicalPoints)  
15    }  
16 }
```

This extracts the details we're interested in from the `CLLocation` object, and updates the display appropriately.

If you build and run an app which contains this code in iOS 8, you'll see something like:



Doesn't Work

Eh? The updates aren't appearing. This would have worked in iOS 7 (with a bit of hand-waving about Swift).

This is because in iOS 7, the location manager was responsible for asking the user whether they wanted to allow access to the CoreLocation services. This is no longer true in iOS 8 - despite the lack of error messages, there's a problem with your current code.

New Methods on CoreLocationManager

iOS 8 adds finer granularity on the privacy controls for location access. Namely, a user can choose whether an app should be allowed to track their location even when it's in the background, as opposed to in the foreground or not at all.

To support this, there are two new methods on `CLLocationManager` - `requestWhenInUseAuthorization()` and `requestAlwaysAuthorization()`. The names of these methods make it pretty obvious what they do:

- `requestWhenInUseAuthorization()` If the user agrees then your app will have access to CoreLocation only when it is in the foreground.

- `requestAlwaysAuthorization()` If the user agrees your app can access CoreLocation even when it is in the background.

The alert displayed to the user will make this clear. If you requested the *always* option then the user will be prompted after your app has been using CoreLocation in the background - allowing them to confirm that they're happy with this.

So updating your location manager configuration code should fix your problems:

```

1 // Prepare the location manager
2 locationManager.delegate = self
3 locationManager.desiredAccuracy = 20
4 // Need to ask for the right permissions
5 locationManager.requestWhenInUseAuthorization()
6 locationManager.startUpdatingLocation()

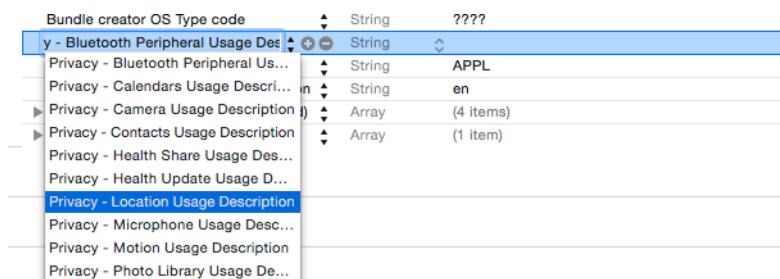
```

However, if you run it up again, then you'll see that it still doesn't prompt the user, and your delegate method won't receive any updates. Surely there's not more to do? Turns out, despite the lack of error messages, that there is indeed more to do. Excellent.

Providing Usage Strings

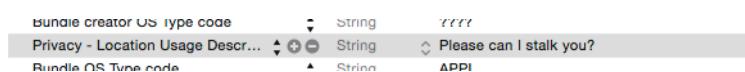
CoreLocation now wants to display a message to the user asking for their consent to start sending their tracking data around. The problem is that it doesn't know why you want it, and it's not prepared to ask the user for their permission when it doesn't know why.

You need to specify a so-called "usage string", which will be displayed to the user explaining what your app will do with the location updates it has requested. You do this in the `Info.plist` file:



Incorrect plist

And fill in an appropriate string:



Don't write this

If you run your app up yet again you'll notice that it still doesn't work. This is getting kinda ridiculous now.

This is because that key (`NSLocationUsageDescription`) is ignored in iOS 8 and later. Instead, two new keys have been introduced, corresponding to the two privacy modes introduced before:

- `NSLocationAlwaysUsageDescription` foreground and background use
- `NSLocationWhenInUseUsageDescription` foreground only use

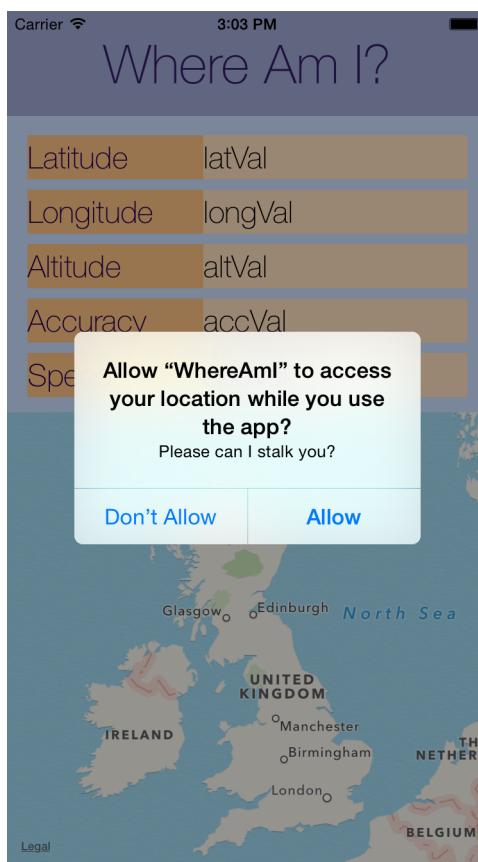
Helpfully, these keys don't have nice descriptive names in Xcode, so they don't autocomplete in the default view of the plist editor.

You can enter the key name directly:

Application requires iPhone environment	Boolean	YES
NSLocationWhenInUseUsageDescription	String	Please can I stalk you?

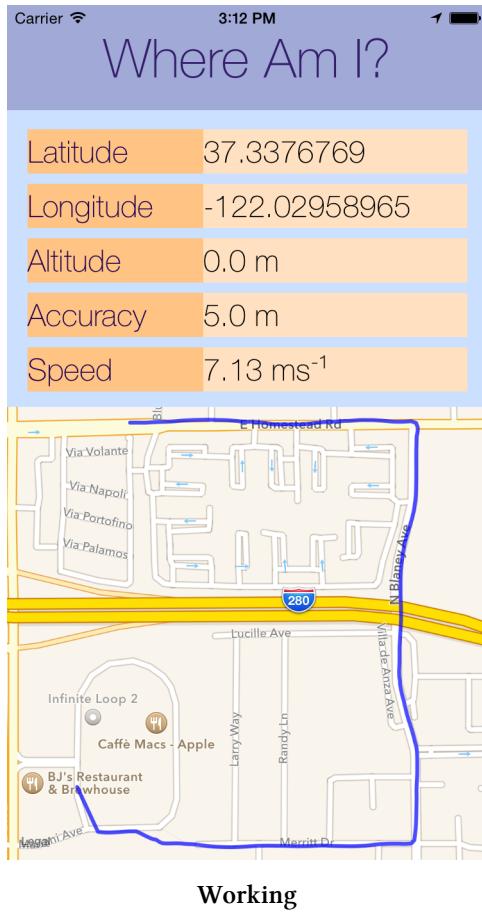
Correct plist

You can run the app up again, and finally you'll see the location privacy request, complete with your string (please don't use this example):



Asking Permission

If you grant access, then the app will then work as expected - receiving location updates from the location manager via its delegate:



Conclusion

Today's post has been quite short, but nonetheless important. The lack of error reporting associated with this change in usage of `CLLocationManager` is sure to trip people up - especially the required keys in `Info.plist`.

On the plus-side this really does represent an improvement in users' privacy. They then have really fine-grained control on which apps are able to do what, and in easy to understand terms.

As ever, the code associated with today's post is available on the ShinobiControls github at: github.com/ShinobiControls/iOS8-day-by-day⁹².

⁹²<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 35 :: CoreMotion



The **CoreMotion** framework is used in iOS to receive data from the plethora of motion sensors available on a device. This includes the magnetometer, the pedometer and the gyroscope. iPhone 6 and 6 Plus include an altimeter, so CoreMotion has been updated to support this new sensor. iOS 8 also sees some other changes to the framework.

In today's chapter you'll see what replaces `CMStepCounter` due to its deprecation, and the new type of activity classification available. There is an accompanying app called **LocoMotion** that demonstrates both how to present live motion data, and how to query for historical data. You can grab the source code from the ShinobiControls github at github.com/ShinobiControls/iOS8-day-by-day⁹³.

Motion Activity Data

At the highest level, CoreMotion offers estimates of "activity" data - i.e. an estimate of what the user is actually doing. This is the result of a classification algorithm which takes all the motion sensors (along with other data) into account. In iOS 7, the activities could be classified into **running**, **walking**, **automotive** and **stationary**. iOS 8 adds a new classification in the form of **cycling**.

Getting hold of motion activity data can take place in two different ways - dependent on your use case. You can either request up to 7 days worth of historical data, or you can ask for a live stream of updates, as the device recognizes changes in the activity state.

`CMMotionActivityManager` is the class that is responsible for motion activities, and a historical 'pull' query uses the `queryActivityStartingFromDate(_:, toDate:, toQueue:, withHandler:)`. Notice that you have to provide a queue upon which your results will be delivered - this is great for maintaining a responsive app, and is just an instance of `NSOperationQueue`:

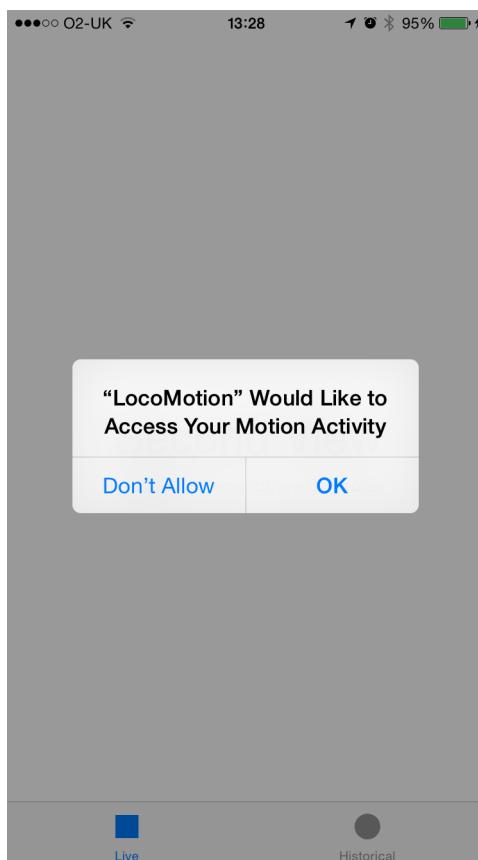
```
1 let oneWeekInterval = 24 * 3600 as NSTimeInterval
2 motionActivityManager.queryActivityStartingFromDate(
3             NSDate(timeIntervalSinceNow: -oneWeekInterval),
4             toDate: NSDate(), toQueue: motionHandlerQueue) {
5     (activities, error) in
6     if error != nil {
7         println("There was an error retrieving the motion results: \(error)")
8     }
9     self.activityCollection =
10        ActivityCollection(activities: activities as [CMMotionActivity])
11 }
```

⁹³<https://github.com/ShinobiControls/iOS8-day-by-day>

The handler is called with an array of `CMMotionActivity` objects and an error. If the user hasn't authorized use of the motion sensors then you can inspect the error and see that it has a type of `CLErrorMotionActivityNotAuthorized`.

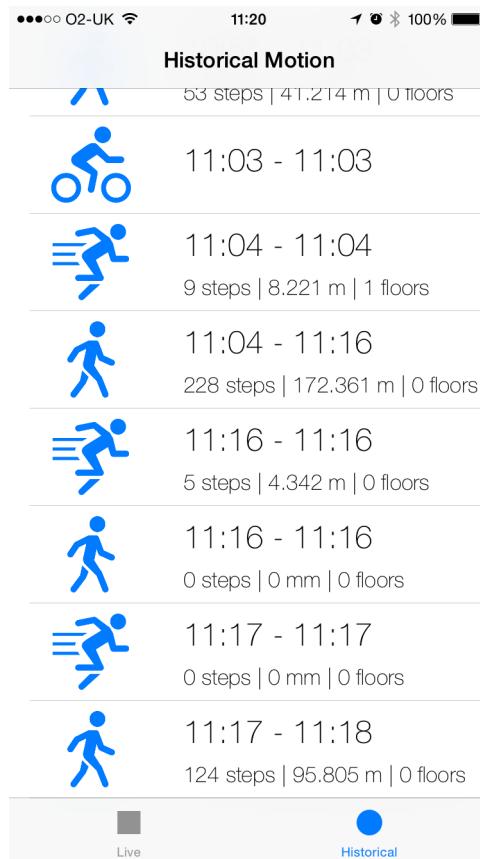
A `CMMotionActivity` object has some `Bool` properties, which determine which kind of activity was detected. These comprise stationary, walking, running, automotive, cycling and unknown. There are various rules which specify when each of these can occur (e.g. unknown will only be true when the device has just been turned on) and most of them were present in iOS 7. iOS 8 adds the cycling state. The activity also has a `startDate` and a `confidence`.

The first time you run an app, CoreMotion will present the user with an alert asking whether they are happy to share their motion data:



Request Permission

LocoMotion pulls the last 24 hours of activity data out, and displays in in a table:



Historical Data

Getting live activity data from `CMMotionActivityManager` uses the `startActivityUpdatesToQueue(_:, withHandler:)` method. Again this requires an `NSOperationQueue` to determine which queue the results should be delivered on, and a handler, which has a single `CMMotionActivity` argument.

The following method updates an image view with an appropriate logo given the current activity type:

```

1 activityManager.startActivityUpdatesToQueue(dataProcessingQueue) {
2     data in
3     dispatch_async(dispatch_get_main_queue()) {
4         if data.running {
5             self.activityImageView.image = UIImage(named: "run")
6         } else if data.cycling {
7             self.activityImageView.image = UIImage(named: "cycle")
8         } else if data.walking {
9             self.activityImageView.image = UIImage(named: "walk")
10    } else {
11        self.activityImageView.image = nil
12    }
13 }
```

```
13     }
14 }
```

Note that unless you request that results are delivered on the main queue, then you'll need to marshal any UI updates over to the main queue.

You use the `stopActivityUpdates()` method to cancel the delivery of updates.

To determine whether the current device supports motion activities, you can use the `isActivityAvailable()`

Pedometer Data

iOS 7 introduced the ability to quantify the walking behavior of the user via the `CMStepCounter` class. Well, that lasted all of a year - iOS 8 deprecates it in place of the much more generically named `CMPedometer`. This new class adds functionality for measuring distance and floors climbed, where available.

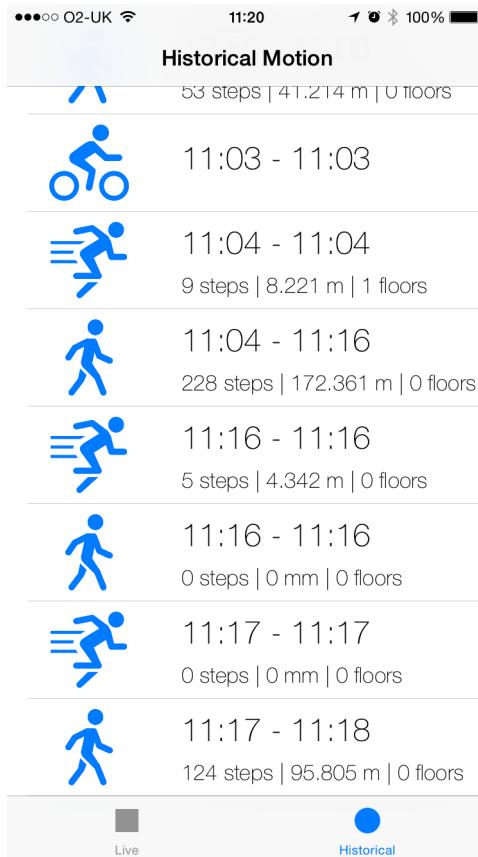
The class methods `isStepCountingAvailable()`, `isDistanceAvailable()` and `isFloorCountingAvailable()` provide info on what capabilities are available on the current device.

Again, you can request both live data, and historical data. Historical data is provided through the `queryPedometerDataFromDate(_:, toDate:, withHandler:)` and live data deliveries are provided via `startPedometerUpdatesFromDate(_:, withHandler:)` and `stopPedometerUpdates()`.

Both of these handlers have two parameters: an `NSError` and a `CMPedometerData` object which contains properties for `numberOfSteps`, `distance`, `floorsAscended` and `floorsDescended`. The following code queries for pedometer data between two dates:

```
1 pedometer?.queryPedometerDataFromDate(activity?.startDate,
                                         toDate: activity?.endDate) {
2
3     (data, error) -> Void in
4     if error != nil {
5         println("There was an error requesting data from the pedometer: \(error)")
6     } else {
7         dispatch_async(dispatch_get_main_queue()) {
8             self.pedometerLabel.text = self.constructPedometerString(data)
9         }
10    }
11 }
```

This renders the appropriate data for the running and walking activities in the historical table view:



Historical Data

Since ‘step count’ is a cumulative property, requesting live data also requires you provide a start date. Then, each call to your handler closure will give you the current cumulative total from the start date you provided:

```

1 pedometer.startPedometerUpdatesFromDate(NSDate()) {
2     (data, error) in
3     if error != nil {
4         println("There was an error obtaining pedometer data: \(error)")
5     } else {
6         dispatch_async(dispatch_get_main_queue()) {
7             self.floorsLabel.text = "\(data.floorsAscended)"
8             self.stepsLabel.text = "\(data.numberOfSteps)"
9             self.distanceLabel.text =
10                "\((self.lengthFormatter.stringFromMeters(data.distance)))"
11        }
12    }
13 }
```

Note that again, you should marshal any UI updates to the main queue, even though (somewhat

inconsistently) you don't provide the pedometer with a queue that you'd like your results delivered on.



Altimeter Data

iPhone 6 and 6 Plus added a barometer, allowing you to measure the air pressure, and therefore infer the relative change in altitude over small periods of time. It doesn't make sense to present historical relative altitude, and therefore you can only access data from this sensor on a live basis. The API is, however, very similar to that of the previous APIs you've seen in this article.

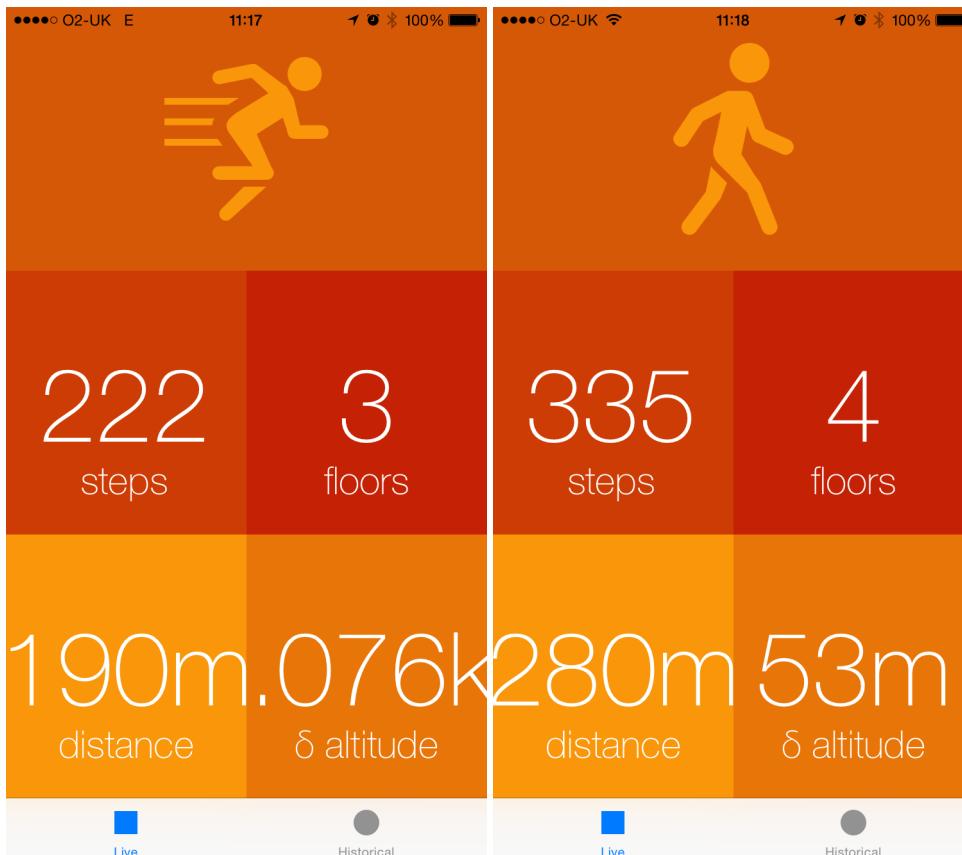
`CMAltimeter` has a class method `isRelativeAltitudeAvailable()` that will tell you whether or not the current device supports relative altitude measurements. Provided it is, you can then use `startRelativeAltitudeUpdates(_, withHandler:)` and `stopRelativeAltitudeUpdates()` to receive the updates.

The handler is called with two parameters - a `CMAltitudeData` and `NSError`. `CMAltitudeData` has a `pressure` property that represents the current pressure (measured in kilopascals) and a `relativeAltitude` property. This `relativeAltitude` property represents the *change* in altitude since the previous measurement.

The following implements this functionality:

```
1 altimeter.startRelativeAltitudeUpdatesToQueue(dataProcessingQueue) {  
2     (data, error) in  
3     if error != nil {  
4         println("There was an error obtaining altimeter data: \(error)")  
5     } else {  
6         dispatch_async(dispatch_get_main_queue()) {  
7             self.altChange += data.relativeAltitude  
8             self.altitudeLabel.text =  
9                 "\(\self.lengthFormatter.stringFromMeters(self.altChange))"  
10        }  
11    }  
12 }
```

This ensures that the displayed altitude delta represents the change since the beginning of the session.



Conclusion

These new additions to CoreMotion are primarily there to wrap the new functionality provided by the improvements to hardware. They do offer some interesting possibilities around data driven apps, and represent the versatility of the motion co-processor chips that Apple has created.

I have a few issues with the API - although all these classes are performing very similar operations, some of the methods allow you to provide a queue, whilst others don't. Maybe somebody could tell me why this is.

Today's app (**Locomotion**) demos both the live and historical data functionality, and is available on github at github.com/ShinobiControls/iOS8-day-by-day⁹⁴.

⁹⁴<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 36 :: Location Notifications



iOS 8 saw some changes to the way notifications work - particularly around authorization and local notifications. Local notifications can be triggered on a timer, and new to iOS based on location region monitoring.

It was possible to set up geofencing and present users with notifications when they enter or leave an specified region in iOS 7, but it was far from easy to set up. This has all become a lot easier in iOS 8, but location-based notifications gaining first-class support.

The app which accompanies today's post is called **NearlyThere** and allows a user to drop a pin on the map and then ask to be notified when they arrive within 50m of said pin. The code is available in the [iOS8-day-by-day](https://github.com/ShinobiControls/iOS8-day-by-day) repo at github.com/ShinobiControls/iOS8-day-by-day⁹⁵.

Requesting Authorization

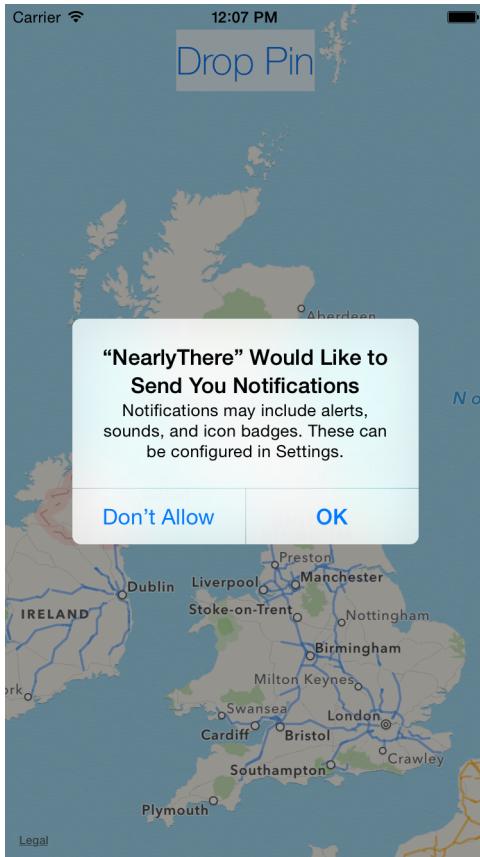
Since you're going to requesting location-based notifications, you need to ask the user for permission to send them notifications and to use their location. Both of these have been covered in iOS 8 Day-by-Day (notifications on day 25 and location on day 34), but it's worth having a brief summary.

User notifications are defined as sounds, alerts and badges that can interrupt the user whilst your app isn't in the foreground. These can be triggered both by remote and local (both temporal and spatial) notification events. To request permission from the user, you need to specify what types of notifications you'd like to use, and then register for them. Here, the only type requested is alert:

```
1 // Ask for permission for notifications
2 let notificationSettings =
3     UIUserNotificationSettings(forTypes: UIUserNotificationType.Alert,
4                                 categories: nil)
5 UIApplication.sharedApplication()
6         .registerUserNotificationSettings(notificationSettings)
```

This will present the user with the following alert:

⁹⁵<https://github.com/ShinobiControls/iOS8-day-by-day>



Notification Permission

If you don't have this permission, and attempt to schedule a local notification then you'll get an error in the console:

```
2014-10-20 12:03:39.059 NearlyThere[89924:9102136] Attempting to schedule a local notification <UIConcreteLocalNotification: 0x7f94c4c62d40>{region = CLCircularRegion (identifier:'Destination', center:<-55.38832900,-3.39586200>, radius:5000.00m), regionTriggersOnce = YES, user info = (null)} with an alert but haven't received permission from the user to display alerts
```

No Permission

Asking the user permission to use their location is a little more complicated - and is covered in depth in day 34. You need to create a `CLLocationManager()`, that will stay around for the lifetime of the view controller (i.e. a property):

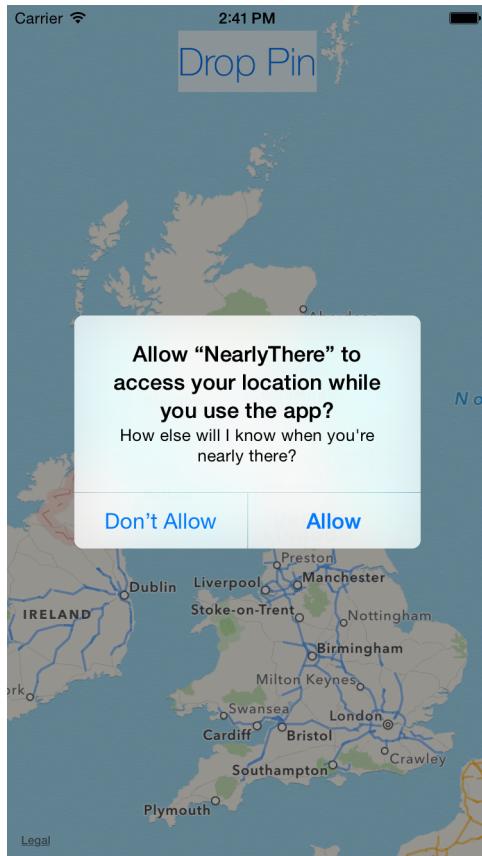
```
1 let locationManager = CLLocationManager()
```

And then request permission:

```
1 // Ask for permission for location
2 locationManager.delegate = self
3 locationManager.requestWhenInUseAuthorization()
```

Interesting, in order to use location notifications, your app doesn't need to request permission to use location in the background. This is because the geofencing is taken care of CoreLocation in the OS, and your app will only be away of the location when the user chooses to bring the app into the foreground - i.e. in response to a location.

You must also remember to set the `NSLocationWhenInUseUsageDescription` field in `Info.plist` to an appropriate string, as described in day 34. Once you've done all this, then the user will be presented with the following when they launch the app:



Location Permission

Note that there will be no errors if you don't get this right. So be sure to check it.

If you schedule a location notification whilst your app doesn't have permission to use the user's location, then the notification will not fire or be delivered to your app. And you will not be informed of this.

Creating Notifications

Location notifications are exactly the same as temporal notifications - i.e. they are local notifications. Therefore they are created as instances of `UILocalNotification`:

```
1 let notification = UILocalNotification()  
2 notification.alertBody = "You're nearly there!"
```

To specify that a notification is a location notification you need to provide a `CLRegion` object to the `region` property. `CLRegion` has two concrete subclasses - `CLCircularRegion` and `CLBeaconRegion`. This means that you can specify an alert region based both on absolute location on the planet, or relative to a given iBeacon.

The following specifies a circular region with a radius of 50m around the coordinate of the pin drop annotation:

```
1 notification.region = CLCircularRegion(center: annotation!.coordinate,  
2                                         radius: 50, identifier: "Destination")  
3 notification.regionTriggersOnce = true
```

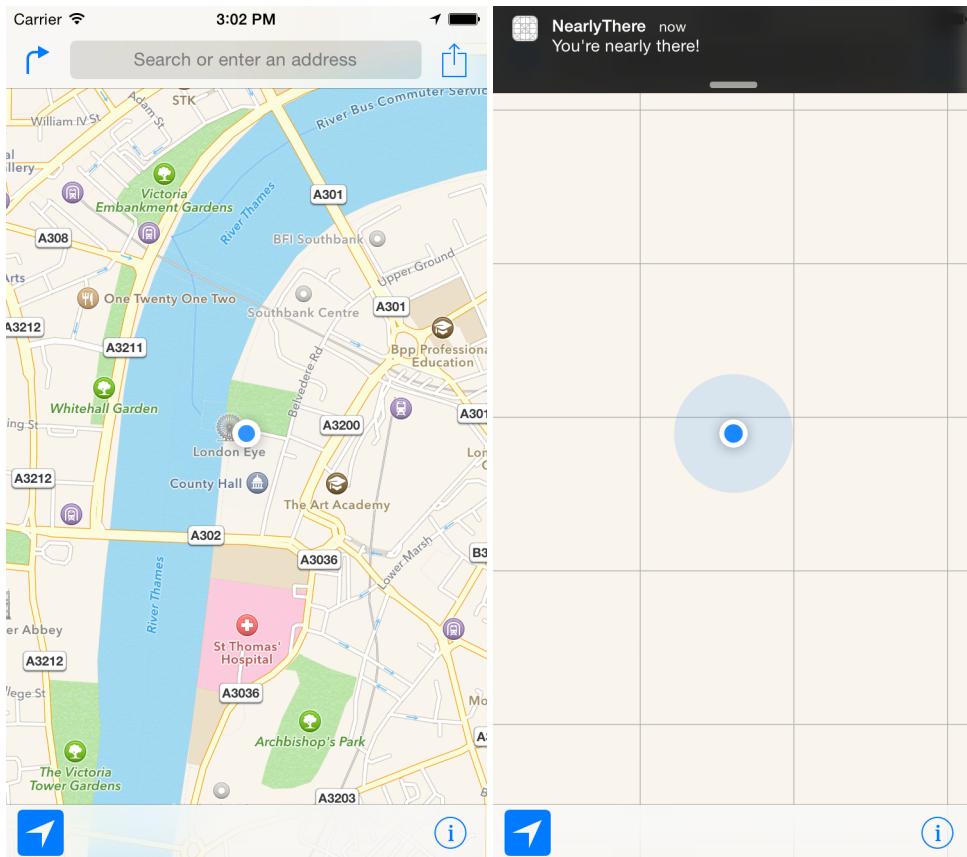
The `regionTriggersOnce` property allows you specify whether a notification should expire once it has triggered once. The alternative is for the notification to fire every time the user crosses the boundary into the region.

Once you've created the notification you need to schedule it, using the `scheduleLocalNotification()` method on `UIApplication`:

```
1 UIApplication.sharedApplication().scheduleLocalNotification(notification)
```

You remove notifications with the `cancelLocalNotification(_)` and `cancelAllLocalNotifications()` methods on `UIApplication`.

This notification should now fire as expected:



Responding to Notifications

As with all local notifications, the application delegate has a method which will be called when it fires - `application(_ didReceiveLocalNotification:)`. If the notification was triggered by a location notification, then it will have a non-nil `region` property:

```
1 func application(application: UIApplication,
2                   didReceiveLocalNotification notification: UILocalNotification) {
3     println("\(notification)")
4     if notification.region != nil {
5         println("It's a location notification!")
6     }
7 }
```

This method will be called irrespective of whether your app is currently in the foreground - but remember that if your app is in the foreground, then the system won't show any UI - that's left up to you.

If the app is in the background then the system will show an alert UI (in whatever form the user has configured for your app). If the user decided to tap on the event, then your app will be launched with an options dictionary which contains the `UIApplicationLaunchOptionsLocationKey` key:

```
1 func application(application: UIApplication, didFinishLaunchingWithOptions
2                     launchOptions: [NSObject: AnyObject]?) -> Bool {
3     // Override point for customization after application launch.
4     if launchOptions?[UIApplicationLaunchOptionsLocationKey] != nil {
5         println("It's a location event")
6     }
7     return true
8 }
```

Conclusion

This functionality has been available in iOS since background location monitoring was possible, but the process to wire it all together was a but arduous. This new local notification functionality greatly improves the approach for geo-fencing like use cases.

The source code for today's app is available on github at github.com/ShinobiControls/iOS8-day-by-day⁹⁶. Take a look and fix any problems in a pull-request - it'd be much appreciated.

⁹⁶<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 37 :: Autosizing Collection View Cells



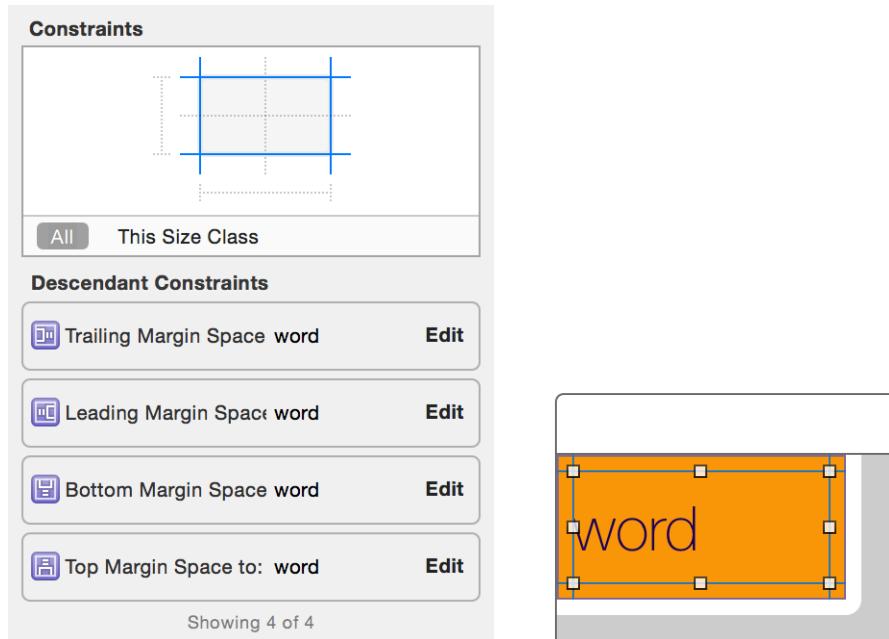
Way back in chapter 5, we took a look at the new functionality within table views that allows cells to define their own height - using the power of auto layout. Well, imagine a world where you could extend that same principle to collection views. How cool would that be?

Well, in iOS 8, you can! This functionality is built in to the flow layout, and is easy to access when building your own layouts. In today's brief post you'll discover how to use auto-sizing cells within a flow layout, and a little bit about the underlying implementation. The sample app is available in the iO8 Day-by-Day repo on github at github.com/ShinobiControls/iOS8-day-by-day⁹⁷.

Enabling Sizing in a Flow Layout

When using a `UICollectionViewFlowLayout`, you hardly need to do anything in order to enable cell auto-sizing. The key thing, as in autosizing `UITableViewCell`s, is to ensure that the size is completely defined using auto layout constraints.

For example, in this custom cell, the label has constraints that link it to every side of the cell:



⁹⁷<https://github.com/ShinobiControls/iOS8-day-by-day>

The other thing that you need to do to enable autosizing is specify an estimated item size on the flow layout. This is new to iOS 8 and plays a similar role to its counterpart in UITableView.

To size cells in the past you had two options; one was to set the `itemSize` on UICollectionViewFlowLayout - applying the same size to every cell within the collection view. For more fine-grained control on the cell sizes you could implement the `collectionView(_:, layout:, sizeForItemAtIndexPath:)` method on UICollectionViewDelegateFlowLayout, however the responsibility for calculating the size for each item is down to you - rather than the layout engine built into UIKit.

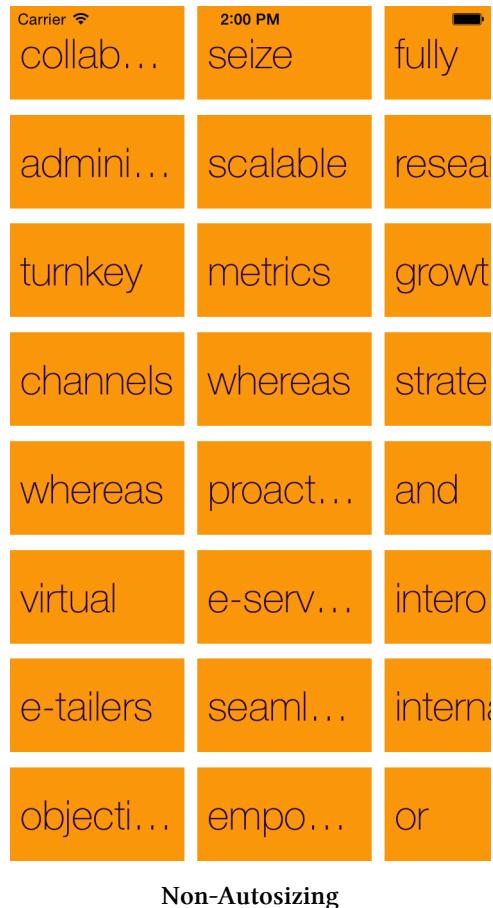
The `estimatedItemSize` property has a default of `CGSizeZero`, but setting it to a non-zero size will enable the auto-sizing of cells.

The following code inside a UICollectionViewController subclass will enable auto-sizing for it's flow layout.

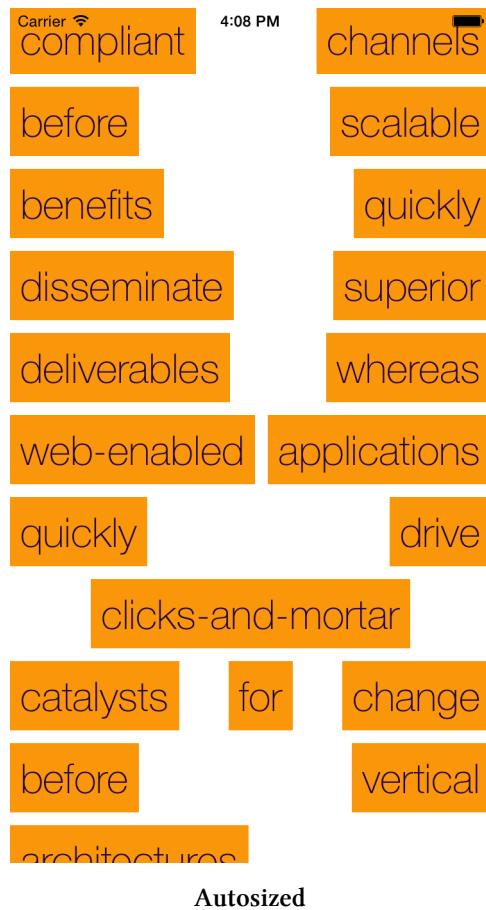
```
1 override func viewDidLoad() {
2     super.viewDidLoad()
3     // Do any additional setup after loading the view, typically from a nib.
4     if let cvl = collectionViewLayout as? UICollectionViewFlowLayout {
5         cvl.estimatedItemSize = CGSize(width: 150, height: 75)
6     }
7 }
```

Note that since the name of this property includes *estimated*, you don't have to be hugely accurate in the value you specify. In fact, the cells will only be laid out and sized as they are about to arrive on screen. The estimated value is used to size the scroll bars appropriately.

Enabling this line will take a layout that looked like this:



and change its appearance to match this:



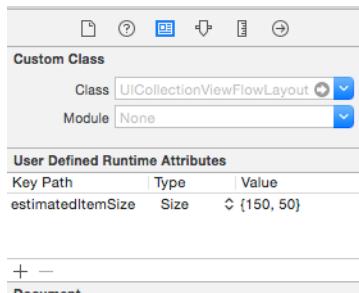
Autosizing via Interface Builder

It seems a shame to have to drop to code to set the estimated size - all other sizing properties associated with a flow layout can be configured within Interface Builder:



Sizing in IB

However, you'll notice that there is nowhere obvious to configure the `estimatedItemSize` property. Hopefully in future updates to Xcode this issue will be addressed, but until that point, it is still possible to set the value via the **User Defined Runtime Attributes** panel inside the **Identity Inspector** for the `UICollectionViewFlowLayout`.



Setting Estimated Size

This panel allows you to set properties on objects via key-value coding - which is essentially what the specialized panels in IB do anyway.

Mechanics of Autosizing

In order to allow a cell (or indeed any reusable item in a collection view) to determine its own size, a new method has been added to `UICollectionViewReusableView -preferredLayoutAttributesFittingAttributes(_ :)`. This method gives a cell the opportunity to return the layout attributes it would like to be displayed with, given the ones the layout has provided.

The default implementation of this in `UICollectionViewCell` just changes the `size` property to match the size determined via autolayout. In the flow layout, the result from this method is only used if the `estimatedItemSize` property is non-zero, but you could use it in whatever way you would like in your own custom layouts.

Importantly, the default implementation does all the auto-layout magic for you. If you wish to alter other attributes you can do that, but you don't have to get involved with any of the autolayout unless you want to.

Conclusion

This new feature has a fairly small API - just two new items across the whole of the collection view classes. In the past achieving the same effect was possible, but it was a lot of work, and also pulled the sizing of elements out into a different place - effectively duplicating the same functionality.

Since the cells are likely to be fairly simple, and they have to be self-contained, the autolayout engine will not add much complexity to the collection view layout process - so you can be confident that you won't be crippling your app.

The sample app that accompanies today's article is available as part of the iOS8 day-by-day repo on github at [github.com/ShinobiControls/iOS8-day-by-day⁹⁸](https://github.com/ShinobiControls/iOS8-day-by-day). It's pretty simple - I wanted to demo some more advanced uses with a custom layout, but didn't get around to it. You should fork the repo and add a cool thing that demos this. I might even send you some stickers/pens/t-shirt type stuff...

⁹⁸<https://github.com/ShinobiControls/iOS8-day-by-day>

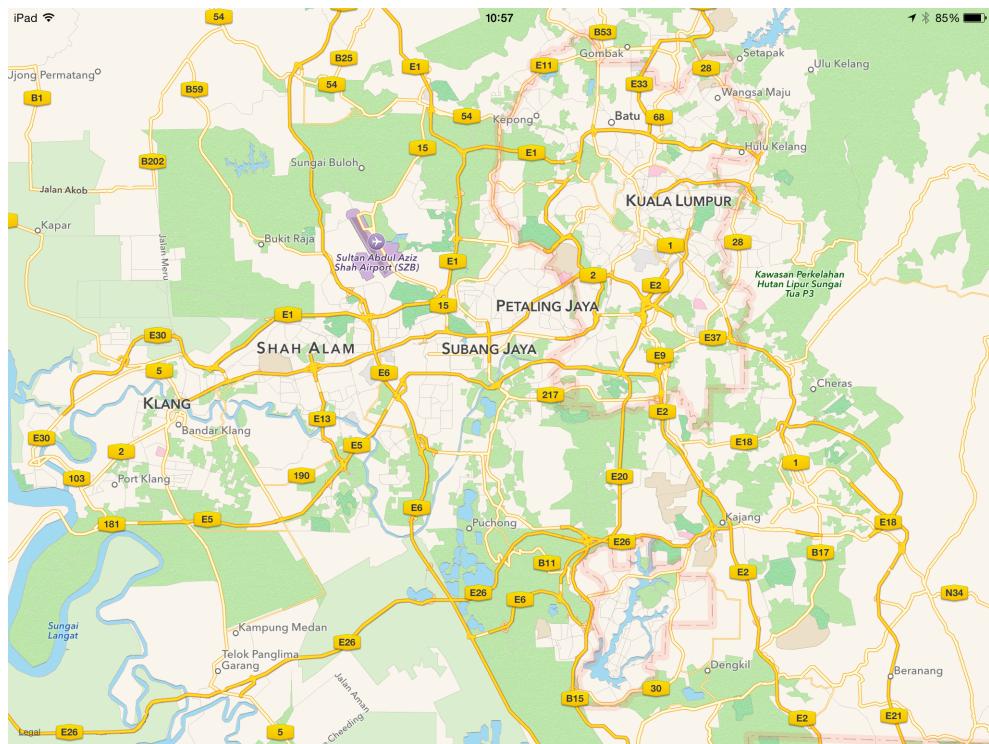
Day 38 :: Handoff



One of the key new concepts introduced at WWDC in 2014 was that of Continuity, describing the ability to seamlessly switch between devices, and continue your task. The underlying technology behind Continuity is called Handoff, and it works across both iOS and Yosemite.

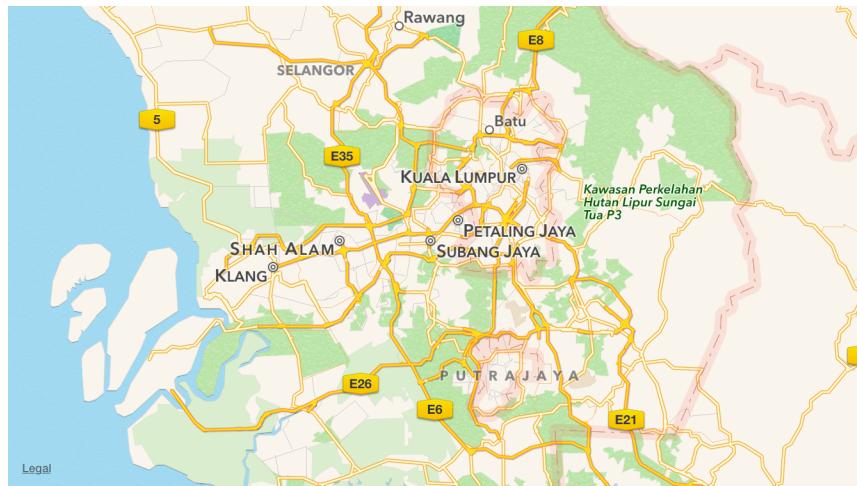
Today's article introduces just one of the vectors through which Handoff can work - native iOS app to native iOS app. The app which accompanies the post is called **MapOff** and is a simple map view. When you open the app via Handoff on a different device, then the currently visible region on the first map will be transferred to the receiving device.

As ever, the code for today's app is available on github at github.com/ShinobiControls/iOS8-day-by-day⁹⁹.



iPad Map

⁹⁹ <https://github.com/ShinobiControls/iOS8-day-by-day>



iPhone Map

Handoff Logistics

Handoff is based around user activities. These are objects which encapsulate the current state of what the user is doing, or at least enough to recreate the ‘session’ elsewhere. An activity has a type associated with it, which allows the operating system to determine whether it has an app which can resume a given activity.

Handoff works by first using bluetooth LE to discover devices in the locality that are signed in to the same iCloud account. If the current app is handoff enabled, and it can find a device that is able to resume the current activity, then the inactive device will show an icon notifying the user that they can Handoff from one device to the other.

When the user chooses to continue this activity, the sending device will transfer the activity object to the continuing device, allowing it to continue the same activity.

As a developer, you can either integrate with Handoff manually, or, if you have a document-based app, you can rely on the deep integration with `UIDocument`. This article will take a look at the manual approach - for further details on integration with `UIDocument`, take a look at the [Handoff programming guide¹⁰⁰](#).

Preparing an App for Handoff

In order to Handoff from an app, it needs to prepare and maintain one or more `NSUserActivity` objects. These encapsulate what the user is currently doing, allowing it to be resumed on another device. iOS 8 introduces some new features on `UIResponder` for handling user activities - one of

¹⁰⁰https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/Handoff/HandoffFundamentals/HandoffFundamentals.html#/apple_ref/doc/uid/TP40014338-CH3-SW1

which is a `userActivity` property. Since `UIViewController` is a subclass of `UIResponder`, you can use the new features to ease the Handoff workflow.

In `MapOff`, the user activity is created and configured in `viewDidLoad()`:

```
1 let activityType = "com.shinobicontrols.MapOff.viewport"
2
3 override func viewDidLoad() {
4     super.viewDidLoad()
5     if userActivity?.activityType != activityType {
6         userActivity?.invalidate()
7         userActivity = NSUserActivity(activityType: activityType)
8     }
9     userActivity?.needsSave = true
10 ...
11 }
```

A user activity object has an `activityType` property, which allows the system to determine whether there is an app that can continue the activity. This is a string, and should be in reverse-DNS form - as shown above.

You'll also notice that you set the `needsSave` property to `true`. Every time the user interacts with your app, and you need to update the 'resume instructions', you need to repeat this. This property allows the system to lazy batch updates rather than with every user interaction. When this property is set to `true`, the system will periodically call `updateUserActivityState()` on your `UIViewController` subclass. This gives you an opportunity to save the state.

In `MapOff`, this method is used to get the currently visible range from the map, and save it into the `userInfo` dictionary:

```
1 override func updateUserActivityState(activity: NSUserActivity) {
2     let regionData = NSData(bytes: &mapView.region,
3                             length: sizeof(MKCoordinateRegion))
4     activity.userInfo = ["region" : regionData]
5 }
```

The `userInfo` dictionary can use simple types such as `NSString`, `NSNumber`, `NSData`, `NSURL` etc. Since these types don't include the `MKCoordinateRegion` struct, it's necessary to pack it into an `NSData` object. Note that you might expect to use some kind of archiver, but `MKCoordinateRegion` is a pure Swift struct, so doesn't implement the `NSCoding` protocol. Since it is a struct value-type (i.e. doesn't contain references to non-value objects) it's possible to use `NSData` to copy the bytes. **This won't always be the case**, so be careful.

If you've got lots of data to transfer then the `userInfo` dictionary isn't the place to do it - in fact Apple recommends that the the user activity object should be kept below 3kb. However, user activities

provide functionality for setting up and consuming streams between the devices - at which point you can send whatever you wish.

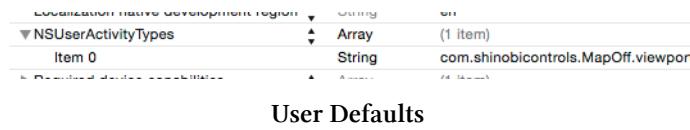
As mentioned before, the `needsSave` property should be set to `true` every time the state of the UI updates. In `MapOff` this occurs every time the user pans or zooms the map - i.e. the region changes. Implementing the following delegate method has the desired effect:

```
1 // MARK:- MKMapViewDelegate
2 func mapView(mapView: MKMapView!, regionDidChangeAnimated animated: Bool) {
3     userActivity?.needsSave = true
4 }
```

At this point, you've implemented everything you need to for an app to advertise that it supports Handoff, but you also need an app that the user can resume their activity on.

Resuming an Activity

When the system can see that a device is advertising the availability of Handoff in the local area, it has to decide whether or not it has an app that can handle it. Apps can use the `NSUserActivityTypes` key in `Info.plist` to specify which activities it is capable of handing off. This is an array of strings, and should include all the reverse-DNS activity types that you require:



Note that you can only Handoff between apps which are signed by the same developer.

Once you've done this, then when a user chooses to continue an activity, your app will be started, and two new methods on your application delegate will be called:

```
1 func application(application: UIApplication,
2                   willContinueUserActivityWithType userActivityType: String) -> Bool {
3     println("Will continue \(userActivityType)")
4     return true
5 }
6
7 func application(application: UIApplication,
8                   continueUserActivity userActivity: NSUserActivity,
9                   restorationHandler: ([AnyObject]!) -> Void) -> Bool {
10    if let rootVC = window?.rootViewController {
11        restorationHandler([rootVC])
12    }
}
```

```
13     return true  
14 }
```

The first of these, `application(_:, willContinueUserActivityWithType:)` notifies you that the activity is going to continue. At this stage, only the type of activity is available to you, not the activity itself. This is because it might take some time for the activity to be transferred between the two devices. At this stage you can prepare your app to continue the specified activity - which is especially useful if your app can continue a variety of different activities. This could involve loading a specific view controller, or navigating to a particular area of your app.

Once the `NSUserActivity` has arrived on the receiving device, the second delegate method will be called: `application(_:, continueUserActivity:, restorationHandler:)`. At this point you can extract the `userInfo` dictionary from the `userActivity` and use the supplied information to update the UI appropriately.

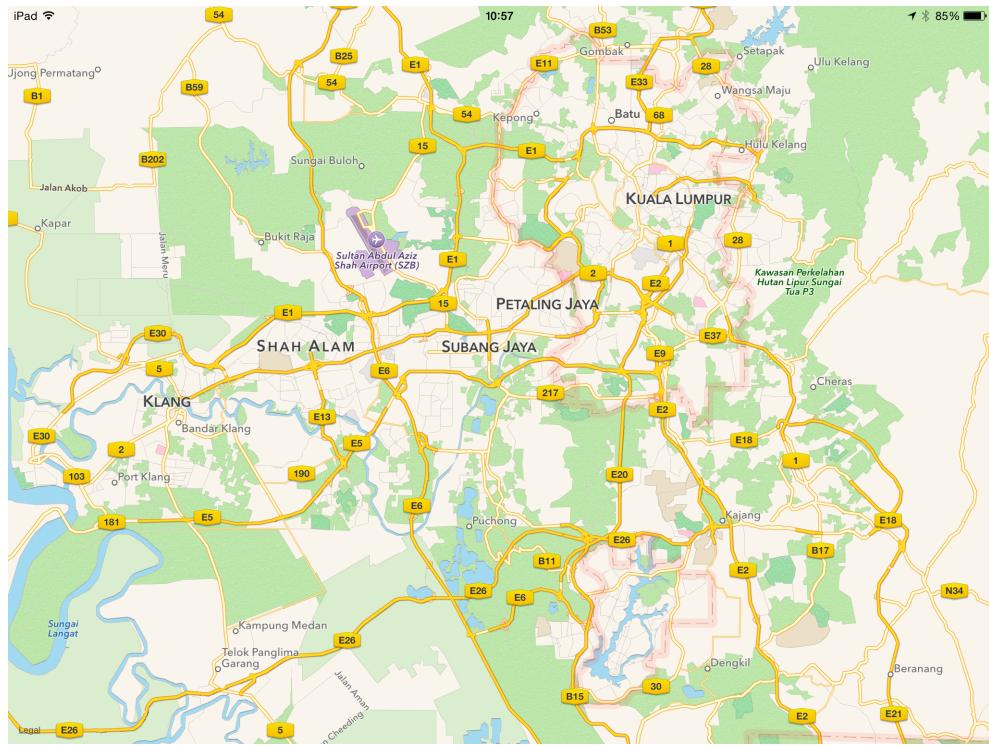
The method also supplies a `restorationHandler` closure. This takes an array of `UIResponder` objects, and calls the `restoreActivityState(_)` method on each of them. Since `MapOff` is centered around a single view controller, this is what is used there.

Dropping back to the view controller, the implementation of this method is as follows:

```
1 override func restoreUserActivityState(activity: NSUserActivity) {  
2     if activity.activityType == "com.shinobicontrols.MapOff.viewport" {  
3         // Extract the data  
4         let regionData = activity.userInfo!["region"] as NSData  
5         // Need an empty coordinate region to populate  
6         var region = MKCoordinateRegion(  
7             center: CLLocationCoordinate2D(latitude: 0.0, longitude: 0.0),  
8             span: MKCoordinateSpan(latitudeDelta: 0.0, longitudeDelta: 0.0))  
9         regionData.getBytes(&region, length: sizeof(MKCoordinateRegion))  
10        mapView.setRegion(region, animated: true)  
11    }  
12 }
```

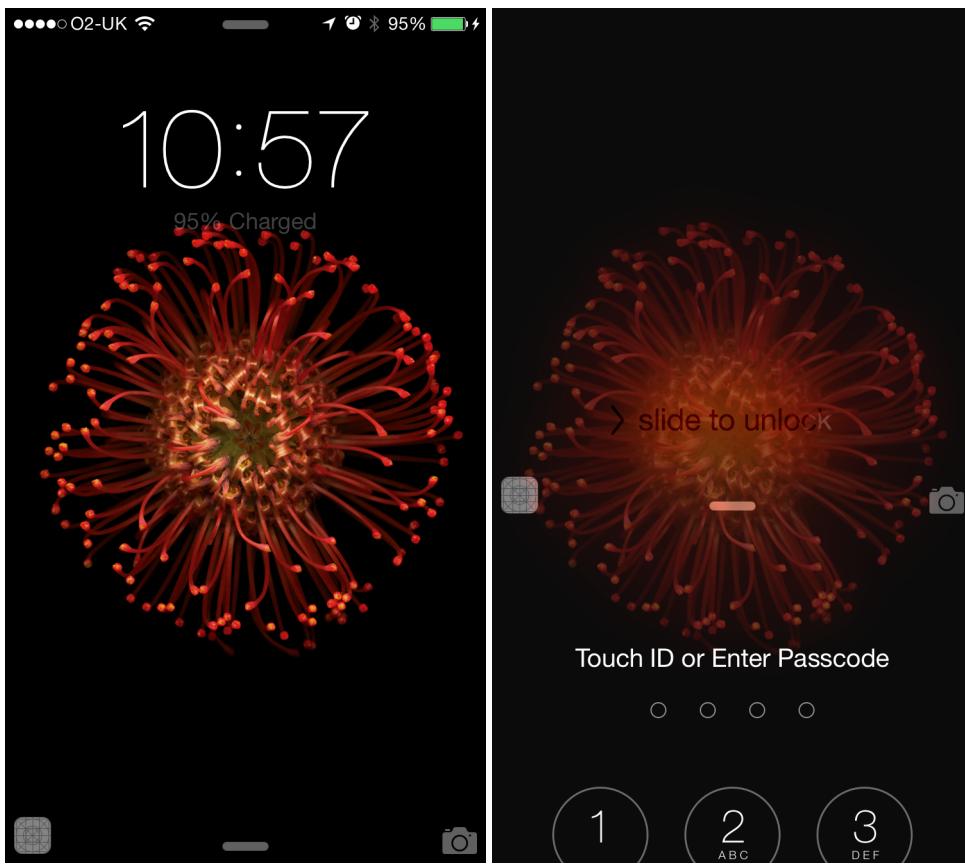
This implementation checks that it is responding to the correct activity type, before extracting the data from the `userInfo` dictionary. Note once again that this process is a little more complex than you might be expecting, since `MKCoordinateRegion` doesn't implement `NSCoder`. Once the region has been extracted, the map view is updated to show the same region.

Starting with the following view on an iPad:

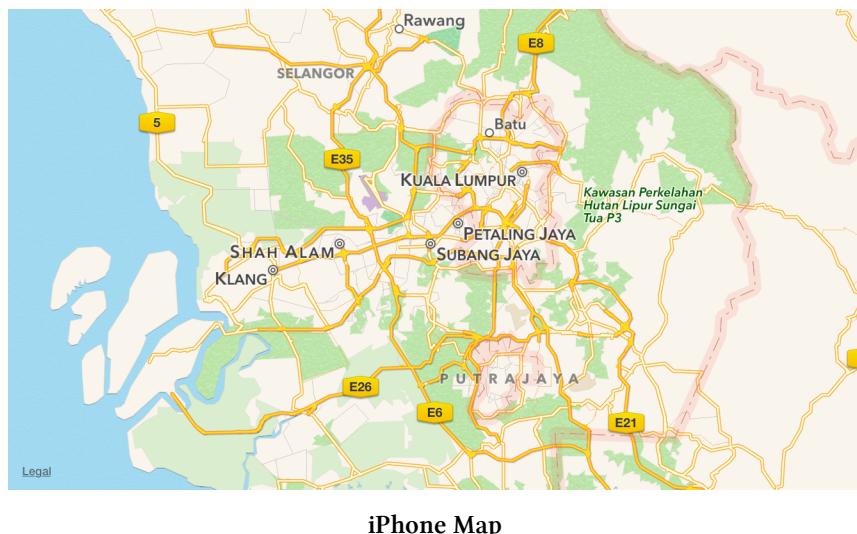


iPad Map

Since **MapOff** doesn't have a nice icon, the generic icon appears on the lock screen to signify that Handoff is available. Dragging this up will invoke the Handoff:



This starts MapOff on the iPhone, and passes it the user activity. This causes the map to display the same range:



iPhone Map

Conclusion

Handoff is extremely powerful, and this article only just scratches the surface. In addition to being able to Handoff between different the same app on different iOS devices, you can also Handoff between OSX apps and iOS apps. You can even Handoff between native apps and webpages - in both directions! This means that you could visit a website on your Mac, and then Handoff to an iOS app on your iPhone - I really can't wait for more websites to start implementing this kind of functionality.

Handoff is one of the coolest things to be introduced in iOS 8. It feels to me like a window into the future of personal computing, and goes some way to solving a common problem. It's a very "Apple" problem and solution: "Why should it matter which of my devices I'm using?". That's a technology problem, and this is a good step towards a solution. I'll be really interested to see how different apps use this, and whether the integration within the OS is good enough to drive significant adoption and changes in my behavior.

The code for today's sample project, **MapOff** is available to clone, fork or download from the iOS 8 Day-by-Day repo on github at github.com/ShinobiControls/iOS8-day-by-day¹⁰¹.

¹⁰¹<https://github.com/ShinobiControls/iOS8-day-by-day>

Day 39 :: WatchKit

8 39

Although the Apple Watch was unveiled back in September 2014, developers have only recently got hold of an SDK - hidden away inside iOS 8.2. It seems that the Apple Watch has really caught the imagination of iOS developers from across the world, and as such there have been loads of in-depth articles, opinion pieces and videos created covering it. iOS 8 Day-by-Day has always been intended to be short overview of new technologies so that intermediate developers can hit the ground running. Today's post on Apple Watch is no different. If you want to learn everything there is to know about the SDK then read the documentation, and then soak up the excellent content that's been created in the short time the SDK has been alive.

This article will discuss what's different between an iOS app, and an Apple Watch app, what we can do as developers, and a very quick guide to getting started. The app that accompanies this post is called **NightWatch** and provides the wearer with instant access to some of the best quotes from this excellent movie. (Well, I say excellent, I've never seen it, and hadn't actually heard of it until I started this project.) The source code for the app is available on github at github.com/ShinobiControls/iOS8-day-by-day¹⁰².

What can (and can't) I do on a watch?

Apple has made it very clear that the Apple Watch SDK in iOS 8.2 is the first implementation, and that we can expect changes in coming iOS releases. This might go some way to explaining what you might initially think is a fairly restrictive API:

1. You cannot write code to run on the watch

When you add a watch target to an existing project, it creates both a watch app, and an extension. The watch app itself cannot contain code - it only contains image assets and a storyboard. This is what will be pushed to the watch. The extension runs on an iPhone and interacts with the layout that you created in the aforementioned storyboard.

This actually closely simulates the impression that you're running code on the watch, but it's important to be aware of the effect that this "remote display" will have on the functionality and experience you wish to implement.

¹⁰²<https://github.com/ShinobiControls/iOS8-day-by-day>

2. There are three “tasks” you can customize on a watch

In addition to creating a full watch-app, you can also create glances and notifications. Notifications are exactly the same as existing notifications on iOS - with the added ability to customize their appearance. A notification appears full-screen and has different visualizations depending on the battery level and the user’s response. Users can interact with notifications in the same way that they can with the new notification interactions introduced to iOS 8.

A glance is an entirely new concept to the watch. The user will be able to swipe through a collection of glances, and as such they cannot interact with them. They’re designed to show you the most important info in a short amount of time, i.e. a glance.

The app itself is a lot more heavyweight - allowing the user to fully interact with the UI components. Don’t be fooled into thinking it’s just a little phone app though - there are restrictions on what is achievable, along with the completely different user interaction.

3. The API is heavily focused around energy efficiency for the watch

Something that might seem strange to you when you start working with outlets in storyboards, is that you cannot read properties from any of them - they are write only. This is because the code is running on the iPhone, with the UI update instructions being sent over bluetooth to the watch. This is an expensive operation (in terms of power), so all unnecessary communications are eliminated.

You should also note that setting properties on UI controls is not an instantaneous procedure. Instead, the operations are batched up and pushed to the watch in one go.

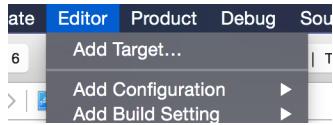
4. You cannot get access to any of the watch’s sensors

This is a little disappointing, since there are going to be some great ideas for apps using these new sensors. I think that this is a side effect of the lack of native apps on the watch in the initial release, and that some kind of interaction with the sensors will arrive in due course. This means that your watch apps aren’t able to do anything that your iPhone apps couldn’t do. In fact, they’re very much meant to augment your iPhone app, for quick access to relevant summary info.

Getting Started

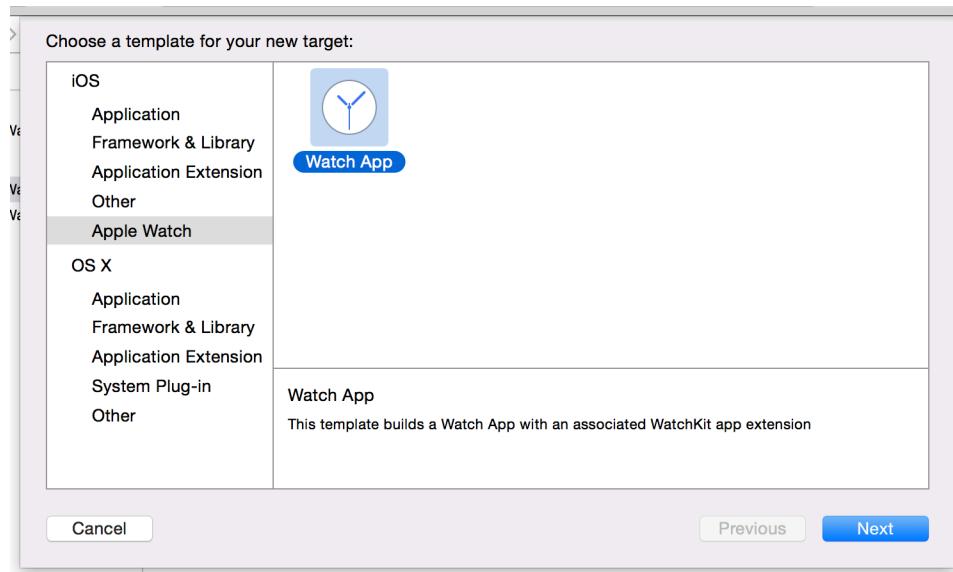
Watch apps can’t exist on their own - they are packaged up as part of an iPhone app. This reinforces the notion that Apple Watch apps are designed to supplement the functionality provided by an iPhone app, and are not meant to be standalone.

As such, creating a watch app requires that you have an existing iPhone app, to which you can add a target:



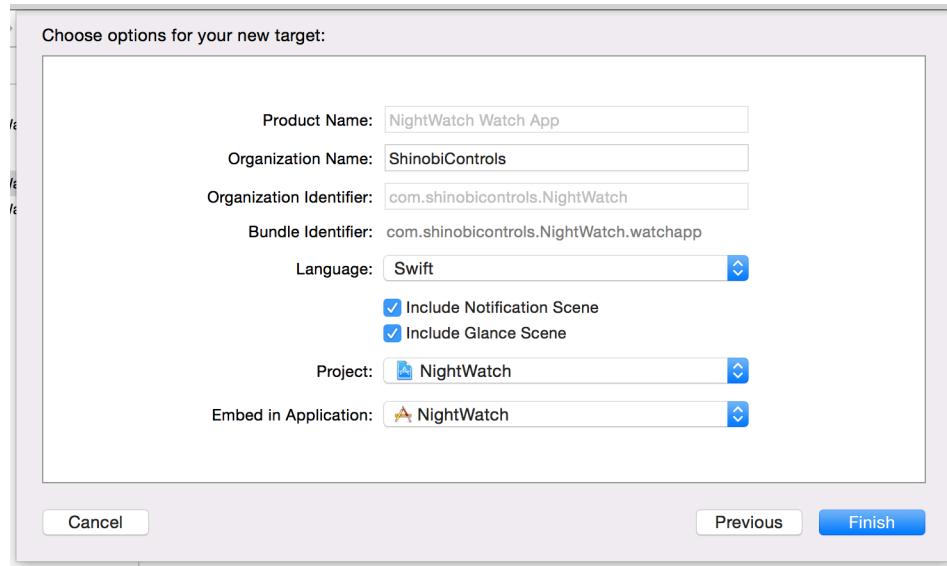
Add Target

There is a new section called **Apple Watch**, from which you can select **Watch App**:



Add Watch App

The options screen allows you to specify whether you want a glance and notification to be created alongside the app itself:



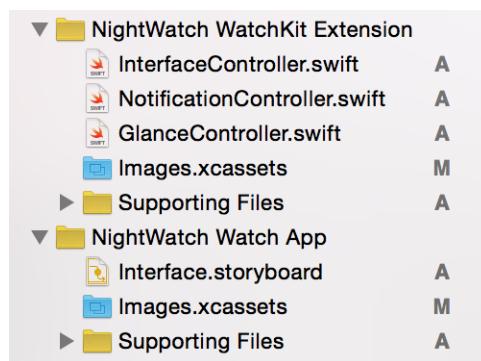
Watch Creation

This will create two new targets - one for the watch app, and one for the extension:



Extension and App

As mentioned before, the app itself runs on the phone, and doesn't currently allow any custom code. It contains static assets - e.g. images and a storyboard for the layout. The watch extension runs on the paired iPhone, and references the storyboard that exists on the watch:



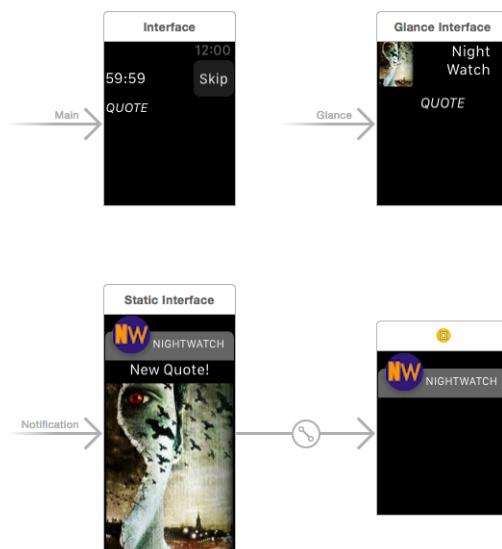
Project Layout

In the rest of this section you'll learn about building a watch app, glances and notifications.

Watch App

The watch app itself is made up of two parts - the UI is created entirely in a storyboard which exists on the watch itself, and the code that controls this interface runs as an extension on the iPhone. This means that the code that is used to power a watch app is subject to the same restrictions that extensions are - e.g. it can't run any background tasks.

The storyboard in the watch app contains scenes for the three different interfaces:



Storyboard Overview

You can see that the scenes are sized appropriately for the watch. Since the watch is so small, and low-powered, the layout doesn't use auto layout, but instead relies on a system of groups and simple metrics. Size classes are also not supported, but you can use similar techniques in IB to provide slightly different layouts for the two watch sizes (38mm and 42mm):



As you look through the object library lots of it will look familiar from traditional iOS apps. The appearance of the standard controls has been configured to work well on a watch, and there is actually only a subset of the controls you might expect. Some of the important differences are as follows:

- The only possibility for creating animations is through an animated image - which can be a collection of frames named correctly.
- A table doesn't have a datasource, but instead works by adding cells individually.

- Since updating UI is expensive, there are specialized label subclasses for displaying a day and a timer. These autoupdate on the watch, and don't require interaction with the watch app extension.
- Maps are not interactive, but are rendered on the phone and then pushed as an image to the watch.
- A group has no visual appearance, but instead acts as a container for other display elements. It arranges them in a line, either vertically or horizontally. This is a very powerful control which allows you to construct a whole host of different layouts.

Watch apps can have one of two forms of navigation - either page based or hierarchy based. You cannot mix between these, and they are formed through collections of `WKInterfaceController` objects, corresponding to the interface controllers in the storyboard.

Watch extension code

A layout isn't useful on its own - it needs to be able to dynamically update to show different information. Since the watch app cannot run any code, this responsibility is left to the watch kit extension.

If you take a look inside `InterfaceController.swift`, you'll see that `InterfaceController` is a subclass of `WKInterfaceController`. This is roughly equivalent to `UIViewController` in iOS, with the remote display magic built in.

A `WKInterfaceController` has four methods that form the core of its lifecycle:

- `init()` Standard for all objects. You can create data objects and things here.
- `awakeWithContext(_:)` You can use the context object whenever you are instantiating a new interface controller, to provide it with data or navigation info. There is no restriction on the type of context you send. Should prepare the interface controls.
- `willActivate()` Called just before the interface appears on screen - this is roughly equivalent to `viewWillAppear()` in `UIViewController`
- `willDeactivate()` Called as the interface is disappearing. You can use it to tidy up.

Interfacing with the layout in the watch app story board is achieved in the same way that it is in an iPhone app - via IBOutlets and IBActions. You can drag from IB into your `WKInterfaceController` as expected to create outlets and actions:

```
1 @IBOutlet weak var quoteLabel: WKInterfaceLabel!
2 @IBOutlet weak var quoteChangeTimer: WKInterfaceTimer!
3
4 @IBAction func handleSkipButtonPressed() {
5     ...
6 }
```

Notice here the types of `@IBOutlet` properties - `WKInterfaceLabel` instead of `UILabel`, and `WKInterfaceTimer`. These new types represent the difference in functionality between iPhone and

Apple Watch - where communicating with the user interface is an expensive operation. These `WKInterface*` types are read only - i.e. although you can set values, you cannot read them back. For example, setting the text on a label looks like this:

```
1 quoteLabel.setText(quotes[currentQuote])
```

`WKInterfaceTimer` is a specialized interface object that will count up or down to or from a specified `NSDate`. Since this is also read only, you can interact with it as follows:

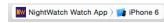
```
1 quoteChangeTimer.setDate(NSDate(timeIntervalSinceNow: quoteCycleTime))
2 quoteChangeTimer.start()
```

Check the documentation for further details on these and other interface object controls that are available as part of the Apple Watch SDK.

Some of the controls allow user interaction - such as buttons or switches. These can be wired up to IBActions, in the way you'd expect. Note that you won't receive a reference to the button in question - which therefore means you will probably have to have a different method for each of the button actions.

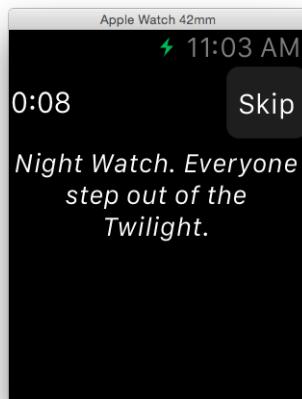
Running A Watch App

You can run the watch app in a simulator by selecting the correct scheme from the selection menu:



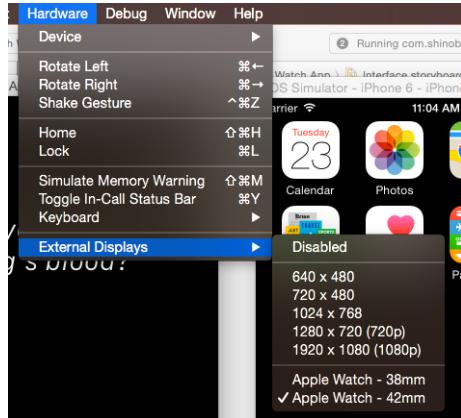
Scheme Selector

This will start the specified iOS simulator, and it should show your app in a secondary display window representing the watch:



Watch Window

If it doesn't, or to switch between different watch sizes, you can use the **External Displays** list in the **Hardware** menu:



Hardware menu

You can interact with this external display as you'd expect to be able to - so you can put your new watch app through its paces even though you are not yet able to buy an Apple Watch.

Glance

When you created the watch target, you had the option of creating a glance. If you chose to, then inside the watch storyboard, you'll be able to see a layout for the glance:



Glance Storyboard

You can use pretty much the same techniques for designing your glance layout as you did for the full app - with one exception. Users cannot interact with a glance - tapping a glance will open the associated watch app.

A glance interface has its counterpart class in the WatchKit extension, in the same way that the watch app does. It is again a subclass of `WKInterfaceController`, and therefore has the same lifecycle.

The simple example in the accompanying project just shows a random quote from the *NightWatch* film:

```

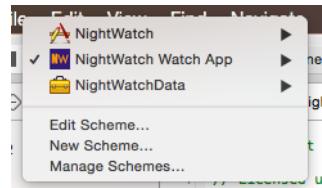
1 class GlanceController: WKInterfaceController {
2     @IBOutlet weak var quoteLabel: WKInterfaceLabel!
3     let quoteGenerator = NightWatchQuotes()
4
5     override func awakeWithContext(context: AnyObject!) {
6         quoteLabel.setText(quoteGenerator.randomQuote())
7     }
8 }
```

You should use the `awakeWithContext(_:)` method to prepare and interface objects, and the `willActivate()` method to do more time-critical operations such as starting timers.

Running a Glance

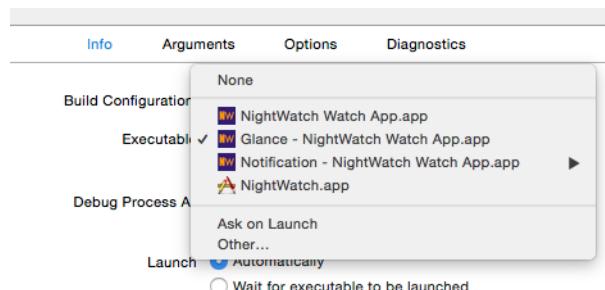
Since the ‘external display’ approach of simulating an Apple Watch doesn’t give you a full simulator to play with, it’s not immediately obvious how you can run your glance. In fact, your watch target creates three different executables - one for the app itself, one for the glance and one for the notification. The scheme it creates defaults to using the watch app. To test the glance, you need to create a new scheme which will run the glance executable.

Use the scheme selector to edit the current watch app scheme:



Scheme Selector

From this panel you can duplicate the watch app scheme, and call it something like **Glance - WatchApp**. Then, using the **Run** tab, you can change the **Executable** to **Glance - NightWatch Watch App.app**:



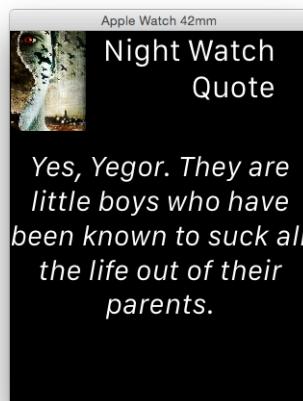
Select Executable

This creates a new scheme, which you can select and run:



Run Glance

The resultant app now runs in the external display window, as expected:



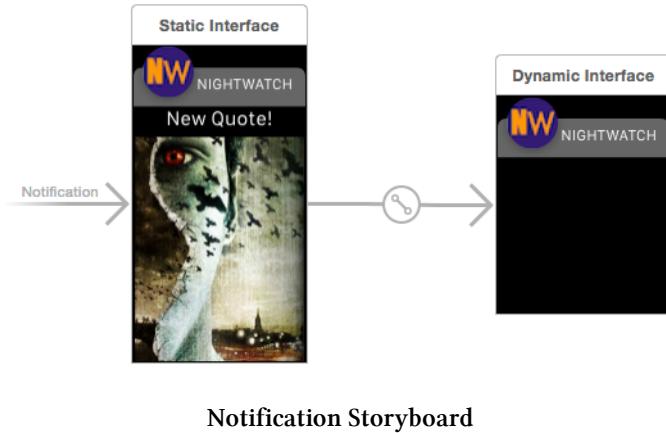
Glance

Notifications

Out of everything possible use of a watch, you'd imagine that notifications would be quite significant. It makes perfect sense to grab a quick look at your wrist to check whether you need to deal with a new notification, rather than digging around in your pocket for your phone.

You'll also be aware that iOS has a pretty comprehensive notifications system - involving both remote and local notifications. It won't come as a surprise that notifications on the Apple Watch 'plug-in' to the same system.

If you selected to create a notification when you added the watch target, then you'll be able to see the notification layout in the storyboard:



Notification Storyboard

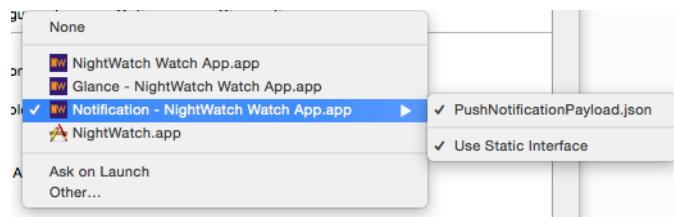
This looks slightly different to the interface controllers you've seen for the app and the glance - in that it appears to contain a segue. A notification can appear in three different forms - the short notification just shows the app icon and the title of the notification. If the user makes an appropriate gesture, then the short notification will be replaced with a longer version, containing more info. There are two forms of this long notification - static and dynamic.

The static version allows you to provide a static layout, which will have the title of the notification inserted into a label. You have a lot more control over the dynamic version - giving you access to the same level of control as you had with the glance. You can specify that the system should only use the static version, or you can provide a dynamic version. Note that the system might decide to use the static version even if there is a dynamic version available if it is low on power.

The dynamic version of the notification has a class backing it in the WatchKit extension - a subclass of `WKUserNotificationInterfaceController`. This provides some notification-specific methods in addition to the lifecycle methods from `WKInterfaceController` - in the form of `didReceiveLocalNotification(_:, withCompletionHandler:)` and `didReceiveRemoteNotification(_:, withCompletionHandler:)`. These can be used to prepare the content for the dynamic layout.

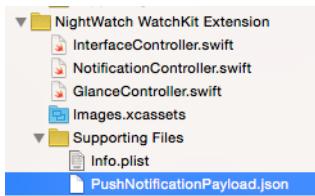
Testing Notifications

In the same way as you did for the glance target, you'll need to create a new scheme to associate with the notification executable. Notice this time that you can select a payload JSON file:



Selecting Payload Notification

This is a file that the template created, and can be found inside **Supporting Files**:



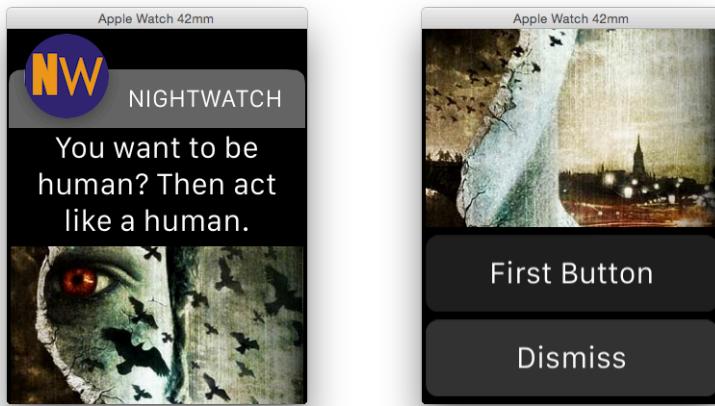
Payload Notification

This is just a JSON file which allows you to simulate what notification would have been received to drive the notification's appearance:

```
1  {
2      "aps": {
3          "alert": "You want to be human? Then act like a human.",
4          "title": "New Quote!",
5          "category": "myCategory"
6      },
7
8      "WatchKit Simulator Actions": [
9          {
10             "title": "First Button",
11             "identifier": "firstButtonAction"
12         }
13     ],
14
15     "customKey": "Use this file to define a testing payload."
16 }
```

You can provide the title and category as expected, and also provide the actions that will be turned into buttons at the bottom of the notification.

Once you've set this up, you can run the notification scheme to see how your notification executable behaves:



See the alert text appearing on the correct label and the introduction of an alert action button as specified in the notification payload file. The dismiss button is always provided by the system.

Code sharing

Your watch app has to be part of an iPhone app, and representing the same kind of functionality. This is likely to mean that you'll have code that you'd like to use both in the phone app and in the WatchKit extension. There are a few options for tackling this problem:

- **Copy/Paste** Yeah, we know it's the root of all evil, but you might find it easier to start with this approach and then refactor your code to something nicer later on.
- **Framework** iOS 8 introduced the ability to create dynamic frameworks - i.e. places where shared code and functionality can reside, and be accessed from other frameworks (i.e. apps and extensions) from within the same application bundle. This is actually the approach taken in **NightWatch**. The **NightWatchData** framework provides a **NightWatchQuotes** class that can then be used from both the WatchKit extension and the iPhone app.
- **Host in iPhone app** It's possible for the WatchKit extension to request that the iPhone app is opened, and then for the iPhone app to reply with some kind of data. This has multiple uses - one of them is for an action on the watch to trigger something on the phone, or also for the WatchKit extension to request that the iPhone app perform some kind of task. This might seem a little strange at first, but remember that there are restrictions on what an extension is allowed to do. There are no such restrictions on the container app, and therefore you could load it up in the background and use this inaccessible functionality. One example of this could be performing a background network request.

These techniques are likely to evolve as the SDK does, but it's good to know that there are alternatives to copy-paste right from the beginning.

Conclusion

The Apple Watch looks to be a really exciting new addition to Apple's hardware range. I'm really interested to see how well it gets adopted. How many people who don't currently wear a watch will suddenly need one?

This first release of WatchKit seems to be really well thought out. You don't get all the functionality you're used to from UIKit and wider iOS, but there's a lot there to get started with. You can create really quite powerful apps without a huge amount of difficulty. That being said, I'm looking forward to seeing how this SDK evolves. It'll be great to have access to the new sensors, and to see what level of access we, as developers, are allowed by Apple to the rest of the device. It's great to see that the entire API design has held power efficiency, and hence user experience, at the highest importance.

As ever, the code for today's post is available on the ShinobiControls github, at github.com/ShinobiControls/iOS8-day-by-day¹⁰³.

¹⁰³<https://github.com/ShinobiControls/iOS8-day-by-day>

Appendix

The road ahead

So where next? In completing this book, you now know everything there is to know about iOS 8 right? Unfortunately, that's not quite true. You've got a good grounding in the most important new technologies available, but by no means all of them. You can find out about some of the others by watching the WWDC videos, or reading the Apple documentation for iOS 8.

You'll also need to dig deeper into the features you wish to utilize in your apps, but once again, the Apple documentation is a good place to head. Don't forget that, particularly with newer technologies, there is often additional information available in the header files associated with the frameworks. It's always worth investigating those if you can't find the answers you need in the formal documentation.

The most important aspect to improving as an iOS developer (as in life) is to try stuff out. We learn by playing. Grab the source code from a chapter you like, and run it up. Then try and work out how to do something slightly different. Yeah, things will break, but that's how we learn. Luckily, in most of software, it doesn't matter that things break. In fact, we make a living out of it.

Learn by Playing

Useful resources

Some of the resources that you might find useful to expand your knowledge of iOS 8, and to take your apps to the next level:

- [shinobicontrols.com/iOS8DayByDay¹⁰⁴](http://shinobicontrols.com/iOS8DayByDay). The original blog post series from which this book was born.
- [github.com/ShinobiControls/iOS8-day-by-day¹⁰⁵](https://github.com/ShinobiControls/iOS8-day-by-day). **iOS 8 Day-by-Day Source Code** is all hosted on Github, so you can clone, download and fork it. Give it a go, and feel free to send across pull requests.
- [developer.apple.com/devcenter/ios¹⁰⁶](https://developer.apple.com/devcenter/ios). **Apple Documentation** is the canonical place to go to find out about all the technologies in iOS.

¹⁰⁴<http://shinobicontrols.com/iOS8DayByDay>

¹⁰⁵<https://github.com/ShinobiControls/iOS8-day-by-day>

¹⁰⁶<https://developer.apple.com/devcenter/ios/index.action>

- [developer.apple.com/videos/wwdc/2014¹⁰⁷](https://developer.apple.com/videos/wwdc/2014). **WWDC Videos** are the official source of new feature launches, recorded at WWDC in San Francisco back in June 2014. Note that there are likely to have been some changes to the APIs between what was announced in these videos and works now, but that doesn't make them any less useful.
- [developer.apple.com/swift/blog/¹⁰⁸](https://developer.apple.com/swift/blog/). **Apple Swift Blog**. This blog appears to be representative of the slight 'opening-up' of Apple. It contains some great posts about how to get started with Swift, through to some more advanced techniques.
- [asciwwdc.com¹⁰⁹](http://asciwwdc.com) **ASCII WWDC** contains searchable transcripts from the recent years of WWDC - including 2014. This is an excellent resource if you can't face trawling through yet another video.
- [Ray Wenderlich Swift/iOS 8 Books Bundle¹¹⁰](http://www.raywenderlich.com/store/swift-tutorials-bundle?source=visualputty). Ray's site is full of fantastic tutorials, and every year the team puts together some tutorial based books about the new releases. This year was no different - in fact I wrote 5 chapters about adaptive layout. If you prefer your learning in tutorial form, then look no further than this excellent resource.
- [iosdevweekly.com¹¹¹](http://iosdevweekly.com). **iOS Dev Weekly** is a stalwart of news about iOS. It's an excellently compiled newsletter that keeps you informed about everything Apple and iOS development.
- [swiftnews.curated.co¹¹²](https://swiftnews.curated.co). **This week in Swift** is a weekly newsletter curated expertly by @NatashaTheRobot¹¹³. Includes a great collection of articles specifically about Swift.

¹⁰⁷<https://developer.apple.com/videos/wwdc/2014/>

¹⁰⁸<https://developer.apple.com/swift/blog/>

¹⁰⁹<http://asciwwdc.com/>

¹¹⁰<http://www.raywenderlich.com/store/swift-tutorials-bundle?source=visualputty>

¹¹¹<http://iosdevweekly.com/>

¹¹²<https://swiftnews.curated.co/>

¹¹³<https://twitter.com/natashatherobot>