Project 4. Easy Browser with WKWebView

Overview

Brief: Embed Web Kit and learn about delegation, KVO, classes and UIToolbar.
Learn: loadView(), WKWebView, delegation, classes and structs, NSURLRequest, UIToolbar, UIProgressView., key-value observing.

- Setting up
- Creating a simple browser with WKWebView
- Choosing a website: UIAlertController
- Monitoring page loads: UIToolbar and UIProgressView
- Refactoring for the win
- Wrap up

Setting up

In this project you're going to build on your new knowledge of UIBarButtonItem, UIAlertController and NSURL by producing a simple web browser app. Yes, I realise this is another easy project, but learning is as much about tackling new challenges as going over what you've already learned.

To sweeten the deal, I'm going to use this opportunity to teach you lots of new things: WKWebView (Apple's extraordinary web widget), UIToolbar (a toolbar component that holds UIBarButtonItems), UIProgessView, delegation, classes and structs, key-value observing, and how to create your views in code. Plus, this is the last easy app project, so enjoy it while it lasts!

To get started, create a new Xcode project using the Single View Application template, and call it Project4. Choose iPhone for the device, and make sure Swift is selected for the language, then save the project on your desktop. Open up Main.storyboard, select the view controller, and choose Editor > Embed In > Navigation Controller – that's our storyboard finished.

Creating a simple browser with WKWebView

In Projects 1 and 2, we used Interface Builder for a lot of layout work, but here our layout will be so simple we can do the entire thing in code. You see, before we were adding buttons and images to our view, but in this project the web view is going to take up all the space so it might as well be the view controller's main view.

So far, we've been using the viewDidLoad() method to configure our view once its layout has loaded. This time we need to override the actual loading of the view – we don't want that empty thing on the storyboard, we want our own code. It will still be placed inside the navigation controller, but the rest is up to us.

Open ViewController.swift for editing, and before viewDidLoad() put this:

```
override func loadView() {
    webView = WKWebView()
    webView.navigationDelegate = self
    view = webView
}
```

It isn't at all necessary to put loadView() before viewDidLoad() – you could put it anywhere between class ViewController: UIViewController

{ down to the last closing brace in the file. But I do encourage you to structure your methods in an organised way, and because loadView() gets called before viewDidLoad() it makes sense to position the code above it too.

Anyway, there are only three things we care about, because by now you should understand why we need to use the override keyword. (Hint: it's because there's a default implementation, which is to load the layout from the storyboard!) First, we create a new instance of Apple's WKWebView web browser component and assign it to a variable called webView. Third, we make our view (the root view of the view controller) that web view.

Yes, I missed out the second line, and that's because it introduces new concept: delegation. Delegation is what's called a programming pattern – a way of writing code – and it's used extensively in iOS. And for good reason: it's easy to understand, easy to use, and extremely flexible.

A delegate is one thing acting in place of another, effectively answering questions and responding to events on its behalf. In our example, we're using WKWebView: Apple's powerful, flexible and efficient web renderer. But as smart as WKWebView is, it doesn't know (or care!) how our application wants to behave, because that's our custom code.

The delegation solution is brilliant: we can tell WKWebView that we want to be told when something interesting happens. In our code, we're setting the web view's navigationDelegate property to self, which means "when any web page navigation happens, please tell me".

When you do this, two things happen:

1. You must conform to the `protocol`. This is a fancy way of saying, "`if you're telling me you can handle being my delegate, here are the methods you need to implement`". In the case of `navigationDelegate`, all these methods are optional, meaning that we don't need to implement any methods.
2. Any methods you do implement will now be given control over the `WKWebView's` behavior. Any you don't implement will use the default behavior of `WKWebView`.

Before we get any further, you may have noticed that your code doesn't actually compile. There are three reasons, but all three take just seconds to fix.

`Reason #1: Swift` doesn't know what `WKWebView` is. As you can see, it doesn't start with "`UI`" so `WKWebView` is not part of `UIKit`. As a result, we need to import a new framework before we can use it. The "`WK`" in `WKWebView` stands for WebKit, so go to the top of your file and modify it to this:

```
import UIKit
import WebKit
```

`Reason #2:` We haven't declared a `webView` property, and yet we're assigning to it. To fix this, go to the line before our `loadView()` method and add this:

```
var webView: WKWebView!
```

That declares an implicitly unwrapped optional `WKWebView` instance called `webView`. I'll explain one more time just so we're clear: that `!` at the end of `WKWebView` is needed because the property starts out as `nil` before being set later on.

Reason #3: When you set any delegate, you need to conform to the protocol. Yes, all the navigationDelegate protocol methods are optional, but we Swift doesn't know that yet. All it knows is that we're promising we're a suitable delegate for the web view, and yet haven't implemented the protocol.

The fix for this is simple, but I'm going to hijack it to introduce something else the same time, because this is an opportune moment. First, the fix: find this line:

```
class ViewController: UIViewController {
```

…and change it to this:

```
class ViewController: UIViewController, WKNavigationDelegate
{
```

That's the fix. But what I want to discuss is the class bit, because I've been using words like "data type", "component" and "instance" so far, without really being clear – and I promise you there are developers out there that are absolutely seething as a result. Hello, haters!

There are two types of complex data types in Swift: structures ("structs") and classes. They are extremely similar in Swift, and really there are only two differences likely to matter to you at this stage, or indeed any stage over the next six months or so.

The first difference is that one class can inherit from another. We already talked about this in Project 1, where our view controller inherited from UIViewController. This class inheritance means you get to build on all the amazing power when you inherit from UIViewController, and add your own customisations on top.

The second difference is that when you pass a struct into a method, a copy gets passed in. This means any changes you make in the method won't affect the struct outside of the method. On the other hand, when you pass an instance of a class into a method, it's passed by reference, meaning that the object inside the method is the same one outside the method; any changes you make will stay.

In terms of which is which: `Int`, `Double`, `Float`, `String` and `Array` are all structs, `UIViewController` and any `UIView` are all classes. In practice, this means that whenever you pass an array into a method, it gets copied. That might sound grossly inefficient, particularly if the array contains a huge amount of data, but don't fret about it: `Swift` will avoid any performance penalty as best it can using a technique called copy on write.

Back to our code: all this is important, because I want you to understand exactly what the line of code does. Here it is again:

```
class ViewController: UIViewController, WKNavigationDelegate
{
```

As you can see, the line kicks off with "`class`", showing that we're declaring a new class here. The line ends with an opening brace, and everything from that opening brace to the closing brace at the end of the file form part of our class. The next part, `ViewController`, is the name of our class. Not a great name in a big project, but for a `Single View Application` template project it's fine.

The interesting stuff comes next: there's a colon, followed by `UIViewController`, then a comma and `WKNavigationDelegate`. If you're feeling fancy, this part is called a type inheritance clause, but what it really means is that this is the definition of what the new `ViewController` class is made of: it inherits from

`UIViewController` (the first item in the list), and implements the `WKNavigationDelegate` protocol.

The order here really is important: the parent class (superclass) comes first, then all protocols implemented come next, all separated by commas. We're saying that we conform to only one protocol here (`WKNavigationDelegate`) but you can specify as many as you need to.

So, the complete meaning of this line is "`create a new subclass of UIViewController called ViewController, and tell the compiler that we promise we're safe to use as a WKNavigationDelegate`".

This program is almost doing something useful, so before you run it let's add three more lines. Please place these in the `viewDidLoad() method`, just after the `super` call:

```
let url = NSURL(string: "https://www.hackingwithswift.com")!
webView.loadRequest(NSURLRequest(URL: url))
webView.allowsBackForwardNavigationGestures = true
```

The first line creates a new NSURL, as you saw in the previous project. I'm using `hackingwithswift.com` as an example website, but please change it to something you like. **Warning: you need to ensure you use https:// for your websites, because iOS 9 does not like apps sending or receiving data insecurely.** If this is something you want to override, click here to read about App Transport Security in iOS 9.

The second line does two things: it creates a new `NSURLRequest` object from that NSURL, and gives it to our web view to load.

Now, this probably seems like pointless obfuscation from `Apple`, but `WKWebViews` don't load websites from strings like `www.hackingwithswift.com`, or even from an NSURL made out of those strings. You need to turn the string into an NSURL, then put the NSURL into an `NSURLRequest`, and `WKWebView` will load that. Fortunately it's not hard to do!

Warning: Your URL must be complete, and valid, in order for this process to work. That means including the `https://` part.

The third line enables a property on the web view that allows users to swipe from the left or right edge to move backward or forward in their web browsing. This is a feature from the `Safari` browser that many users rely on, so it's nice to keep it around.

Press `Cmd + R` to run your app now, and you should be able to view your website. Step one done!

Choosing a website: UIAlertController

We're going to lock this app down so that it opens websites selected by the user. The first step to doing this is to give the user the option to choose from one of our selected websites, and that means adding a button to the navigation bar.

Somewhere in `viewDidLoad()` (but always after it has called `super.viewDidLoad()`), add this:

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

We did exactly this in the previous project, except here we're using a custom title for our bar button rather than a system icon. It called the `openTapped()` method, which doesn't exist, when the button is tapped, so let's add that now. Put this method below `viewDidLoad()`:

```
func openTapped() {
    let ac = UIAlertController(title: "Open page…", message:
nil, preferredStyle: .ActionSheet)
    ac.addAction(UIAlertAction(title: "apple.com",
style: .Default, handler: openPage))
    ac.addAction(UIAlertAction(title:
"hackingwithswift.com", style: .Default, handler: openPage))
    ac.addAction(UIAlertAction(title: "Cancel",
style: .Cancel, handler: nil))
    presentViewController(ac, animated: true, completion:
nil)
}
```

Warning: if you did not set your app to be targeted for `iPhone` at the beginning of this chapter, the above code will not work correctly. Yes, I know I told you to set `iPhone`, but a lot of people skip over things in their rush to get ahead. If you chose `iPad or Universal`, you will need to add `ac.popoverPresentationController?.barButtonItem = self.navigationItem.rightBarButtonItem` to the `openTapped()` method before presenting the alert controller.

We used the `UIAlertController` class in `Project 2`, but here it's slightly different for three reason:

1. We're using nil for the message, because this alert doesn't need one.
2. We're using the `preferredStyle` of `.ActionSheet` because we're prompting the user for more information.
3. We're adding a dedicated `Cancel` button using style `.Cancel`. It has a handler of nil which will just hide the alert controller.

Both our website buttons point to the `openPage()` method, which, again, doesn't exist yet. This is going to be very similar to how we loaded the web page before, but now you will at least see why the handler method of `UIAlertAction` takes a parameter telling you which action was selected!

Add this method directly beneath the `openTapped()` method you just made:

```
func openPage(action: UIAlertAction!) {
    let url = NSURL(string: "https://" + action.title!)!
    webView.loadRequest(NSURLRequest(URL: url))
}
```

This method takes one parameter, which is the UIAlertAction object that was selected by the user. Obviously it won't be called if Cancel was tapped, because that had a nil handler rather than openPage.

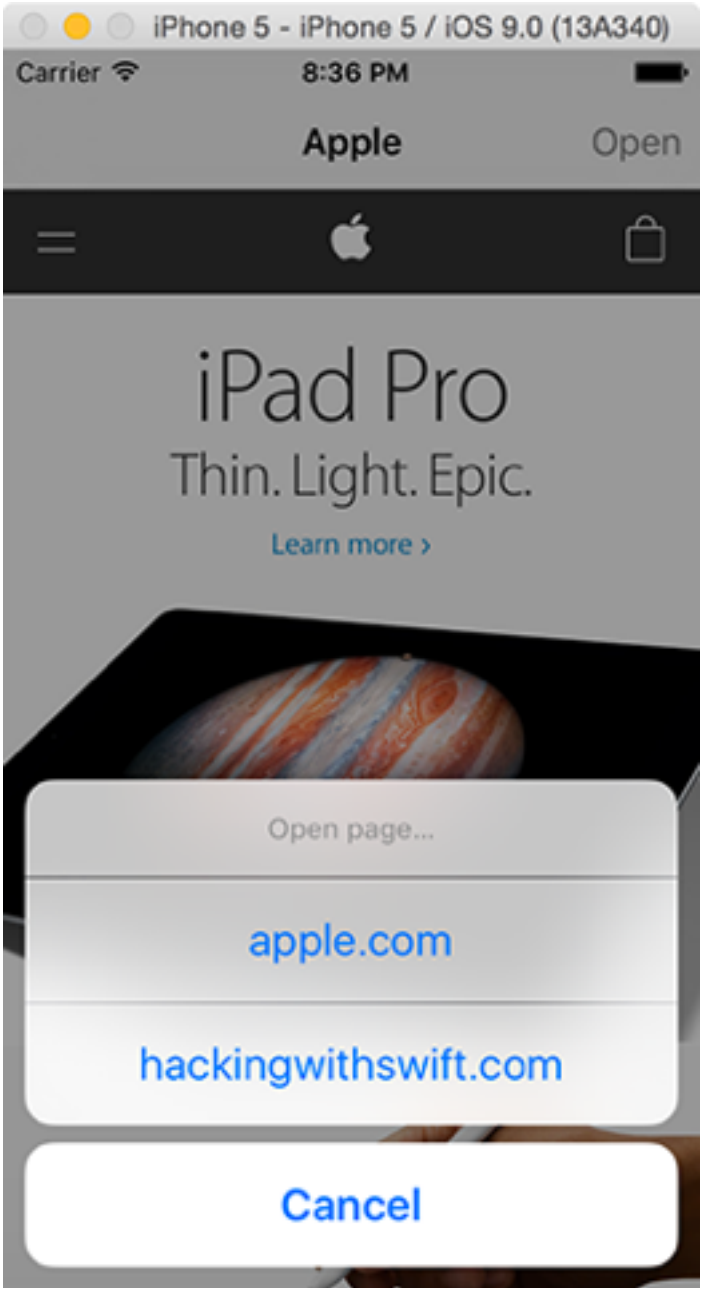
What the method does is use the title property of the action (apple.com, hackingwithswift.com), put "https://" in front of it to satisfy App Transport Security, then construct an NSURL out of it. It then wraps that inside an NSURLRequest, and gives it to the web view to load. All you need to do is make sure the websites in the UIAlertController are correct, and this method will load anything.

You can go ahead and test the app now, but there's one small change we can add to make the whole experience more pleasant: setting the title in the navigation bar. Now, we are the web view's navigation delegate, which means we will be told when any interesting navigation happens, such as when the web page has finished loading. We're going to use this to set the navigation bar title.

As soon as we told Swift that our ViewController class conformed to the WKNavigationDelegate protocol, Xcode updated its code completion system to support all the WKNavigationDelegate methods that can be called. As a result, if you go below the openPage() method and start typing "web" you'll see a list of all the WKNavigationDelegate methods we can use.

Scroll through the list of options until you see didFinishNavigation and press return to have Xcode fill in the method for you. Now modify it to this:

```
func webView(webView: WKWebView, didFinishNavigation
navigation: WKNavigation!) {
    title = webView.title
}
```



All this method does it update our view controller's title property to be the title of the web view, which will automatically be set to the page title of the web page that was most recently loaded.

Press `Cmd + R` now to run the app, and you'll see things are starting to come together: your initial web page will load, and when the load finishes you'll see its page title in the navigation bar.

Monitoring page loads: UIToolbar and UIProgressView

Now is a great time to meet two new `UIView` subclasses: `UIToolbar` and `UIProgressView`. `UIToolbar` holds and shows a collection of `UIBarButtonItem` objects that the user can tap on. We already saw how each view controller has a `rightBarButton` item, so a `UIToolbar` is like having a

whole bar of these items. `UIProgressView` is a colored bar that shows how far a task is through its work, sometimes called a "`progress bar`".

The way we're going to use `UIToolbar` is quite simple: all view controllers automatically come with a toolbarItems array that automatically gets read in when the view controller is active inside a `UINavigationController`.

This is very similar to the way `rightBarButtonItem` is shown only when the view controller is active. All we need to do is set the array, then tell our navigation controller to show its toolbar, and it will do the rest of the work for us.

We're going to create two `UIBarButtonItems` at first, although one is special because it's a flexible space. This is a unique `UIBarButtonItem` type that acts like a spring, pushing other buttons to one side until all the space is used.

In `viewDidLoad()`, put this new code directly below where we set the `rightBarButtonItem`:

```
let spacer =
UIBarButtonItem(barButtonSystemItem: .FlexibleSpace, target:
nil, action: nil)
let refresh = UIBarButtonItem(barButtonSystemItem: .Refresh,
target: webView, action: #selector(webView.reload))

toolbarItems = [spacer, refresh]
navigationController?.toolbarHidden = false
```

The first line is new, or at least part of it is: we're creating a new bar button item using the special system item type `.FlexibleSpace`, which creates a flexible space. It doesn't need a target or action because it can't be tapped. The second line

you've seen before, although now it's calling the `reload()` method on the web view rather than using a method of our own.

The last two lines are new: the first puts an array containing the flexible space and the refresh button, then sets it to be our view controller's toolbarItems array. The second sets the navigation controller's toolbarHidden property to be false, show the toolbar will be shown – and its items will be loaded from our current view.

That code will compile and run, and you'll see the refresh button neatly aligned to the right – that's the effect of the flexible space automatically taking up as much room as it can on the left.

The next step is going to be to add a `UIProgressView` to our toolbar, which will show how far the page is through loading. However, this requires two new pieces of information:

- You can't just add random `UIView` subclasses to a `UIToolbar`, or to the `rightBarButtonItem` property. Instead, you need to wrap them in a special `UIBarButtonItem`, and use that instead.
- Although `WKWebView` tells us how much of the page has loaded using its `estimatedProgress` property, the `WKNavigationDelegate` system doesn't tell us when this value has changed. So, we're going to ask `iOS` to tell us using a powerful technique called key-value observing, or KVO.

First, let's create the progress view and place it inside the bar button item. Begin by declaring the property at the top of the `ViewController` class next to the existing `WKWebView` property:

```
var progressView: UIProgressView!
```

Now place this code directly before the `let spacer = line in viewDidLoad()`:

```
progressView = UIProgressView(progressViewStyle: .Default)
progressView.sizeToFit()
let progressButton = UIBarButtonItem(customView:
progressView)
```

All three of those lines are new, so let's go over them:

1. The first line creates a new `UIProgressView` instance, giving it the default style. There is an alternative style called `.Bar`, which doesn't draw an unfilled line to show the extent of the progress view, but the default style looks best here.
2. The second line tells the progress view set its layout size so that it fits its contents fully.
3. The last line creates a new `UIBarButtonItem` using the `customView` parameter, which is where we wrap up our `UIProgressView` in a `UIBarButtonItem` so that it can go into our toolbar.

With the new `progressButton` item created, we can put it into our toolbar items anywhere we want it. The existing spacer will automatically make itself smaller to give space to the progress button, so I'm going to modify my `toolbarItems` array to this:

```
toolbarItems = [progressButton, spacer, refresh]
```

That is, progress view first, then a space in the center, then the refresh button on the right.

If you run the app now, you'll just see a thin gray line for our progress view – that's because it's default value is `0`, so there's nothing colored in. Ideally we want to set this to match our webView's `estimatedProgress` value, which is a number from `0` to `1`, but `WKNavigationDelegate` doesn't tell us when this value has changed.

`Apple's` solution to this is huge. `Apple's` solution is powerful. And, best of all, `Apple's` solution is almost everywhere in its toolkits, so once you learn how it works you can apply it elsewhere. It's called `key-value observing (KVO)`, and it effectively lets you say, "`please tell me when the property X of object Y gets changed by anyone at any time`".

We're going to use KVO to watch the `estimatedProgress` property, and I hope you'll agree that it's extremely easy. First, we add ourselves as an observer of the property on the web view by adding this to `viewDidLoad()`:

```
webView.addObserver(self, forKeyPath: "estimatedProgress",
options: .New, context: nil)
```

The `addObserver()` method takes four parameters: who the observer is (we're the observer, so we use self), what property we want to observe (we want the `estimatedProgress` property), which value we want (we want the value that was just set, so we want the new one), and a context value.

`forKeyPath` and context bear a little more explanation. `forKeyPath` isn't named `forProperty` because it's not just about entering a property name. You can actually specify a path: one property inside another, inside another, and so on. More advanced key paths can even add functionality, such as averaging all elements in an array!

`context` is easier: if you provide a unique value, that same context value gets sent back to you when you get your notification that the value has changed. This allows you to check the context to make sure it was your observer that was called. There are some corner cases where specifying (and checking) a context is required to avoid bugs, but you won't reach them during any of this series.

Warning: in more complex applications, all calls to `addObserver()` should be matched with a call to `removeObserver()` when you're finished observing – for example, when you're done with the view controller.

Once you have registered as an observer using KVO, you must implement a method called `observeValueForKeyPath()`. This tells you when an observed value has changed, so add this method now:

```
override func observeValueForKeyPath(keyPath: String?,
ofObject object: AnyObject?, change: [String : AnyObject]?,
context: UnsafeMutablePointer<Void>) {
     if keyPath == "estimatedProgress" {
          progressView.progress =
Float(webView.estimatedProgress)
     }
}
```

As you can see it's telling us which key path was changed, and it also sends us back the context we registered earlier so you can check whether this callback is for you or not.

In this project, all we care about is whether the `keyPath` parameter is set to `estimatedProgress` – that is, if the `estimatedProgress` value of the web

view has changed. And if it has, we set the progress property of our progress view to the new `estimatedProgress` value.

Minor note: `estimatedProgress` is a `Double`, which as you should remember is one way of representing decimal numbers like `0.5` or `0.55555`. Unhelpfully, `UIProgressView's` progress property is a `Float`, which is another (lower-precision) way of representing decimal numbers. `Swift` doesn't let you put a `Double` into a `Float`, so we need to create a new `Float` from the `Double`.

If you run your project now, you'll see the progress view fills up with blue as the page loads.

Refactoring for the win

Our app has a fatal flaw, and there are two ways to fix it: double up on code, or refactor. Cunningly, the first option is nearly always the easiest, and yet counter-intuitively also the hardest.

The flaw is this: we let users select from a list of websites, but once they are on that website they can get pretty much anywhere else they want just by following links. Wouldn't it be nice if we could check every link that was followed so that we can make sure it's on our safe list?

One solution – doubling up on code – would have us writing the list of accessible websites twice: once in the `UIAlertController` and once when we're checking the link. This is extremely easy to write, but it can be a trap: you now have two lists of websites, and it's down to you to keep them both up to date. And if you find a bug in your duplicated code, will you remember to fix it in the other place too?

The second solution is called refactoring, and it's effectively a rewrite of the code. The end result should do the same thing, though. The purpose of the rewrite is to make it more efficient, make it easier to read, reduce its complexity, and to make it more flexible. This last use is what we'll be shooting for: we want to refactor our code so there's a shared array of allowed websites.

Up where we declared our two properties `webView` and `progressView`, add this:

```
var websites = ["apple.com", "hackingwithswift.com"]
```

That's an array containing the websites we want the user to be able to visit.

With that array, we can modify the web view's initial web page so that it's not hard-coded. In `viewDidLoad()`, change the initial web page to this:

```
let url = NSURL(string: "https://" + websites[0])!
webView.loadRequest(NSURLRequest(URL: url))
```

So far, so easy. The next change is to make our `UIAlertController` use the websites for its list of `UIAlertActions`. Go down to the `openTapped()` method and replace these two lines:

```
ac.addAction(UIAlertAction(title: "apple.com",
style: .Default, handler: openPage))
ac.addAction(UIAlertAction(title: "hackingwithswift.com",
style: .Default, handler: openPage))
```

…with this loop:

```
for website in websites {
```

```
    ac.addAction(UIAlertAction(title: website,
style: .Default, handler: openPage))
}
```

That will add one `UIAlertAction` object for each item in our array. Again, not too complicated.

The final change is something new, and it belongs to the `WKNavigationDelegate` protocol. If you find space for a new method and start typing "`web`" you'll see the list of `WKWebView`-related code completion options. Look for the one called `decidePolicyForNavigationAction` and let Xcode fill in the method for you.

This delegate callback allows us to decide whether we want to allow navigation to happen or not every time something happens. We can check which part of the page started the navigation, we can see whether it was triggered by a link being clicked or a form being submitted, or, in our case, we can check the URL to see whether we like it.

Now that we've implemented this method, it expects a response: should we load the page or should we not? When this method is called, you get passed in a parameter called `decisionHandler`. This actually holds a function, which means if you "`call`" the parameter, you're actually calling the function.

If your brain has just turned to soup, let me try to clarify. In Project 2 I talked about closures: chunks of code that you can pass into a function like a variable and have executed at a later date. This `decisionHandler` is also a closure, except it's the other way around – rather than giving someone else a chunk of code to execute, you're being given it and are required to execute it.

And make no mistake: you are required to do something with that `decisionHandler` closure. That might make sound an extremely complicated way of returning a value from a method, and that's true – but it's also underestimating the power a little! Having this `decisionHandler` variable/function means you can show some user interface to the user "`Do you really want to load this page?`" and call the closure when you have an answer.

So, we need to evaluate the URL to see whether it's in our safe list, then call the `decisionHandler` with a negative or positive answer. Here's the code for the method:

```
func webView(webView: WKWebView,
decidePolicyForNavigationAction navigationAction:
WKNavigationAction, decisionHandler:
(WKNavigationActionPolicy) -> Void) {
    let url = navigationAction.request.URL

    if let host = url!.host {
        for website in websites {
            if host.rangeOfString(website) != nil {
                decisionHandler(.Allow)
                return
            }
        }
    }

    decisionHandler(.Cancel)
}
```

There are some easy bits, but they are outweighed by the hard bits so let's go through every line in detail to make sure:

1. First, we set the constant url to be equal to the NSURL of the navigation. This is just to make the code clearer.
2. Second, we use `if/let` syntax to unwrap the value of the optional `url.host`. Remember I said that NSURL does a lot of work for you in parsing URLs properly? Well, here's a good example: this line says, "`if there is a host for this URL, pull it out`" – and by "`host`" it means "`website domain`" like `apple.com`. NB: we need to unwrap this carefully because not all URLs have hosts.
3. Third, we loop through all sites in our safe list, placing the name of the site in the website variable.
4. Fourth, we use the `rangeOfString() String` method to see whether each safe website exists somewhere in the host name.
5. Fifth, if the website was found (if `rangeOfString()` is not nil) then we call the decision handler with a positive response: allow loading.
6. Sixth, if the website was found, after calling the `decisionHandler` we use the return statement. This means "`exit the method now`".
7. Last, if there is no host set, or if we've gone through all the loop and found nothing, we call the decision handler with a negative response: cancel loading.

The `rangeOfString()` method can take quite a few parameters, however all but the first are optional so the above usage is fine. To use it, call `rangeOfString()` on one string, giving it another string as a parameter, and it will tell you where it was found, or nil if it wasn't found at all.

You've already met the `hasPrefix()` method in `Project 1`, but `hasPrefix()` isn't suitable here because our safe site name could appear anywhere in the URL. For example, `slashdot.org` redirects to `m.slashdot.org` for mobile devices, and `hasPrefix()` would fail that test.

The return statement is new, but it's one you'll be using a lot from now on. It exits the method immediately, executing no further code. If you said your method returns a value, you'll use the return statement to return that value.

Your project is complete: press Cmd + R to run the finished app, and enjoy!

Wrap up

Another project done, another huge pile of things learned. You should be starting to get into the swing of things by now, but don't let yourself become immune to your success. In this tutorial alone you've learned about loadView(), WKWebView, delegation, classes and structs, NSURLRequest, UIToolbar, UIProgressView, KVO and more, so you should be proud of your fantastic accomplishments!

There is a lot of scope for improvement with this project, so where you start is down to you. I would suggest at the very least that you investigate changing the initial view controller to a table view like in Project 1, where users can go choose their website from a list rather than just having the first in the array loaded up front.

Once you have completed Project 5, you might like to return here to add in the option to load the list of websites from a file, rather than having them hard-coded in an array.