Project 7: Whitehouse Petitions: JSON and NSData

Overview

Brief: Make an app to parse Whitehouse petitions using JSON and a tab bar.
Learn: JSON, NSData, UITabBarController.

- Setting up
- Creating the basic UI: UITabBarController
- Parsing JSON: NSData and SwiftyJSON
- Rendering a petition: loadHTMLString
- Finishing touches: didFinishLaunching
- Wrap up

Setting up

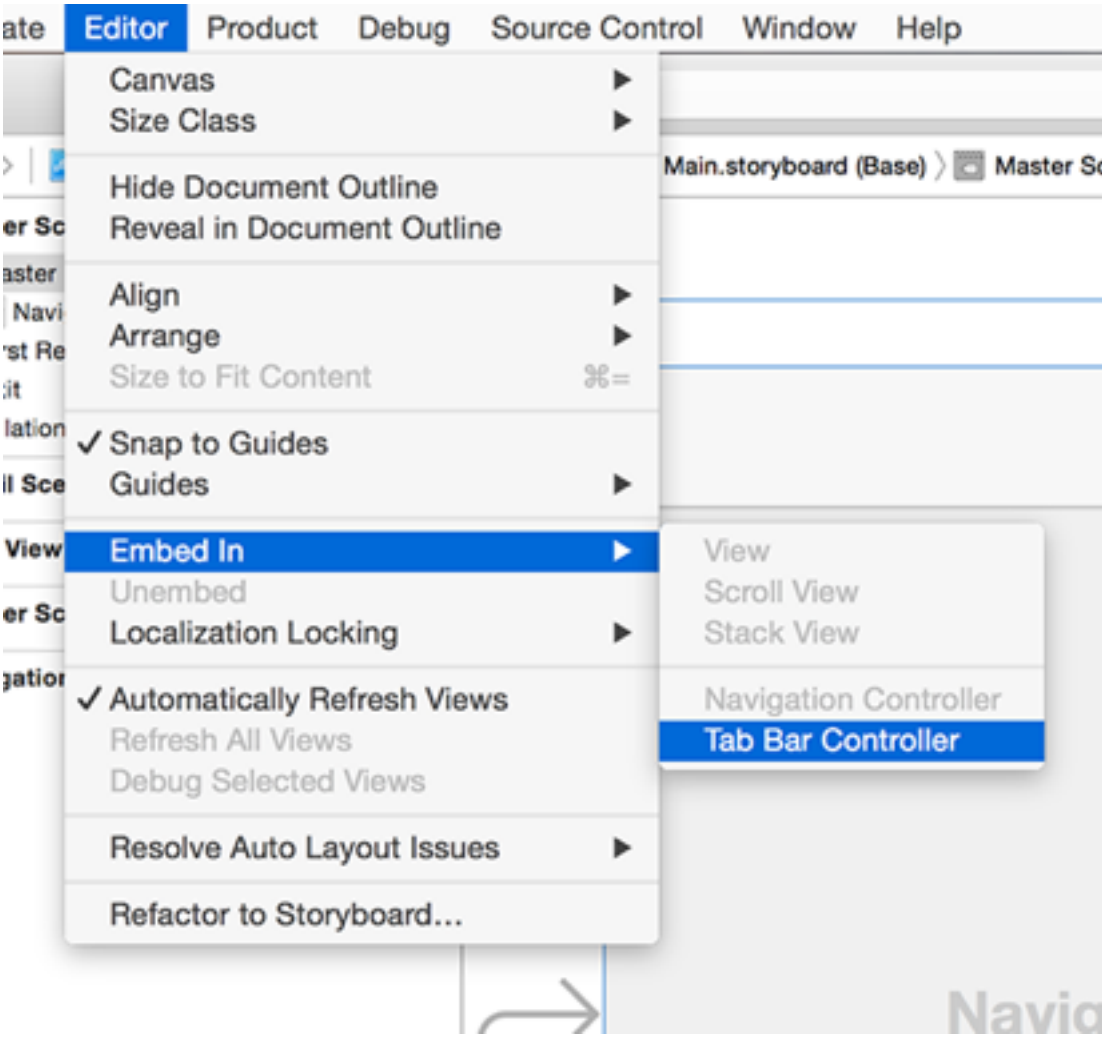
This project will take a data feed from a website and parse it into useful information for users. As per usual, this is just a way of teaching you some new iOS development techniques, but let's face it – you already have two apps and two games under your belt, so you're starting to build up a pretty good library of work!

This time, you'll be learning about UITabBarController, NSData, and more. You'll also be using a data format called JSON, which is a popular way to send and receive data online. It's not easy to find interesting JSON feeds that are freely available, but the option we'll be going for is the "We the people" Whitehouse petitions in the US, where Americans can submit requests for action, and others can vote on it.

Some are entirely frivolous (“We want the US to build a Death Star”), but it has good, clean JSON that's open for everyone to read, which makes it perfect. Lots to learn, and lots to do, so let's get started: create a new project in Xcode by choosing the Master-Detail Application template. Now name it Project7, set its target to be iPad, and save it somewhere.

Creating the basic UI: UITabBarController

The Master-Detail application template gives us a lot of stuff we don't need, but rather than delete it we're just going to modify it as needed. For the user interface we need to make only a handful of changes, so please open Main.storyboard in Interface Builder.



Behind the scenes, UITabBarController manages an array of view controllers that the user can choose between. You can do often do most of the work inside Interface Builder, but not in this project. We're going to use one tab to show recent petitions, and another to show popular petitions, which is the same thing really – all that's changing is the data source.

Doing everything inside the storyboard would mean duplicating our view controllers, which is A Bad Idea, so instead we're just going to design one of them in the storyboard then create a duplicate of it using code.

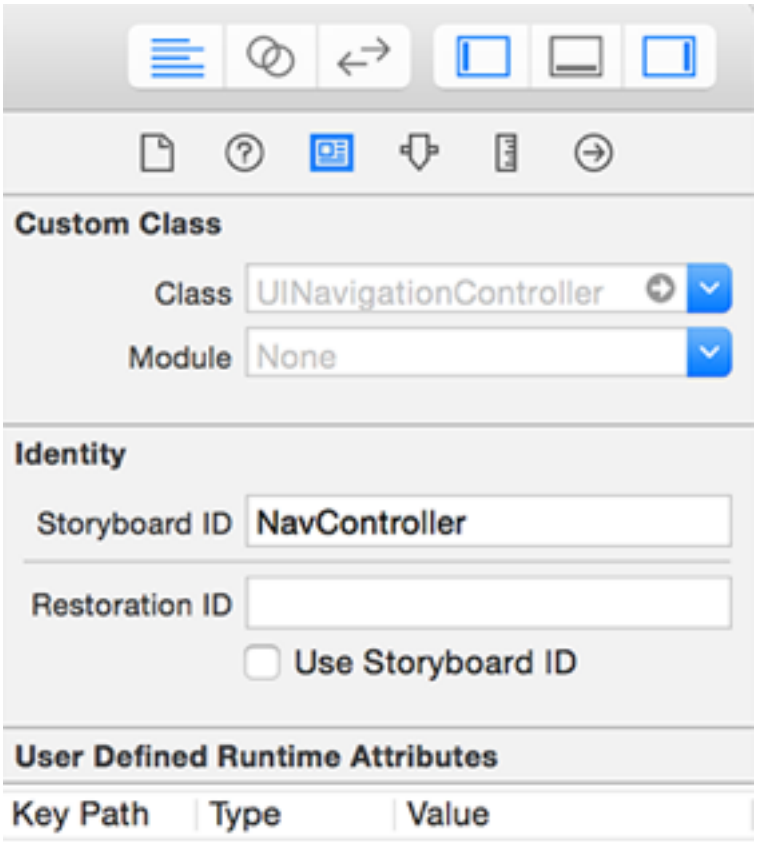
Now that our navigation controller is inside a tab bar controller, it will have acquired a gray strip along its bottom in Interface Builder. If you click that now, it will select a new type of object called a UITabBarItem, which is the icon and text used to represent a view controller in the tab bar. In the attributes inspector (Alt + Cmd + 4) change System Item from "Custom" to "Most Recent".

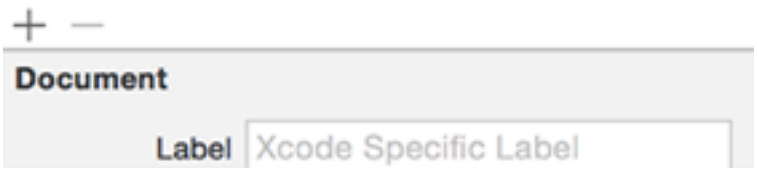
One important thing about UITabBarItem is that when you set its system item, it assigns both an icon and some text for the title of the tab. If you try to change the text to your own text, the icon will be removed and you need to provide your own. This is because Apple has trained users to associate certain icons with certain information, and they don't want you using those icons incorrectly!

Select the navigation controller itself (just click where it says Navigation Controller in big letters in the center of the view controller), then press Alt + Cmd + 3 to select the identity inspector. We haven't been here before, because it's not used that frequently. However, here I want you to type "NavController" in the text box to the right of where it says, "Storyboard ID". We'll be needing that soon enough!

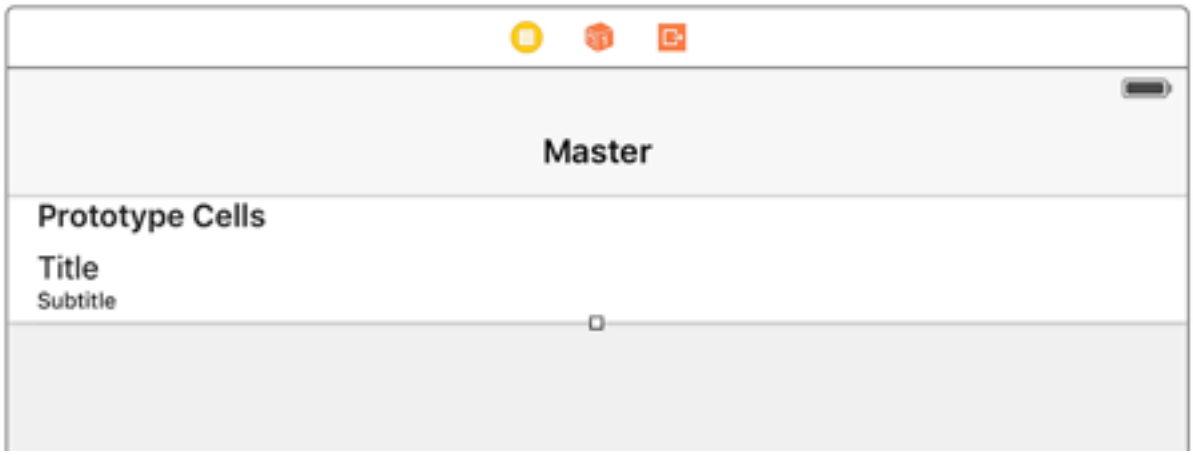
In the picture below you can see how the identity inspector should look when configured for your navigation controller. You'll be using this inspector in later projects to give views a custom class by changing the first of these four text boxes.

In your storyboard, you'll see there are two navigation controllers: one at the top that connects to a table view on its right, and another at the bottom that connects to a view saying `"Detail view content goes here"` on its right. Choose the top one, then choose `Editor > Embed In > Tab Bar Controller`. Like `UINavigationController`, `UITabBarController` is a common element in `iOS` user interface design. A tab bar is that strip of icons across the bottom that shows various screens, and it appears in the `App Store`, in the music app, in the phone app, and more.





The other change we're going to make to the storyboard is in the table view controller. Click once where it says `"Title"` (just below where it says `Prototype Cells`), and in the attributes inspector you'll some options for the cell. You'll know when you have it right because it will say `"Table View Cell"` at the top of the attributes inspector. Change the style from `"Basic"` to `"Subtitle"`, which adds an extra line of text underneath the title.



We're done with `Interface Builder`, so please open the file `MasterViewController.swift` so we can make a few basic changes. First, delete the `insertNewObject()` method entirely, delete everything from `viewDidLoad()` except the call to `super.viewDidLoad()`, delete the table view's `commitEditingStyle` and `canEditRowAtIndexPath` methods, and finally delete the `as! NSDate` text from the `prepareForSegue()` and `cellForRowAtIndexPath` methods – not the whole line, just the bit that says `as! NSDate`.

Step one is now complete: we have a basic user interface in place, and we've cleaned the unwanted cruft out of the `Xcode` template. Now for some real code…

Parsing JSON: NSData and SwiftyJSON

`JSON` – short for `JavaScript Object Notation` – is a way of describing data. It's not the easiest to read yourself, but it's compact and easy to parse for computers, which makes it popular online where bandwidth is at a premium.

In `Project 6` you learned about using dictionaries with `Auto Layout`, and in this project we're going to use dictionaries more extensively. What's more, we're going to put dictionaries inside an array to make an array of dictionaries, which should keep our data in order.

You declare a dictionary using square brackets, then entering its key type, a colon, and its value type. For example, a dictionary that used strings for its keys and `UILabels` for its values would be declared like this:

```
var labels = [String: UILabel]()
```

And as you'll recall, you declare arrays just by putting the data type in brackets, like this:

```
var strings = [String]()
```

Putting these two together, we want to make an array of dictionaries, with each dictionary holding a string for its key and another string for its value. So, it looks like this:

```
var objects = [[String: String]]()
```

Put that in place of the current objects definition at the top of MasterViewController.swift – it holds AnyObject right now, which won't do the job.

It's now time to parse some JSON, which means to process it and examine its contents. This isn't easy in Swift, so a number of helper libraries have appeared that do a lot of the heavy lifting for you. We're going to use one of them now: download the files for this project from GitHub then look for a file called SwiftyJSON.swift. Add that your project, making sure "Copy items if needed" is checked.

SwiftyJSON lets us read through JSON in an extremely intuitive way: you can effectively treat almost everything as a dictionary, so if you know there's a value called "information" that contains another value called "name", which in turns contains another value called "firstName", you can use json["information"]["name"]["firstName"] to get the data, then ask for it as a Swift value by using the string property.

Before we do the parsing, here is a tiny slice of the actual JSON you'll be receiving:

```
{
    "metadata":{
        "responseInfo":{
            "status":200,
            "developerMessage":"OK",
        }
    },
    "results":[
        {
```

```
            "title":"Legal immigrants should get freedom before undocumented immigrants – moral, just and fair",
            "body":"I am petitioning President Obama's Administration to take a humane view of the plight of legal immigrants. Specifically, legal immigrants in Employment Based (EB) category. I believe, such immigrants were short changed in the recently announced reforms via Executive Action (EA), which was otherwise long due and a welcome announcement.",
            "issues":[
                {
                    "id":"28",
                    "name":"Human Rights"
                },
                {
                    "id":"29",
                    "name":"Immigration"
                }
            ],
            "signatureThreshold":100000,
            "signatureCount":267,
            "signaturesNeeded":99733,
        },
        {
            "title":"National database for police shootings.",
            "body":"There is no reliable national data on how many people are shot by police officers each year. In signing this petition, I am urging the President to bring an end to this absence of visibility by creating a federally
```

```
controlled, publicly accessible database of officer-involved
shootings.",
            "issues":[
                {
                    "id":"28",
                    "name":"Human Rights"
                }
            ],
            "signatureThreshold":100000,
            "signatureCount":17453,
            "signaturesNeeded":82547,
        }
    ]
}
```

You'll actually be getting between `2000-3000` lines of that stuff, all containing petitions from US citizens about all sorts of political things. It doesn't really matter (to us) what the petitions are, we just care about the data structure. In particular:

1. There's a metadata value, which contains a responseInfo value, which in turn contains a status value. `Status 200` is what internet developers use for `“everything is OK”`.
2. There's a results value, which contains a series of petitions.
3. Each petition contains a title, a body, some issues it relates to, plus some signature information.
4. `JSON` has strings and integers too. Notice how the strings are all wrapped in quotes, whereas the integers aren’t.

Now that you have a basic understanding of the `JSON` we'll be working with, it's time to write some code. We're going to update the `viewDidLoad()` method for

`MasterViewController` so that it downloads the data from the Whitehouse petitions server, converts it to a `SwiftyJSON` object, and checks that the status value is equal to `200`.

To make this happen, we're going to use `NSURL` alongside a new `NS` class called `NSData`. This is a class designed to hold data in any form, which might be a string, it might be an image, or it might be something else entirely. You already saw that `NSString` can be created using `contentsOfFile` to load data from disk. Well, `NSData` (and `NSString`) can be created using `contentsOfURL`, which downloads data from a `URL` (specified using `NSURL`) and makes it available to you.

Here's the new `viewDidLoad` method:

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

    let urlString = "https://api.whitehouse.gov/v1/
petitions.json?limit=100"

    if let url = NSURL(string: urlString) {
        if let data = try? NSData(contentsOfURL: url,
options: []) {
            let json = JSON(data: data)

            if json["metadata"]["responseInfo"]
["status"].intValue == 200 {
                // we're OK to parse!
            }
        }
```

```
    }
}
```

Let's focus on the new stuff:

- `urlString` points to the `Whitehouse.gov` server, accessing the petitions system.
- We use if/let to make sure the `NSURL` is valid, rather than force unwrapping it. Later on you can return to this to add more `URLs`, so it's good play it safe.
- We create a new `NSData` object using its `contentsOfURL` method. This returns the content from an `NSURL`, which is all we need – hence why we're using `[]` for options. This might throw an error (i.e., if the internet connection was down), so we also need to use `try?`.
- If the `NSData` object was created successfully, we create a new `JSON` object from it. This is a `SwiftyJSON` structure.
- Finally, we have our first bit of `JSON` parsing: if there is a `"metadata"` value and it contains a `"responseInfo"` value that contains a `"status"` value, return it as an integer, then compare it to `200`.
- The `"we're OK to parse!"` line starts with `//`, which begins a comment line in `Swift`. Comment lines are ignored by the compiler; we write them as notes to ourselves.

The reason `SwiftyJSON` is so good at `JSON` parsing is because it has optionality built into its core. If any of `"metadata"`, `"responseInfo"` or `"status"` don't exist, this call will return `0` for the status – we don't need to check them all individually. If you're reading a string value, `SwiftyJSON` will return either the string it found, or if it didn't exist then an empty string.

This code isn't perfect, in fact far from it. In fact, by downloading data from the internet in `viewDidLoad()` our app will lock up until all the data has been transferred. There are solutions to this, but to avoid complexity they won't be covered until `Project 9`.

For now, we want to focus on our `JSON` parsing. We already have an objects array that is ready to accept dictionaries of data. We want to parse that `JSON` into dictionaries, with each dictionary having three values: the title of the petition, its body text, and how many signatures it has. Once that's done, we need to tell our table view to reload itself.

Are you ready? Because this code is shockingly simple given how much work it's doing:

```
func parseJSON(json: JSON) {
    for result in json["results"].arrayValue {
            let title = result["title"].stringValue
            let body = result["body"].stringValue
            let sigs = result["signatureCount"].stringValue
            let obj = ["title": title, "body": body,
"sigs": sigs]
            objects.append(obj)
    }

    tableView.reloadData()
}
```

Place that method just underneath `viewDidLoad()` method, then replace the existing `// we're OK to parse!` line in `viewDidLoad()` with this:

```
parseJSON(json)
```

The `parseJSON()` method reads the `"results"` array from the `JSON` object it gets passed. If you look back at the `JSON` snippet I showed you, that results array contains all the petitions ready to read. When you use `arrayValue` with `SwiftyJSON`, you either get back an array of objects or an empty array, so we use the return value in our loop.

For each result in the results array, we read out three values: its title, its body, and its signature count, with all three of them being requested as strings. The signature count is actually a number when it comes in the `JSON`, but `SwiftyJSON` converts it for us so we can put it inside our dictionary where all the keys and values are strings.

Each time we're accessing an item in our result value using `stringValue`, we will either get its value back or an empty string. Regardless, we'll have something, so we construct a new dictionary from all three values then use `objects.append()` to place the new dictionary into our array.

Once all the results have been parsed, we tell the table view to reload, and the code is complete.

Well, the could would have been complete if the Whitehouse actually used good HTTPS. Even though the `URL` we're hitting starts with `https://api.whitehouse.gov`, the form of HTTPS is so weak at the time of writing that `iOS 9` doesn't trust it. So, if you try running this code you'll get an error: `"NSURLSession/NSURLConnection HTTP load failed (kCFStreamErrorDomainSSL, -9802)"`.

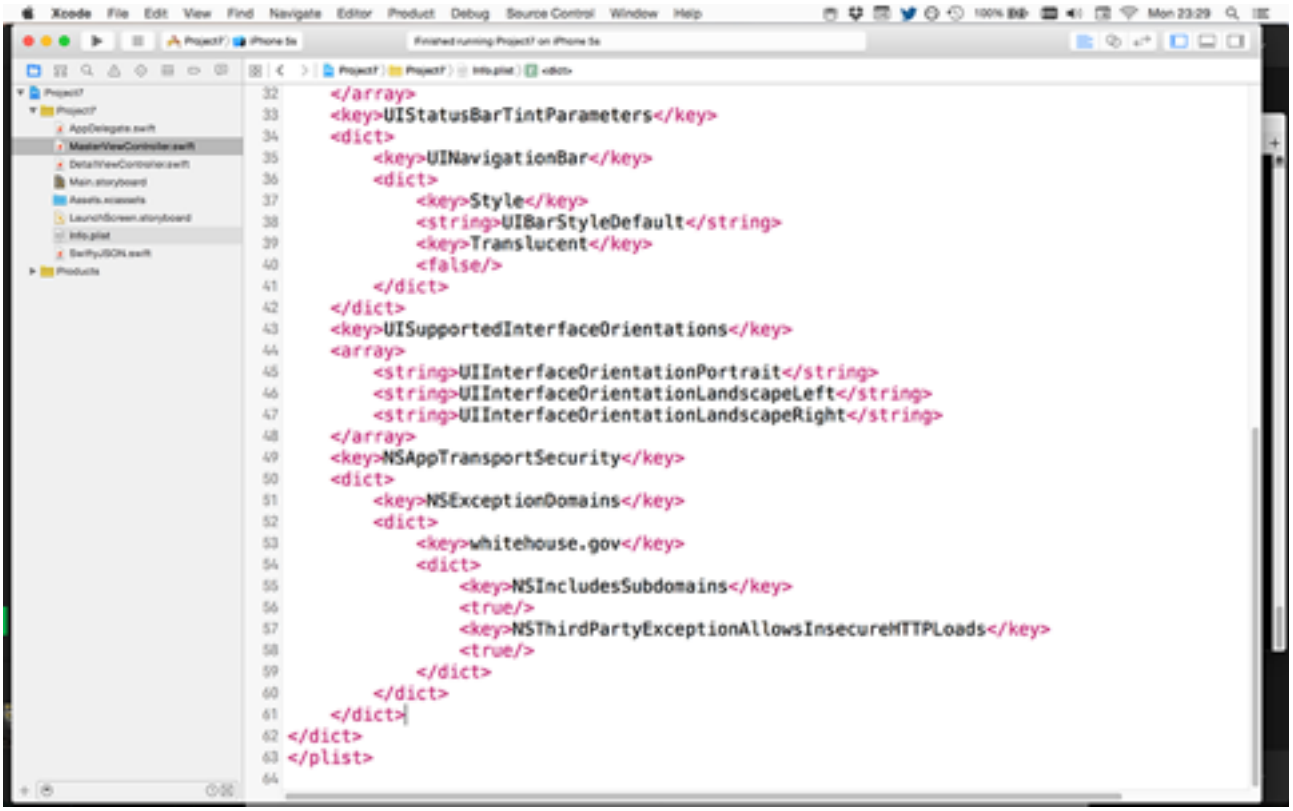
The solution here is to have the Whitehouse upgrade to more secure HTTPS. Failing that, we can ask `iOS` to allow an exception for this insecure domain by customizing its `App Transport Security Settings`. This is an annoyance, and I wouldn't show it to you unless it was strictly necessary, but I'm afraid there's no other choice.

So: look in the project navigator for a file called `Info.plist`. Right-click on it, and choose `Open As > Source Code`. It should end like this:

```
</dict>
</plist>
Just before that, I'd like you to paste this:
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
    <dict>
        <key>whitehouse.gov</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>

<key>NSThirdPartyExceptionAllowsInsecureHTTPLoads</key>
            <true/>
        </dict>
    </dict>
</dict>
```

That adds an `App Transport Security exception` that means `iOS` won't refuse to work with the Whitehouse's weak certification.



You can run the program now, although it's not optimal: you'll see some strangely formatted text in the rows. This is because the built-in `Xcode` project has the following in the table view's `cellForRowAtIndexPath` method inside `MasterViewController.swift`:

```
cell.textLabel!.text = object.description
```

The text label for the cell expects a string, not a dictionary, but `Xcode's` default template code uses object.description to ask the object to describe itself in string format. For dictionaries, this prints out a nicely formatted key/value layout that shows you what the dictionary contains.

We want to modify this so that it prints out the title value of our dictionary, but we also want to use the subtitle text label that got added when we changed the cell type from `“Basic”` to `“Subtitle”` in the `storyboard`. To do that, change the current line (the one above) to this:

```
cell.textLabel!.text = object["title"]
cell.detailTextLabel!.text = object["body"]
```

We set the title, body and sigs keys in the dictionary, and now we can read them out to configure our cell correctly.

If you run the app now, you'll see things are starting to come together quite nicely – every table row now shows the petition title, and beneath it shows the first few words of the petition's body. The subtitle automatically shows "…" at the end when there isn't enough room for all the text, but it's enough to give the user a flavor of what's going on.

Rendering a petition: loadHTMLString

After all the `JSON` parsing, it's time for something easy: we need to update the `DetailViewController` class so that it can draw the petition content in a nice way. The easiest way for rendering complex content from the web is nearly always to use a `WKWebView`, and we're going to use the same technique from `Project 4` to modify `DetailViewController` so it has a web view.

Replace all the `DetailViewController` code with this:

```
import UIKit
import WebKit

class DetailViewController: UIViewController {
    var webView: WKWebView!
    var detailItem: [String: String]!
```

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

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

This is almost identical to the code from Project 4, but you'll notice I've added a detailItem property that will contains our dictionary of data.

That was the easy bit. The hard bit is that we can't just drop the petition text into the web view, because it will probably look tiny. Instead, we need to wrap it in some HTML, which is a whole other language with its own rules and its own complexities. Now, this series isn't called "Hacking with HTML", so I don't intend to go into much detail here. However, I will say that the HTML we're going to use tells iOS that the page fits mobile devices, and that we want the font size to be 150% of the standard font size. All that HTML will be combined with the body value from our dictionary, then sent to the web view.

Place this in viewDidLoad(), directly beneath the call to super.viewDidLoad():

```
guard detailItem != nil else { return }

if let body = detailItem["body"]  {
    var html = "<html>"
    html += "<head>"
```

```
    html += "<meta name=\"viewport\" content=\"width=device-
width, initial-scale=1\">"
    html += "<style> body { font-size: 150%; } </style>"
    html += "</head>"
    html += "<body>"
    html += body
    html += "</body>"
    html += "</html>"
    webView.loadHTMLString(html, baseURL: nil)
}
```

There's a new `Swift` statement in there that is important: `guard`. This is used to create an `“early return”`, which means you set your code up so that it exits immediately if critical data is missing. In our case, we don't want this code to run if detailItem isn't set, so guard will run return is detailItem is set to nil.

I've tried to make the HTML as clear as possible, but if you don't care for HTML don't worry about it. What matters is that there's a `Swift` string called html that contains everything needed to show the page, and that's passed in to the web view's `loadHTMLString()` method so that it gets loaded. This is different to the way we were loading HTML before, because we aren't using a website here, just some custom HTML.

That's it for the detail view controller, it really is that simple. Go ahead and run the project now by pressing `Cmd + R` or clicking play, then tap on a row to see more detail about each petition.

Finishing touches: didFinishLaunching

Before this project is finished, we're going to make two changes. First, we're going to add another tab to the `UITabBarController` that will show popular petitions, and second we're going to make our `NSData` loading code a little more resilient by adding error messages.

As I said previously, we can't really put the second tab into our storyboard because both tabs will host a `MasterViewController` and doing so would require me to duplicate the view controllers in the storyboard. You can do that if you want, but please don't – it's a maintenance nightmare!

Instead, we're going to leave our current storyboard configuration alone, then create the second view controller using code. This isn't something you've done before, but it's not hard and we already took the first step, as you'll see.

Open the file `AppDelegate.swift`. This has been in all our projects so far, but it's not one we've had to work with until now. Look for the `didFinishLaunching` method, which should be at the top of the file. This gets called by `iOS` when the app is ready to be run, and we're going to hijack it to insert a second `MasterViewController` into our tab bar.

It should already have some default `Apple` code in there, but we're going to add some more just before the return true line:

```
let tabBarController = splitViewController.viewControllers[0]
as! UITabBarController
let storyboard = UIStoryboard(name: "Main", bundle: nil)
```

```
let vc =
storyboard.instantiateViewControllerWithIdentifier("NavContro
ller") as! UINavigationController
vc.tabBarItem = UITabBarItem(tabBarSystemItem: .TopRated,
tag: 1)

tabBarController.viewControllers?.append(vc)
```

Every line of that is new, so let's dig in deeper:

- Our storyboard automatically creates a window in which all our view controllers are shown. This window needs to know what its initial view controller is, and that gets set to its `rootViewController` property. This is all handled by our storyboard.
- In the `Master-Detail Application template`, the root view controller is the `UISplitViewController`, which itself has a property called `viewControllers`. This stores two items: the first is the view controller on the left (our table view) and the second is the view controller on the right (our detail view).
- We need to create a new `MasterViewController` by hand, which first means getting a reference to our `Main.storyboard` file. This is done using the `UIStoryboard` class, as shown. You don't need to provide a bundle, because nil means "`use the current app bundle`".
- We create our view controller using the extraordinarily long method `instantiateViewControllerWithIdentifier()`, passing in the `storyboard ID` of the view controller we want. Earlier we set our navigation controller to have the storyboard ID of "`NavController`", so we use pass that in and typecast the result to be a `UINavigationController`.

- We create a new `UITabBarItem` object for the new view controller, giving it the `"Top Rated"` icon and the `tag` `1`. That tag is important, but not just yet.
- We add the new view controller to our tab bar controller's `viewControllers` array, which will cause it to appear in the tab bar.
- So, the code creates a duplicate `MasterViewController` wrapped inside a navigation controller, gives it a new tab bar item to distinguish it from the existing tab, then adds it to the list of visible tabs. This lets us use the same class for both tabs without having to duplicate things in the storyboard.

The reason we gave a tag of `1` to the new `UITabBarItem` is because it's an easy way to identify it. Remember, both tabs contain a `MasterViewController`, which means the same code is executed. Right now that means both will download the same `JSON` feed, which makes having two tabs pointless. But if you modify `urlString` in `MasterViewController's viewDidLoad()` method to this, it will work much better:

```
let urlString: String

if navigationController?.tabBarItem.tag == 0 {
    urlString = "https://api.whitehouse.gov/v1/
petitions.json?limit=100"
} else {
    urlString = "https://api.whitehouse.gov/v1/
petitions.json?signatureCountFloor=10000&limit=100"
}
```

That adjusts the code so that the first instance of `MasterViewController` loads the original `JSON`, and the second loads only petitions that have at least `10,000` signatures.

The project is almost done, but we're going to make one last change. Our current loading code isn't very resilient: we have lots of if statements checking that things are working correctly, but no else statements showing an error message if there's a problem. This is easily fixed by adding a new `showError()` method that creates a `UIAlertController` showing a general failure message:

```
func showError() {
    let ac = UIAlertController(title: "Loading error",
message: "There was a problem loading the feed; please check
your connection and try again.", preferredStyle: .Alert)
    ac.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))
    presentViewController(ac, animated: true, completion:
nil)
}
```

You can now adjust the `JSON` downloading and parsing code to call this error method everywhere a condition fails, like this:

```
if let url = NSURL(string: urlString) {
    if let data = try? NSData(contentsOfURL: url, options:
[])  {
            let json = JSON(data: data)

            if json["metadata"]["responseInfo"]
["status"].intValue == 200  {
                parseJSON(json)
```

```
            } else {
                showError()
            }
    } else {
            showError()
    }
} else {
    showError()
}
```

Now that error messages are shown when the app hits problems, we're done!

Wrap up

As your Swift skill increases, I hope you're starting to feel the balance of these projects move away from explaining the basics and toward presenting and dissecting code. Working with JSON is something you're going to be doing time and time again in your Swift career, and you've cracked it in about an hour of work – while also learning about NSData, UITabBarController, and more. Not bad!

If you're looking to extend this project some more, you might like to look at the original API documentation – it's at https://petitions.whitehouse.gov/developers and contains lots of options. If you add more view controllers to the tab bar, you'll find you can add up to five before you start seeing a "More" button. This More tab hides all the view controllers that don't fit into the tab bar, and it's handled for you automatically by iOS.