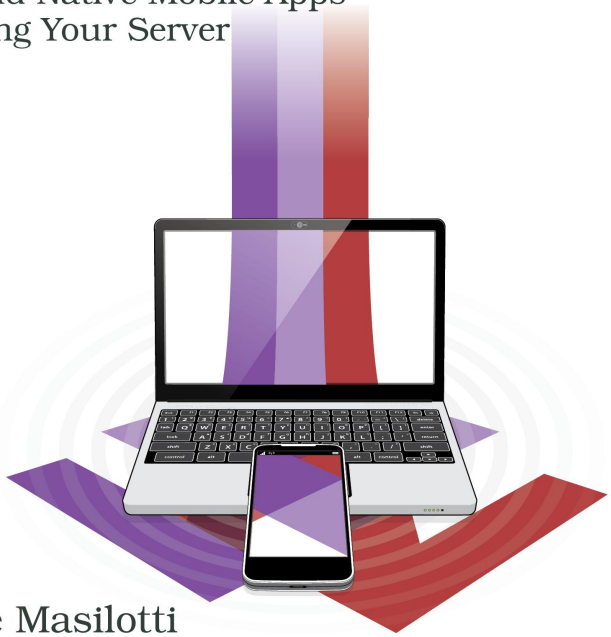
The Pragmatic Programmers

Facets of Ruby
Series editor: Noel Rappin

Hotwire Native for Rails Developers

Build Native Mobile Apps Using Your Server



Joe Masilotti
edited by Nicole Taché

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Create the SceneDelegate class

Like Ruby, Swift files often contain a single class or concept named after the filename. After the import statements, create a new class with the `class` keyword and call it `SceneDelegate`:

ch01_02/ios/HikingJournal/SceneDelegate.swift

```
import HotwireNative
import UIKit

class SceneDelegate {
}
```

The "delegate" suffix refers to the *delegate* design pattern in iOS. This provides a way for objects to act on behalf of others to handle specific events, like when the app launches or a push notification is received. In Rails, delegates are similar to `ActiveRecord` callbacks like `before_validation` and `after_save`.

Add the window and navigator properties

Inside the `SceneDelegate` class, add a property named `window`. This holds the actual interface that the user sees:

ch01_03/ios/HikingJournal/SceneDelegate.swift

```
import HotwireNative
import UIKit

class SceneDelegate {
    var window: UIWindow?
}
```

Unlike Ruby, variables in Swift must be explicitly typed. The question mark at the end of the `UIWindow` type makes this property *optional*. Optional properties behave like Ruby ones - they can also be `nil`. Non-optional properties in Swift, referenced by omitting the question mark, must always contain a value and can never be `nil`.

`window` needs to be optional because it is assigned after `SceneDelegate` is instantiated. You don't ever assign anything to the window, iOS takes care of that when the app launches.

After the window, create a `Navigator` property. Part of Hotwire Native, this property abstracts the navigation between web screens and acts as our main interface to the framework.

ch01_04/ios/HikingJournal/SceneDelegate.swift

```
import HotwireNative
import UIKit

class SceneDelegate {
    var window: UIWindow?
```

```
    private let navigator = Navigator()
}
```

The navigator is an implementation detail of SceneDelegate. No one else needs to know about it, so we can make it private. It is also declared with let, making it a constant. This means that, unlike our window property declared with var above, the value of navigator can never be changed.

Swift encourages immutability whenever possible. Knowing that a variable can never change often makes it easier to understand how data will flow through the code in your app.

Inherit and implement

When the app launches, iOS calls a specific function of a specific class. SceneDelegate needs to inherit from this class and implement the function for our app to launch. To make this happen, add a colon after SceneDelegate and add the UIResponder class and UIWindowSceneDelegate protocol:

ch01_05/ios/HikingJournal/SceneDelegate.swift

```
import HotwireNative
import UIKit

class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?

    private let navigator = Navigator()
}
```

Protocols in Swift are most similar to abstract classes in Rails. On their own, they only contain property and function definitions, no implementation.

After our navigator property, add the function required for the app to launch:

ch01_06/ios/HikingJournal/SceneDelegate.swift

```
import HotwireNative
import UIKit

class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?

    private let navigator = Navigator()

    func scene(
        _ scene: UIScene,
        willConnectTo session: UISceneSession,
        options connectionOptions: UIScene.ConnectionOptions
    ) {

    }
}
```

The function being called is our trigger to start rendering some content.

Functions in Swift

Functions in Swift look pretty different than methods in Ruby. Like properties of a class, each parameter in Swift must also have a type.

The following function takes two parameters of type Int and returns another Int:

```
func add(x: Int, y: Int) -> Int {
    x + y
}

add(x: 1, y: 2) // Returns 3
```

We call functions in Swift just like we call Ruby methods, with named parameters. We can *omit* a parameter from the call site by using an underscore:

```
func add(_ x: Int, _ y: Int) -> Int {
    x + y
}

add(1, 2) // Returns 3
```

Or *rename* a parameter for the implementation:

```
func add(first x: Int, second y: Int) -> Int {
    x + y
}

add(first: 1, second: 2) // Returns 3
```

Functions are referenced by their call site parameters without types. The three functions you just saw would be referenced in documentation as add(x:y:), add(_:_:) and add(first:second:).

Set a root view controller

For iOS to actually render anything, we need to attach a *view controller* to the main window. View controllers, subclasses of UIViewController, are the building blocks of iOS applications. They are kind of like combining a Rails controller and a Rails view into one, as view controllers manage the state of the UI and are responsible for rendering it. With a few exceptions, individual screens and view controllers share a 1:1 relationship.

Grab the rootViewController from Navigator and assign it to the window's root view controller property:

ch01_07/ios/HikingJournal/SceneDelegate.swift

```
import HotwireNative
import UIKit

class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?
```

```
    private let navigator = Navigator()

    func scene(
        _ scene: UIScene,
        willConnectTo session: UISceneSession,
        options connectionOptions: UIScene.ConnectionOptions
    ) {
➤       window?.rootViewController = navigator.rootViewController
    }
}
```

The question mark after `window` is similar to the Safe Navigation Operator in Ruby, `&`. We can chain calls with the question mark to safely work with optionals. If the underlying value is `nil` then nothing happens - just like in Ruby.

When the app launches, it will now render the navigator's screen hierarchy.

Initiate the Visit

Next, tell `Navigator` to visit your homepage. Do this by declaring a constant outside of the `SceneDelegate` class to make it accessible across the entire code-base, and creating a `URL` pointing to the local server address: `http://localhost:3000`. We'll deal with production URLs in *the (as yet) unwritten Chapter 10, Deploy to Physical Devices with TestFlight and Internal Testing, .*

ch01_08/ios/HikingJournal/SceneDelegate.swift

```
import HotwireNative
import UIKit

➤ let baseURL = URL(string: "http://localhost:3000")!

class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?

    private let navigator = Navigator()

    func scene(
        _ scene: UIScene,
        willConnectTo session: UISceneSession,
        options connectionOptions: UIScene.ConnectionOptions
    ) {
        window?.rootViewController = navigator.rootViewController
    }
}
```

There's something to notice with the `baseURL` variable: there's an exclamation point at the end of the `URL` initializer. This is because `URL(string:)` returns an optional URL. The compiler can't validate the arbitrary string we pass in at compile time, so the check happens at runtime. Adding an exclamation point

tells the compiler to *force unwrap* the optional variable, returning a non-optional URL class.

Force Unwrapping Optionals

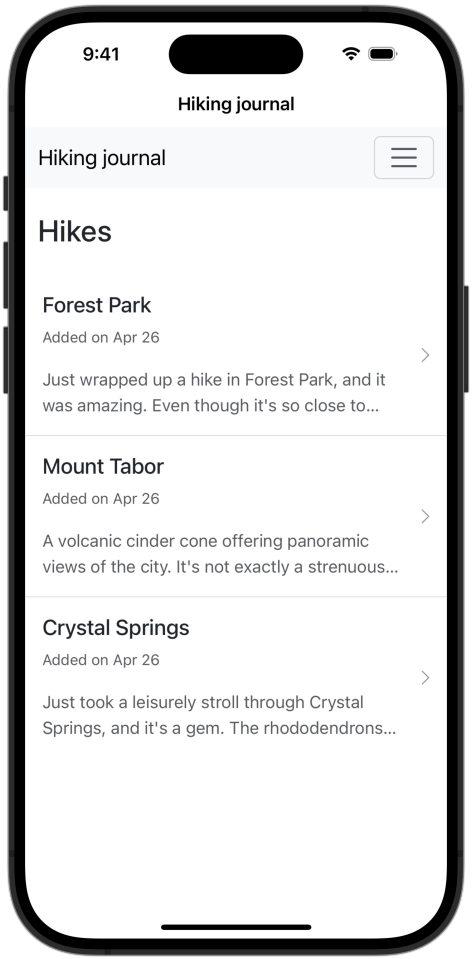
A heads up that force unwrapping a nil value will crash your app! This should only be done when you are 100% confident there won't be a nil value present. We'll touch on safer operations to access potentially nil values later in the book.

Last but not least, we need to actually visit the URL. Tell the navigator to render the homepage of the Rails app by calling route(_:). Append the /hikes path to baseURL to visit the index page for hikes.

ch01_09/ios/HikingJournal/SceneDelegate.swift

```
import HotwireNative
import UIKit

let baseURL = URL(string: "http://localhost:3000")!

class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?

    private let navigator = Navigator()

    func scene(
        _ scene: UIScene,
        willConnectTo session: UISceneSession,
        options connectionOptions: UIScene.ConnectionOptions
    ) {
        window?.rootViewController = navigator.rootViewController
➤       navigator.route(baseURL.appending(path: "hikes"))
    }
}
```

Click Product → Run or press ⌘ R to build and run the app. The homepage of the Rails app will appear in the simulator.



Sign in by opening the hamburger menu at the top of the screen and tapping "Sign in". The form is populated with credentials for the demo account so just tap the submit button to authenticate. Once signed in, add a hike via the "Add a hike" button at the bottom of the screen.

We didn't build *any* of this functionality into the Rails app. We get it all for free from Hotwire Native. Amazing!