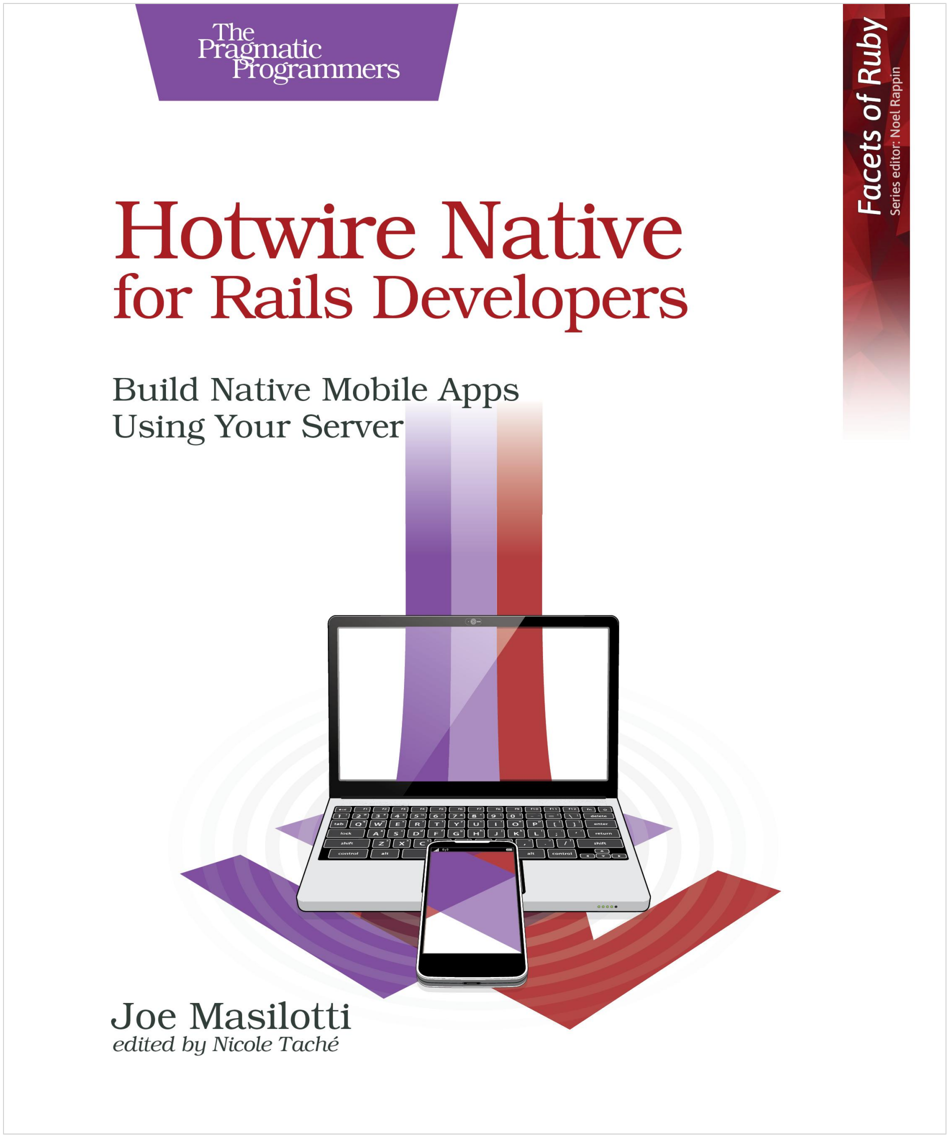

The Pragmatic Programmers

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

Route the URL via Path Configuration

Like we learned in Chapter 3, Navigate Gracefully with Path Configuration, on page ?, our path configuration helps us keep our business logic on the server. This creates low-maintenance, server-driven apps. In our scenario here, instead of hard-coding which URL paths to render as maps, we can apply path properties from our remote JSON file.

Start by adding a new rule to the path configuration on the server, matching the path for a hike's map to assign the view_controller property.

ch05_05/rails/app/controllers/configurations_controller.rb

```
class ConfigurationsController < ApplicationController
  def ios_v1
    render json: {
      settings: {},
      rules: [
        {
          patterns: [
            "/new$",
            "/edit$"
          ],
          properties: {
            context: "modal"
          }
        },
➤       {
➤         patterns: [
➤           "/hikes/[0-9]+/map"
➤         ],
➤         properties: {
➤           view_controller: "map"
➤         }
➤       }
      ]
    }
  end

  def android_v1
    # ...
  end
end
```

Like context, Hotwire Native is aware of the view_controller property, too. The framework exposes it when the user taps a new link via the NavigatorDelegate. A quick refresher from Chapter 1, Build Your First Hotwire Native Apps, on page ? that *delegates* are design pattern in iOS. They provide a way for objects to act on behalf of others to handle specific events, like when a link is tapped.

Our TabBarController is the one creating the Navigators, so that's a great place to implement NavigatorDelgate. At the bottom of TabBarController.swift, add another extension that implements the NavigatorDelegate protocol and implements this function.

ch05_06/ios/App/Controllers/TabBarController.swift

```
import HotwireNative
import UIKit

class TabBarController: UITabBarController {
    // ...
}

extension TabBarController: UITabBarControllerDelegate {
    // ...
}

➤ extension TabBarController: NavigatorDelegate {
➤     func handle(proposal: VisitProposal) -> ProposalResult {
➤     }
➤ }
```

handle(proposal:) is called every time the user taps a link. It gives us an opportunity to customize what type of screen is rendered. To do that, the function requires us to return a ProposalResult.

⌘-click on ProposalResult to jump to the definition. (This is a handy way of navigating around code in Xcode.) Here's what the contents of ProposalResult.swift show:

```
// hotwire-native-ios:Source/Turbo/Navigator/Helpers/ProposalResult.swift

import UIKit

/// Return from `NavigatorDelegate.handle(proposal:)` to route a custom controller.
public enum ProposalResult: Equatable {
    /// Route a `VisitableViewController`.
    case accept

    /// Route a custom `UIViewController` or subclass
    case acceptCustom(UIViewController)

    /// Do not route. Navigation is not modified.
    case reject
}
```

The ProposalResult class is an *enumeration*, which means it defines common types for a group of related values and enables you to work with them in a type-safe way.

The three cases are used like so:

- .accept - Route a web view for rendering web content.

- .acceptCustom - Route a custom view controller.
- reject - Cancel and ignore the proposal.

We'll use the .acceptCustom case and pass in our fancy new MapController class for map routes.

ch05_07/ios/App/Controllers/TabBarController.swift

```swift
import HotwireNative
import UIKit

class TabBarController: UITabBarController {
    // ...
}

extension TabBarController: UITabBarControllerDelegate {
    // ...
}

extension TabBarController: NavigatorDelegate {
    func handle(proposal: VisitProposal) -> ProposalResult {
➤       switch proposal.viewController {
➤       case "map": .acceptCustom(MapController(url: proposal.url))
➤       }
    }
}
```

And for everything else, .accept will render the default web view provided by Hotwire Native.

ch05_08/ios/App/Controllers/TabBarController.swift

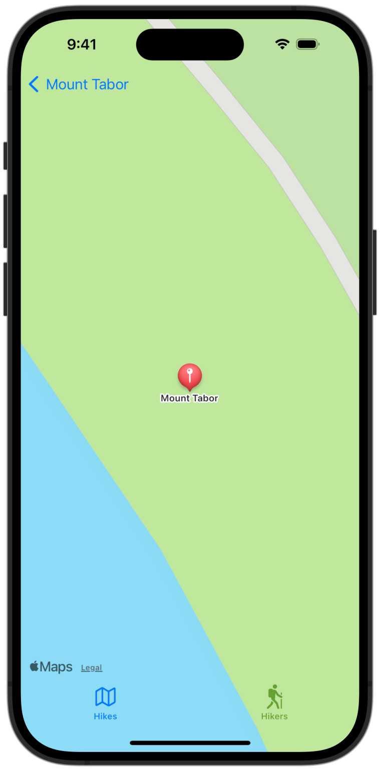
```swift
import HotwireNative
import UIKit

class TabBarController: UITabBarController {
    // ...
}

extension TabBarController: UITabBarControllerDelegate {
    // ...
}

extension TabBarController: NavigatorDelegate {
    func handle(proposal: VisitProposal) -> ProposalResult {
        switch proposal.viewController {
        case "map": .acceptCustom(MapController(url: proposal.url))
➤       default: .accept
        }
    }
}
```

Wrap up the routing by assigning self to the delegate of each Navigator instance created in makeViewControllers().

ch05_09/ios/App/Controllers/TabBarController.swift

```
import HotwireNative
import UIKit

class TabBarController: UITabBarController {
    // ...

    private func makeViewControllers() -> [UIViewController] {
        return Tab.all.map { tab in
➤           let navigator = Navigator(delegate: self)
            navigators.append(navigator)

            let controller = navigator.rootViewController
            controller.tabBarItem.title = tab.title
            controller.tabBarItem.image = UIImage(systemName: tab.image)
            return controller
        }
    }
}

// ...
```

Run the app and navigate to a hike page. When you tap the Map button, you'll see a native map!



We’ll address the satellite view a bit later. But for now, take a second to learn how to manipulate the map in the simulator. This will give you a good idea of how much better the user experience is compared to a web-based map.

Manipulating the Map

In the simulator, you can hold down the ⌥ key to create a second “finger”, which is useful for zooming in and out on the map. You can then hold down ⇧ to “stick” them together for two-finger scrolls.