
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

Prerequisites

Some tutorials and workshops at RWDevCon 2018 have prerequisites.

If you plan on attending any of these tutorials or workshops, **be sure to complete the steps below before attending the tutorial**.

If you don't, you might not be able to follow along with those tutorials.

> **Note**: All talks require **Xcode 9.2** installed, unless specified otherwise (such as in the ARKit tutorial and workshop).

11: Advanced WKWebView

You'll need NodeJS v8.x installed. The installer package is actually included in its session's materials.

Look in the "Server" folder and you'll see the pkg you can run. Also, since you'll be running a NodeJS Server in this directory, it's recommended that you copy the materials (demo folders and the server folder) to somewhere on your Mac's drive.

11: Advanced WKWebView

Web Kit's WKWebView is Apple's replacement for the limited UIWebView control and it's more than just for "web browser apps". Introduced with iOS 8, WKWebView has the same power as the built-in Safari App but as a public API. Performance optimizations, such as the Nitro JIT JavaScript engine, make WKWebView a powerful platform to display HTML and CSS (and to interpret javascript) easily within your apps.

With Web Kit you can re-use content coming from a web CMS or local html file, even share existing JavaScript logic! In this session, you will implement WKWebView to load both local and remote content that interacts with your app seamlessly. You will learn how to properly use CSS and custom fonts with your local content, how to intercept hyperlink taps (to redirect to in-app actions) and more.

31 Advanced WKWebView: Demo 1

By Eric A. Soto

In this demo, you will add a WKWebView to the Demo1 app and use it to fetch html from a local node server. Also, since we need a web server to access, we're going to install NodeJS and fire up a local node server.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work in the **Server** directory of the materials for this tutorial.

1) Install/verify NodeJS

Open up a new terminal window and switch to the **Server** directory. Let's check to see if you already have NodeJS installed.

```
node --version
```

> **Note**: An easy way to get the path of the **Server** directory is to drag that folder into terminal. Type `cd` then drag the directory over and terminal will add the complete path to your command line.

If you have NodeJS version 8 or better, skip to the next step!

To install NodeJS, we've already provided the installer package. Still on the **Server** directory, open the installer package:

```
open node-v8.9.0.pkg
```

Follow the prompts to install NodeJS.

2) Start up the Deskbook server

Now that we have NodeJS running, let's start our server.

First, we change into the **DeskbookServer** directory, then we start up the server.

Type the following commands into terminal:

```
cd DeskbookServer
./xStartDev.sh
```

This will start up the NodeJS server and you should see something like:

```
> deskbookserver@0.0.0 start-dev /your/path/to/DeskbookServer
> nodemon ./bin/www

[nodemon] 1.12.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./bin/www`
```

Then, open up a web browser and visit http://127.0.0.1:3000/. Welcome to Deskbook! Note, clicking on any row takes you to the profile page for the staff member.

If you look back at your terminal, you'll see that NodeJS outputs console information as you interact with the app. We don't really need to pay attention to that, so feel free to minimize terminal (but don't close it because we need that process running!)

3) Becoming familiar with Deskbook

Open up the **Demo1/starter/Deskbook** project by double-clicking on **Deskbook.Xcodeproj**. This should open up Xcode with our starter app.

Select an iOS Simulator then Build & Run.

The list of staff appears and you can tap on any staff member to see their details. Notice the **bio** shows raw html! Not good!

If you don't see the list, make sure the NodeJS server is running.

4) Change the bio to a WKWebView

Open **Main.storyboard** and go to the **Staff Member Scene**.

In the Document Outline, expand: **Staff Member**, **View**, **Stack View**, **Bio Label**.

Select the **Bio:** label and change it to "More Information".

Expand: **Staff Member**, **View**, **Stack View**, **Bio Row**, **View**.

Select **Bio Text** and delete it. (Click on **Bio Text** and press your delete key.)

From the **Object Library**, filter for "WebKit" and drag a **WebKit View** into the view that had our Bio Text earlier.

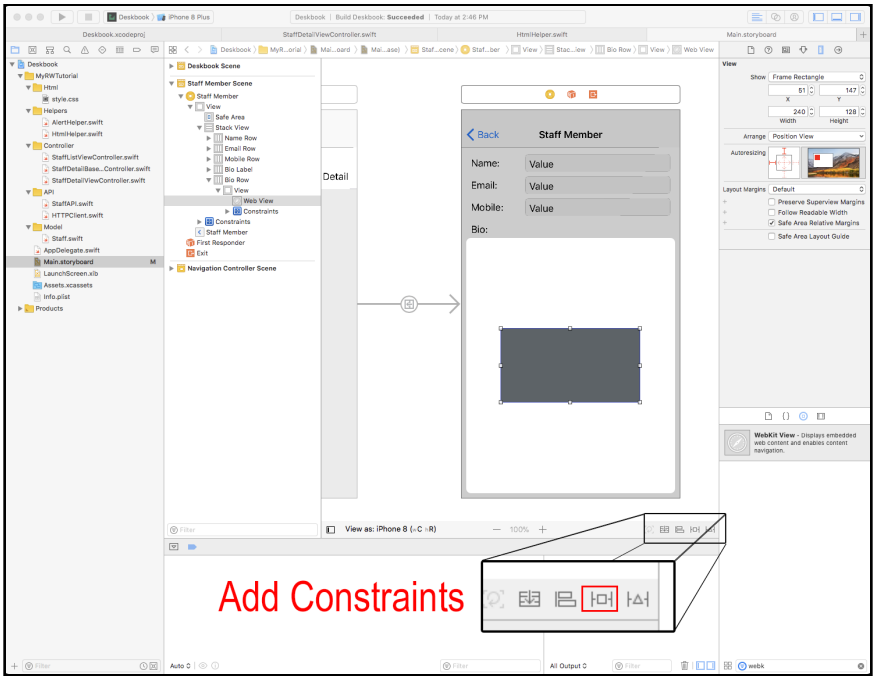
Rename it to **Bio Web View** for easy identification later.

5) Add constraints to the web view

Now, let's add a few constraints for the Bio Web View.

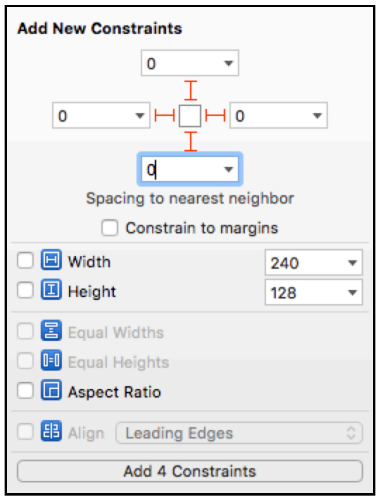
Click on **Bio Web View** so that it is selected.

You want to add several constraints to the WKWebView to size it properly in our **More Information** area. To do this, with the web view selected, click on the **Add Constraints** button towards the bottom of the canvas.



Add Constraints

On the **Add New Constraints** pop-up, enter `0` in the top, right, bottom, left input areas. Then, click on **Add 4 constraints** to actually add the constraints.



Add Constraints

> **Note**: There seems to be a bug in Xcode Interface Builder and after the constraints update, the Web View does not appear to take up the entire view that it is constrained to. Ignore that as later on once IB repaints, this goes away and it displays properly. In any case, it functions fine even with this glitch.

6) Connect the Web View to our class

Still in interface builder, show the **Assistant editor** and ensure it is pointing to **StaffDetailViewController.swift**. The **Assistant editor** can be opened by clicking on the **Show the Assistant editor** icon on the top right of your Xcode window.

Scroll the file to where the other IBOutlets are grouped.

Control-drag from the **Bio Web View** to the file to add a new IBOutlet for the web view. Call it `bioWebView` and click **connect**.

Ignore any errors in code for now. We'll fix them next.

7) Use the new Web View

Switch back to the standard editor and open **StaffDetailViewController.swift**. Add the WebKit import at the top:

```
import WebKit
```

Any code errors should now disappear.

Delete the line:

```
@IBOutlet weak var bioText: UITextView!
```

Find the **updateStaffValues** method and change the bio text code:

```
// From
self.bioText.text = self.staffMember.bio

// To
self.bioWebView.loadHTMLString(self.staffMember.bio, baseURL: nil)
```

Build & Run and go into any staff detail. Hm... the html is not great.

8) Use Safari Developer Tools

A very cool tool for working with WebKit Web Views is **Safari Developer Tools**, which we can use to inspect the HTML and debug css issues.

With the app running on Simulator or your device, open Safari on your Mac.

Click **Develop** on the Safari menu bar.

> **Note:** Don't see **Develop**? Check your Safari Preferences, under **Advanced** and make sure you have the option to "Show Develop Menu in Menu Bar".

Notice you have one item per each device connected as well as one item for **Simulator**. Select whichever you're using (your device or simulator), then **Deskbook about:blank**.

You should see the typical web inspector, but for your WKWebView.

Make sure you're on the **Elements** tab and Notice the `<Head>` tag is empty! (You can't expand it nor see anything inside.) This means that structure is missing. If you've ever done any HTML, you know that "Normal HTML" has some basic structure that is missing in our web view right now.

9) Fix the HTML

"Proper" html needs a few things. Let's fix the HTML so it has the right structure (we'll get into this more in the next demo).

Open **HtmlHelper.swift** (in the **Helpers** group) and uncomment the following static method to the class:

```
  static func wrap(html: String, withClass wrapClass: String = "") ->
String {
    // "Proper" Html needs some things:
    // <head></head><body></body>
    // in <head> ->
    //    <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
    //    <style></style>
    let wrappedHtml = """
      <head>
        <title>Deskbook WebView</title>
        <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
        \(getCss())
      </head>
      <body>
        <div class="\(wrapClass)">\(html)</div>
      </body>
    """
    return wrappedHtml
  }
```

This will wrap any HTML we pass it with the proper structure, even including the CSS in our bundle.

Now, let's use this new method to "wrap" our html. Back in **StaffDetailViewController.swift** change the loadHTMLString to:

```
self.bioWebView.loadHTMLString(HtmlHelper.wrap(html:
self.staffMember.bio, withClass: "bio"), baseURL: nil)
```

Build & Run and go into any staff detail. Yes! The html looks great. In the next demo, we're going to make it even better.

However, click on the Twitter link. Oh no! That's not great! We're going to deal with that and more next.

10) That's it!

Congrats, at this time you should have a basic understanding of WKWebView! It's time to move into more advanced uses.

32 Advanced WKWebView: Demo 2

By Eric A. Soto

In this demo, you will enhance the WKWebView to stylize the html better, intercept links and redirect them for a better experience. You're going to continue using the NodeJS service we used before.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo2\starter**. It's important to use the **Demo2** because it's slightly different than the previous demo.

1) Start up the Deskbook server

If you stopped the Deskbook server from Demo1, you want to start it back up. If you still have it running, great! Skip this step.

Remember, you want to start from the **Server** directory of the materials for this tutorial.

```
cd DeskbookServer
./xStartDev.sh
```

This will start up the NodeJS server and you should see something like:

```
> deskbookserver@0.0.0 start-dev /your/path/to/DeskbookServer
> nodemon ./bin/www

[nodemon] 1.12.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./bin/www`
```

2) Use Safari Developer Tools

Open up the **Demo2/starter/Deskbook** project by double-clicking on **Deskbook.Xcodeproj**. This should open up Xcode with our starter app.

Build & Run to Simulator, then let's have another look now that we have structure. With the app running on Simulator or your device, open Safari on your Mac.

Once again, click **Develop** on the Safari menu bar.

Notice you have one item per each device connected as well as one item for **Simulator**. Select whichever you're using (your device or simulator), then **Deskbook Deskbook WebView**.

You should again see the typical web inspector, but for your WKWebView.

> **Note**: If you don't see **Simualator** in Safari, be sure you're not running a BETA of Xcode. Also, if you don't see HTML, be sure to click on the **Elements** tab on the inspector.

Expand **Head**, and **Style** and notice your CSS is indeed in there now! Awesome!

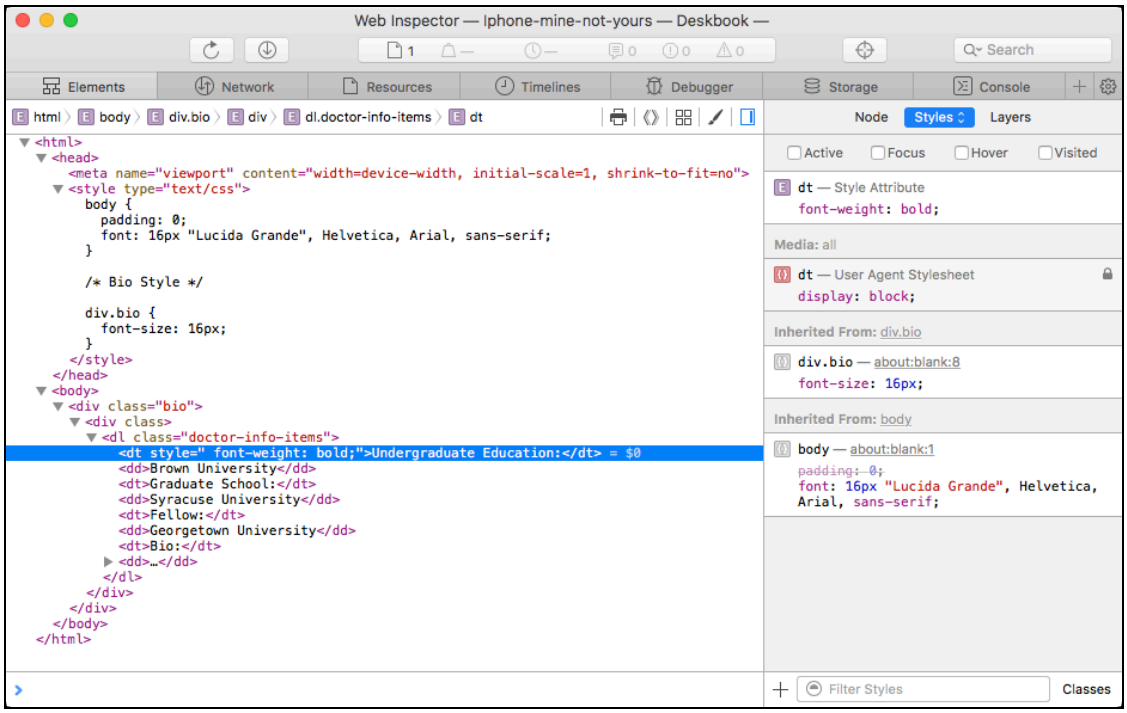
Let's experiment with some CSS to make the content look better.

Expand **Body** and keep expanding to inspect the elements that make up your content. Notice how the "titles" for each of the bio items uses a `<dt>` tag, but the title is not standing out. Let's fix that.

Click on one of the `<dt>` lines and on the Styles editor, click into **dt - Style Attribute** and add:

```
font-weight: bold;
```

Once you're done, your web inspector should look something like this:



Web Inspector

Now, look back at your device or simulator and notice that the label you had selected is now **Bold** and indeed stands out better. This confirms that if we bold the `<dt>` tags, the content looks better in general.

Now add this back into your App styles.

3) Edit the CSS

Back in Xcode, all the CSS for the app is in **style.css**. Open that file and add:

```
div.bio dt {
  font-weight: bold;
}
```

Build & Run and go into any staff detail. Notice it now looks much better as the bold makes the titles stand out more.

Unfortunately, we still have a problem with the link to Twitter. Tap on it so that you see what it looks like.

Let's deal with that next.

4) Implement the WKNavigationDelegate

Open **StaffDetailViewController.swift**. At the very bottom let's add an extension that implements our `WKNavigationDelegate`:

```
extension StaffDetailViewController: WKNavigationDelegate {

  func webView(_ webView: WKWebView, decidePolicyFor navigationAction:
WKNavigationAction, decisionHandler: @escaping (WKNavigationActionPolicy)
-> Void) {

    // You will implement logic here

  }

}
```

In this case, you're using `webView(_:decidePolicyFor navigationAction:decisionHandler:)` which allows you to allow or block the link based on its URL. Take note that there is another version of this method `webView(_:decidePolicyFor navigationResponse:decisionHandler:)` that instead uses the response of the link. In our case, the URL is sufficient so that's what you'll use. Let's do some more with this.

5) Add logic to the Navigation handler

Next, let's add some basic code for the handler.

Inside `webView(_:decidePolicyFor navigationAction:decisionHandler:)` add:

```
    // 1
    guard let urlTarget = navigationAction.request.url else {
      decisionHandler(.cancel)
      return
    }

    // 2
    let urlString = urlTarget.absoluteString

    // 3
    if urlString == "about:blank" {
      decisionHandler(.allow)
      return
    }

    // 4
    decisionHandler(.cancel)
```

Let's review what this code does:

1. First, you do some sanity checks. Check to see if you did not receive a URL or perhaps it's invalid. If it's empty or invalid, you exit and reject the navigation action URL (you don't allow the link to load at all.) Notice the `decisionHandler()` closure is how you communicate back to WKWebView to allow or disallow a

load.

2. Next, you extract the URL as a string so it can be inspected later a little easier.
3. Then, you check the navigation action link to see if it's `about:blank`, which is how the WKWebView loads initially (as a blank Web View before we inject HTML into it.) If you don't allow this, the WKWebView won't load at all! Don't forget to `return` right after this since we've already made a decision and we must only call the `decisionHandler()` once or our app will crash.
4. Finally, since you must respond explicitly with, for every other scenario, we'll simply disallow the navigation. This way, nothing can "sneak in" on us that we would not want to display!

> **Note**: `webView(_:decidePolicyFor navigationAction:decisionHandler:)` requires you to call the `decsionHandler()` or your app will crash. Allow or disallow, you must communicate a decision and you must do so only once!

6) Set our WKWebView delegate

Now, you need to set the delegate of the WKWebView so that the protocol implementation is used. Back in `viewDidLoad()`, add the following just above the other delegate declarations:

```
        bioWebView.navigationDelegate = self
```

This sets the navigation delegate for your WKWebView to the current controller so that you can receive its navigation events.

Now, Build & Run.

But nothing changed you say! Indeed. You wired up an intercept of the Navigation but you're not quite doing anything with it. However, the fact that the app built and that you still are able to click on Twitter is a good sign that our implementation is headed in the right direction.

7) Open Safari to show Twitter

Begin at the top of **StaffDetailViewController.swift**, add an import statement at the top to bring in the **Safari View Controller**:

```
import SafariServices
```

Next, inside the `StaffDetailViewController` class, just under the `startCall()` method, add another method to open a **Safari View Controller** when we pass it a

URL:

```
    fileprivate func openWithSafariVC(url: URL) {
      // 1
      let safariVC = SFSafariViewController(url: url)
      // 2
      safariVC.modalPresentationStyle = .overFullScreen
      // 3
      self.present(safariVC, animated: true, completion: nil)
    }
```

Reviewing this code we see:

1. You instantiate an `SFSafariViewController` with a URL passed as an argument to the initializer.
2. For a more iOS consistent UI, since the Safari View Controller is not part of our Navigation controller, we want to show it modally. This takes care of displaying correctly as a modal and animates from the bottom.
3. You present the **Safari View Controller** which adds that view to the view stack.

8) Intercept Twitter

Let's implement logic to trap a link to Twitter and then do something with it.

Back in `webView(_:decidePolicyFor navigationAction:decisionHandler:)`, replace the last `decisionHandler(.allow)` with the following:

```
      // 1
      if urlString.hasPrefix("https://twitter.com")  {
        // 2
        decisionHandler(.cancel)
        // 3
        openWithSafariVC(url: urlTarget)
        return
      }
      // 4
      decisionHandler(.cancel)
```

1. Check to see if the navigation link is for Twitter.
2. Since you want to handle Twitter, you actually CANCEL the navigation on the WKWebView. You don't want the web view to handle these.
3. Now that you have canceled the navigation, you better handle it with something else. This is were we hook up the `openWithSafariVC` we created earlier. Notice that after this line, you exit the method. Also, you've already done `.cancel` so the requirement of the protocol closure has been met.
4. Finally, you disallow anything else. Why? You want to keep tight control of what appears in the WKWebView. If it's not something you explicitly want and allow, then it should not actually navigate.

Build and Run and tap on any user to see their detail view. Then, tap on the **Twitter** link for the user.

Twitter in a SFSafariViewController

Profit! Twitter (and any links of your choosing) can now be intercepted and redirected in any way you want.

9) That's it!

Congrats, at this time you should have a good understanding of WKWebView and intercepting navigation actions so you can redirect them. Take a break, and then it's time to move onto more WKWebView customization.

33 Advanced WKWebView: Demo 3

By Eric A. Soto

In this demo, you will enhance the WKWebView to load the styles and custom fonts included in the bundle. You're going to continue using the NodeJS service we used before.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

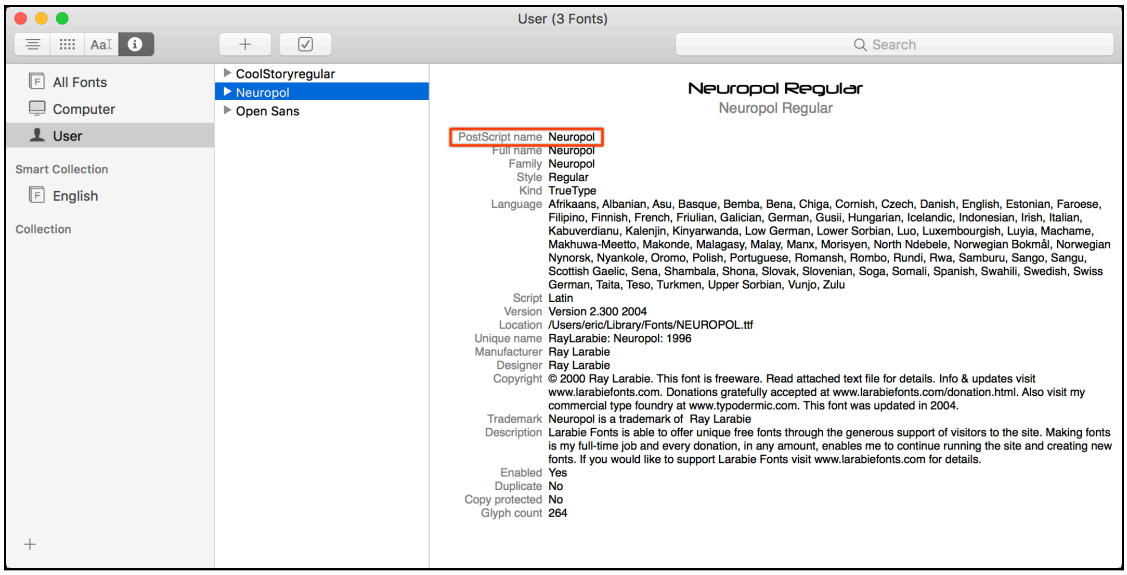
> **Note**: Begin work with the starter project in **Demo3/starter**. It's important to use **Demo3** because it's slightly different than the previous demo (it includes the font files you'll need).

1) Install the font on your mac

In order to use the font in Interface Builder, you need to install it on your mac. Navigate to **Demo3/starter/fonts** and double click on **NEUROPOL.ttf**. This will launch **Font Book**.

Click **Install Font** to install the font on your mac.

Once installed, inspect **User Fonts** and notice **Neuropol** shows on the list. Click on the **Font Info** button and take special note of the **PostScript name** because we'll need that later:



Font Book

2) Bring the font file into your project

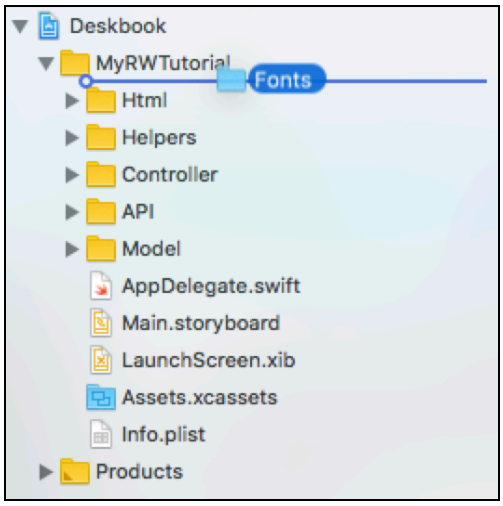
You also need to bring the font file into your project so that Xcode knows to include it in the bundle.

Open up the **Demo3/starter/Deskbook** project by double-clicking on **Deskbook.Xcodeproj**. This should open up Xcode with our starter app.

Then, switch over to **Finder** and navigate to **Demo3**. Copy the entire **Fonts** directory (click on the directory and press CMD+C).

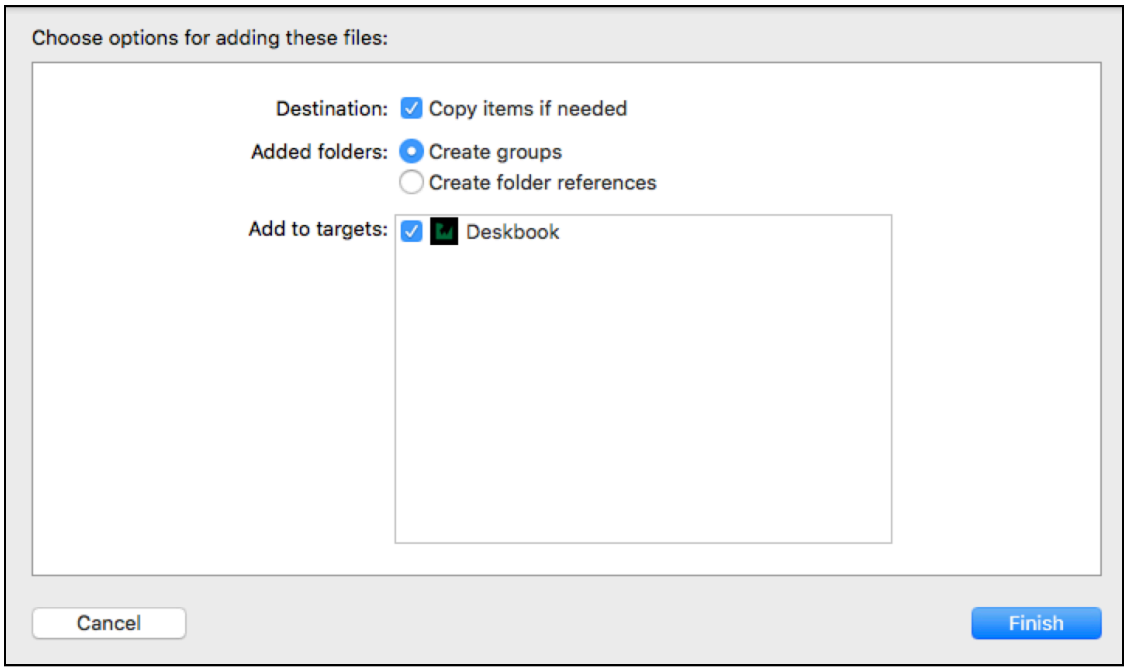
Still in **Finder**, navigate over to **Demo3/starter/Deskbook/MyRWTutorial** and paste in the entire **Fonts** directory.

Now with the files in the right spot, put **Finder** and **Xcode** windows side by side so you can drag from one to the other. Drag the directory **Demo3/starter/Deskbook/MyRWTutorial/Fonts** into the **Project navigator** in Xcode.



Drag the Fonts Directory

In the dialog **"options for adding these files"** be sure you've checked **Copy items if needed**, **Create groups** and that the **Deskbook** target is checked.



Options for adding files

Great, now the font file is part of your Xcode project and will be included in the bundle when you build.

3) Add the font to info.plist

Before we can use the font in our app, we must tell Xcode about it.

Click on **Info.plist** to open it.

Click on the **+** icon next to **Information Property List** to add a new entry. Edit the new entry to select **Fonts provided by application**.

Click on the **>** (disclosure triangle) next to the new **Fonts provided by application** to expand it.

On **Item 0**, add the name of your font file: **NEUROPOL.ttf**, taking care to ensure spelling and capitalization match the font file name. The entry here must be the *file name* including the extension.

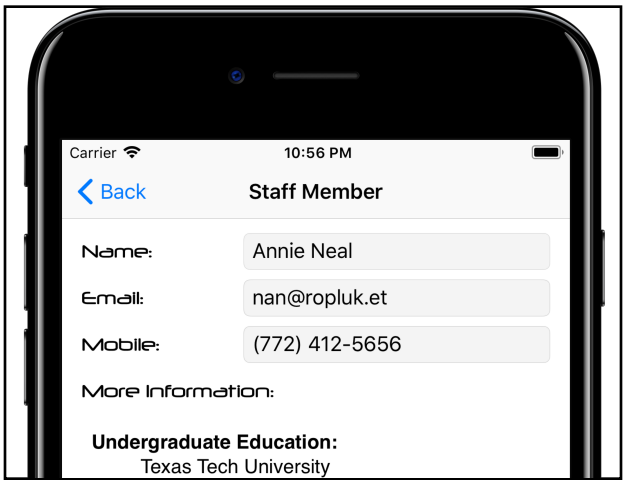
4) Use the new font in Interface Builder

Open **Main.storyboard** and navigate to the **Staff Member** scene.

Click on the **Name** label, then go to the **Attributes inspector**. In the **Font** selector, click **Custom**, then in **Family** select **Neuropol**. You should see the font change in the scene.

Repeat for the other three labels in that scene (Email, Mobile, More Information).

Make sure your server is running and Build & Run. You should see the custom font in the labels:



Custom Fonts on the App

> **Note:** If you don't see the custom font, double check the file name you provided in **Info.plist** (it should be **NEUROPOL.ttf**).

5) Setup Base Path for the WKWebView

Now that the font is properly wired up into the project, let's set the foundation to be able to use it inside the WebKit Web View.

Open **StaffDetailViewController** and replace the line in `updateStaffValues()` that loads the WKWebView html as follows:

```
self.bioWebView.loadHTMLString(HtmlHelper.wrap(html:
self.staffMember.bio, withClass: "bio"), baseURL: Bundle.main.bundleURL)
```

The interesting part of this new line is `baseURL: Bundle.main.bundleURL`.

6) Wire up the font in the CSS

Next, you need to connect the custom font in your CSS file also.

Open **Html/style.css** and add the `@font-face` reference, after the closing `/* Bio Style */` comment:

```
/* 1 */
@font-face {
  /* 2 */
  font-family: Neuropol;
  /* 3 */
  src: url('NEUROPOL.ttf');
}
```

What's going on here?

1. The `@font-face` CSS rule allows you to specify a font to display the text in your HTML. Generally used for loading font files from a web server, in our case, we'll use it to load a font-file from our Bundle.
2. In the `@font-face` rule, **font-family** must match the **Postscript name** of the font.
3. Finally, `src` tells the web view where it should locate the font file. In our case, we're just passing a name of a font-file, so the WKWebView's **Base Path** is going to be used. Once again, capitalization and spelling matter!

Basically, you're telling WKWebView (via CSS) to load a custom file included in your bundle with a specific file name and font family.

7) Apply the font to our Html labels

Still in **Html/style.css**, edit the style for `div.bio dt` so that it uses the font. The edited style should look like this:

```
div.bio dt {
```

```
    font-weight: bold;
    font-family: Neuropol;
}
```

8) Re-factor the way we import our CSS

Let's fix how we bring in the CSS. Previously, we were injecting it into the `<HEAD>` of the HTML inside a `<style>` tag. Though that worked, it's not the best and it causes a bit of "blink" when rendering the content.

Let's fix this!

With **Base Path** set previously, we can actually import the CSS for the app using an HTML reference straight from the bundle.

Open **Helpers/HtmlHelper** and in the `wrap(html:withClass)` method, change the `wrappedHtml` line as follows.

Change the line that has:

```
\(getCss())
```

Into:

```
<link href="style.css" rel="stylesheet" type="text/css">
```

All you are doing here is referencing **style.css** as a standard CSS file include instead of "injecting" it into a `<style></style>` tag.

Build & Run.

Wooh wooh! Oh no, where's the content?

9) Allow bundle files

If you recall our prior demos, `webView(decidePolicyFor navigationAction:decisionHandler:)` gets to control not just links, but every URL load in the WKWebView. The code we added earlier will disallow by default, therefore, we have to add code that allows bundle resources to load.

We need to change the navigationAction so that it allows files inside the bundle.

Back in **Controller\StaffDetailViewController**, scroll to `webView(_:decidePolicyFor navigationAction:decisionHandler:)`. Change the block that begins with `if urlString == "about:blank"` to add another condition:

```
    // 1
    if urlString.hasPrefix(Bundle.main.bundleURL.absoluteString) ||
urlString == "about:blank" {
      decisionHandler(.allow)
      return
    }
```

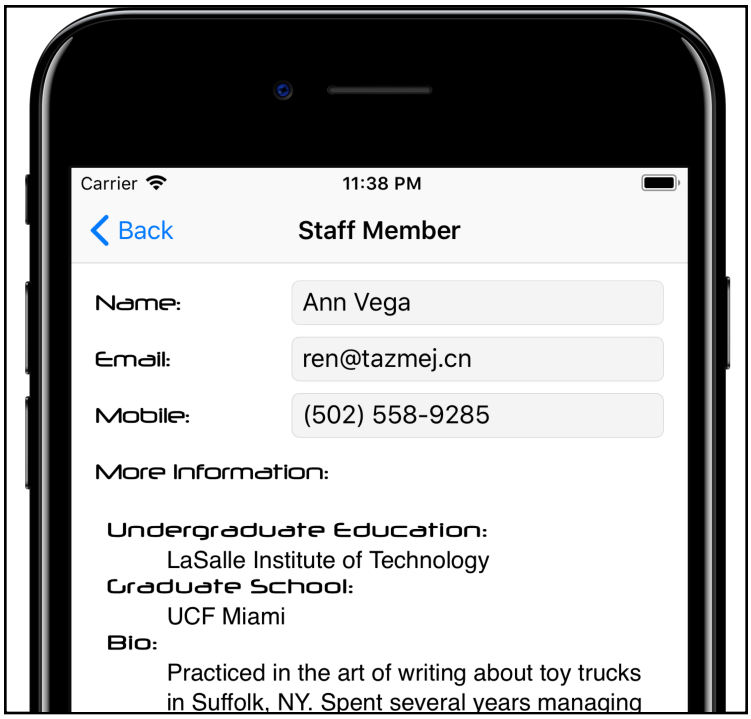
What's going on here? You check to see if the Navigation Target is the bundle path (which is how the WKWebView loads initially), or that the Navigation Target starts with the bundle path (which is how we load bundle resources). If either of these is true, we want to allow the Navigation action.

> **Note:** What is in `Bundle.main.bundleURL.absoluteString`? Actually, `bundleURL` (when paired up with `absoluteString`) gives us the same as `"file://" + Bundle.main.bundlePath`, which is the absolute location of your App's bundle in the file system.
>
> Sure, but what is it? For example, when running on my Mac with Simulator, I see `file:///Users/eric/Library/Developer/CoreSimulator/Devices/28207CF3-D406-4BA1-80B1-FD404894FFB3/data/Containers/Bundle/Application/BB0A1F8C-1F97-4201-96EB-13612ABD2AC9/Deskbook.app/`. The `file://` part is standard notation in HTML for a local file. Compare this to `http://` and `https://` for remote URLs.

Now, you will be able to load the css file, the font file and any other resources you choose to include with the bundle!

Build & Run.

Profit yet again. The custom font is a GO!



Custom Fonts, WKWebView

10) Add an image to our staff card

Just like you can bring in the font and style files, you can also include images from the bundle.

You're going to do something simple. You're going to add a simple badge to the contact card. Though we won't add any logic (we'll add this for everyone), it would be simple to add the badge to certain staff members based on some other logic.

Notice that the project already has a badge in **Html/badge.png**.

Edit **Html/style.css** to add a new section:

```
div.showBadge {
  background: url("badge.png");
  background-repeat: no-repeat;
  background-size: 100px 100px;
  background-position: right top;
}
```

All you're doing here is adding a background image (coming from the bundle since you're not providing a path). The CSS adds this to a `<div>` with class `showBadge`. Since we don't have that anywhere yet, let's add that class to the staff card `<div>`.

The good news is that we already have a hook to add the class to the div. Back in **Controller/StaffDetailViewController** in `updateStaffValues()`, change the

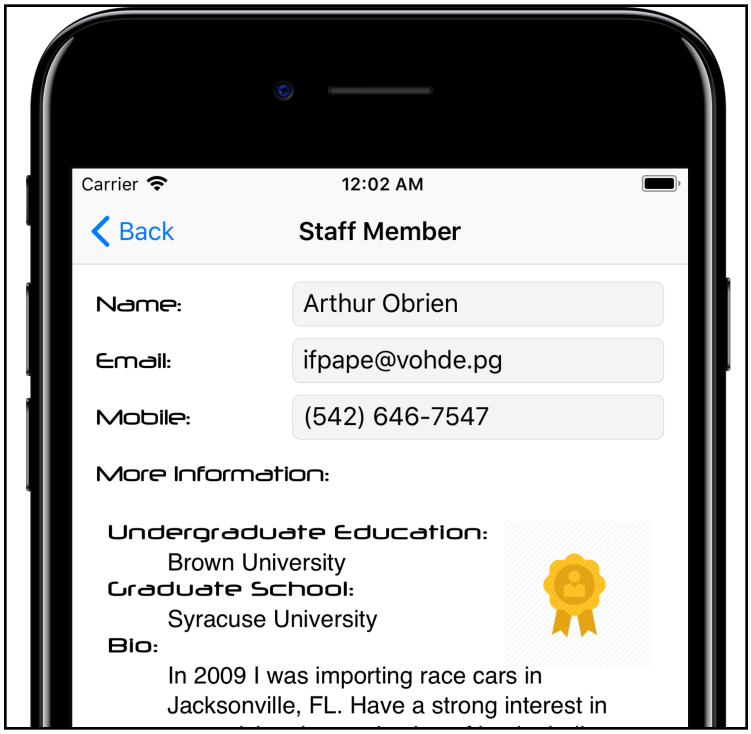
`loadHtmlString` line to:

```
self.bioWebView.loadHTMLString(HtmlHelper.wrap(html:
self.staffMember.bio, withClass: "bio showBadge"), baseURL:
Bundle.main.bundleURL)
```

Notice how we changed `withClass:` to "bio showBadge", essentially adding a second class to the `<div>`.

Build and Run.

Winning! Gold badge for all!



WKWebView, Image from Bundle

11) That's it!

Congrats, at this time you should have a good understanding of WKWebView and how you can bring in styles, fonts, images and other files bundled with your app.