How to Create a Framework for iOS

Learn how to create a framework for iOS, allowing you to elegantly and efficiently package up and redistribute your code across different apps and teams

5/5 1 Rating

In the previous tutorial, you learned how to create a reusable knob control. However, it might not be obvious how to make it easy for other developers to reuse it.

One way to share it would be to provide the source code files directly. However, this isn't particularly elegant. It exposes implementation details you might not want to share. Additionally, developers might not want to see everything, because they may just want to integrate a portion of your brilliant code into their own codebase.

Another approach is to compile your code into a static library for developers to add to their projects. However, this requires you to distribute public header files in tandem, which is awkward at best.

You need a simpler way to compile your code, and it needs to be easy to share and reuse across multiple projects. What you need is a way to package a static library and its headers in a single component, which you can add to a project and use immediately.

Good thing that's the focus of this tutorial. What you'll learn will help you solve this exact problem, by making use of frameworks. OS X has the best support for them because Xcode offers a project template which includes a

default build target and can accommodate resource files such as images, sounds and fonts. You can create a framework for iOS, but it's a touch trickier; if you follow along you'll learn how to work around the many roadblocks.

As you work through this tutorial, you'll:

- Build a basic static library project in Xcode
- Build an app with a dependency on this library project
- Discover how to convert the static library project into a fully-fledged framework
- Finally, at the end, you'll see how to package an image file to go along with your framework in a resource bundle

Getting Started

The main purpose of this tutorial is to explain how to create a framework for use in your iOS projects, so unlike other tutorials on the site, there will only be a small amount of Objective-C code used throughout the tutorial, and this is mainly just to demonstrate the concepts we'll cover.

Start by downloading the source files for the `RWKnobControl` available here. As you go through the process of creating the first project in the section ***Creating a Static Library Project***, you'll see how to use them.

All of the code, and project files you'll create whilst making this project are available on Github, with a separate commit for each build-stage of the tutorial.

What is a Framework?

A framework is a collection of resources; it collects a static library and its

header files into a single structure that Xcode can easily incorporate into your projects.

On OS X, it's possible to create a Dynamically Linked framework. Through dynamic linking, frameworks can be updated transparently without requiring applications to relink to them. At runtime, a single copy of the library's code is shared among all the processes using it, thus reducing memory usage and improving system performance. As you see, it's powerful stuff.

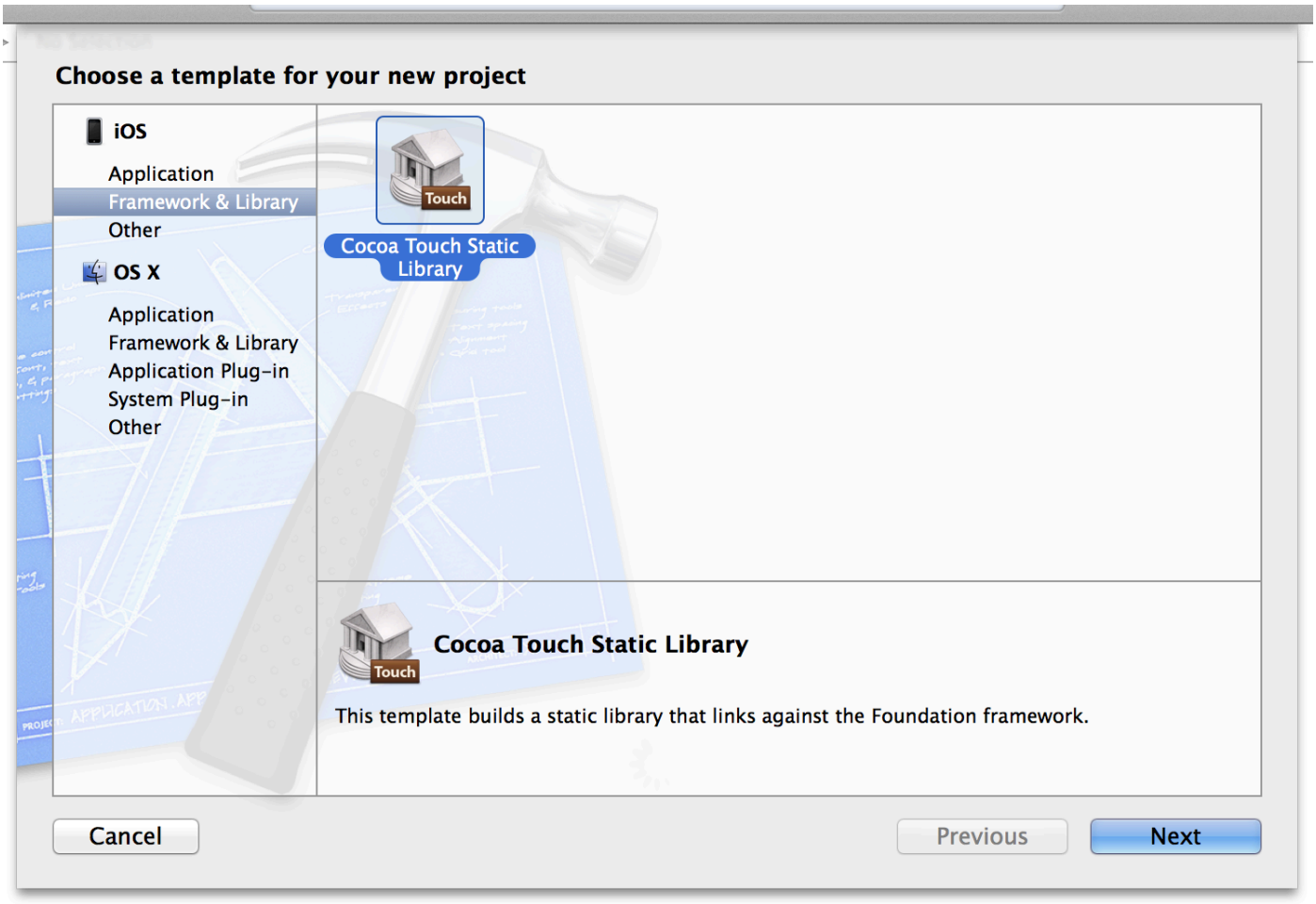
On iOS, you cannot add custom frameworks to the system in this manner, so the only dynamically linked frameworks are those that Apple provides.

However, this doesn't mean frameworks are irrelevant to iOS. Statically Linked frameworks are still a convenient way to package up a code-base for re-use in different apps.

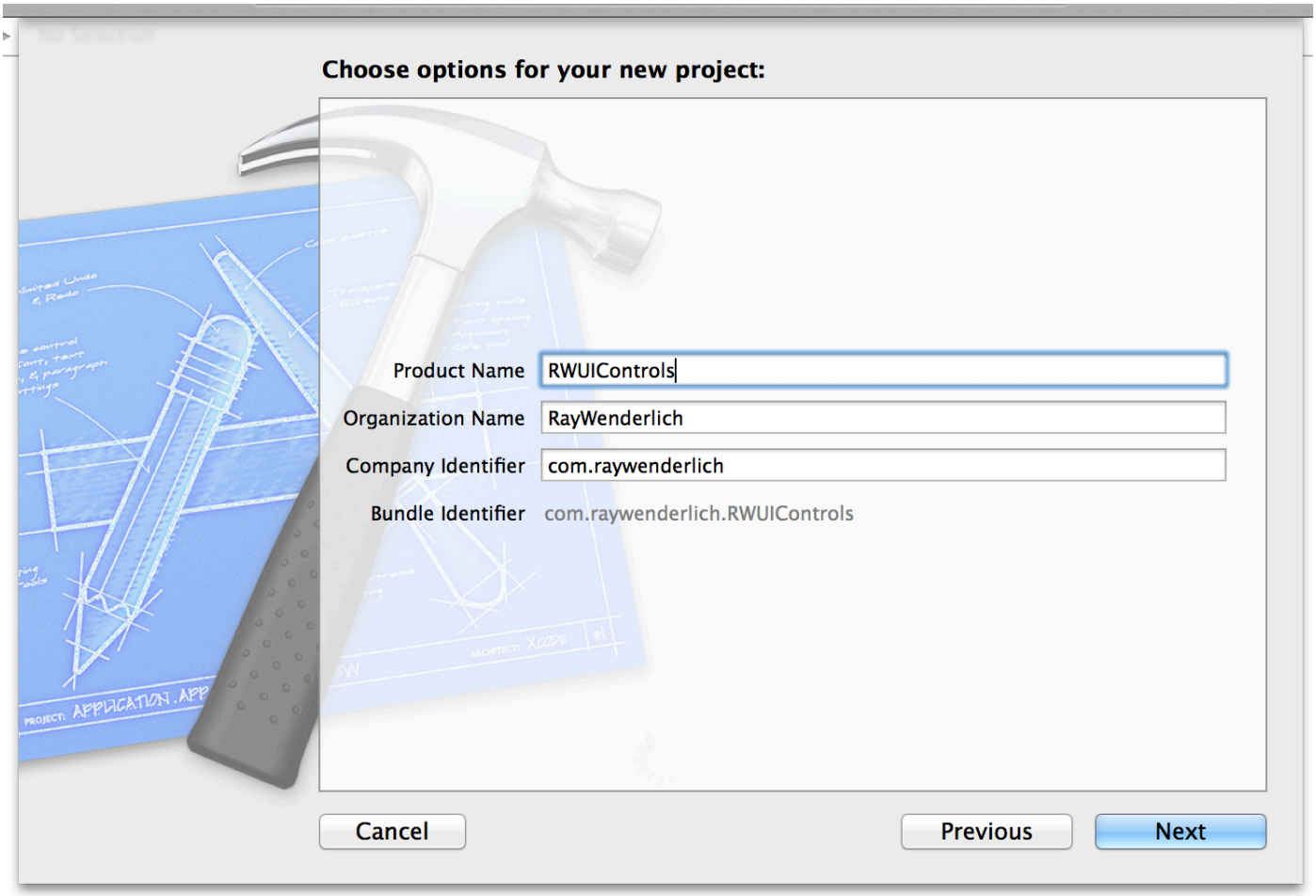
Since a framework is essentially a one-stop shop for a static library, the first thing you'll do in this tutorial is learn how to create and use a static library. When the tutorial moves on to building a framework, you'll know what's going on, and it won't seem like smoke and mirrors.

Creating a Static Library Project

Open Xcode and create a new static library project by clicking ***File|New|Project*** and selecting ***iOS|Framework and Library|Cocoa Touch Static Library***.



Name the product `RWUIControls` and save the project to an empty directory.



A static library project is made up of header files and implementation files, which are compiled to make the library itself.

To make life easier for developers that use your library and framework, you're going to make it so they only need to import a single header file to access all the classes you wish to make public.

When creating the static library project, Xcode added ***RWUIControls.h*** and ***RWUIControls.m***. You don't need the implementation file, so right click on ***RWUIControls.m*** and select ***delete***; move it to the trash if prompted.

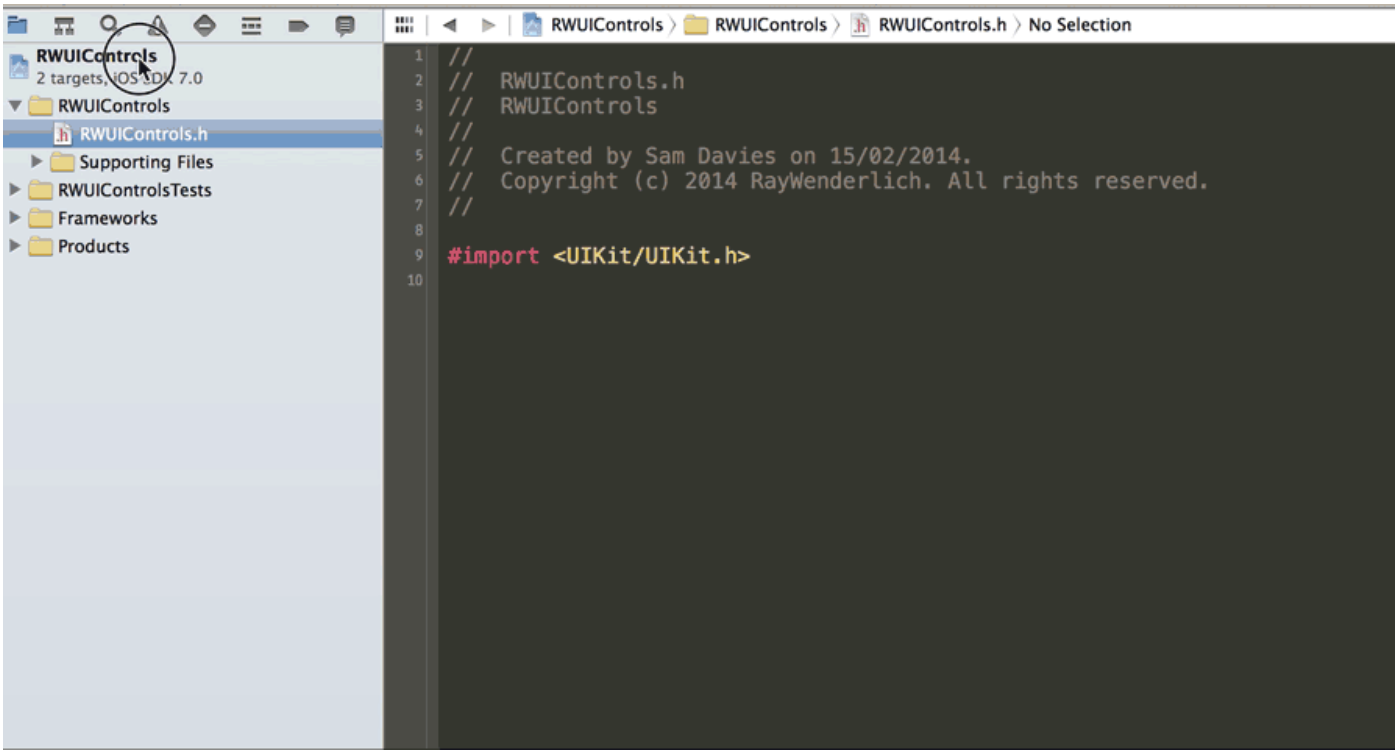
Open up ***RWUIControls.h*** and replace the contents with the following:

```
#import <UIKit/UIKit.h>
```

This imports the umbrella header of ***UIKit***, which the library itself needs. As you create the different component classes, you'll add them to this file, which ensures they'll be accessible to the libraries users.

The project you're building relies on ***UIKit***, but Xcode's static library project doesn't link against it by default. To fix this, add ***UIKit*** as a dependency. Select the project in the navigator, and in the central pane, choose the ***RWUIControls*** target.

Click on ***Build Phases*** and then expand the ***Link Binary with Libraries*** section. Click the ***+*** to add a new framework and navigate to find ***UIKit.framework***, before clicking ***add***.

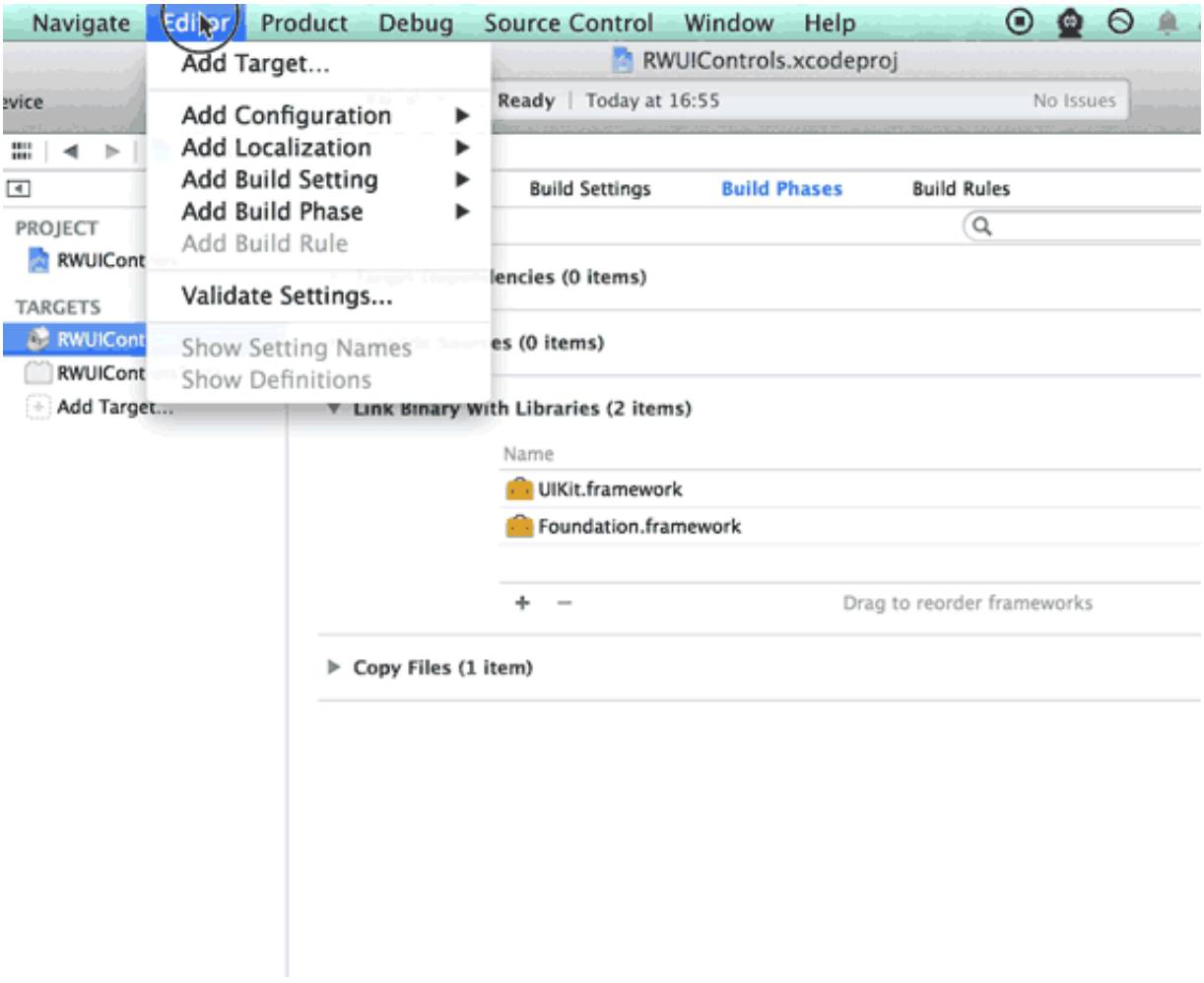


A static library is of no use unless it's combined with header files. These compile a manifest of what classes and methods exist within the binary. Some of the classes you create in your library will be publicly accessible, and some will be for internal use only.

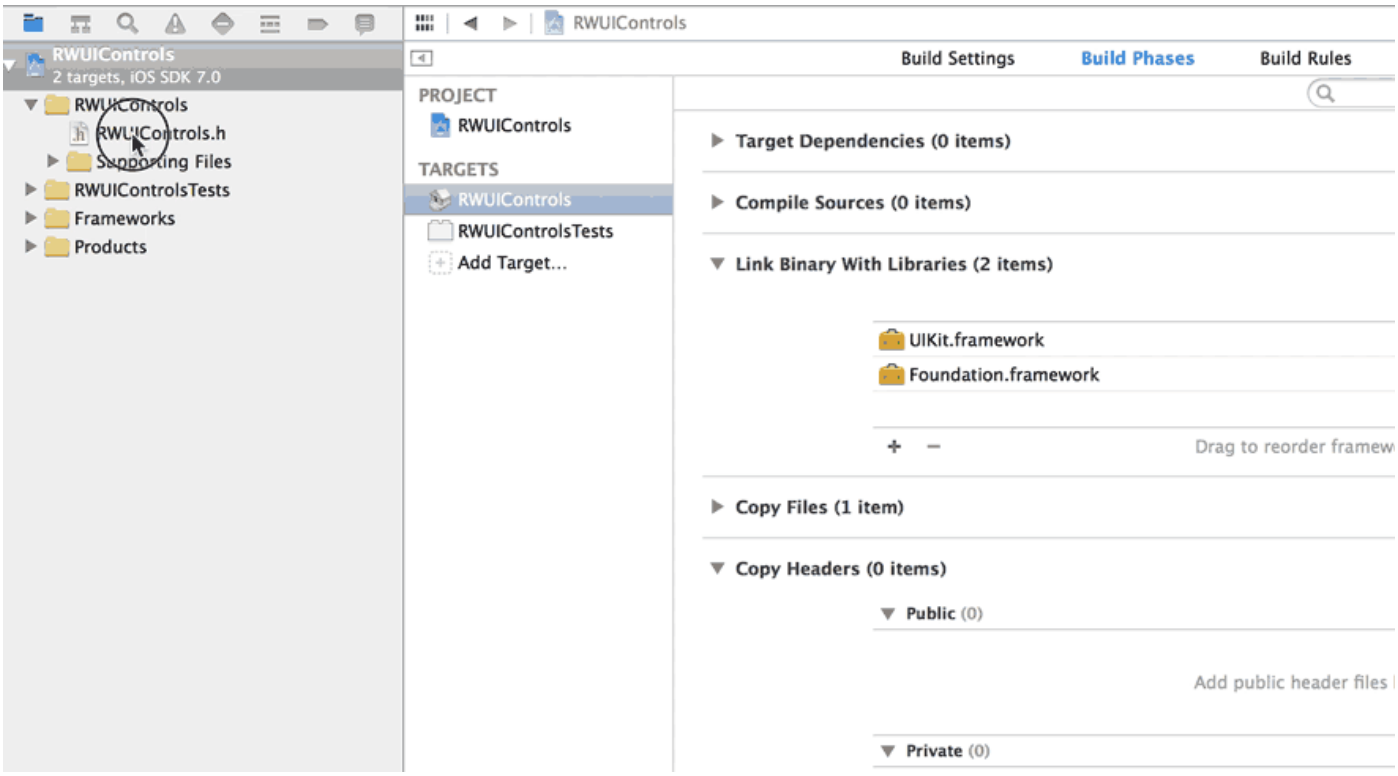
Next, you need to add a new phase in the build, which will collect the public header files and put them somewhere accessible to the compiler. Later, you'll copy these into the framework.

While still looking at the ***Build Phases*** screen in Xcode, choose ***Editor|Add Build Phase|Add Copy Headers Build Phase***.

Note: If you find the option is grayed out, try clicking in the white area directly below the existing build phases to shift the focus, and then try again.



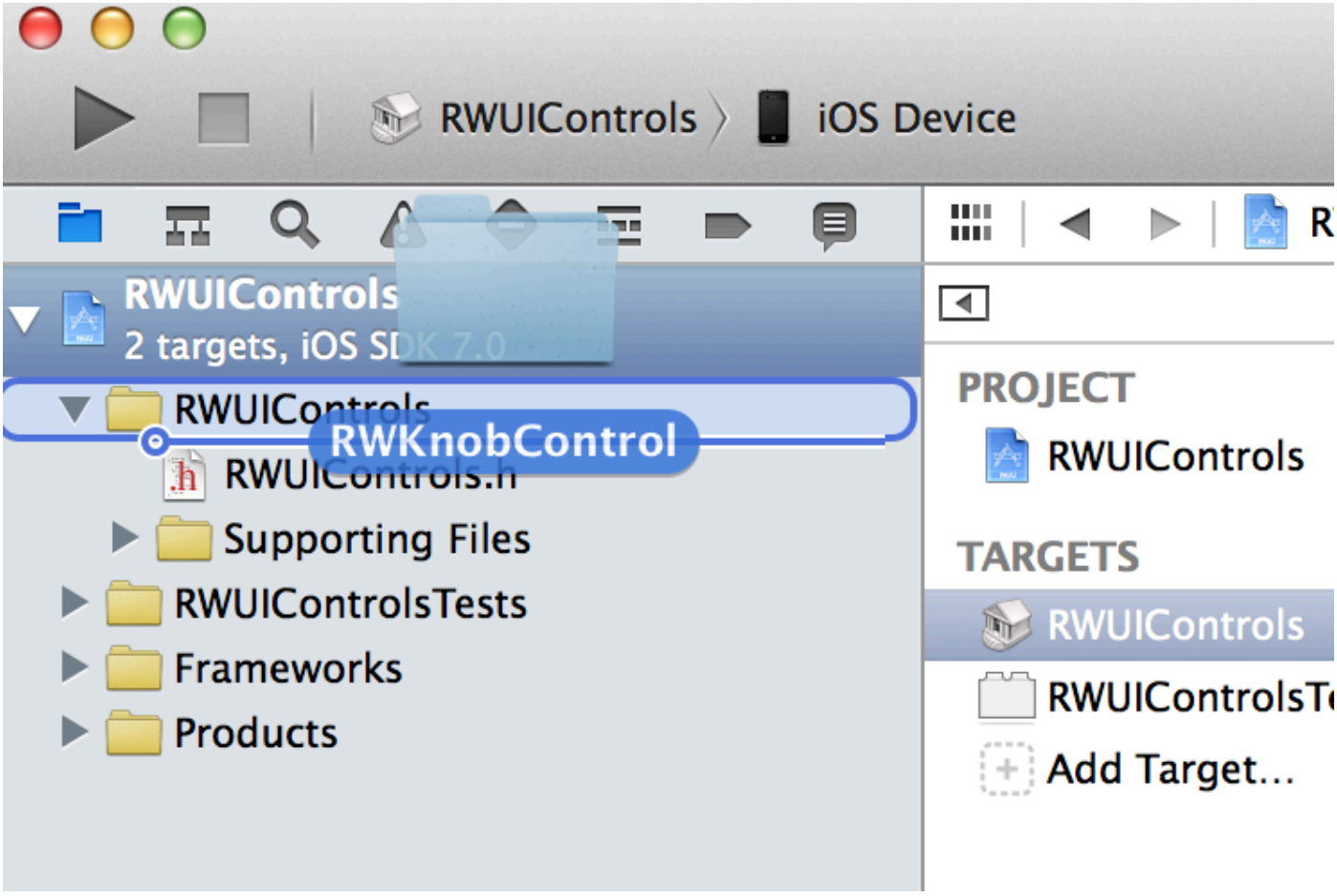
Drag ***RWUIControls.h*** from the navigator to the ***Public*** part of the panel to place it in the public section under ***Copy Headers***. This ensures the header file is available to anybody who uses your library.



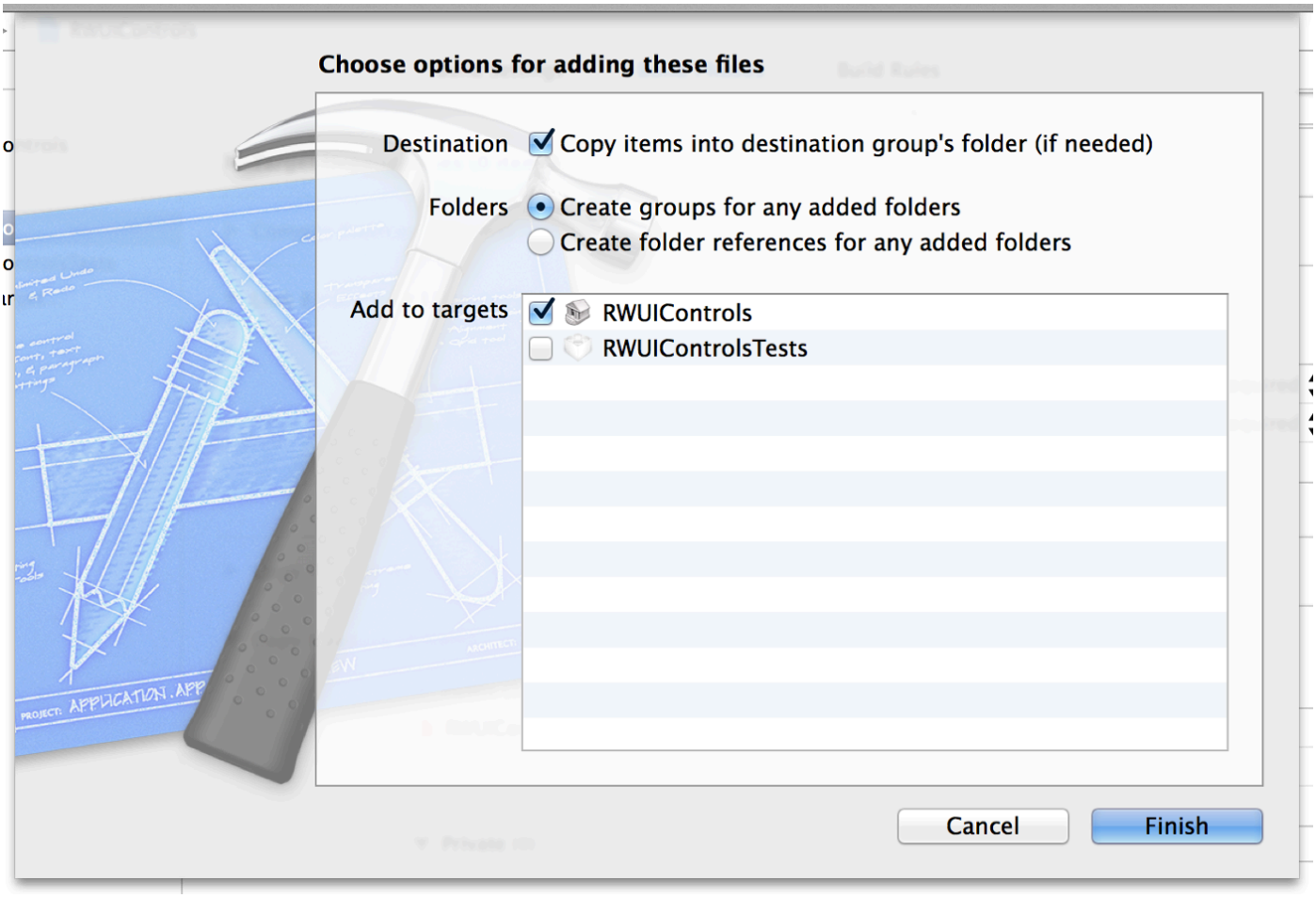
Note: It might seem obvious, but it's important to note that all header files included in any of your public headers must also be made public. Otherwise, developers will get compiler errors while attempting to use the library. It's no fun for anybody when Xcode reads the public headers and then cannot read the headers you forgot to make public.

Creating a UI Control

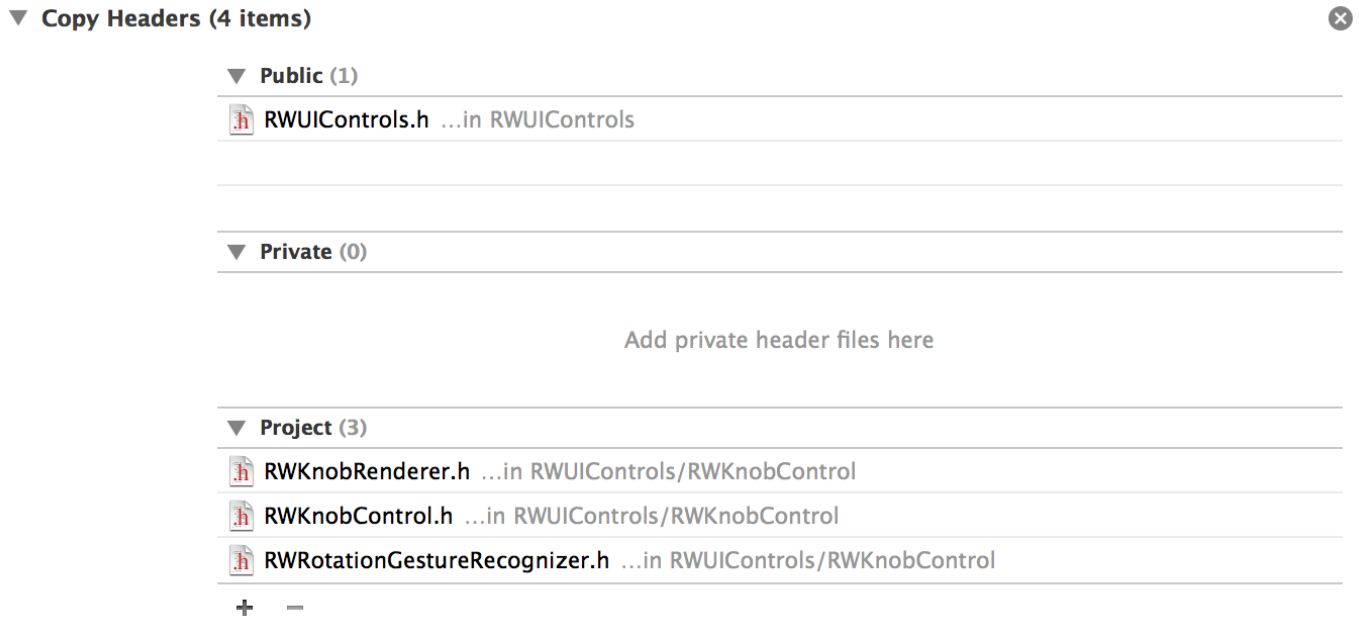
Now that you've set up your project, it's time to add some functionality to the library. Since the point of this tutorial is to describe how to build a framework, not how to build a UI control, you'll borrow the code from the last tutorial. In the zip file you downloaded earlier you'll find the directory ***RWKnobControl***. Drag it from the finder into the ***RWUIControls*** group in Xcode.



Choose to ***Copy items into destination group's folder*** and ensure the new files go to the ***RWUIControls*** static library target by ticking the appropriate box.

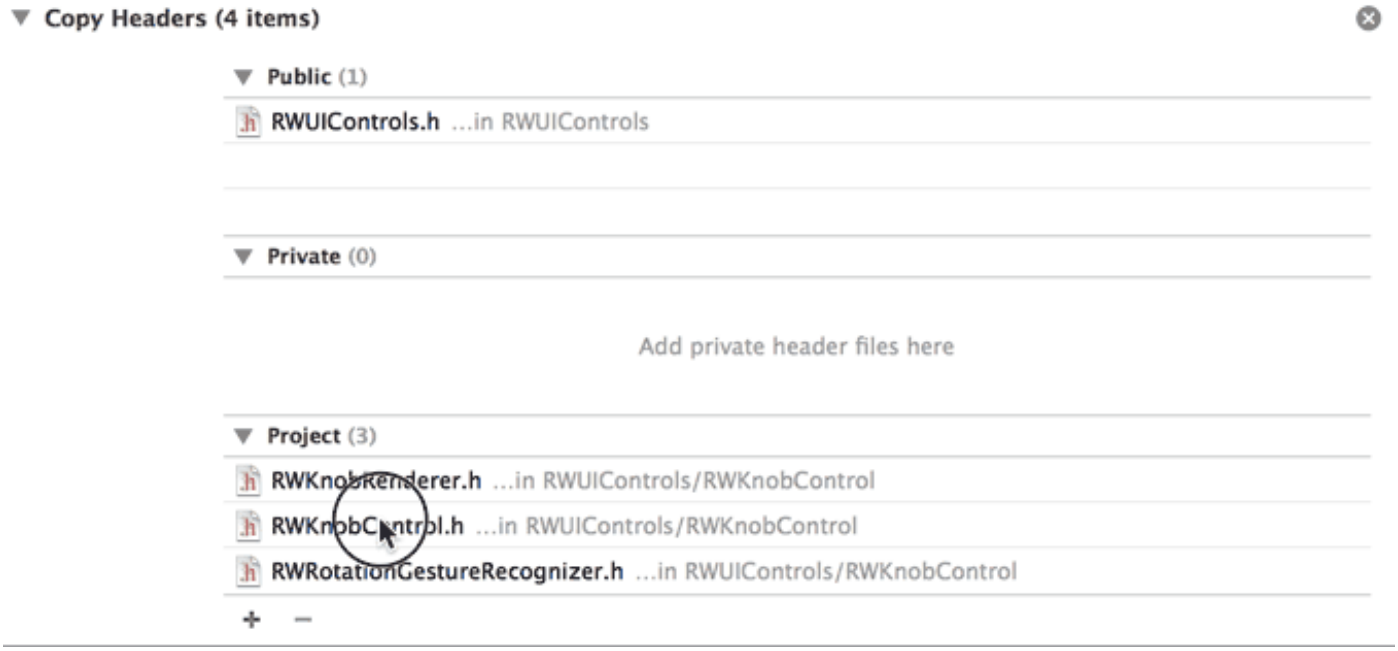


This will add the implementation files to the compilation list and, by default, the header files to the ***Project*** group. This means that they will be private.

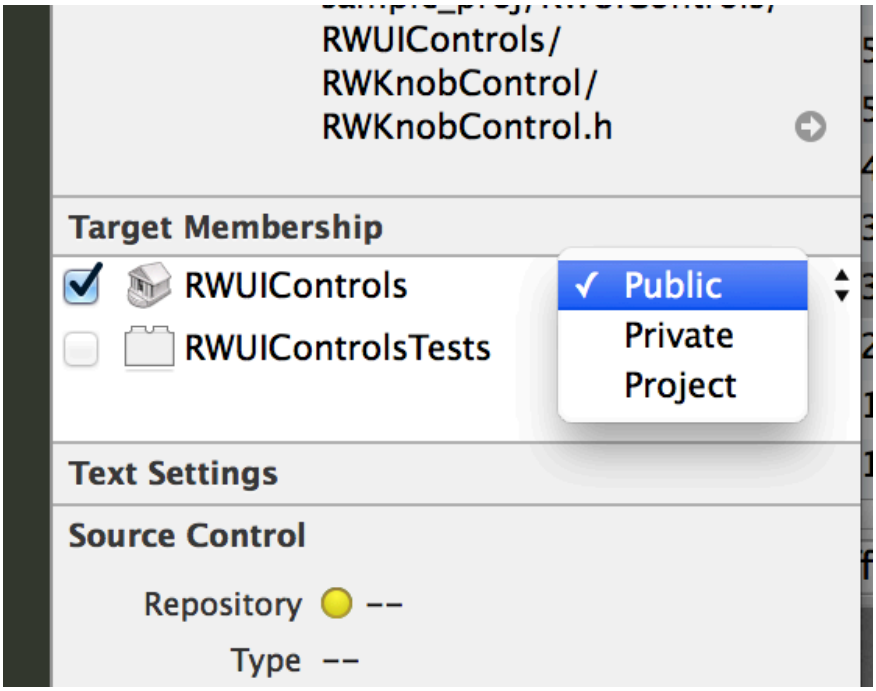


Note: The three section names can be somewhat confusing until you break them down. ***Public*** is just as you'd expect. ***Private*** headers are still exposed, which is a little misleading. And ***Project*** headers are those specific to your project which are, somewhat ironically, private. Therefore, you'll find more often than not that you'll want your headers in either the ***Public*** or ***Project*** groups.

Now you need to share the main control header, ***RWKnobControl.h***, and there are several ways you can do this. The first is to drag the file from the ***Project*** group to the ***Public*** group in the ***Copy Headers*** panel.



Alternatively, you might find it easier to change the membership in the ***Target Membership*** panel when editing the file. This option is a bit more convenient as you can continue to add and develop the library.



Note: As you continue to add new classes to your library, remember to keep their membership up-to-date. Make as few headers public as possible, and ensure the remainder are in the ***Project*** group.

The other thing to do with your control's header file is add it to the library's main header file, ***RWUIControls.h***. With such a main header file, a developer using your library only needs to include one file like you see below, instead of having to sort out which pieces they need:

```
#import <RWUIControls/RWUIControls.h>
```

Therefore, add the following to ***RWUIControls.h***:

```
// Knob Control
#import <RWUIControls/RWKnobControl.h>
```

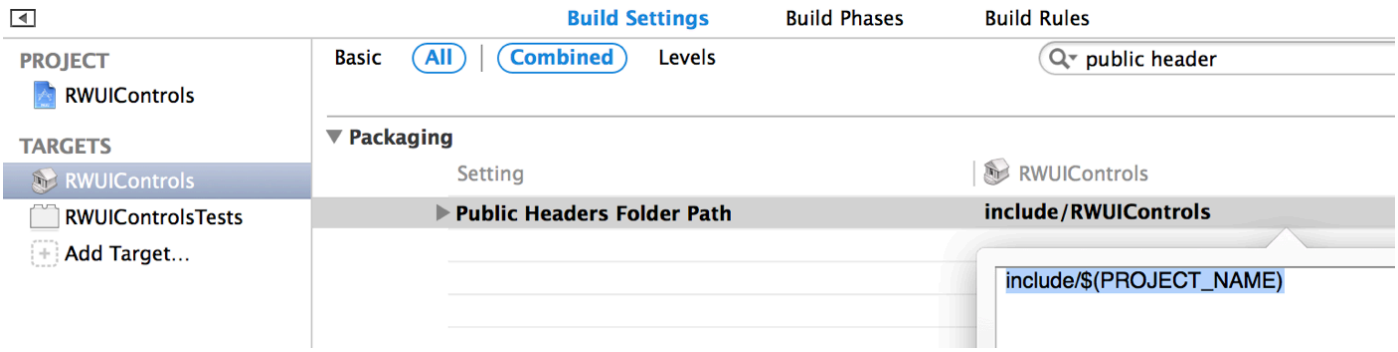
Configuring Build Settings

You are now very close to building this project and creating a static library. However, there are a few settings to configure to make the library as user-friendly as possible.

First, you need to provide a directory name for where you'll copy public headers. This ensures you can locate the relevant headers when you use the static library.

Click on the project in the Project Navigator, and then select the ***RWUIControls*** static library target. Select the ***Build Settings*** tab, then search for *public header*. Double click on the ***Public Headers Folder Path*** setting and enter the following in the popup:

```
include/$(PROJECT_NAME)
```



You'll see this directory later.

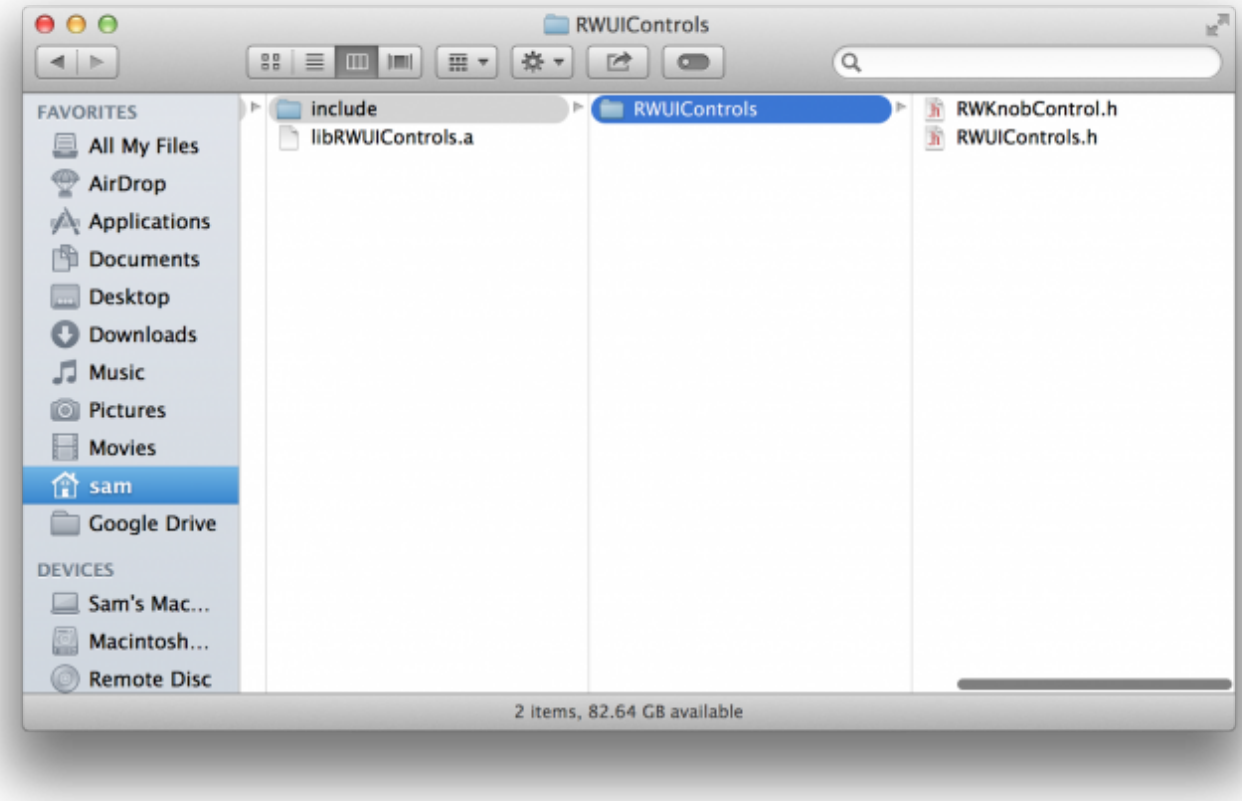
Now you need to change some other settings, specifically those that remain in the binary library. The compiler gives you the option of removing dead code; code which is never accessed. And you can also remove debug symbols i.e. function names and other debugging related details.

Since you're creating a framework for others to use, it's best to disable both and let the user choose what's best for their project. To do this, using the same search field as before, update the following settings:

- ***Dead Code Stripping*** – Set this to `NO`
- ***Strip Debug Symbols During Copy*** – Set this to `NO` for all configurations
- ***Strip Style*** – Set this to `Non-Global Symbols`

Build and run. There's not a lot to see yet, but it's still good to confirm the project builds successfully and without warnings or errors.

To build, select the target as ***iOS Device*** and press ***cmd+B*** to perform the build. Once completed, the ***libRWUIControls.a*** product in the ***Products*** group of the Project Navigator will turn from red to black, signaling that it now exists. Right click on ***libRWUIControls.a*** and select ***Show in Finder***.



In this directory you'll see the static library itself, ***libRWUIControls.a***, and the directory you specified for the public headers, ***include/RWUIControls***.

Notice the headers you made public can be found in this folder, just as you might expect.

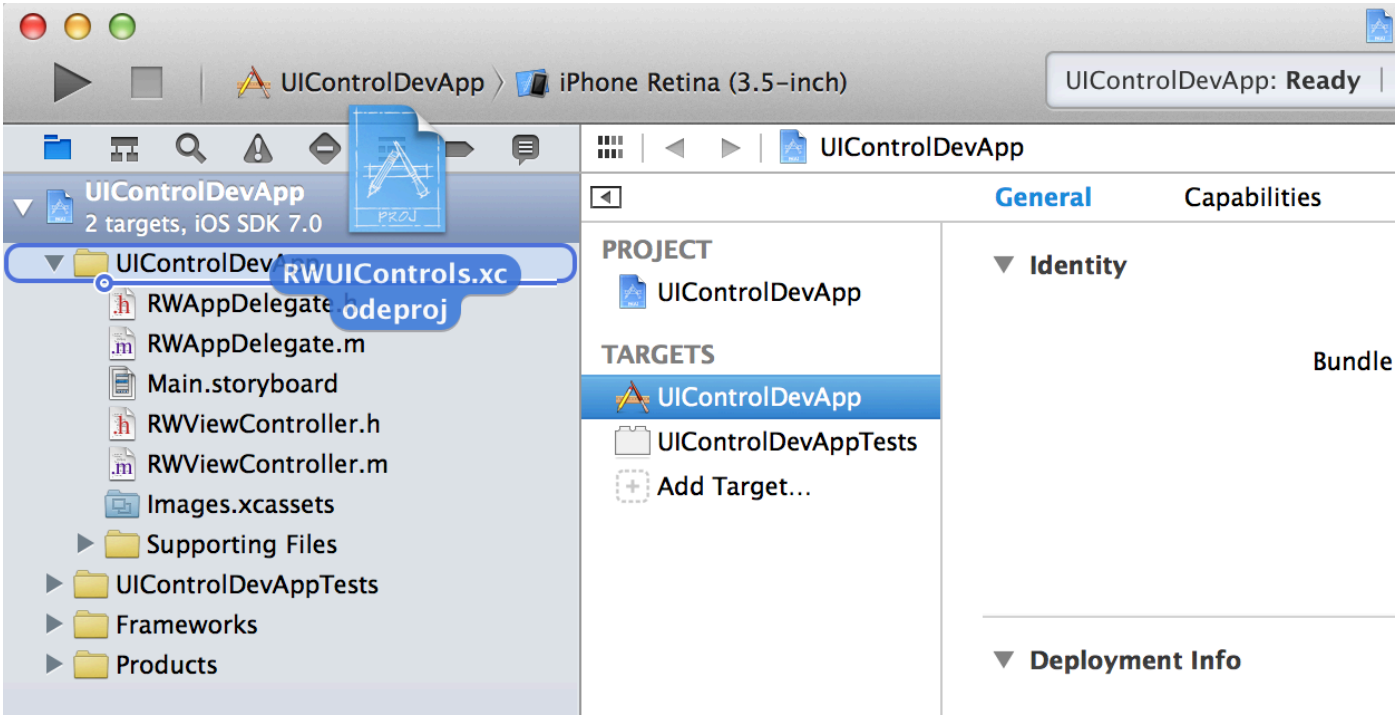
Creating a Dependent Development Project

Developing a UI controls library for iOS is extremely difficult when you can't actually see what you're doing, and that seems to be the case now.

Nobody wants you to work blindly, so in this section you're going to create a new Xcode project that will have a dependency on the library you just created. This will allow you to develop the framework using an example app. Naturally, the code for this app will be kept completely separate from the library itself, as this makes for a much cleaner structure.

Close the static library project by choosing ***File/Close Project***. Then create a new project using ***File/New/Project***. Select ***iOS/Application/Single View Application***, and call the new project ***UIControlDevApp***. Set the class prefixes to ***RW*** and specify that it should be ***iPhone*** only. Finally save the project in the same directory you used for ***RWUIControls***.

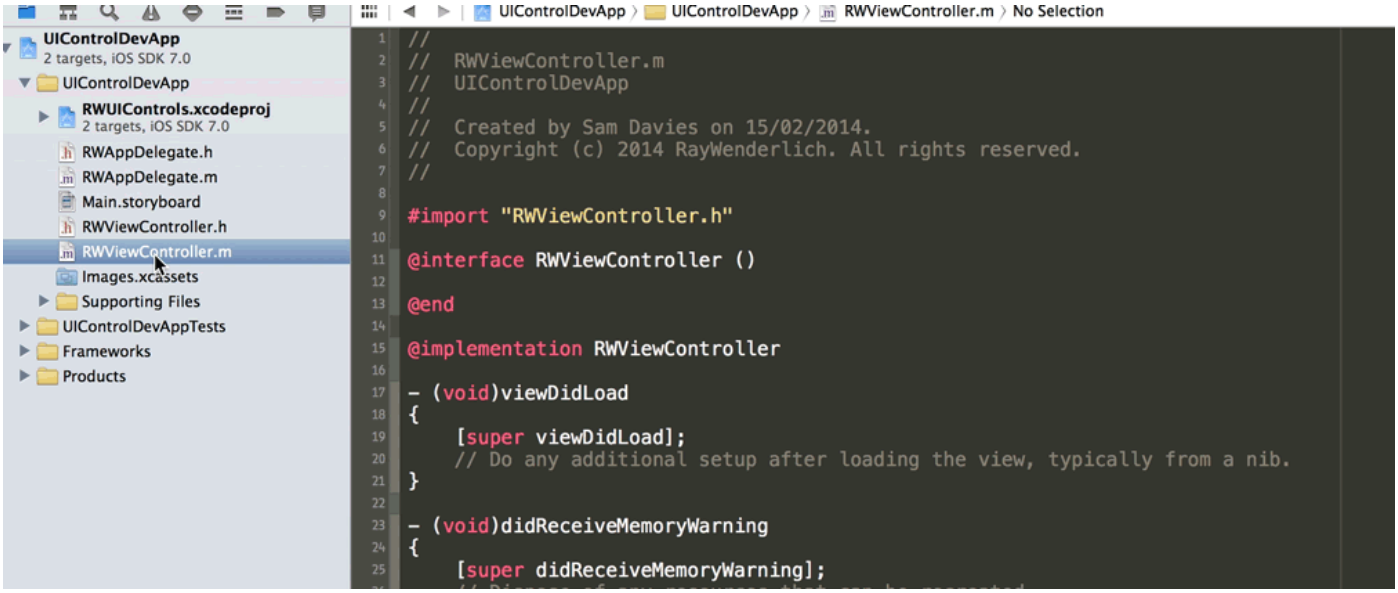
To add the ***RWUIControls*** library as a dependency, drag ***RWUIControls.xcodeproj*** from the finder into the ***UIControlDevApp*** group in Xcode.



You can now navigate around the library project, from inside the app's project. This is perfect because it means that you can edit code inside the library and run the example app to test the changes.

Note: You can't have the same project open in two different Xcode windows. If you find that you're unable to navigate around the library project, check that you don't have it open in another Xcode window.

Rather than recreate the app from the last tutorial, you can simply copy the code. First, select ***Main.storyboard***, ***RWViewController.h*** and ***RWViewController.m*** and delete them by right clicking and selecting ***Delete***, choosing to move them to the trash. Then copy the ***DevApp*** folder from the zip file you downloaded earlier right into the ***UIControlDevApp*** group in Xcode.

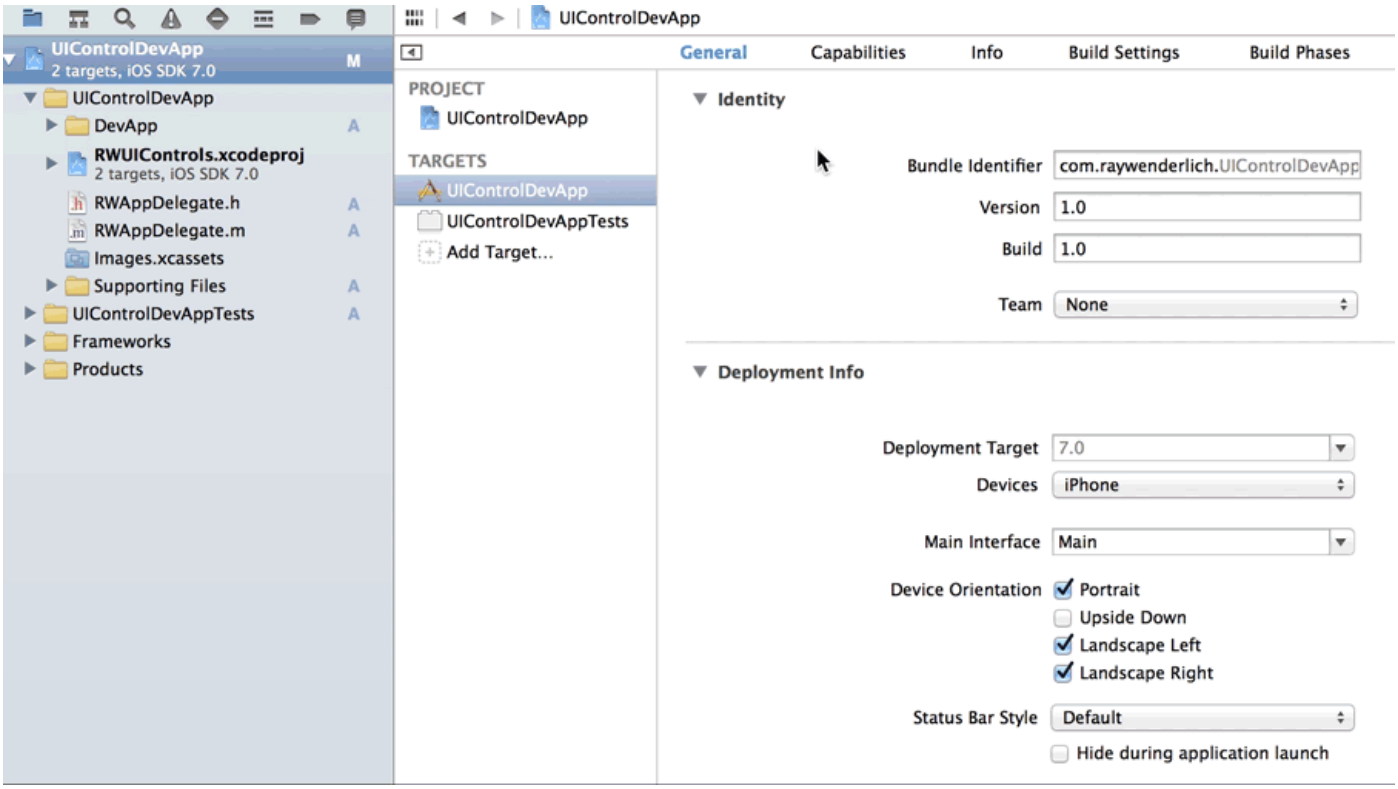


Now you're going to add the static library as a build dependency of the example app:

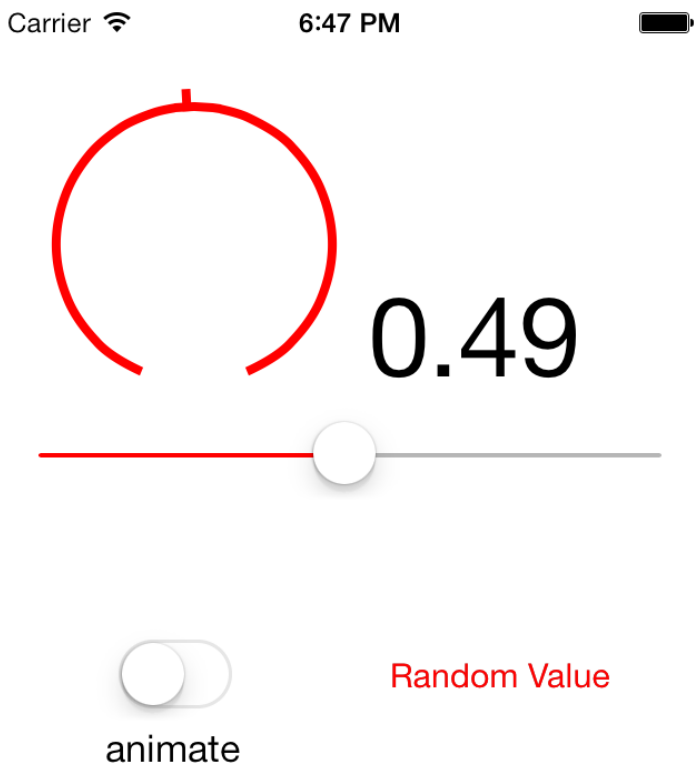
- Select the ***UIControlDevApp*** project in the Project Navigator.
- Navigate to the ***Build Phases*** tab of the ***UIControlDevApp*** target.
- Open the ***Target Dependencies*** panel and click the ***+*** to show the picker.
- Find the ***RWUIControls*** static library, select and click ***Add***. This action means that when building the dev app, Xcode will check to see whether the static library needs rebuilding or not.

In order to link against the static library itself, expand the ***Link Binary With Libraries*** panel and again click the ***+***. Select ***libRWUIControls.a*** from the ***Workspace*** group and click ***Add***.

This action makes it so that Xcode will link it against the static library, just as it links against system frameworks like ***UIKit***.



Build and run to see it in action. If you followed the previous tutorial on building a knob control, you'll recognize the simple app before your eyes.



The beauty of using nested projects like this is that you can continue to work on the library itself, without ever leaving the example app project, even as you maintain the code in different places. Each time you build the project, you're also checking that you have the public/project header membership set correctly. The example app won't build if it's missing any required headers.

Building a Framework

By now, you're probably impatiently tapping your toes and wondering where the framework comes into play. Understandable, because so far you've done a lot of work and yet there is no framework in sight.

Well, things are about to change, and quickly too. The reason that you've not created a framework is because it's pretty much a static library and a collection of headers – exactly what you've built so far.

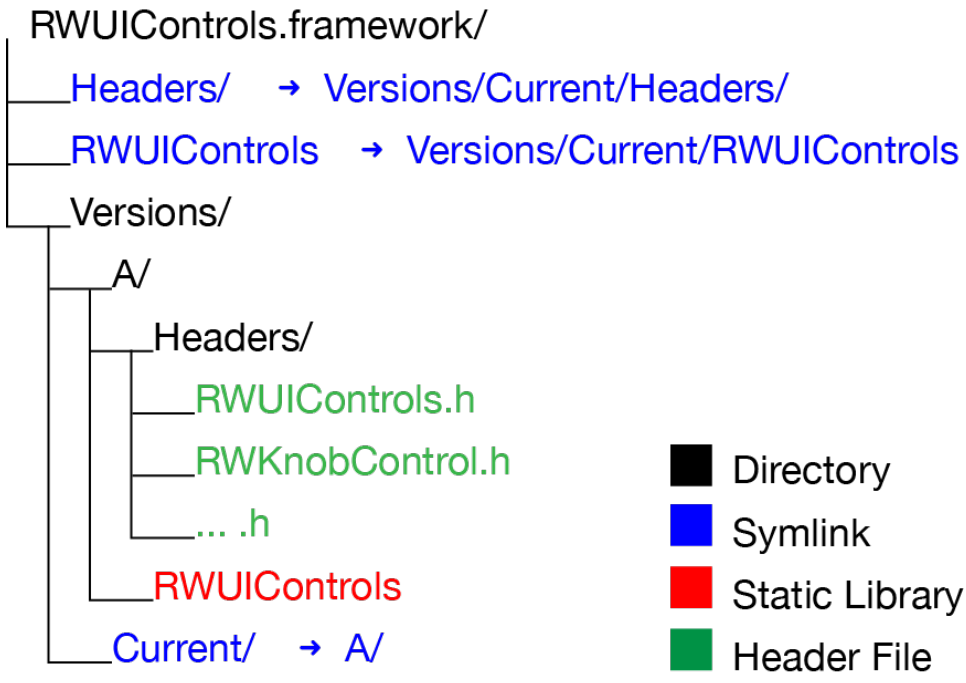
There are a couple of things that make a framework distinct:

1. The directory structure. Frameworks have a special directory structure that is recognized by Xcode. You'll create a build task, which will create this structure for you.
2. The *Slices*. Currently, when you build the library, it's only for the currently required architecture, i.e. i386, arm7, etc. In order for a framework to be useful, it needs to include builds for all the architectures on which it needs to run. You'll create a new product which will build the required architectures and place them in the framework.

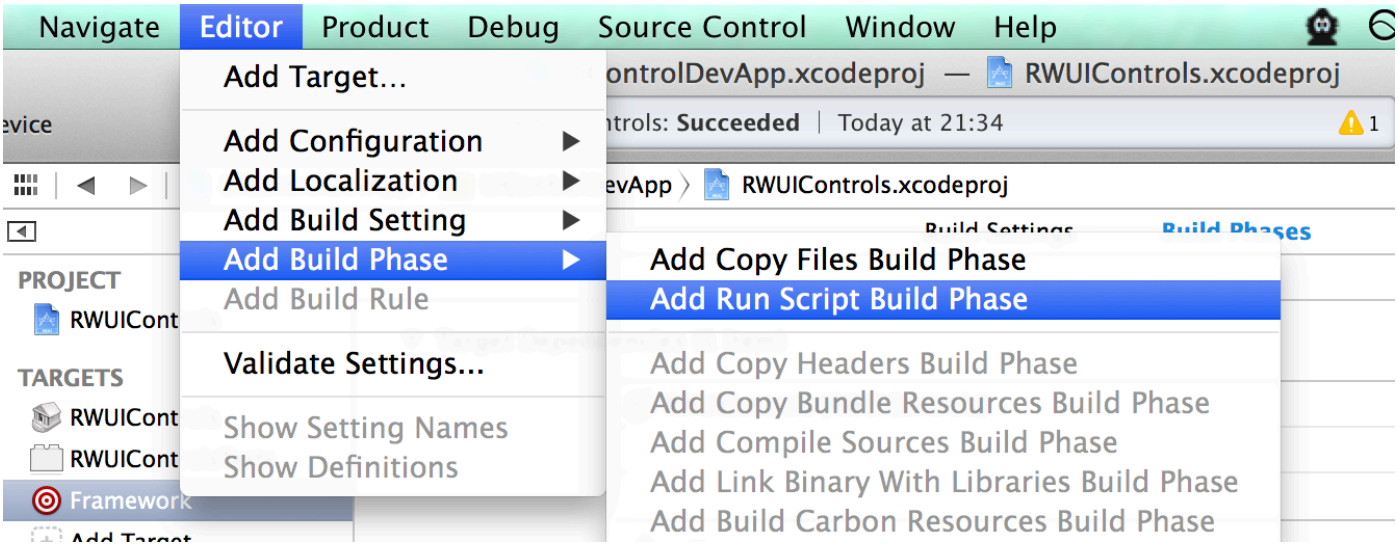
There is a quite a lot of scripting magic in this section, but we'll work through it slowly; it's not nearly as complicated as it appears.

Framework Structure

As mentioned previously, a framework has a special directory structure which looks like this:

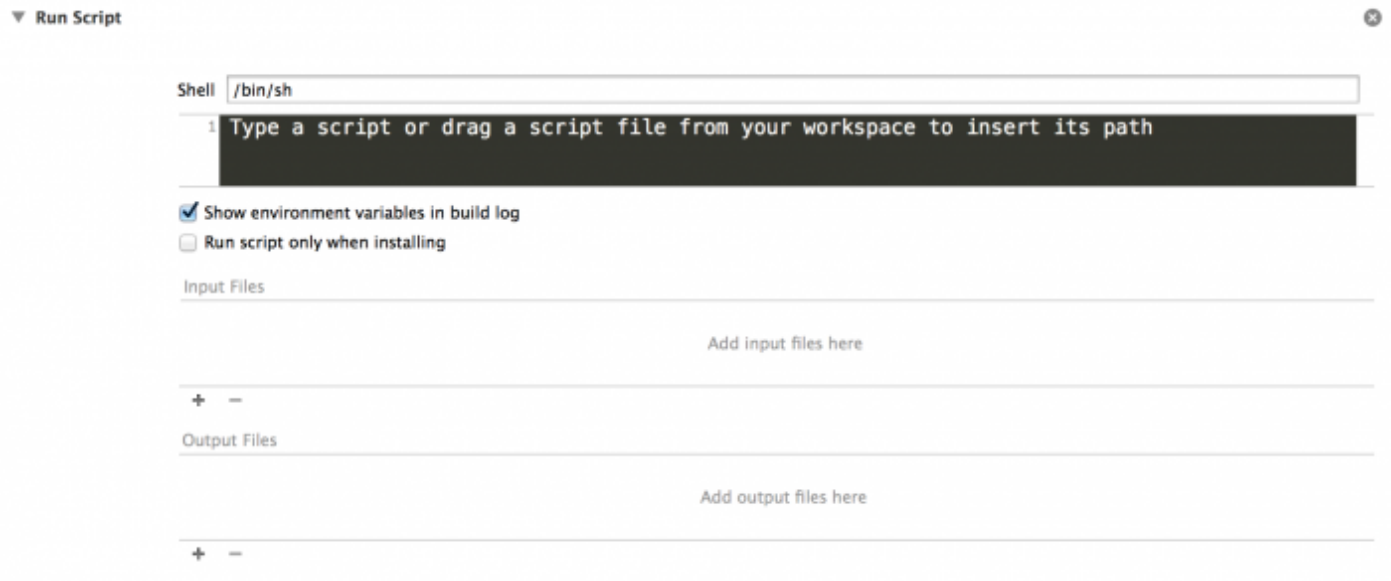


Now you'll add a script to create this during the static library build process. Select the ***RWUIControls*** project in the Project Navigator, and select the ***RWUIControls*** static library target. Choose the ***Build Phases*** tab and add a new script by selecting ***Editor/Add Build Phase/Add Run Script Build Phase***.

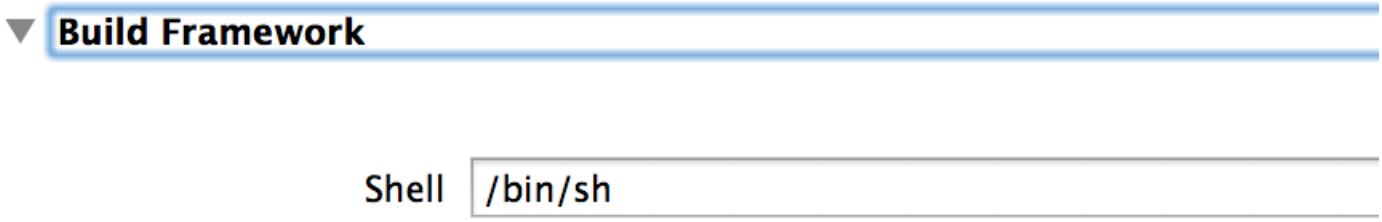


This creates a new panel in the build phases section, which allows you to run an arbitrary Bash script at some point during the build. Drag the panel around in the list if you want to change the point at which the script runs in

the build process. For the framework project, run the script last, so you can leave it where it's placed by default.



Rename the script by double clicking on the panel title ***Run Script*** and replace it with ***Build Framework***.



Paste the following Bash script into the script field:

```
set -e

export FRAMEWORK_LOCN="${BUILT_PRODUCTS_DIR}/${PRODUCT_NAME}.framework"

# Create the path to the real Headers die
mkdir -p "${FRAMEWORK_LOCN}/Versions/A/Headers"

# Create the required symlinks
/bin/ln -sfh A "${FRAMEWORK_LOCN}/Versions/Current"
/bin/ln -sfh Versions/Current/Headers "${FRAMEWORK_LOCN}/Headers"
```

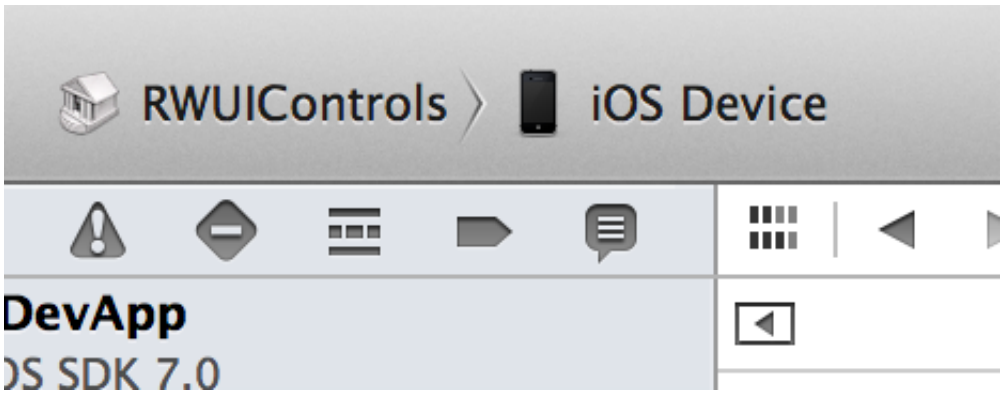
```
/bin/ln -sfh "Versions/Current/${PRODUCT_NAME}" \
             "${FRAMEWORK_LOCN}/${PRODUCT_NAME}"

# Copy the public headers into the framework
/bin/cp -a "${TARGET_BUILD_DIR}/${PUBLIC_HEADERS_FOLDER_PATH}/" \
           "${FRAMEWORK_LOCN}/Versions/A/Headers"
```

This script first creates the ***RWUIControls.framework/Versions/A/Headers*** directory before then creating the three symbolic links required for a framework:

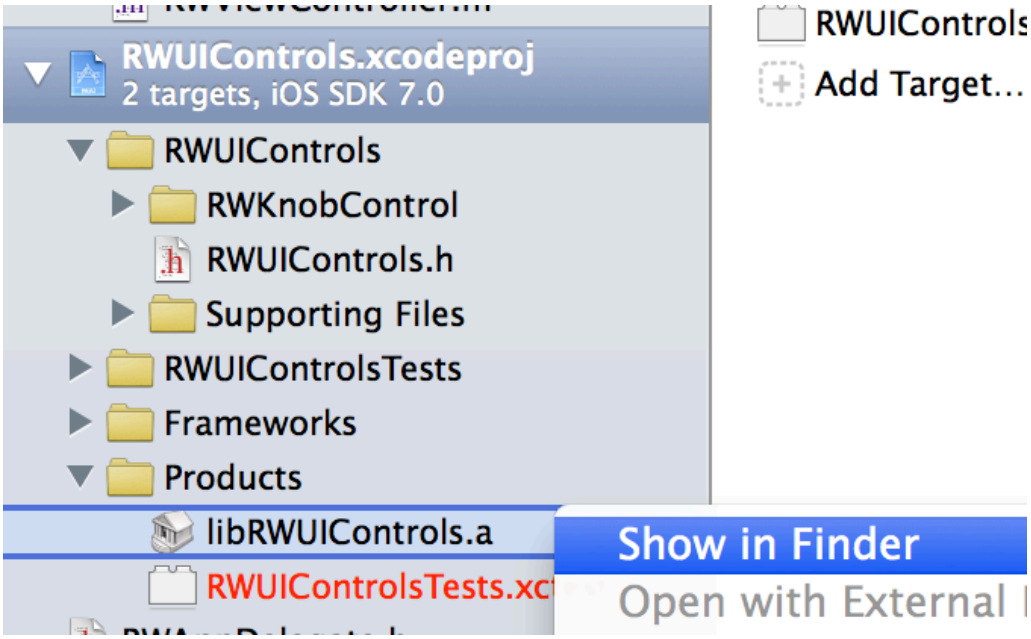
- ***Versions/Current*** => ***A***
- ***Headers*** => ***Versions/Current/Headers***
- ***RWUIControls*** => ***Versions/Current/RWUIControls***

Finally, the public header files copy into the ***Versions/A/Headers*** directory from the public headers path you specified before. The ***-a*** argument ensures the modified times don't change as part of the copy, thereby preventing unnecessary rebuilds.

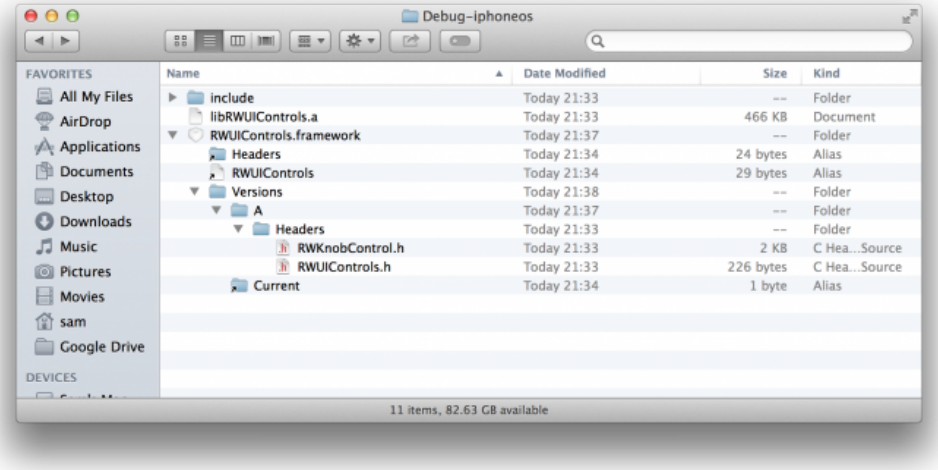
Now, select the ***RWUIControls*** static library scheme and the ***iOS Device*** build target, then build using ***cmd+B***.



Right click on the ***libRWUIControls.a*** static library in the ***Products*** group of the ***RWUIControls*** project, and once again select ***Show In Finder***.



Within this build directory you can access the ***RWUIControls.framework***, and confirm the correct directory structure is present and populated:



This is a leap forward on the path of completing your framework, but you'll notice that there isn't a static lib in there yet. That's what you're going to sort out next.

Multi-Architecture Build

iOS apps need to run on many different architectures:

- ***arm7***: Used in the oldest iOS 7-supporting devices
- ***arm7s***: As used in iPhone 5 and 5C
- ***arm64***: For the 64-bit ARM processor in iPhone 5S
- ***i386***: For the 32-bit simulator
- ***x86_64***: Used in 64-bit simulator

Every architecture requires a different binary, and when you build an app Xcode will build the correct architecture for whatever you're currently working with. For instance, if you've asked it to run in the simulator, then it'll only build the i386 version (*or x86_64 for 64-bit*).

This means that builds are as fast as they can be. When you archive an app or build in release mode, then Xcode will build for all three ARM architectures, thus allowing the app to run on most devices. What about the other builds though?

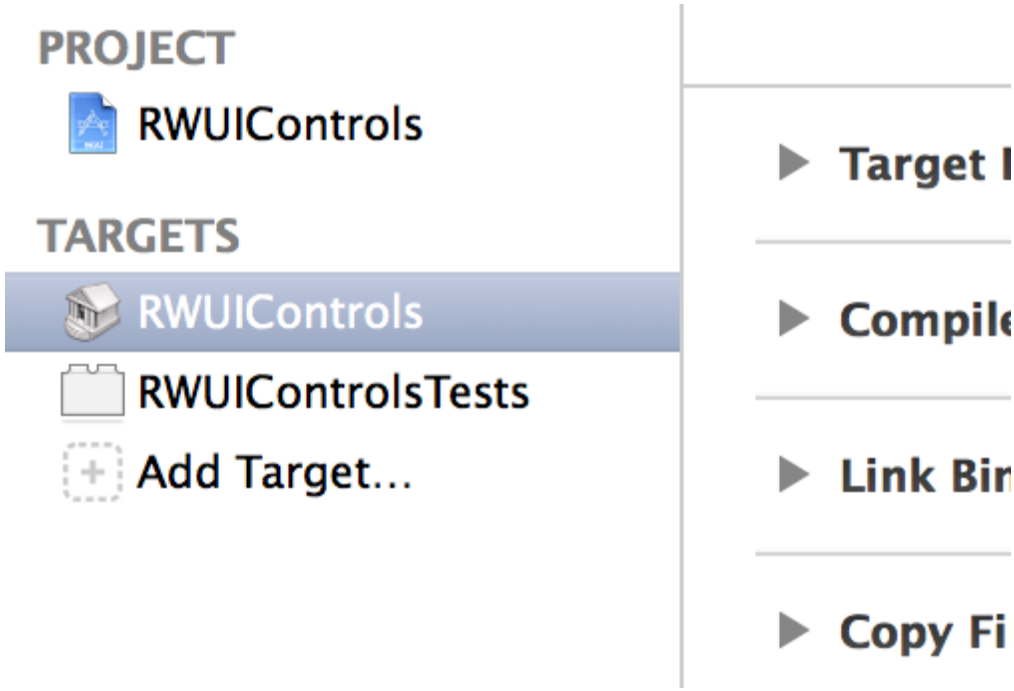
Naturally, when you build your framework, you'll want developers to be able to use it for all possible architectures, right? Of course you do since that'll mean you can earn the respect and admiration of your peers.

Therefore you need to make Xcode build for all *five* architectures. This process creates a so-called *fat* binary, which contains a slice for each of the architectures. Ah-ha!

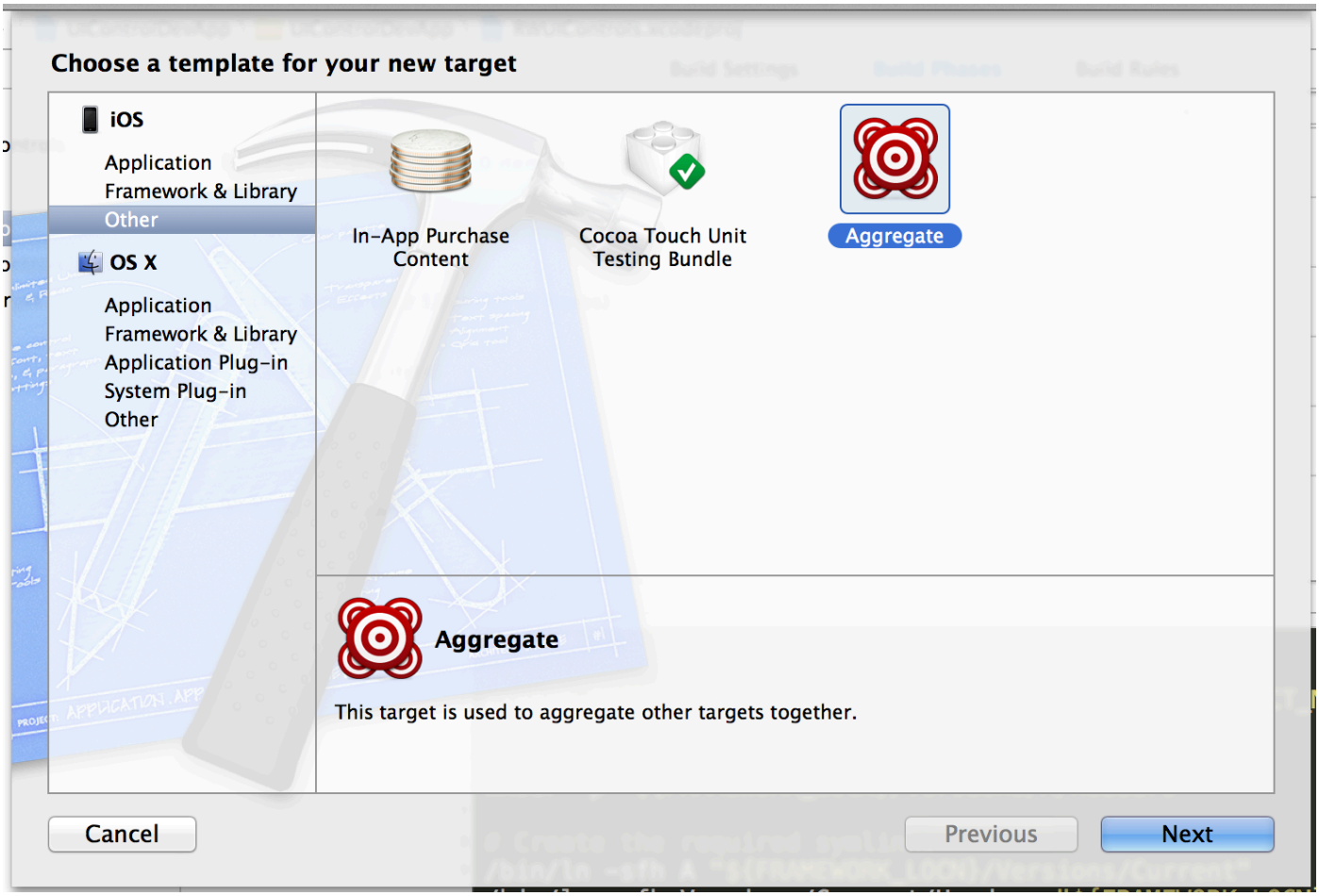
Note: This actually highlights another reason to create an example app which has a dependency on the static library: the library only builds for the architecture required by the example app, and will only rebuild if something changes. Why should this excite you? It means the development cycle is as quick as possible.

The framework will be created using a new target in the ***RWUIControls*** project. To create it, select the ***RWUIControls*** project in the Project

Navigator and then click the ***Add Target*** button shown below the existing targets.

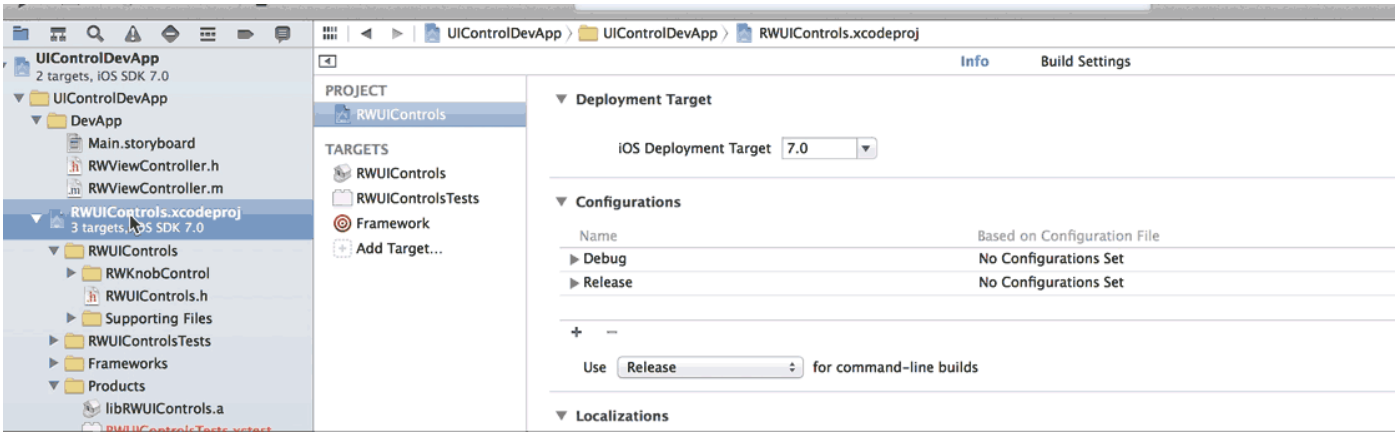


Navigate to ***iOS/Other/Aggregate***, click ***Next*** and name the target ***Framework***.

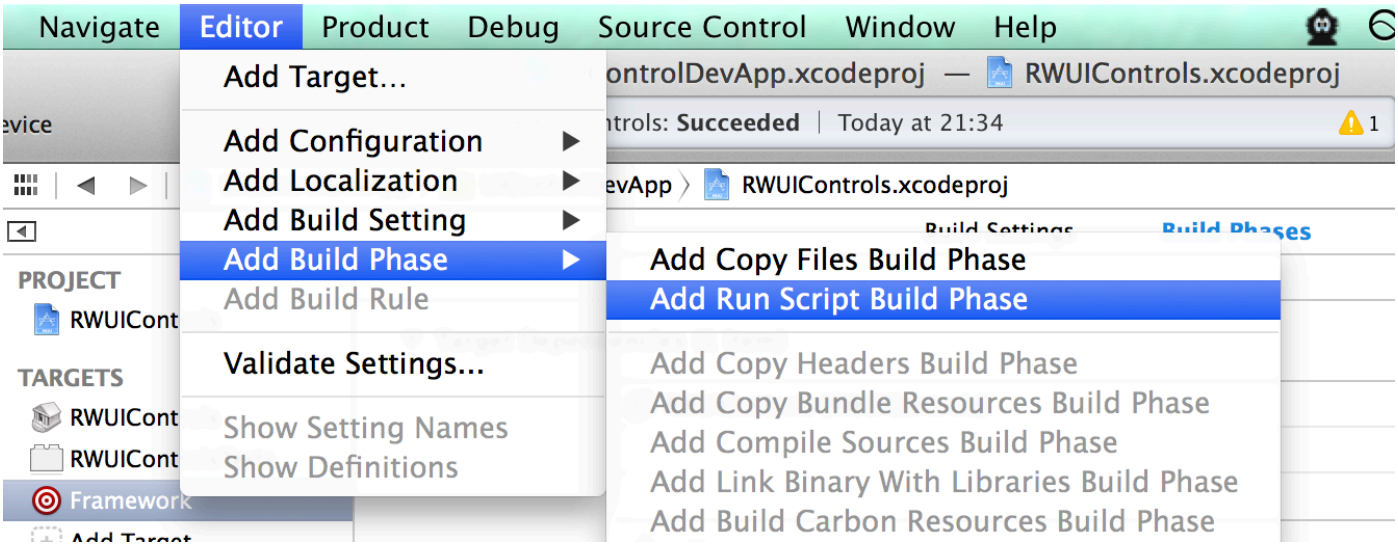


Note: Why use an *Aggregate* target to build a Framework? Why so indirect? Because Frameworks enjoy better support on OS X, as reflected by the fact that Xcode offers a straightforward ***Cocoa Framework*** build target for OS X apps. To work around this, you'll use the aggregate build target as a hook for bash scripts that build the magic framework directory structure. Are you starting to see the method to the madness here?

To ensure the static library builds whenever this new framework target is created, you need to add a dependency on the static library target. Select the ***Framework*** target in the library project and add a dependency in the ***Build Phases*** tab. Expand the ***Target Dependencies*** panel, click the ***+*** and choose the ***RWUIControls*** static library.



The main build part of this target is the multi-platform building, which you'll perform using a script. As you did before, create a new ***Run Script*** build phase by selecting the ***Build Phases*** tab of the ***Framework*** target, and clicking ***Editor/Add Build Phase/Add Run Script Build Phase***.



Change the name of the script by double clicking on ***Run Script***. This time name it ***MultiPlatform Build***.

Paste the following Bash script into the script text box:

```
set -e

# If we're already inside this script then die
if [ -n "$RW_MULTIPLATFORM_BUILD_IN_PROGRESS" ]; then
```

```
  exit 0
fi
export RW_MULTIPLATFORM_BUILD_IN_PROGRESS=1

RW_FRAMEWORK_NAME=${PROJECT_NAME}
RW_INPUT_STATIC_LIB="lib${PROJECT_NAME}.a"
RW_FRAMEWORK_LOCATION="${BUILT_PRODUCTS_DIR}/${RW_FRAMEWORK_NAME}.framework"
```

- `set -e` ensures that if any part of the script should fail then the entire script will fail. This helps you avoid a partially-built framework.
- Next, the `RW_MULTIPLATFORM_BUILD_IN_PROGRESS` variable determines whether the script was called recursively. If it has, then quit.
- Then set up some variables. The framework name will be the same as the project i.e. ***RWUIControls***, and the static lib is ***libRWUIControls.a***.

The next part of the script sets up some functions that the project will use later on. Add the following to the very bottom of the script:

```
function build_static_library {
    # Will rebuild the static library as specified
    #     build_static_library sdk
    xcrun xcodebuild -project "${PROJECT_FILE_PATH}" \
                     -target "${TARGET_NAME}" \
                     -configuration "${CONFIGURATION}" \
                     -sdk "${1}" \
                     ONLY_ACTIVE_ARCH=NO \
                     BUILD_DIR="${BUILD_DIR}" \
                     OBJROOT="${OBJROOT}" \
                     BUILD_ROOT="${BUILD_ROOT}" \
                     SYMROOT="${SYMROOT}" $ACTION
}

function make_fat_library {
    # Will smash 2 static libs together
    #     make_fat_library in1 in2 out
    xcrun lipo -create "${1}" "${2}" -output "${3}"
}
```

- `build_static_library` takes an ***SDK*** as an argument, for example ***iphoneos7.0***, and will build the static lib. Most of the arguments pass directly from the current build job, the difference being that `ONLY_ACTIVE_ARCH` is set to ensure all architectures build for the current SDK.
- `make_fat_library` uses `lipo` to join two static libraries into one. Its arguments are two input libraries followed by the desired output location. Read more about lipo here

The next section of the script determines some more variables which you'll need in order to use the two methods. You need to know what the other SDK is, for example iphoneos7.0 should go to iphonesimulator7.0 and vice versa, and to locate the build directory for that SDK.

Add the following to the very end of the script:

```
# 1 - Extract the platform (iphoneos/iphonesimulator) from the SDK name
if [[ "$SDK_NAME" =~ ([A-Za-z]+) ]]; then
  RW_SDK_PLATFORM=${BASH_REMATCH[1]}
else
  echo "Could not find platform name from SDK_NAME: $SDK_NAME"
  exit 1
fi

# 2 - Extract the version from the SDK
if [[ "$SDK_NAME" =~ ([0-9]+.*$) ]]; then
  RW_SDK_VERSION=${BASH_REMATCH[1]}
else
  echo "Could not find sdk version from SDK_NAME: $SDK_NAME"
  exit 1
fi

# 3 - Determine the other platform
if [ "$RW_SDK_PLATFORM" == "iphoneos" ]; then
  RW_OTHER_PLATFORM=iphonesimulator
else
```

```
  RW_OTHER_PLATFORM=iphoneos
fi

# 4 - Find the build directory
if [[ "$BUILT_PRODUCTS_DIR" =~ (.*)$RW_SDK_PLATFORM$ ]]; then
  RW_OTHER_BUILT_PRODUCTS_DIR="${BASH_REMATCH[1]}${RW_OTHER_PLATFORM}"
else
  echo "Could not find other platform build directory."
  exit 1
fi
```

All four of these statements are very similar, they use string comparison and regular expressions to determine `RW_OTHER_PLATFORM` and `RW_OTHER_BUILT_PRODUCTS_DIR`.

The four `if` statements in more detail:

1. `SDK_NAME` will be of the form `iphoneos7.0` or `iphonesimulator6.1`. This regex extracts the non-numeric characters at the beginning of this string. Hence, it results in `iphoneos` or `iphonesimulator`.
2. This regex pulls the numeric version number from the end of the `SDK_NAME` variable, `7.0` or `6.1` etc.
3. Here a simple string comparison switches `iphonesimulator` for `iphoneos` and vice versa.
4. Take the platform name from the end of the build products directory path and replace it with the other platform. This ensures the build directory for the other platform can be found. This will be critical when joining the two static libraries.

Now you can trigger the build for the other platform, and then join the resulting static libraries.

Add the following to the end of the script:

```
# Build the other platform.
build_static_library "${RW_OTHER_PLATFORM}${RW_SDK_VERSION}"

# If we're currently building for iphonesimulator, then need to rebuild
#   to ensure that we get both i386 and x86_64
if [ "$RW_SDK_PLATFORM" == "iphonesimulator" ]; then
    build_static_library "${SDK_NAME}"
fi

# Join the 2 static libs into 1 and push into the .framework
make_fat_library "${BUILT_PRODUCTS_DIR}/${RW_INPUT_STATIC_LIB}" \
                 "${RW_OTHER_BUILT_PRODUCTS_DIR}/${RW_INPUT_STATIC_LIB}" \
                 "${RW_FRAMEWORK_LOCATION}/Versions/A/${RW_FRAMEWORK_NAME}"
```

- First there's a call to build the other platform using the function you defined beforehand
- If you're currently building for the simulator, then by default Xcode will only build the architecture for that system, e.g. ***i386*** or ***x86_64***. In order to build both architectures, this second call to `build_static_library` rebuilds with the `iphonesimulator` SDK, and ensures that both architectures build.
- Finally a call to `make_fat_library` joins the static lib in the current build directory with that in the other build directory to make the multi-architecture fat static library. This is placed inside the framework.

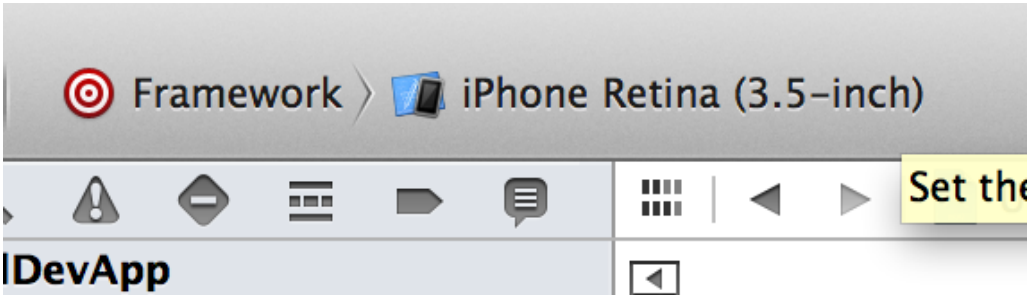
The final commands of the script are simple copy commands. Add the following to the end of the script:

```
# Ensure that the framework is present in both platform's build directories
cp -a "${RW_FRAMEWORK_LOCATION}/Versions/A/${RW_FRAMEWORK_NAME}" \

"${RW_OTHER_BUILT_PRODUCTS_DIR}/${RW_FRAMEWORK_NAME}.framework/Versions/A/${RW


# Copy the framework to the user's desktop
ditto "${RW_FRAMEWORK_LOCATION}"
```

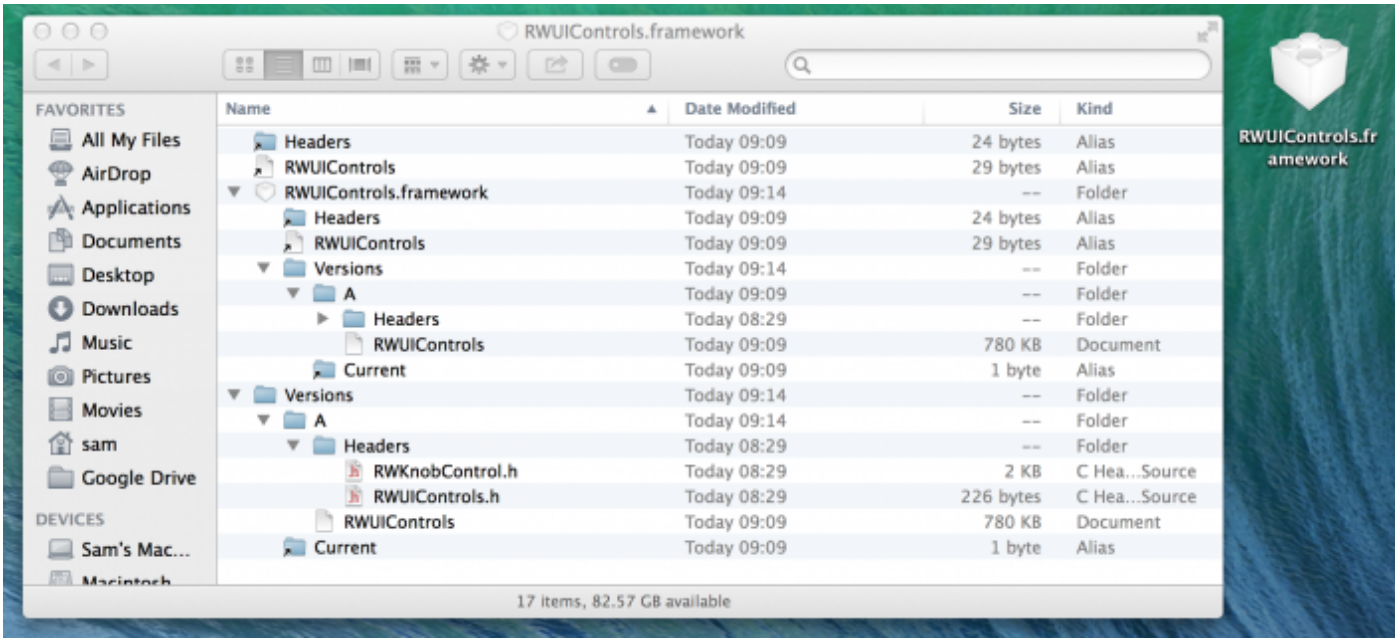
```
"${HOME}/Desktop/${RW_FRAMEWORK_NAME}.framework"
```

- The first command ensures that the framework is present in both platform's build directories.
- The second copies the completed framework to the user's desktop. This is an optional step, but I find that it's a lot easier to place the framework somewhere that is easily accessible.

Select the ***Framework*** aggregate scheme, and press ***cmd+B*** to build the framework.



This will build and place a ***RWUIControls.framework*** on your desktop.

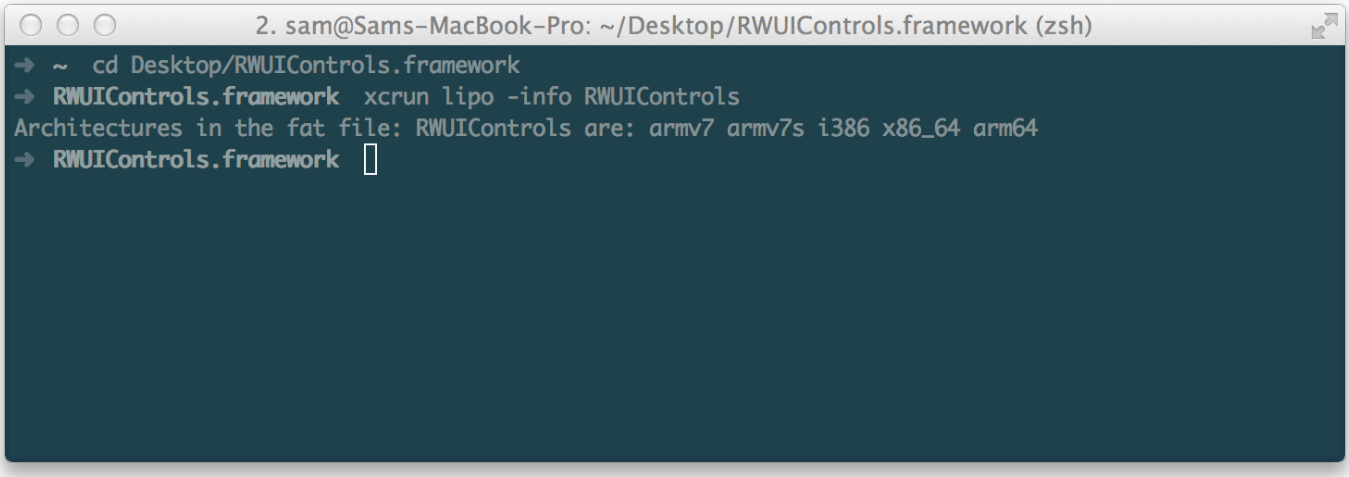


In order to check that the multi-platform build worked, fire up a terminal and

navigate to the framework on the desktop, as follows:

```
$ cd ~/Desktop/RWUIControls.framework
$ RWUIControls.framework  xcrun lipo -info RWUIControls
```

The first command navigates into the framework itself, and the second line uses the `lipo` command to get the required information on the***RWUIControls*** static library. This will list the slices that are present in the library.



You can see here that there are five slices: ***i386***, ***x86_64***, ***arm7***, ***arm7s*** and ***arm64***, which is exactly what you set out to build. Had you run the `lipo -info` command beforehand, you would have seen a subset of these slices.

How to Use a Framework

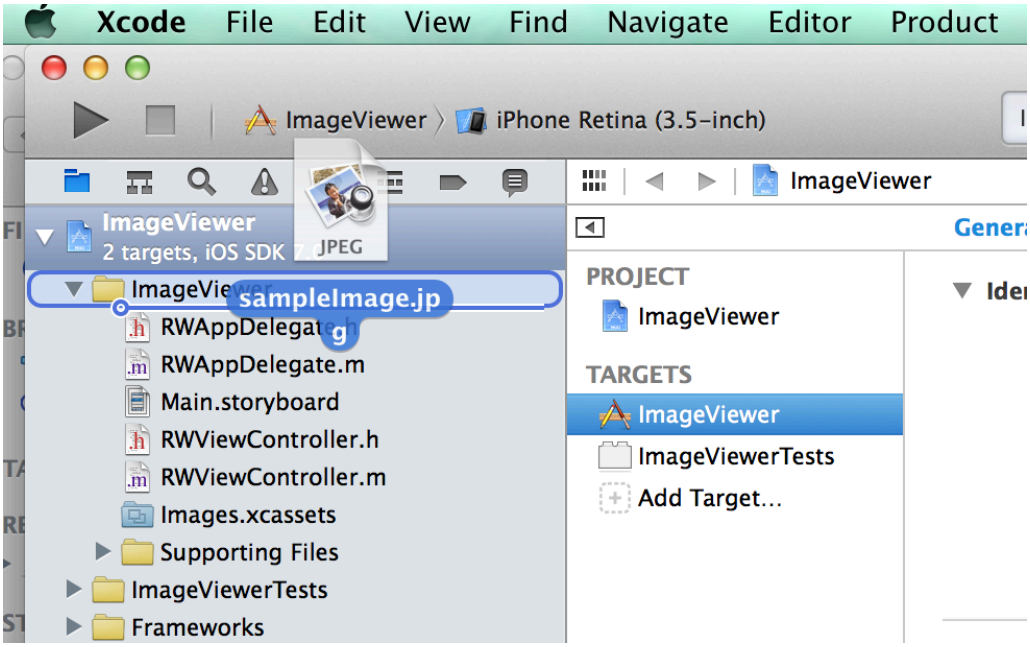
Okay, you have a framework, you have libraries and they're elegant solutions for problems you've not yet encountered. But what's the point of all this?

One of the primary advantages in using a framework is its simplicity in use. Now you're going to create a simple iOS app that uses the

RWUIControls.framework that you've just built.

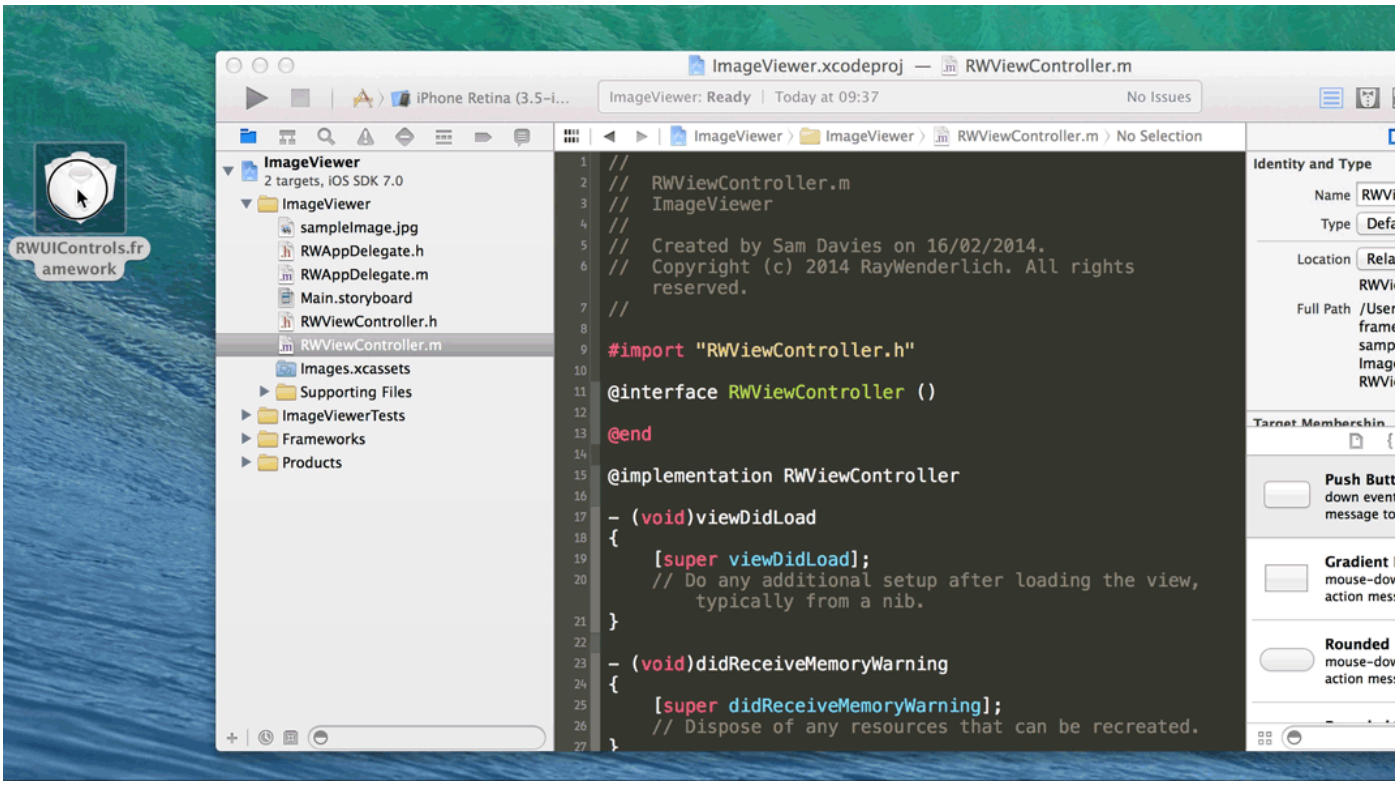
Start by creating a new project in Xcode. Choose ***File/New/Project*** and select ***iOS/Application/Single View Application***. Call your new app ***ImageViewer***; set it for ***iPhone*** only and save it in the same directory you've used for the previous two projects. This app will display an image and allow the user to change its rotation using a ***RWKnobControl***.

Look in the ***ImageViewer*** directory of the zip file you downloaded earlier for a sample image. Drag ***sampleImage.jpg*** from the finder into the ***ImageViewer*** group in Xcode.



Check the ***Copy items into destination group's folder*** box, and click ***Finish*** to complete the import.

Importing a framework follows a nearly identical process. Drag ***RWUIControls.framework*** from the desktop into the ***Frameworks*** group in Xcode. Again, ensure that you've checked the box before ***Copy items into destination group's folder***.



Open up ***RWViewController.m*** and replace the code with the following:

```
#import "RWViewController.h"
#import <RWUIControls/RWUIControls.h>

@interface RWViewController ()
@property (nonatomic, strong) UIImageView *imageView;
@property (nonatomic, strong) RWKnobControl *rotationKnob;
@end

@implementation RWViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Create UIImageView
    CGRect frame = self.view.bounds;
    frame.size.height *= 2/3.0;
    self.imageView = [[UIImageView alloc] initWithFrame:CGRectInset(frame, 0,
20)];
    self.imageView.image = [UIImage imageNamed:@"sampleImage.jpg"];
    self.imageView.contentMode = UIViewContentModeScaleAspectFit;
```

```
    [self.view addSubview:self.imageView];

    // Create RWKnobControl
    frame.origin.y += frame.size.height;
    frame.size.height /= 2;
    frame.size.width  = frame.size.height;
    self.rotationKnob = [[RWKnobControl alloc]
initWithFrame:CGRectInset(frame, 10, 10)];
    CGPoint center = self.rotationKnob.center;
    center.x = CGRectGetMidX(self.view.bounds);
    self.rotationKnob.center = center;
    [self.view addSubview:self.rotationKnob];

    // Set up config on RWKnobControl
    self.rotationKnob.minimumValue = -M_PI_4;
    self.rotationKnob.maximumValue = M_PI_4;
    [self.rotationKnob addTarget:self
                          action:@selector(rotationAngleChanged:)
                forControlEvents:UIControlEventValueChanged];
}

- (void)rotationAngleChanged:(id)sender
{
    self.imageView.transform =
CGAffineTransformMakeRotation(self.rotationKnob.value);
}

- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskPortrait;
}

@end
```

This is a simple view controller that does the following:

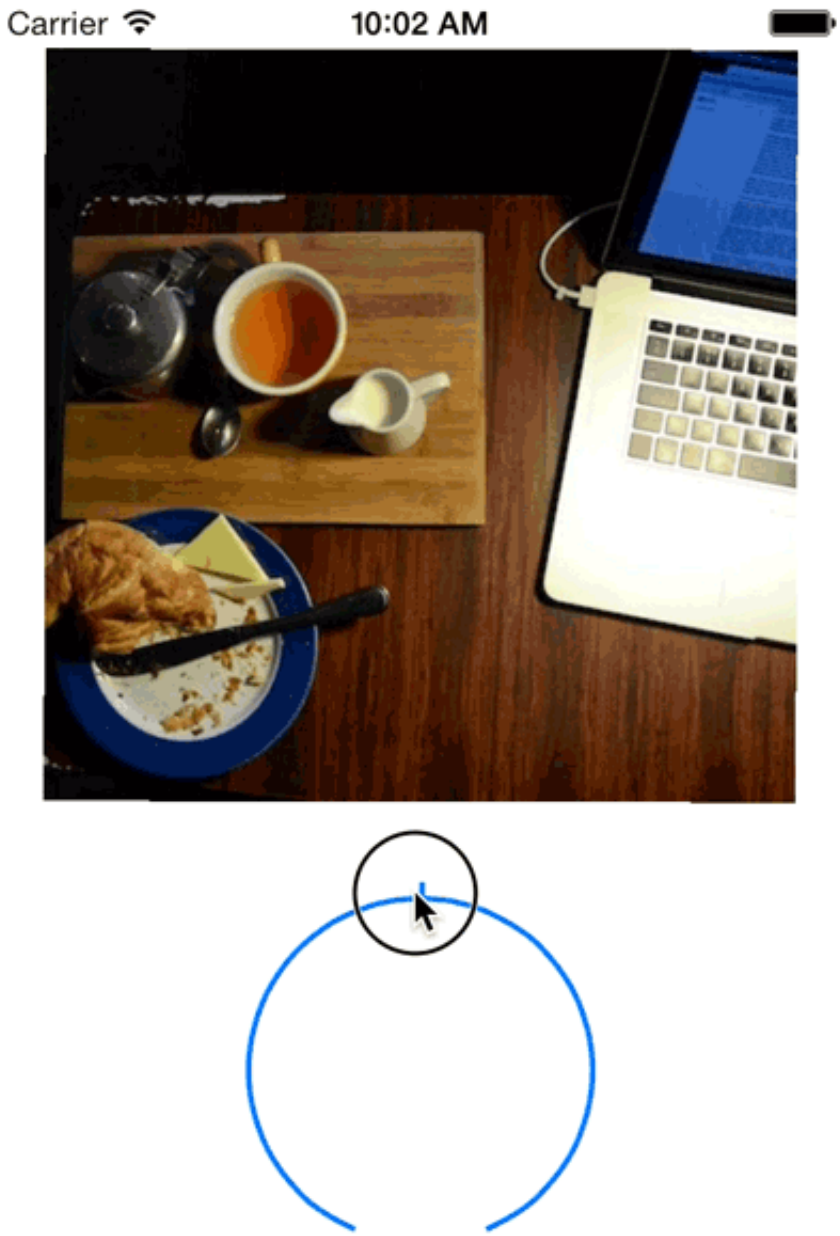
- Import the framework's header with `#import <RWUIControls/RWUIControls.h>`.
- Set up a couple of private properties to hold the `UIImageView` and the

`RWKnobControl`.

- Create a `UIImageView`, and use the sample image that you added to the project a few steps back.
- Create a `RWKnobControl` and position it appropriately.
- Set some properties on the knob control, including setting the change event handler to be the `rotationAngleChanged:` method.
- The `rotationAngleChanged:` method simply updates the `transform` property of the `UIImageView` so the image rotates as the knob control is moves.

For further details on how to use the `RWKnobControl` check out the previous tutorial, which explains how to create it.

Build and run. You'll see a simple app, which as you change the value of the knob control the image rotates.



Using a Bundle for Resources

Did you notice that the ***RWUIControls*** framework only consists of code and headers? For example, you haven't used any other assets, such as images. This is a basic limitation on iOS, where a framework can only contain header files and a static library.

Now buckle up, this tutorial is about to take off. In this section you'll learn

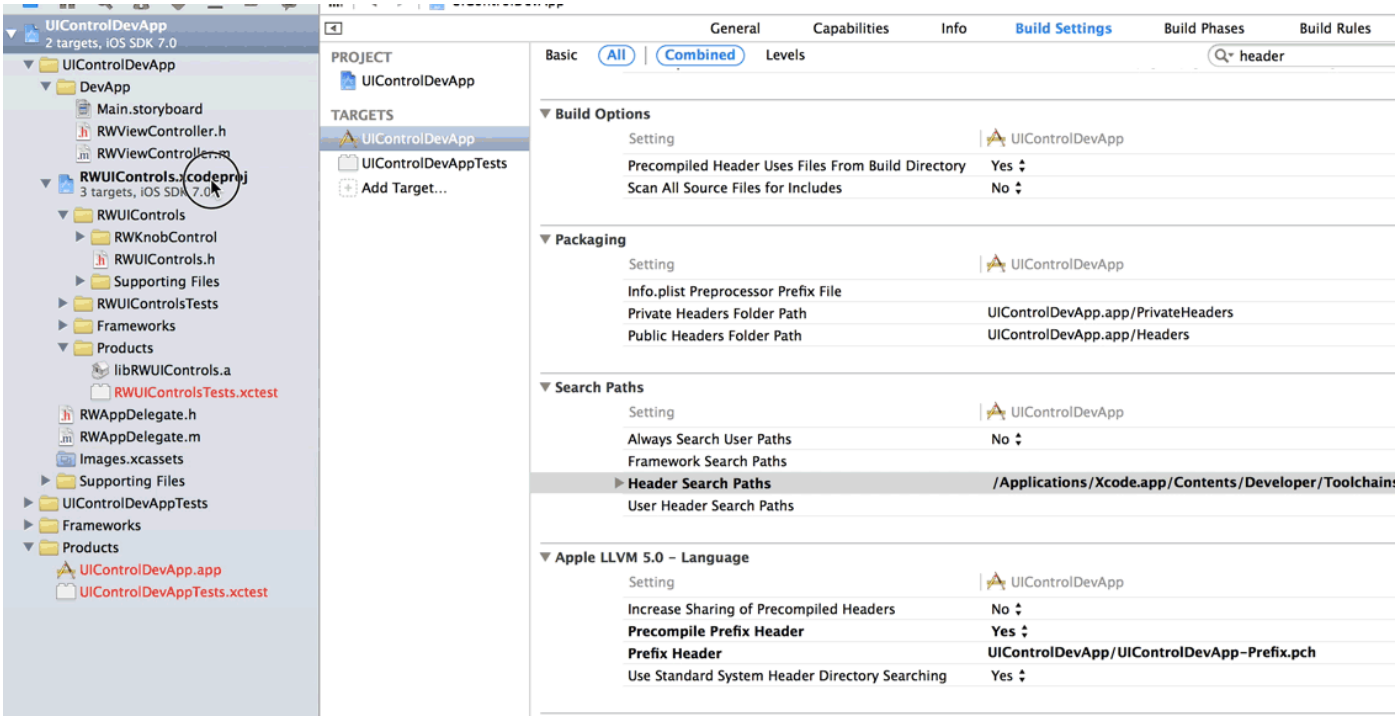
how to work around this limitation by using a bundle to collect assets, which can then be distributed alongside the framework itself.

You're going to create a new UI control to be part of the ***RWUIControls*** library; a ribbon control. This will place an image of a ribbon on the top right hand corner of a `UIView`.

Creating a bundle

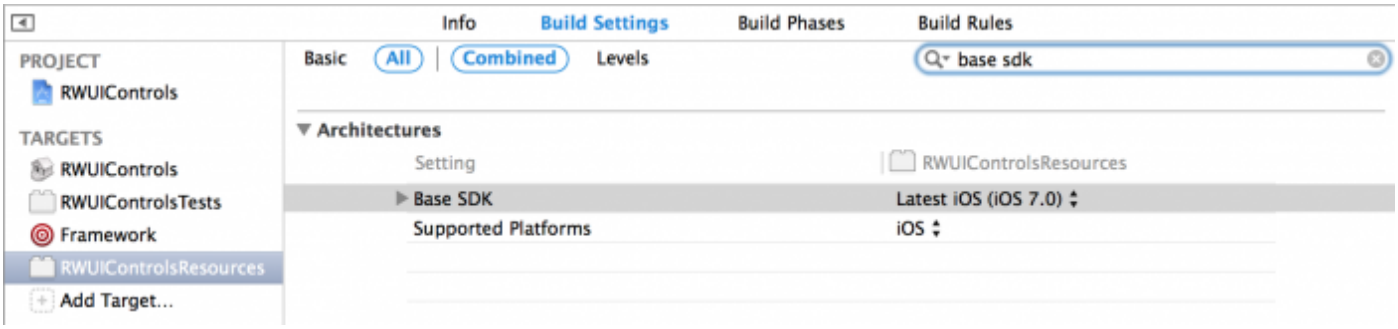
The resources will be added to a bundle, which takes the form of an additional target on the ***RWUIControls*** project.

Open the ***UIControlDevApp*** project, and select the ***RWUIControls*** sub-project. Click the ***Add Target*** button, then navigate to ***OS X/Framework and Library/Bundle***. Call the bundle ***RWUIControlsResources*** and select ***Core Foundation*** from the framework selection box.

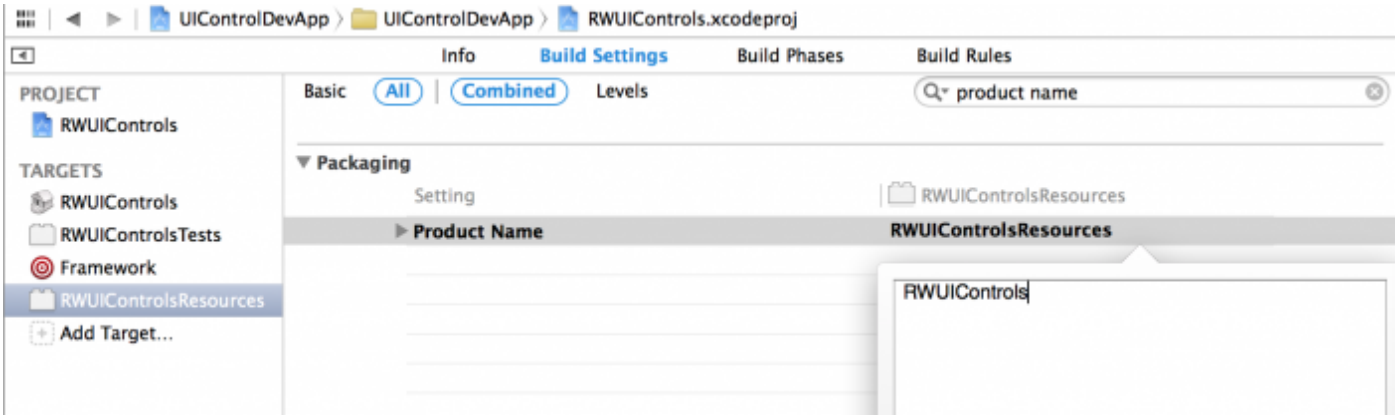


There are a couple of build settings to configure since you're building a bundle for use in iOS as opposed to the default of OSX. Select the

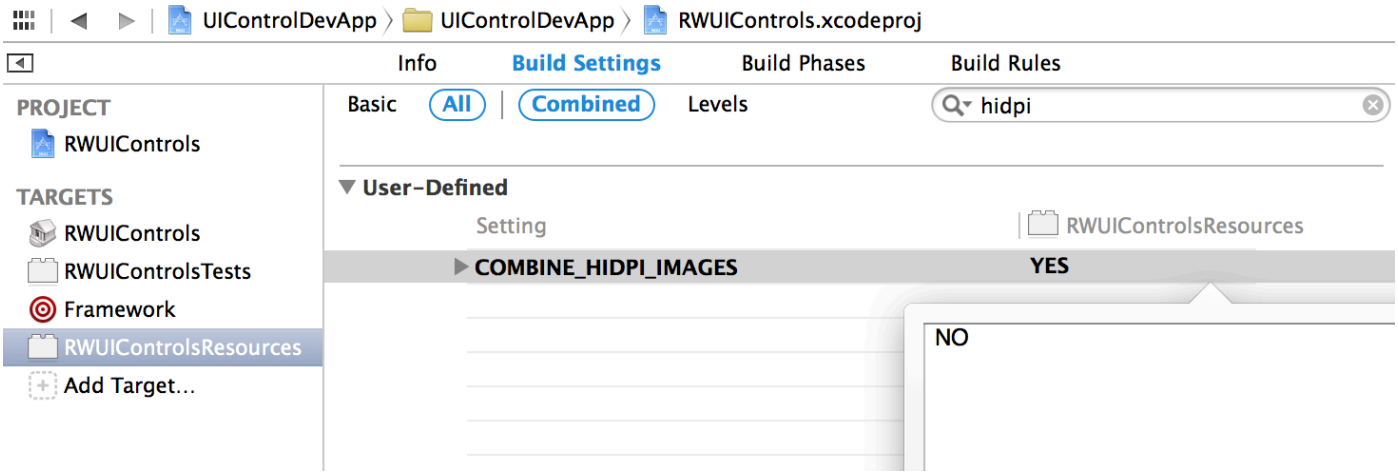
RWUIControlsResources target and then the ***Build Settings*** tab. Search for *base sdk*, select the ***Base SDK*** line and press ***delete***. This will switch from OSX to iOS.



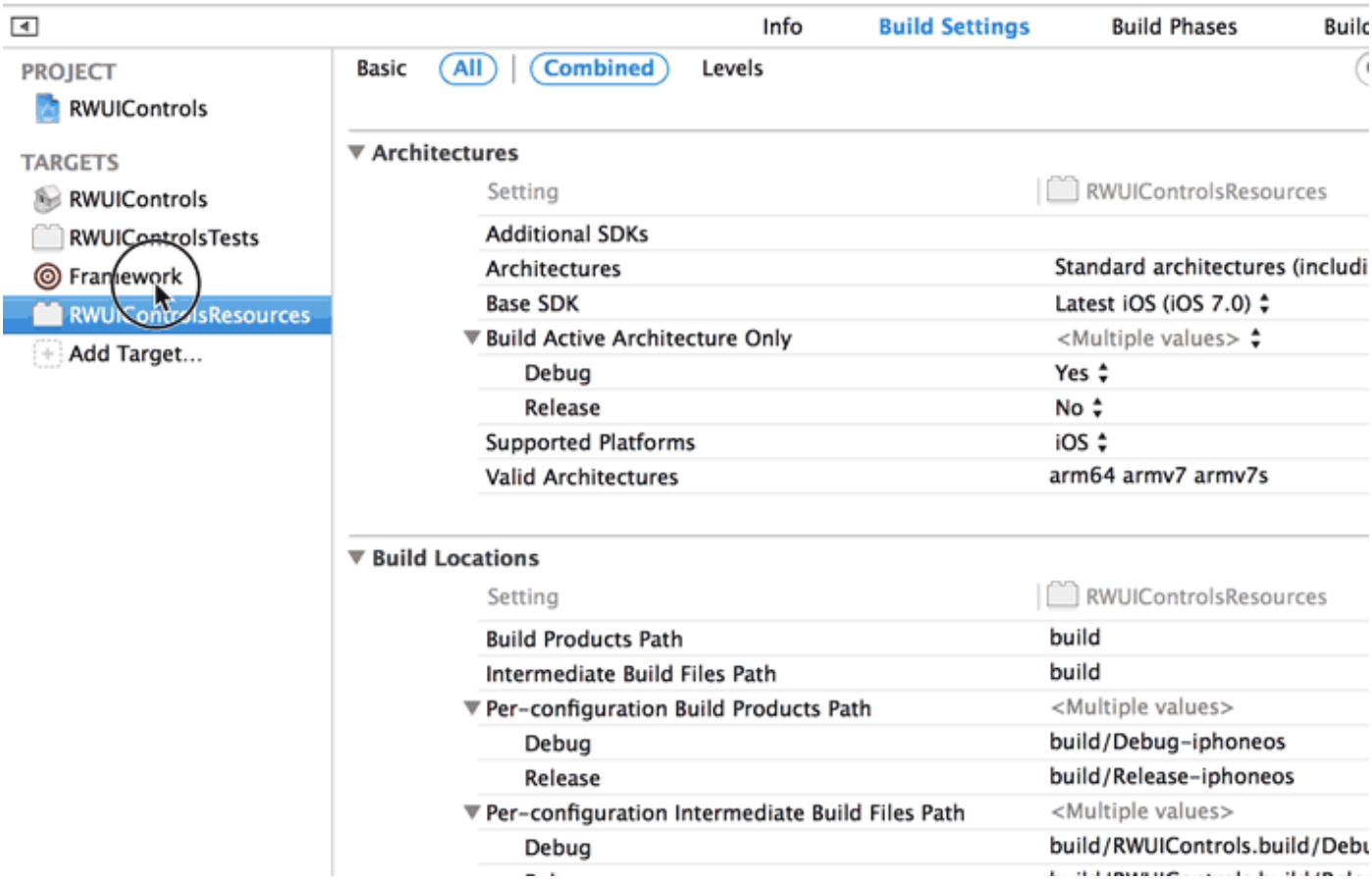
You also need to change the product name to ***RWUIControls***. Search for *product name* and double-click to edit. Replace ***${TARGET_NAME}*** with ***RWUIControls***.



By default, images which have two resolutions can produce some interesting results; for instance when you include a retina @2x version. They'll combine into a multi-resolution TIFF, and that's not a good thing. Search for *hidpi* and change the ***COMBINE_HIDPI_IMAGES*** setting to `NO`.



Now you'll make sure that when you build the framework, the bundle will also build and add the framework as a dependency to the aggregate target. Select the ***Framework*** target, and then the ***Build Phases*** tab. Expand the ***Target Dependencies*** panel, click the ***+***, and then select the ***RWUIControlsResources*** target to add it as a dependency.



Now, within the ***Framework*** target's ***Build Phases***, open the ***MultiPlatform Build*** panel, and add the following to the end of the script:

```
# Copy the resources bundle to the user's desktop
ditto "${BUILT_PRODUCTS_DIR}/${RW_FRAMEWORK_NAME}.bundle" \
      "${HOME}/Desktop/${RW_FRAMEWORK_NAME}.bundle"
```

This command will copy the built bundle to the user's desktop. Build the framework scheme now so you can see the bundle appear on the desktop.

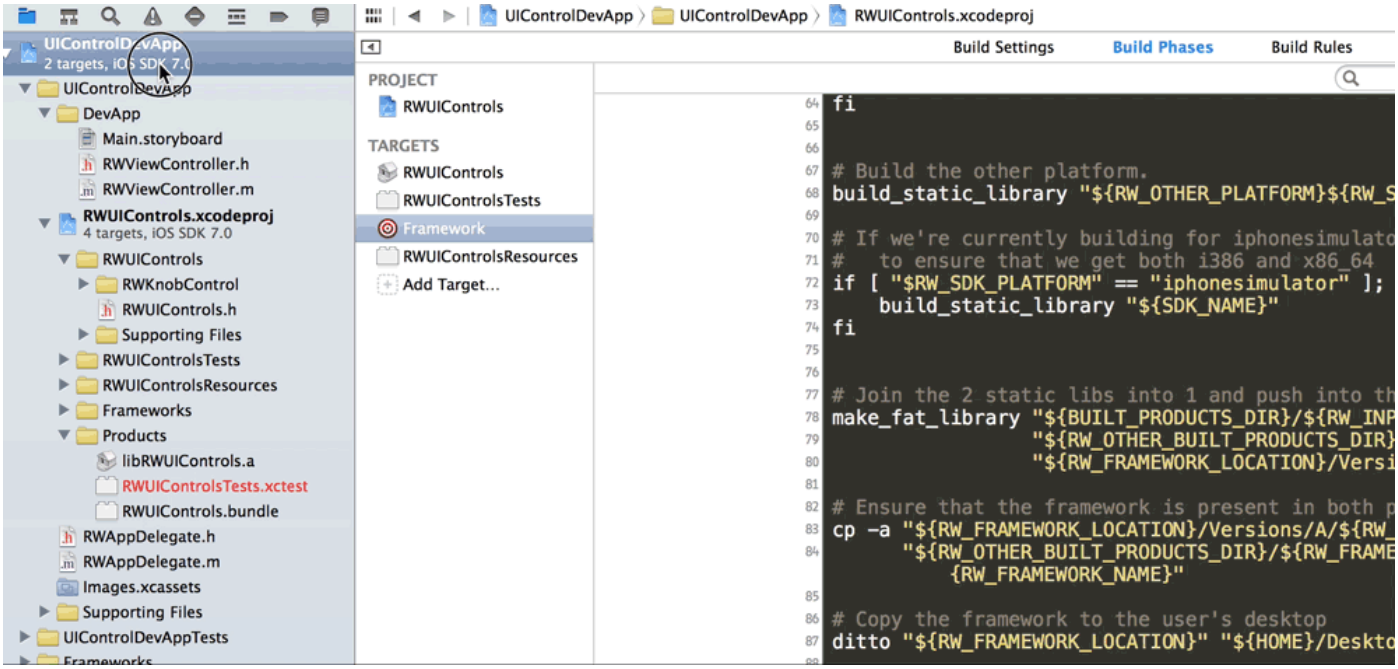
Importing the Bundle

In order to develop against this new bundle, you'll need to be able to use it in the example app. This means you must add it as both a dependency, and an object to copy across to the app.

In the Project Navigator, select the ***UIControlDevApp*** project, then click on the ***UIControlDevApp*** target. Expand the ***Products*** group of the ***RWUIControls*** project and drag ***RWUIControls.bundle*** to the ***Copy***

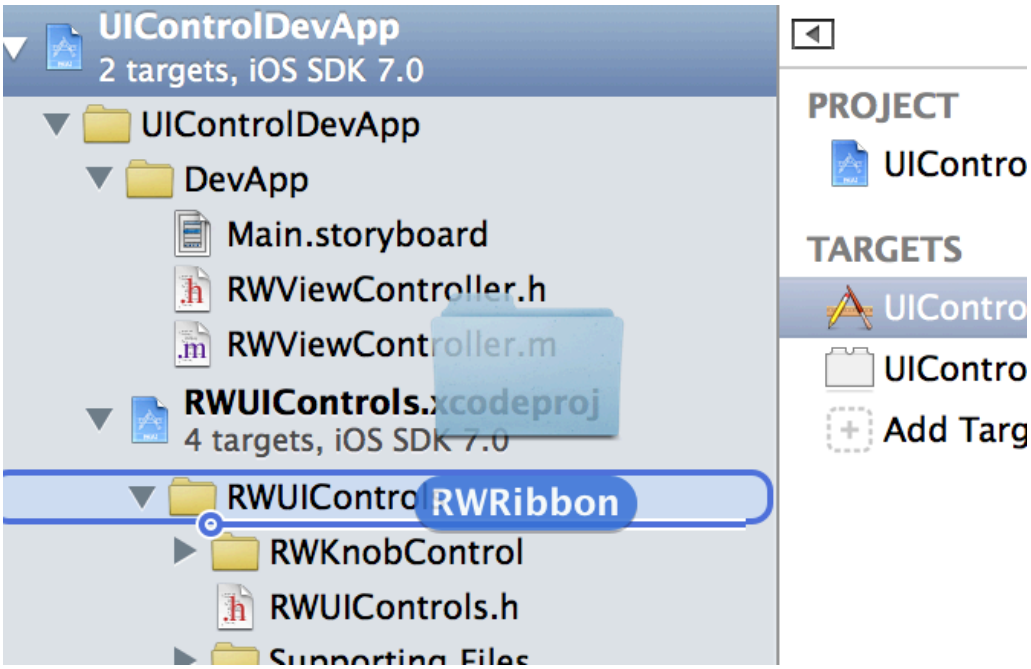
Bundle Resources panel inside the ***Build Phases*** tab.

In the ***Target Dependencies*** panel, click the ***+*** to add a new dependency, and then select ***RWUIControlsResources***.

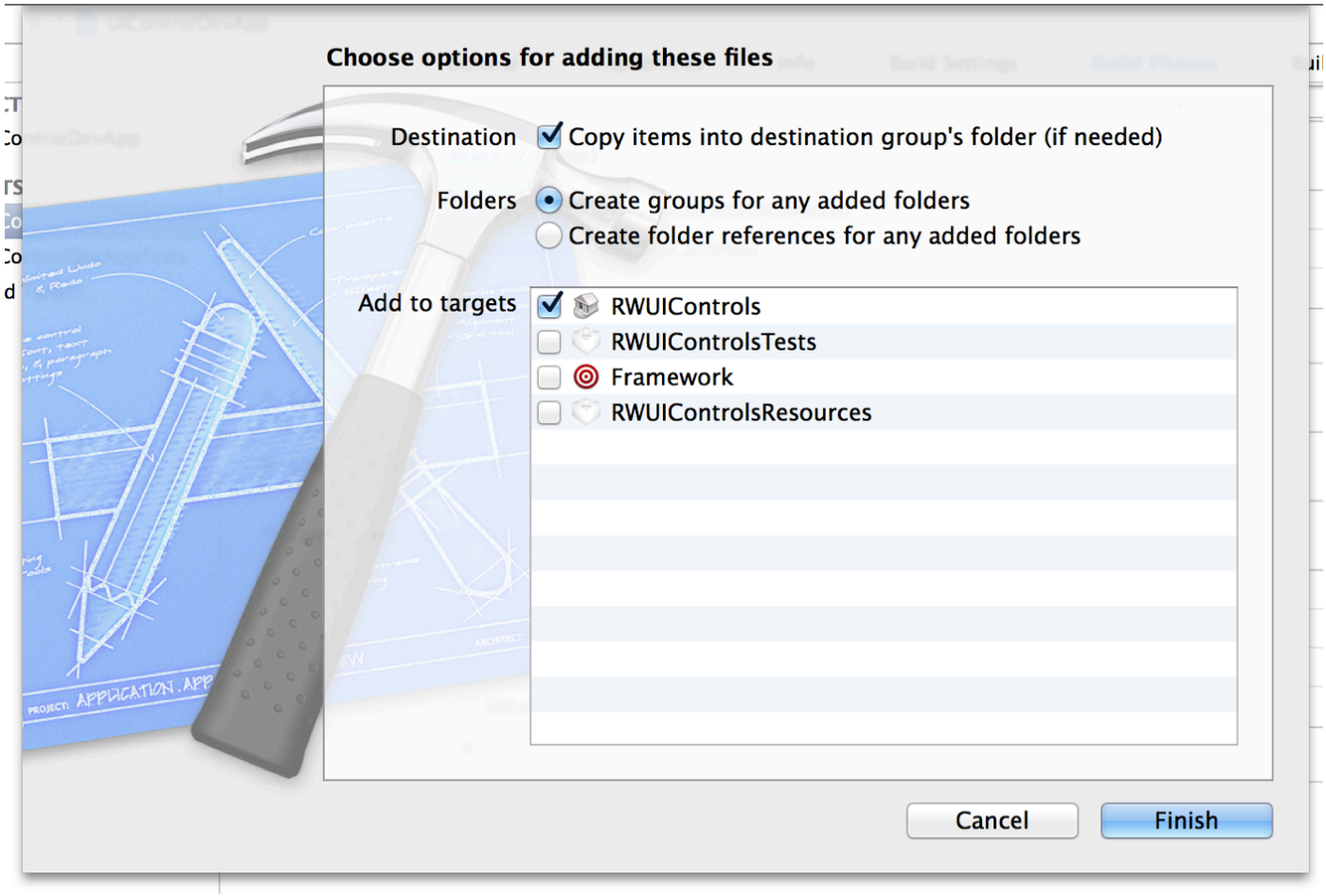


Building a Ribbon View

That's all the setup required. Drag the ***RWRibbon*** directory from inside the zip file you downloaded earlier into the ***RWUIControls*** group within the ***RWUIControls*** project.



Choose ***Copy the items into the destination group's folder***, making sure they are part of the ***RWUIControls*** static lib target by ticking the appropriate box.

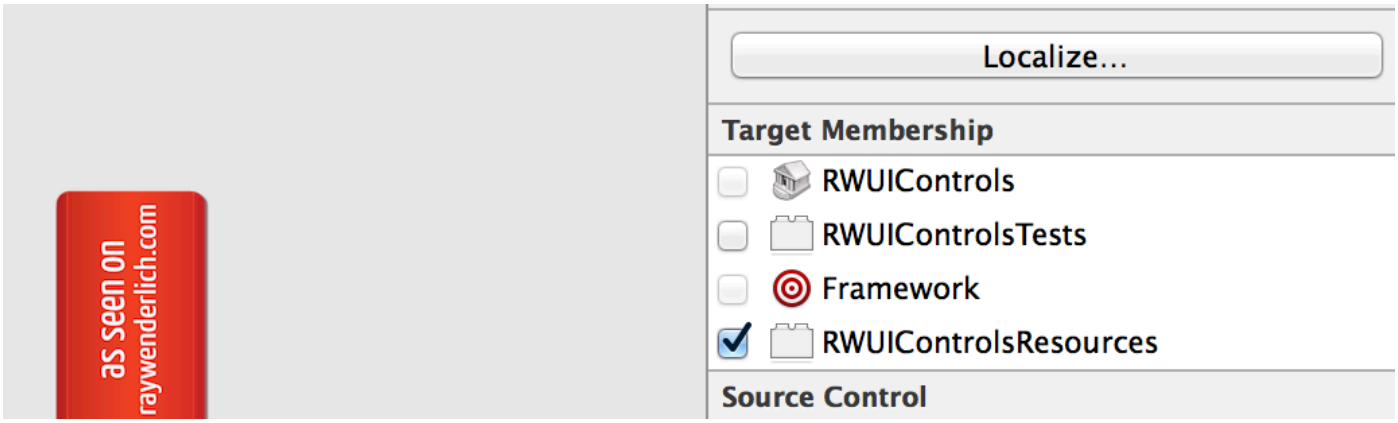


An important part of the source code is how you reference images. If you take a look at `addRibbonView` inside the ***RWRibbonView.m*** file you'll see the relevant line:

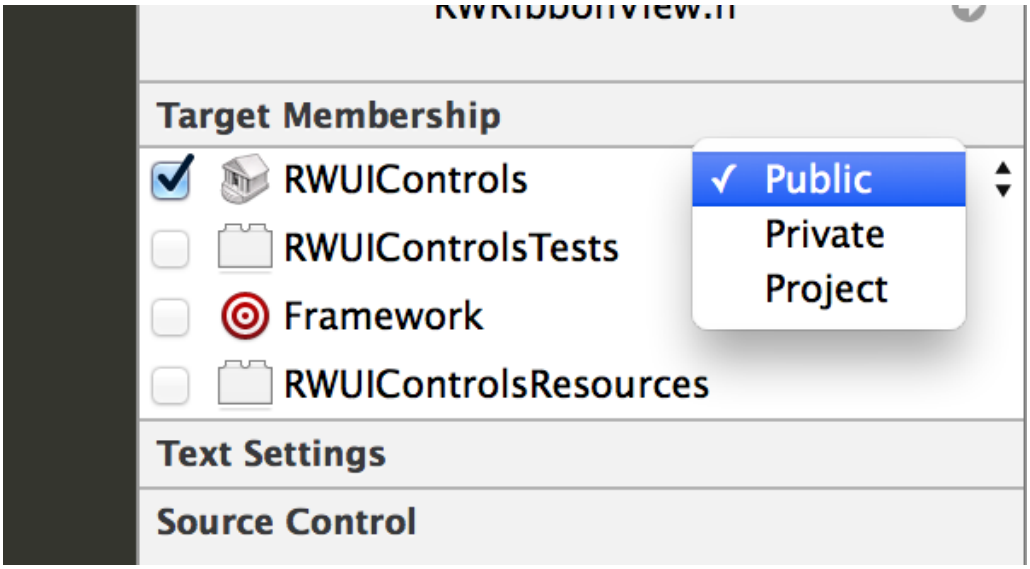
```
UIImage *image = [UIImage imageNamed:@"RWUIControls.bundle/RWRibbon"];
```

The bundle behaves just like a directory, so it's really simple to reference an image inside a bundle.

To add the images to bundle, choose them in turn, and then, in the right hand panel, select that they should belong to the***RWUIControlsResources*** target.



Remember the discussion about making sure the framework is accessible to the public? Well, now you need to export the ***RWRibbon.h*** header file, select the file, then choose ***Public*** from the drop down menu in the ***Target Membership*** panel.



Finally, you need to add the header to the framework's header file. Open ***RWUIControls.h*** and add the following lines:

```
// RWRibbon
#import <RWUIControls/RWRibbonView.h>
```

Add the Ribbon to the Example App

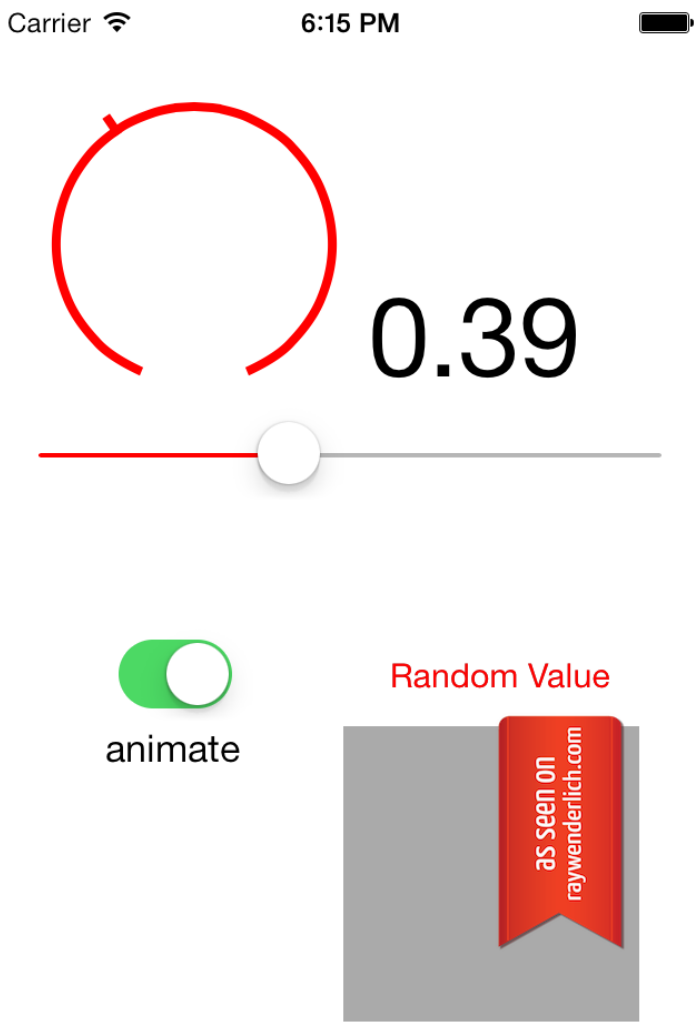
Open ***RWViewController.m*** in the ***UIControlDevApp*** project, and add the following instance variable between the curly braces in the `@interface` section:

```
RWRibbonView  *_ribbonView;
```

To create a ribbon view, add the following at the end of `viewDidLoad`:

```
// Creates a sample ribbon view
_ribbonView = [[RWRibbonView alloc]
initWithFrame:self.ribbonViewContainer.bounds];
[self.ribbonViewContainer addSubview:_ribbonView];
// Need to check that it actually works :)
UIView *sampleView = [[UIView alloc] initWithFrame:_ribbonView.bounds];
sampleView.backgroundColor = [UIColor lightGrayColor];
[_ribbonView addSubview:sampleView];
```

Build and run the ***UIControlDevApp*** scheme and you'll see the new ribbon control at the bottom of the app:



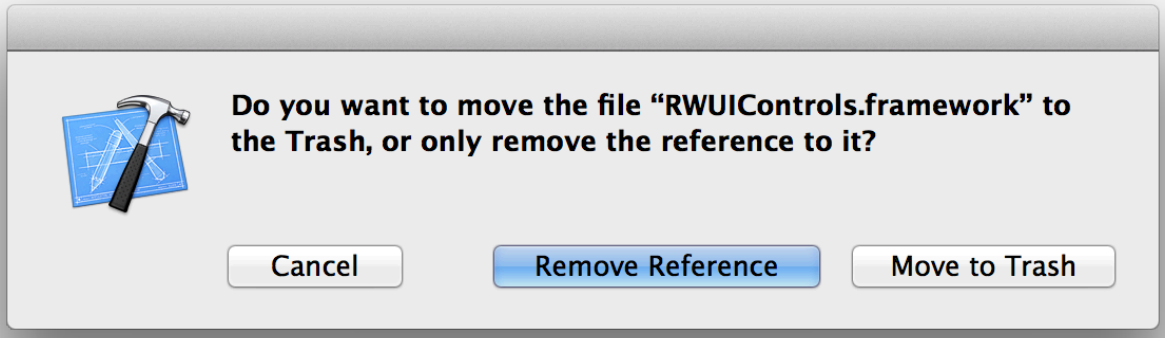
Using the Bundle in ImageViewer

The last thing to share with you is how to use this new bundle inside another app, the ***ImageViewer*** app you created earlier.

To start, make sure your framework and bundle are up to date. Select the ***Framework*** scheme and then press ***cmd+B*** to build it.

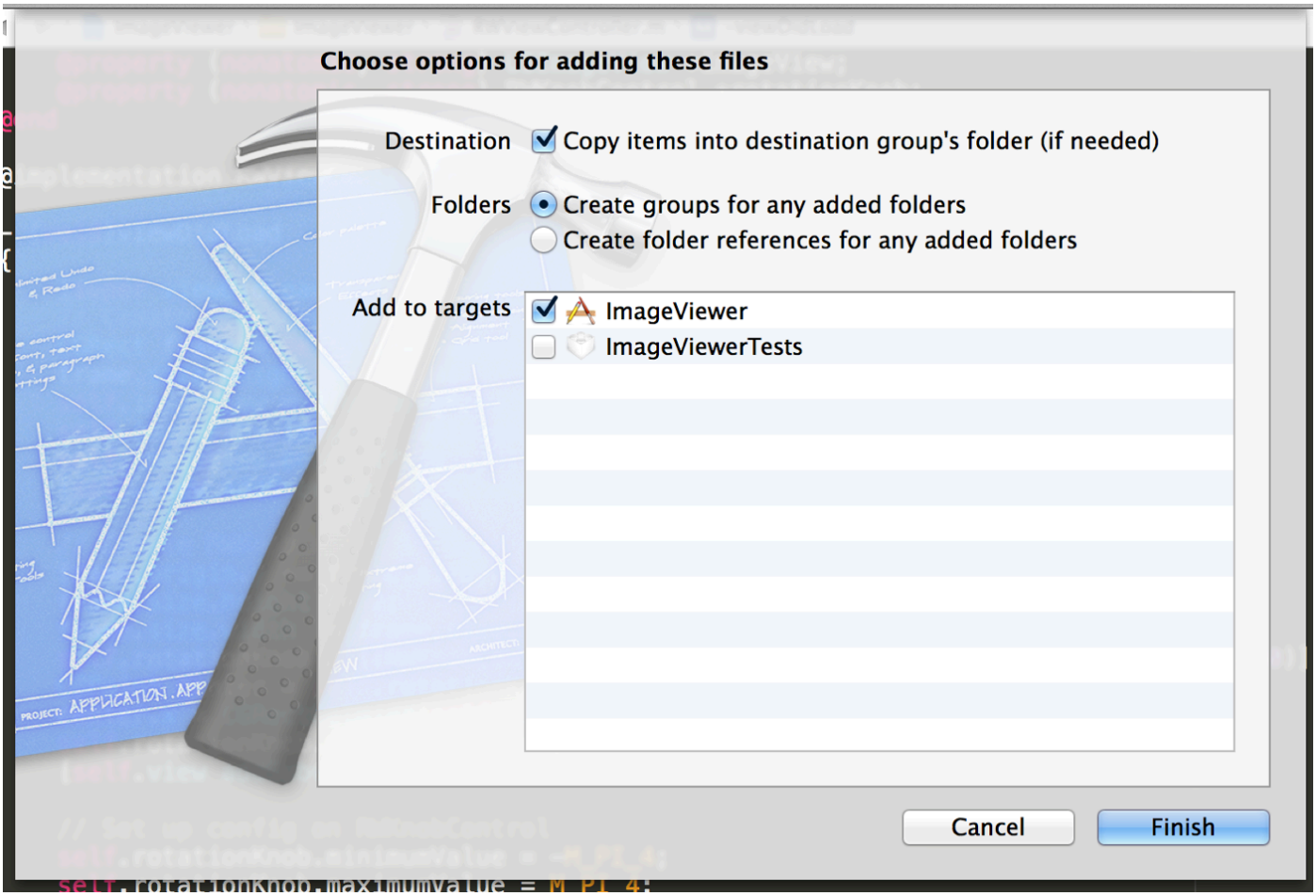
Open up the ***ImageViewer*** project, find the ***RWUIControls.framework*** item inside the ***Frameworks*** group and delete it, choosing ***Move to Trash*** if you're prompted. Then drag the ***RWUIControls.framework*** from your desktop to the ***Frameworks*** group. This is necessary because the

framework is much different than it was when you first imported it.



Note: If Xcode refuses to let you add the framework, then it might not have properly moved it to the trash. If this is the case then delete the framework from the ***ImageViewer*** directory in Finder and retry.

To import the bundle, simply drag it from the desktop to the ***ImageViewer*** group. Choose to ***Copy items into destination group’s folder*** and ensure that it’s added to the ***ImageViewer*** target by ticking the necessary box.



You're going to add the ribbon to the image, which rotates, so there are a few simple changes to make to the code in***RWViewController.m***.

Open it up and change the type of the `imageView` property from `UIImageView` to `RWRibbonView`:

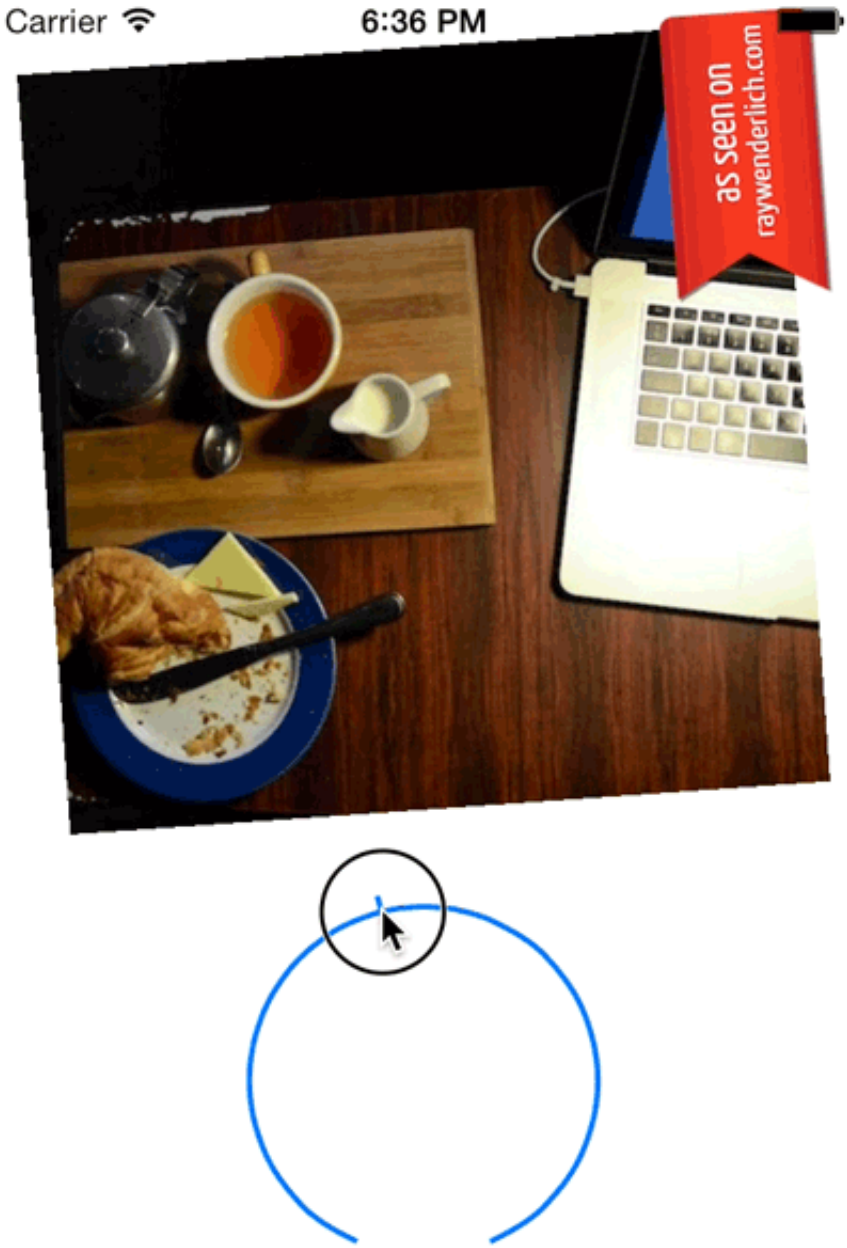
```
@property (nonatomic, strong) RWRibbonView *imageView;
```

Replace the first part of the `viewDidLoad` method which was responsible for creating and configuring the `UIImageView`, with the following:

```
[super viewDidLoad];
// Create UIImageView
CGRect frame = self.view.bounds;
frame.size.height *= 2/3.0;
```

```
self.imageView = [[RWRibbonView alloc] initWithFrame:CGRectInset(frame, 0,
20)];
UIImageView *iv = [[UIImageView alloc] initWithFrame:self.imageView.bounds];
iv.image = [UIImage imageNamed:@"sampleImage.jpg"];
iv.contentMode = UIViewContentModeScaleAspectFit;
[self.imageView addSubview:iv];
[self.view addSubview:self.imageView];
```

Build and run the app. You'll see you're now using both the ***RWKnobControl*** and the ***RWRibbonView*** from the***RWUIControls*** framework.



Where To Go From Here?

In this tutorial, you've learned everything you need to know about building a framework for use in your iOS projects, including the best way to develop frameworks and how to use bundles to share assets.

Do you have a favored functionality that you use in lots of different apps? The concepts you've learned can make your life easier by creating a library

you can access over and over again. A framework offers an elegant way to procure a library of code, and gives you the flexibility to access whatever you need for your next series of awesome apps.

The source code for the completed project is available on Github, with a commit for each build step, or as a downloadable zip file.

Add a rating for this content