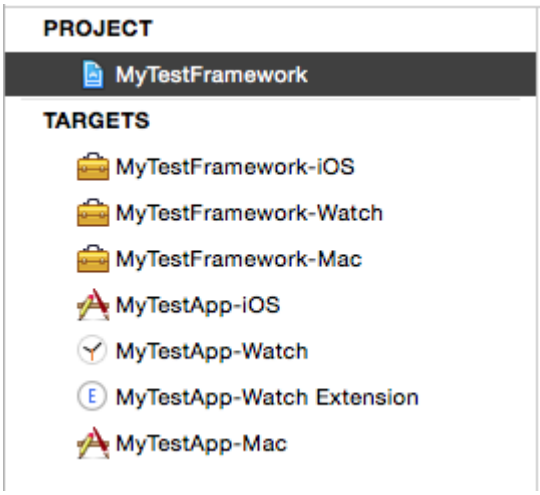
Using Xcode Targets

Background

Recently Apple announced two new development platforms, watchOS and tvOS. As a result developers are scrambling to try out these new platforms and add support to their existing libraries and apps.



This might be a familiar sight to you. While this is an entirely viable way to build your code for multiple platforms, it does introduce a number of complications that greatly increase the risk of breaking something due to user error.

Repeated Actions

Source files must be added to all the framework targets by hand, same goes for any new files you add. Since this is very easy to miss doing from the new file dialog, it is generally discouraged to add the same file to multiple targets (if what the targets are building are not significantly different). You not only have to maintain the code for the frameworks individually, but also all the build settings for them. This plus adding tests for all the targets quickly becomes an unreasonable goal to set for any development team without breaking all the frameworks into separate entities -- and away from a singular code-base.

Understanding Intent

While having separate targets makes it easier to explicitly link your application to your framework, it doesn't make understanding the frameworks any easier. While working on the code you may understand that the separate targets only

exist as a convenience of explicit linking at the expense of some technical debt of maintaining that code. However in 6 months time, is that still the case? Have your separate frameworks become distinctly different for each platform, or has code been written multiple times because someone forgot to add a file and didn't have time to audit if the code was already written somewhere else? While good documentation can cover these cases to a degree, it doesn't make understanding the whole system any easier.

Future Compatibility

The multiple targets approach also has little to no future compatibility support. If yet another development platform is introduced you are going to compound the problems of repeated actions for maintainability and understanding intent with another target that may cause you to deviate from the original architecture style entirely.

Targets

In Xcode, a target contains a collection of:

- files
- environment variables
- build system rules
- build system actions

The target also defines the relationship between all these collections. First by grouping the files with build system actions, this organization is called "Build Phases". These phases are very generic definitions of actions. Then the target will then filter each file through the set of build rules to resolve the specific action to take based on the phase it is in. Finally, the environment variables (build settings) are applied to the actions to invoke additional functionality.

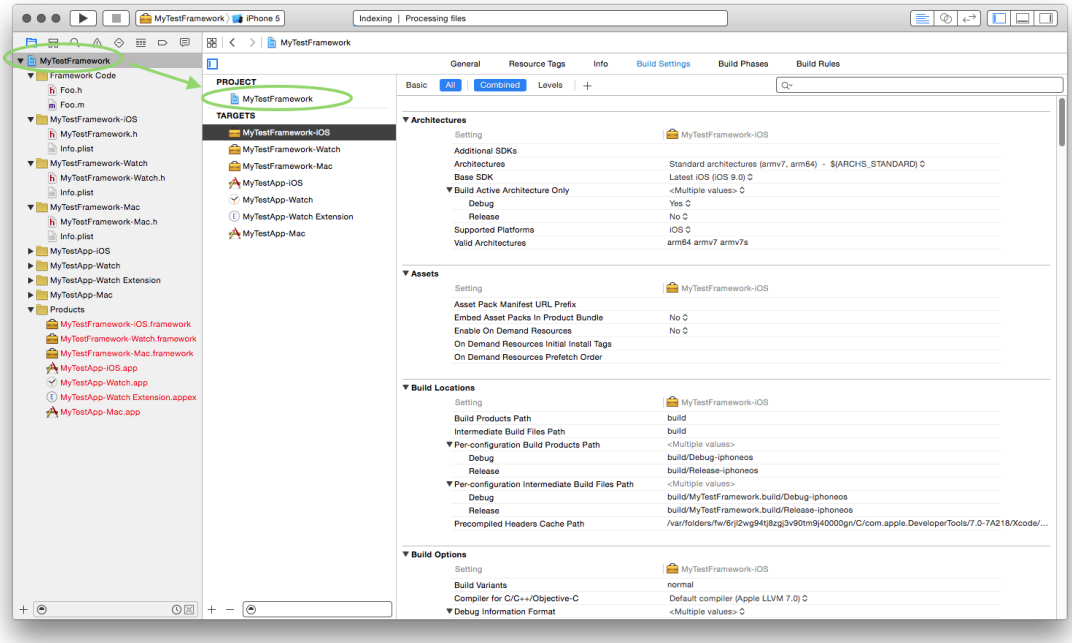
Managing Targets

The intent of a target is to define an output (build product) for a set of conditions. Over time these conditions can become extremely complex, duplicating that complexity with minor changes across all targets can become a recipe for disaster. Fortunately Xcode provides a number of tools that allow us to consolidate the complexity into a single place where it is easier to manage.

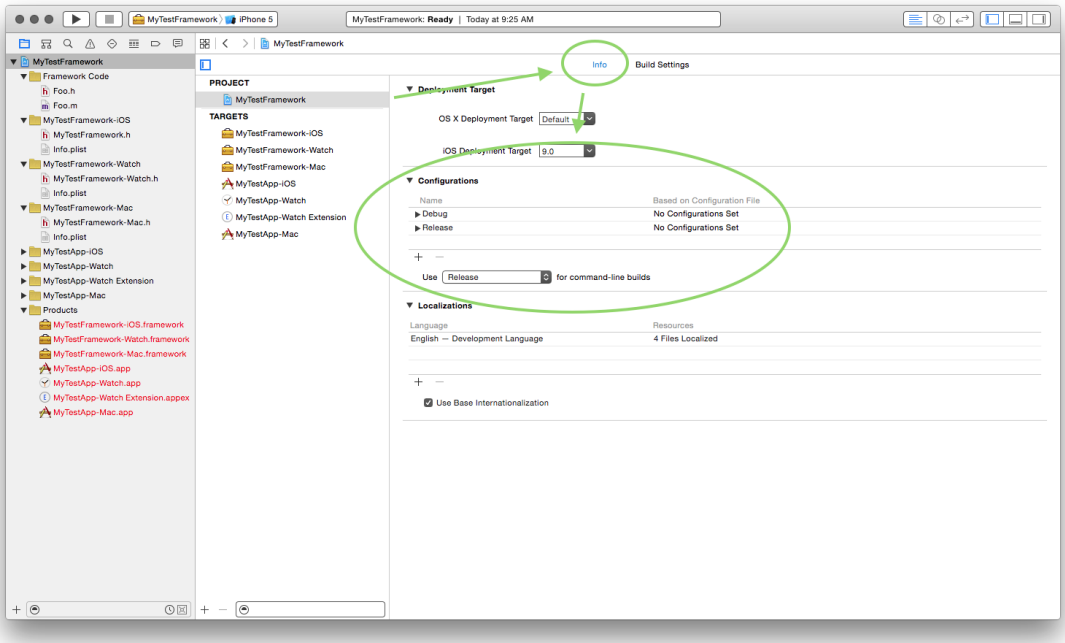
Build Configurations

Build configurations are a way to conditionally manage build settings per target.

Under the "Build Settings" panel of a target you can expand each setting to select a different option per configuration. Commonly they are used to change the signing identity and provisioning profile based on building "Debug" vs "Release". However they can be put to a lot more use for you.



Accessing the settings for your project, settings that are configured here will be inherited by all the targets in your project.



Accessing the build configuration editor for this project file.

When adding new build configurations, new entries will be added for each target in the project file. As mentioned in section about build configurations in "Managing Xcode", each target can have a different xcconfig file assigned to it based on build configuration. If you have multiple targets that differ in how the settings are configured rather than differing in code implementation, then using build configurations to abstract those changes can help prevent introducing the complexity of another target to your project file.

Note: Please take note of the dropdown menu underneath the list of build configurations. When running `xcodebuild` it will automatically pick a build configuration if there is none specified. This dropdown menu selects the default configuration it will use. When performing a build via `xcodebuild` and not the Xcode GUI (application) you should always specify the build configuration name to use. This should be done even if you are specifying a scheme for it to build.

Switching from using multiple tagets to multiple build configurations is going to be non-trivial for a project of any size, and I wouldn't recommend doing that directly. To make the transition easier you should first migrate your existing build settings per target into their own xcconfig file.

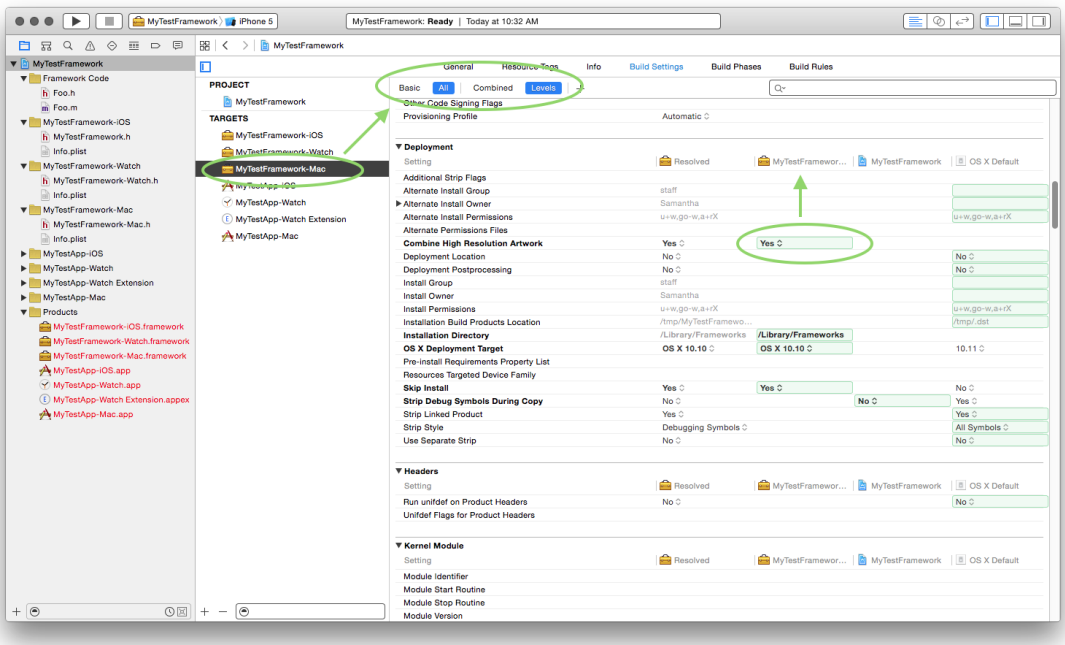
xcconfig files

If you haven't read my post on xcconfig files yet, please check it out before continuing.

While changing anything in a working build system is a risk, there are some big benefits you get for using xcconfig files instead of build settings stored on the project file.

1. Xcode.app cannot change your build settings, build settings can be changed on the project file, these don't impact the values in the xcconfig file.
2. The xcconfig files exist as separate plain text files, this means they will rarely conflict when merging changes and will be easy to resolve when they do (unlike project files).
3. The build settings panel in Xcode has no ability to undo changes. Since xcconfig files are regular text files they will have undo history and the ability to add comments; this makes editing your project settings a less daunting task and can communicate more about your build requirements.

Migrating build settings from the project file to an xcconfig file is made very simple. For the following steps I would recommend you open the build settings panel for the desired target in Xcode and select the options "All" and "Levels" to display the levels of build setting inheritance. You should be focusing on the values that are listed under the column with the name of your target in it. Look for cells in this column that are green and have bold text. These are values set for this target specifically.



1. Start by creating a new xcconfig file and add it to your project.
2. Open that as a separate window, you will need to see this along-side the settings of the target.

3. Select the row of the setting you want to migrate out of the project file and into the xcconfig file.
4. Perform a Copy (`cmd+C` or from the menu Edit->Copy).
5. Now select the xcconfig file and paste, you will get the setting and the values based on configuration.

Note: The copy action isn't very "smart" so sometimes you may need to expand the build setting and select the row of the build configuration for that setting to extract the value. Sometimes you cannot get the right value and it may have to be recreated entirely (please file a radar on this). As mentioned please consult my guide to xcconfig files (linked to at the top of this section) as this can be challenging to migrate complex projects.

Warning: You may run into some problems with having pairing the test target with the framework. I am going to try to get another post up with a holistic explanation of migrating build settings to xcconfig files as soon as possible and link to it here. Until then be aware this can cause issues due to Xcode wanting to resolve test host targets by string name. This is a fault with **Xcode** not with the targets or build configurations.

When you have copied all your settings specific to that target into the xcconfig file, you can go back to the build configurations and assign the xcconfig file to that target for a specific build configuration. Before doing anything else you should back up your xcode project now. To make sure you set up the xcconfig file correctly you will have to remove the values from the target. You can clear a build setting by pressing the "delete" when selecting a row that has a green cell in the target's column. If everything went well, the column named "Resolved" (the left-most column in the build settings panel) will not change, because the value assigned to the target should be pulled from the xcconfig file now.

Schemes

Schemes are the final step in managing build configurations. Schemes are for creating associations between types of build actions (Build, Run, Profile, Test, Archive, Export). Xcode will helpfully automatically create new schemes for each of your targets, but you don't have to let this dictate how you use schemes. The important thing to understand about schemes is that they allow you to manage not only the targets to build for a particular action but the build configuration that those targets should use.

So to recap this:

```
|-- Project File
    |-- Targets
        |-- Files
        |-- Build settings (these are configurable based on build confi
    |-- Build Configurations
        |-- xcconfig file is associated to each target in the project f
    |-- Schemes
        |-- Build Actions (Build, Run, Profile, Test, Archive, Export)
            |-- Targets (Built using a specific build configuration, th
```

A scheme can specify a single build configuration to be used for multiple targets. This means that instead of having schemes for each framework and then each application target, you can have a single scheme that tells a single framework target to be built using a specific configuration and associated xcconfig file (you can set this up to be able to change the requirements of building based on target OS, so a single target could build for iOS, Mac, watchOS, and tvOS) before building the application target. Doing this means cutting the footprint of your code down significantly as well as removing the complexity of managing multiple targets.

The ability to configure Xcode to work this way is by no means "new" functionality. All the functionality mentioned here has existed since Xcode 4. If you examine many of Apple's open source packages they are built using the methods talked about in this post. Abstracting the steps of the build system like this is a fairly standard practice. There are many ways to go about doing this organization, this is only one way to reduce the complexity of interfacing with the build system.

If you have any questions about this post or any other post, please check out my Q&A.
