Team Tech Guide: Xcode's Targets, Schemes, & Configurations

Written by Sean Langhi on 2021-02-01 for the members of the Afterpay Mobile Rewrite iOS team.

Purpose of this document

You are a developer.
You're involved in an iOS app development project.
You have a need—whether you realize it or not—to understand the vocab and key ideas surrounding Xcode's build system. In particular, you want to grasp the "what" and "why" behind *targets*, *schemes*, and *configurations*.

This guide is designed to help you get there.
It provides beginner-friendly commentary aimed at lowering the barrier to initial comprehension.
It also includes links to relevant Apple docs.

This guide does not aspire to *replace* Apple's documentation.
The Apple docs are detail-rich and thorough, and the spirit of this subject really dwells in the details. You'll need to spend time with the official material in order to master this subject.

However, the official material is scattered across several docs, each of which presumes some background knowledge—so ramping up in this subject area can be challenging.
Reading this guide can help you transition from "no prior knowledge" to a more comfortable position from which to approach Apple's material.

Vocabulary

Product

A *product* is the output created when Xcode runs its build process. It's a ready-to-use app or library: in other words, a binary artifact made from your source code. The binary artifact may be wrapped up in a "bundle", which is a structured directory that includes resources (like media assets and an Info.plist). In the same way that earthworms eat dirt and produce soil, the Xcode build system eats your source code and produces "products".

What you do with a *product* is publish it or run it—for instance, by uploading your app to the App Store, or by booting your app in a Simulator, or by hosting your library on GitHub for others to download and use.

Target

Xcode needs instructions for how to make a desired product. The recipe containing those instructions is called a "target".

A *target* is a "plan" that describes which files will be scoped into a build, what the build system will do to them, and what the build's output (the product) will be called.

In other words, *targets* are all about **builds**. The spirit of a *target* is **"source in, product out"**.

A target lives inside an Xcode project. That's a one-to-many relationship: one project, many targets. A project is the parent of one or more targets, and every target belongs to exactly one project.

Targets live in projects because *files* live in projects, and targets are very hands-on with specific files.

An app project often has one eponymous target for its app executable, which we might call its "main target", and another 1–2 targets for its test executables. (If it includes both unit tests and UI tests, those would be built and run as separate executables, so they get separate targets.)

Targets can depend on the products of other targets, which means their builds have to be chained together. Note that test targets typically depend on the project's main target, since they test its code. That's why, when you add new source files to a project, you only have to add them to the main target, and not to the test targets.

Developers are likely familiar with "build phases", which are recipe steps like running custom scripts in the middle of a build or copying files to a bundle. These fall under the domain of a *target*. When you add, remove, or change build phases, you're editing a target.

Developers are also likely familiar with Xcode's "build settings", which are the 500+ configuration options you can pass to the build system. These settings will affect the behavior of the compiler and linker, which may in turn produce differences in your app's runtime behavior. You can specify build settings by editing a *target*…but please don't! There are actually multiple affordances in Xcode that allow you to specify build settings. These methods can be used at the same time, they're not equally discoverable, and they clobber each other silently according to complex rules, creating a maintenance nightmare. The most hygienic approach is to use `.xcconfig` files exclusively and leave all the other methods alone. We'll discuss this more in a later section about `.xcconfig` files.

Scheme action

A *scheme action* is an **Xcode verb**. Xcode comes with six verbs out of the box: Build, Run, Test, Profile, Analyze, and Archive. These verbs live in the Product section of the top-of-screen toolbar menu, and most of them have default key bindings. If you've ever pressed Cmd+R in Xcode to run your app, or equivalently clicked the "Play" triangle in Xcode's top toolbar, you've invoked a scheme action.

“Build” is the root of all verbs. Run, Test, Profile, Analyze, and Archive all *leverage* Build as a preliminary step. (If you look hard enough, you can find exotic menu commands like “Run without building”, which tells Xcode to run the cached result of the most recent build. That's not really an exception; it's just reusing the result of an earlier build.)

As it turns out, all of these “scheme actions” have programmable behavior. You can program a set of behaviors for all of the “scheme actions” by creating a “scheme”.

Scheme

A *scheme* is a **set of definitions for the six verbs** (scheme actions). You can create as many schemes as you want; at any given time, no more than one of them can be currently *active* in the Xcode IDE. The active scheme is the one that currently defines the six verbs, telling Xcode what behaviors to perform when a given action is requested. (If you delete all your schemes, you will have no active scheme, and Xcode will yell at you when you try to perform a verb like "Build" or "Run", because it doesn't know what behaviors it should do.) You can change the active scheme using the bespoke segmented drop-down menu at the top of the Xcode window. The left segment chooses the scheme, and the right segment chooses the destination (device or Simulator), which is technically outside the scope of the scheme.

Choosing one or more targets

The scheme’s first and most important job is to dictate *which target(s) to build* when you run each of the six different verbs. Recall that a target defines a complete build process, and you can have multiple targets in your project. Note that any given verb can build a chain of *multiple*

targets, if it wants to. The mighty *scheme* brings order and certainty to this swarm of possibilities using a grid of checkboxes in the “Build” section of the Scheme Editor: the rows are targets, the columns are verbs, and the checkboxes correlate the two. The order of the rows matters—higher rows are built first—and you can drag the rows to reorder the targets they represent.

Defining the other five verbs

The verbs “Run”, “Test”, and “Profile” all want to *execute* the built product. That means the scheme has to provide some additional instructions about how to invoke the binary. Recall that *targets* have nothing to say about how a product actually gets *executed*; they deal only with the logistics of the build process, and their purview ends as soon as the product gets created. So, if you want your executable to be invoked with certain command-line arguments or environment variables in place, the scheme editor is the place to specify them.

The verbs “Run”, “Test”, “Profile”, “Analyze”, and “Archive” all adopt a *build configuration*, in addition to their other settings. (A “build configuration” is a simple noun selected from a drop-down menu.) The choice of build configuration determines which `.xcconfig` files will be used to define custom build settings. We'll discuss this more in an upcoming section about `.xcconfig` files.

Pre-actions and post-actions

Did you know that any Xcode verb can include hooks that run custom scripts or send automatic emails? It’s true. Schemes may attach these extra hooks to the six scheme actions. You can set this up in the Scheme Editor by clicking the disclosure triangle next to any of the scheme actions.

Build Configuration File (`.xcconfig`)

Typing in build settings by hand using Xcode’s UI is a bit of a mess. More importantly, it plays poorly with Git: diffs and merge conflicts are very hard to read and resolve. So, in a stroke of mercy, Apple eventually rolled out a new tool called the “Build Configuration File”. It’s easier to refer to these files by their extension: we call them `.xcconfig`s.

An `.xcconfig` is a simple text file that follows a restrictive grammar. It contains some lines of the form “A = B”, where “A” is a build setting’s technical name and “B” is a value or formula. It can also contain comments prepended with `//`. Blank lines are allowed, too. These files are easy to read, edit, and compose by hand.

The point of an `.xcconfig` file is that it gets roped into the build system somehow—more on that in a moment—and then the `.xcconfig` applies its settings forcefully, clobbering any conflicting build settings specified using the manual UI from the project editor.

The moral of the story is, we should use `.xcconfigs` all the time. If you want to change a build setting, do it in an `.xcconfig`. This convention makes our lives easier in two ways: it means we enjoy a pleasant Git experience, and it reduces confusion about where the effective build settings are coming from.

As an important detail, note that **`.xcconfig`s can include other `.xcconfig`s**. This is useful because it allows you to place universal defaults in a single file, which your other, configuration-specific files can import. The importing file has precedence over the imported file, so it's feasible to import a large set of defaults and then override some of them.

Build configuration

A *build configuration* is a noun that sits in a drop-down menu. By default, an Xcode project has two such nouns: "Debug" and "Release". You can delete or change these, and you can make up your own.

These nouns each contain a tiny bit of data: a simple mapping of *targets* to *build configuration files* (`.xcconfig` files). The right-hand side can be blank if desired, meaning that a target uses no `.xcconfig` files. The mapping also includes one extra row where the left-hand side isn't a target, but is actually the *project*. An `.xcconfig` that dwells there will provide its settings as project-wide defaults.

Configurations have no intrinsic behavior; they're passive, declarative entities. To put a configuration into effect, hook it up to a *scheme action*; then, the scheme action's build will use settings from the configuration's `.xcconfig` files.

Many Players Determine Build Settings

Precedence order

Now that we have all the info from the "Vocabulary" section, we're ready to discuss how build settings are really determined.

It's a five-rung ladder, listed below from bottom to top. Mightier rungs (listed later) clobber weaker rungs (listed first).

At the bottom, we have an invisible, implied rung:

1. System defaults. These do exist for every setting. Many of them are simply a blank value. You can see the system default value for each project setting by opening the UI editor for project settings and clicking an affordance to change its view mode to "Computed".

Now we're into the explicit realm, starting with project-land.

2. Settings from the very humble *UI editor for the project-level settings*. **Don't use this.** The UI editor is deprecated. **`.xcconfig` files are strictly better.**

3. Settings from the *`.xcconfig` file mapped to the project-level scope by the build configuration selected for the current scheme action in the currently active scheme*. Stated more simply: "the project scope, via `.xcconfig`".

Now we're out of project-land and into target-land.

4. Settings from the *UI editor for the target's settings*. Note that this is the **one** weird case where the UI editor can clobber an `.xcconfig`—and it happens only because targets clobber the project scope. That said…**never use this. Use `.xcconfig`s, always.**

5. Settings from the *`.xcconfig` file mapped to the target-level scope for the current target by the build configuration selected for the current scheme action in the currently active scheme*. Stated more simply: "the target scope, via `.xcconfig`". This is the high and mighty, the über-setting over which nothing can prevail. If you specify a value here, it's *set* that way, full stop.

Too many cooks: Just use `.xcconfig`s

Five sources of truth is TOO MANY. Nobody wants to think that hard about where a project setting's value is being effectively specified! Fortunately, we can improve our developer experience by agreeing as a team to use `.xcconfig`s and nothing else. For further benefit, we can keep all our `.xcconfig`s in a single folder, so there's no mystery about how many of them exist and where we can find them.

Learn this lesson well: use a collection of `.xcconfig`s, keep them all in a single folder in your project, and ***never*** *modify build settings through the UI editor accessed by clicking on the project*.

Have an `.xcconfig` called "Project Defaults". Apply it at the project level in each of your configurations. If you need your project-level defaults to vary per configuration, create an `.xcconfig` for each variant and let it import the one called "Project Defaults".

If your targets need specific settings, create `.xcconfig`s for those too, and hook them up to your schemes appropriately.

If you use this strategy, there will be only two places to look if you want to understand where your overrides are happening: the scheme editor, and the `.xcconfig` files named in the scheme editor. You also will enjoy the benefits of Git, and of convenient manual editing. This is a big win for the whole team!

External links

Apple docs

Text articles

- Xcode Concepts
- Building Your App
- Managing Schemes
    - (This is not the same as the Xcode help article with a similar name, listed below.)
- Manage Schemes (Xcode help article)
- Configuration Settings File (xcconfig) format
- Add a build configuration (xcconfig) file
- What is the build system?
- Evaluate build setting inheritance

Handy diagrams

- Build system interactions
- Build setting inheritance

Third-party articles

- Cocoacasts: Managing Build Configurations in Xcode

Lists of all build settings

- Official Apple reference
- Unofficial site with better formatting (xcodebuildsettings.com)