Managing Xcode

There is no silver bullet to organizing code. Everything from the code design patterns to third-party dependency management solutions change based on how a project is architected. This post aims to provide an explanation of the concepts and tools that can be used to better organize and manage your own projects.

Table of Contents

Workspace Files

A workspace file (".xcworkspace") is a container for other types of files, typically Xcode project files. Workspace files can also contain schemes. This is an organizational tool to help define implicit as well as explicit relationships between the files contained in the workspace.

Managing Projects

The primary use for workspace files is the management of multiple project files. A workspace file serves as a means of creating an implicit relationship between all the files it contains. What this means for project files, is that the built products from one project can be used to build another.

For example, this enables the separation of library code from your app's code. In practice you may a project that has: one target for building your app, one target for building a library for an iOS device, one target for building a library for the iOS Simulator, and one aggregate target to combine the two library targets into one binary. By using a workspace to manage this, you could make a separate project file that is only for building the library, and contain the library-related targets. This makes your app's project file much cleaner and only contain code relevant to the building the app. This will create an implicit dependency between building the library and building the app.

↑ Parent

↑ Table of Contents

Project Files

Project files are used to define build environment, tools, resources, and contain references to code files. They create explicit relationships between the contents. This means that project files should be used as a container for targets that have explicit dependencies for code to be built successfully. Like workspace files, project files can also contain schemes.

Nested Projects

To define explicit dependencies between targets in separate project files, project files can be nested. This means that a target can set explicit dependencies of any other target within the same project file, or any nested child project file. A target in a child project file cannot set explicit dependencies of any target in the parent project file.

This behavior is useful for writing wrappers around another set of APIs. It cuts out a lot of work in resolving implicit dependencies on a per-target basis and will communicate build errors in a more clear manner.

↑ Parent

↑ Table of Contents

Targets

A target defines a specific set of build settings that accompany a set of build rules. Targets do not have to produce any output files, they can be used as both a means to create as well as to organize.

Applications

Application targets are used to build executable binaries. For many developer this is the type of target that is dealt with the most. They contain explicit dependencies, as well as implicit dependencies on other code libraries and resource assets. The built product from application targets are what is deployed to be run and tested.

I use application targets fairly often in the development process. When starting to implement any major code or UI feature I will create a separate project/workspace to implement it in a clean environment. I have found this to be beneficial to the development process for a number of reasons:

- Small code footprint, faster for prototyping ideas and understanding edge-cases.
- By writing new code in isolated environments, the chance of introducing bugs into the main app is greatly reduced.
- Checking functionality and usability is easier for the developer to test.
- Doesn't start with any code cruft, and only brings in dependencies it absolutely needs.

As a bonus benefit: if you encounter any bugs in Apple's frameworks, you are already working from a minimal project you can use to isolate and attach to a radar.

↑ Parent

Libraries and Frameworks

Library and framework targets build code library binaries (for an in-depth explanation of different types of libraries and frameworks see this post). Libraries act as a means of code separation and dependency management. Each code library has a specific purpose and function with the rest of the code it gets integrated with. Separating code out into separate libraries and frameworks makes it both easier to manage and maintain for the developer.

For complete separation of code, new project files can be added to a workspace that only contain individual code libraries. This can speed up total compilation time on a project because Xcode can resolve the implicit dependencies and build some libraries in parallel.

↑ Parent

External

Unlike app or library targets, external build targets do not use Xcode's build environment. Instead, the work of configuring and building code is delegated to an external tool such as `make`, `cmake`, `autotools`, or any external executable script.

External build targets are extremely useful for integrating any code that doesn't come with a .xcodeproj of its own. They can also be used to build external Xcode projects without having to load the target

project's contents into your open workspace.

When using external build targets to build code, it is recommended that you use a build tool directly rather than a script. This is because Xcode's build log will attempt to parse the output given by the specified build tool and display it with similar formatting to non-external targets.

↑ Parent

Aggregate

Aggregate targets do not directly produce a build product. They act as a means of organizing the building of other targets in the same project file into a single step. This is particularly useful when dealing with multiple layers of dependencies. Like other types of targets, additional build phases can be added (run scripts, copy files, etc).

Aggregate targets can also be used for building multiple disparate targets. For example, building both a dynamic framework and a static library version of your code from the same dependency target. This makes aggregate targets extremely flexible and useful for configuring builds.

↑ Parent

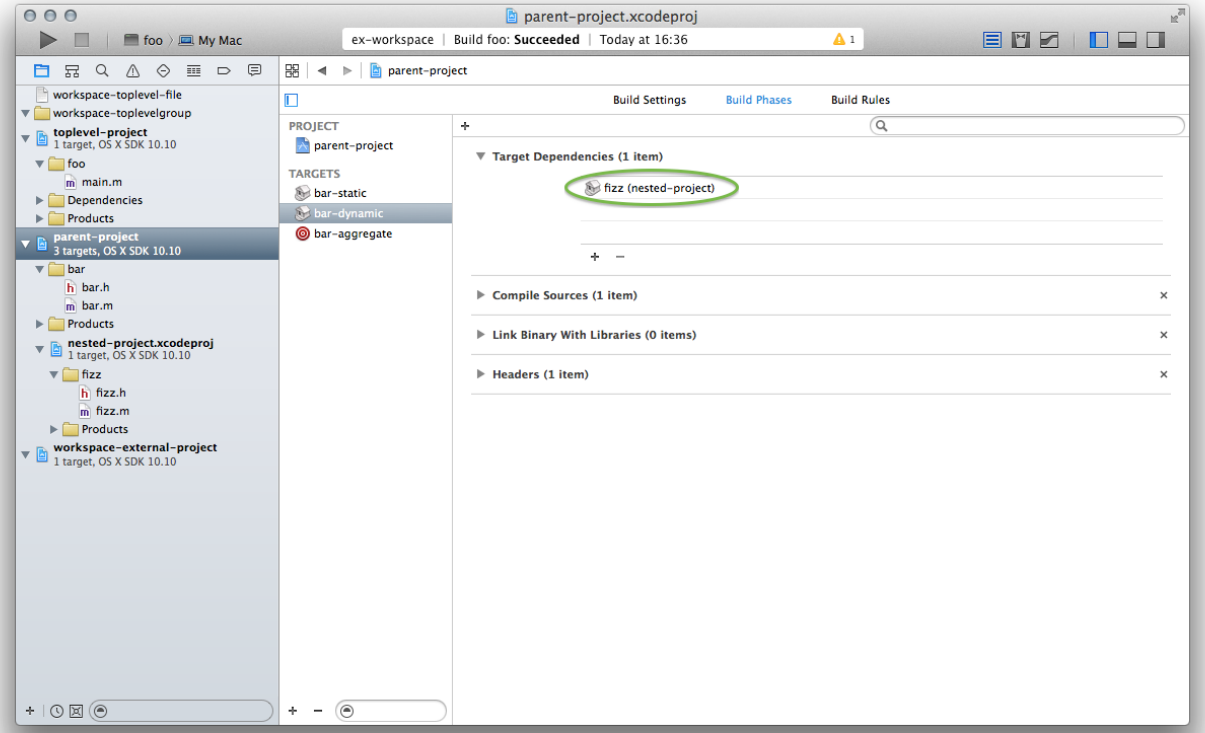
↑ Table of Contents

Dependencies

Dependencies are simply pre-requisites for building a target. Target dependency management is one of the more complex aspects in Xcode. Understanding of Xcode's toolchain and build environment configuration helps with making dependencies more manageable.

Explicit

Explicit dependencies are dependencies that are visible by a target. A target's explicit dependencies can be found in the "Target Dependencies" section under "Build Phases", as shown below.



This is a dependency that states that it must be explicitly built before building the rest of the target. There is ordering to building this type of dependency.

↑ Parent

Implicit

Implicit target dependencies are dependencies that are necessary to successfully build a target, but aren't explicitly defined. Any target that requires a target from another non-nested project file to be run for it to build successfully has an implicit dependency. For example, an application target that must link a set of libraries, has an implicit dependency to the targets that produce those libraries. By using separate projects and targets for each component of your code, Xcode can resolve the built products to find the targets that need to be built first. Using implicit dependencies is preferred, as it allows Xcode to optimize by building multiple libraries at the same time (this, and edge-cases, are covered in more detail in the "Build" section of scheme configuration).

The most common implicit dependency is by adding linked libraries to a target. When adding a linked library to a target, a dialog appears that lists all the build products that are defined in the workspace. Implicit dependencies are highly dependent on the configuration of the current scheme to be resolved correctly.

↑ Parent

External

"External" dependencies are implicit dependencies that Xcode cannot resolve on its own. This is another edge-case to implicit dependencies, but it deserves special mention as it is one of the more

complex things that can be done with the dependency system. Examples of external dependencies include:

- Auto-generated headers or other files as part of another build process
- Libraries managed by something external to the project
- Product of a script that cannot be validated immediately

External build tool targets can sometimes result having external dependencies. It depends on how the build system was setup and configured. However most external build tool targets integrate into Xcode, so it can resolve the implicit dependencies for you.

↑ Parent

↑ Table of Contents

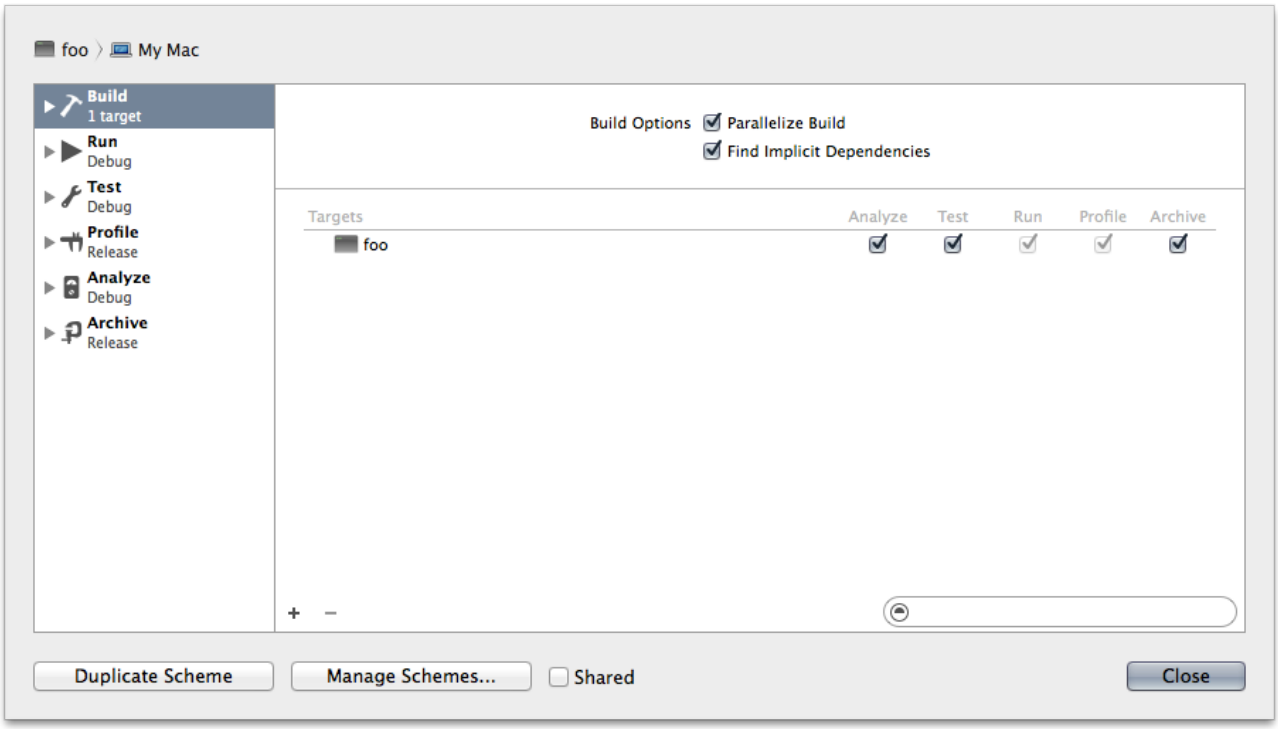
Schemes

Schemes are one of the most powerful organizational tools in Xcode. These allow you to define dependencies and give them ordering, also they allow fine-grain tweaking of the build settings associated with a target based on the build configuration. A scheme supports multiple actions that can be configured to perform specific tasks based on the action.

Actions

Build

The "Build" action allows you to configure which targets should be built for other actions. The ordering of the targets list is meaningful to the order that they are built in when the action is run. The ordering is represented by top-to-bottom as first-to-last build order.



This window also contains two very important checkboxes for changing how builds work.

Parallelize Build -- This option allows Xcode to speed up total build time by building targets that do not depend on each other at the same time. This is a time-saver on projects with many smaller dependencies that can easily be run in parallel.

Find Implicit Dependencies -- This is a very powerful option that allows Xcode to resolve what targets need to be built for the primary target of the scheme to be built successfully. However, this does come with some sharp edges that you have to be aware of.

Situation: You link a library a library against your application target and create an implicit dependency to that library's target.

- Scenario 1: "Find Implicit Dependencies" is enabled.
    - Result: The library will get built prior to building the application target. The application target will then link against the library and build successfully.
- Scenario 2: "Find Implicit Dependencies" is disabled.
    - Result: The library will not get built prior to building the application target. The application target fail to build.
    - Fix: To ensure that the second scenario does not happen, you must add the necessary targets into the targets list and order them correctly.

However there are some edge-cases to this behavior. If you find yourself in a situation where you cannot rely on Xcode's ability to resolve the necessary target dependencies correctly, then both "Parallelize Build" and "Find Implicit Dependencies" should be disabled on the scheme.

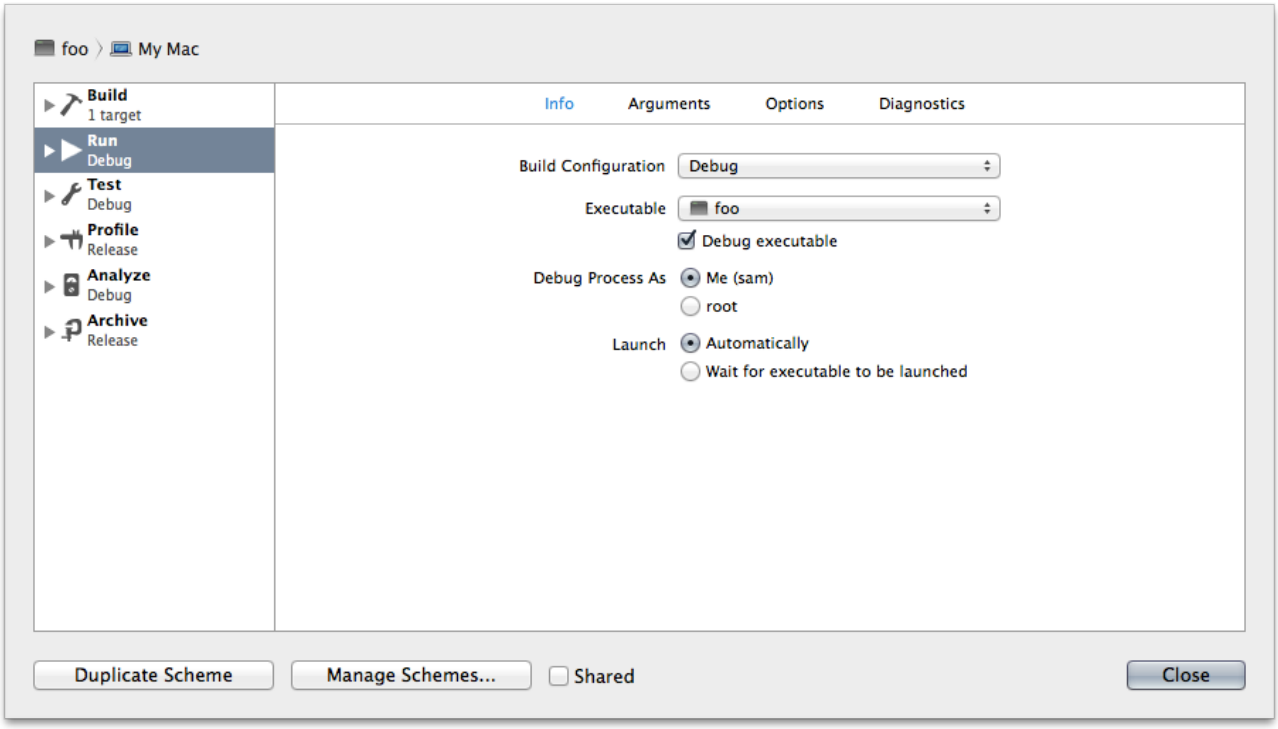
As of this writing, both Xcode 5 and 6 exhibit some very confusing behavior with these scheme build options. Specifically if you need to disable these options on any particular scheme, the schemes for the targets that you add MUST also have these options disabled on them as well. If the options are not disabled, then they will bleed through to the scheme you are building. This is important to note because the ordering of the targets list is not static while "Find Implicit Dependencies" is enabled. Xcode can and will re-order the targets listed there as it attempts to optimize the build. When working with external dependencies or requirements that Xcode isn't able to resolve, this will result in broken builds.

This type of build configuration is very complex and difficult to manage in Xcode. I have written a tool that uses `xcodebuild` to help in this situation by allowing you to define an external configuration that will dictate scheme build ordering and settings.

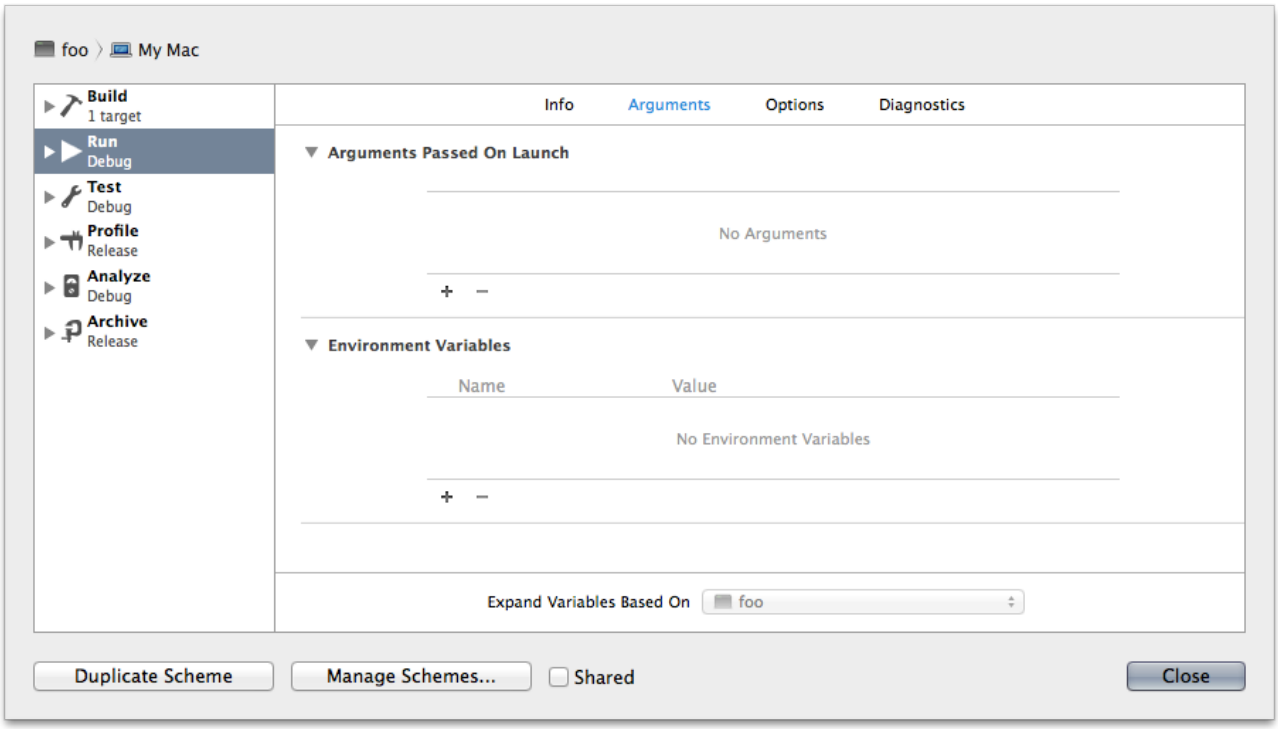
Run

The "Run" action allows you to configure how the application will launch and the environment it has.

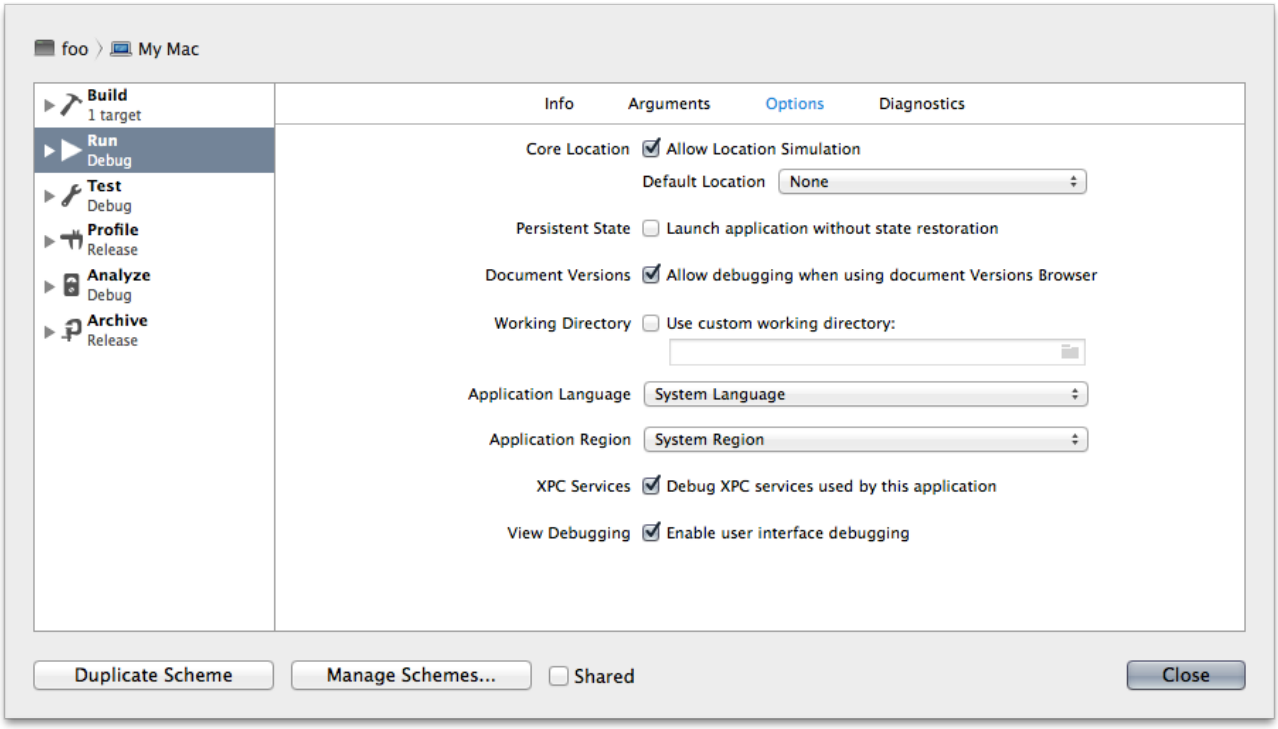
- Info: You can configure the build configuration that the scheme will use, the built product/executable, and how that will get debugged and run.



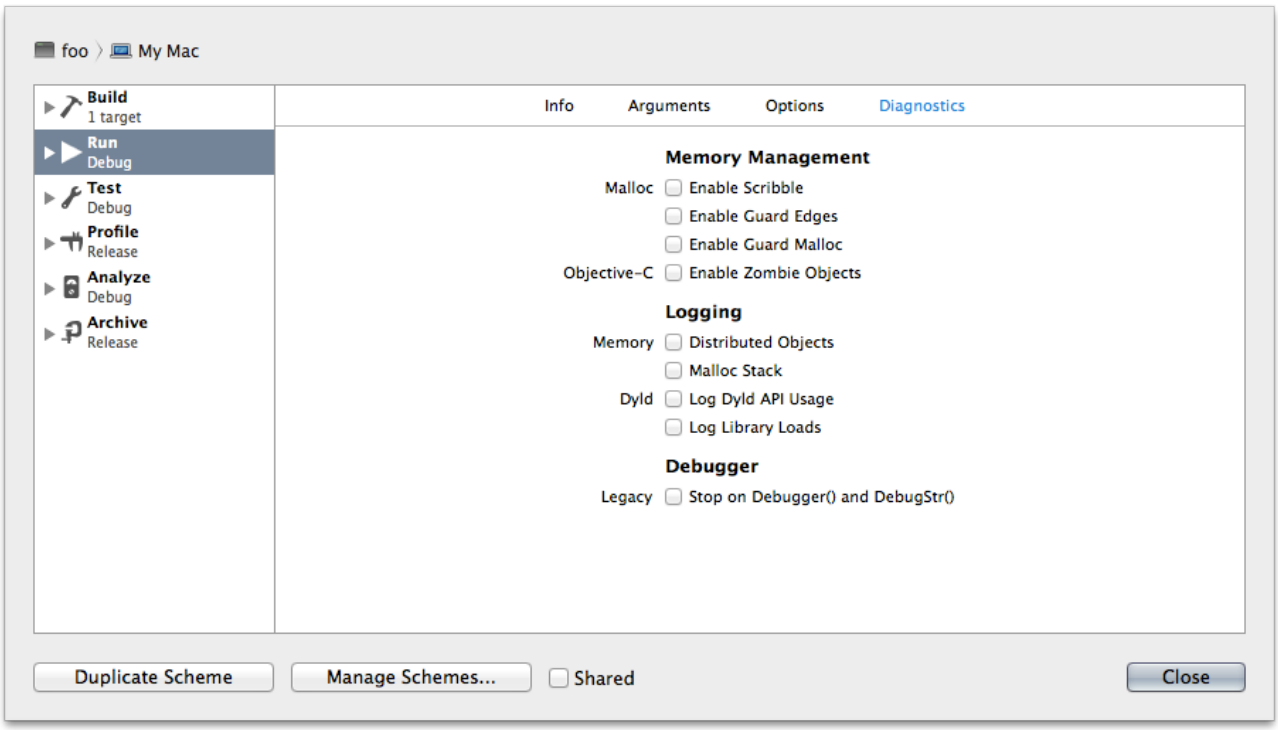
- Arguments: Use this section to configure passed launch arguments and environment variables. Existing environment variables, such as `SRCROOT` can be used here and will be expanded before being passed.



- Options: This is a set of additional options relevant to the state of running the application from Xcode.



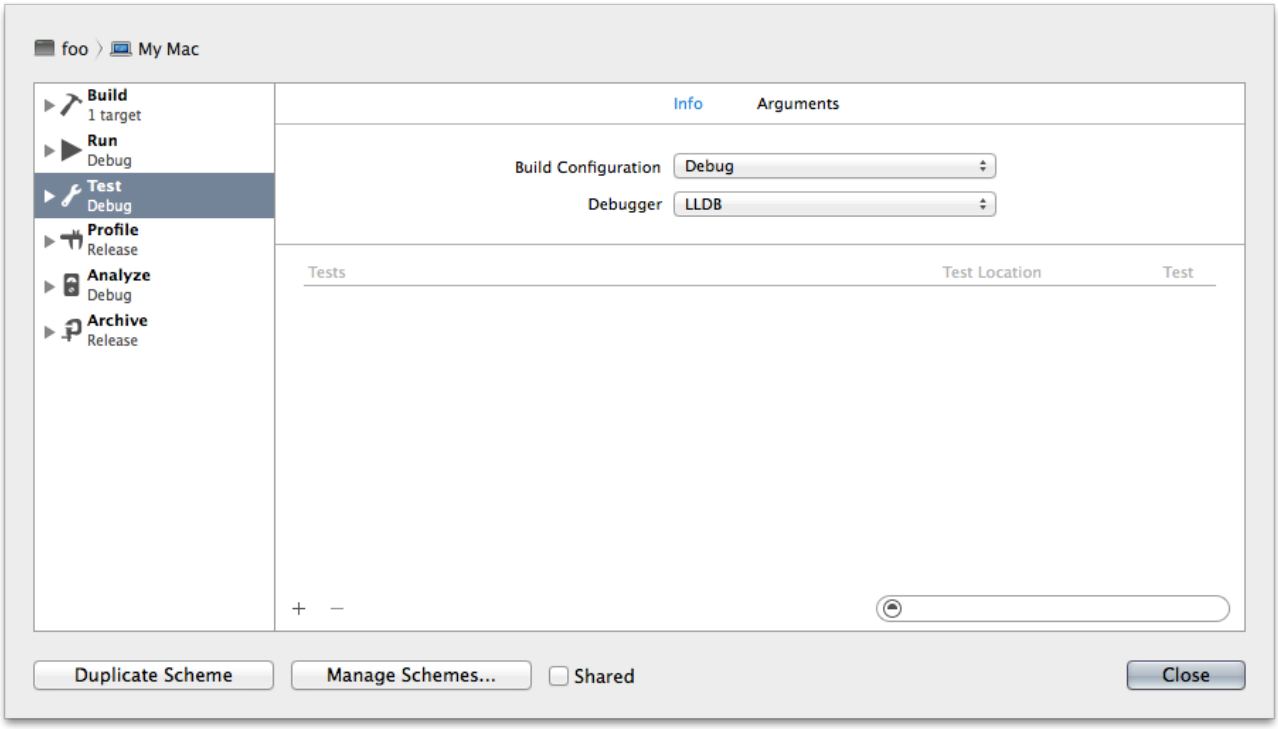
- Diagnostics: These are additional tools that can be enabled to help debug memory management issues as well as some more advanced information logging.



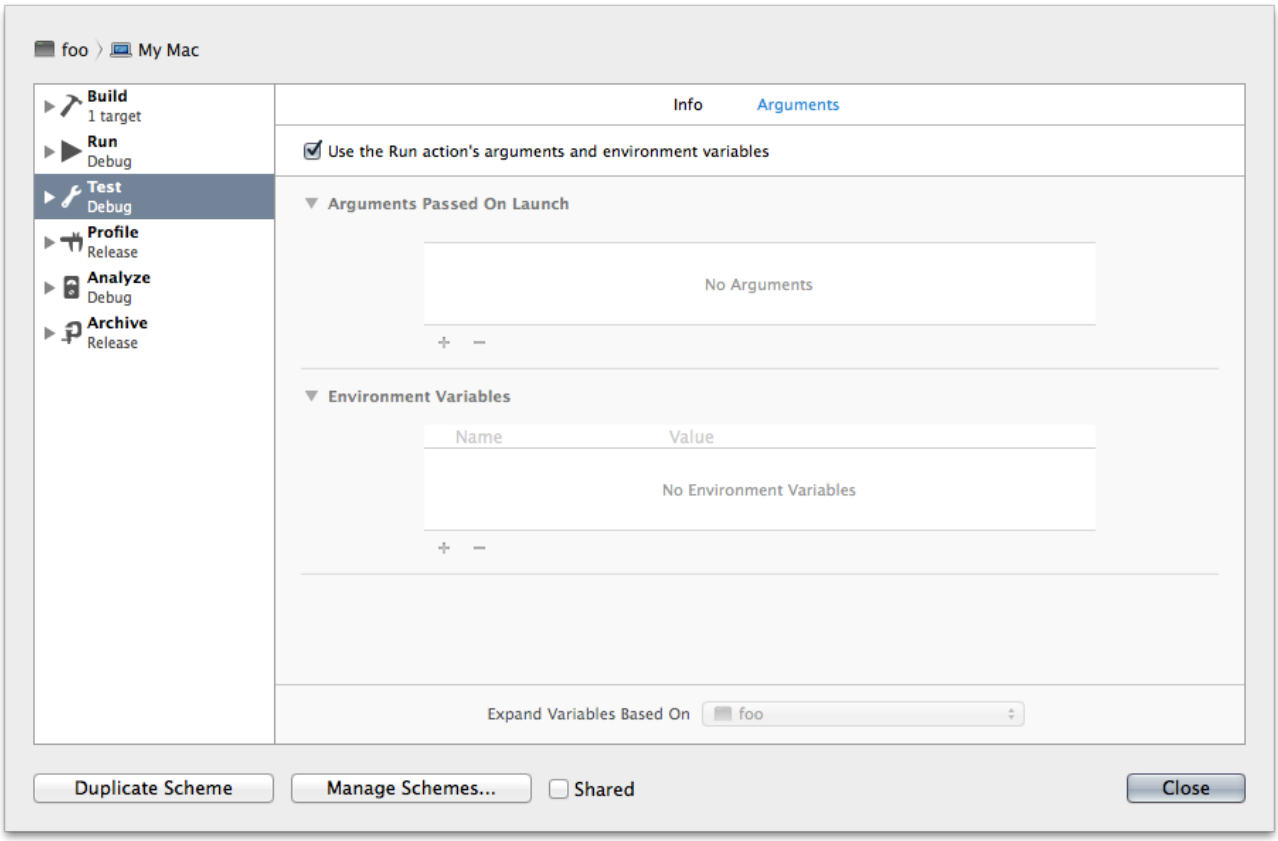
Test

The "Test" scheme action is used to run associated unit test bundles.

- Info: Selecting the build configuration and debugger used when running tests. This view provides a list of unit test targets that are associated with the scheme. This also displays the state of all the test methods in each unit test bundle. Individual test methods can be enabled or disabled from being called when running the test action on the scheme.



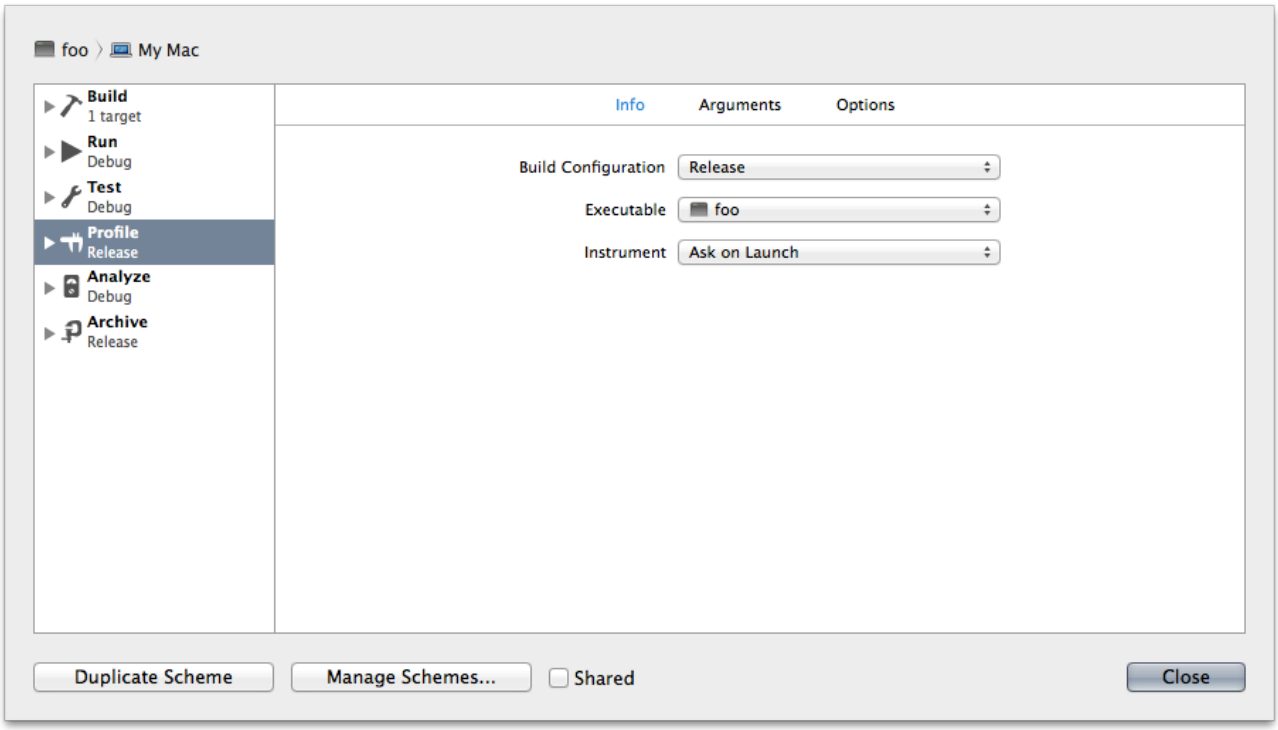
- Arguments: Similar to the "run" action, the test action can be passed specific environment variables and launch arguments. These can be inherited directly from the entries in the "run" scheme action.



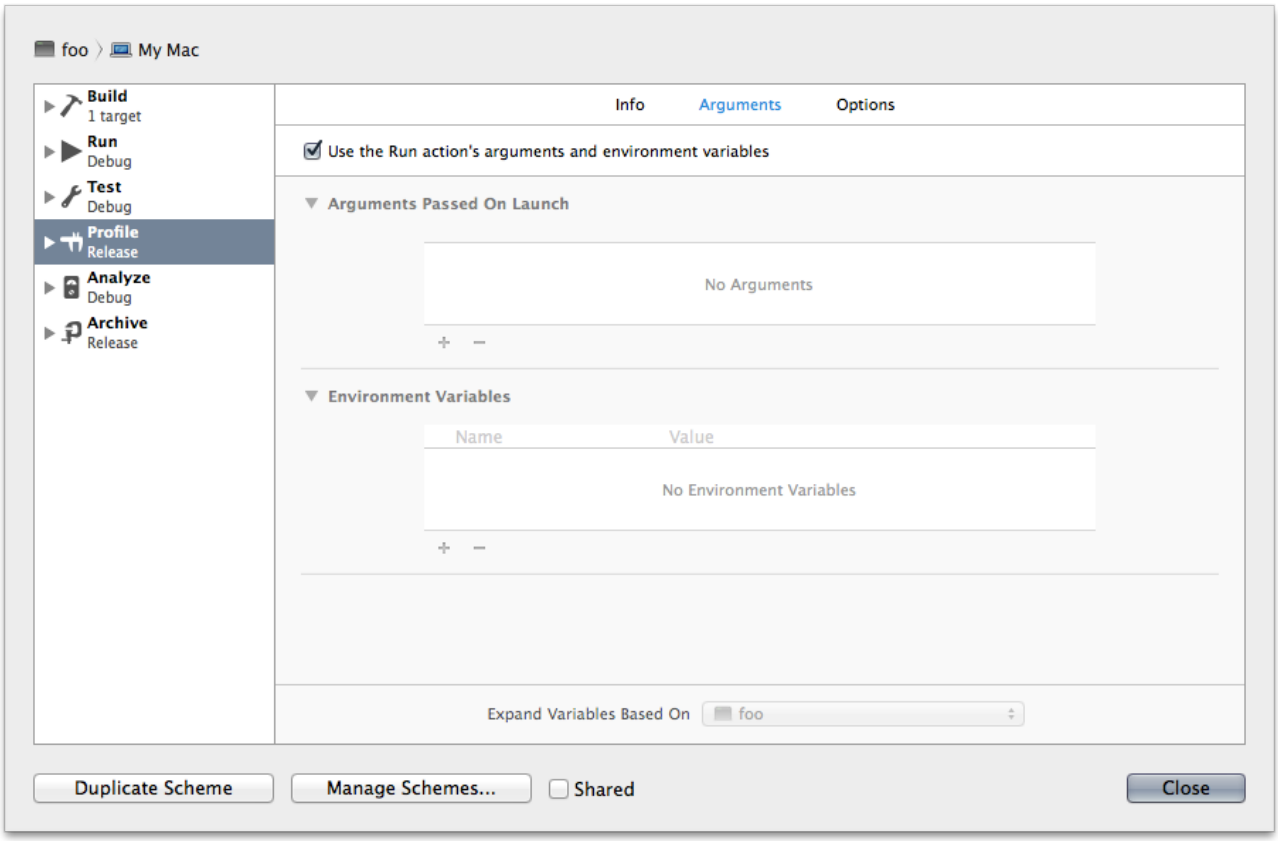
Profile

The "Profile" scheme action allows you to run an app while attached to additional debugging instrumentation. This is crucial to track down memory leaks, examine threading behavior, and make performance optimizations.

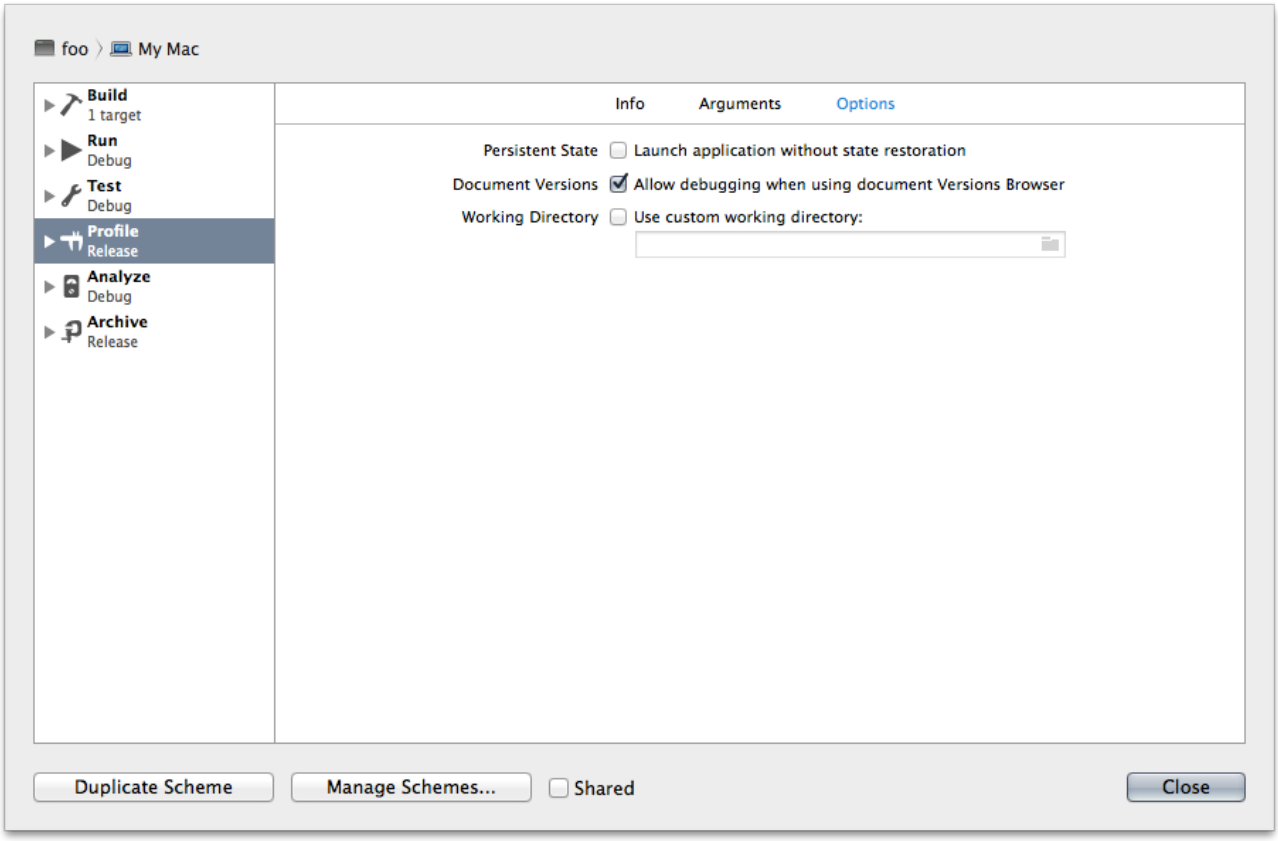
- Info: This will configure the initial behavior performed by the profile action. The build configuration, executable to attach to, and the default action that Instruments.app should take are configurable from here.



- Arguments: Similar to the "run" action, the profile action can be passed specific environment variables and launch arguments. These can be inherited directly from the entries in the "run" scheme action.

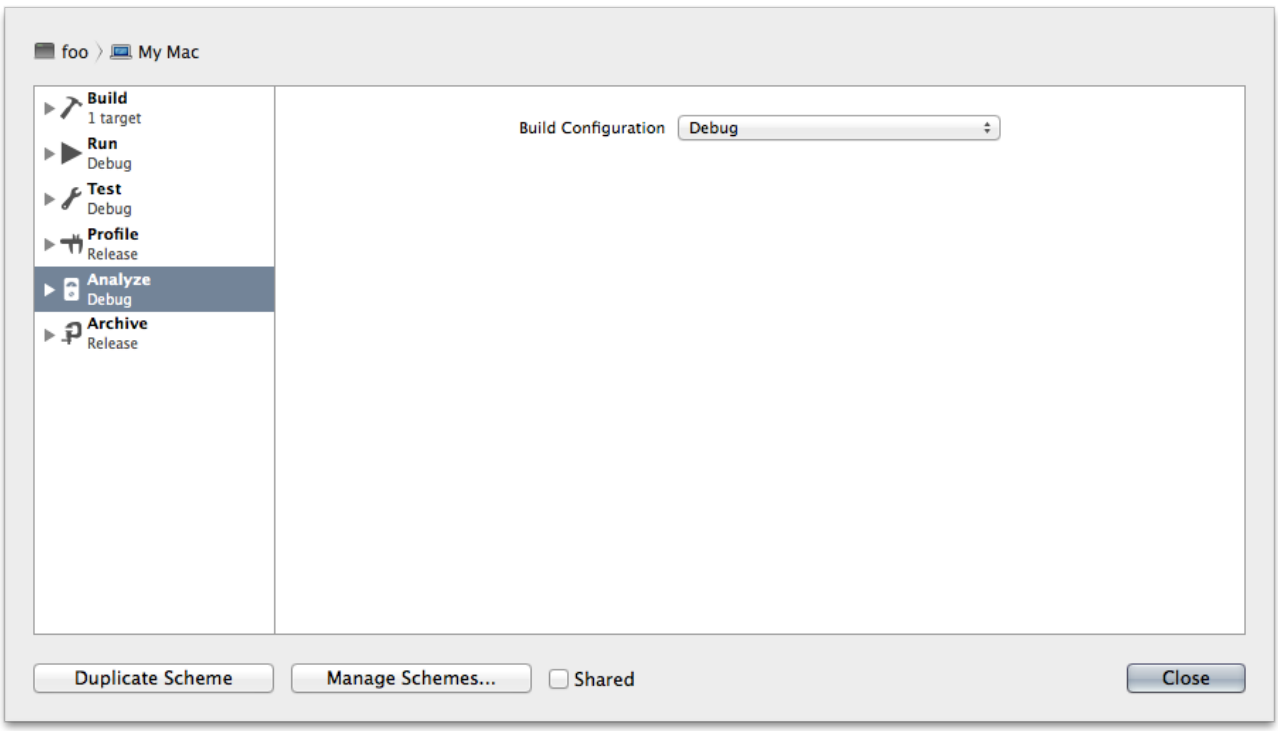


- Options: This includes some additional launch and application state options.



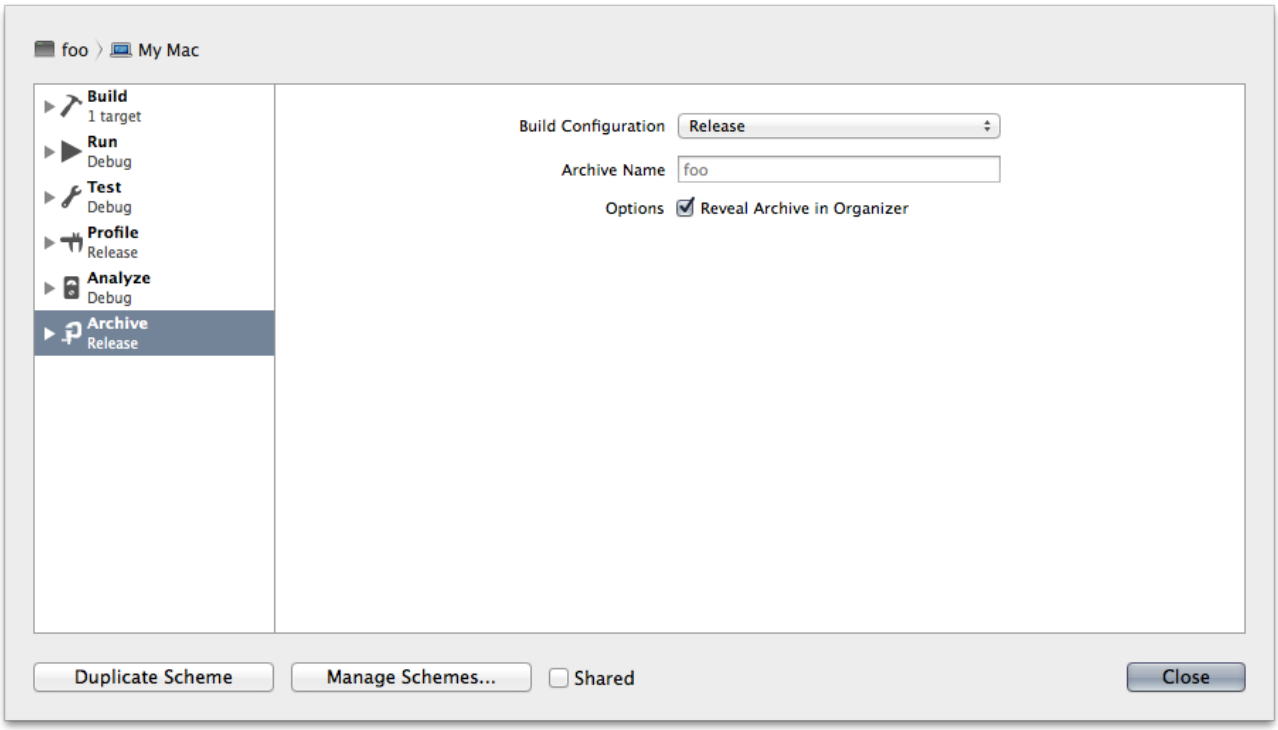
Analyze

This scheme action runs code through the llvm static analyzer. The only configurable option for this is the build configuration it uses. It is recommended to regularly run code through the analyzer, as it will find bugs and help you write cleaner code. The "analyze" action performs a deep analysis by default. By using the target's build settings, you can enable the static analyzer to be run when performing the "build" action on a scheme as well. This is disabled by default and will only perform a shallow analysis on a "build", but can be configured to perform a deep analysis if desired.



Archive

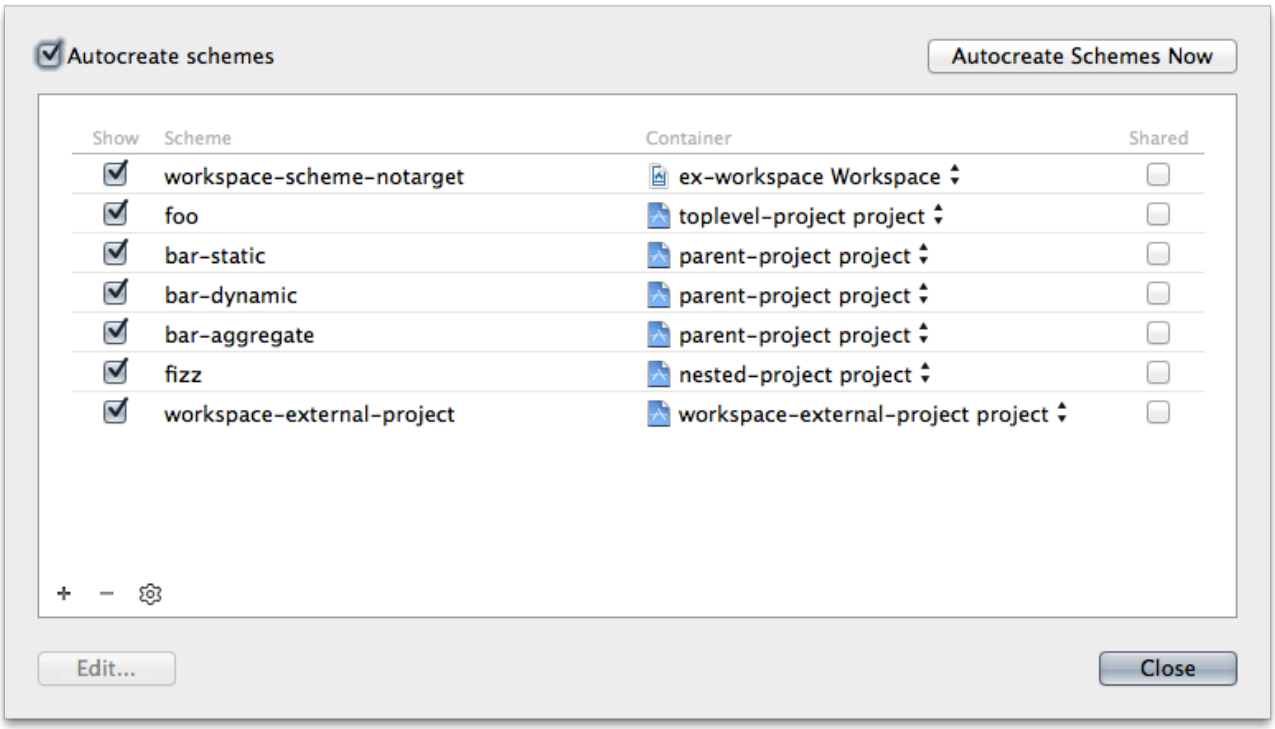
The "Archive" scheme action is used to cut builds for uploading to distribute. These get exported to `<Xcode Archives Path>/<YYYY-MM-DD>/` and are labeled by name and time of export. (See "Build Locations", for more information on the Archives path)



↑ Parent

Management

Schemes can be auto-created for each target you add to a project. In addition you can add new schemes for organize how the target schemes are run and grouped. This is similar to using aggregate targets for organizing inside of a project.



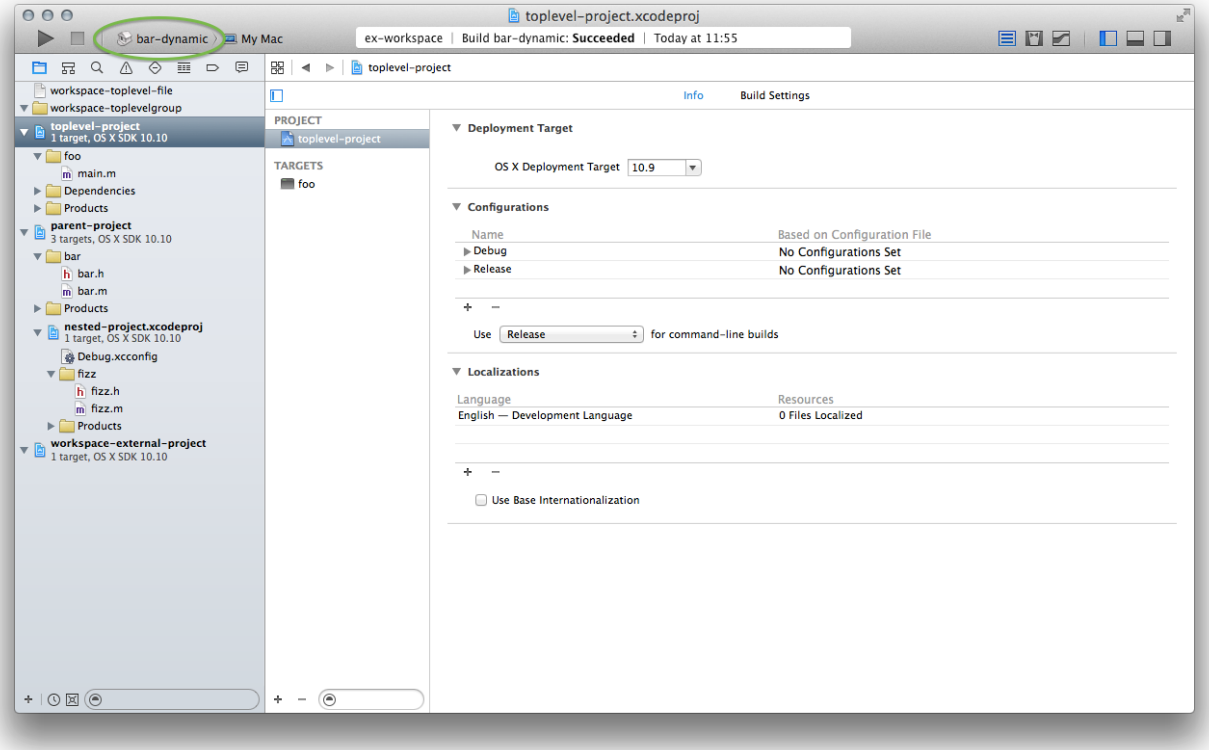
Schemes are stored in whatever container they were created in, either in a project or workspace. The manner in how they are stored is governed by if they are "shared" schemes or "user" schemes. By default new schemes are created as "user" schemes.

- User: User schemes are stored in `<.xcodeproj or .xcworkspace>/xcuserdata/$USER.xcuserdatad/xcschemes/`. These are useful when needing to configure a build around an unique case that wouldn't hold true to anyone else working on the project. Since user schemes are stored in the `xcuserdata` directory, they are often completely ignored by source control. In addition, tools such as xctool and Carthage do not use user schemes so it is important to audit schemes and tag them as "user" or "shared" as needed.

- Shared: Shared schemes are stored in `<.xcodeproj or .xcworkspace file>/xcshareddata/xcschemes/`. A shared scheme is visible to all users of a project or workspace file. Since shared schemes are stored in a common location, they are often used by external tools. Shared schemes offer the benefit of ensuring that the build process is consistent. As a result of being stored in a common location, shared schemes can be checked into source control so they are visible to all users.

To perform any build action on a specific target, a scheme must exist for that target. Xcode enables the option to auto-create schemes for targets that exist in a project or workspace but don't have a corresponding visible scheme. A scheme's visibility is determined by either being a shared scheme or by having a scheme already defined for the specific user within the workspace or project file. As mentioned, user specific schemes are not visible to users other than the one that created it. This means that most schemes in a project should be marked as shared schemes to ensure the same configurations are used when performing builds.

To make the list of schemes more manageable, there is an additional checkbox to toggle a scheme from being listed in the schemes selection drop-down menu on the Xcode window. By only making the most commonly used schemes visible in the dropdown, this can reduce clutter and make switching build configurations (e.g. Development vs Production environment builds) easier.



↑ Parent

↑ Table of Contents

Build Configuration

Build configurations are used to apply specific variations to a target's build settings. By default, projects have two configurations "Debug" and "Release". These configurations allow for fine-tune control over specific build settings and flags that should be passed or over-ride the existing settings of a target. An example of this in action would be turning off code optimizations for Debug builds, but enabling code optimization for Release builds.

Configurations

Build configurations are stored on a per-project level and allow for specifying a single additional configuration file (`.xcconfig` file) on a per target basis. A configuration is an additional layer to the build settings of a target. All targets in a project can inherit the build settings dictated on the project level, and those settings can be over-ridden on a per-target basis. Like-wise, each build configuration inherits the build settings from an individual target, but can over-ride any of them to only apply when building with that specific configuration. This can be extremely helpful when building the same set of code for different platforms. Instead of maintaining two separate build targets that only primarily build against the OS X or iOS SDKs, these can be done via different build configurations. This removes the burden of keeping the settings of both targets in sync and allows for easier management via schemes.

↑ Parent

XCConfig Files

Please see the Unofficial Guide to xcconfig files now.

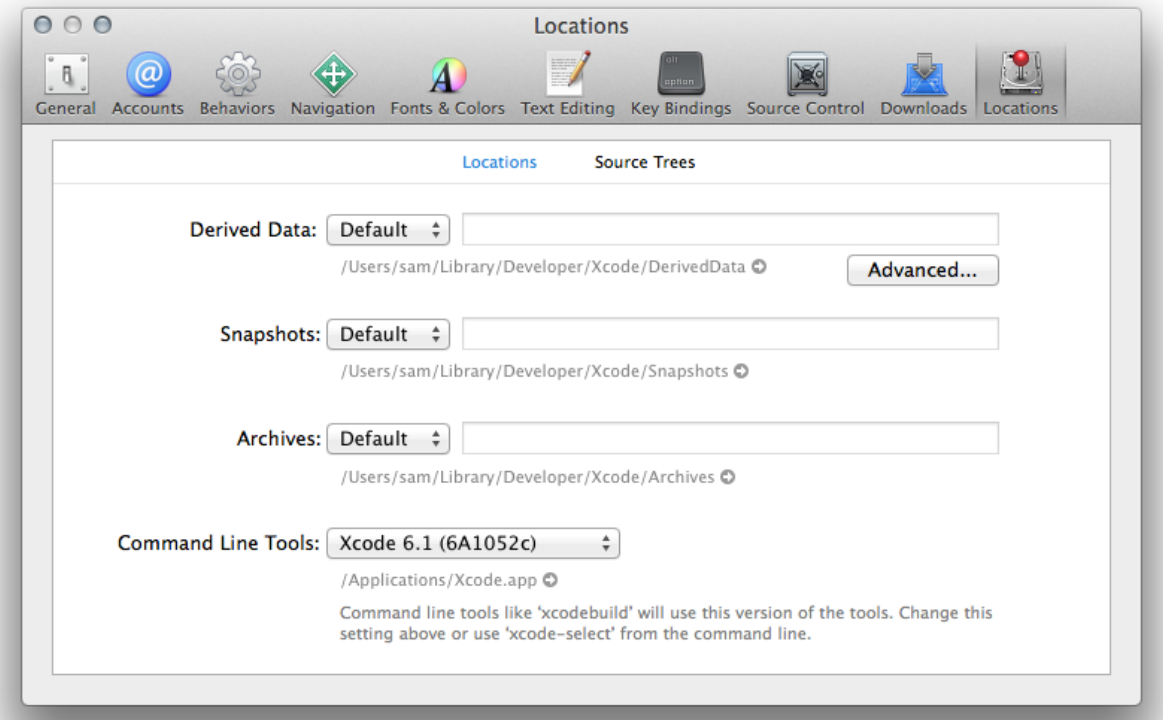
↑ Parent

↑ Table of Contents

Build Locations

> Note: All of the settings discussed in this section apply globally. All of your projects and workspaces will use these settings. Use caution and check thoroughly before changing any of these settings.

In addition to managing the organization of code inside of Xcode, you can also customize the output locations of any built products. Build locations are significant to managing the resolution of target dependencies. From this panel you can set where on the filesystem Xcode should set the default locations for the build process.



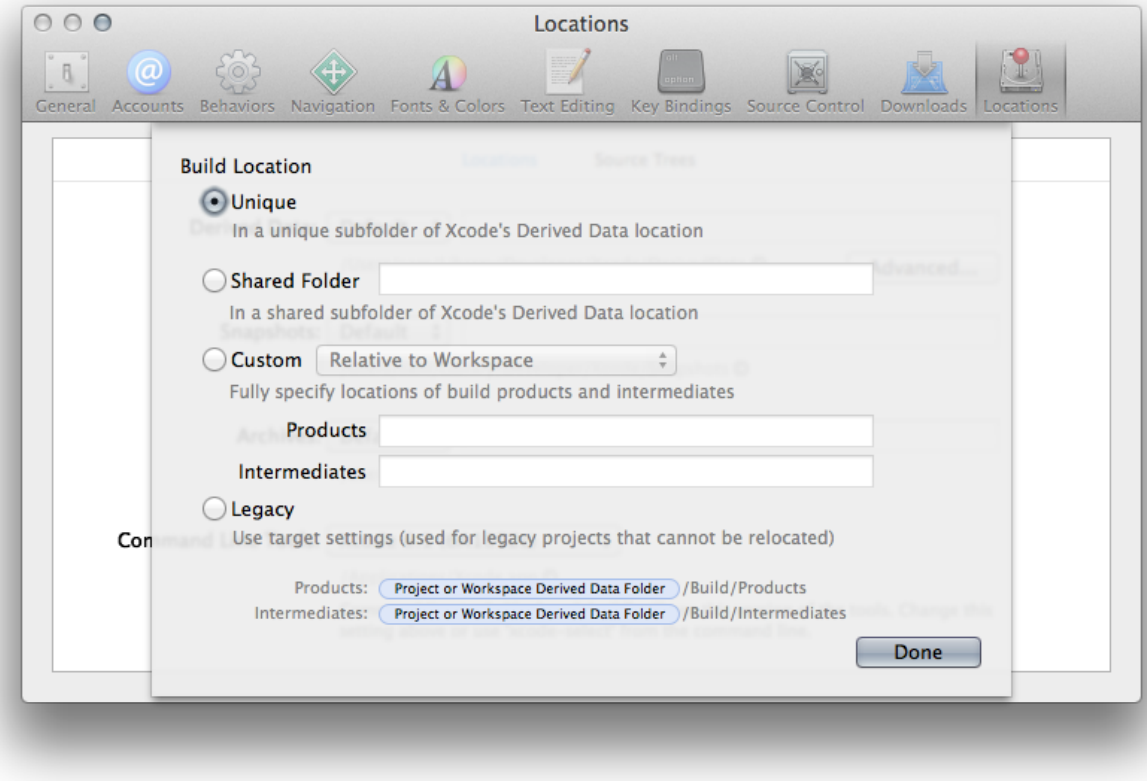
- Derived Data: The Derived Data location is, by default, the location where all the intermediate and final products of the build process are stored. This field sets the global location of where the Derived Data directory is stored. However there are a number of ways that the build locations can be configured. The settings for these options can be configured by clicking on the "Advanced..." button.
- Snapshots: The snapshots directory is for storing Xcode snapshots. Snapshots are a way of saving state when the changes being made aren't covered as part of the undo manager. Modifications made to build settings or multiple files (e.g. global find and replace) are examples of such actions.

- Archives: This is where the output from the "Archive" scheme action is stored.
- Command Line Tools: This option is the equivalent of running `xcode-select -s <[path to Xcode]/Contents/Developer>`. This changes the version of the tools accessed via the command line. The selected version does not have to match the current version of Xcode you are using, but be aware that if you use any external build tool (including `xcodebuild`) it will use this version instead of the version of the Xcode.app.

> Note: The following subsections are dedicated to the advanced location controls that are part of the **Derived Data** build location. Please note that when building products, they are placed in a directory, named after the build configuration used, inside of the build directory location.

Unique

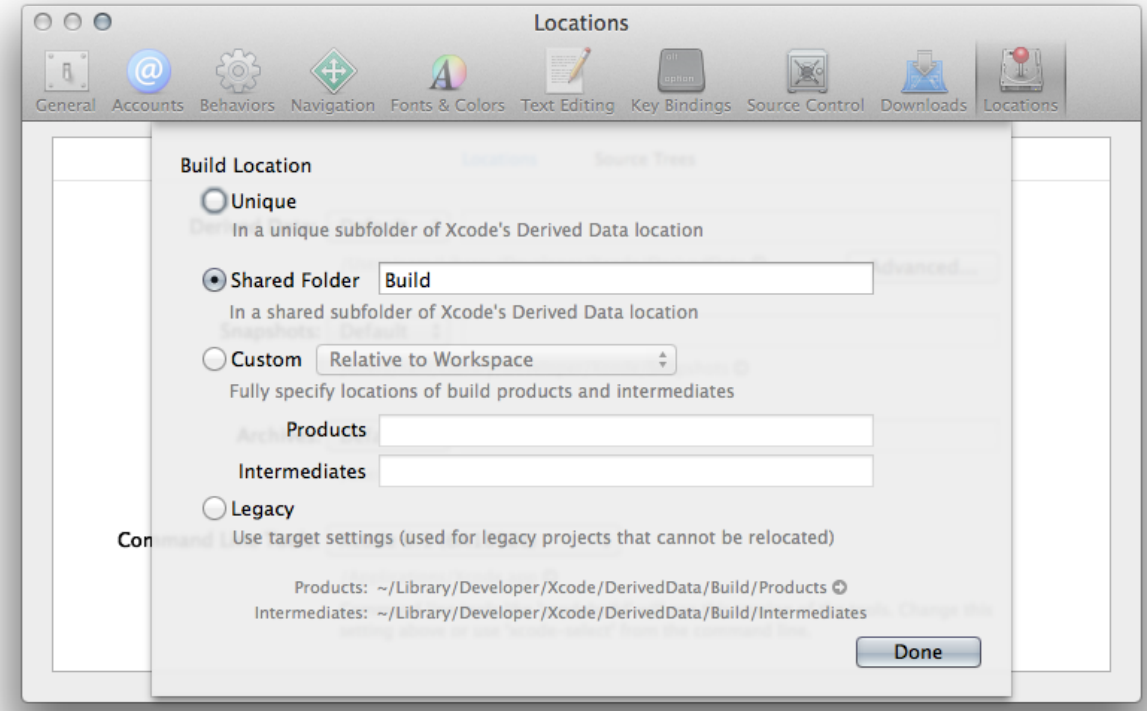
This is the default setting used by Xcode since version 4. This setting will create a unique folder based on the open project or workspace that will contain all of the built products and intermediate files needed for the build process. These folders are stored in the directory set as the DerivedData location on the main build locations pane. The unique build folder name that is created is based on the absolute path of the root project or workspace that is open. See this post for details on how the unique folder name is generated.



↑ Parent

Shared

The "shared" location setting creates a universal build directory. This is not unique to a project or workspace, so conflict can occur when using different versions of a library or application that has the same name. This shared build folder is located in a directory within the Derived Data directory.

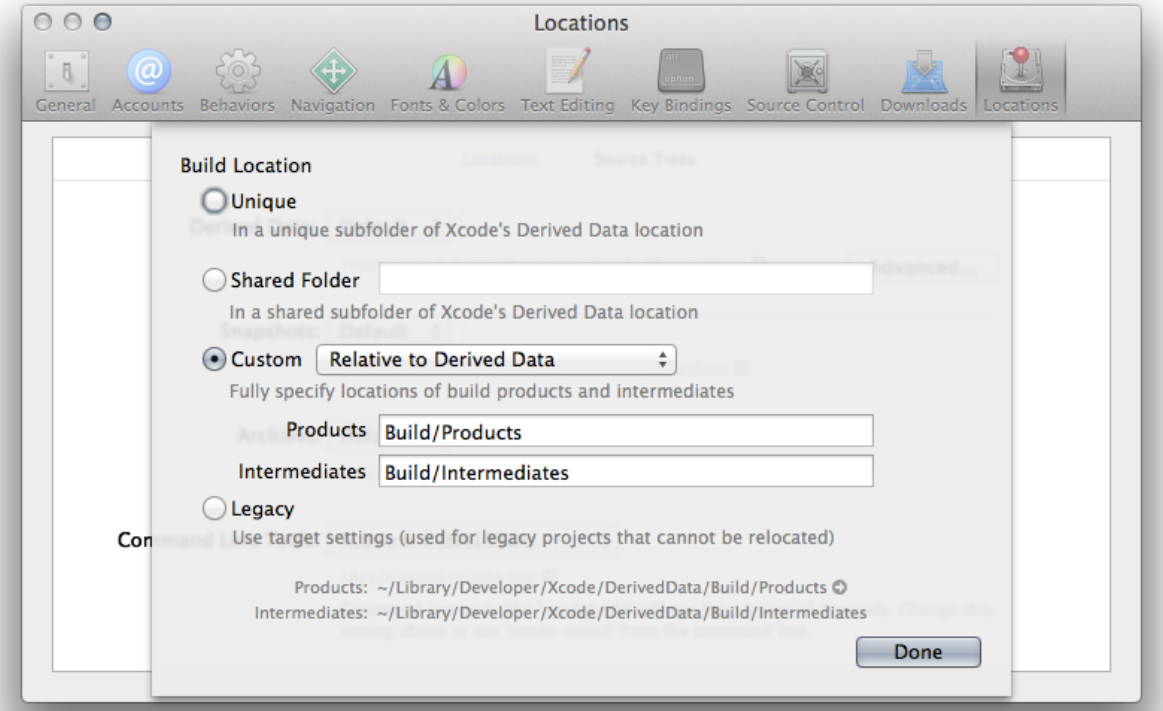


↑ Parent

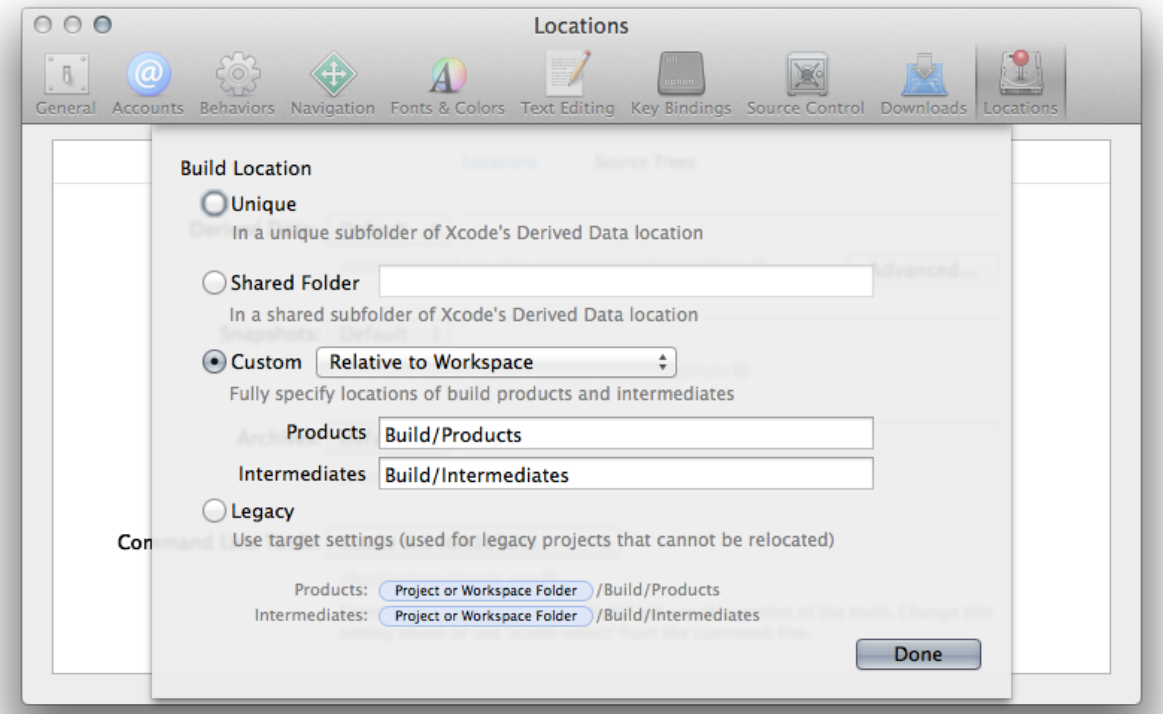
Custom

The "custom" location setting allows for some varied behavior in where build locations are set.

- Derived Data: By default, this setting uses the same paths as the "Shared Folder" build location setting. The path here is able to be set relative to the Derived Data directory path.

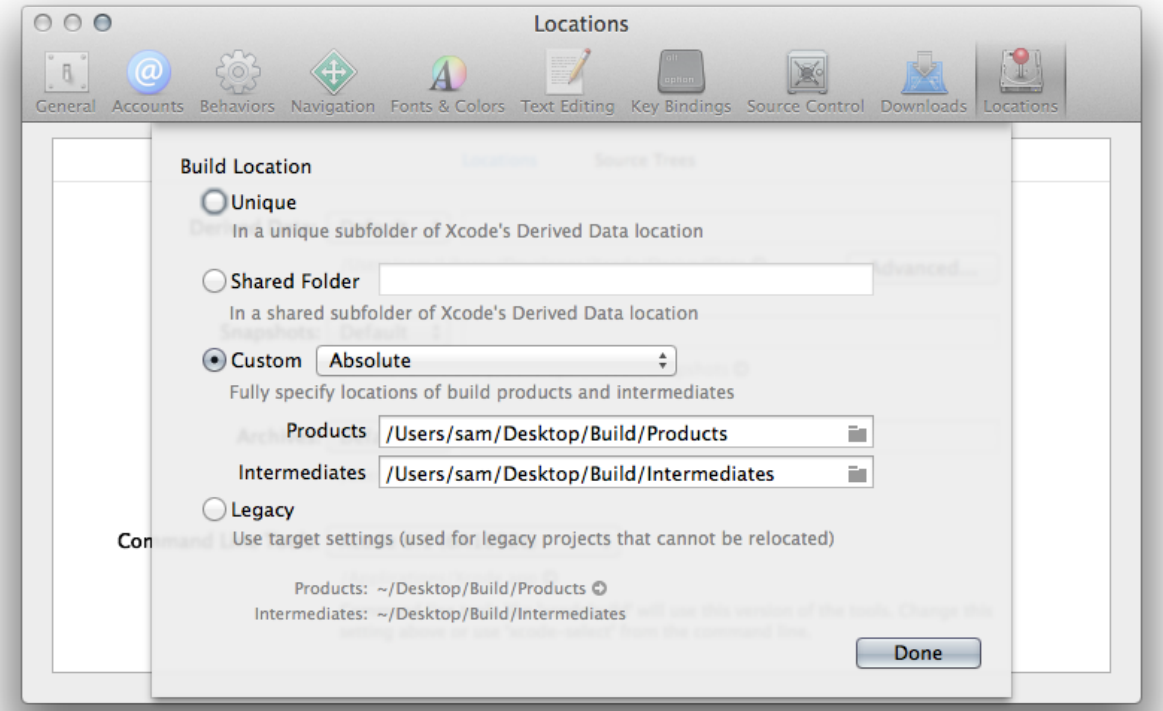


- Workspace: This setting is similar to the "Unique" setting, except the build directory is in the same directory as the open workspace or top-most project file instead of a unique folder in the Derived Data directory.



- Absolute: This setting is similar to the "Shared Folder" setting, except it is not a sub-directory of the Derived Data directory. This sets an absolute path to a location on your file-system that will house

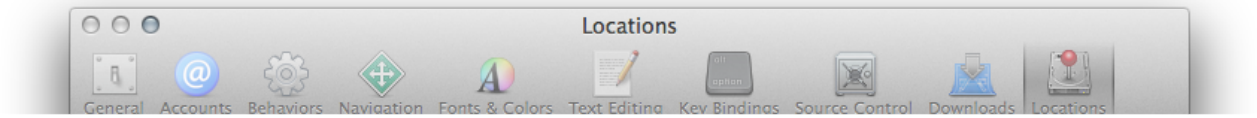
all built products and intermediates. As this is an absolute path and not a relative path, this can cause conflicts if your targets are configured assuming absolute search paths that might not exist for other users or in other environments.



↑ Parent

Legacy

The "Legacy" option, was default option for Xcode prior to version 4. This settings requires that each project have `SYMROOT` and `OBJROOT` defined as part of each target's build settings. This enables specific build directories to be set per target in each project. By default the value for `SYMROOT` is `build/`, which means a directory named "build" on the same level as the project file. When working with workspaces and nested project files, you may need to adjust this value accordingly to create a common build directory for the top-most project or workspace file by adjusting the relative location. For example, setting `SYMROOT` of a nested project to be `../build` so that it uses the same build directory as the parent project. Alternatively, you can adjust the search paths of a parent project to look in the build directory of a child project to resolve dependencies.



↑ Parent

↑ Table of Contents

Resources

- Example workspace and project files to explore
- Xcode Build Setting Reference
- Xcode Build Locations Documentation
- Xcode Concepts
- Xcode Continuous Integration
- Xcode Overview
- Managing Schemes
- Abandoning the Build Panel
- Using XCConfig files

↑ Table of Contents