Article

Writing Custom Build Scripts

Extend your Xcode Cloud workflows with custom build scripts that perform custom tasks or install additional tools.

On This Page
Overview
See Also

Overview

Xcode Cloud leverages your project's configured schemes and offers settings to create advanced workflows. However, Xcode Cloud workflows can offer even more flexibility to meet your project's requirements. For example, you might need to install an additional third-party tool to successfully build your project, upload build artifacts to private storage, use a different app icon for nightly builds, and so on.

> Note
> Xcode Cloud is a feature of Xcode 13 and currently in beta. Visit Xcode Cloud Beta to submit a request for access.

If you need additional flexibility in your Xcode Cloud workflows, create a *custom build scripts* to perform a specific task at a designated time. Xcode Cloud recognizes three different script types:

- A *post-clone script* that runs after Xcode Cloud clones your Git repository.
- A *pre-`xcodebuild` script* that runs before Xcode Cloud runs `xcodebuild`.
- A *post-`xcodebuild` script* that Xcode Cloud runs after running `xcodebuild`.

For Xcode Cloud to recognize your custom build scripts, you'll need to place them at a specific location: the `ci_scripts` directory. Additionally, name the scripts according to conventions listed below. When it starts a new build, Xcode Cloud recognizes your custom build scripts automatically and runs them for every action.

For additional information about custom build scripts, see WWDC21: Customize Your Advanced Xcode Cloud Workflows.

Create the CI Scripts Directory

Custom build scripts reside in a directory named `ci_scripts` that's located in the same directory as your Xcode project or workspace.

To create the `ci_scripts` directory:

1. Open your project or workspace in Xcode and navigate to the Project navigator.
2. In the Project navigator, Control-click your project and choose New Group to create the group and its corresponding directory.
3. Name the new group `ci_scripts`.

Create a Custom Build Script

When Xcode Cloud performs an action you've added to a workflow, it performs a series of steps. If you add a custom build script, Xcode Cloud runs it at a specific moment between these steps.

The name of a custom script's corresponding file determines when Xcode Cloud runs the script; only use the following file names for your scripts:

`ci_post_clone.sh`
: The post-clone script runs after Xcode Cloud clones your Git repository. You might use a post-clone script to install an additional tool, or to add a new entry to a property list.

`ci_pre_xcodebuild.sh`
: The pre-`xcodebuild` script runs before Xcode Cloud runs the `xcodebuild` command. You might use a pre-`xcodebuild` script to compile additional dependencies.

`ci_post_xcodebuild.sh`
: The post-`xcodebuild` script runs after Xcode Cloud runs the `xcodebuild` command — even if the `xcodebuild` command fails. You might use a post-`xcodebuild` script to upload artifacts to storage or another service.

To create a custom build script:

1. Open your project or workspace in Xcode and navigate to the Project navigator.
2. Control-click the `ci_scripts` group you created earlier and choose New File.
3. Choose the Shell Script template.
4. Name the shell script `ci_post_clone.sh`, `ci_pre_xcodebuild.sh`, or `ci_post_xcodebuild.sh`.
5. Create the file without adding it to a target.
6. In Terminal, make the shell script an executable by running `chmod +x ci_post_clone.sh`.
7. Add code to your custom build script and add it to your Git repository. Xcode Cloud runs it automatically when it starts the next build.

> Note
> You can't obtain administrator privileges by using `sudo` in your custom build scripts.

Add Resources to the CI Scripts Directory

Custom build scripts run in a temporary build environment where your source code may not be available. As a result, you need to place all resources your custom scripts access in the `ci_scripts` directory. For example, place artwork or `.plist` files in the directory.

> Note
> Use sub-directories inside the `ci_scripts` directory to organize its content. However, make sure to place the three custom build scripts `ci_post_clone.sh`, `ci_pre_xcodebuild.sh`, and `ci_post_xcodebuild.sh` at the top level of the `ci_scripts` directory.

In some cases, a custom build script needs to access to a file that's located in the repository, but placing it in the `ci_scripts` directory isn't practical. In this case, create a symbolic link to the file in the `ci_scripts` folder. Xcode Cloud detects the symbolic link and makes it available within the `ci_scripts` folder in a subsequent phase of the running action.

Access Environment Variables

Environment variables are key to custom scripts because they allow you to write flexible custom scripts with advanced control flows. For example, the following code snippet checks if the `CI_PULL_REQUEST_NUMBER` variable is present to only run a command when Xcode Cloud runs the script as part of a build from a pull request:

```
#!/bin/sh

if [[ -n $CI_PULL_REQUEST_NUMBER ]];
then
        echo "This build started from a pull request."

        # Perform an action only if the build starts from a pull request.
fi
```

For a list of predefined environment variables, see Environment Variable Reference.

Define Custom Environment Variables

In addition to predefined environment variables, you can define custom environment variables in a workflow's Environment section. To learn more about configuring custom environment variables, see Custom Environment Variables

Add Debug Information

The output of your custom build scripts appears in the build report's build logs. Log information that may be helpful when you debug a custom build script. However, never log sensitive information like API keys or access tokens in your shell script output unless you use a secret custom environment variable. If you mark a custom environment variable to be a secret, Xcode Cloud replaces its value with asterisks (`**********`) in build logs.

Write Resilient Scripts

Custom build scripts can perform critical tasks like installing dependencies or uploading build artifacts to storage. Write resilient code that handles errors gracefully, and, if a command fails, return a non-zero exit code. By returning a non-zero exit code in your custom build script, you let Xcode Cloud know that something went wrong and let it fail the build to let you know that there's an issue.

The following build script sets the `-e` option to stop the script if a command exits with a non-zero exit code. Additionally, return a non-zero exit code in case something goes wrong:

```
#!/bin/sh

# Set the -e flag to stop running the script in case a command returns
# a non-zero exit code.
set -e

# A command or script succeeded.
echo "A command or script was successful."
exit(0)

...

# Something went wrong.
echo "Something went wrong. Include helpful information here."
exit(1)
```

Use Helper Scripts in Complex Custom Build Scripts

If your repository contains more than one project, you'll likely need to configure several Xcode Cloud workflows. Because Xcode Cloud only recognizes one `ci_scripts` directory in your repository — and that directory can only contain the three custom build *scripts* — you'll need to add *helper scripts*. Helper scripts are shell scripts you place in the `ci_scripts` directory that you can use to split up the tasks of the default build scripts.

You can use the helper scripts to create a more flexible build environment, where you offload most of the logic you want Xcode Cloud to perform in a custom build script into a separate shell script file. For example, you might use environment variables to determine which Xcode Cloud workflow runs the build script, what started the build, the platform of the running build, and so on, to run logic only when applicable.

When it comes to naming your helper scripts, use a filename that makes it easy to recognize the script's purpose. For example, you might prefix helper scripts with the name of their intended platform and name the following script `platform-detect.sh` because it uses the `CI_PRODUCT_PLATFORM` variable to detect the platform for the current build:

```
#!/bin/sh

if [ CI_PRODUCT_PLATFORM = 'macOS' ]
then
        ./macos_perform_example_task.sh
else
        ./iOS_perform_example_task.sh
fi
```

To help make your helper scripts more recognizable, you should avoid prefixing the script names with `ci_`.

See Also

Custom Build Scripts

Environment Variable Reference
Review predefined environment variables you use in custom build scripts.