

Automating your CI/CD mobile workflow with GitHub Actions: An Introduction

Author: Vincent Frascello, Senior Mobile Developer, REA Group

Introduction

Any modern development team today is using Continuous Integration and/or Continuous Delivery in their engineering practices. CI/CD ensures software organizations can stay true to some of the most important software design principles: revision control, code checking, and automated testing and deployment. By using automation to help test and manage version control, developers can spend more time focusing on features and fixing bugs. In the early days of automation, everything was made-to-order and entire teams in organizations had to spend a lot of time and effort maintaining automation systems. Over the past decade, the transition to more efficient and effective automation and CI/CD systems has been a huge focus of leading software organizations all over the world. Jenkins, Travis CI, Terraform, Circle CI and many other solutions have been maturing and have greatly improved our automation processes. However for a lot of software professionals, these automation tools are either a black box that is managed by someone else at their organization, or scary and complex software integrations that are best to be avoided whenever possible. Therefore, simplicity and understandability are key factors to consider when working with automation.

In the past couple years, a new offering has been made available by Microsoft's GitHub - Github Actions. Actions iterates on the mission and vision of CI/CD and takes advantage of vertical integration to provide new and exciting opportunities to streamline your CI/CD and automation processes even further. I think it's much simpler to use, easier to understand, and quicker and cheaper to implement - But what does GitHub Actions really provide, and how can you benefit from it?

This article should be general enough to help anyone who is interested in learning about Github Actions. Perhaps you've never set up a continuous integration pipeline and you're looking at Actions as a potential choice; or maybe you or your organization have a mature CI/CD pipeline but are frustrated or limited by certain aspects and are looking to explore a change. No matter who you are or where you and your organization are on your CI/CD journey, hopefully this article will help you understand the basics. The associated tutorial is designed for

someone who has never used automated CI/CD, but understands the basics of building, testing, and deploying mobile applications.

Why Github Actions?

Pros / Benefits

It's simple, and it's fast.

Just to be frank, Github Actions are quite easy to set up and maintain. Simply include your actions in a YAML file in a specific folder, push up to GitHub, and you're ready to go. Spend a little bit of time to set-up your self-hosted runner (details on how later) and you have control of your runner as well. All your runs are containerized so maintaining the environment state is taken care of for you. When I first transitioned our project to actions, I had the majority of our CI/CD pipelines running in less than a day. Furthermore, I've saved over 50% computation time running using Actions compared to Jenkins.

Low to No Maintenance.

To speak a bit more about maintenance, managed runners are all run in dockerized containers so no maintenance is required as they should be spun up the same each time. You can install and update software as needed as part of your action workflow, if needed. If you self-host, you simply need to maintain the node for things like security updates and version updates (and you could use actions to do that as well, via cron jobs).

It's free and open-source.

The actions themselves as well as the runner software is all open-source. Some paid or enterprise actions from the marketplace may be closed source, but that's typical and to be expected. Furthermore, most actions are written in standardized languages like Typescript, Javascript, Go, and Dockerfile. This makes it quite easy to develop your own actions, or to fork and use others.

The community is engaged.

In 2019 there were a lot of issues related to actions, and many have been resolved. The speed at which new features and actions are being created and improved is directly related to the previous point: the community is engaged (and of course, Microsoft is investing heavily to be the leader in this space). Documentation is getting better day after day, and the number of questions on Stack Overflow about `github-actions` is about to top `jenkins`.

It's not all or nothing.

Assuming you're already using GitHub, you can start working in actions slowly, without abandoning your current pipeline. On my projects, I first transitioned our Build and Test pipeline, and once I was happy with it, I moved over our dependency updater. For a while I had both our Jenkins and Actions pipelines both active, and once I was happy with results from Actions, I disabled Jenkins.

Automate the Automation

Actions live within repositories, so just like your code, your actions are version controlled, and can be checked, peer reviewed, and even the actions can be automated. No longer are your CI/CD configurations dependent on one specialist who may be on vacation or have left the company: everything is right there in your repository - you just need to train your team on the basics.

Cons / Drawbacks

The Ecosystem.

The biggest drawback is a quite obvious one: You must be using Github (Although Microsoft does offer Github Actions for Azure, which contains many of the standard actions). If you use another version control system, you would have to hack together a solution to bridge between your VCS and Actions. I've seen it done with GitLab, but the whole point of using Actions (in my opinion) is simplicity. So if you're on another VCS you may want to consider the cost/benefit ratio of developing a bridging solution to use Actions, or switching your VCS, or sticking with your current solution.

Personally, I love GitHub and have no plans to move away from it, so I don't think there is a big drawback to investing energy into the Microsoft/GitHub ecosystem, but I definitely recommend discussing your future plans with your engineering team.

It costs money.

Wait, you said it was free?

Well yes, it is free, for public repositories and for self-hosted repositories. If you want to use GitHub's managed servers, you gotta pay up! That being said, GitHub's managed runners are pretty affordable, although there is a 10x multiplier on cost for MacOS hosted runners. You can see GitHub's [current billing rates here](#). Most organizations will opt to host their own MacOS runners on-site or on cloud servers, so they don't have to pay GitHub to operate the MacOS runners, and the cost for linux runners is pretty minimal for our use case.

Not everything is included.

A good example is caching. When Github Actions launched, there was no solution for caching of dependencies and build outputs - and there still isn't out of the box.. However, GitHub's team developed a custom action for it on the marketplace, which is now one of the most popular actions in use. Although Actions is still relatively new and may be missing some features, the marketplace likely will contain the solution you are looking for. If you have some very particular scenarios then you'll be more likely to have to build them yourself, but you can always start from a fork of something similar on the marketplace. For more in depth of complex actions, there are GitHub Apps that can be purchased (usually on a subscription basis).

It's (sorta) new.

There is always a risk in making a large change to your processes with a newer solution. GitHub Actions is absolutely production ready, but like every software solution there are always gaps to address. However, I believe these gaps are quickly filled by how active the community is. The documentation is good and getting better each day, new features and being discussed and added weekly. I do not see any significant changes coming to GitHub Actions that would impact its usefulness. I would expect that someday, billing rates would increase, but that can be mitigated by self-hosting.

Why switch from Jenkins?

To put it simply, if you already using GitHub, you already have actions for free (if you stay within the allotted minutes for your organization, or self-host). You can avoid spending considerable time maintaining Jenkins servers, fixing buggy plugins, and dealing with plugins going out of support. In early 2022 I began testing Actions on some internal projects and was pretty happy with the results. My first Action workflows were able to build, test, and deploy our applications in less than half of the time as Jenkins on the same hardware. The workflows were easy to write, simple to understand, and easy to port to other projects. Once it became clear that Actions was superior to Jenkins (for our use case), it was an obvious choice to transition from an organizational standpoint.

Since my transition I have only improved our automation and have felt more confident that I have made the right choice.

GitHub Actions: The Basics

In a nutshell, Actions are declarative step-wise workflows that can access pretty much any API input or output on github (or other APIs with the use of cross-platform API integrations).

Actions are made of workflows that live in a special place in your repository: `.github/workflows/`. Here you can create any number of workflows by defining them in YAML/YML files. Syntax is easy to understand, and as soon as your workflows are pushed up to GitHub, they are available to run (depending on the event trigger, some actions must be merged into the default branch before they are active and visible).

Actions are enabled by default on every repository.

Actions are powerful. It's not just about CI/CD of applications, although that's the focus of this article. Because almost any events in your repository can trigger a workflow, you can do a lot of interesting things.

For example,

- Build, Test, and Deploy your app nightly, or whenever a PR is opened.
- Synchronize code between cloud services.
- Scan your code for known security vulnerabilities.
- Automatically format and lint your code.
- Generate Snapshot UI testing reports and upload for your QA team to review.
- Spin up or tear down cloud infrastructure.
- Turn on the espresso machine every time a PR review is requested.

GitHub handles the **how**. You just tell it **what** to do and **where** to do it.

In this article I'll show you a simplified example on how to implement GitHub Actions for Building and Testing of iOS and Android mobile applications. The applications themselves will

be simple as the content of the application doesn't really matter to explain how Actions works (although more complex building processes can be incorporated into your action).

So let's dig into Actions! Feel free to jump to either the Xcode or Android section below. You can also just read along: actions are easy enough to understand, you don't really have to code-along.

Prerequisites:

A GitHub Account.

A Project, preferably with test targets. (You can use our sample projects, linked below).

Note: I'll be explaining how to write these workflows from a practical point of view; for an in-depth understanding of all the details and specifics of GitHub Actions, please [read the documentation](#) - it's really good!

Xcode Build and Test Tutorial

Introduction

In this tutorial I'm going to show you how to set up a build and test action for your CI/CD workflow. The majority of software organizations follow a pull request approval and merging strategy. New code will be pushed from a working branch, and it will be checked by CI to make sure it compiles and all the tests pass, before human review and merging. (Some organizations follow a forking workflow, but the same concepts can be applied using forks instead of branches). It's important that every time new code is pushed to the branch, our automation workflow checks that code for errors and failed tests. This saves us a lot of time and prevents bugs from entering production.

Before continuing, make sure you have an Xcode project that compiles without errors locally, and contains at least one test target with at least one test.

In our example project we have created a simple SwiftUI app called ActionsDemo. ActionsDemo is a single page app that converts Celsius to Fahrenheit; there is one conversion function and a test target to make sure the function is correct. Just enough code to make a buildable and testable app.

You can find the example Xcode project at my public github - github.com/vfrascello

Creating the Build And Test workflow

The first thing you are going to want to do is create a new workflow file. While this can actually be done using a template on the GitHub UI under the Actions tab, we are going to do it manually.

1. Navigate to your project repository folder locally (Finder or Terminal)
2. Create a new special folder called `.github` (If you are on a Mac and using finder, you may want to show system folders: 'CMD+SHIFT+'.)
3. Create another folder beneath this one called `workflows`
4. Open your favorite text editor and create a new file called `buildandtest.yml`
5. Paste in the following code as a starting template:

Note: If your repository is private and you want to avoid using your allotted minutes, change runs-on: macos-latest to self-hosted and follow the instructions at the end of the article on how to self-host.

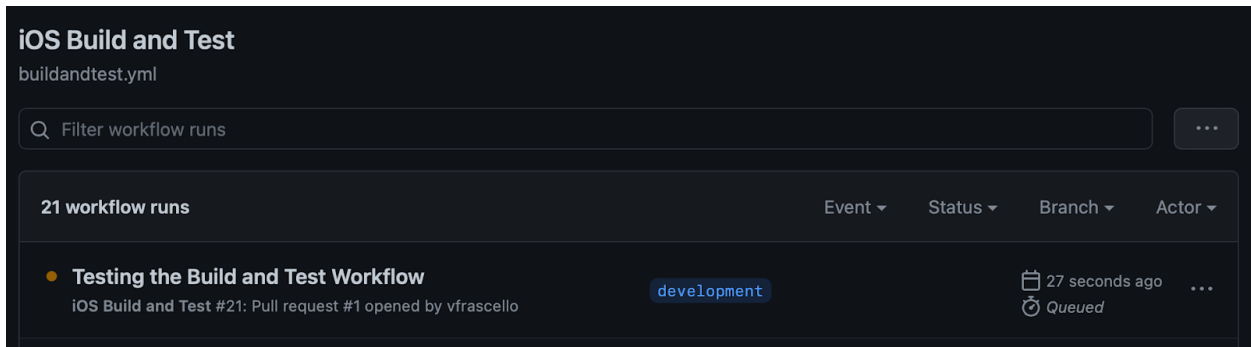
```
name: iOS Build and Test

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    name: Build and Test
    runs-on: macos-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Hello World
        run: |
          echo "Hello World!"
```

6. Save and commit this file and push it to your remote repository.
7. Now open your repository on GitHub and navigate to the Actions tab. You should see this workflow listed on the left side, and it may be in the process of running (or even completed) by the time you've navigated here.



This workflow is pretty intuitive and you probably can already tell how it works, but let's break it down simply.

The first line is the name of the workflow action.

```
name: iOS Build and Test
```

The second block of code is the `on:` block.

```
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

This is the **when** of the workflow: run this action every time something is pushed to `main` or a pull request is opened on `main` (either the merging or base branch). You can modify this syntax to be very specific about what you want:

```
pull_request:
  branches: [ "main", "development", "staging" ]
```

Will run this workflow on the associated branches. You can also use `branches: "**"` For wildcarding, and you can either use array syntax or list them out sequentially with a dash.

Workflows are written in standard YAML syntax, so familiarize yourself with it and you'll have no problem.

Next you have the `jobs:` block. You can chain multiple jobs together in one workflow, or you can reference jobs in other workflows. A `job` consists of `steps`.

```
jobs:
```



```
build:
  name: Build and Test
  runs-on: macos-13
```

The first (and only) job in this example is referred to as `build` and its descriptor follows in the `name` field. `runs-on: macos-latest` tells us **where** to run this action. GitHub provides MacOS, Windows, and Linux [runner images](#), with various options for architecture and versions. They come with a lot of pre-installed software but not necessarily everything you need, so if you are using managed runners, you may need a setup or configuration step in your workflow. `self-hosted` is the descriptor for your own self-hosted runner, which you install and configure to the repository beforehand.

Each job consists of a number of `steps`. In our example, we have two steps.

```
steps:
  - name: Checkout
    uses: actions/checkout@v3
  - name: Hello World
    run: |
      echo "Hello World!"
```

The first references another open source action built by the GitHub team, called `actions/checkout@v3`.

This does exactly what you would expect: it loads an action from another user (in this case, GitHub's actions organization), which checks out the repository on the local instance of the machine running this job. If you're using GitHub hosted runners, they will spin up a containerized docker image for each workflow, so you need to checkout the repository each time (and obviously, you've been updating it, so you want the new code to be checked out). If you're self-hosting you won't have this fresh instance each time, but each workflow is still segregated from previous workflows. Most CI workflows are going to start with a checkout step.

The second action is an example of running a shell script. This one uses `run: |` which executes the following code in the default shell. This simply prints `Hello World!` out to the console.

Your first action should have run by now, and if you click on the workflow run, you'll see each step was executed. If you expand "Hello World", you should see the following:

```
✓ Hello World 0s
1 ▼ Run echo "Hello World!"
2   echo "Hello World!"
3   shell: /bin/bash -e {0}
4   Hello World!
```

Congratulations! You've successfully implemented GitHub Actions. Now, let's go ahead and update our workflow to actually do what we came here to do: build and test an iOS application.

8. Update your YAML file as so:

```
name: iOS Build and Test

on:
  pull_request:
    branches:
      - '**'

  paths-ignore:
    - 'README.md'
    - 'documentation/**'
    - '.github/workflows/**'
    - '!.github/workflows/buildandtest.yml'

concurrency:
  # Ensure only the latest build of this PR runs.
  group: ${{ github.ref_name }}
  # Cancel any inflight builds when a duplicate is detected.
  cancel-in-progress: true

jobs:
  build:
    name: Build and Test
    timeout-minutes: 15
    runs-on: macos-13
    env:
      scheme: ${{ ActionsDemo }}
      platform: ${{ 'iOS Simulator' }}
      project_type: ${{ 'project' }}
      project_name: ${{ 'ActionsDemo.xcodeproj' }}
      device: ${{ 'iPhone 15 Pro' }}
    steps:
      - name: Checkout
        uses: actions/checkout@v3
```

```

- name: Build
  run: |
    xcodebuild build-for-testing -scheme "$scheme" -"$project_type"
"$project_name" -destination "platform=$platform,name=$device"
- name: Test
  run: |
    xcodebuild test-without-building -scheme "$scheme"
- "$project_type" "$project_name" -destination
"platform=$platform,name=$device"

```

OK, so what do we have going on here?

First of all, you'll notice I've modified the `on:` section. This workflow will run any time there is a Pull Request in our repository, except for PRs that only affect the documentation, Readme, or workflows other than this one. *(Note that if you're following along step by step, this action will no longer kick off each time you push, unless you open a PR. You may want to keep*

`push:`

```
branches: [ "main" ]
```

event we had before, until you're happy with your workflow, and then you can update your event triggers.

You'll also see a new `concurrency:` section. This will automatically check if any other workflows are running on this specific commit, and cancel them if true. This prevents potential duplicate workflow runs that can happen if you have multiple PRs open.

You'll also see a new section called `env:` where we set up our environment variables. These variables persist depending on their scope. Because I've defined them in the `build:` job, they are available to both the Build and Test steps, but you can also define them at the step level, in which case they would have local scope to the step only. I used [context](#) syntax, which is a way to make the variable available both on the virtual machine runner and GitHub Actions itself; you can also define environment variables, for example `device: iPhone 15 Pro` and they would still be interpolated on the destination runner, but they aren't available in the parts of the workflow that do not get sent to the runner. In this way contexts are a bit more flexible, although it doesn't matter in this use case. I've also added a timeout parameter; it's always a good idea to prevent your job from running indefinitely, especially if you're going to be billed for it.

The checkout step is the same as before, but now we've got two new steps: `Build` and `Test` that run shell commands. These commands are going to run on the default shell on the runner (unless you change shells first), so you'll be using bash syntax for this. The command to build an Xcode application using the CLI `xcodebuild` tool is listed in the build step with all the parameters necessary to tell Xcode what to build, and you can see we are referencing the environment contexts we made before. The test step has a similar command, although at this

point the build will have been completed and is chilling in the DerivedData folder, so we just use `test-without-building` to run our tests.

Let's commit and push this workflow! If you kept the original event trigger, you'll see your action kick off - if not, open a PR. After a few minutes, you should see the result of your build and test action. Did it pass? Did it fail? Click on your workflow run for more details.

In your workflow run, you'll be able to see the console output from the local machine, and it will tell you all the details about how your build and testing process went. From here you can diagnose any issues you may have. These could either be with your action workflow itself (perhaps you didn't pass the right environment variables), or something with the machine (perhaps you need to select a different version of xcode before building).

Note: Seems silly to hard code our scheme, destination, and project type in our workflow, right? What if we change the scheme or add some pods and create a workspace? For this introductory article, we just wanted to introduce you to actions. However, if you're looking to develop a more robust workflow that can work on multiple projects, you have options. You can use the `iOS Starter Workflow` template that GitHub provides when you click "New Workflow" in the GUI that uses shell scripts and Ruby to parse out your scheme and project type (although last time I checked, it wasn't working properly). You can use an action available on the marketplace, such as [sersoft-gmbh/xcodebuild-action@v3](#), or you can write your own. Many companies may already have a powerful automation suite that uses fastlane and ruby, and you'd be welcome to try that out by checking out our [fastlane-template](#). If you want to stick purely with actions, the action linked above isn't perfect, but one of the better open source options on the marketplace.

What about Dependencies?

Our sample project doesn't contain any package dependencies or cocoapods, so we didn't have to worry about them, but most applications will have to resolve dependencies before building. Luckily it's not particularly challenging to resolve your dependencies or install pods.

```
- name: Install Cocoapods
  run: |
    bundle exec pod install
```

```
- name: Resolve package dependencies
  run: |
    xcodebuild -resolvePackageDependencies -workspace
    MyProject.xcodeproj/project.xcworkspace -scheme MyProject
    -clonedSourcePackagesDirPath .
```

Note: If you are using private packages, you'll have to authenticate over SSH first.

Since your dependencies are less likely to change than your source code, sometimes it makes sense to cache them so you don't need to spend valuable runtime building them over and over again. You can learn about how to [cache dependencies](#) in the GitHub Documentation. If you're self-hosting, you can also create workflows to clean your build artifacts, derived data, and caches, and only run those workflows as needed - as your artifacts and dependencies should remain on your runner. It may be prudent to have a clean workflow with a dependency updater workflow to complement your build and test workflow.

For iOS specifically, you can see these examples for how to cache SPM and Cocoapods.

Using actions/cache@v3 with Swift Package Manager:

```
- uses: actions/cache@v3
  with:
    path: .build
    key: ${{ runner.os }}-spm-${{ hashFiles('**/Package.resolved') }}
    restore-keys: |
      ${{ runner.os }}-spm-
```

Using actions/cache@v3 with Cocoapods:

```
- uses: actions/cache@v3
  with:
    path: Pods
    key: ${{ runner.os }}-pods-${{ hashFiles('**/Podfile.lock') }}
    restore-keys: |
      ${{ runner.os }}-pods-
```

Note: actions/cache@v3 is for caching files that don't change often; if you're looking to upload build outputs or logs, use actions/upload-artifact@v1.

And.... that's it? You now have CI setup for your project. Every time you open a PR, this workflow will run. Pretty simple and easy to set up and maintain compared to some of the options of the past. Of course, this is a simple workflow. For many real-life projects, you'll need to add some additional steps into your workflow to make this action robust:

- Project configurations and production environments
- Resolving package dependencies, if using SPM or remote packages.
- Simulator selection and state refresh, and setup (set region, location, etc) - especially if self-hosting

- Upload test results and/or screenshots to generate QA reports

Now we will continue on to a slightly more complicated workflow: deployment.

Xcode Continuous Delivery - Upload to TestFlight Tutorial

Now I'm going to walk you through an example workflow for continuous deployment. For iOS applications, many organizations use Firebase App Distribution or Apple's TestFlight for deploying test builds. Using App Store Connect API, we can deliver our builds directly to TestFlight whenever we specify. For some organizations this may be nightly builds, for others this may be every beta build, or others just use this process for deploying App Store Releases.

The general workflow for deployment of Xcode applications is similar (whether it's ad-hoc, enterprise, or App Store), although before we can write a workflow we need to deal with another area we didn't touch on the build and test workflow: secrets management. You can build and test a project locally without needing any credentials, but to archive, export, and upload an App Store build, you'll need to safely share your Apple/iOS distribution certificate and App Store Connect API Key without compromising your security. Luckily, GitHub has provided an easy way to store secrets in our repository.

First thing first, you'll need to be familiar with [App Store Connect API](#), as I won't go into the details here. You also should be familiar with how to codesign and distribute an application manually, as that is what we are about to automate. Finally, you should be familiar with keychain assistant, credentialing, and how to export your distribution certificate into a password-protected p12 file. (If your organization prefers to use keychain databases, you can do that as well following a similar process).

NOTE: This workflow is much more involved and highly dependent on the way an organization manages their credentials, and therefore many organizations maintain their own servers with private keys, distribution certificates, and provisioning profiles safely stored on the machine. Therefore, I'm just going to explain the general steps of what you need to do to deploy your build to TestFlight on a GitHub hosted runner. If you're looking for a quick solution and OK with using a GitHub hosted runner, I've built [Xcode-Deploy](#), an open-source action you can try out. This action is in early development and likely needs additional support for more complex projects - contributions are welcome or feel free to fork to meet your needs.

Before continuing, make sure you have access to your:

GitHub:

- Remote repository settings. If you are part of an organization your permissions may not allow you to see this, so contact your administrator.

App Store Connect:

- App Store Connect API Key ID.
- App Store Connect Issuer ID.
- App Store Connect Private Key (.p8)

Apple Developer:

- A password-protected and exported Apple distribution certificate and the private key that created the CSR that was used to generate it, exported together as a .p12 file.
- The password associated with the above file.
- A manually created provisioning profile for your export method (App Store) that references the certificate above.

Additionally:

- An ExportOptions.plist file.
 - When you export and upload a build using Xcode GUI, you navigate through a series of screens and select some export options. This needs to be created manually for CLI uploads. There are some generation tools out there, but I found it's easiest just to manually export your project in the Xcode GUI, and open the folder where the .IPA is stored and take the ExportOptions.plist file from there and commit it to your source repository. You can also use `/usr/exec/PListBuddy` to generate your export options plist.

Secrets Management

GitHub secrets are stored in the repository settings and like most secret managers, you can only Create, Update, and Delete (No Reading allowed!). You cannot upload binaries or files to the repository so the best practice is to convert your files into base64 strings, which can be saved in the secrets, and then decode them in your workflow.

1. Navigate to your repository on GitHub.com
2. Click Settings
3. Click Secrets -> Actions
4. Click New Secret
5. Open Terminal and type `base64` and then drag your secret file from Finder to the terminal, and hit enter.
6. A long string will appear. Copy that string into the secret field on GitHub and Save.
7. Give the secret a name that references what it is. Convention is to use capital letters and underscores, e.g. `APP_STORE_PROVISIONING_PROFILE`.

For this workflow, You need to add six secrets, three of which need to be converted to base 64:

- App Store Provisioning Profile (base64 string)
- Distribution Certificate p12 (base64 string)
- App Store Connect API private key (base64 string)
- App Store Connect ID
- App Store Connect Issuer ID
- The password of the p12 container that was set when the p12 was exported







When you're complete, you'll have something like this:

Actions secrets

New repository secret

Secrets are environment variables that are **encrypted**. Anyone with **collaborator** access to this repository can use these secrets for Actions.

Secrets are not passed to workflows that are triggered by a pull request from a fork. [Learn more](#).

 APPSTORE_PROVISIONING_PROFILE	Updated 2 days ago	Update	Remove
 AUTH_KEY_ID	Updated yesterday	Update	Remove
 AUTH_KEY_ISSUER_ID	Updated 2 days ago	Update	Remove
 AUTH_KEY_P8	Updated yesterday	Update	Remove
 DISTRIBUTION_CERTIFICATE_P12	Updated 2 days ago	Update	Remove
 DISTRIBUTION_CERTIFICATE_PASSWORD	Updated 2 days ago	Update	Remove

Most of these values should be relatively static; The Distribution Certificate and Provisioning Profile will renew on an annual basis if not refreshed sooner. If any of these secrets do change, you just need to click the update button to update them. Once again, you can't read from here: so if you need to store the secret in a way that you can read it back, use a secure vault or a secrets manager that allows you to safely read and share secrets.

Building The Archive, Export, and Upload Workflow

Now that the secrets are stored in GitHub, let's build the action.

NOTE: The following tutorial is for a GitHub Hosted Runner, but you can also use it on your own runner, provided:

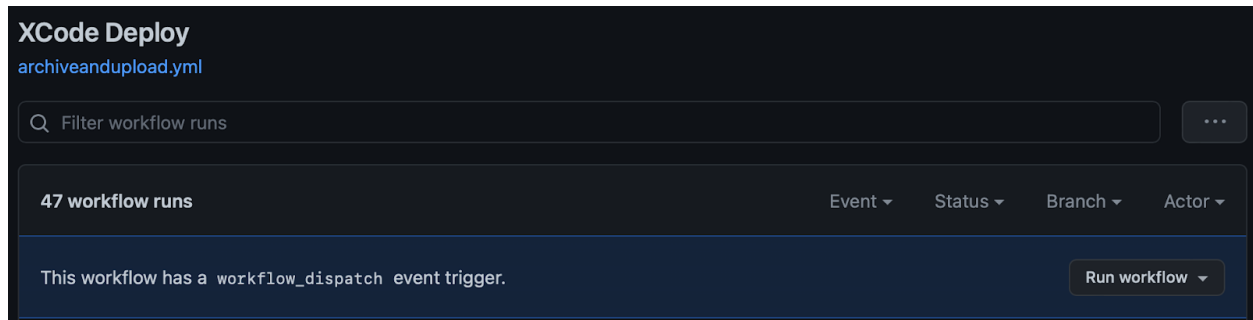
- *The proper version of Xcode and Xcode CLI Tools needs to be installed and selected on the machine.*
 - *You may need to enter your login keychain password for the workflow to add the codesign keychain to your user account.*
 - *You may (probably will) have to adjust the workflow to work with your machine.*
 - *If you manually install your certificates and provisioning profiles on your self hosted runner, you won't need to do those steps in the tutorial.*
1. First, create a new file called `deploy.yml` in your `.github/workflows` folder. Let's start with the `name:` and `on:` block.

```
name: Archive and Upload to TestFlight

on:
  workflow_dispatch:
    inputs:
      configuration:
        description: 'Configuration (default: Release)'
        required: true
        default: 'Release'
      scheme:
        description: 'The scheme to archive.'
        required: true
      path-to-export-options:
        description: 'Relative path and filename to ExportOptions.plist'
        required: true
        default: 'ExportOptions.plist'

concurrency:
  # Ensure only archives one at a time.
  group: 'Xcode-Deploy'
  # Cancel any other archives when a duplicate is detected.
  cancel-in-progress: true
```

There are a couple new things here. Firstly, `workflow_dispatch:` is how you tell GitHub this is a manually triggered workflow. Once this workflow is merged into the default branch of your repository, this workflow will show up in the Actions tab, and there will be a button available to trigger it manually. [Workflow_dispatch](#) is a powerful event trigger that can be used to schedule jobs.



Secondly, you'll notice there are `inputs:` listed under the event. These inputs appear when you click on your Run Workflow button. This is one way you can provide the values that xcodebuild needs to properly archive. You can also just hard-code these values into your action, if you don't need the flexibility or expect them to change. If you plan to call this workflow from another workflow, you'll need to set them up as inputs as we did here.

Run workflow ▼

Use workflow from

Branch: main ▼

Configuration (default: Release) *

Release

The scheme to archive. *

Relative path and filename to ExportOptions.plist *

ExportOptions.plist

Run workflow

2. Now that's add information about the job and the first couple steps.

```
jobs:
  xcode-archive:
    name: Archive iOS for App Store Release
    timeout-minutes: 15
    runs-on: macos-13
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Determine File To Build
        run: |
          if [ "`ls -A | grep -i \\.xcworkspace\$`" ]; then
            filetype_parameter="workspace" \
```

```

    && file_to_build="\ls -A | grep -i \\.xcworkspace\$`; \
    else filetype_parameter="project" && file_to_build="\ls -A | grep
-i \\.xcodeproj\$`; fi
    file_to_build=`echo $file_to_build | awk '{$1=$1;print}'`
    echo "TYPE=$filetype_parameter" >> $GITHUB_ENV
    echo "FILE_TO_BUILD=$file_to_build" >> $GITHUB_ENV
    echo "PROJECT_NAME=$(echo "$file_to_build" | cut -f 1 -d '.')" >>
$GITHUB_ENV

```

Just like on our build and test workflow, we specify the job, timeout, and what kind of image to use for the runner. Then we checkout the repository, just as before. The next step is a shell script that will automatically find the project or workspace in the repository and set some environment variables that will be passed to xcodebuild later. This script came from the actions team and seems to work well, but you can replace all this by hard-coding these flags later on, if you're only building for one application and don't expect these to change.

3. Now we are going to reference some other open-source actions on the marketplace for importing our certificates and provisioning profiles. If you are security conscious you can review these actions before implementing them as they will be handling our secrets. They also aren't too complicated so you can implement them yourself, if you wish.

```

- name: Import Certificates
  uses: apple-actions/import-codesign-certs@v1
  with:
    p12-file-base64: ${ secrets.DISTRIBUTION_CERTIFICATE_P12 }
    p12-password: ${ secrets.DISTRIBUTION_CERTIFICATE_PASSWORD }
    keychain: codesign
- name: Install App Store Profile
  uses: dietdoctor/install-ios-provisioning-profile@latest
  with:
    profile-base64: ${ secrets.APPSTORE_PROVISIONING_PROFILE}

```

The first step will create a temporary keychain with our distribution certificate and unlock it, and shows you how to call another action from within your workflow. You'll see the new **uses:** syntax, which is how you reference another action. The syntax is the username of the account hosting the action / the action name @ the tagged release (you can also use a hash). This action requires three inputs that we pass in with **with:**. And now you can see how easy it is to reference our secrets: simply use context syntax **`\${ secrets.NAME_OF_SECRET}`**. It's important to note that these secrets will never be outputted to the console and will appear as *******.

The second action will take our provisioning profile in as a base64 string, decode it, and install it on the runner (by copying it to the appropriate directory:

`~/Library/MobileDevice/Provisioning Profiles`

Once again, these actions can be replaced by your own implementation. Chances are you have one already, if you're migrating from Jenkins or another CI tool.

4. Now that we have all the inputs needed to build the archive, let's build it. Add another step:

```
- name: Build and Archive
  uses: sersoft-gmbh/xcodebuild-action@v3
  with:
    action: archive
    ${{ env.TYPE }}: ${{ env.FILE_TO_BUILD }}
    scheme: ${{ github.event.inputs.SCHEME }}
    sdk: iphoneos
    build-settings: >
      -archivePath ${{ env.PROJECT_NAME }}.xcarchive
    derived-data-path: build/derivedData
    destination: generic/platform=iOS
    configuration: ${{ github.event.inputs.configuration }}
```

A couple things to note here. Once again, I'm referencing another popular action on the marketplace that is essentially a wrapper for xcodebuild. You don't need to use this and can use the same syntax we used in our previous workflow, build and test. I just like using this action because it's simple and I've checked on the implementation on the other side. The inputs are pretty self-explanatory so I won't go into detail here, but you can review the man pages for xcodebuild if you need to.

You'll notice a couple of contexts here. On the fourth and eighth lines, I'm referring to environment variables I set in the script on the second step (after checkout). These will resolve to `project: ActionsDemo.xcodeproj` and `ActionsDemo`, if you're using our sample project. If you didn't use the script, you can just hard code these values. You can set environment variables similar to how we did on Build and Test, or you can write directly to the environment as we did in the script, and then access them as so. On the fifth and last line of the code snippet you can see how we can access our inputs. The input that someone will type on the Run Workflow screen will be accessible using this context. If everything was properly inputted and resolved, this will kick off an archive task on `xcodebuild`, and a few minutes later we will have `ActionsDemo.xcarchive` available in the base of the GitHub workspace.

5. Now that we have the archive we are getting close to exporting and uploading. First, we need a small utility to help us extract our final base64 encoded string; the App Store Connect private key.

```
- name: Get App Store Connect API Key
  uses: timheuer/base64-to-file@v1.1
  with:
    fileName: AuthKey_${{ secrets.AUTH_KEY_ID }}.p8
    fileDir: ${github.workspace }}/private_keys
    encodedString: ${secrets.AUTH_KEY_P8 }
```

This will place the private key in a subfolder `private_keys` with the name that xcodebuild expects. Once again, review this action or implement this yourself if you are worried about your secrets being processed with external code.

6. At this point we just need to export the archive to create the IPA, and then upload it. I broke it up into two steps: exporting the archive and then uploading it separately.

```
- name: Export Xcode archive
  run: |
    xcodebuild -exportArchive -verbose \
      -sdk iphoneos \
      -archivePath ${github.workspace}/${env.PROJECT_NAME} \
      -exportOptionsPlist ${github.workspace}/${github.event.inputs.path-to-export-options} \
      -exportPath ${github.workspace} \
      -authenticationKeyIssuerID ${secrets.AUTH_KEY_ISSUER_ID} \
      -authenticationKeyID ${secrets.AUTH_KEY_ID} \
      -authenticationKeyPath ${github.workspace}/private_keys/AuthKey_${secrets.AUTH_KEY_ID}.p8 \
```

Once again, hard code your project name here if you didn't use the script. Make sure to include your ExportOptions.plist in the repository or this command will fail. You can also generate your ExportOptions.plist using a script before invoking xcodebuild. Although most of the work here is done locally, xcodebuild will ping App Store Connect using your credentials to check a few things, such as the app record, your developer account status, if you have API access, etc. If you run into an error here, you'll get a JWT token back from App Store Connect explaining the error and you can usually find out why on the Developer Forums.

7. Finally, you can upload your submission to the App Store and TestFlight using altool:

```
- name: Upload to App Store Connect
  run: |
    xcrun altool --upload-app -f ${github.workspace}/${env.PROJECT_NAME}.ipa -t iOS \
      --apiIssuer ${secrets.AUTH_KEY_ISSUER_ID} --apiKey ${secrets.AUTH_KEY_ID}
```

Once again, hard code your ipa name if you didn't use the script in step two. If all goes well, you'll be getting an email from App Store Connect saying your submission was successful in a few minutes.

8. Lastly, it's always good to be extra sure you've cleaned up your environment and scrubbed it of any secrets. Although hosted Runners are containerized and are terminated after the run, it's still best practice to do so. Some of the actions we've referenced have post-run cleanup steps build in, but I've gone ahead and deleted our temporary keychain and removed the App Store Connect API Key from the instance.

```
- name: Cleanup
  run: |
    security delete-keychain codesign.keychain
    rm -rf ${github.workspace}/private_keys || true
```

And like that, you've now automated your continuous delivery pipeline using GitHub Actions! But where to go from here? Chances are if you're working on a production application, you'll need to add some more steps to your workflow. You may want to do different things where you are releasing. You also should consider sanitizing inputs because shell scripts are susceptible to script injections.

Android - Build, Test, and Upload to Firebase Tutorial

Now let's take a look at setting up a workflow to build and test a debug build for your Android application. My experience with this is using Firebase App Distribution for testing of Android debug builds, so I'll show you how to do that as well.

Android is even easier than iOS (in my opinion) because there are some really powerful setup actions already available to use on the marketplace. Starting with a sample application that builds successfully and has some tests, we create a workflow file in the same way as before:

1. Navigate to your project repository folder locally (Finder or Terminal)
2. Create a new special folder called `.github` (If you are on a Mac and using finder, you may want to show system folders: 'CMD+SHIFT+'.)
3. Create another folder beneath this one called `workflows`
4. Open your favorite text editor and create a new file called `buildandtest.yml`
5. Paste in the following code as a starting template:

Note: If your repository is private and you want to avoid using your allotted minutes, change runs-on: macos-latest to self-hosted and follow the instructions at the end of the article on how to self-host.

```
name: Android Build and Test
on:
  pull_request:
    branches:
      - development

  workflow_dispatch:
    inputs:
      branch:
        description: 'Select branch to generate debug build.'
        default: "development"
        required: true
jobs:
  build-test-upload:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: set up JDK 11
        uses: actions/setup-java@v3
        with:
          java-version: '11'
          distribution: 'temurin'
      - name: Setup Gradle
        uses: gradle/gradle-build-action@v2
      - name: Grant execute permission for gradlew
        run: chmod +x gradlew
      - name: Build with Gradle
        run: ./gradlew clean assembleDebug
      - name: Run Debug Tests
        run: ./gradlew testDebug
```



```

- name: Upload Test Reports
  if: ${{ always() }}
  uses: actions/upload-artifact@v2
  with:
    name: test-reports
    path: '**/build/reports/tests/'
- name: Upload a Build Artifact
  uses: actions/upload-artifact@v2
  with:
    name: debug
    path: app/build/outputs/apk/debug/app-*.apk
- name: Upload artifact to Firebase App Distribution
  uses: wzieba/Firebase-Distribution-Github-Action@v1.3.4
  with:
    appId: ${{secrets.FIREBASE_APP_ID}}
    token: ${{secrets.FIREBASE_TOKEN}}
    debug: true
    releaseNotes: "Android Debug Build / PR:
${{github.event.pull_request.id}} (${{github.ref_name}}) / PR Link:
${{github.event.pull_request.url}}"
    groups: my-group
    file: app/build/outputs/apk/debug/app-debug.apk

```

9. Save and commit this file and push it to your remote repository.
10. Now open your repository on GitHub and navigate to the Actions tab. If you don't see the workflow file here, you may need to open a Pull Request or merge this workflow file into your default branch.

Let's break down this workflow file step by step.

First we name the workflow.

Then we define in the **on:** block when this workflow runs. In this case, any PR to the **development** branch will trigger this workflow. There is also a manual workflow trigger, **workflow_dispatch:** that will allow you to build and test debug apk manually, and specify which branch you want to use by providing a job **input:**. This input appears on your workflow in the GitHub Repository Actions tab.

Next, we define the **jobs:** for our workflow: In this case, we have a single job, although you could split Building, Testing, and Uploading into separate jobs if you wanted to - you would just have to set the outputs of each job accordingly.

In this case, we have one job, called `build-test-upload:` that `runs-on:` a GitHub hosted runner using the `ubuntu-latest` runner image. Even if you use a Mac for development at work or home, you can build, test, and upload Android applications solely from a linux command line (for iOS applications, you must use a Mac or Mac VM). If you wanted to set up your own runner, you could do so by switching `ubuntu-latest` to `self-hosted` and follow the instructions on how to self-host in the documentation.

Now we define the steps: for the job. This job has eight steps in total.

1. The first step, `Checkout`, uses the GitHub Actions' teams standard checkout action. This will pull down the code from your repository into the runner's workspace. By default this will just grab the most recent commit on HEAD, but you can provide `fetch-depth: 0` to grab the entire commit history if you need to do something like set the build number based on the number of commits, or report diffs.
2. The second step, `set up JDK 11`, is another popular action that GitHub built for setting up Java. While this action includes an option for gradle caching, it's better to use the official action by the gradle team, `gradle/gradle-build-action@v2`. Using this action is great because we don't have to set up gradle caching ourselves, and if we want to customize our gradle version or fine tune our caching, this action has options for that.
3. The next three steps are the actual build steps.
 - a. First, we make gradlew executable.
 - b. Second, we clean and build the debug build with `./gradlew clean assembleDebug`
 - c. Third, we run the tests with `./gradlew testDebug`

In each of these cases, we are using a `run:` step which will run the commands in the default shell of the runner, in this case, bash. All of these commands should look familiar if you've built an android application from the command line or watched the terminal when Android Studio is sucking up 99% of your CPU.

4. The next step uses another popular action, `actions/upload-artifact@v2`. To upload the test reports to the workflow run. This action is used when you want to upload reports, logs, APKs, or build outputs and is one of the most popular actions in use, for obvious reasons. This action requires some inputs which are listed using `with:` and we pass in the name and path of the artifact. We also have a conditional here `if:` that allows us to specify when we want to run a step. In this case, it's set to always, but in this context you may want to only upload the reports if it's a nightly build or if a PR build, etc.
5. The next step is very similar to the previous step, in that it also uses `actions/upload-artifact@v2` to upload the actual APK to the workflow run.
6. The final step shows us how to upload the build to firebase using another popular action, `wzieba/Firebase-Distribution-Github-Action@v1.3.4`. This one requires a few more inputs, including two secrets. Secrets are stored in your

repository settings. You can paste your secrets for the Firebase App ID and Firebase CI Token, give them a name, and reference these as so: `${{secrets.MY_SECRET_NAME}}` for more detail on how to manage secrets, check the iOS section where I go into more detail. We also add the path to our artifact we created earlier, as well as a flag for debug and we build some release notes using some contexts available to us by the github runner itself.

If you've added your secrets appropriately, and tweaked this workflow to work with your project, open a PR or trigger it manually, and you should have successfully built, tested, and deployed your project to Firebase using GitHub Actions! Of course (just like in the Xcode tutorial) you're bound to run into some speed bumps along the way - and that's OK - the nice thing about actions being so intuitive and fast is that usually you can work through them.

Self-hosting

As mentioned multiple times in this article, there are some real benefits of self-hosting your runner. For larger organizations, the biggest factor is going to be cost: GitHub only includes a certain number of minutes with your account, and it's quite easy to exceed that. GitHub charges a premium for Windows and Mac hosted runners (currently 3x and 10x multiplier), but a self-hosted runner is free to use (of course there are costs for hosting). Although there are real benefits to run your CI/CD pipelines from a clean image, sometimes it's just not feasible or practical, especially if you need to install and configure a lot of software (which will take up billing minutes, of course), need to whitelist or coordinate with remote resources (although you can do that), or just in general need more control of your pipeline.

Self-hosting is really easy so I won't go into detail here. You can follow the instructions in the [documentation](#) and you shouldn't have any problems.

Generally, you will want to:

1. Prepare the machine for hosting
 - a. Install all the software needed for your CI/CD pipeline
 - b. Configure the software as needed
 - c. Navigate any networking considerations specific to your organization (firewalls, VPNs, etc).
2. [Install and configure the runner software.](#)
 - a. One machine can run multiple runners, but only one repository can attach to a runner at a time, unless you have an organizational or enterprise account with GitHub.
 - b. Configure the runner
 - c. Execute `./run.sh`

- d. Check your repository settings and you should see the runner listed under Actions -> Runners.

The open source community has also developed automations to scale your GitHub runners as the needs of your organization grow. One popular solution uses [Terraform and AWS](#) to manage your runner infrastructure.

Conclusion

If you made it this far, thanks for reading! I hope you learned something. Feel free to provide any feedback or suggestions, clarifications, or corrections by emailing me at vincent.frascello@gmail.com.