Denys Fiediaiev Follow
Sep 27, 2021 · 8 min read · Listen

Building CI/CD for Xamarin apps with GitHub Actions

Today, I will describe my implementation of CI/CD pipeline for Xamarin mobile app using GitHub Actions in 7 steps.

GitHub Actions is a trendy CI platform for web development, but it is not entirely suitable for mobile development due to self-handled app signing and expensive macOS plans.

But why am I interested in GitHub Actions and applied it for my project despite more convenient mobile CI providers, like Bitrise?

Well…

As a developer, I use dozens of tools and services every day, and having both the CI pipeline and the code available on the single platform — GitHub — is a significant advantage for me.

1. Workflow setup

Let's start from the basics.

Everything on GitHub Actions is configured in a single workflow file (`.github/workflows/ci.yml`) stored right in the project repository. To change the configuration, you need to update that file and make a commit. GitHub automatically handles that file, and no more actions are required.

In that configuration file, we specify the conditions to trigger a workflow:

- On Push to a specific branch or a branch name pattern
- On Tag creation
- On Pull Request creation

I would not focus on the branching model and trigger the workflow on any branch push:

```
on:
  push:
    branches: [ develop ]
  pull_request:
    branches: [ develop ]

workflow_dispatch:
```


Then we need to tell GitHub Actions the platform we want to run the workflow on.

The macOS runner is suitable if you're going to build for iOS and Android platforms, and the Windows runner is suitable for Android-only.

Our demo Xamarin app targets both iOS and Android platforms, so I will use a macOS runner:

```
jobs:
  build:
    runs-on: macos-latest
```

Or, if you are using a self-hosted runner, you can specify it here:

```
jobs:
  build:
    runs-on: self-hosted
```

Now we are ready to start implementing the actual building process.

2. Repository checkout

First of all, we need to clone the repository:

```
steps:
   - name: Checkout
     uses: actions/checkout@v2
```

3. Signing setup

I would set up the app signing ASAP and let the build flow fail early if something went wrong to save expensive build time.


As GitHub Actions does not support secure file storage, we need another way to store these files. That could be a link to external cloud storage, like Google Drive.

To make things secure, we can also use GitHub's internal Secrets storage.

The problem arises because GitHub Secrets supports text values only, and we have to store files in some encoded format, like base64.

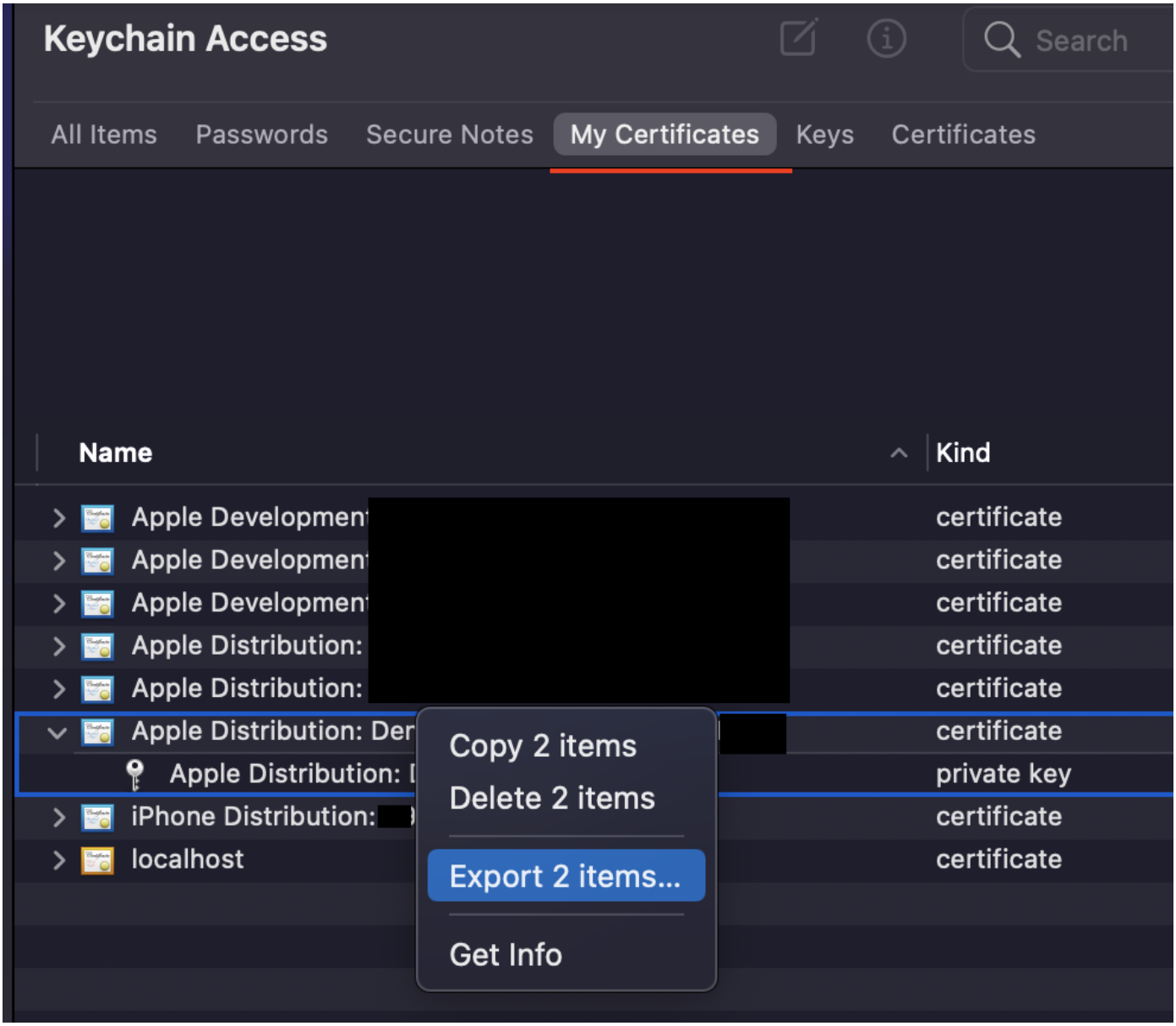
This is the simple step-by-step instruction for using base64 encoded files in GitHub Actions:

1. Encode and copy a file using the terminal command: `base64 FILE | pbcopy`.
2. Paste it to a new repository /organization secret without the empty line at the end.
3. Decode that secret into a real file in the workflow: `(echo ${{ secrets.FILE_SECRET }} | base64 — decode) > ./PATH/TO/FILE` or use it in a particular step.

3.1. iOS signing setup

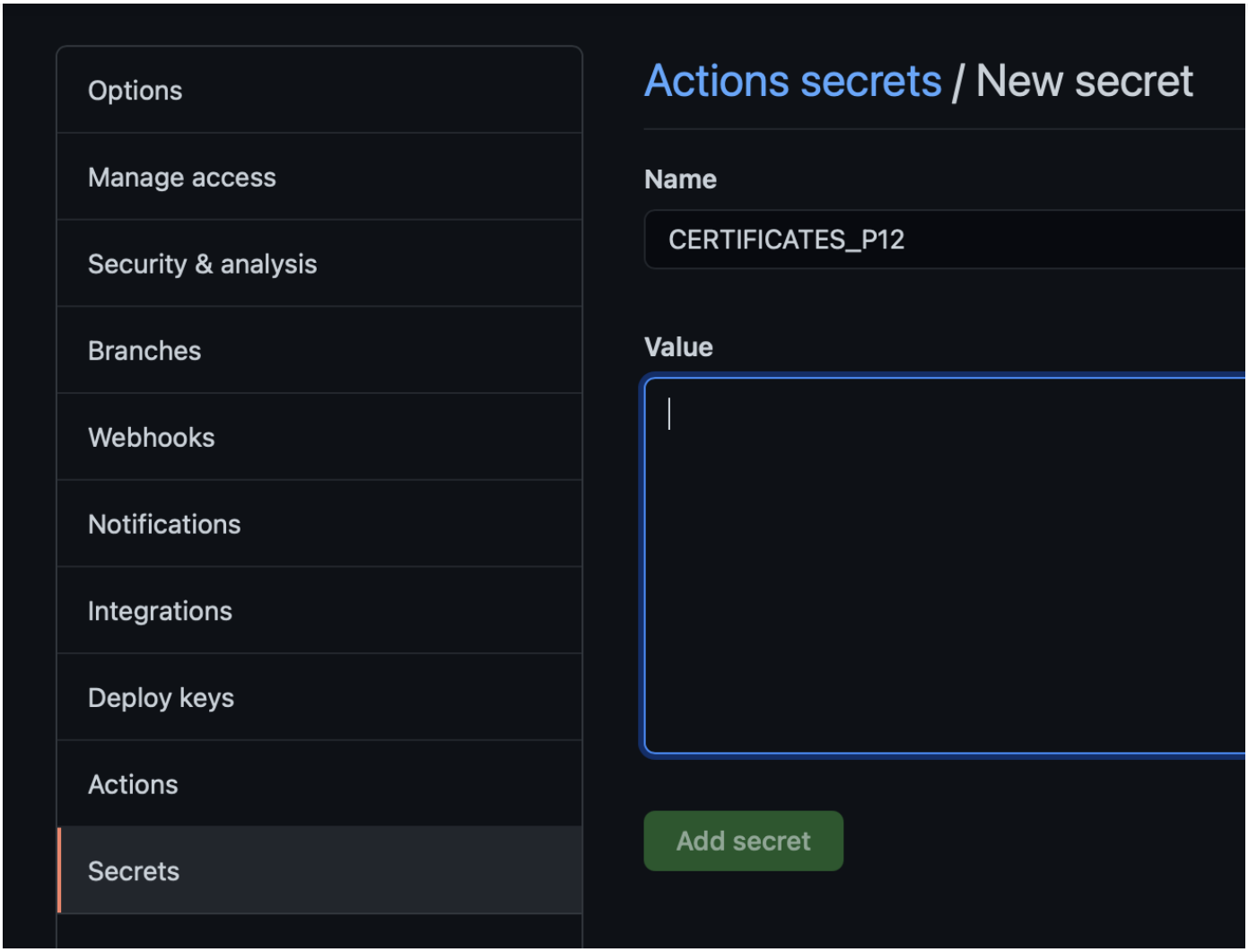
iOS signing setup can be divided into two steps: certificate setup and provision profiles downloading.

You can export a development/distribution certificate from the Keychain Access app:

Both the certificate and its key should be exported.

Then, the file needs to be converted to base64 string and saved to `CERTIFICATES_P12` variable in the Secrets section of the repository:



A secret can be added in the repository settings.

> ⚠️ *Note, any base64-encoded files must* ***not*** *have an empty line or a newline symbol at the end. Otherwise, it will not be possible to decode that file.*

The password for that certificate needs to be added as plain text into the `CERTIFICATES_P12_PASSWORD` variable.

To install that certificate, we use a third-party import-codesign-certs step:

```
- name: Setup iOS Certificates
  uses: apple-actions/import-codesign-certs@v1
  with:
```

Now we need to download a provisioning profile required to sign the iOS app. Again, we can use another third-party step called download-provisioning-profiles here to avoid manual work.

That step uses an App Store Connect API that requires an API key. You could read about generating this key from the Apple developer documentation here:

Apple Developer Documentation
Edit description
developer.apple.com

> ⚠️ *Be careful: the generated API key can be downloaded only once. So make sure to store it in a secure place.*

Three environment variables need to be created to use the API key: `APPSTORE_ISSUER_ID`, `APPSTORE_KEY_ID`, and the `APPSTORE_PRIVATE_KEY`.

The last one starts with `BEGIN PRIVATE KEY` and ends with `END PRIVATE KEY`.

Finally, the iOS provisioning step should look like this:

```
- name: Setup iOS Provisioning Profiles
  uses: apple-actions/download-provisioning-profiles@v1
  with:
    bundle-id: 'dev.sbyte.githubactionsxamarin'
    profile-type: 'IOS_APP_ADHOC'
    issuer-id: ${{ secrets.APPSTORE_ISSUER_ID }}
    api-key-id: ${{ secrets.APPSTORE_KEY_ID }}
    api-private-key: ${{ secrets.APPSTORE_PRIVATE_KEY }}
```

3.2. Android signing setup

Android signing for a Xamarin application is more straightforward: we only need to specify a Keystore file path, an app alias, and passwords required.

```
- name: Setup Android signing
  run: (echo ${{ secrets.KEYSTORE }} | base64 — decode) >
./GitHubActions.Android/keystore.jks
```

> ⚠️ *Note: you must save the Keystore file into the Android project directory so that MSBuild will find it.*

4. App version update

Another essential thing is updating the app version information. That information contains two things to make the build uniquely identifiable: the build number and the actual version name.

On iOS, you need to update the Info.plist file and set the app version to `CFBundleShortVersionString` key and the build number to `CFBundleVersion` key. We can use the embedded macOS tool called PlistBuddy here:

```
- name: Set iOS version
  run: |
/usr/libexec/PlistBuddy -c "Set :CFBundleShortVersionString ${{
secrets.APP_VERSION }}" ./GitHubActions.iOS/Info.plist
/usr/libexec/PlistBuddy -c "Set :CFBundleVersion ${{ github.run_number
}}" ./GitHubActions.iOS/Info.plist
```

On the Android platform, the version info is stored in the AndroidManifest.xml file.

Therefore, we should assign the version name to the `versionName` property and the build number to the `versionCode` property.

Unfortunately, there are no built-in tools to set these properties, and we have to use a custom regex-based parser or the existing step:


```
      version-name: ${{ secrets.APP_VERSION }}
      version-code: ${{ github.run_number }}
```

> ⚠️ *It is a good idea to follow some standards in version names, for instance, semantic versioning.*

5. NuGet packages restoring

Now, when the signing is successfully installed, we can restore NuGet packages for the whole solution:

```
- name: Restore NuGet packages
  run: nuget restore
```

6. iOS and Android apps building

At this moment, apps are ready to be built.

So we use two simple MSBuild commands to build both iOS and Android apps:

```
- name: Build iOS
  run: MSBuild /t:Build /p:Configuration=Release /p:Platform=iPhone
/p:BuildIpa=true ./GitHubActions.iOS/GitHubActions.iOS.csproj

- name: Build Android
  run: MSBuild /t:SignAndroidPackage /p:Configuration=Release
/p:AndroidPackageFormat=apk /p:AndroidKeyStore=true
/p:AndroidSigningKeyAlias=githubactionsxamarin
/p:AndroidSigningKeyPass=${{ secrets.KEYSTORE_PASSWORD }}
/p:AndroidSigningKeyStore=keystore.jks /p:AndroidSigningStorePass=${{
secrets.KEYSTORE_PASSWORD }}
./GitHubActions.Android/GitHubActions.Android.csproj
```

The first one builds a release version of the iOS app and generates an IPA file that we can

We can use that APK to distribute the app internally.

To upload the app to Google Play Market, you will need an AAB file.

The output file format is controlled by the `AndroidPackageFormat` parameter that accepts `apk` or `aab` values.

7. App Center distribution

When both apps are built successfully, we can distribute them using any app distribution service or upload them directly to the stores.

App Center is the most convenient way for Xamarin apps, but Firebase App Distribution is also a good alternative, despite being in a beta test now.

To prepare the App Center distribution, we need to create an app on https://appcenter.ms and obtain a token to use the command-line interface (CLI).

Then, we could save that token into the repository secret called `APP_CENTER_TOKEN`.

In the build script, the first step is to install the App Center CLI:

```
- name: Setup App Center CLI
  run: npm install -g appcenter-cli
```

Then we are ready to upload the iOS app and silently distribute it to users in the Collaborators group:

```
- name: Upload iOS app to App Center
  run: appcenter distribute release --silent --file
./GitHubActions.iOS/bin/iPhone/Release/GitHubActions.iOS.ipa --app
SByteDev/GitHubActions.Xamarin-iOS --group Collaborators --token ${{
secrets.APP_CENTER_TOKEN }}
```


```
- name: Upload Android app to App Center
  run: appcenter distribute release — silent — file
./GitHubActions.Android/bin/Release/dev.sbyte.githubactionsxamarin-
Signed.apk — app SByteDev/GitHubActions.Xamarin-Android — group
Collaborators — token ${{ secrets.APP_CENTER_TOKEN }}
```

To learn more about the CLI, you can run the `appcenter help` command from the Terminal or check the official repository documentation:

GitHub - microsoft/appcenter-cli: Command-line Interface (CLI) for Visual Studio App Center

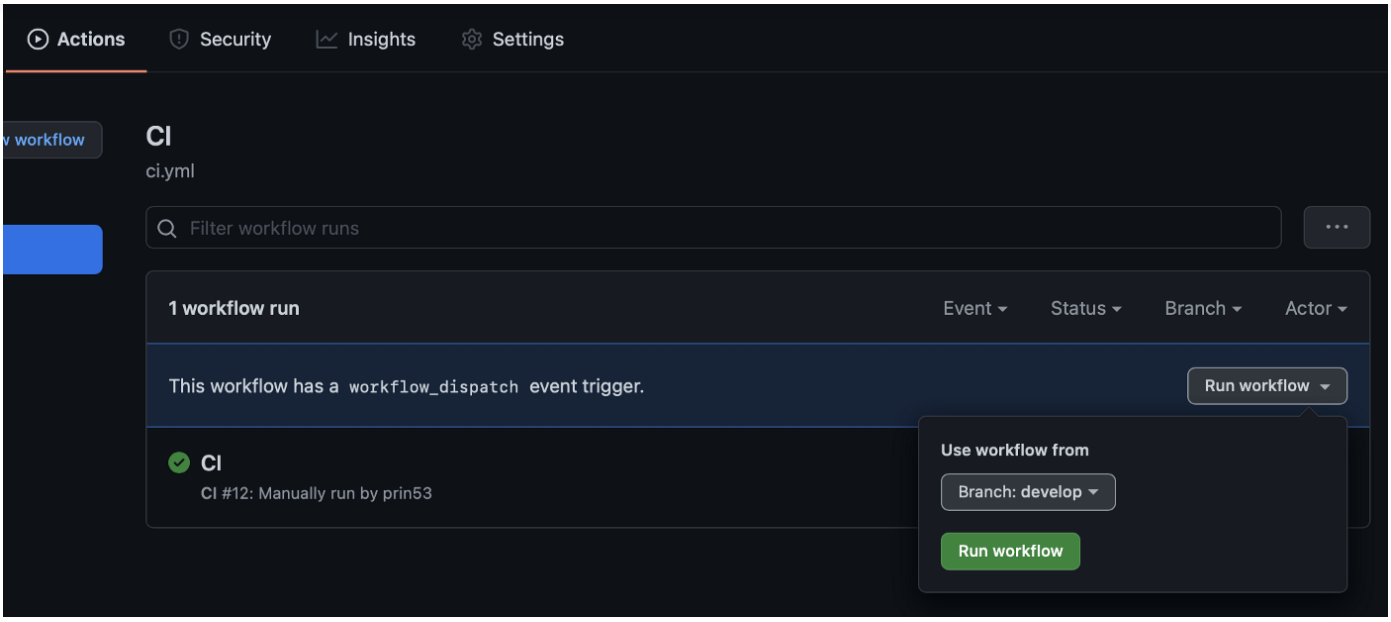
Visual Studio App Center command line interface (CLI) is a unified tool for running App Center services from the...

github.com

And that's it!

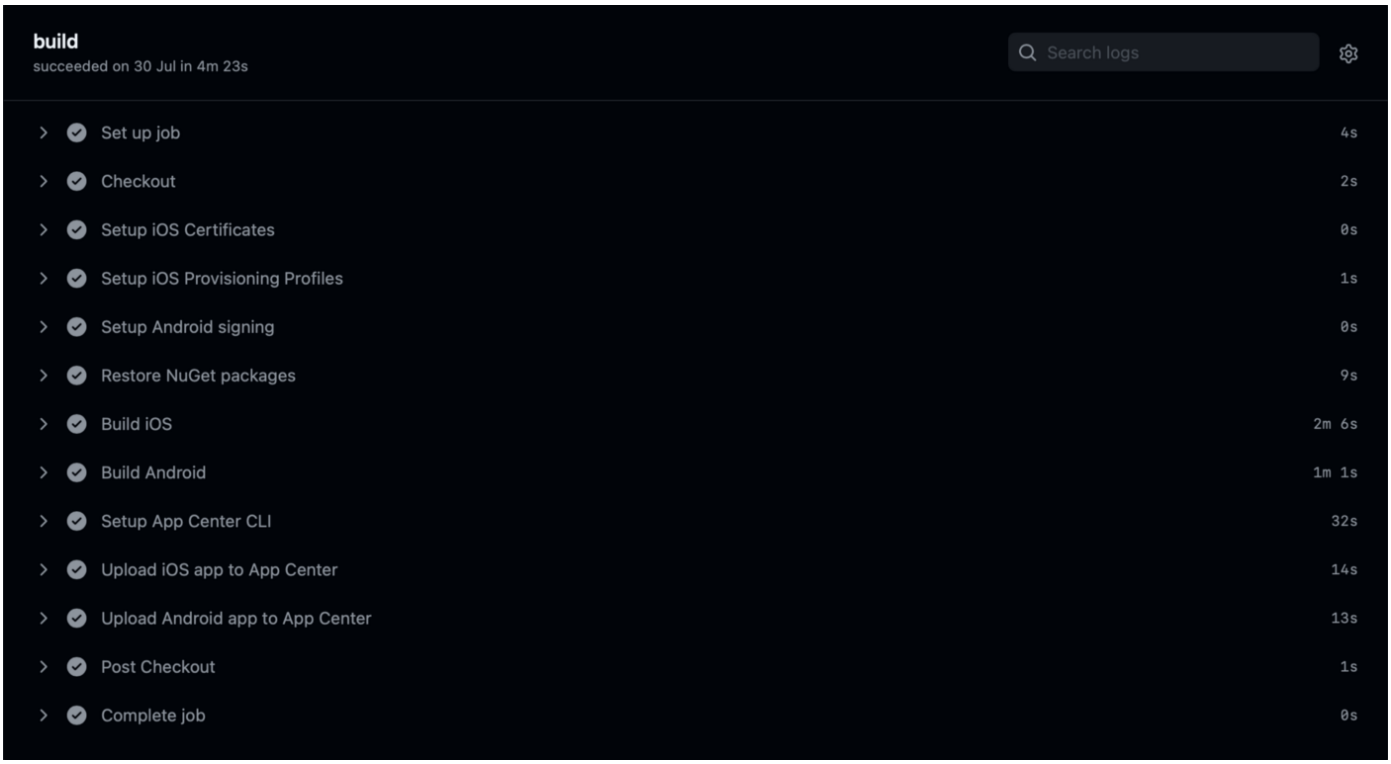
The simple Github Actions workflow for Xamarin is ready to build and distribute your apps.

The first run will be triggered automatically by GitHub when you commit the workflow updates, but you can manually run it from the Actions tab:

Running the workflow manually allows to select the branch to run on.

Later, you can examine the build logs right from the Github Actions interface.



The detailed build logs are available for the completed workflow.

An example demonstrating a simple Xamarin app CI using GitHub Actions. - GitHubActions.Xamarin.Demo/ci.yml at master ·...
github.com

Make sure to also check the whole repository:

GitHub - SByteDev/GitHubActions.Xamarin.Demo: An example demonstrating a simple Xamarin app CI...

An example demonstrating a simple Xamarin app CI using GitHub Actions. - GitHub - SByteDev/GitHubActions.Xamarin.Demo...

github.com