Xojo GitHub Actions

Would you like to automate the build processes of Xojo built applications with GitHub Actions? With no user interaction the whole process is being performed: Build all Targets, CodeSign Windows executables, create and sign the installer, package the macOS app in a .dmg and notarize the app, and for Linux a .tgz package.

What are GitHub Actions?

It's a lot. If you want to read all about it- read their documentation. To get started, you should at least know about the following terms.

WORKFLOW
A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to your repository.

JOB
A job is a set of steps in a workflow that execute on the same runner. Each step is either a shell script that will be executed, or an action that will be run.

RUNNER
A runner is a machine that runs your workflows when they're triggered. GitHub provides ready to use runners as preconfigured virtual machines. And of course you can host your own runners.

A GitHub Actions workflow can configured to be triggered when an event occurs in the repository. So how to leverage that? Let's talk about the Git Branching strategy that I'm going to use in this example first.

Git Branching Strategy

Using GitHub obviously means the one is using Git as Source Control. There are quite many common strategies and workflows to use Git. For this example I'll be using the following:

- Default Branch: `main`
  I want to make sure that all code in the main branch can be compiled at all times. Every push (or Pull Request against the main branch) should automatically build a Beta Version of the app.
  At some time I want to use the latest commit to build a final version of the application.
- Feature branches: `feature/xyz`
  To start working on a new feature or bug fix, I'll create a feature branch. As I often commit I don't want to build every single commit. There might be commits with "work in progress" which won't even compile.
  Once ready: I create a Pull Request against the main branch.
- Pull Request *(against main branch)*
  This means a feature or bug fix is ready. So a GitHub Action should kick in and build the app. If that fails it simply means the feature isn't ready for the main branch. The built Beta Version can be downloaded to do some more manual tests before merging into the main branch.
  Once the Pull Request is merged *(always using a squash commit)*, this will result in a single commit to the main branch (of the whole new feature). Same procedure: a Beta of the app is automatically built and ready for to download and test.
- Release a Final Build
  After adding couple of new features and bug fixes I want to release a final version.

Whenever I feel ready, a GitHub Actions Workflow can be started manually. That will build a Final Version, create a Release in the GitHub Repository and upload all Builds to the Release.

Alright... let's get started to set this all up!

Setup Build Mac

This subtitle says it all. A Mac of our own is needed. The reason for that is that Xojo isn't set up on GitHub's virtual macOS machines *(one would need to download and install it every single time)*. And even if it were, we would need to setup and register Xojo with a License. And Xojo Licenses take a seat on every machine - so no good idea for a fresh virtual machine on every build.

HARDWARE
That's why a Mac is needed. One that isn't being used to work with ourselves all day since it will fire up Xojo, do automated builds. So just a Mac that sits somewhere and waits for tasks to be run. It could even be just a virtual machine running in the background on your own Mac. It just needs to persist because of the Xojo licensing.

SOFTWARE
Let's start with the basics before I'm going to install the GitHub runner. For this example I have set up:

- macOS 12.6
  Username: `xojo`
- Xcode 14
  No need to "sign in" or download/install certificates on the Build Mac. Just launch Xcode at least once so that the command line build tools are being installed.
- Xojo 2022r2
  Since I always use various Xojo Versions, I'm going to put it here (and rename the .app of the Xojo IDE):
  `/Applications/Xojo/Xojo 2022 Release 2/Xojo 2022r2.app`
  Add the Plugins you're using, launch Xojo, sign in and assign a license.
  Set the following Xojo Preferences:
  - Check for updates: never
  - Don't show built apps in Finder

Xojo IDE Communicator

Build Automation with Xojo requires to communicate with the Xojo IDE. I'm going to use the project that Xojo ships with 2021r1.1 in the examples in "Advanced -> IDE Scripting -> IDE Communicator -> v2", but modify it a bit:

- add a new Parameter: -x
  This will allow to set the IPCPath. The Xojo IDE will later be launched with a custom IPCPath and make it distinct for the Xojo Version. So the own IDE Communicator can make sure it talks to the Xojo Version specified with this parameter.
- If an error occurs, never `Quit(0)`. Always Quit with an Error Code.
  The reason is that the GitHub Action will fail if a command to the IDE Communicator quits with an Error Code > 0.
- One exception to the above rule... if the Script that's being sent to the Xojo IDE contains `QuitIDE`, then we `Quit(0)`. The IDE can't confirm that it has Quit, so the IDE Communicator will get an error when sending this command. But we let the GitHub Action know that the command has been successful.

You'll find the Source Code of the modified Xojo IDE Communicator here: GitHub jo-tools/xojo-github-actions

Build it, and copy it to the Build Mac:
`/Applications/Xojo/XojoIDECommunicator/XojoIDECommunicator`

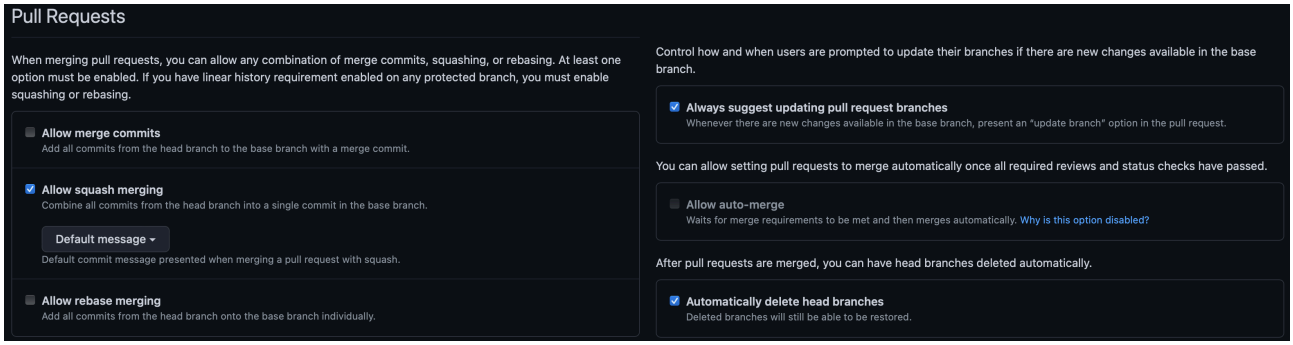
By now, the Build Mac should look like this:

GitHub Repository Settings

Remember the Branching Strategy - and the intention to use GitHub Actions? Here is how to configure the GitHub repository.

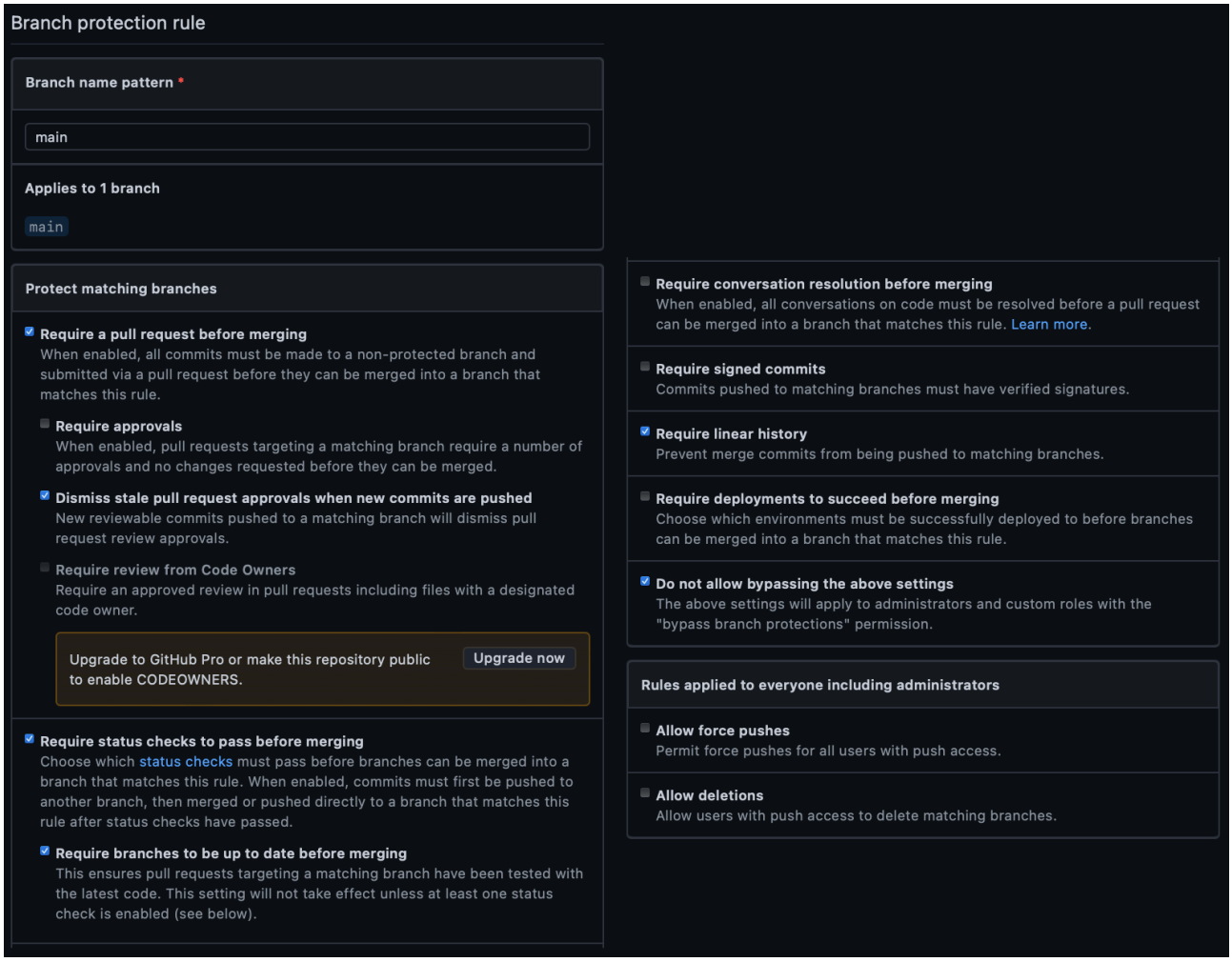
PULL REQUESTS
Settings -> General -> Pull Requests

I personally prefer a linear history. That's why I set up my repositories to allow only squash merging on Pull Requests:



BRANCH PROTECTION RULES
Settings -> General -> Branches

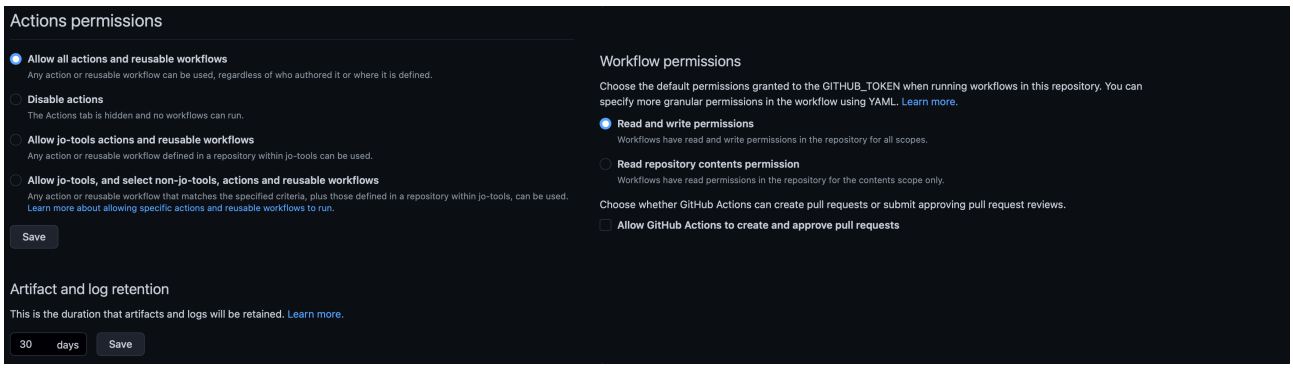
Let's add some rules for the branch `main`: Require Pull Requests and Require status checks to pass before merging - which means that the upcoming Workflows will need to succeed before merging to the main branch will be allowed.



ACTIONS
Settings -> Actions: General

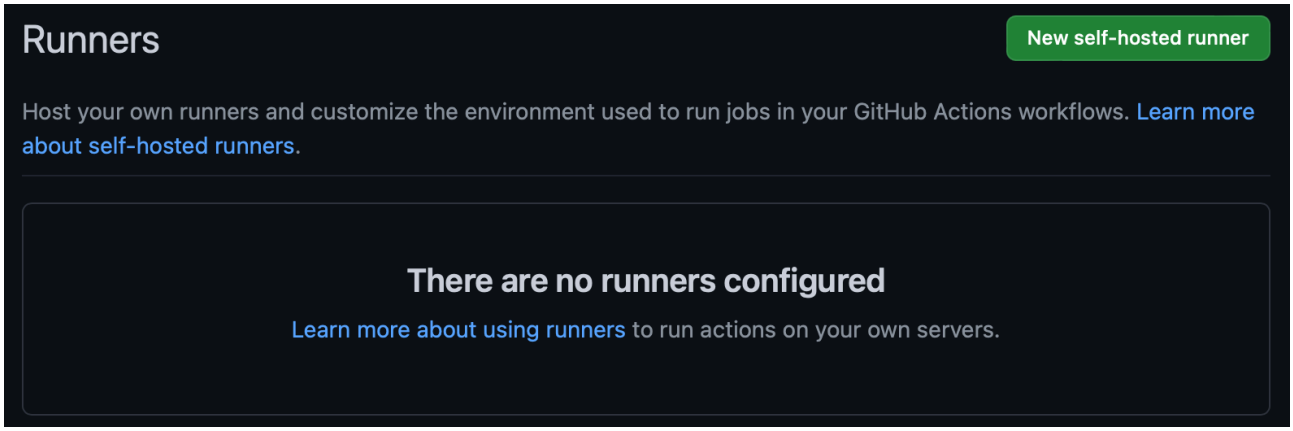
Here we need to allow actions and workflows. I tend to reduce the Artifact and Log rentention days, as I don't need GitHub to store build logs and artifacts such as beta builds for months.

In order for a workflow to to be able to create a release on GitHub, it needs read and write permissions in the repository.

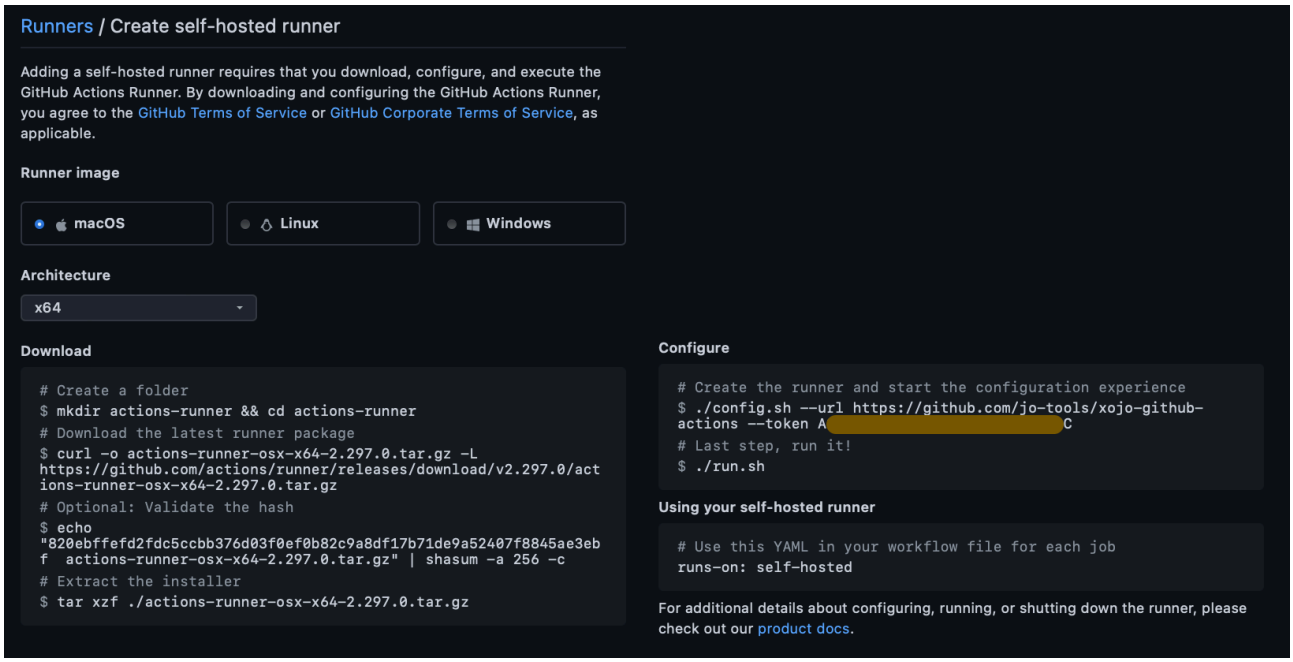


Setup self-hosted GitHub Runner

To setup a Build Mac as a self-hosted GitHub Runner, start in the GitHub Repository Settings: Actions -> Runners - and click the green button:



GitHub then shows exactly what ones need to do:

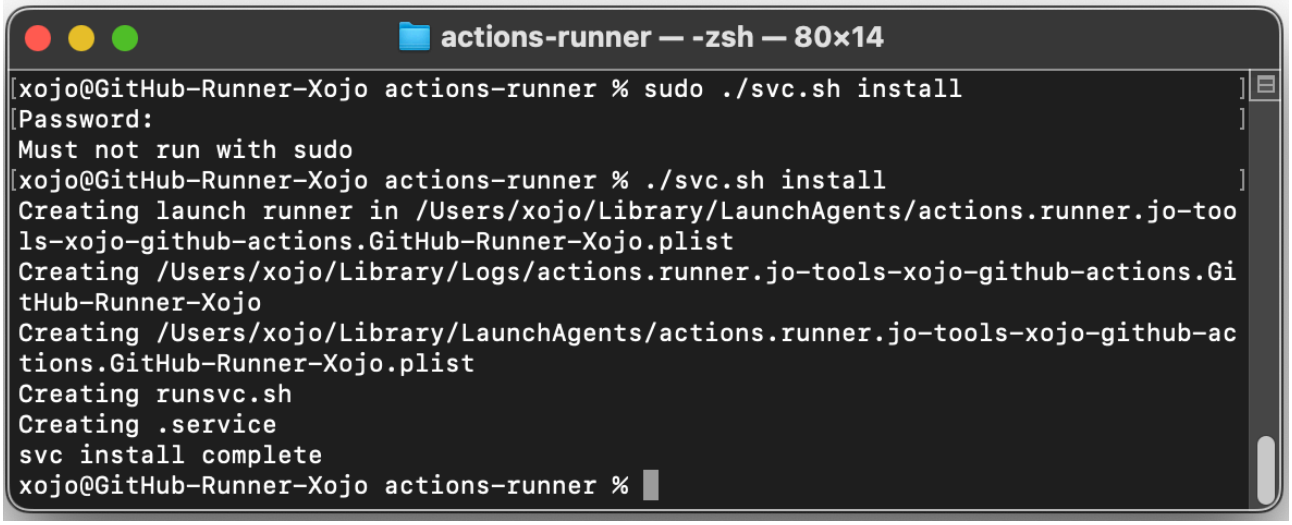


So let's copy and paste the commands in Terminal on the Build Mac. I'm going to do that in the user home folder: `/Users/xojo`

CONFIGURATION
When running the `config.sh`, then:

- add a custom Label: `Xojo`
  I want the Xojo Build Mac to have all the Labels: `self-hosted, macOS, Xojo`.
  The reason is that in the workflows I'm going to define they can be run on a machine with all these three labels. So the workflow won't be able to run on a GitHub hosted macOS virtual machine, as there is no Xojo installed there. Only on the self-hosted Mac's with Xojo installed.
- don't do the `run.sh` yet!
  I'd suggest to install the runner as a service first, so that it gets started when rebooting the Mac.

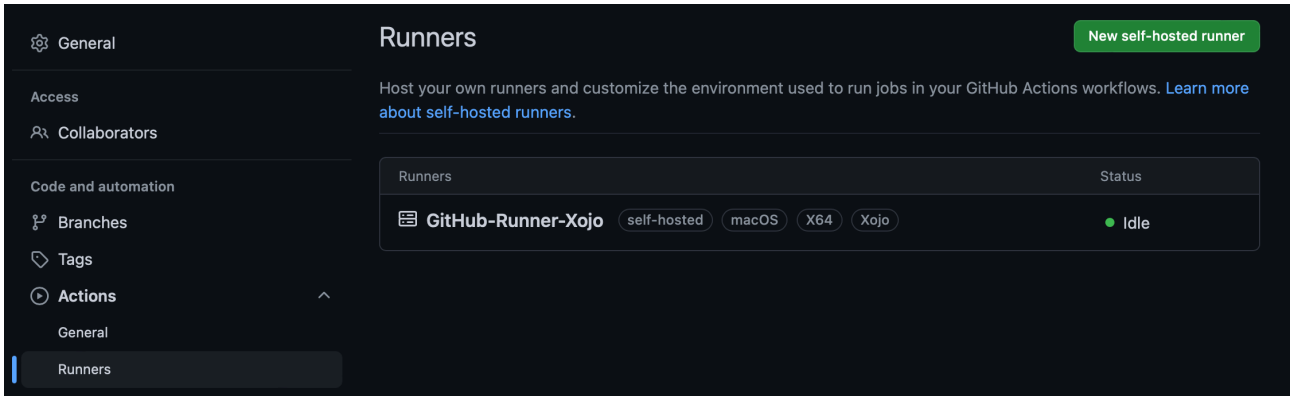
After configuration, install the runner as a service: Note, maybe "sudo" isn't required.



Let's try if the LaunchAgent works and automatically starts the GitHub runner - and reboot the Build Mac.

Once the BuildMac is up and running again, one can confirm that the runner is working and linked against the GitHub repository. Go once more to Repository: Settings -> Actions: Runners.

The self-hosted runner I have just set up should now show with a "green" Status-Icon as "idle". Congratulations - the runner is waiting for workflows and jobs to be executed.



Xojo Project

After all, I intend to build a Xojo Project. So I've added a simple example project `Xojo-GitHub-Actions.xojo_project` to the repository: GitHub jo-tools/xojo-github-actions.

Next up is to add pre and post build resources. I'm going to put all scripts and resources needed in the build process in the folder `_build`.

That way, all build-related content is under source control, too. If possible, I avoid having build-scripts just on a build server. Why? Simply because the scripts might need some changes with a newer version of the app. And should one ever need to release a bugfix of an older release... you get it: it's good to just check out any commit and have both source and build fit together.

I'm going to comment on what the scripts and workflows will be doing. Some code might be shown in this article, but not all. I'd suggest that you look at the code in the repository GitHub jo-tools/xojo-github-actions while reading this article. Side by side this all might make sense to you then.

All the scripts and workflows are written to be as reusable as possible for other Xojo projects in other repositories.

PREBUILD XOJO SCRIPT
`_build/prebuild.xojo_script`

The GitHub Actions Workflows will use shell commands to let the Xojo IDE Communicator automate the Xojo IDE. This generic PreBuild Xojo Script will do:

- Open the Xojo Project
  `OpenFile(EnvironmentVariable("GITHUB_WORKSPACE") + "/" + EnvironmentVariable("XOJO_PROJECT_FILE"))`
  You probably notice that it's being driven by Environment Variables. We'll set up them in the GitHub Actions Workflows.
- Verify some build settings
  Make sure that ShortVersion is exactly in the format: Major.Minor.Bug
  I want to have the NonReleaseVersion to be always 0.
  And make sure the Copyright is set. Don't worry when looking at the code, it's not a mistake - this one needs to be set in `PropertyValue("App.LongVersion")`.

- Set Stage Code
  Again: according to an Environment Variable.
  In the workflows I want to do Beta and Final Builds. The workflow defines what kind of build it is, and the PreBuild script makes sure the Xojo Project is set up accordingly before being built.

LINUX POST BUILD SCRIPT
`_build/linux/postbuild.sh`

This shell script gets called after Xojo has finished building all targets. Again it's a generic script that one can easily use for other projects, too. It's purpose is:

- Get Input Parameters
  They define the location of the folders, and the desired `.tgz` filename
- Make sure the build folder exists
  Otherwise fail and exit with an Error Code.
- Cleanup the build folder
  Remove any unneeded system files such as `.DS_Store` or `Thumbs.db`.
- Create a .tgz of the Linux Build

WINDOWS CODESIGN SCRIPTS
`_build/windows/codesign_x86-32bit.ps1`
`_build/windows/codesign_x86-64bit.ps1`

These Powershell Scripts will CodeSign *(SHA1 and SHA256)* all `.exe` and `.dll` files of the built windows application. They expect Environment Variables to be set up, which will be set in the GitHub Actions Workflow, as well as the parameter with the CodeSigning Certificate (`.pfx`) password. How to deal with the Certificate will come later.

What needs to be check and modified depending on the Xojo project is the list of files to be codesigned *(you should CodeSign all `.dll` and `.exe` files)*. The example signs all `.exe` and `.dll`'s in the main build folder, and all `.dll`'s in the application's Libs folder:

```
# CodeSign: List of Files to be codesigned
Do-Codesign("*.exe")
Do-Codesign("*.dll")
Do-Codesign("${env:BUILD_WINDOWS_APP_FOLDER_NAME} Libs\*.dll")
```

WINDOWS INNOSETUP SCRIPTS
`_build/windows/innosetup_x86-32bit.iss`
`_build/windows/innosetup_x86-64bit.iss`

To build an Installer I'm going to use InnoSetup. These InnoSetup scripts are quite similar to those in the Xojo Documentation. I've tried to make them as resuable as possible for other projects by defining constants at the top. They obviously need to be modified according to the Xojo Project. Regarding build automation the interesting parts are:

- `SourceDir={#sourcepath}`
  This variable will later be set by the GitHub Action workflow when it launches InnoSetup. The Workflow knows where the files are, the InnoSetup script doesn't need to know that in a hardcoded way.
- `Signtool=CodeSignSHA1`
  `Signtool=CodeSignSHA256`
  `SignedUninstaller=yes`
  InnoSetup should sign the Installer and Uninstaller it builds. The Script only knows about our

defined CodeSignSHA1/256. How CodeSigning effectively is done will be defined in the GitHub Actions workflow later.

MACOS: XOJO2DMG

`_build/xojo2dmg/`

For macOS CodeSign, DMG creation and Notarization I'm going to use Xojo2DMG. Here is how to use it in this example project:

- Copy the contents of the folder `scripts` to the project in `_/build/xojo2dmg`
- In the Xojo IDE: Copy and paste the macOS Post Build Script to the example project.
  Make sure it runs **after** Xojo's Sign step.
- Make a modification in the script:
  `Dim sFolderScripts As String = "/_build/xojo2dmg"`
  That's because I have just copied Xojo2DMG to this location in the repository.
- Add the CodeSign Ident:
  `Dim sCODESIGN_IDENT As String = "Developer ID Application: YOUR NAME"`
  That will result in Builds and DebugRuns on the development machine to be CodeSigned the same, too. The GitHub Action Workflow will use a different way when it runs on the Build Mac.
- If you're going to use Xojo2DMG in your own projects, you'll want to customize the DMG settings, too. Just follow the explanations in the script or learn more about Xojo2DMG by visiting that repository.

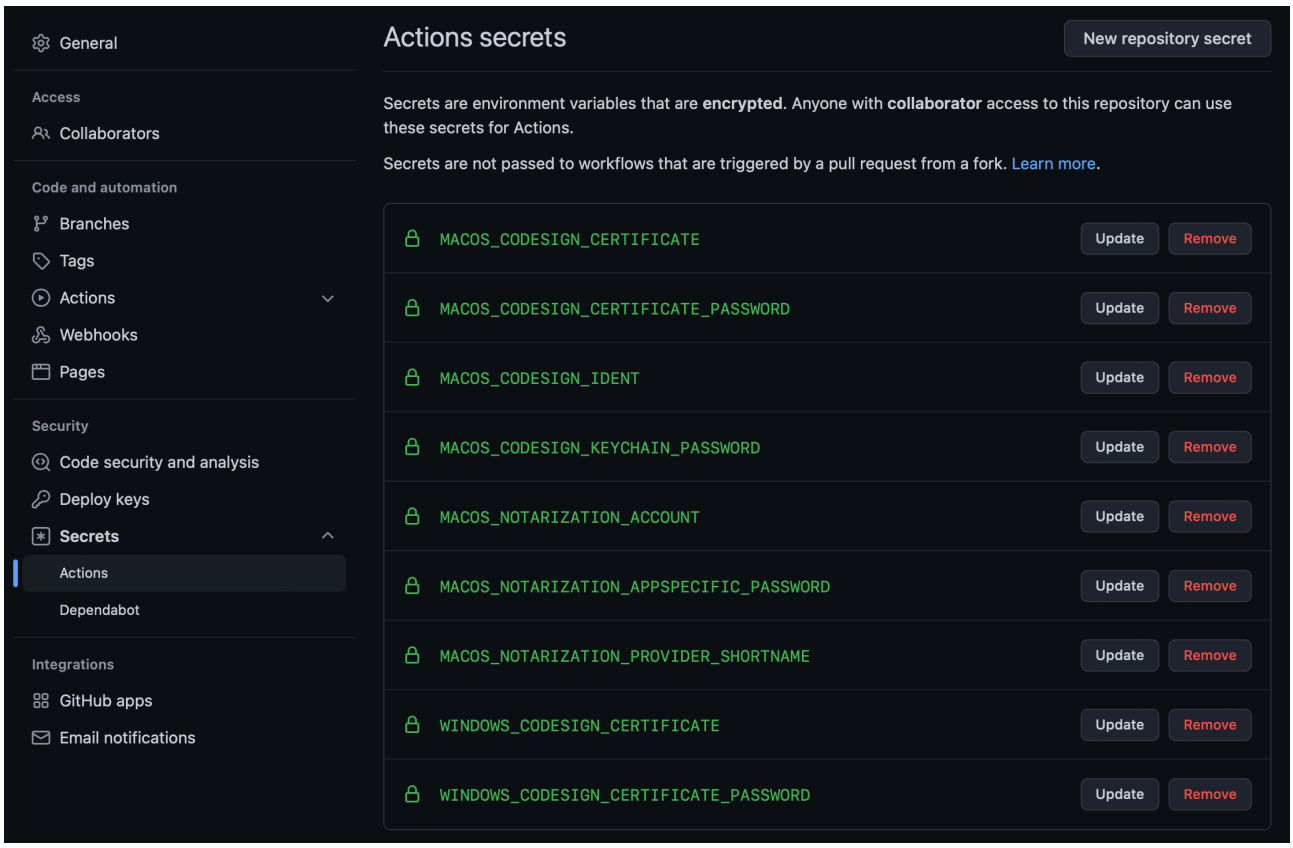
Note: The Certificates for CodeSigning and Notarization for the GitHub Actions Workflow will be set up later.

GitHub Actions: Secrets

The Xojo Project and Build resources should be ready by now. We finally can start thinking about and writing the Workflows.

How to deal with CodeSigning Certificates, it's passwords? Most importantly: don't store them (plaintext) in the code or scripts. Here is where the GitHub Actions Secrets come into play. They are encrypted environment variables which can be set in the repository settings. And workflows can use them.

Here is what needs to be set up in the GitHub Repository -> Settings -> Actions: Secrets

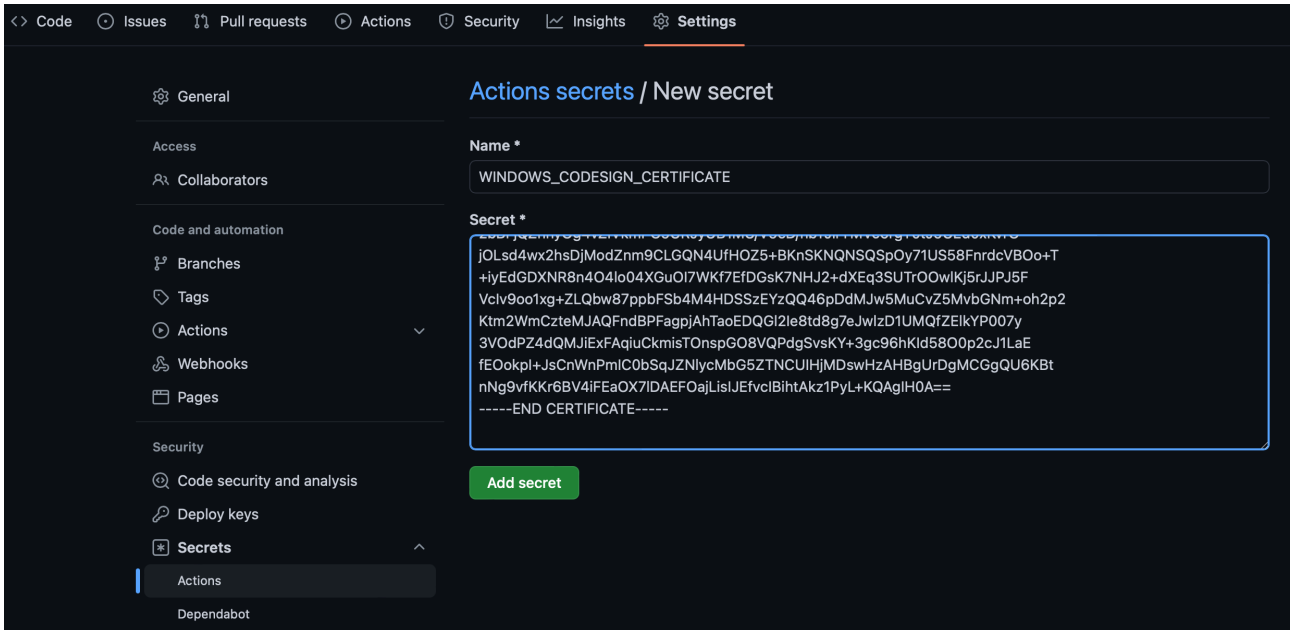


WINDOWS_CODESIGN_CERTIFICATE
This assumes that you have a CodeSigning Certificate as a `.pfx` file (and that you know it's password).

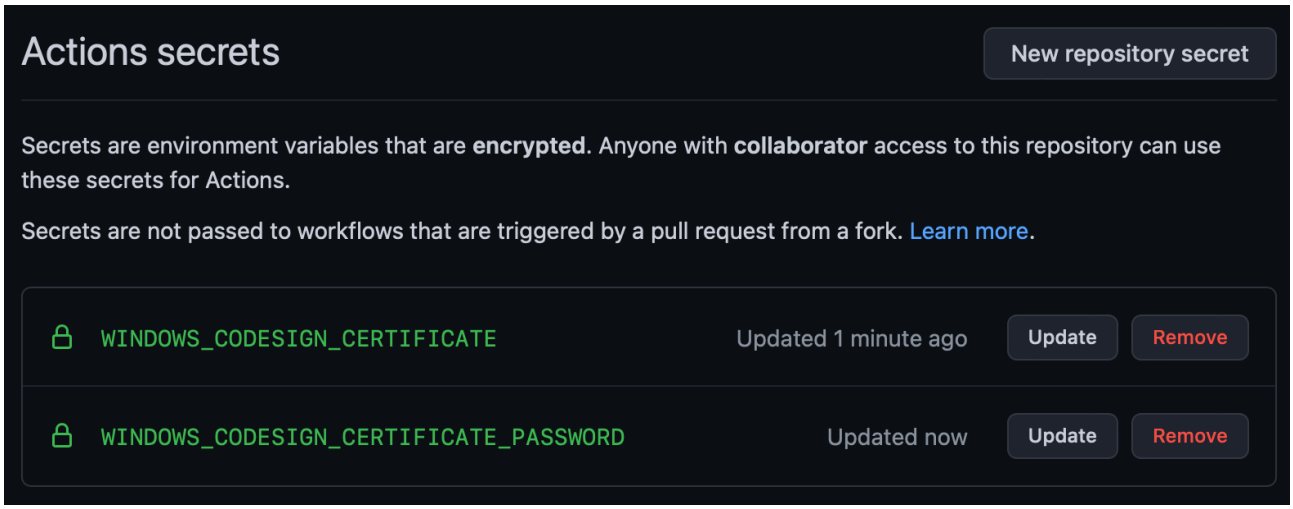
On a Windows machine, encode the `.pfx` file to Base64. To do that, use Powershell:

```
certutil -encode '.\path\certificate.pfx' '.\path\certificate_base64.txt'
```

Click 'New repository secret' and copy and paste the contents of the .txt file:



WINDOWS_CODESIGN_CERTIFICATE_PASSWORD
This one should be obvious, right? Add that secret, too:



MACOS_CODESIGN_IDENT
One obviously needs to be a registered Apple Developer. Then it's easiestto get the DeveloperID Certificate using Xcode:

Xcode -> Preferences -> Account: [Manage Certificates]

Then open "Keychain Access", look at the certificates and find the DeveloperID certificate name, e.g.: `Developer ID Application: Your Name (OrganisationKey)`

Add this full certificate name (e.g.: "Developer ID Application: Juerg Otter") to this secret - omitting the brackets *(that way it works for me)*.

MACOS_CODESIGN_KEYCHAIN_PASSWORD
Just create a Password. There is no need to remember it or save it somewhere. This will be used to set up a new (temporary) Keychain during a Workflow. This "build Keychain" will be used to import the CodeSign certificates later.

MACOS_CODESIGN_CERTIFICATE / MACOS_CODESIGN_CERTIFICATE_PASSWORD
The GitHub Docs have an article about this: Installing an Apple certificate on macOS runners

In short:

- Open "Keychain Access", find your DeveloperID certificate
- Right-click and Export as "Personal Information Exchange (`.p12`)"
  This will as for a password for this `.p12` file - enter one and save it to the GitHub Actions Secrets as `MACOS_CODESIGN_CERTIFICATE_PASSWORD`.
- Convert the saved `.p12` to Base64. To do that, use Terminal.app and type:
  `base64 ./path/ExportedCertificate.p12 | pbcopy`
  This will copy the Base64 of the certificate to the Clipboard.
  Paste this to the secret: `MACOS_CODESIGN_CERTIFICATE`

MACOS_NOTARIZATION_ACCOUNT, MACOS_NOTARIZATION_APPSPECIFIC_PASSWORD
Notarization will be done by the Xojo2DMG script. That project requires to set up a Keychain Item. However, for a GitHub Action I'm going to store the required information as secrets.

- Set up an app specific password for the AppleID that is registered as a Developer.
- Store the AppleID in the secret `MACOS_NOTARIZATION_ACCOUNT`.

- Paste the generated appspecific password in the secret `MACOS_NOTARIZATION_APPSPECIFIC_PASSWORD`.

MACOS_NOTARIZATION_PROVIDER_SHORTNAME

This information is needed if the AppleID is associated with multiple teams. It doesn't hurt to provide this even if that's not the case. This information can be looked up in Terminal.app:

- Execute the command:
  `xcrun altool --list-providers -u "my-apple-id@icloud.com"`
  Note: If asked for a password, then use the previously created appspecific password.
- Store the value of the column "ProviderShortname" to this secret.

And that's it. Now everything is ready to finally write the GitHub Actions Workflows. And they can use the just set up secrets for CodeSigning and Notarization to automate the whole process.

GitHub Actions: Workflows

Let's write the GitHub Actions Workflows. Where to start?

ABOUT WORKFLOWS

Workflows are defined by a YAML file checked in to the repository in the `.github/workflows` directory.

YAML FILES

A word of precaution when writing and editing YAML files:

- Indentation is done by spaces (not by tabs!)
- Indentation has to be exact (use always 2 spaces), or expect to get syntax errors

XOJO GITHUB ACTIONS WORKFLOWS

According to the Git Branching Strategy I want to build Beta Releases *(on Pull Requests and on Push to the main branch)*, and I want to manually create a Release with a Final Build.

This means that there will be two Workflows:

- Beta Build
- Create Release

They have something in common. Both need to build the Xojo project and do the postbuild steps. To avoid duplication I'm going to write a reusable workflow:

- Xojo *(Build and PostBuild)*

A workflow *(Beta Build | Create Release)* that uses another workflow *(Xojo Build and PostBuild)* is referred to as a "caller" workflow. The reusable workflow is a "called" workflow.

XOJO.YAML
This reusable workflow will automate the build of the Xojo Project and the Post Build steps. Again it's written as reusable as possible. It can build all Targets, but can be configured by Input parameters to e.g. just build a single target.

It's structure looks like this:

```
name: Xojo

on:
  workflow_call:
    inputs:
    secrets:
    outputs:

env:

jobs:
  build:
    name: Build
    runs-on: [self-hosted, macOS, Xojo]
    steps:
  postbuild:
    name: Post Build
    runs-on: [self-hosted, macOS, Xojo]
    needs: build
    steps:
  postbuild-windows:
    name: Post Build Windows
    runs-on: windows-latest
    needs: [build, postbuild]
    if: ${{ inputs.build-windows-x86-32bit == true || inputs.build-windows-x86-64bit == true }}
    steps:
  publish-artifacts:
    name: Publish Artifacts
    runs-on: [self-hosted, macOS, Xojo]
    needs: [build, postbuild, postbuild-windows]
    if: always()
    outputs:
    steps:
```

To explain this workflow a bit:

- `Name:` What will be shown in GitHub Actions
- `On:` This reusable workflow will be executed by a "workflow call"
  - `Inputs:` The caller workflow must set these inputs (otherwise the default values if set will be used). The inputs needed here are: Which Build Targets? Stage Code (Beta/Final)? Artifact Retention Days.
    Storage on GitHub is not endless - so one can customize how many days the Artifacts (Build Output, Beta Versions, ...) should be accessible before GitHub is going to delete them.
  - `Secrets:` A resuable workflow can't directly access the repository secrets, so the caller workflow will need to hand them over
  - `Outputs:` This workflow will build .zip's, .dmg's, .exe's. The outputs define which files are the result of the build process. When creating a final release I want the caller workflow to upload them to the release. And sending back information is done with the output's.
- `Env:` Environment Variables for this Workflow.
  This section needs to be modified according to the Xojo Project. Xojo-related information is defined here.
  - Which Xojo Version to use, and where Xojo is located on the Build Mac.
  - Which Xojo Project File to build, which PreBuild Xojo Script to use.
  - Define how many seconds to wait after launching Xojo (wait for the Xojo IDE to become ready, loading Plugins, ... set this higher if using a lot of Plugins or on a slow Build Mac). The next

IDE Communicator command will fail (and therefore the workflow run) if it's sent too early. Define how many seconds to wait between each IDE Communicator command.

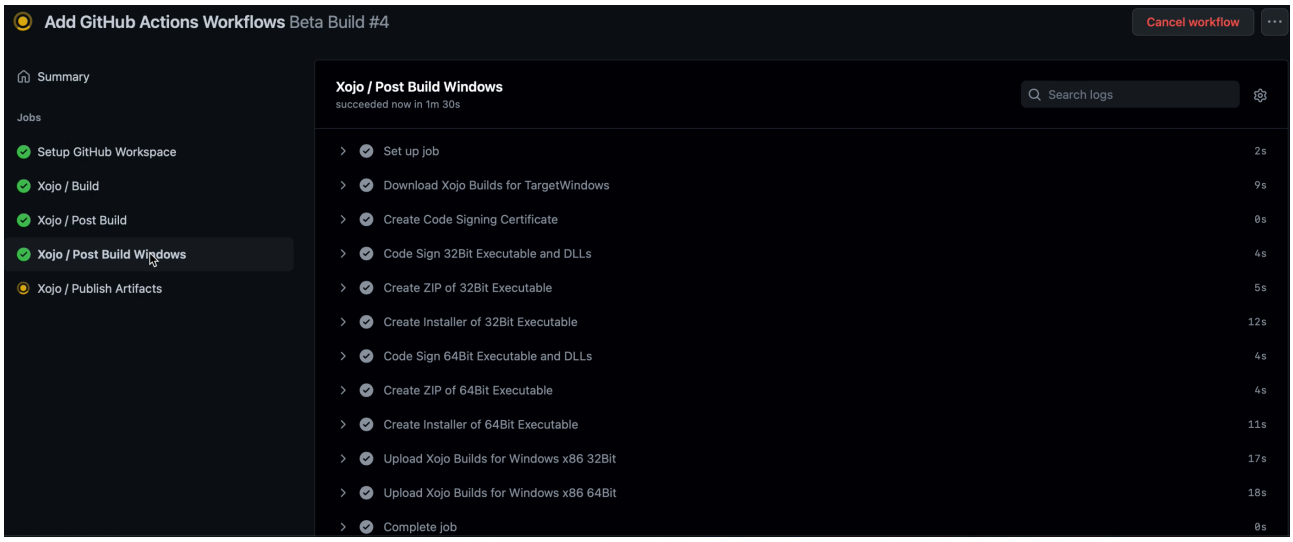
- Information about the Build Output of Xojo (Build Folder names, built application Names, ...)
  The workflow needs to know what files are going to be post-processed.
  Depending on the used Xojo Version this may change (e.g. they have renamed the format of the main Builds folder at some Xojo Version...)
- Information about PostBuild (where and which InnoSetup- and CodeSignScripts) and expected results (name of Windows Installers, macOS application .dmg).
  Note: the macOS PostBuild script `call_xojo2dmg.sh` will be written dynamically by the Xojo2DMG PostBuild Script.

- `Jobs:` All Jobs (each with multiple steps) to be executed by this workflow.
  - `Runs-on: [self-hosted, macOS, Xojo]`
    Remember the Label to be configured on the Build Mac? This makes sure the job is being executed on the self-hosted BuildMac with Xojo installed.
  - `Needs:` ...the previous jobs
    In order to run the jobs one-after-another (and certainly never in parallel).
  - `If:` only execute this job if...
    The "Post Build Windows" job will only be executed if the Input Parameters define that a Windows Build is needed.
    The "Publish Artifacts" job will run "always". That's because if Windows is not being built, the above job 'postbuild-windows' won't be executed. Conditional '`needs:`' are not available. That's why this job needs to run always. But it will fail if the required outputs don't exist.

What are the jobs doing?

- **Build**
  Launch Xojo and use the XojoIDECommunicator to execute the PreBuild Xojo Script, and then build all configured BuildTargets. Finally Quit Xojo.
- **Postbuild**
  Calls the Linux PostBuild Scripts, which will create a .tgz.
  Sets up a Keychain for CodeSigning/Notarization the macOS application, and calls Xojo2DMG to perform this. Then the temporary KeyChain will be removed again.
  The Windows Builds (such as produced by Xojo) will be upladed to GitHub as an Artifact... one can't CodeSign and create Windows Installers on the BuildMac.
- **Postbuild Windows**
  This Job will run on a GitHub provided virtual machine. Those come with SignTool, InnoSetup installed. The CodeSign Certificate `.pfx` will be written (it's content is stored in the secrets - remember?). Using PowerShell commands the provided CodeSigning scripts will be executed. Then InnoSetup will be launched. If you look at the code you'll notice that I'm defining the "`CodeSign SHA1/256`" for InnoSetup here, so that InnoSetup can write a code-signed installer.
  At the end of this job, the signed Installers, and the created `.zip` containing the code-signed executable will be uploaded to GitHub as an Artifact - so that our BuildMac can download it for the next job(s).
  Note: the artifact retention days is set to 1. That's because I don't want to use a lot of storage on GitHub for this temporary files... and it's not quite easy to delete artifacts in a workflow - but at least GitHub will delete it after 1 day.
- **Publish Artifacts**
  This job double-checks if all expected build outputs (`.tgz, .dmg, .zip, .exe`) are available. Otherwise, it will fail (by exiting with an ErrorCode > 0).
  It sets the output parameters (which the Create Release Workflow can use) and uploads all final built files to an Artifact named "Builds".

The following ScreenShot shows what's happening in the Post Build Windows job:



BETA-BUILD.YAML
This workflow will build a Beta version of the Xojo Project on Pull Request and on pushes to the main branch.

It's structure looks like this:

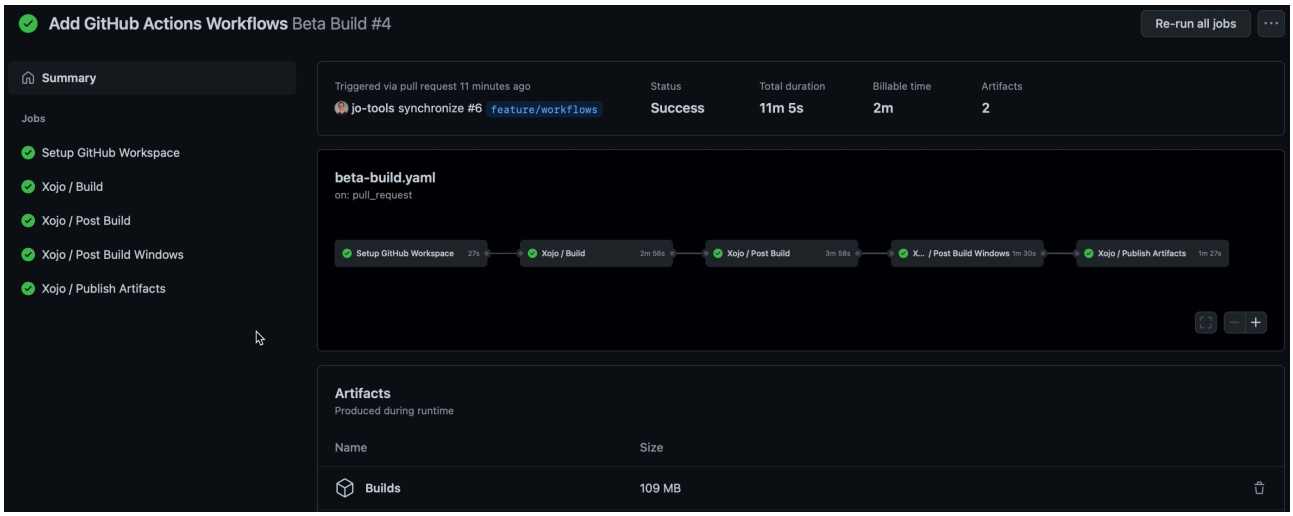
```
name: Beta Build

on:
  workflow_dispatch:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  setup:
    name: Setup GitHub Workspace
    runs-on: [self-hosted, macOS, Xojo]
    outputs:
    steps:
      - name: Cleanup GitHub Workspace
      - name: Check out Repository
      - name: Setup Variables
  xojo:
    name: Xojo
    uses: ./.github/workflows/xojo.yaml
    needs: setup
    with:
    secrets:
```

To explain this workflow a bit:

- `Name`: What will be shown in GitHub Actions
- `On`: The workflow will run on Pushes to the main branch, on Pull Requests targeting the main branch. And the workflow could be started manually (workflow_dispatch).
- `Jobs`: All Jobs (each with multiple steps) to be executed by this workflow.
  - **Setup**
    Runs on the Build Mac with Xojo and outputs the number of days to keep artifacts (2 for Pull Requests, 7 for builds in main branch).

- **Xojo**
  This job will call the reusable workflow to build the Xojo Project.
  In the section "`with`" can be configured which BuildTargets should be built, the stage code and the artifacts retention days is handed over.
  This workflow can access the repository secrets for CodeSigning and Notarization. It hands over the required secrets to the reusable Xojo workflow.

Once this workflow has finished running, there will be an artifact `Builds.zip` available which contains all built products of the Xojo application. It can be downloaded and used for Beta testing.



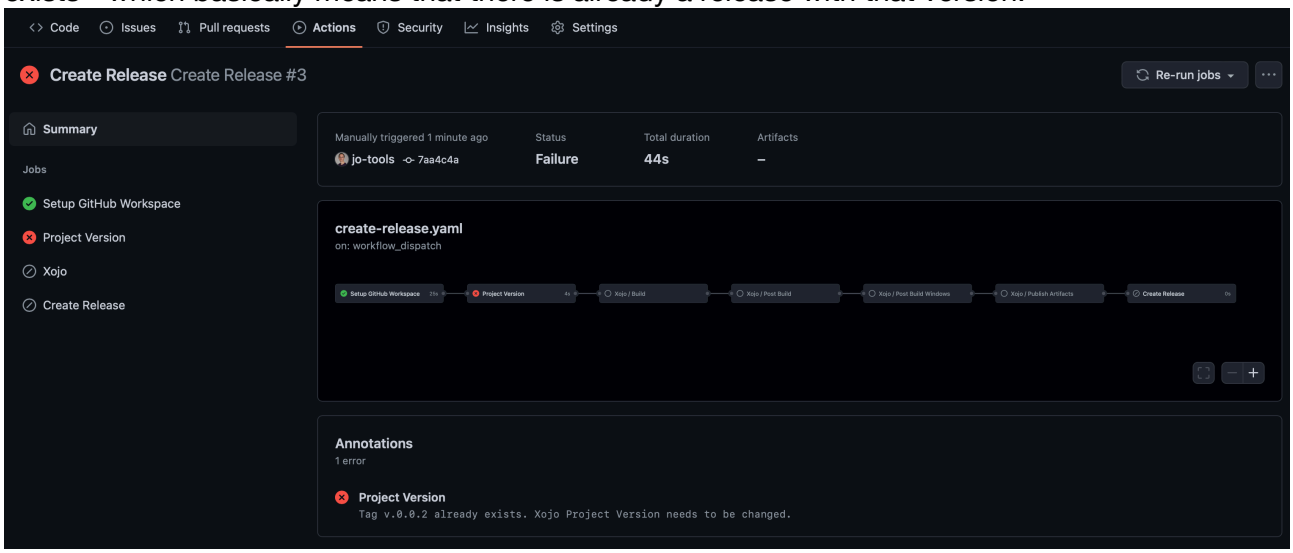
CREATE-RELEASE.YAML
This workflow will create a Release in the GitHub repository. Each release is being tagged with a Git tag.

It's structure looks like this:

```
name: Create Release

on:
  workflow_dispatch:
    branches:
      - main

env:
  XOJO_PROJECT_FILE:

jobs:
  project-version:
    name: Project Version
    runs-on: [self-hosted, macOS, Xojo]
    outputs:
      version-tag: ${{ steps.project-version.outputs.version-tag }}
    steps:
      - name: Cleanup Github Workspace
      - name: Check out repository code
      - name: Get Xojo Project Version
      - name: Check Release Version Tag

  xojo:
    name: Xojo
    uses: ./.github/workflows/xojo.yaml
    needs: project-version
    with:
    secrets:

  create-release:
    name: Create Release
    runs-on: [self-hosted, macOS, Xojo]
    needs: [project-version, xojo]
    steps:
      - name: Create release
      - name: Upload release assets
```

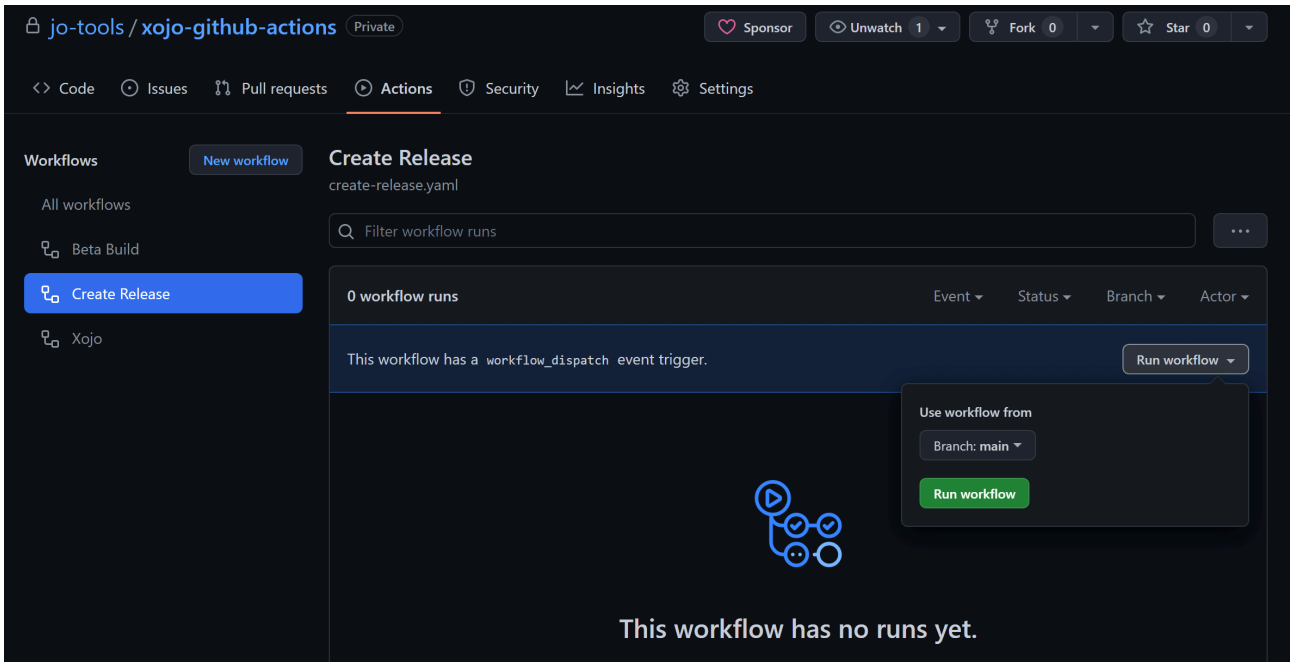
To explain this workflow a bit:

- `Name:` What will be shown in GitHub Actions
- `On:` The workflow has to be started manually (workflow_dispatch) on the main branch.
- `Env:` One Job will need to know the Xojo Project file, so we have to set this as an environment variable.
- `Jobs:` All Jobs (each with multiple steps) to be executed by this workflow.
  - **Setup**
    Cleanup Workspace, Check out Repository
  - **Project Version**
    This will read the Major/Minor/Bugversion of the Xojo Project. That version is being used as a Git tag and might look like this: `v.1.0.0`. So there is a step which checks if this Tag already exists - which basically means that there is already a release with that version.

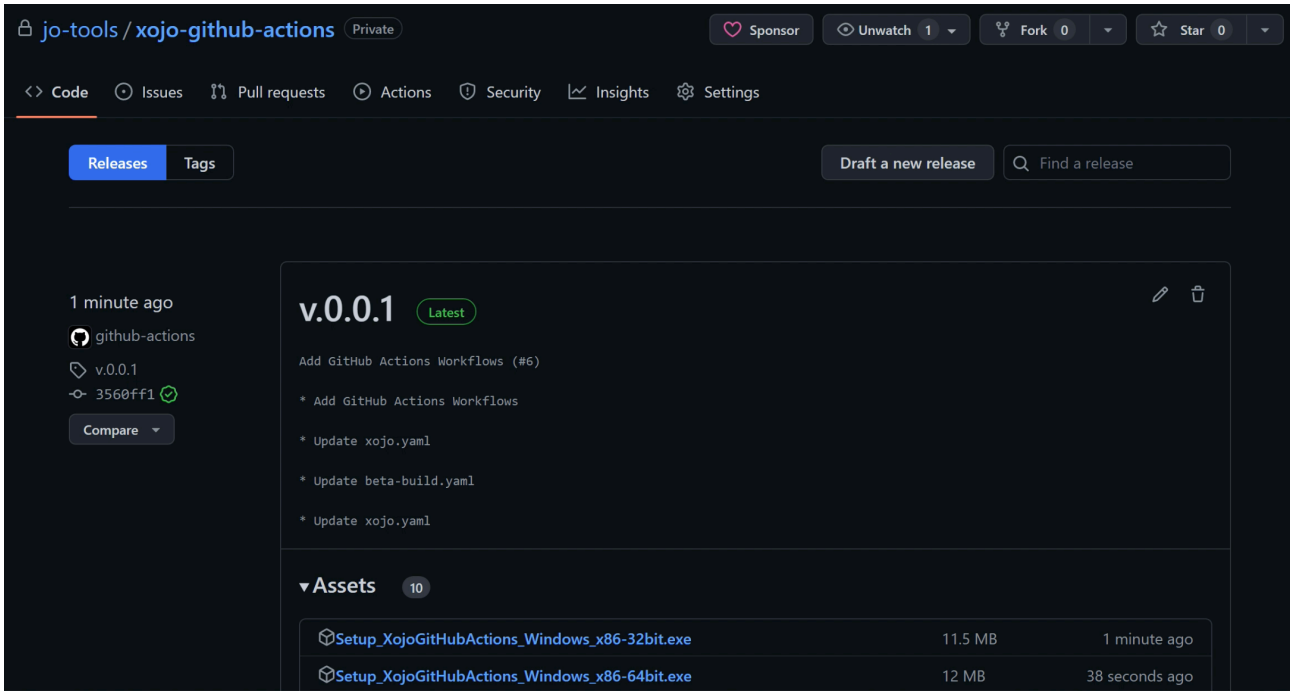


  - **Xojo**
    This job will call the reusable workflow to build the Xojo Project.
  - **Create Release**
    Will use the outputs of the reusable Xojo workflow, create a release and upload all release assets.

This workflow can be run manually from GitHub -> Actions: Create Release -> Run Workflow



The final builds and the post processed Assets (Installers) will then be available in the Releases of the GitHub Repository:



How to use in your own Xojo Projects?

Look at the Workflows and Build Scripts in the Repository GitHub jo-tools/xojo-github-actions as an example. They show all bits and pieces to make this work. You'll need to edit a couple of pieces according to your Project (Environment variables in Workflows, InnoSetup Scripts, Secrets, ...), such as explained above.

However, there is no right and wrong. What is shown in this example is not the only way of doing things. Maybe it doesn't make sense for you to store Beta Builds as Artifacts on GitHub (especially since storage is limited). You could modify the Workflow to save it to a network share instead. Or you don't want to use GitHub's Windows runner, but use an own machine. Setup a self-hosted runner then - and you can even get rid of uploading the Artifact which the Windows machine needs to process by saving it in some shared location. The same goes for Releases and Final Builds. You could modify the Workflow to upload the files to your webserver.

I know that many Xojo Developers are using AppWrapper for macOS Post Build and Notarization. They'll need to figure out a way how to replace the call to Xojo2DMG with other tools.

At the very least I hope this article has provided helpful hints, ideas and approaches. Should you consider using GitHub Actions with your Xojo projects, I hope this gives a head start.

A POSSIBLE IMPROVEMENT
There is one thing I don't quite like in this example...

The Xojo Plugins are important to the built application. For this reason, the Plugins should be under source control in the repository - and not only on the Build Mac. Quite similar to why build scripts should be under source control in the repository, too.

This should be relatively easy to implement. Once the Plugins are in the repository, they get checked out on the Build Mac. Before launching the Xojo IDE the Workflow would need to replace the Xojo Plugins with the ones from the current commit.

It's just that this example is already quite long - so I'll leave this up to you.

That's all Folks!

I hope this brief introduction of how Build Automation for Xojo built Application can be used with GitHub Actions Workflows has been helpful to some, food for thought to others.

Do you like it? Does it help you? Has it saved you time and money?

You're welcome - it's free...

If you want to say thanks I appreciate a message or a small donation.

- Website: https://www.jo-tools.ch/
- Contact: xojo@jo-tools.ch
- PayPal: https://paypal.me/jotools

Jürg Otter is a long term user of Xojo and working for CM Informatik AG. Their Application CMI LehrerOffice is a Xojo Design Award Winner 2018. In his leisure time Jürg provides some bits and pieces for Xojo Developers.