Cross-Platform Compilation

Tauri relies heavily on native libraries and toolchains, so meaningful cross-compilation is **not possible** at the current moment. The next best option is to compile utilizing a CI/CD pipeline hosted on something like GitHub Actions, Azure Pipelines, GitLab, or other options. The pipeline can run the compilation for each platform simultaneously making the compilation and release process much easier.

For an easy setup, we currently provide Tauri Action, a GitHub Action that runs on all the supported platforms, compiles your software, generates the necessary artifacts, and uploads them to a new GitHub release.

Tauri GitHub Action

Tauri Action leverages GitHub Actions to simultaneously build your application as a Tauri native binary for macOS, Linux, and Windows, and automates creating a GitHub release.

This GitHub Action may also be used as a testing pipeline for your Tauri app, guaranteeing compilation runs fine on all platforms for each pull request sent, even if you don't wish to create a new release.

CODE SIGNING

To setup code signing for both Windows and macOS on your workflow, follow the specific guide for each platform:

- Windows Code Signing with GitHub Actions
- macOS Code Signing with GitHub Actions

Getting Started

To set up Tauri Action you must first set up a GitHub repository. You can use this action on a repo that doesn't have Tauri configured since it automatically initializes Tauri before building and configuring it to use your artifacts.

Go to the Actions tab on your GitHub project and choose "New workflow", then choose "Set up a workflow yourself". Replace the file with the Tauri Action production build workflow example. Alternatively, you may set up the workflow based on the example at the bottom of this page

Configuration

You can configure Tauri with the `configPath`, `distPath` and `iconPath` options. See the actions Readme for details.

When your app isn't on the root of the repo, use the `projectPath` input.

You may modify the workflow name, change the triggers, and add more steps such as `npm run lint` or `npm run test`. The important part is that you keep the below line at the end of the workflow, since this runs the build script and releases the artifacts:

```
- uses: tauri-apps/tauri-action@v0
```

How to Trigger

The release workflow in the README examples linked above is triggered by pushes on the "release" branch. The action automatically creates a tag and title for the GitHub release using the application version specified in `tauri.config.json`.

You can also trigger the workflow on the push of a version tag such as "app-v0.7.0". For this you can change the start of the release workflow:

```
name: publish
on:
  push:
    tags:
      - 'app-v*'
  workflow_dispatch:
```

Example Workflow

Below is an example workflow that has been setup to run every time a new version is created on git.

This workflow sets up the environment on Windows, Ubuntu, and macOS latest versions. Note under `jobs.release.strategy.matrix` the platform array which contains `macos-latest`, `ubuntu-20.04`, and `windows-latest`.

The steps this workflow takes are:

1. Checkout the repository using `actions/checkout@v3`

2. Set up Node LTS and a cache for global npm/yarn/pnpm package data using `actions/setup-node@v3`.
3. Set up Rust and a cache for the `target/` folder using `dtolnay/rust-toolchain@stable` and `swatinem/rust-cache@v2`.
4. Installs all the dependencies and run the build script (for the web app).
5. Finally, it uses `tauri-apps/tauri-action@v0` to run `tauri build`, generate the artifacts, and create the GitHub release.

```
name: Release
on:
  push:
    tags:
      - 'v*'
  workflow_dispatch:

jobs:
  release:
    permissions:
      contents: write
    strategy:
      fail-fast: false
      matrix:
        platform: [macos-latest, ubuntu-20.04, windows-latest]
    runs-on: ${{ matrix.platform }}

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Install dependencies (ubuntu only)
        if: matrix.platform == 'ubuntu-20.04'
        # You can remove libayatana-appindicator3-dev if you don't use the system tray feature.
        run: |
          sudo apt-get update
          sudo apt-get install -y libgtk-3-dev libwebkit2gtk-4.0-dev libayatana-appindicator3-
dev librsvg2-dev

      - name: Rust setup
        uses: dtolnay/rust-toolchain@stable

      - name: Rust cache
        uses: swatinem/rust-cache@v2
        with:
          workspaces: './src-tauri -> target'

      - name: Sync node version and setup cache
        uses: actions/setup-node@v3
        with:
          node-version: 'lts/*'
          cache: 'yarn' # Set this to npm, yarn or pnpm.

      - name: Install frontend dependencies
        # If you don't have `beforeBuildCommand` configured you may want to build your frontend
here too.
        run: yarn install # Change this to npm, yarn or pnpm.

      - name: Build the app
        uses: tauri-apps/tauri-action@v0

        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          tagName: ${{ github.ref_name }} # This only works if your workflow triggers on new
tags.
          releaseName: 'App Name v__VERSION__' # tauri-action replaces \_\_VERSION\_\_ with the
app version.
          releaseBody: 'See the assets to download and install this version.'
          releaseDraft: true
          prerelease: false
```

GitHub Environment Token

The GitHub Token is automatically issued by GitHub for each workflow run without further configuration, which means there is no risk of secret leakage. This token however only has read permissions by default and you may get a "Resource not accessible by integration" error when running the workflow. If this happens, you may need to add write permissions to this token. To do this go to your GitHub Project Settings, and then select Actions, scroll down to "Workflow permissions" and check "Read and write permissions".

You can see the GitHub Token being passed to the workflow below:

```
env:
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Usage Notes

Make sure to check the documentation for GitHub Actions to understand better how this workflow works. Take care to read the Usage limits, billing, and administration documentation for GitHub Actions. Some project templates may already implement this GitHub action workflow, such as tauri-svelte-template. You can use this action on a repo that doesn't have Tauri configured. Tauri automatically initializes before building and configuring it to use your web artifacts.

Experimental: Build Windows apps on Linux and macOS

Tauri v1.3 added a new Windows installer type based on the NSIS installer framework. In contrast to WiX, NSIS itself can also work on Linux and macOS which makes it possible to build many Tauri apps on non-Windows hosts. Note that this is currently considered highly experimental and may not work on every system or for every project. Therefore it should only be used as a last resort if local VMs or CI solutions like GitHub Actions don't work for you. **Note that, at this time, signing cross-platform builds is currently unsupported.**

Since Tauri officially only supports the MSVC Windows target, the setup is a bit more involved.

First, make sure all your Tauri dependencies are at least version 1.3, check out the dependency update guide if you're not sure how.

Install NSIS

Some Linux distributions have NSIS available in their repositories, for example on Ubuntu you can install NSIS by running this command:

Ubuntu

```
sudo apt install nsis
```

But on many other distributions you have to compile NSIS yourself or download Stubs and Plugins manually that weren't included in the distro's binary package. Fedora for example only provides the binary but not the Stubs and Plugins:

Fedora

```
sudo dnf in mingw64-nsis
wget https://github.com/tauri-apps/binary-releases/releases/download/nsis-3/nsis-3.zip
unzip nsis-3.zip
sudo cp nsis-3.08/Stubs/* /usr/share/nsis/Stubs/
sudo cp -r nsis-3.08/Plugins/** /usr/share/nsis/Plugins/
```

On macOS you will need Homebrew to install NSIS:

macOS

```
brew install nsis
```

Install LLVM and the LLD Linker

Since the default Microsoft linker only works on Windows we will also need to install a new linker. To compile the Windows Resource file which is used for setting the app icon among other things we will also need the `llvm-rc` binary which is part of the LLVM project.

Ubuntu

```
sudo apt install lld llvm
```

macOS

```
brew install llvm
```

On macOS you also have to add `/opt/homebrew/opt/llvm/bin` to your `$PATH` as suggested in the install output.

Install the Windows Rust target

Assuming you're building for 64-bit Windows systems:

```
rustup target add x86_64-pc-windows-msvc
```

Install the Windows SDKs

To get the Windows SDKs required by the msvc target we will use the xwin project:

```
cargo install xwin
```

Then you can use the `xwin` CLI to install the needed files to a location of your choice. Remember the location, we will need it in the next step. In this guide we will create a `.xwin` directory in the Home directory.

```
xwin splat --output ~/.xwin
```

If that fails with an error message like this:

```
Error: failed to splat Microsoft.VC.14.29.16.10.CRT.x64.Desktop.base.vsix

Caused by:
    0: unable to symlink from .xwin/crt/lib/x86_64/LIBCMT.lib to libcmt.lib
    1: File exists (os error 17)
```

you can try adding the `--disable-symlinks` flag to the command:

```
xwin splat --output ~/.xwin --disable-symlinks
```

Now, to make the Rust compiler use these files, you first have to create a `.cargo` directory in your project and create a `config.toml` file in it with the following content. Make sure to change the paths accordingly.

.cargo/config.toml

```toml
[target.x86_64-pc-windows-msvc]
linker = "lld"
rustflags = [
  "-Lnative=/home/username/.xwin/crt/lib/x86_64",
  "-Lnative=/home/username/.xwin/sdk/lib/um/x86_64",
  "-Lnative=/home/username/.xwin/sdk/lib/ucrt/x86_64"
]
```

Keep in mind that this file is specific to your machine so we don't recommend checking it into git if your project is public or will be shared with anyone.

Building the App

NOTE

If your application has dependencies to C libraries such as `ring` or `libsqlite3-sys`, building your application cross-platform gets a little trickier, as you will also need to set up the appropriate environment variables and packages to cross-compile those C libraries on your system. These may include setting the `CC`, `CXX`, `AR` and other environment variables, although this depends heavily on the set up of your build environment. Please refer to the documentation of the libraries you are using for more information about any additional configuration for cross-compilation.

Now it should be as simple as adding the target to the `tauri build` command:

npm Yarn pnpm **Cargo**

```
$ cargo tauri build --target x86_64-pc-windows-msvc
```

The build output will then be in `target/x86_64-pc-windows-msvc/release/bundle/nsis/`.

Edit this page

Last updated on ***Oct 13, 2023***