Reducing App Size

With Tauri, we are working to reduce the environmental footprint of applications by using fewer system resources where available, providing compiled systems that don't need runtime evaluation, and offering guides so that engineers can go even smaller without sacrificing performance or security. By saving resources we are doing our part to help you help us save the planet -- which is the only bottom line that companies in the 21st Century should care about.

So if you are interested in learning how to improve your app size and performance, read on!

You can't improve what you can't measure

Before you can optimize your app, you need to figure out what takes up space in your app! Here are a couple of tools that can assist you with that:

- `cargo-bloat` - A Rust utility to determine what takes the most space in your app. It gives you an excellent, sorted overview of the most significant Rust functions.
- `cargo-expand` - Macros make your rust code more concise and easier to read, but they are also hidden size traps! Use `cargo-expand` to see what those macros generate under the hood.
- `rollup-plugin-visualizer` - A tool that generates beautiful (and insightful) graphs from your rollup bundle. Very convenient for figuring out what JavaScript dependencies contribute to your final bundle size the most.
- `rollup-plugin-graph` - You noticed a dependency included in your final frontend bundle, but you are unsure why? `rollup-plugin-graph` generates Graphviz-compatible visualizations of your entire dependency graph.

These are just a couple of tools that you might use. Make sure to check your frontend bundlers plugin list for more!

Checklist

1. Minify Javascript
2. Optimize Dependencies

3. Optimize Images
4. Remove Unnecessary Custom Fonts
5. Allowlist Config
6. Rust Build-time Optimizations
7. Stripping
8. UPX

Minify JavaScript

JavaScript makes up a large portion of a typical Tauri app, so it's important to make the JavaScript as lightweight as possible.

You can choose from a plethora of JavaScript bundlers; popular choices are Vite, webpack, and rollup. All of them can produce minified JavaScript if configured correctly, so consult your bundler documentation for specific options. Generally speaking, you should make sure to:

Enable tree shaking

This option removes unused JavaScript from your bundle. All popular bundlers enable this by default.

Enable minification

Minification removes unnecessary whitespace, shortens variable names, and applies other optimizations. Most bundlers enable this by default; a notable exception is rollup, where you need plugins like rollup-plugin-terser or rollup-plugin-uglify.

Note: You can use minifiers like terser and esbuild as standalone tools.

Disable source maps

Source maps provide a pleasant developer experience when working with languages that compile to JavaScript, such as TypeScript. As source maps tend to be quite large, you must disable them when building for production. They have no benefit to your end-user, so it's effectively dead weight.

Optimize Dependencies

Many popular libraries have smaller and faster alternatives that you can choose from instead.

Most libraries you use depend on many libraries themselves, so a library that looks inconspicuous at first glance might add **several megabytes** worth of code to your app.

You can use Bundlephobia to find the cost of JavaScript dependencies. Inspecting the cost of Rust dependencies is generally harder since the compiler does many optimizations.

If you find a library that seems excessively large, Google around, chances are someone else already had the same thought and created an alternative. A good example is Moment.js and its many alternatives.

But keep in mind: **The best dependency is no dependency**, meaning that you should always prefer language builtins over 3rd party packages.

Optimize Images

According to the Http Archive, images are the biggest contributor to website weight. So if your app includes images or icons, make sure to optimize them!

You can choose between a variety of manual options (GIMP, Photoshop, Squoosh) or plugins for your favorite frontend build tools (vite-imagetools, vite-plugin-imagemin, image-minimizer-webpack-plugin).

Do note that the `imagemin` library most of the plugins use is officially unmaintained.

Use Modern Image Formats

Formats such as `webp` or `avif` offer size reductions of **up to 95%** compared to jpeg while maintaining excellent visual accuracy. You can use tools such as Squoosh to try different formats on your images.

Size Images Accordingly

No one appreciates you shipping the 6K raw image with your app, so make sure to size your image accordingly. Images that appear large on-screen should be sized larger than images that take up less screen space.

Don't Use Responsive Images

In a Web Environment, you are supposed to use Responsive Images to load the correct image size for each user dynamically. Since you are not dynamically distributing images over the web, using Responsive Images only needlessly bloats your app with redundant copies.

Remove Metadata

Images that were taken straight from a camera or stock photo side often include metadata about the camera and lens model or photographer. Not only are those wasted bytes, but metadata properties can also hold potentially sensitive information such as the time, day, and location of the photo.

Remove Unnecessary Custom Fonts

Consider not shipping custom fonts with your app and relying on system fonts instead. If you must ship custom fonts, make sure they are in modern, optimized formats such as `woff2`.

Fonts can be pretty big, so using the fonts already included in the Operating System reduces the footprint of your app. It also avoids FOUT (Flash of Unstyled Text) and makes your app feel more "native" since it uses the same font as all other apps.

If you must include custom fonts, make sure you include them in modern formats such as `woff2` as those tend to be much smaller than legacy formats.

Use so-called **"System Font Stacks"** in your CSS. There are a number of variations, but here are 3 basic ones to get you started:

Sans-Serif

```
font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Helvetica, Arial,
  sans-serif, 'Apple Color Emoji', 'Segoe UI Emoji';
```

Serif

```
font-family: Iowan Old Style, Apple Garamond, Baskerville, Times New Roman, Droid
    Serif, Times, Source Serif Pro, serif, Apple Color Emoji, Segoe UI Emoji, Segoe
    UI Symbol;
```

Monospace

```
font-family: ui-monospace, SFMono-Regular, SF Mono, Menlo, Consolas, Liberation
    Mono, monospace;
```

Allowlist Config

You can reduce the size of your app by only enabling the Tauri API features you need in the `allowlist` config.

The `allowlist` config determines what API features to enable; disabled features will **not be compiled into your app**. This is an easy way of shedding some extra weight.

An example from a typical `tauri.conf.json`:

```
{
  "tauri": {
    "allowlist": {
      "all": false,
      "fs": {
        "writeFile": true
      },
      "shell": {
        "execute": true
      },
      "dialog": {
        "save": true
      }
    }
  }
}
```

Rust Build-Time Optimizations

Configure your cargo project to take advantage of Rusts size optimization features. Why is a rust executable large ? provides an excellent explanation of why this matters and an in-depth walkthrough. At the same time, Minimizing Rust Binary Size is more up-to-date and has a couple of extra recommendations.

Rust is notorious for producing large binaries, but you can instruct the compiler to optimize the final executable's size.

Cargo exposes several options that determine how the compiler generates your binary. The "recommended" options for Tauri apps are these:

```
[profile.release]
panic = "abort" # Strip expensive panic clean-up logic
codegen-units = 1 # Compile crates one after another so the compiler can optimize better
lto = true # Enables link to optimizations
opt-level = "s" # Optimize for binary size
strip = true # Remove debug symbols
```

NOTE

There is also `opt-level = "z"` available to reduce the resulting binary size. `"s"` and `"z"` can sometimes be smaller than the other, so test it with your application!

> We've seen smaller binary sizes from `"s"` for Tauri example applications, but real-world applications can always differ.

For a detailed explanation of each option and a bunch more, refer to the Cargo books Profiles section.

Disable Tauri's Asset Compression

By default, Tauri uses Brotli to compress assets in the final binary. Brotli embeds a large (~170KiB) lookup table to achieve great results, but if the resources you embed are smaller than this or compress poorly, the resulting binary may be bigger than any savings.

Compression can be disabled by setting `default-features` to `false` and specifying everything except the `compression` feature:

```
[dependencies]
tauri = { version = "...", features = ["objc-exception", "wry"], default-features = false }
```

Unstable Rust Compression Features

> **CAUTION**
>
> The following suggestions are all unstable features and require a nightly toolchain. See the Unstable Features documentation for more information on what this involves.

The following methods involve using unstable compiler features and require the rust nightly toolchain. If you don't have the nightly toolchain + `rust-src` nightly component added, try the following:

```
$ rustup toolchain install nightly
$ rustup component add rust-src --toolchain nightly
```

To tell Cargo that the current project uses the nightly toolchain, we will create an Override File at the root of our project called `rust-toolchain.toml`. This file will contain the following:

rust-toolchain.toml

```
[toolchain]
channel = "nightly-2023-01-03" # The nightly release to use, you can update this to the most recent one if you want
profile = "minimal"
```

The Rust Standard Library comes precompiled. This means Rust is faster to install, but also that the compiler can't optimize the Standard Library. You can apply the optimization options for the rest of your binary + dependencies to the std with an unstable flag. This flag requires specifying your target, so know the target triple you are targeting.

```
$ cargo tauri build --target <Target triple to build for> -- -Z build-std
```

If you are using `panic = "abort"` in your release profile optimizations, you need to make sure the `panic_abort` crate is compiled with std. Additionally, an extra std feature can further reduce the binary size. The following applies to both:

```
$ cargo tauri build --target <Target triple to build for> -- -Z build-std=std,panic_abort -Z
build-std-features=panic_immediate_abort
```

See the unstable documentation for more details about `-Z build-std` and `-Z build-std-features`.

Stripping

Use strip utilities to remove debug symbols from your compiled app.

Your compiled app includes so-called "Debug Symbols" that include function and variable names. Your end-users will probably not care about Debug Symbols, so this is a pretty surefire way to save some bytes!

The easiest way is to use the famous `strip` utility to remove this debugging information.

```
$ strip target/release/my_application
```

See your local `strip` manpage for more information and flags that can be used to specify what information gets stripped out from the binary.

> **INFO**
>
> Rust 1.59 now has a builtin version of `strip`! It can be enabled by adding the following to your `Cargo.toml`:
>
> ```
> [profile.release]
> strip = true  # Automatically strip symbols from the binary.
> ```

UPX

UPX, **Ultimate Packer for eXecutables**, is a dinosaur amongst the binary packers. This 23-year-old, well-maintained piece of kit is GPL-v2 licensed with a pretty liberal usage declaration. Our understanding of the licensing is that you can use it for any purposes (commercial or otherwise) without needing to change your license unless you modify the source code of UPX.

Maybe your target audience has very slow internet, or your app needs to fit on a tiny USB stick, and all the above steps haven't resulted in the savings you need. Fear not, as we have one last trick up our sleeves:

UPX compresses your binary and creates a self-extracting executable that decompresses itself at runtime.

> **CAUTION**
>
> You should know that this technique might flag your binary as a virus on Windows and macOS - so use it at your own discretion, and as always, validate with Frida and do real distribution testing!

Usage on macOS

```
brew install upx
yarn tauri build
upx --ultra-brute src-tauri/target/release/bundle/macos/app.app/Contents/macOS/app

                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2018
UPX 3.95        Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 26th 2018

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
    963140 ->    274448   28.50%   macho/amd64   app
```

Edit this page

Last updated on ***Mar 25, 2023***