Electron vs. Tauri

Eric Richardson

November 13, 2025

INTEGRATION 6 min read

The Dolt Workbench is an open-source SQL workbench supporting MySQL, Postgres, Dolt, and Doltgres databases. We built the workbench using Electron, which is a popular framework that allows you to convert web apps built with traditional web technologies like HTML, CSS, and Javascript to desktop applications. Since the workbench shares much in common with DoltHub and Hosted Dolt, the architecture is very similar to those products. That is, the workbench uses Next.js for the frontend with an additional GraphQL layer that handles database interactions. For this reason, it made a lot of sense to use Electron to get the desktop version of our application up and running.

That said, Electron comes with a few rather significant drawbacks, and those drawbacks have started to become more apparent as the workbench has matured. Because of this, I spent some time exploring Tauri, a newer framework that supports the same web-to-desktop use case as Electron. In this article, we'll discuss how well Electron and Tauri integrate with the workbench, and weigh some pros and cons between the two frameworks.

Next.js Support

Next.js doesn't translate very cleanly to a desktop application context. This is primarily due to the framework's architecture around server-side rendering and API routing features. In a desktop app, there's no application server interacting with a client; we just need to render HTML, CSS, and JavaScript in a window. For these reasons, Electron only loosely supports Next.js applications. That's not to say you can't build an Electron app with Next.js, but it requires some workarounds to make it function properly. One of the more popular workarounds is a project called **Nextron**, which aims to wire Next.js applications to the Electron framework and streamline the build process. This is the project we use for the workbench. The issue is that, at the time of writing, it appears that Nextron is no longer being maintained, and we started hitting a few bugs with it.

Tauri is largely frontend-framework agnostic. For Next, specifically, you still can't use the server-side features, but Tauri makes the integration process much simpler by relying on Next's static-site generation capabilities. To make a Next app work with Tauri, you just need to set `output: 'export'` in your Next configuration file, and Tauri handles the rest.

The Webview

The biggest difference between Electron and Tauri comes from how they render the UI. The Electron framework comes with a full Chromium browser engine bundled in your application, which is the same engine that backs Google Chrome. This is useful because it means you don't have to worry about browser compatibility issues. Regardless of the end user's machine or architecture, the same Chromium instance renders your application UI. This results in a very standardized experience that ensures your app will look the same regardless of where it's running. However, this also results in a fair amount of bloat. For the vast majority of desktop apps, a full Chromium browser engine is overkill. Even the simplest "Hello World" applications using Electron can run you up to 150 megabytes of disk space.

Tauri solves this problem by leveraging the system's native webview. Instead of bundling a full browser engine, Tauri uses a library called **WRY**, which provides a cross-platform interface to the appropriate webview for the operating system. As you'd expect, this makes Tauri apps far more lightweight. The downside here is that you no longer have a hard guarantee on compatibility. From what I can tell, however, this mostly seems to be a non-issue. Compatibility issues across system webviews are exceedingly rare, especially for the major operating systems.

Node.js vs. Rust

Another major difference between the two frameworks is how they handle the "main" process. This refers to the backend process that orchestrates the application windows, menus, and other components of a desktop app that require interaction with system APIs. In Electron, the main process runs in a Node.js environment. This means you get access to all the typical Node APIs, you can import things like normal, and, perhaps most importantly, you can write your Electron-specific code in pure JavaScript. This is a huge bonus for Electron's target audience: web developers.

Tauri, on the other hand, uses Rust. All the framework code and the main process entrypoint are written in Rust. Obviously, this makes it a bit less accessible to the average web developer. That said, Tauri provides a fairly robust set of JavaScript APIs to interact with the Rust layer. For most applications, these APIs will be sufficient to do what you need to do. In the case of the workbench, I was able to fully replicate the functionality of the Electron version using the JavaScript APIs and some minimal Rust code.

In my experience, I found the Tauri APIs to fit more naturally in our application code. With Electron, if you need the main process to do something, you must always use inter-process communication, even for the simplest of tasks. If you want to write to a file on the host machine, for instance, your frontend needs to send a signal to the Electron main process, which will then spawn a new process and run the function you wrote that performs the write. With Tauri, you can just use Tauri's filesystem API directly in your application code. Under the hood, the same sort of IPC pattern is happening, but I think the Tauri abstraction is a bit nicer.

Sidecars

Since Electron runs on Node.js, it also bundles a full Node.js runtime with your application. This comes with some pros and cons. For the workbench, specifically, this is beneficial because the GraphQL layer is itself a separate Node.js application that needs to run alongside the frontend. Since Electron ships with Node.js, this means we can directly spin up the GraphQL server from the Electron main process using the Node runtime. This eliminates a lot of the headache associated with bundling and running a typical sidecar process. For instance, our app also ships with a copy of Dolt, which allows users to start up local Dolt servers directly from the workbench. To make this work, we have to bundle the appropriate Dolt binary with each workbench release that corresponds to the correct architecture. Without the Node runtime, we'd have to do something similar for the GraphQL layer.

With Tauri, this is exactly the problem we run into. To get around it, we need to compile the GraphQL server into a binary using a tool like `pkg`, then run it as a sidecar the same way we run Dolt. Thankfully, this seems to be a fairly common use case for Tauri applications, and they have a useful guide on **how to run Node.js apps as a sidecar**.

It's also worth mentioning that the full Node.js runtime is quite heavy, which also contributes to bloated Electron app sizes. After building the workbench using both Electron and Tauri, the difference in size was substantial. The left is the Electron version and the right is Tauri:

Limitations

After replicating the workbench's functionality in Tauri, we're holding off on making the full transition for a couple reasons:

1. **Lack of support for .appx and .msix bundles on Windows** - Currently, Tauri only support .exe and .msi bundles on Windows. This means your Microsoft Store entry will only link to the unpacked application. The workbench is currently bundled and published using the .appx format. To address this, we would need to take down the workbench entirely from the Microsoft store and create a new application that uses the .exe format.

2. Issues with MacOS universal binaries - This is more an annoyance than a bug, but I ran into a few issues related to codesigning universal binaries for MacOS. Namely, Tauri doesn't seem to be able to create Mac universal binaries from their arm64 and x64 subcomponents. It also seems to be codesigning the Mac builds twice.

Neither of these are hard blockers, but they're annoying enough that I'm holding off on migrating until they're resolved or our issues with Nextron become more problematic. For now, I'm leaving my branch with the migration open and hope to revisit soon. If you're on the Tauri team, let us know if you have solutions!

Conclusion

Overall, I'm impressed with Tauri. It eliminates much of the classic Electron bloat and integrates naturally with our existing codebase. If you're curious about Tauri or the Dolt Workbench, let us know on Discord.

< Blog

Install Dolt   Create an account ↗

Or join our mailing list to get product updates.

your email address

Submit

Documentation
Blog
Team
Security
Privacy
Terms