Chapter 1. What Is Electron?

Chances are high that you are already using a desktop app powered by web technologies. Popular apps like Spotify, Visual Studio Code, Atom, GitHub Desktop, and Slack all rely on a JavaScript stack—and even software powerhouses like Adobe's Creative Suite and the Nvidia Windows Drivers come with Node.js bundled in.

Electron is an open source framework that allows developers to use web technologies to build desktop applications. Initial development started in 2013 at GitHub to support the hackable text and code editor Atom. Both were made publicly available in the spring of 2014, but Electron has since surpassed Atom in popularity and become GitHub's biggest open source project to date. In 2017, Electron is being maintained by a team at GitHub with support from the community. Microsoft, Slack, Figma, Brave, and other companies, as well as dedicated individuals, continue to invest in the development and maintenance of the project.

This short guide introduces Electron, the most popular framework and runtime for cross-platform desktop applications built with web technologies (and, optionally, with native components too). Though far

from a complete reference, it briefly introduces each aspect of developing and distributing an Electron-based app, allowing you to get the 30,000-foot overview.

Building Desktop Apps with Chromium, Node.js, and C++

At its core, Electron consists of three components. It embeds Chromium's rendering library (known as *libchromiumcontent*), which is the open source foundation for Google's browser Chrome. With it comes support for the latest web standards, a performant JavaScript engine, and constantly improving developer tools. It is a common misconception that Electron includes Chrome in its entirety; to keep Electron as lean as possible, it really does just contain Chromium's rendering library.

The second component is Node.js, the popular JavaScript runtime built on top of V8 (the same JavaScript engine that powers libchromiumcontent and Chrome). It brings a vibrant open source ecosystem to the table: countless modules on npm offer battle-tested solutions for common development tasks and easy integration with virtually any service. Node.js rightfully claims to be the "largest ecosystem of open source libraries in the world," a benefit that can dramatically reduce the amount of time needed to develop solutions.

The third component is a thick layer of C++, extending the usual set of available APIs with native implementations for common operating system operations such as sending native notifications, accessing system preferences, and creating native dialogs. In addition, Node.js allows the use of native modules—native components written in C, C++, Rust, or Objective-C—ensuring that the developer is never limited to JavaScript should a complicated problem require the use of a low-level programming language.

Combining Chromium and Node.js, and throwing in a large number of natively implemented APIs, Electron enables developers to build powerful desktop applications with little effort. All three technologies are cross-

platform, supporting Windows, macOS, and Linux alike. Developers with experience in building desktop software are often surprised by how small the teams behind popular Electron applications are: Microsoft's Visual Studio Code was initially built by merely seven engineers, Slack's desktop app by just two, and GitHub's desktop app by four. Electron's true power is that it allows developers to build a large percentage of their desktop apps using nothing but web technologies, leaving only the truly computing-intensive tasks to be implemented natively.

Main and Renderer Processes

Electron knows two different process types: *main process* and *renderer process*. When an Electron application is started, it launches the main process. Within one application instance, there is always exactly one main process. This process has no access to DOM (Document Object Model) APIs, is windowless, and behaves a lot like a Node.js process (see Figure 1-1).

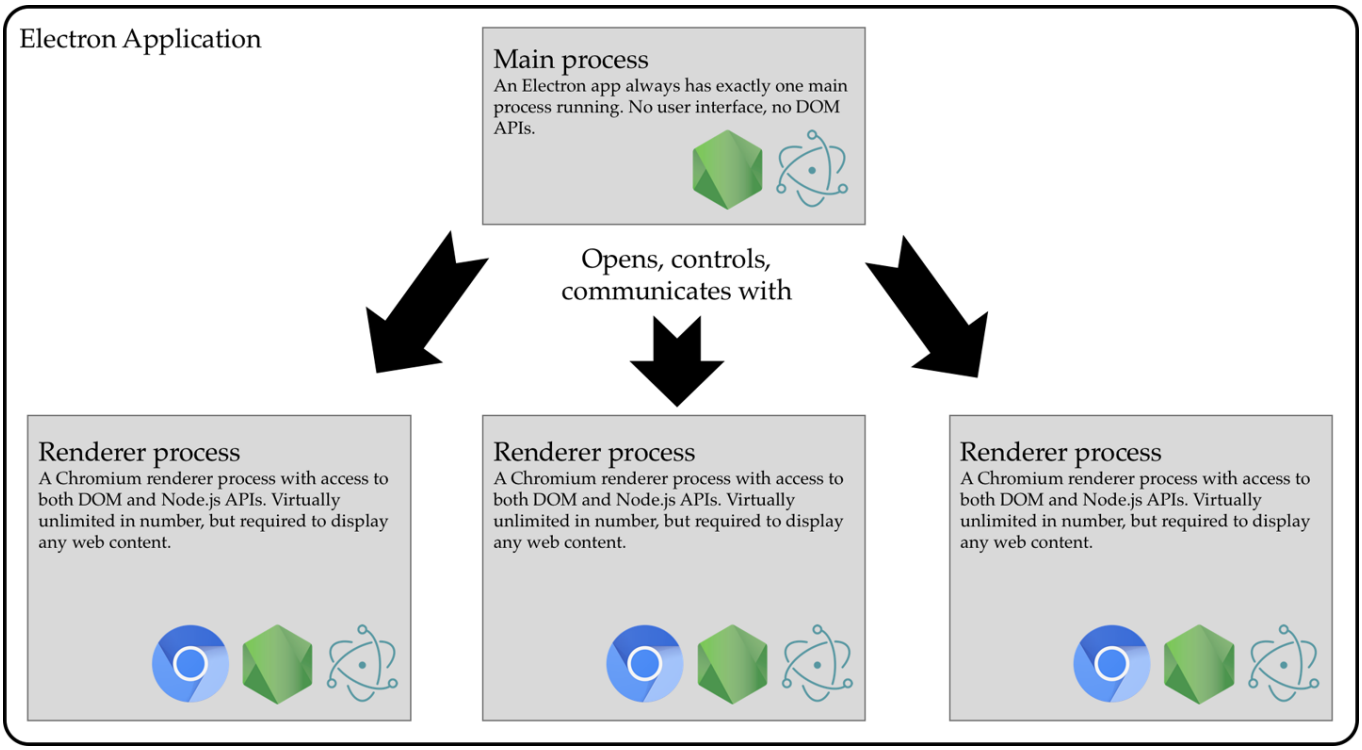

Figure 1-1. The two Electron process types

As soon as the main process has initialized, it is able to open windows. Those windows run in their own process and are referred to as *renderer processes*. In Electron, each web page runs in its own process. The distinction is important: Renderer processes have the familiar DOM with `window` and `document` objects, can create and render HTML elements, and have Chromium's developer tools available, whereas the main process is really just a Node.js process. At the same time, many of Electron's APIs can only be accessed from the main process: calling methods that perform native GUI operations from a renderer process is risky, likely to leak resources, and therefore simply not allowed.

THE MYSTERIOUS BROWSER PROCESS

In older technical documentation, you will sometimes see the main process referred to as the *browser process*, which—naturally—can be confusing, but has its origin in Chrome's terminology. In Chrome's world, the main process is in charge of managing windows and tabs, thus it is dubbed the "browser process."

The main process manages and facilitates communication with each renderer process. The renderer processes communicate in turn with the main process, but do not usually bother talking to other renderer processes. This communication is facilitated by Chromium's multiprocess architecture, enabling developers to pass objects between processes and to even call methods on one process from another. Even though Electron imposes strict rules on which API can be called from which process, it also makes communication between processes easy enough that the restrictions are never limiting.

The difference between the main process and its attached renderer processes is important: each one has different capabilities and is responsible for different aspects of your application's life cycle. Throughout this guide, you will find references explaining which process

is responsible for a given task. This distinction is sometimes confusing for beginners but will quickly feel natural.

Node.js in the Browser

Once a window is created, web developers will quickly feel like they have been unshackled. Electron combines Node.js and Chromium, meaning you can make full use of all of Node's APIs, including `require` and any module you can find on npm. Naturally, this encompasses native modules like sqlite or nodegit.

Making Node.js available inside the web context has two important implications. First, unleashing full system access onto a web page truly removes any limitations a web developer may have run into. Consider Visual Studio Code and its powerful debugger as an example: thanks to the combination of web technologies and Node.js, it is able to build, inspect, and debug C# or C++ processes. From an operating system perspective, an Electron application has full access to user space and is in its access limitations equal to a fully native application.

It also removes any safety boundaries a web developer may be used to. A compromised Electron application is far more dangerous than a compromised website, given that an Electron app has full access to the operating system. This short introduction dedicates a section to security, but as a basic rule, loading a remote web page in an Electron renderer process with Node.js integration enabled is a bad idea. To mitigate this risk, Electron allows developers to create renderer processes that have the Node.js integration entirely disabled. Later, we will discuss scenarios in which you would like to load remote content and offer a limited set of APIs only available through Node.js.

- Get unlimited access to books, videos, and live training
- Never lose your place—all your devices are synced
- Learn during your commute with online and offline access