SAMPLE CHAPTER

Electron IN ACTION

Steve Kinney

MANNING

Electron in Action

by Steven Kinney

Chapter 1

Copyright 2019 Manning Publications

brief contents

1 Introducing Electron

This chapter covers

- Understanding what Electron is
- Learning which technologies Electron is built on
- Understanding how using Electron differs from traditional web applications
- Structuring Electron applications
- Using Electron in production to build real-world applications

One of the big things that the web has going for it is ubiquity. It's an amazing platform for creating collaborative applications that can be accessed from a wide range of devices running different operating systems. That said, entire classes of applications can't be built in the browser environment. Web applications can't access the filesystem. They can't execute code that isn't written in JavaScript. They can't hook into many of the operating system APIs that desktop applications can. Most web applications aren't available when there isn't a reliable internet connection.

For a long time, building for the desktop has involved adopting a completely different skill set. Many of us don't have the bandwidth to take on the long learning curve necessary for learning new languages and frameworks. With Electron, you

can use your existing skills as a web developer to build applications that have many of the capabilities of a native desktop application.

1.1 What is Electron?

Electron is a runtime that allows you to create desktop applications with HTML5, CSS, and JavaScript. It's an open source project started by Cheng Zhao (aka zcbenz), an engineer at GitHub. Previously called Atom Shell, Electron is the foundation for Atom, a cross-platform text editor by GitHub built with web technologies.

You may have heard of—or used—Apache Cordova or Adobe PhoneGap for building web applications—wrapped in native shells—for mobile operating systems such as iOS, Android, and Windows Phone. If so, then it might be helpful to think of Electron as a similar tool for building desktop applications.

Electron allows you to use the web technologies you already know to build applications that you wouldn't otherwise build. In this book, you'll learn how to build applications that hook into native operating system APIs on Windows, macOS, and Linux.

Electron combines the Chromium Content Module and Node.js runtimes. It allows developers to build GUIs with web pages as well as access native operating system capabilities on Windows, macOS, and Linux through an OS-agnostic API.

Chromium and Node are both wildly popular application platforms in their own right, and both have been used independently to create ambitious applications. Electron brings the two platforms together to allow you to use JavaScript to build an entirely new class of application. Anything you can do in the browser, you can do with Electron. Anything you can do with Node, you can do with Electron.

The exciting part is what you can do with the two technologies together. You can build applications that take advantage of both platforms and build applications that wouldn't otherwise be possible on only one. That's what this book is all about. Electron is not only a great choice for building web applications that behave like native desktop applications; it's also a great choice for building a GUI around Node applications that would otherwise be limited to a command-line interface. See figure 1.1.

Let's say that you want to build an application that allows you to view and edit a folder of images on your computer. Traditional browser applications can't access the filesystem. They couldn't access the directory of photographs, load any of the photographs in the directory, or save any of the changes that you made in the application. With Node, you could implement all those features, but you couldn't provide a GUI, which would make your application difficult to use for the average user. By combining the browser environment with Node, you can use Electron to create an application where you can open and edit photographs as well as provide a UI for doing so. See figure 1.2.

Electron isn't a complicated framework—it's a simple runtime. Similar to the way you might use `node` from the command line, you can run Electron applications using the `electron` command-line tool. You don't have to learn many conventions to get started, and you're free to structure your application however you'd like—although I'll provide tips and best practices throughout this book.

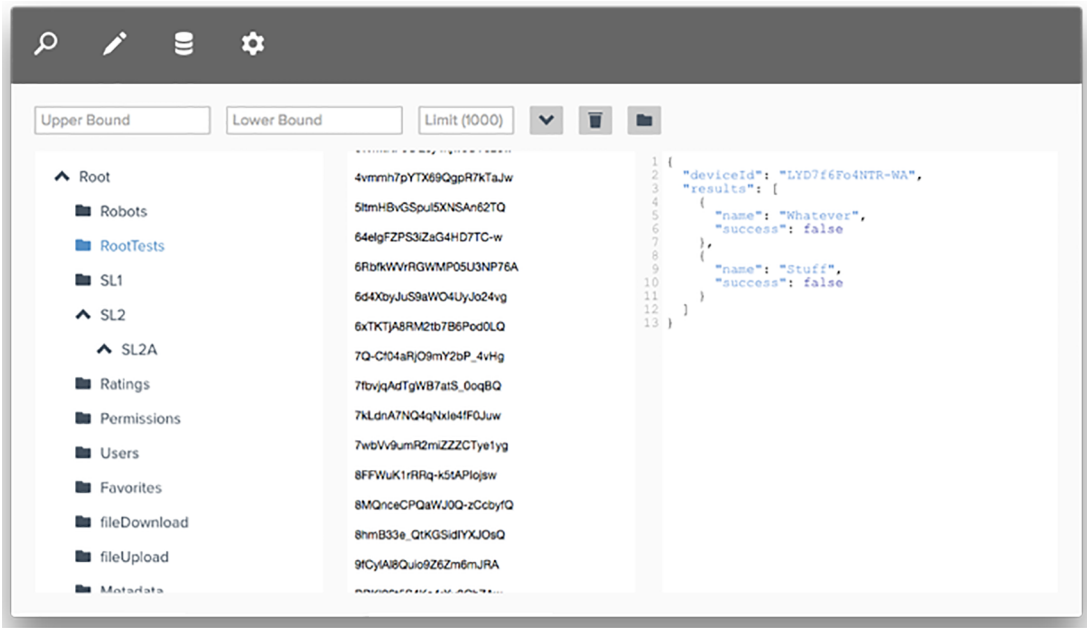

Figure 1.1 LevelUI is a GUI for Node's LevelUp database built with Electron. You couldn't build this application in a traditional browser because it wouldn't have the ability to access a local database on the user's computer. It also couldn't use the LevelUI library because it's a compiled C++ module, which only Node—and not the browser—can use.

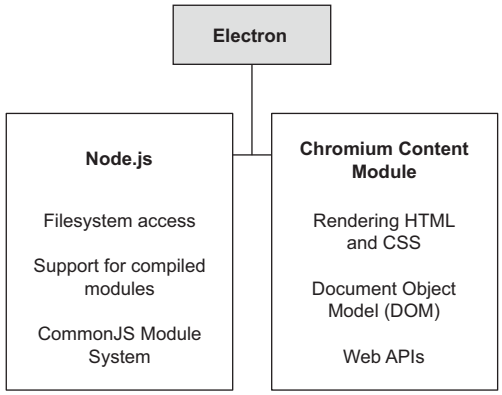

Figure 1.2 Electron combines the core web browsing component of Chromium with the low-level system access of Node.

1.1.1 What is the Chromium Content Module?

Chromium is the open source version of Google's Chrome web browser. It shares much of the same code and features with a few minor differences and different licensing. The Content Module is the core code that allows Chromium to render web pages in independent processes and use GPU acceleration. It includes the Blink rendering

engine and the V8 JavaScript engine. The Content Module is what makes a web browser a web browser. It handles fetching and rendering HTML from a web server, loading any referenced CSS and JavaScript, styling the page accordingly, and executing the JavaScript.

The easiest way of thinking about the Content Module is to consider what it doesn't do. The Content Module doesn't include support for Chrome extensions. It doesn't handle syncing your bookmarks and history with Google's cloud services. It doesn't handle securely storing your saved passwords or automatically filling them in for you when you visit a page. It doesn't detect if a page was written in another language and subsequently call on Google's translation services for assistance. The Content Module includes only the core technologies required to render HTML, CSS, and JavaScript.

1.1.2 What is Node.js?

For the first 15 years of its existence, JavaScript was traditionally isolated within the web browser. There wasn't much in the way of support for running JavaScript on the server. Projects existed, but they never got any traction. The Node.js project was initially released in 2009 as an open source, cross-platform runtime for developing server-side applications using JavaScript. It used Google's open source V8 engine to interpret JavaScript and added APIs for accessing the filesystem, creating servers, and loading code from external modules.

Over the last few years, Node has enjoyed a surge of interest and popularity and is used for a wide range of purposes, from writing web servers to controlling robots to—you guessed it—building desktop applications. Node comes bundled with a package manager called npm, which makes it easy to lean on the more than 250,000 libraries available in its registry.

1.2 Who's using Electron?

Electron is used by companies, large and small, to build desktop applications. As discussed earlier, it was originally developed as the foundation for GitHub's Atom text editor. Atom needed access to the filesystem to fulfill its duties as a text editor. Similarly, other companies have turned to Electron as the foundation of their text-editing applications. Facebook released Nuclide as a package on top of Atom that turns the text editor into a full-fledged integrated development environment (IDE) with first-class support for working with React Native, Hack, and Flow projects. Microsoft also uses Electron for its cross-platform Visual Studio Code editor, which runs on macOS, Windows, and Linux.

You can build more than text editors with Electron. Slack, the popular messaging application, uses Electron for its Windows and Linux versions. Nylas used Electron for its N1 email client, which is designed to look beautiful across all the major platforms. It also supports a JavaScript plugin architecture that allows third-party developers to add features and extend the UI.

Particle, which produces development kits for creating custom hardware, uses Electron for its IDE, which lets users write code and deploy it to hardware devices through a cellular or Wi-Fi network. Using Mapbox Studio, users can import data stored locally and process it on their computers without having to send it over the internet to Mapbox's servers. The result is a faster and better experience that allows designers to create custom maps easily.

Dat is an open source tool for sharing, syncing, and versioning decentralized data. The grant-funded project consists of a team of three web developers. Despite being a relatively small team, Dat released a desktop application for the project using Electron. In 2015, Wiredcraft—a software consultancy—used Electron to build an offline-friendly Windows application for collecting and correcting voter registration information in Myanmar. The firm needed an application that could store the collected data offline and then publish it when the device was connected to the network. The company chose Electron as an alternative to building it using C++ because it allowed Wiredcraft to take advantage of its existing HTML, CSS, and JavaScript prowess instead of relearning those skills for a different ecosystem.

Brave—a new browser focused on speed and security by Brendan Eich, the creator of JavaScript—is itself built on top of Electron. See figure 1.3. That's right, you can even use web technologies to build a web browser.

New projects built on top of Electron are being released every day as companies and developers see the value in building products that use the power afforded to desktop applications while still maintaining the web's intrinsic platform agnosticism. By

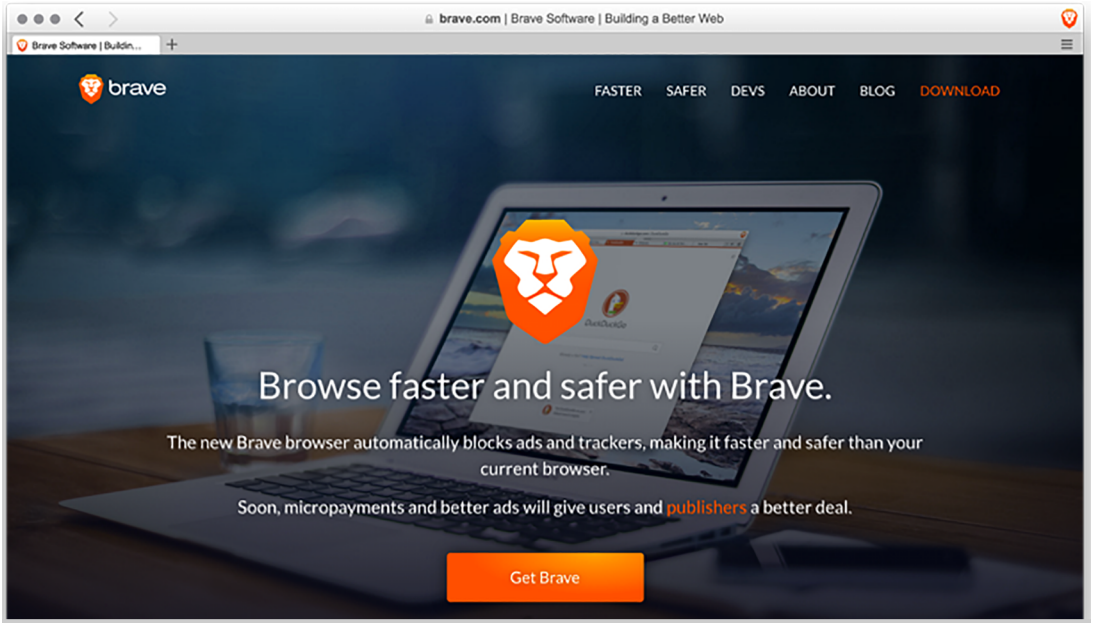

Figure 1.3 Brave is an entire web browser built on top of Electron.

the end of this book, you'll take your existing web development skills and apply them to create new applications that wouldn't have been possible in the traditional browser environment.

1.3 What do I need to know?

Let's start with what you don't need to know. This book is for web developers who want to use their existing skill set to create desktop applications that wouldn't be possible in the traditional browser environment. You don't need any experience building desktop applications to get value out of this book.

That said, you should be comfortable with writing JavaScript, HTML, and CSS, but by no means do you need to be an expert. I won't be covering variables or conditionals in this book, but if you're familiar with general language features of JavaScript, then you probably have the requisite skills to follow along. It's also helpful if you're familiar with some of the conventions and patterns from Node.js, such as how the module system works. We'll explore these concepts as we come across them.

1.4 Why should I use Electron?

When you're writing applications for a web browser, you have to be conservative in what technologies you choose to use and cautious in how you write your code. This is because—unlike many server-side situations—you're writing code that will be executed on someone else's computer.

Your users could be using the latest version of a modern browser such as Chrome or Firefox, or they could be using an outdated version of Internet Explorer. You have little to no say in where your code is being rendered and executed. You have to be ready for anything.

You typically must write code for the lowest common denominator of features that have the widest support across all versions of all browsers in use today. Even if a better, more efficient, or generally more appealing solution exists to a problem, you might not be able to use that approach. When you decide to reach for a modern browser feature, you usually need to implement a contingency plan of graceful fallbacks, feature detection, and progressive enhancement that adds a nontrivial amount of friction to your development workflow.

When you build your applications with Electron, you're packaging a particular version of Chromium and Node.js, so you can rely on whatever features are available in those versions. You don't have to concern yourself with what features other browsers and their versions support. If the build of Chromium included with your application supports the Service Worker API, for example, then you can confidently rely on that API in your application. See figure 1.4.

Electron allows you to use cutting-edge web platform features because it includes a relatively recent version of Chromium. Generally speaking, the version of Chromium in Electron is about one to two weeks behind the most recent stable release—and a

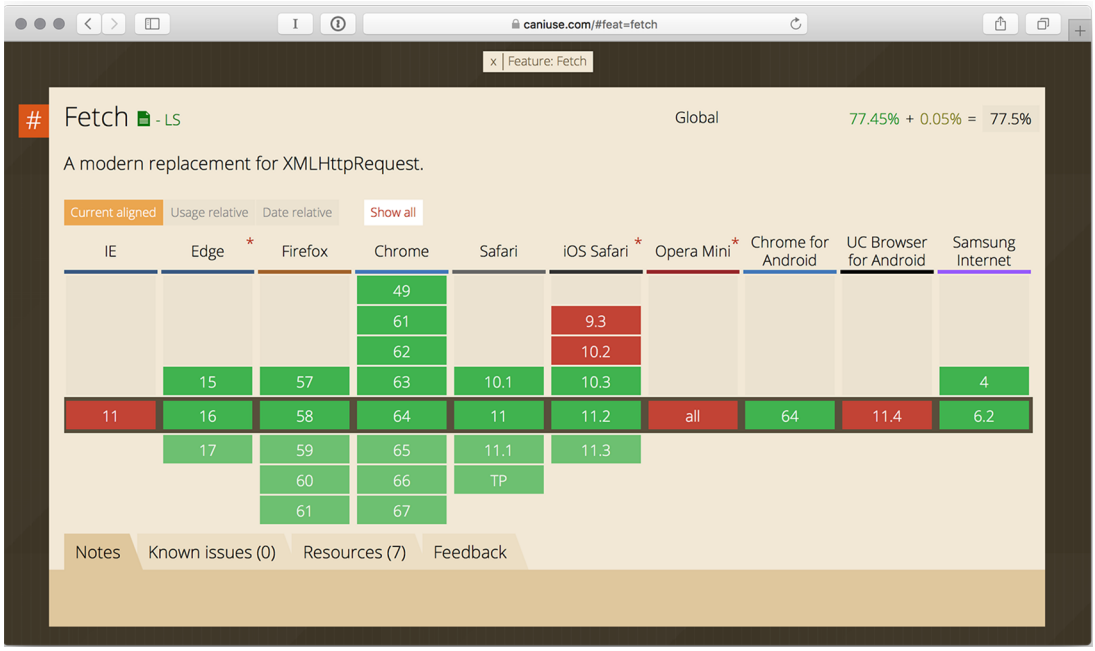

Figure 1.4 In a browser-based web application, it might not be practical to rely on the Fetch API, given its inconsistent support. But in your Electron applications, you're bundling the current stable build of Chromium with full support for the Fetch API.

new stable release comes out every six weeks. Electron typically includes new versions of Node.js about a month after they're released to ensure it contains the most recent version of V8. Electron already includes a modern build of V8 from Chromium and can afford to wait for minor bug fixes before upgrading to the latest version of Node.

1.4.1 Building on your existing skill set

If you're like me, you probably have much more experience building web applications than desktop applications. You'd love to add the ability to create desktop applications to your set of tools, but you don't have the bandwidth to learn not only a new programming language but likely a new framework as well.

Learning a new language or framework is an investment that's not to be taken lightly. As a web developer, you're used to writing applications that work equally well for all your users—even if that means fighting with idiosyncrasies of a particular browser or screen size. But when you're contemplating building traditional desktop applications, you're talking not only about learning one language and framework. You're also looking at learning at least three different languages and frameworks if you want to target Windows, macOS, and Linux.

Individuals and small teams can use Electron to offer desktop applications in situations where they couldn't otherwise. For a small team, hiring a developer skilled in building applications for each of those platforms may not be an option. Electron lets you use your existing skill set and deploy your application to all the major platforms. With Electron, you can support multiple operating systems with less effort than you're normally used to for supporting multiple browsers.

1.4.2 Access to native operating system APIs

Electron applications are similar to any other desktop application. They live in the filesystem with the rest of your native applications. They sit in the dock in macOS or taskbar in Windows and Linux where all the other native applications hang out. Electron applications can trigger native Open and Save File dialog boxes. These dialog boxes can be configured to allow the operating system to select only files with a particular file extension, whole directories, or multiple files at the same time. You can drag files onto your Electron applications and trigger different actions.

Additionally, Electron applications can set custom application menus like any other application. See figure 1.5. They can create custom context menus that spring into action when the user right-clicks from within the application. You can use Chromium's notification API to trigger system-level notifications. They can read from the system clipboard and write text, images, and other media to it as well.

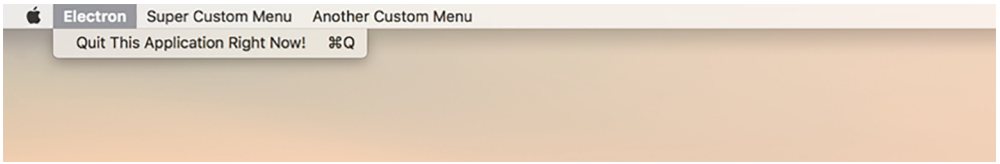

Figure 1.5 Electron allows you to create custom application menus.

Unlike traditional web applications, Electron applications aren't limited to the browser. You can create applications that live in the menu bar or the system tray. See figure 1.6. You can even register global shortcuts to trigger these applications or any of their abilities with a special keystroke from anywhere in the operating system.

Electron applications have access to system-level information—such as whether the computer is on battery power or plugged into the wall. They can also keep the operating system awake and prevent it from going into power-saving mode, if necessary.

1.4.3 Enhanced privileges and looser restrictions

The web is the largest distributed application platform in history. It's so ubiquitous that web developers take many of the associated headaches for granted. Building web applications involves carefully choreographing the communication between the server-side application and the potentially thousands of instances of the client-side application. Your client-side code runs in the user's web browser—far removed from the server.

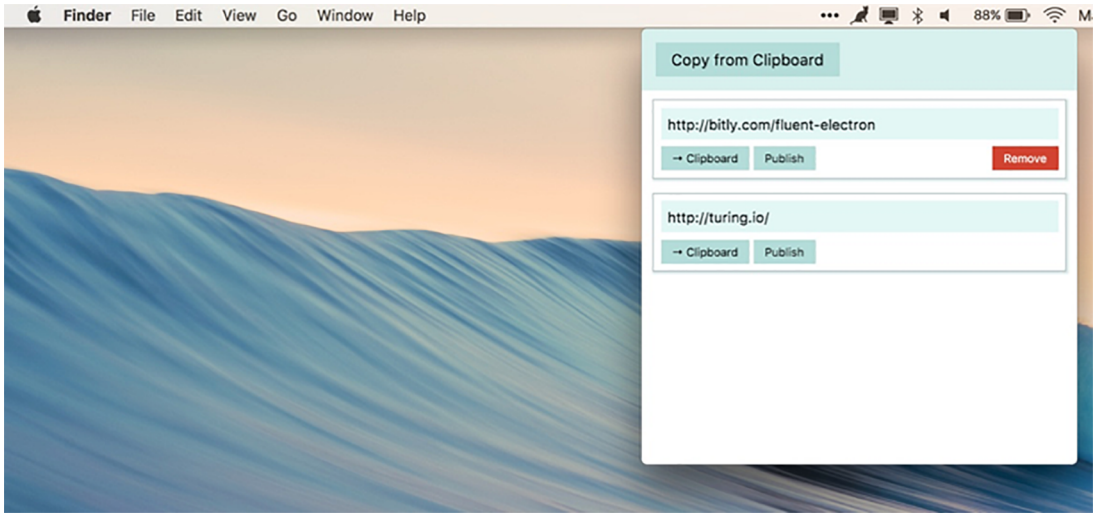

Figure 1.6 You can create an application that lives in the operating system's menu bar or system tray.

Anything that happens in the client is unique to that browser session unless the changes are sent back to your server. By the same token, if anything changes on your end, you have to wait until the client sends another HTTP request asking for updates; or you can potentially send the updates over WebSockets, if you've implemented that capability on both the client and the server.

Desktop applications enjoy a wider range of abilities and fewer restrictions on what they're allowed to do because the user explicitly went out of their way to download, install, and open the application. When you're browsing the web, however, you don't have the same amount of agency. You're executing code that you didn't choose to install on your computer. As a result, web applications have many limits on what they're allowed to do.

When the browser visits a page on the web, it happily downloads all the assets referenced in the HTML code of the document it's loading, as well as any additional dependencies added by those first assets, and then begins executing the code. Over the years, browser vendors have added restrictions to what the browser can do to prevent malicious code from harming the user or other sites on the internet.

I'm not a bad person, but let's say—for the sake of argument—that I am. Let's also say that I run a popular site that sells artisanal, hand-crafted widgets. One day, a competitor pops onto my radar selling equally pretentious widgets at a steep discount. My site is still getting more traffic for now, but this new challenger is affecting my beauty sleep.

Being a bad person, I decide to add JavaScript to my website that fires off an AJAX request every few milliseconds to my competitor's site with the hope that the thousands of visitors to my site will download this code and effectively flood my sworn enemy's server and make it unable to handle any legitimate request. It will also degrade the

experience my visitors have on my site, but that's a price I'm willing to pay to bring my competitor's website to its knees.

Despite the diabolical nature of my plan, it won't work. Modern browsers restrict client-side code from making requests to a third-party server unless that server explicitly declares a policy that it allows such requests.

Generally speaking, most sites don't do this. If you want to send a request to a third-party server, then you have to first make a request to your own server, have it contact the third party, and relay the results back to the client. In the previous example, this adds my server as a bottleneck for those thousands of requests, which would make it infeasible for me to launch this kind of attack and trivially easy for my competitor to block my single IP address as opposed to the IPs of the thousands of visitors to my site.

The browser also places strict limits on what client-side code has access to and what it can do. All of this makes for a safer, more secure, and—ultimately—better experience for the user. It's all incredibly practical and is part of what makes the web such a fantastic and approachable platform for users.

That said, all these useful and important security restrictions severely limit the kinds of applications you can build using web technologies. The user explicitly downloads and installs Electron applications like any other native application. You're free to access the filesystem like any native desktop application or server-side Node process would. You're also free to make requests to third-party APIs without going through a Node server because you have access to the same privileges and capabilities as any other Node process. See figure 1.7.

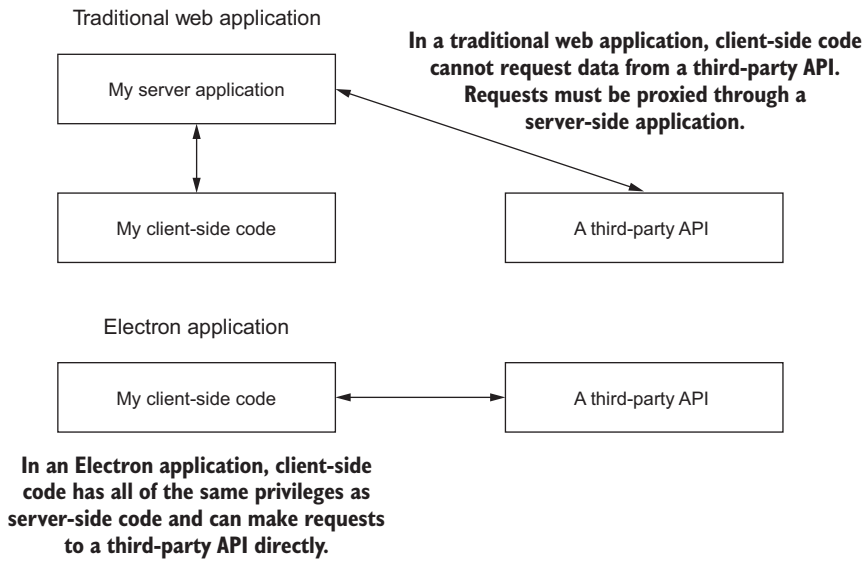

Figure 1.7 Electron applications can use their Node.js runtimes to make requests to third-party APIs.

1.4.4 Accessing Node from the browser context

Along with granting access to the filesystem and the ability to fire up a web server, Node.js uses a module system based on the CommonJS modules specification. From its earliest incarnations, Node has supported the ability to break out code into multiple modules and explicitly include ones you require from within a given file.

Packaging any nontrivial amount of JavaScript code for the browser hasn't always been so easy. For a small amount of code, you can include it in your markup between a matching pair of opening and closing `<script>` tags. For larger blocks of code, you can use the `src` attribute to reference an external JavaScript file. You're welcome to do that as many times as you wish, but you'll have to pay the performance penalties as the browser fires off an additional request to fetch each external asset.

You're welcome to use a build tool such as webpack or Browserify if you like, but it's often not necessary in Electron applications because all of Node's global properties (for example, `require`, `module`, and `exports`) are available in the browser content. You can use Node's module system on what you'd traditionally think of as the client side without needing to add a build process to your application.

You can access all of Node's APIs from the browser context of your Electron application. On top of taking advantage of Node's module system, you can also use compiled modules with native extensions, access the filesystem, as well as do a bevy of other things that aren't typically supported in the browser environment.

1.4.5 Offline first

As anyone who has ever taken a computer on a transcontinental flight can attest, most browser-based web applications aren't much good without a connection to the internet. Even advanced web applications using any of the popular client-side frameworks like Ember, React, or Angular typically need to connect to a remote server to download their assets.

Electron applications have already been downloaded to the user's computer. Typically, they load a locally stored HTML file. From there, they can request remote data and assets if a connection is available. Electron even provides APIs that allow you to detect if a connection is available. No special manifests or bleeding-edge technologies are necessary to build an offline application using Electron—it's the default state unless the application explicitly requests something from the internet. Barring a special circumstance—you're building a chat client, for example—Electron applications work as well offline as any other application.

1.5 How does Electron work?

Electron applications consist of two types of processes: the main process and zero or more renderer processes. Each process plays a different role in the application. The Electron runtime includes different modules to assist you in building your application. Certain modules, such as the ability to read and write from the system's clipboard, are

available in both types of processes. Others, such as the ability to access an operating system's APIs, are limited to the main process. See figure 1.8.

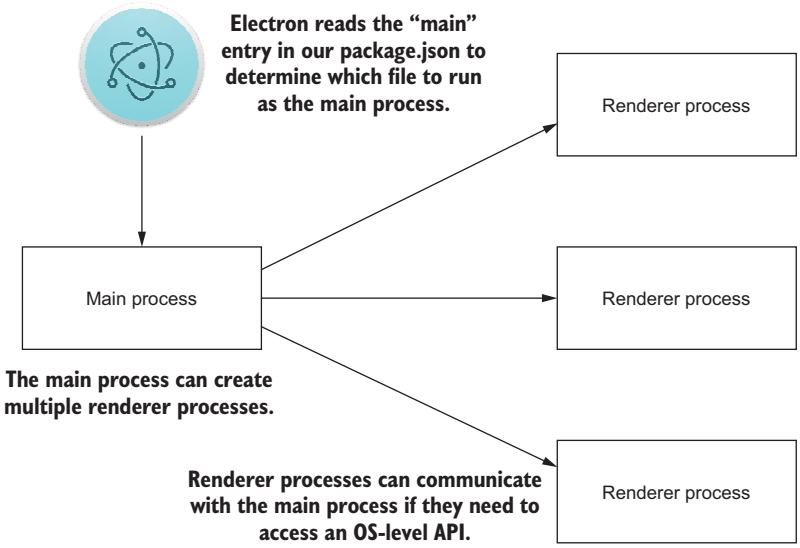

Figure 1.8 Electron's multiprocess architecture

When Electron starts up, it turns to the start entry in your package.json manifest included in your project to determine the entry point of your application. This file can be named anything you'd like, as long as it's included properly in package.json. Electron runs this file as your main process.

1.5.1 The main process

The main process has a few important responsibilities. It can respond to application lifecycle events such as starting up, quitting, preparing to quit, going to the background, coming to the foreground, and more. The main process is also responsible for communicating to native operating system APIs. If you want to display a dialog box to open or save a file, you do it from the main process.

1.5.2 Renderer processes

The main process can create and destroy renderer processes using Electron's `BrowserWindow` module. Renderer processes can load web pages to display a GUI. Each process takes advantage of Chromium's multiprocess architecture and runs on its own thread. These pages can then load in additional JavaScript files and execute code in this process. Unlike normal web pages, you have access to all the Node APIs in your renderer processes, allowing you to use native modules and lower-level system interactions.

Renderer processes are isolated from each other and unable to access operating system integration APIs. Electron includes the ability to facilitate communication between processes to allow renderer processes to communicate with the main process in the event that they need to trigger an Open or Save File dialog box or access any other OS-level integration.

1.6 Electron vs. NW.js

Electron is similar to another project called NW.js (previously known as node-webkit). The two have much in common. In fact, zcbenz was a heavy contributor to NW.js before starting work on Electron. That said, they're different in several important ways, as shown in table 1.1.

Table 1.1 A comparison of some of the main differences between Electron and NW.js

	Electron	NW.js
Platform	Officially supported Chromium Content Module from recent build	Forked version of Chromium
Process model	Separate processes	Shared Node process
Crash reporting	Built in	Not included
Auto-updater	Built in	Not included
Windows support	Windows 7 and later	Windows XP and later

NW.js uses a forked version of Chromium. Electron uses Chromium and Node.js but doesn't modify them. This makes it easier for Electron to keep pace with the most recent versions of Chromium and Node. Electron also includes modules for automatically downloading updates and reporting crashes. NW.js doesn't.

NW.js applications start from an HTML page. Each browser window shares a common Node process. If more than one window is opened, they all share the same Node process. Electron keeps the Node and browser processes separate. In Electron, you start a main process from Node. This main process can open browser windows, each of which is its own process. Electron provides APIs for facilitating communication between the main process and the browser windows, which we call *renderer processes* throughout this book.

If backward compatibility is a concern, then NW.js might be a better choice because it supports Windows XP and Vista. Electron supports only Windows 7 and later. For multimedia-focused applications, Electron is typically a better choice because Chromium's FFmpeg library is a statically linked dependency, so Electron supports more codecs out of the box. With NW.js, you need to manually link the FFmpeg library.

Summary

- Electron is a runtime for building desktop applications using web technologies.
- The project began at GitHub as the foundation for the Atom text editor.
- Electron combines the Chromium Content Module, which is a stripped-down version of the Chrome web browser with Node.
- This combination allows you to build applications that can access the filesystem and compiled modules, as well as render a UI and use web APIs.
- Electron is used by applications large and small such as Atom, Microsoft's Visual Studio Code, and Slack.
- Electron is great for individuals or small teams who may want to target more than one platform without having to learn three or more languages, as well as each platform's frameworks.
- Electron allows web developers to use their existing skill set to build applications that wouldn't otherwise be possible within the browser environment.
- Electron ships with a modern version of Chromium and Node, which means you can use the latest and greatest features of the web platform.
- Electron applications can access operating system APIs such as application and context menus, File Open and Save dialog boxes, battery status and power settings, and more.
- Electron applications are permitted enhanced privileges and have fewer restrictions imposed on their capability as compared to browser-based web applications.
- Electron applications consist of one main process and one or more renderer processes.
- The main process handles OS integration, manages the lifecycle of the application, and creates renderer processes.
- Renderer processes display the UI and respond to user events.
- Electron differs from NW.js in that it uses the officially supported content module from Chromium as opposed to NW.js, which uses a custom fork of Chromium.

PROGRAMMING/WEB DEVELOPMENT

Electron IN ACTION

Steve Kinney

Wouldn't it be great to build desktop applications using just your web dev skills? Electron is a framework designed for exactly that! Fully cross-platform, Electron lets you use JavaScript and Node to create simple, snappy desktop apps. Spinning up tools, games, and utilities with Electron is fast, practical, and fun!

Electron in Action teaches you to build cross-platform applications using JavaScript, Node, and the Electron framework. You'll learn how to think like a desktop developer as you build a text tool that reads and renders Markdown. You'll add OS-specific features like the file system, menus, and clipboards, and use Chromium's tools to distribute the finished product. You'll even round off your learning with data storage, performance optimization, and testing.

What's Inside

- Building for macOS, Windows, and Linux
- Native operating system APIs
- Using third-party frameworks like React
- Deploying to the Mac App Store

Requires intermediate JavaScript and Node skills. No experience building desktop apps required.

Steve Kinney is a principal engineer at SendGrid, an instructor with Frontend Masters, and the organizer of the DinosaurJS conference in Denver, Colorado.

"The definitive source on cross-platform desktop app development with a code-driven narrative."
—Ashwin K. Raj, Innocepts

"Takes you from simply knowing what Electron is about, to actually writing complex Electron applications."
—Alexey Galiullin, Voiceworks

"Allowed me to quickly build my own day-to-day tools."
—Philippe Charrière, GitLab

"Fast to read and easy to understand."
—Jay Kelkar, Kelkar Systems

"Finally, JavaScript is everywhere!"
—William E. Wheeler, consultant

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/electron-in-action

ISBN-13: 978-1-61729-414-3
ISBN-10: 1-61729-414-4
54499
9 781617 294143

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]