SAMPLE CHAPTER

Cross-Platform Desktop Applications

Using Electron and NW.js

Paul B. Jensen

FOREWORD BY Cheng Zhao

MANNING

Cross-Platform Desktop Applications
Using Electron and NW.js

by Paul Jensen

Chapter 6

brief contents

6 Exploring NW.js and Electron's internals

This chapter covers

- Understanding how NW.js and Electron combine Node.js and Chromium
- Developing with Electron's multi-process approach
- Building with NW.js's shared-context approach
- Sharing state by passing messages

Although NW.js and Electron consist of the same software components, and Cheng Zhao has influenced the development of both, the two frameworks have evolved different approaches to how they function under the hood. Analyzing how they operate internally will help you understand what's going on when you're running an app and demystify the software.

In this chapter, we'll look at how NW.js and Electron function internally. We'll take a look at NW.js first to see how it combines Node.js with Chromium (because that was the first Node.js desktop app framework) and then explore how Electron took a different approach to combining those software components. Following that, we'll look at the frameworks' different approaches to context and state. I'll then

elaborate a bit on Electron's use of message passing to transmit data as state between the processes in a desktop app.

We'll also look at some resources for further reading. The goal is that you'll be in a good position to understand how the two frameworks differ in their internal architecture and the implications this has on building desktop apps with them.

6.1 How does NW.js work under the hood?

From a developer's view, NW.js is a combination of a programming framework (Node.js) with Chromium's browser engine through their common use of V8. V8 is a JavaScript engine created by Google for its web browser, Google Chrome. It's written in C++ and was designed with the goal of speeding up the execution of JavaScript in the web browser.

When Node.js was released in 2009, a year after Google Chrome, it combined a multiplatform support library called libuv with the V8 engine and provided a way to write asynchronous server-side programs in JavaScript. Because both Node.js and Chromium use V8 to execute their JavaScript, it provided a way to combine the two pieces of software, which Roger Wang came to understand and figure out. Figure 6.1 shows how those components are combined.

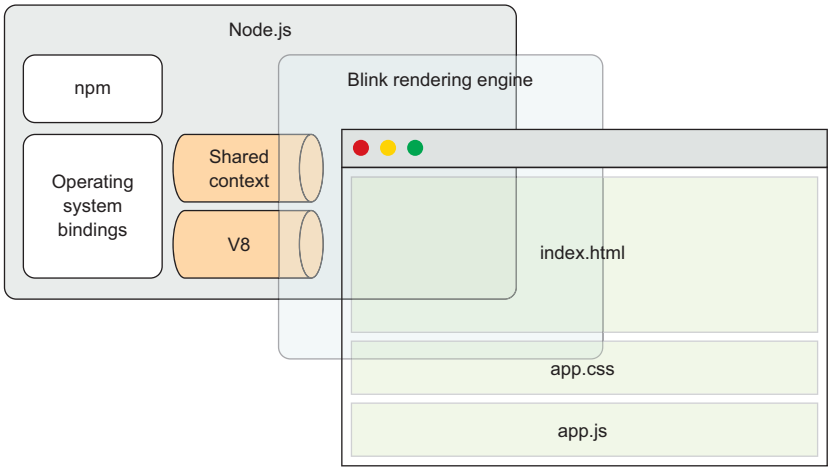

Figure 6.1 Overview of NW.js's component architecture in relation to loading an app

Looking at figure 6.1, you can see that Node.js is used in the back end to handle working with the OS, and that Blink (Chromium's rendering engine) is used to handle rendering the front-end part of the app, the bit that users see. Between them, both Node.js and Blink use V8 as the component that handles executing JavaScript, and it's

this bit that's crucial in getting Node.js and Chromium to work together. There are three things necessary for Node.js and Chromium to work together:

- Make Node.js and Chromium use the same instance of V8
- Integrate the main event loop
- Bridge the JavaScript context between Node and Chromium

NW.js and its forked dependencies

NW.js, a combination of Node.js and the WebKit browser engine, used to be known as node-webkit. Recently, both components were forked: Google created a fork of WebKit called Blink, and in October 2014 a fork of Node.js called IO.js emerged. They were created for different reasons, but as projects that received more regular updates and features, NW.js opted to switch to using them.

As node-webkit no longer used Node.js and WebKit (but IO.js and Blink instead), it was suggested that the project should be renamed; hence, the project was renamed to NW.js.

In May 2015, the IO.js project agreed to work with the Node.js foundation to merge IO.js back into Node.js. NW.js has switched back to using Node.js since.

6.1.1 Using the same instance of V8

Both Node.js and Chromium use V8 to handle executing JavaScript. Getting them to work together requires that a couple of things happen in order. The first thing NW.js does is load Node.js and Chromium so that both of them have their JavaScript contexts loaded in the V8 engine. Node's JavaScript context will expose global objects and functions such as `module`, `process`, and `require`, to name a few. Chromium's JavaScript context will expose global objects and functions like `window`, `document`, and `console`. This is illustrated in figure 6.2 and involves some overlap because both Node and Chromium have a `console` object.

When this is done, the JavaScript context for Node.js can be copied into the JavaScript context for Chromium.

Although that sounds quite easy, the reality is that there's a bit more glue involved for Node.js and Chromium to work together—the main event loop used by both has to be integrated.

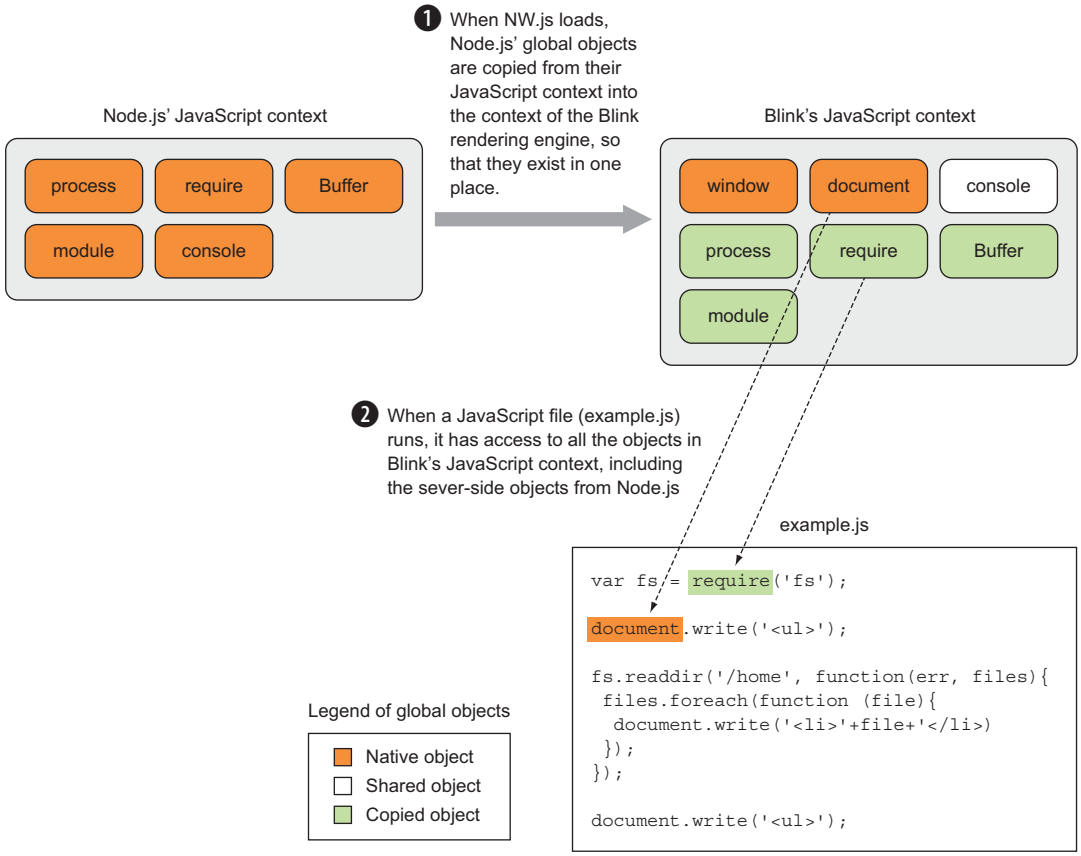

Figure 6.2 How NW.js handles copying the JavaScript context for Node.js into Chromium's JavaScript context

6.1.2 Integrating the main event loop

As discussed in section 5.1.3, Node.js uses the event loop programming pattern to handle executing code in a non-blocking, asynchronous fashion. Chromium also uses the event loop pattern to handle the asynchronous execution of its code.

But Node.js and Chromium use different software libraries (Node.js uses libuv, and Chromium uses its own custom C++ libraries, known as `MessageLoop` and `MessagePump`). To get Node.js and Chromium to work together, their event loops have to be integrated, as illustrated in figure 6.3.

When the JavaScript context for Node.js is copied into Chromium's JavaScript context, Chromium's event loop is adjusted to use a custom version of the `MessagePump` class, built on top of libuv, and in this way, they're able to work together.

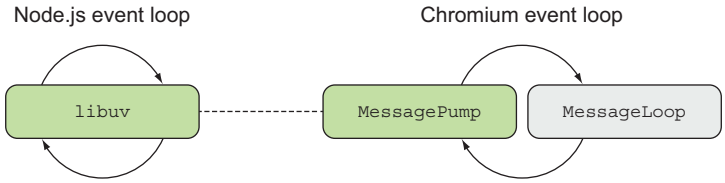

Figure 6.3 NW.js integrates the event loops of Node.js and Chromium by making Chromium use a custom version of `MessagePump`, built on top of libuv.

6.1.3 Bridging the JavaScript context between Node and Chromium

The next step to completing the integration of Node with Chromium is to integrate Node's `start` function with Chromium's rendering process. Node.js kicks off with a `start` function that handles executing code. To get Node.js to work with Chromium, the `start` function has to be split into parts so that it can execute in line with Chromium's rendering process. This is a bit of custom code within NW.js that's used to monkey-patch the `start` function in Node.

Once this is done, Node is able to work inside of Chromium. This is how NW.js is able to make Node.js operate in the same place as the front-end code that's handled by Chromium.

That rounds up a bit about how NW.js operates under the hood. In the next section, we'll explore the different approach taken by Electron.

6.2 How does Electron work under the hood?

Electron's approach shares some similarities in terms of the components used to provide the desktop framework, but differs in how it combines them. It's best to start by looking at the components that make up Electron. To see an up-to-date source code directory, take a look at http://mng.bz/ZQ2J.

Figure 6.4 shows a representation of that architecture at a less-detailed level. Electron's architecture emphasizes a clean separation between the Chromium source code and the app. The benefits of this are that it makes it easier to upgrade the Chromium component, and it also means that compiling Electron from the source code becomes that much simpler.

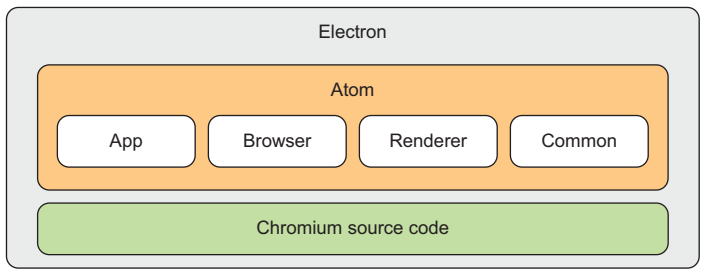

Figure 6.4 Electron's source code architecture. This diagram shows the main blocks of components that make up Electron.

The Atom component is the C++ source code for the shell. It has four distinct parts (covered in section 6.2.2). Finally, there's Chromium's source code, which the Atom shell uses to combine Chromium with Node.js.

How does Electron manage to combine Chromium with Node.js if it doesn't rely on patching Chrome to combine the event loops for Chromium and Node.js?

6.2.1 Introducing libchromiumcontent

Electron uses a single shared library called libchromiumcontent to load Chromium's content module, which includes Blink and V8. Chromium's content module is responsible for rendering a page in a sandboxed browser. You can find this library on GitHub at https://github.com/electron/libchromiumcontent.

You use the Chromium content module to handle rendering web pages for the app windows. This way, there's a defined API for handling the interaction between the Chromium component and the rest of Electron's components.

6.2.2 Electron's components

Electron's code components are organized inside Electron's Atom folder into these sections:

- App
- Browser
- Renderer
- Common

We'll look at what each of those folders contains in a bit more detail.

App

The App folder is a collection of files written in C++11 and Objective-C++ that handles code that needs to load at the start of Electron, such as loading Node.js, loading Chromium's content module, and accessing libuv.

Browser

The Browser folder contains files that handle interacting with the front-end part of the app, such as initializing the JavaScript engine, interacting with the UI, and binding modules that are specific to each OS.

Renderer

The Renderer folder contains files for code that runs in Electron's renderer processes. In Electron, each app window runs as a separate process, because Google Chrome runs each tab as a separate process, so that if a tab loads a heavy web page and becomes unresponsive, that tab can be isolated and closed without killing the browser and the rest of the tabs with it.

Later in this book, we'll look at how Electron handles running code in a main process, and how app windows have their own renderer processes that run separately.

COMMON

The Common folder contains utility code that's used by both the main and renderer processes for running the app. It also includes code that handles integrating the messaging for Node.js' event loop into Chromium's event loop.

Now you have an idea of how Electron's architecture is organized. In the next section, we'll look at how Electron handles rendering app windows in a process that's separate from the main app process.

6.2.3 How Electron handles running the app

Electron handles running apps differently than NW.js. In NW.js, the back-end and front-end parts of the desktop app share state by having the Node.js and Chromium event loops integrated and by having the JavaScript context copied from Node.js into Chromium. One of the consequences of this approach is that the app windows of an NW.js app end up sharing the same reference to the JavaScript state.

With Electron, any sharing of state from the back-end part of the app to the front-end part and vice versa has to go through the ipcMain and ipcRenderer modules. This way, the JavaScript contexts of the main process and the renderer process are kept separate, but data can be transmitted between the processes in an explicit fashion.

The ipcMain and ipcRenderer modules are event emitters that handle interprocess communication between the back end of the app (ipcMain), and the front-end app windows (ipcRenderer), as shown in figure 6.5.

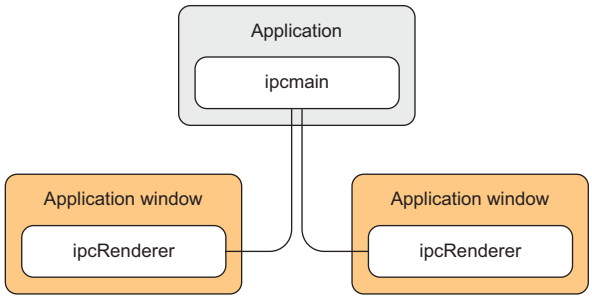

Figure 6.5 How Electron passes state via messaging to and from the app windows. In Electron, each app window has its own JavaScript state, and communicating state to and from the main app process happens via interprocess communication.

This way, you have greater control over what state exists in each app window as well as how the main app interacts with the app windows.

Regardless of which desktop framework you choose to build your app with, keep in mind how you want data to be accessed and altered within your app. Depending on what your app does, you may find that one framework is better suited to your needs than the other, and in cases where you're working with those desktop app frameworks already, you'll want to keep in mind how NW.js and Electron handle JavaScript contexts.

Now let's take a closer look at how Electron and NW.js make use of Node.js.

6.3 How does Node.js work with NW.js and Electron?

Node.js interacts with the hybrid desktop environments of NW.js and Electron similarly to server-side apps. But to understand the few differences, we'll look at the way Node.js is integrated into NW.js.

6.3.1 Where Node.js fits into NW.js

NW.js's architecture consists of a number of components, Node.js being one of them. NW.js uses Node.js to access the computer's file system and other resources that would otherwise not be available due to web browser security. It also provides a way to access a large number of libraries through npm (figure 6.6).

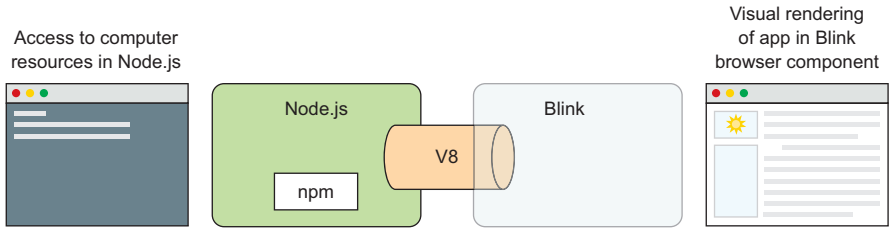

Figure 6.6 How Node.js is used within NW.js for desktop apps

NW.js makes Node.js available through the context of the embedded web browser, which means you can script JavaScript files that access both Node.js's API and API methods related to the browser's JavaScript namespace—such as the `WebSocket` class, for example. In earlier examples in the book, you've written code that has accessed Node.js's file system API in the same file that also accesses the DOM in the screen.

This is possible through the way that NW.js has merged the JavaScript namespaces of Node.js and the Blink rendering engine, as well as merged the main event loops of both, allowing them to operate and interact in a shared context.

6.3.2 Drawbacks of using Node.js in NW.js

Because of how NW.js merges the JavaScript contexts of the Blink rendering engine and Node.js, you should be aware of some of the consequences that come with this approach. I'll describe what those things are and how you can handle them so that they don't trip you up.

THE NODE.JS CONTEXT IS ACCESSIBLE TO ALL WINDOWS

I've talked about Node.js and Blink sharing the same JavaScript context, but how does that work in the context of an NW.js app where there are multiple windows?

In Blink, each window has its own JavaScript context, because each window loads a web page with its own JavaScript files and DOM. The code in one window will operate

in the context of that window only, and not have its context leak into another window—otherwise, this would cause issues with maintaining state in the windows as well as security issues. You should expect the state that exists in one window to be isolated to that window and not leak.

That said, NW.js introduces a way to share state between windows via the way that Node.js's namespace is loaded into the namespace of Blink to create a shared JavaScript context. Even though each window has its own JavaScript namespace, they all share the same Node.js instance and its namespace. This means there's a way to share state between windows through code that operates on Node.js's namespace properties (such as the API methods), including via the `require` function that's used to load libraries. Should you need to share data between windows in your desktop app, you'll be able to do this by attaching data to the *global* object in your code.

COMMON API METHODS IN CHROMIUM AND NODE.JS

You may know that both Node.js and Blink have API methods with the same name and that work in the same way (for example, `console`, `setTimeout`, `encodeURIComponent`). How are these handled? In some cases, Blink's implementation is used, and in other cases, Node.js's implementation is used. NW.js opts to use Blink's implementation of `console`, and for `setTimeout`, the implementation used depends on whether the file is loaded from a Node.js module or from the desktop app. This is worth keeping in mind when you're using those functions, because although they're consistent in their implementations of inputs and outputs, there might be a slight difference in speed of execution.

6.3.3 How Node.js is used within Electron

Electron uses Node.js along with Chromium, but rather than combining the event loops of Node.js and Chromium together, Electron uses Node.js's `node_bindings` feature. This way, the Chromium and Node.js components can be updated easily without the need for custom modification of the source code and subsequent compiling.

Electron handles the JavaScript contexts of Node.js and Chromium by keeping the back-end code's JavaScript state separate from that of the front-end app window's state. This isolation of the JavaScript state is one of the ways Electron is different from NW.js. That said, Node.js modules can be referenced and used from the front-end code as well, with the caveat that those Node.js modules are operating in a separate process to the back end. This is why data sharing between the back end and app windows is handled via inter-process communication, or *message passing*.

If you're interested in learning more about this approach, check out this site from GitHub's Jessica Lord: http://jlord.us/essential-electron/#stay-in-touch.

6.4 Summary

In this chapter, we've exposed some differences between NW.js and Electron by exploring how their software components work under the hood. Some of the key takeaways from the chapter include the following:

- In NW.js, Node.js and Blink share JavaScript contexts, which you can use for sharing data between multiple windows.
- This sharing of JavaScript state means that multiple app windows for the same NW.js app can share the same state.
- NW.js uses a compiled version of Chromium with custom bindings, whereas Electron uses an API in Chromium to integrate Node.js with Chromium.
- Electron has separate JavaScript contexts between the front end and the back end.
- When you want to share state between the front end and back end in Electron apps, you need to use message passing via the ipcMain and ipcRenderer APIs.

In the next chapter, we'll look at how to use the various APIs of NW.js and Electron to build desktop apps—specifically, at the way in which you can craft an app's look and feel. It will be more visual, and hopefully more fun.

JAVASCRIPT/WEB DEVELOPMENT

Cross-Platform Desktop Applications

Paul B. Jensen

Desktop application development has traditionally required high-level programming languages and specialized frameworks. With Electron and NW.js, you can apply your existing web dev skills to create desktop applications using only HTML, CSS, and JavaScript. And those applications will work across Windows, Mac, and Linux, radically reducing development and training time.

Cross-Platform Desktop Applications guides you step by step through the development of desktop applications using Electron and NW.js. This example-filled guide shows you how to create your own file explorer, and then steps through some of the APIs provided by the frameworks to work with the camera, access the clipboard, make a game with keyboard controls, and build a Twitter desktop notification tool. You'll then learn how to test your applications, and debug and package them as binaries for various OSs.

What's Inside

- Create a selfie app with the desktop camera
- Learn how to test Electron apps with Devtron
- Learn how to use Node.js with your application

Written for developers familiar with HTML, CSS, and JavaScript.

Paul Jensen works at Starcount and lives in London, UK.

"You will be shocked by how easy it is to write a desktop app!"
—From the Foreword by Cheng Zhao, Creator of Electron

"Write-once/run-anywhere just became a real thing."
—Stephen Byrne, Dell

"The definitive guide to two paradigm-shifting JavaScript frameworks. Indispensable."
—Clive Harber, Distorted Thinking

"Packed full of examples that will help you write cross-platform desktop apps using JavaScript."
—Jeff Smith, Ascension

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/cross-platform-desktop-applications

ISBN-13: 978-1-61729-284-2
ISBN-10: 1-61729-284-2
54999
9 781617 292842

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]