Development and Deployment of Web Applications as Installable Desktop Applications Using Electron Framework

FH Campus 02

Maximilian Martin Wolf

30 October 2021

Contents

1 Introduction

1.1 What Is Electron?

In December 2012 software engineer Cheng Zhao joined GitHub's team, having previously worked for Intel developing node-webkit, with the task of porting the Atom editor from using Chromium Embedded Framework to node-webkit. Node-webkit being a Node.js module developed by Roger Wang which combined the browser engine used by Chromium, WebKit with Node.js, making Node.js modules accessible from JavaScript code running inside a web page.[**?**]

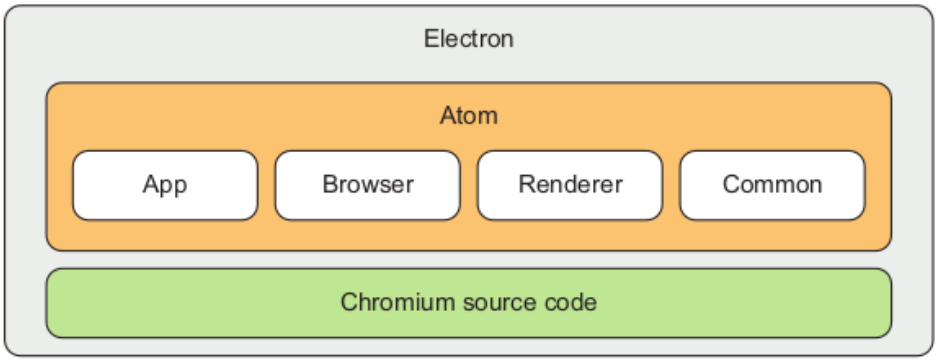
Porting to node-webkit proved difficult, so GitHub abandoned that approach, and it was decided that a new native shell for Atom would be created. Said shell was dubbed *Atom Shell* and shortly after development was finished and the Atom editor was open sourced by GitHub, Atom Shell soon followed suit and was renamed to *Electron*. Initially developed as a way to deliver an editor, numerous widely known applications like Slack, Discord and Visual Studio have started using Electron to develop and deliver their desktop applications. But what exactly is Electron?

Electron is a framework which allows for the development of cross-platform desktop applications using only popular web programming languages like HTML, CSS and JavaScript. While the advantages will be discussed in detail in the next chapter *Why Use Electron?*, the appeal to developers is obvious: Maintaining one codebase while being able to deliver the app to all desktop operating systems.

Now, as described previously the Electron framework serves the same purpose as node-webkit (later renamed to NW.js), but their approaches do differ in certain ways:[**?**] Without going into detail on how NW.js works, Electron and NW.js share some architectural similarities. However, there are some differences in how Electron combines Node.js with Chromium.

Architecturally, Electron places an emphasis on strict separation from Chromium source code, ass seen in figure 1.1. This looser integration allows for easier updates to the Chromium part of the source code, whereas with NW.js Chromium is patched to allow for Node.js and Chromium to use a shared Javascript state.[**?**] On the other hand, this means that Electron has separate JavaScript contexts: A *main* process which starts running with the app window and a *renderer* process for each individual window. Any sharing of state between these contexts, or simply put between the front- and back-end, has to pass through the *icpMain* and *icpRenderer* modules. This means that each JavaScript context is kept separate but data can be

Figure 1: A simplyfied representation of Electron's architecture.[**?**]



explicitly shared, allowing for greater control over what state exists in which app window.[**?**]

Electron itself (the part without Chromium) is made up of for different components: App, Browser, Renderer and Common. **App** contains code written in C++ and Objective-C++ responsible for loading Node.js and Chromium's content module. The **browser** folder contains code which handles interactions with the front-end. This is to say functionality such as loading the JavaScript engine, interacting with the UI and binding operating system specific modules. As for **renderer**, this component handles the different renderer processes. Because Chromium works by running each tab as an individual process, as to not crash the entire browser should one tab become unresponsive, each application window in Electron runs as its own process. **Common** contains code which is used by both the main and renderer processes for running the application. Among other things this folder also contains the integration of Node.js' event loop with Chromium's event loop.[**?**]

One question still left unanswered is why the combination of Node.js and Chromium is required in the first place? Due to browser security, apps would not have access to the computer's resources such as its file system, adding no benefit over a traditional webpage. With Node.js, this access is possible and as another benefit, using Node.js means access to a myriad of libraries through npm.

1.2 Why Use Electron?

oNow the next obvious question is why a framework such as Electron is needed at all. After all, it is "just" a way to have desktop applications developed as web applications. So why not just develop native desktop applications or traditional web applications depending on the use case? To answer this one has to examine the bigger picture:

Over the past decade it seems as though software pricing has moved from perpetual licenses towards subscription-based models. If one examines the data regarding end-user spending on cloud applications it is clear that the *Software as a Service* (SaaS) model has grown considerably in revenue and is projected to do so in the future: The worldwide end user spending for Software as a Service has increased from 31.4 billion US Dollars in 2015 to 120 billion US Dollars in 2020. It is projected this growth will progress with spending reaching 171.9 billion dollars in 2022.[**?**]

Furthermore, Gartner (2021) forecasts that by 2026, cloud spending will exceed 45% of all enterprise IT spending, up from 17% in 2021. This impressive growth can be attributed to two reasons. Reasons either technical and/or financial in nature. One financial benefit of SaaS is economies of scale: By hosting the application centrally and by extension aggregating users together, providers can benefit financially from leveraging economies of scale. At the simple end, this means benefiting from volume pricing on hardware such as data centers, servers, space and so on. Taking this idea further, SaaS providers can also cut costs by sharing hardware across their customers. It is not cost-effective to use one machine for each customer, instead resources should be shared and dynamically allocated on-demand to each customer's needs. Similarly, as user count increases, the cost of adding on single user decreases. These and other reasons are a big financial motivator for providers of software to switch to the SaaS model.

However, technological reasons play a large role as well. According to Jacobs (2005) the ever-increasing maturity of the Web is a major contributor for the rise in popularity of SaaS. Browsers are significantly more powerful than ever, internet access is more widespread and faster, and the number of robust frameworks for web development (be it front-end or back-end), make creating a complex web application easier than ever before.[**?**, **?**, **?**]

List of Figures