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 **3**
1.1 What Is Electron? 3
1.2 Why Use Electron? 5
1.2.1 Desktop Applications 7
1.2.2 Web Applications 8
1.2.3 Electron as a Solution 9

2 Method **10**
2.1 Developing with Electron 10
2.1.1 Requirements Analysis 10
2.1.2 Development Workflow with Electron 10
2.1.3 Finished Project 10

3 Analysis **10**
3.1 Results 10
3.2 Vaadin as an Alternative to Electron 10

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. (Jensen, 2017)

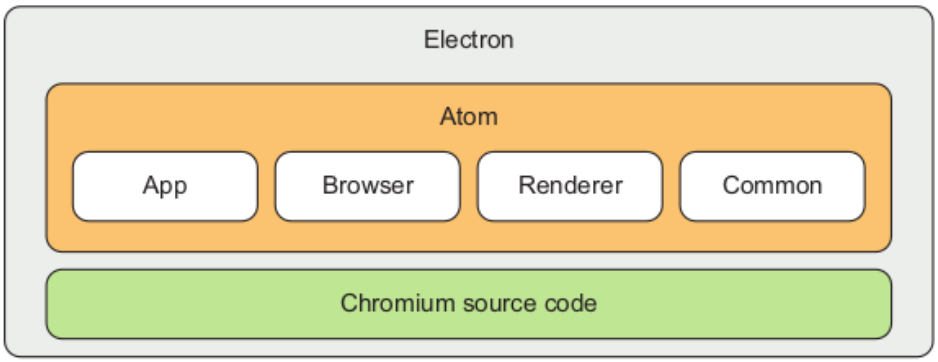
Porting to node-webkit proved difficult, so GitHub abandoned that approach, and it was decided that a new native shell for Atom would be created. Said shell was dubbed *Atom Shell* and shortly after development was finished and the Atom editor was open sourced by GitHub, Atom Shell soon followed suit 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: (Jensen, 2017) 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. (Jensen, 2017)

Figure 1: A simplified representation of Electron's architecture. (Jensen, 2017)



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 explicitly shared, allowing for greater control over what state exists in which app window. (Jensen, 2017)

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. (Jensen, 2017)

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?

Now 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. (Gartner, 2021)

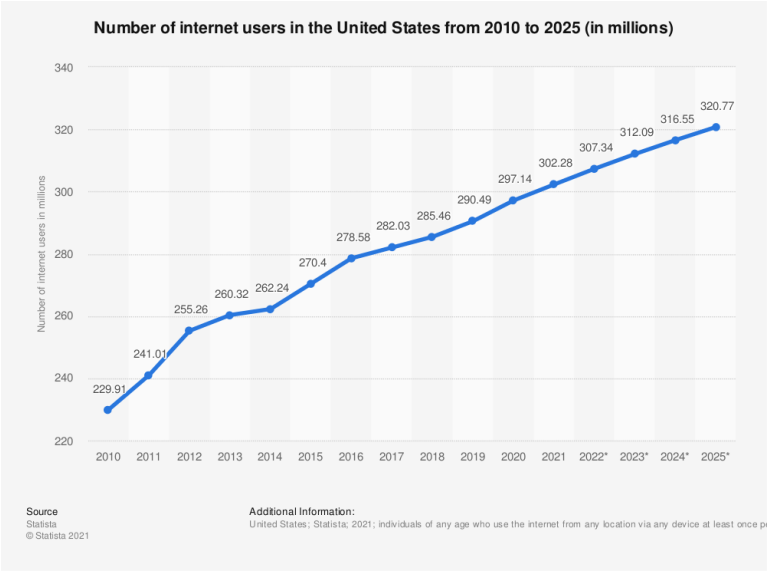
Furthermore, Gartner (2021) forecasts that by 2026, cloud spending will exceed 45% of all enterprise IT spending, up from 17% in 2021. This impressive growth can be attributed to two reasons. Reasons either technical and/or financial in nature. One financial benefit of SaaS is economies of scale: By 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. The *browser wars* of the mid-to-late nineties started with Microsoft and Netscape outdoing each other with new features, faster and overall better browsers leading to significant leaps in browser technology. (Jensen, 2017; Mozilla Foundation, 2021)

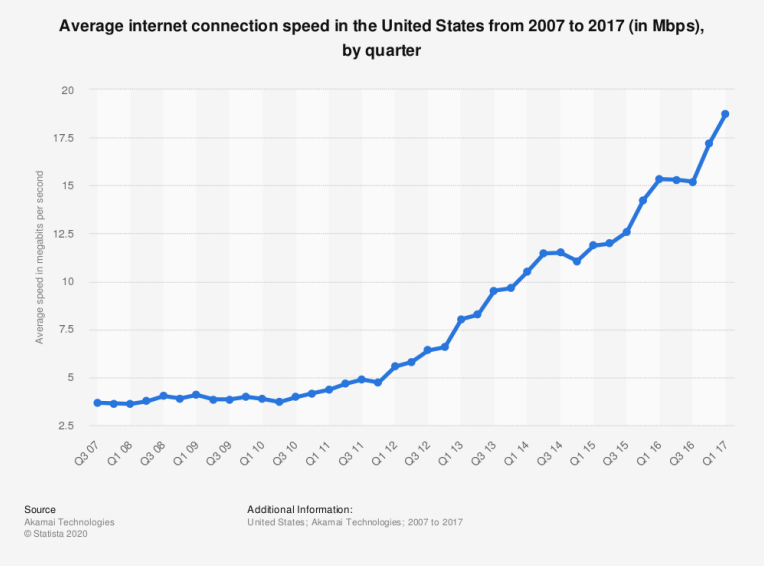
Furthermore, internet access is more widespread than ever. In the United States, the number of internet users rose from 229,91 million in 2010 to 302,28 million in 2021, which constitutes a 31% increase.

Figure 2: Number of internet users in the US from 2010 to 2025. (Statista, 2021)



And not only have the number of internet users risen over the past eleven years, the average connection speed increased as well over the same time period:

Figure 3: Average internet connection speed in the US 2007-2017. (Akamai Technologies, 2017)



As seen above, from Q3 2017 to Q1 2017, average internet speeds across the US rose by 410%. This allowed for much more elaborate websites where larger amounts of data have to be downloaded. Moreover, 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. However, while this explains why SaaS is often the billing model of choice, it doesn't fully explain why specifically web applications have risen in popularity. (Statista, 2021)

After all, SaaS can also be delivered as a Desktop Application, as seen with the Adobe Creative Suite for example. (Adobe Inc., 2021)

To answer this, one should examine how desktop applications and web applications differ in more detail.

1.2.1 Desktop Applications

Desktop applications used to be the standard way of delivering software to the end user. Users had to go to a store, buy a CD-ROM, check the system requirements and install the software on their machine. This does of course come with a number of benefits over web applications.

One advantage is that desktop applications aren't reliant on an internet connection. Web applications obviously fail here, as they are accessed over the internet. Furthermore, this reliance on an internet connection leads to more issues when the application is very feature-rich and/or has to support large files. An image editing software as a web application for example can run into limitations when being used with high-resolution images. Similarly, desktop applications start instantly without having to download resources over the internet. In such cases, desktop applications have an edge over comparable web applications. (Jensen, 2017)

There of course also benefits to using desktop applications from a developer's perspective. As a developer, one does not have to worry about users accessing their web applications over different web browsers as the choice of browser is at the user's discretion. This means not having to consider how different browsers interpret CSS, for example and always being sure about how the UI is being displayed and how the application's code is interpreted. Another benefit is the fact of tighter integration with the user's OS. Browser security limits the use of hardware and can lead to challenges for certain use cases. (Jensen, 2017)

Moreover, having an installable desktop application means not having to continuously support all the necessary infrastructure. There simply is no need to run servers, databases and such when the application is locally installed on each user's machine. (Jensen, 2017)

However, these benefits of desktop applications come with a significant drawback. Developing desktop applications requires developers to be proficient at languages like C++, Objective-C or C#. For a portion of developers, this can be a significant barrier to entry because it means learning an all-new language and in some cases even frameworks as well.

1.2.2 Web Applications

In contrast to desktop applications, the relatively low barrier for entry thanks to the easy of learning the basics of HTML, CSS and JavaScript in web development makes it much easier for developers to create complex web applications. With the amount of open source frameworks developers of web apps have a large selection of different solutions to fit their specific use case. Also, package managers like npm offer a large selection of readily

available, well established packages for developers to use and enhance their projects.

Another big advantage of web applications is that they are platform-independent. A web application can be reached on any reasonably modern device which runs a web browser. There is no need to create a separate version for all the operating system one wants to support and websites can also easily be accessed on mobile devices.

As described in the previous chapter 1.2.1, web applications need continuously running infrastructure such as web servers and databases. While this constitutes a disadvantage, it also comes with a big benefit for developers, as they can strictly control which version a client uses. Furthermore, the access to real-world data in said databases makes reproducing bugs much simpler. (Jacobs, 2005)

1.2.3 Electron as a Solution

However, web applications do come with disadvantages. As described in 1.2.1 web apps have their shortcomings such as browser security preventing access to a user's file system or having no access as soon as the internet connection fails.

This is where frameworks such as Electron manage to strike the near-perfect balance between desktop application and web app. For instance the drawback of not having access to a user's PC's file system does not apply to applications developed using electron, as the npm module *osenv* can for example retrieve the user's home folder among other environment settings. (Schlueter, 2012)

Additionally, the disadvantage of having to consider different browsers (and versions thereof) are a non-issue with electron because as Electron uses Chromium as outlined in 1.1. Furthermore, internet access is not a requirement with Electron and therefore applications can have some offline functionality as opposed to web apps.

These are some features and advantages of Electron, though not an exhaustive list. (Electron Framework, 2021)

Ultimately, it is at the developer's discretion which form of software to use. Desktop, web or frameworks such as Electron all have advantages

and disadvantages and it is important to consider which solution fits an application's and/or user's needs best.

2 Method

2.1 Developing with Electron

2.1.1 Requirements Analysis

2.1.2 Development Workflow with Electron

2.1.3 Finished Project

3 Analysis

3.1 Results

3.2 Vaadin as an Alternative to Electron

References

Adobe Inc. (2021). Plans and pricing for creative cloud apps and more. Retrieved December 14, 2021, from https://www.adobe.com/creativecloud/plans.html

Akamai Technologies. (2017, May 1). Average internet connection speed in the united states from 2007 to 2017 (in mbps), by quarter. Retrieved December 14, 2021, from https://bit.ly/31PSvFV

Electron Framework. (2021). Osenv. Retrieved December 14, 2021, from https://www.electronjs.org/docs/latest

Gartner. (2021, August 1). Global public cloud application services (saas) market size 2015-2022. Retrieved December 14, 2021, from https://bit.ly/3lZb8xV

Jacobs, D. (2005). Enterprise software as service. *Enterprise Distributed Computing.*

Jensen, P. B. (2017, May 1). *Cross-platform desktop applications using node, electron, and nw.js.* Manning Publications. https://bit.ly/3EPy0r1

Mozilla Foundation. (2021). The history of web browsers. Retrieved December 14, 2021, from https://www.mozilla.org/en-US/firefox/browsers/browser-history/

Schlueter, I. (2012). Osenv. Retrieved December 14, 2021, from https://www.npmjs.com/package/osenv

Statista. (2021, January 1). Number of internet users in the united states from 2010 to 2025. Retrieved December 14, 2021, from https://bit.ly/3yq6LkE

List of Figures

1 A simplified representation of Electron's architecture. (Jensen, 2017) 4
2 Number of internet users in the US from 2010 to 2025. (Statista, 2021) 6
3 Average internet connection speed in the US 2007-2017. (Akamai Technologies, 2017) 7