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.[Jensen, 2020]

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, 2020]

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. 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, 2020]

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, 2020]

1.2 Why Use Electron?

1.2.1 Desktop Applications

1.2.2 Web Applications

1.2.3 Electron as a Solution

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

[Jensen, 2020] Jensen, P. B. (2020). *Cross-Platform Desktop Applications Using Node, Electron, and NW.js.* Manning Publications.