Daniyar Alymkulov

DESKTOP APPLICATION DEVELOPMENT USING ELECTRON FRAMEWORK

Native vs. Cross-Platform

Bachelor's thesis
Information Technology

2019

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Daniyar Alymkulov	Bachelor of Engineering	April 2019

Thesis title

Desktop Application Development Using Electron Framework Native vs. Cross-Platform

43 pages
0 pages of appendices

Commissioned by

Xamk

Supervisor

Timo Mynttinen

Abstract

The goal of this thesis was to showcase Electron Framework and to study the differences between native .NET and cross-platform Electron frameworks. Electron is a new open-source framework for the development of cross-platform desktop GUI applications using front-end and back-end components originally developed for web applications.

This thesis contained theory and practice about .NET and Electron frameworks, what features they carry and how they work. The development part of the project includes designing and developing an application using Electron Framework based on the requirements.

The thesis concluded with analyzing the resulting application and drawing the conclusions about the viability of using Electron Framework in large scale projects. The application analysis was performed with various metrics and tools and done from different perspectives, including, but not limited to, performance and framework maturity.

Keywords

Programming, C#, native, cross-platform, .NET, JavaScript, Electron, comparison

CONTENTS

1 INTRODUCTION

The goal of this thesis was to showcase Electron Framework and to study the differences between native .NET and cross-platform Electron frameworks. In addition to native and cross-platform application development, there is a third type: hybrid, sometimes referred to as HTML5.

Native mobile apps are developed in a programming language native to the device or operating system, and require a specific app to be created for each target platform. Cross-platform mobile apps are developed using an intermediate language that is not native to the device's operating system. This means that some, if not all, of the code can be shared across target platforms. Finally, hybrid applications are cross-platform apps but render the user interface using an embedded web browser, leveraging the use of web technologies like HTML, CSS and JavaScript. (Rickard 2016.)

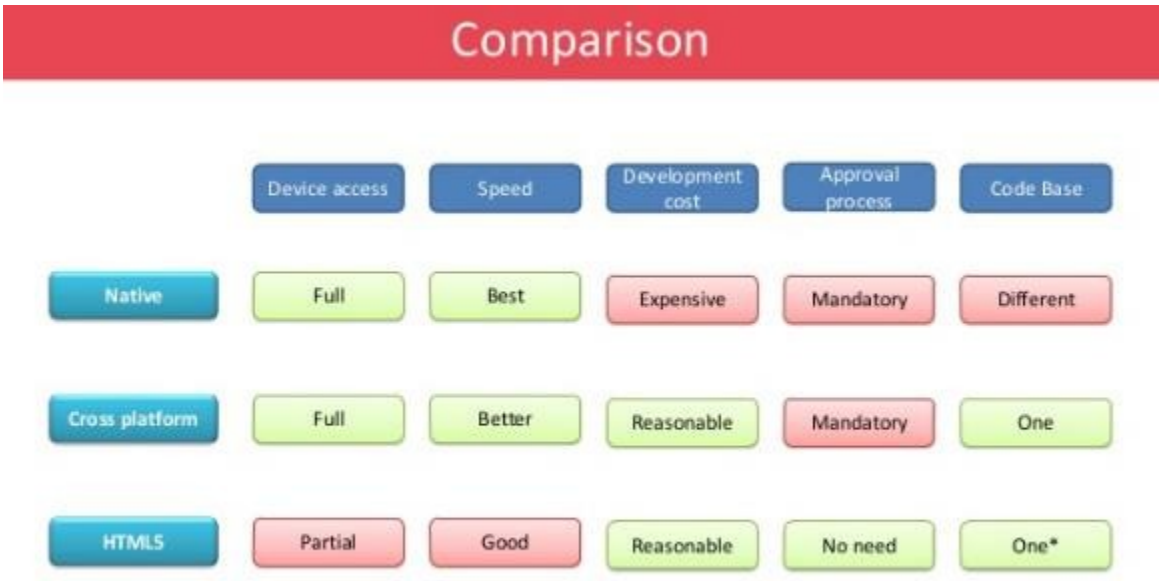

Figure 1. Native vs. Cross-Platform vs. HTML5 (Harish 2013)

Figure 1 shows the characteristics associated with application development and how native, cross-platform and HTML5 hybrid apps correlate to each other based on these characteristics. While native and cross-platform have existed for a long time, hybrid is relatively new, and thus, not subject to analysis in this project.

The rising star of cross-platform application development is Electron Framework. It allows creating native desktop applications using web technologies. Resulting apps look and feel just like native ones on any major desktop OS through one reusable code base written with HTML, CSS and JavaScript. User interfaces, nonetheless, use native elements and tools. Therefore, they have to be platform-specific. (Lord 2017.)

The project has two primary goals. The first goal is to explore Electron Framework and the second is to study the differences between native and cross-platform development processes. This will help developers consider the practicality of using Electron in their projects and see how it stands against a more mature technology. To realize these goals, I intend to build an application using Electron Framework.

2 THEORETICAL BACKGROUND

This chapter contains a closer look into the differences between native and cross-platform application development. Also, the technologies used in the practical part of this thesis are explained and discussed.

2.1 Native Applications

Native applications are written in a programming language or languages designed specifically for the chosen platform and use system APIs (Application Programming Interface) that utilize the resources of the platform to the fullest extent. Naturally, that results in better speed, stability and more functionality available to the developer. (Harish 2013.)

The advantages of developing a Native app:

- It gives access to all the features offered by the device and the OS. Thus, developers will not run up against something other apps can do that their app cannot.

- It allows utilizing advanced features, such as USB input, complex networking, memory management etc.
- There are no limitations in application performance and speed. This is crucial when creating a resource intensive app, such as a game, or an app that will be used by millions of users.
- Apps are built using technologies recommended and used by the device's manufacturer.
- It can provide a user interface native to the device or the OS. If done correctly, the app's user interface will update as the operating system updates over time.
- There are no limitations in terms of advanced user interface customization.

Disadvantages of developing a Native app:

- If the developer is targeting multiple platforms, two or more separate apps will need to be developed.
- No code can be shared between these separate apps.
- Development time can be slow when creating multiple applications.
- Testing time is affected, as two or more completely separate codebases need to be tested.

In the past, native approach was the only one available to the developers. However, with the passage of time, devices started evolving and incorporated an increasing number of features and processing power. Naturally, the cost of development of native applications has risen with time as well. Furthermore, technology has penetrated even the furthest corners of the world, which led to an enormous pressure on the developers to push their product to the market before their competitors. All of this gave rise to cross-platform applications.

2.2 Cross-Platform Applications

Cross-platform, on the other hand, concentrates on providing a unified experience through creating one code base for a variety of platforms. That

approach cuts down considerably in development times and costs. However, there are significant drawbacks that may contaminate the user experience and make the application unusable or undesirable for the consumer. (Rickard 2016.)

Advantages of developing a cross-platform app:

- Code can be shared between different versions of the apps across devices or operating systems.
- Development process is less resource intensive and may require fewer engineers.
- Shorter time of development and larger user reach.
- Future application support is easier, as updates can be pushed simultaneously for all platforms.

Disadvantages of developing a cross-platform app:

- Speed can be impacted, as some of the intermediate language may need to be interpreted “on the fly”.
- Access to the device and the OS features depend on the framework or plugin support.
- User interface customization is reliant on the framework support.

The idea of a cross-platform seems very appealing at the first look. Nonetheless, it has considerable drawbacks which should not be underrated.

2.3 Native vs. Cross-Platform

Each approach has its own pros and cons. Selecting the right path is dependent on the variety of factors (in no particular order): what kind of application is the development team making, what platform and programming language is it using, what user market is it aiming at?

Cross-platform solution is more suitable when creating an application that will not cause a great strain on the resources of your machine. Reusing the code base between the platforms will shorten the development time and reduce the amount

of resources needed. Therefore, the application will have a bigger target audience and can enter the market faster, which can be essential to the success of the product and the expected revenue. However, device functions and hardware may not be accessed by the framework which will limit the functionality of your application. That can be remedied by adding various plugins and libraries if available. However, writing good optimized code and not plugging too many external modules is essential. Otherwise, the app can quickly get “bloated” and become too slow or too big to be usable or comfortable to the average user. (Garbade 2018.)

Cross-platform development can be used to rapidly create applications not only for the desktop but also for mobile operating systems, which will give way to an even greater share of the market. Figure 2 shows the market share of the leading device form factors as reported by a web analytics service StatCounter, basing its estimate on web use. In fact, recently, after a long period of dominance, desktop operating systems have given way to the mobile.

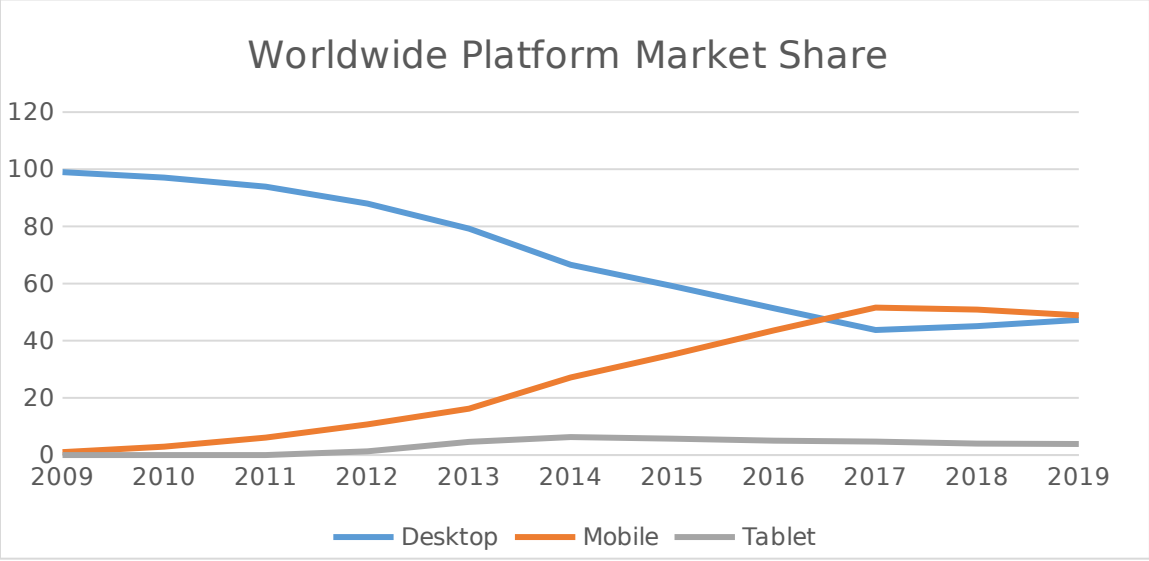

Figure 2. Desktop vs mobile market share (StatCounter 2019)

According to StatCounter, Windows OS unsurprisingly holds the leading position in the desktop market (Figure 3).

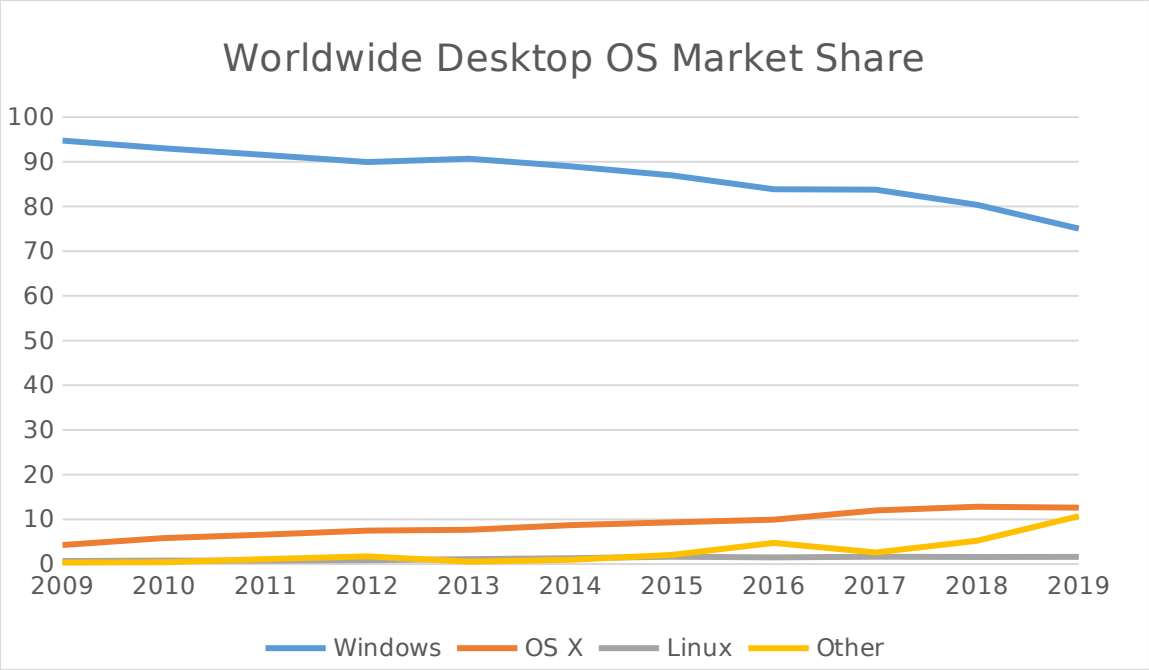

Figure 3. Desktop OS market share (StatCounter 2019)

However, Windows has gradually lost the market share with the rise of OS X (macOS previously) which has doubled worldwide in the last two years. This can be connected to the rise of iPhone and the creation of a unified ecosystem of devices by Apple. Ipso factum, OS X holds a big piece of the pie and therefore should not be neglected, if one wants to reach a maximum number of users and enter a bigger market.

On the other hand, native applications have their uses, too. For instance, video games and resource-intensive software like image and video editing software require a more detailed access to device functionality and a greater control over memory management. Access to native user interface elements can make the application look better and more in-sync with the rest of the OS. The APIs and SDKs for the platform are available out of the box, and if used correctly, can result in faster and more comfortable software with minimum effort. The documentation for native frameworks is usually more common and more detailed, because it is provided by the platform developers themselves. In addition, native applications have easier access to the distribution platforms (app stores) native to the OS, which gives greater visibility and makes distribution easier. (Rickard 2016.)

2.4 Electron Framework

Electron (previously known as Atom Shell) is an open-source framework created by Cheng Zhao, and now developed by GitHub. It allows for the development of desktop GUI applications using front and back-end components originally developed for web applications: Node.js run-time for the back-end and Chromium for the front-end. The applications themselves are written in HTML, CSS and JavaScript. (Maksimchik 2016.)

2.4.1 HTML, CSS, JavaScript

HTML, CSS and JavaScript are the most basic technologies for creating web pages and web applications. They form a triad of cornerstone technologies for the World Wide Web (Figure 4).

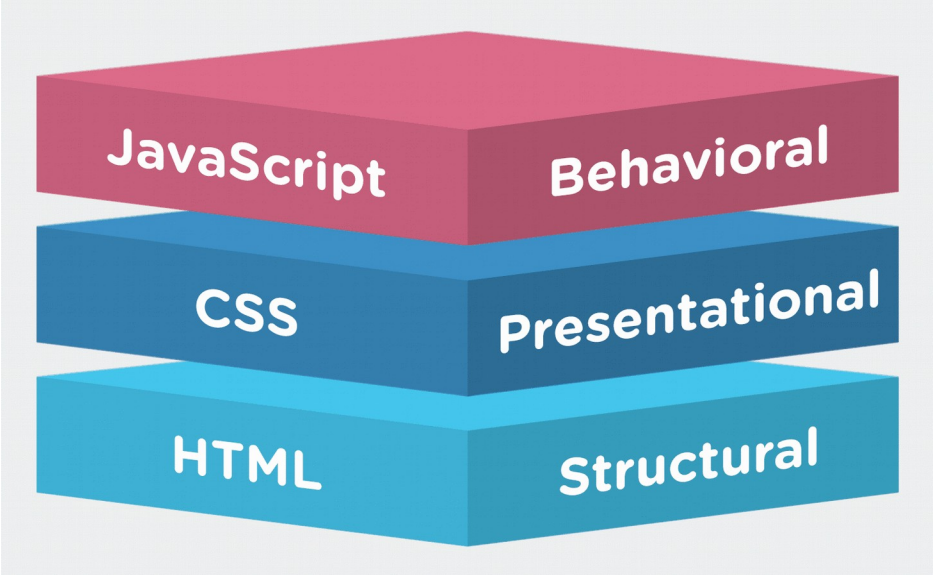

Figure 4. Basic technologies of World Wide Web (Jayasuriya 2015)

Hypertext Markup Language (HTML) is the standard markup language for creating electronic documents. Web browsers read HTML documents from a web server or from local storage and render the documents into visible web pages. HTML is at the core of every web page, regardless of the complexity of a site or number of technologies involved. It provides basic structure and allows adding

multimedia elements like images, audio and video to a web page. *Markup language* means that, rather than using a programming language to perform functions, HTML uses tags to identify different types of content and the purposes they each serve to the webpage. (Jayasuriya 2015.)

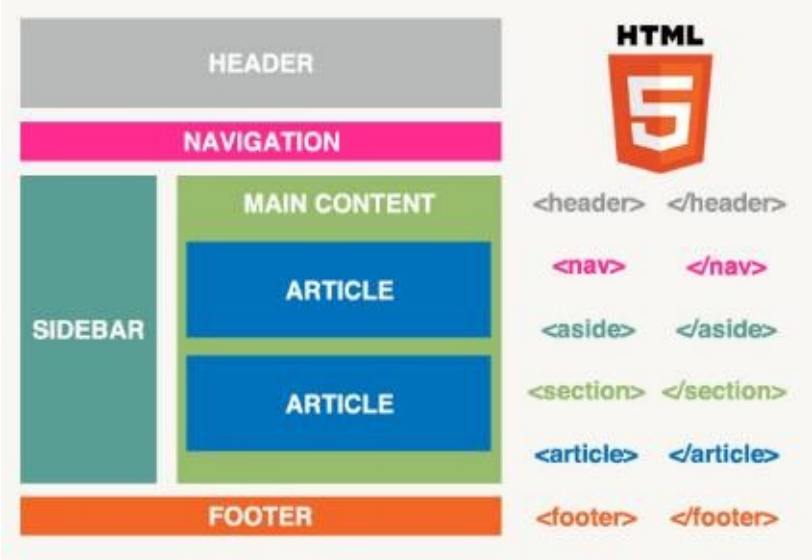

Figure 5. HTML structure (Mali 2016)

The relations between different tags and the composition of the resulting web page are illustrated in Figure 5.

Cascading Style Sheets (CSS) is a style sheet language for describing the way the documents written in HTML or any other markup language are presented. CSS dictates how the HTML elements of a website should actually appear on the front-end of the page and allows aligning the elements on a page and adding style elements like color, backgrounds, fonts, etc. (Figure 6). The name *cascading* means there is a specific priority scheme that determines which style rule applies if more than one rule matches a particular element (Wikipedia 2018).

Figure 6. CSS structure (Wikipedia 2018)

JavaScript is a high-level, interpreted programming language that adds interactivity to web pages. It allows adding interactive elements and modifying website content in response to the user's actions. JavaScript contains APIs for working with text, dates, arrays, regular expressions and DOMs (Document Object Model), but the language itself does not include any input/output functionality, such as storage, networking or graphics, relying for these upon the platform in which it is embedded. (Mali 2016)

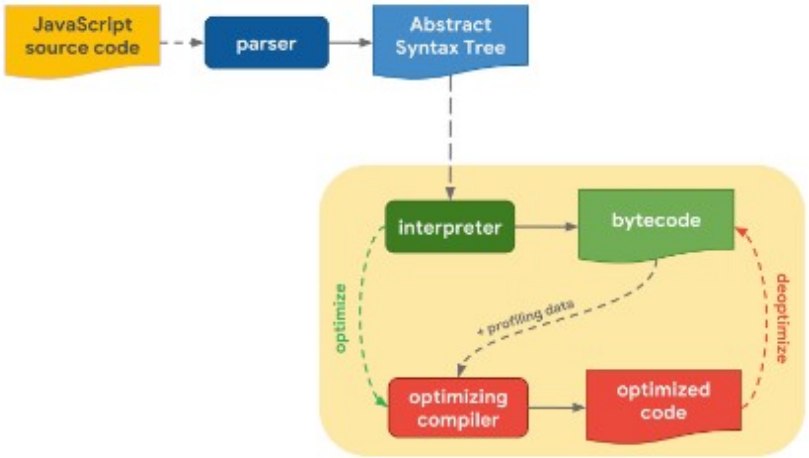

Figure 7. JavaScript engine (Bynens & Meurer 2018)

Figure 7 demonstrates the internal workings of the JavaScript engine. At the start of the process, it parses the source code and turns it into an Abstract Syntax Tree (AST). Based on that AST, the interpreter produces bytecode. To make the engine run faster, the bytecode can be sent to the optimizing compiler along with

profiling data. The optimizing compiler makes certain assumptions based on the profiling data it has, and then produces highly-optimized machine code. If any assumption is revealed to be incorrect, the optimized code goes back to the interpreter. (Bynens & Meurer 2018.)

2.4.2 Architecture

Electron consists of Chromium engine and Node.js runtime as well as a set of custom APIs for native operating system functions like open file dialogs, notifications, icons etc. (Figure 8).

Chromium is a web browser project developed and maintained by Google. It is fully functional and open-source (Google 2019). Despite similar names, it should not be mistaken for Google Chrome: both are made by Google and Chrome is based on Chromium and inherits vast majority of its code. Nonetheless, Chromium lacks a set of features that are present in Chrome (auto-update functionality, licensed codecs etc.). Chrome is not open-source and is distributed under a different license. (Hoffman 2018.)

Node.js is an asynchronous event-driven runtime for writing JavaScript on servers, accessing filesystems and networks, which means that it is designed to handle multiple connections concurrently and to hibernate if no work is present. Node.js was designed for optimal throughput and scalability in web applications with many I/O operations, as well as for real-time web apps. In addition to the standard out of the box Node APIs there is an enormous number of community modules written and hosted on **NPM**, a package manager for Node.js. (Node.js Foundation 2019.)

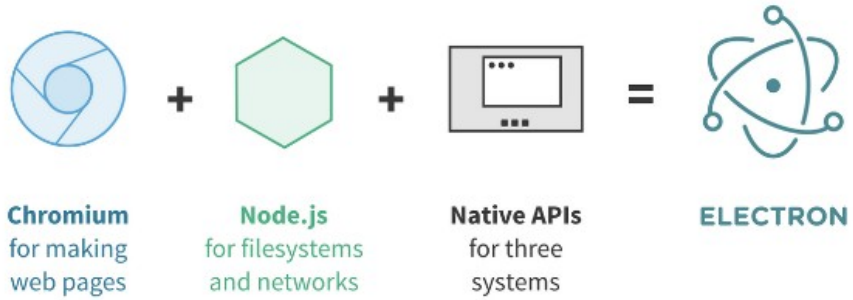

Figure 8. Electron architecture (Lord 2017)

Electron contains two types of processes: main and renderer. The main process handles the back-end functions from behind the scenes while the renderer process is responsible displaying the content and the GUI to the user. An Electron app always has only one main process, no more and no less. Since Electron uses Chromium for displaying web pages, Chromium's multi-process architecture is also used for running as many windows as needed with each window running in its own renderer process. (Lord 2017.)

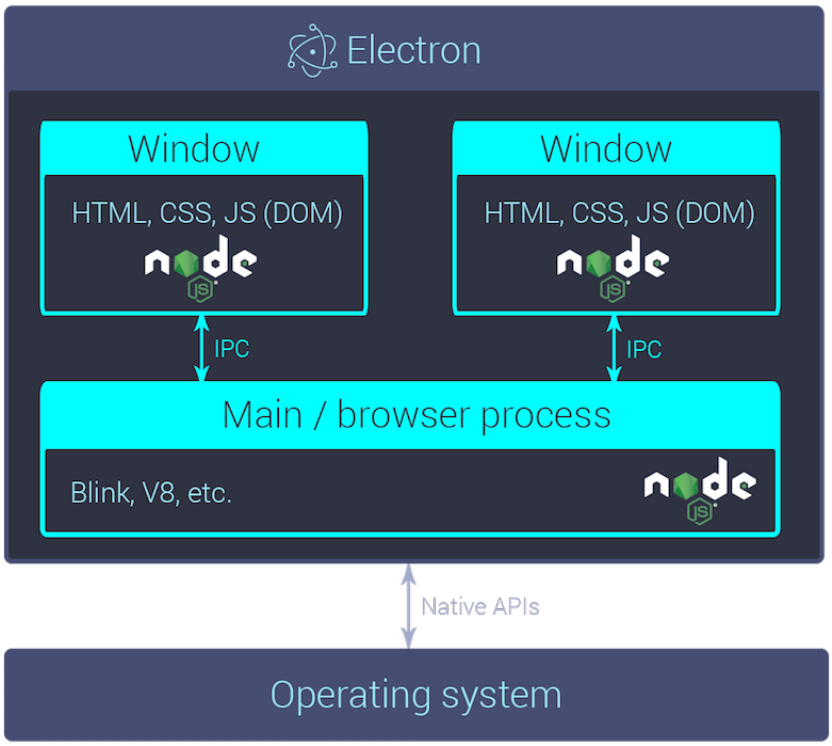

Figure 9. Electron application architecture (Lynch 2017)

The main process manages all web pages and their corresponding renderer processes as illustrated in Figure 9. Each renderer process is isolated and only

cares about the web page running on it, and for that reason Electron uses IPC (interprocess communication) to communicate information between processes.

2.4.3 Features

According to Maksimchik (2016), Electron consists of, but is not limited to, many features, as follows:

- **Cross-platform development using web technologies**
  Web technology stack makes application development fast and easy.
- **Native APIs for GUI components for all supported platforms:**
  This ensures look and feel of the application that is consistent with the rest of the OS.
- **Large number of modules**
  NPM gives the developer access to an enormous number of ready modules that can enhance the application and make it more feature-rich.
- **Auto-updating and crash reporting**
  Apps are capable of auto-updating themselves (only Win and OS X). Crash reports are submitted to the remote server for further analysis which makes finding bugs and issuing updates easier for the developer.
- **Debugging and profiling**
  Chromium's module finds performance bottlenecks and memory leaks.
- **Huge community**
  Electron is popular not only among the enthusiasts but also among professional developers. Due to that, it's not hard to find documentation for Electron and the internet is full of articles, videos and tutorials about it. In addition, it is safe to say that Electron will continue to receive updates and support for the foreseeable future.
- **Corporate support**
  Electron was developed by GitHub during the development of their code editor Atom. Furthermore, it is also used by Microsoft to develop their own code editor VS Code. Many other notable projects are built with Electron under the hood (Figure 10).

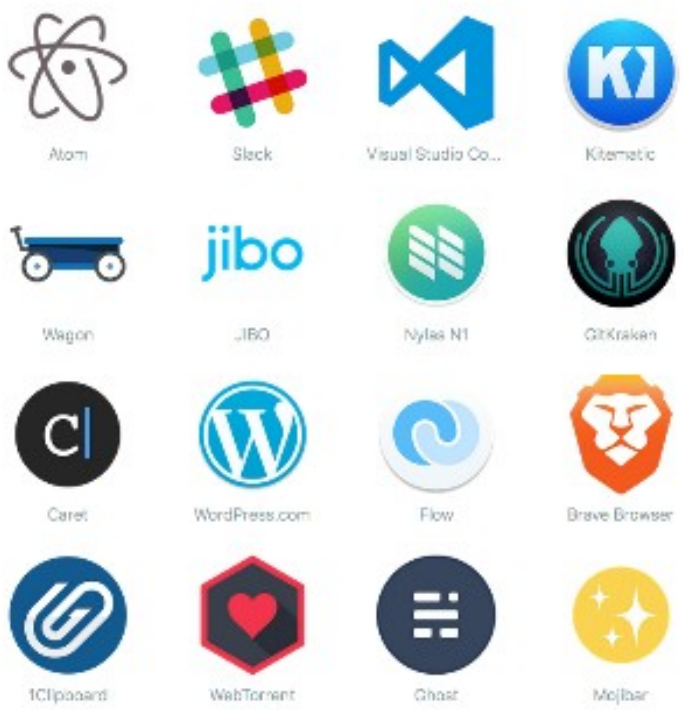

Figure 10. Projects that use Electron (GitHub 2019)

- **Installers**
  Electron supports a variety of tools for creating binaries or installers for all supported platforms and formats
- **Support for application stores**
  It is possible to create binaries specifically for the Windows and OS X stores. This makes publishing and distribution much easier (Maksimchik 2016.)

Electron's API list is extensive and provides plenty of functionality (GitHub Inc. 2019). Table 1 contains the most common of the APIs divided by the type of the process that can call them:

Table 1. List of Electron APIs

Main Process	Renderer Process	Both Processes
app	desktopCapturer	clipboard
browserWindow	fileObject	crashReporter
dialog	ipcRenderer	shell
ipcMain	remote	nativeImage
MenuItem	webFrame	process
powerSaverBlocker	window.open function and more	screen
systemPreferences		synopsis and more
tray and more		

2.5 .NET Framework

.NET is a development platform for building native applications for web, mobile, desktop, gaming, and IoT for Windows, OS X, Linux, Android and many others. With .NET, developers can use multiple languages, editors, and libraries. (Microsoft Corporation 2019.)

There are three implementations of .NET that are suited for different needs:

- **.NET Framework** is used to create websites, services, desktop apps on Windows OS.
- **.NET Core** is a cross-platform .NET implementation for creating websites, servers, and console applications for Windows, Linux, and macOS.
- **Xamarin/Mono** is a .NET implementation for running apps on all major mobile operating systems.

Figure 11 illustrates these implementations of the .NET platform and what they have in common. There are similarities and differences between .NET Framework and .NET Core

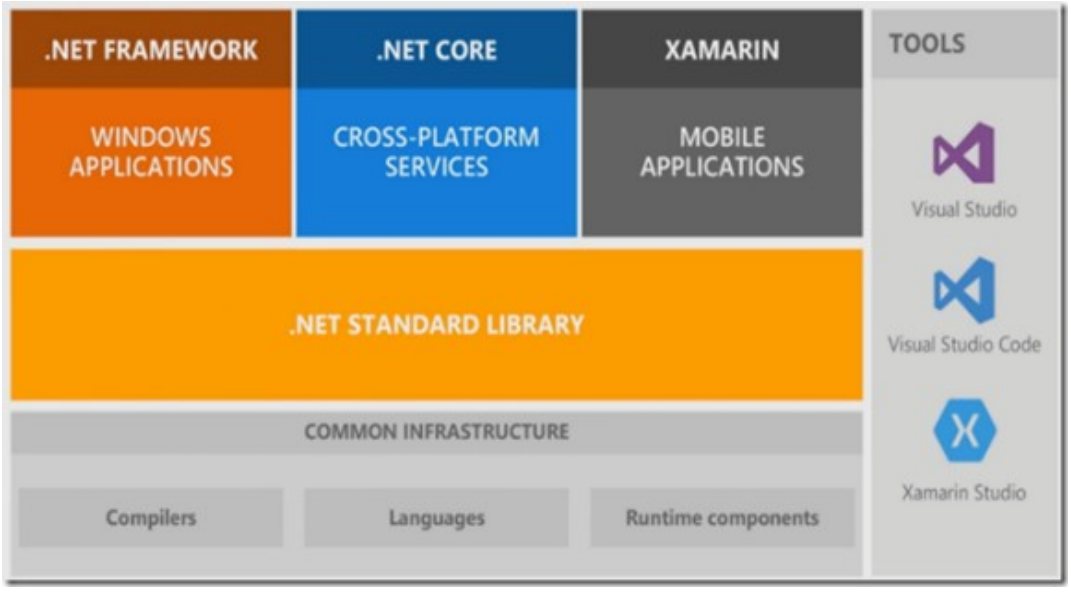

Figure 11. .NET platform (Kavalaparambi 2017)

.NET supports several languages for application development on their platform, for example:

- **C#** is an object-oriented and type-safe programming language.
- **F#** is a functional programming language for .NET.
- **Visual Basic** is an approachable language with a simple syntactical structure for building type-safe, object-oriented apps.

Despite .NET Core and Electron providing similar functionality in regards to development of cross-platform apps, the purpose of this thesis was to compare native and cross-platform development. Thus, I decided to use the .NET Framework for comparison with Electron Framework as opposed to .NET Core. Of all supported langauges, C# is by far the most popular because it was designed to work seamlessly with .NET from the start.

2.5.1 C#

C# is a general-purpose, multi-paradigm programming language created by Microsoft in conjunction with .NET. C# can be used with any .NET implementation.

Based on Ecma International (2017), C# was designed with the following goals in mind:

- C# is designed to be a modern, simple, general-purpose, object-oriented programming language.
- The language and its implementations must support strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection. Software robustness, durability, and programmer productivity are important.
- C# is intended for use in development of software components suitable for deployment in distributed environments.
- Source code and programmer portability are very important especially for those programmers already familiar with C and C++.
- Support for internationalization is very important.
- C# is intended to be suitable for writing applications for both hosted and embedded systems, from the very large that use sophisticated operating systems, down to the very small having dedicated function.
- Although C# applications are intended to be economical in memory and processing power requirements, the language was not intended to compete directly in performance and size with C or assembly language. (Ecma International 2017.)

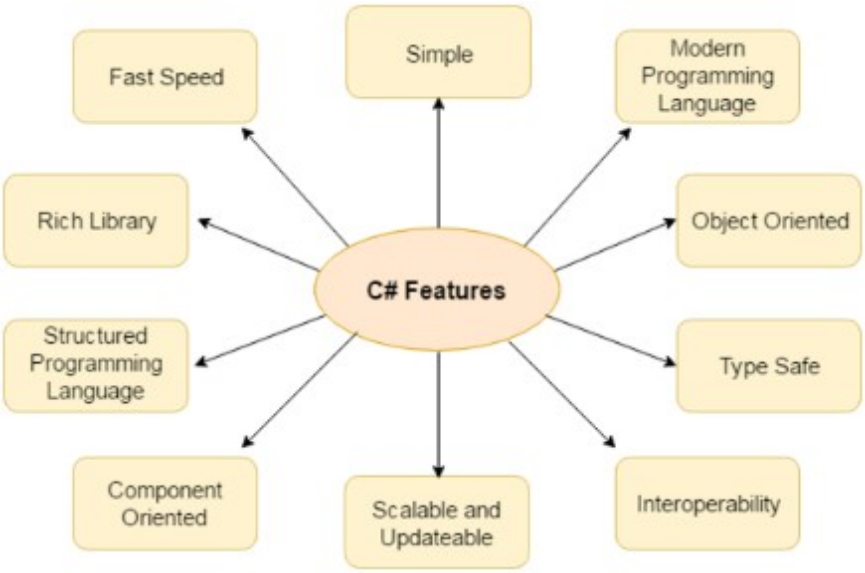

Figure 12. C# features (JavaTpoint 2018)

Figure 12 shows the features that C# possesses.

2.5.2 Architecture

The .NET Framework at its core is comprised of the common language runtime (CLR) and the .NET Framework class library (Microsoft Corporation 2015). In time, new versions of .NET have added new components and features, as shown in Figure 13.

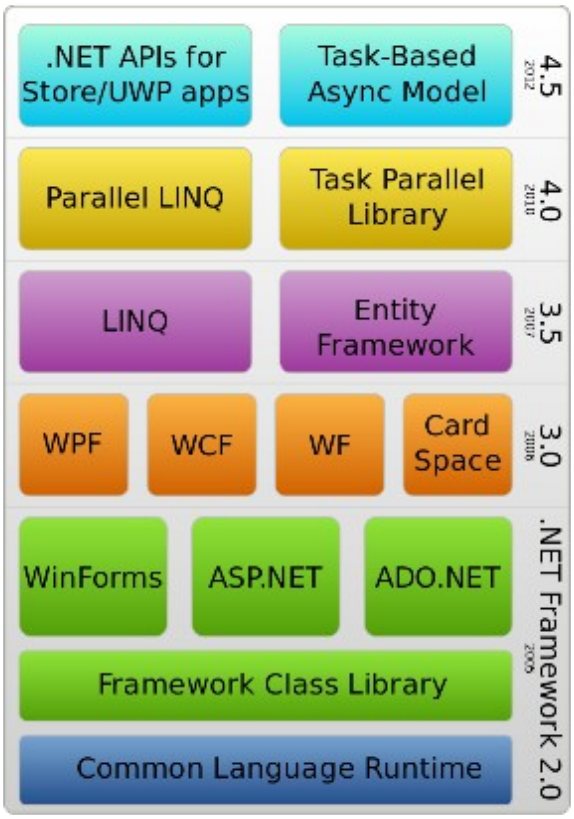

Figure 13. .NET Framework architecture (Wikipedia 2019)

CLR is the foundation of the the .NET Framework. It manages code at execution time, providing core services like memory management, thread management and remote access and execution, while also enforcing strict type safety and other forms of code accuracy that increase security and robustness. CLR handles object layout and manages references to objects automatically, removing them when they are no longer being used. This automatic memory management eliminates two of the most common application errors, memory leaks and invalid memory references. (Microsoft Corporation 2017.)

The class library is a comprehensive, object-oriented collection of reusable types that can be used during the development of apps allowing the developer to accomplish with ease common programming tasks, such as file access, data collection, database connectivity, and string management.

A program written for .NET Framework goes through many steps before executing. Figure 14 illustrates the compile and run-time of C# source code, .NET

Framework class libraries and the CLR.

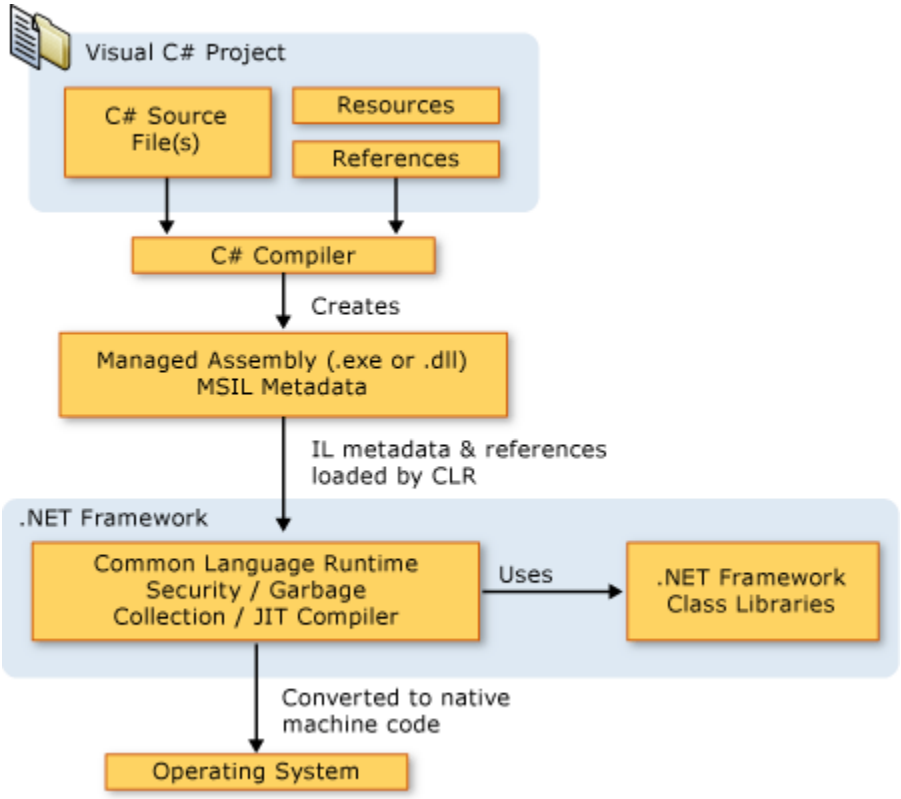

Figure 14. .NET Framework project execution (Microsoft Corporation 2017)

Project source code is compiled into an intermediate language (IL) that conforms to the CLR specifications. The IL code and resources are stored in the system as an assembly executable file. An assembly includes a manifest that contains information about the assembly types, version and security requirements.

When the C# program is executed, the assembly is loaded into the CLR, which might take different actions based on the information in the manifest. Then, if the security requirements are met, the CLR performs just-in-time (JIT) compilation to convert the IL code to native machine code. At last, this code can be directly loaded into computer memory and executed. (Microsoft Corporation 2017.)

2.5.3 Features

According to Microsoft Corporation (2017), with .NET Framework it is possible to:

- Execute code consistently whether it is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- Minimize software deployment and versioning conflicts.
- Execute code safely, including code created by an unknown or semi-trusted third party.
- Eliminate performance problems in scripted or interpreted environments.
- Make the developer experience consistent across a variety of application types, such as Windows and web-based apps.
- Build all communication according to industry standards to ensure that code based on the .NET Framework integrates seamlessly with any other code.

Key feature of the .NET Framework is language interoperability. Because the IL code produced by the C# compiler follows the Common Type Specification (CTS), IL code generated from C# can interact with code that was generated from the .NET versions of Visual Basic, Visual C++, or any one of more than 20 other CTS-compliant languages. A single assembly can contain several modules written in various .NET languages, and the types can refer to each other just as if they were written in the same language. (Microsoft Corporation 2017.)

Moreover, one thing every .NET implementation has in common is a base set of APIs called .NET standard library, as previously shown in Figure 11. Each implementation can also utilize additional APIs that are specific to the operating systems it runs on. In case of the .NET Framework, which is a Windows-only .NET implementation, it includes APIs for accessing the Windows registry.

With the .NET Framework it is possible to develop many various types of applications, such as the following:

- Console apps
- Windows GUI apps (Windows Forms)
- Windows Presentation Foundation (WPF) apps
- ASP.NET apps
- Windows services
- Service-oriented apps using Windows Communication Foundation (WCF)
- Workflow-enabled apps using Windows Workflow Foundation (WF)

In the end, the .NET Framework was created to produce highly-optimized applications for Windows. It has a major number of features designed specifically

for that purpose, not only on the framework level, but built-in inside the C# language as well.

3 DEVELOPMENT AND ANALYSIS

In this chapter, in order to demonstrate the advantages and drawbacks of developing in Electron Framework, an application is written using Electron as a case study. While the project covers both client and server-side, a bigger focus is given to how the framework is used to create a desktop application using web technologies.

3.1 Requirement Analysis

The chosen application is called Price Checker and is used shows the current price of gold, silver and various currencies including bitcoin. It has two windows: first one contains the current exchange rate of currencies and commodities, as well as price difference compared to the previous day, while the second one is used to add new currencies. The exchange rates are displayed in US dollars.

The API used to provide the exchange rates is currencylayer API. It is available at https://currencylayer.com/. Due to the constrains of the API, the only supported source currency is USD and all the exchange rates are displayed relative to it (currencylayer 2019).

3.2 Electron Application Development

This section gives a detailed description of the Electron application and of its development process.

3.2.1 Tools and Development Environment Setup

While there are many IDEs and text editors available that support JavaScript, the one used in this thesis is WebStorm IDE. WebStorm is tailored towards JavaScript and many JS frameworks, including Electron (JetBrains 2019).

In addition to the IDE itself, Node.js and NPM are also needed for proper functioning of Electron Framework. The installation file for Node.js includes npm which simplifies the process.

Once WebStorm and Node.js are installed, an empty folder for the project must be created. A most basic Electron application has the following project structure:

```
app-folder/
├── package.json
├── main.js
└── index.html
```

The starting point is a package.json file. It gives the information about the app and specifies the starting script which will run the main process. The following command will create that file based on user input:

```
npm init
```

The package.json file produced for this project is as follows:

```
{
  "name": "price-checker",
  "version": "0.3.0",
  "license": "MIT",
  "description": "Desktop price checker",
  "main": "main.js",
  "scripts": {
    "start": "node ."
  },
  "keywords": [
    "Price",
    "Cryptocurrency",
    "Electron",
    "Desktop"
  ],
  "author": "golddante"
}
```

The *start* property is used to specify the runtime of the application. To make it an Electron app, **node** must be changed into **electron**.

Now, Electron itself must be installed by executing the command below:

```
npm install --save-dev electron
```

This command installs Electron Framework and also saves it into the list of development dependencies of the package.json file.

Now, the project can be opened in the IDE for the user code. In package.json, main.js is the main script that executes the main process. The *app* and *BrowserWindow* modules are essential to the main process since they control application life and create native browser window. The code below is the bare minimum of an Electron app:

```
1. const {app, BrowserWindow} = require('electron');
2.
3. // This method will be called when Electron has finished
4. // initialization and is ready to create browser windows.
5. app.on('ready', () => {
6.   let mainWindow;
7.
8.   // Create the browser window.
9.   mainWindow = new BrowserWindow({
10.    width: 800,
11.    height: 600,
12.    frame: true,
13.    resizable: false,
14.    title: "Price Checker"
15.  })
16.
17.  // Emitted when the window is closed.
18.    mainWindow.on('closed', function () {
19.      mainWindow = null
20.    })
21.  );
```

At this point, the app just opens the main window without any content inside (Figure 15).

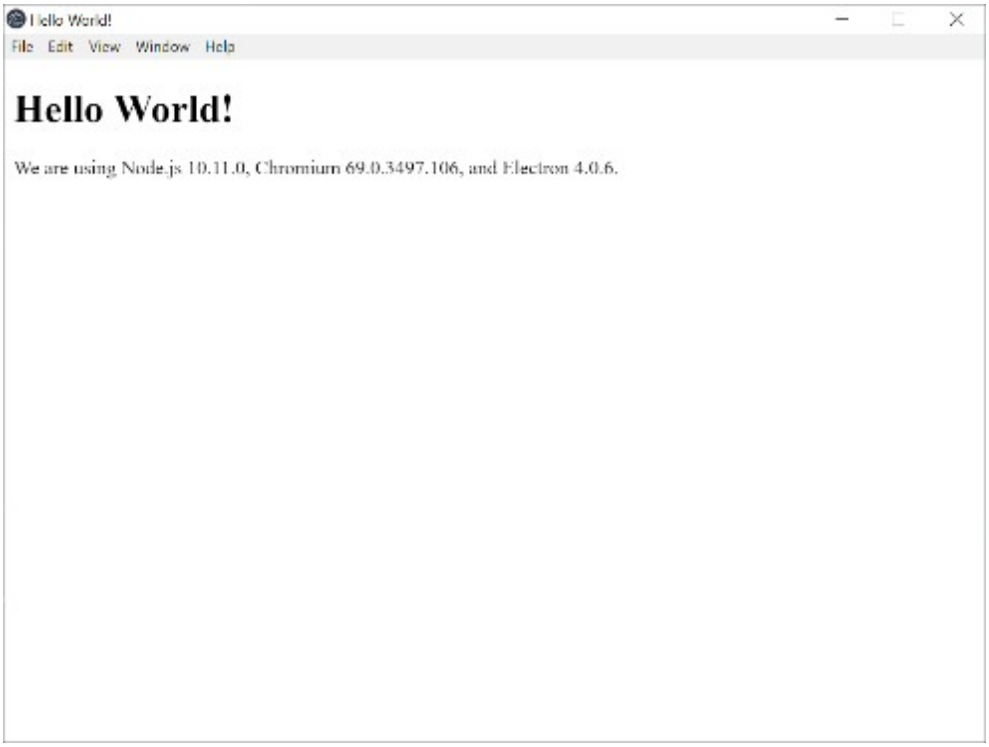

Figure 15. Initial main window

The contents of the main window can now be modified by creating a renderer process.

3.2.2 Main Window Implementation

As stated in Requirement Analysis, the main window displays the current exchange rates for all the supported currencies and assets. When loading a page in the window directly, users can see the page load incrementally and experience white flickering on the screen, which is not a good experience for a native app. The solution to this problem is rather straightforward: the main window is disabled until the application is ready. The following command shows a separate property that must be present inside the *mainWindow* object for that to happen:

```
mainWindow= new BrowserWindow({ show: false })
```

The function below must be called to show the object once the page is ready:

```
mainWindow.on('ready-to-show', () => mainWindow.show());
```

When all the windows are closed, the app quits. However, on OS X it is common for applications and their menu bar to stay active until the user quits explicitly with cmd + Q which is why a conditional statement is added to the closing function. Additionally, OS X users can recreate a window in the app when the dock icon is clicked and there are no other windows open using the command below:

```
1. app.on('window-all-closed', () => {if (process.platform !==
'darwin') app.quit()});

2. app.on('activate', () => {if (mainWindow === null)
createWindow()});
```

All this enhances the OS X user's experience with the application by leveraging the native techniques of the operating system.

The hierarchy of the completed application is displayed in Figure 16. Besides the source code, Bootstrap library was used. Bootstrap is a popular library for creating responsive interfaces with a big collection of UI elements.

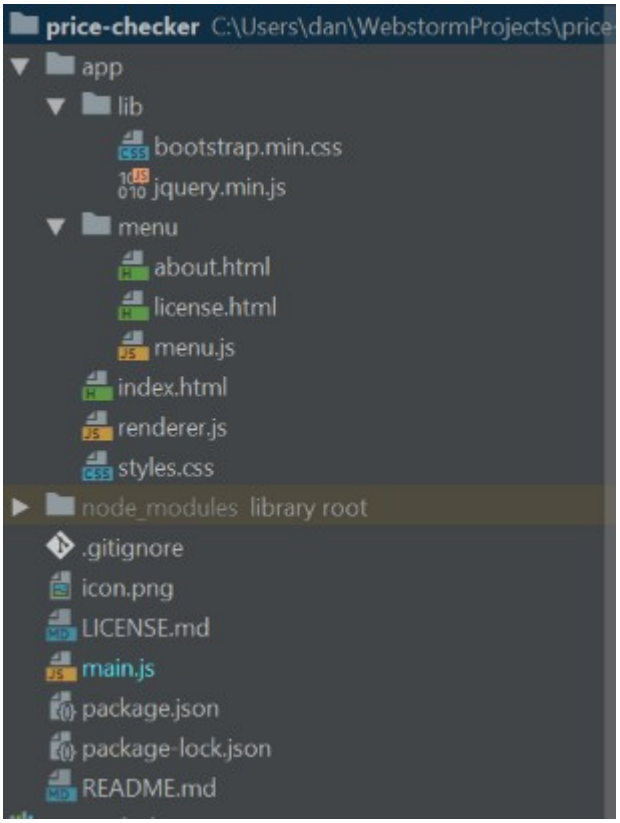

Figure 16. Code hierarchy

Once the main window is opened, main.js calls for index.html to show the actual content of the page (Figure 17).

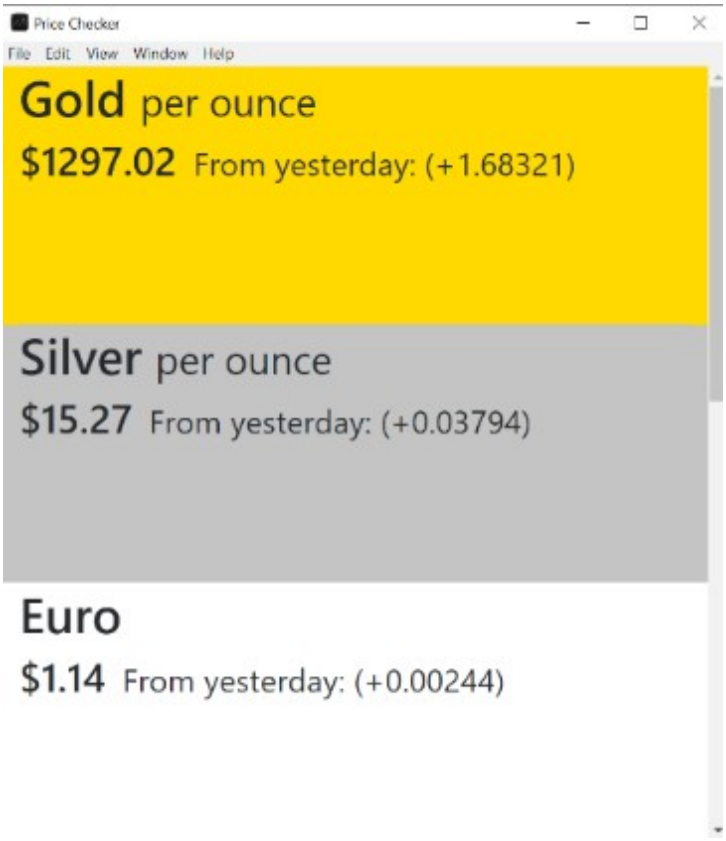

Figure 17. Main window

Index.html uses Bootstrap to make the interface responsive and comfortable on a variety of different screen resolutions and sizes.

Each row is a separate element and is updated separately once the information is available from the API. The following is an example of the rowwhich is responsible for the price of gold:

```
<!-- Gold Row -->
    <div class="row gold">
        <div class="col-12">
            <h1>Gold
                <small>per ounce</small>
            </h1>
            <h2>
                <span id="gold-price" class="price"></span>
                <small id="gold-change" class="change"></small>
             </h2>
```

```
11.          </div>
12.      </div>
```

Since index is an HTML document, it is only capable of displaying the information. To get the currency rates, it requests renderer.js file which contains all the logic for the main window.

In renderer file, the main function is getting current exchange rates. This is handled by standard *HTTP* module that comes with Node.js (Agnew 2017). Since HTTP requests are asynchronous, all the operations with the data from the API must be handled within the methods. Otherwise, the data will not be available outside the method, since the script will continue running without waiting for the REST API to finish its operation.

```
1. http.get('http://apilayer.net/api/live?
   access_key=05b40302019c971cdfaf3d24fc58155e' +
   '&currencies=&source=USD&format=1', (res) => {
2.   let rawData = '';
3.   res.on('data', (chunk) => {
4.     rawData += chunk;
5.   });
6.   res.on('end', () => {
7.     try {
8.       const rates = JSON.parse(rawData).quotes;
9.       for (let symbol in currencies) {
10.        let currentRate = (1 / rates[currencies[symbol]]);
11.        document.getElementById(`${symbol}-price`).innerHTML =
12.        '$' + currentRate.toFixed(2);
13.      }
14.    } catch (e) {
15.      console.error(e.message);
16.    }
17.  });
18. }).on('error', (e) => {
19.          console.error(`Got error: ${e.message}`);
20. });
```

The code above gets the current exchange rates from the currencylayer API.

Additionally, renderer.js contains functions for calculating the changes between current and previous days' currency rates.

3.2.3 Native Menu Implementation

The default menu of Electron applications is shown in Figure 18.

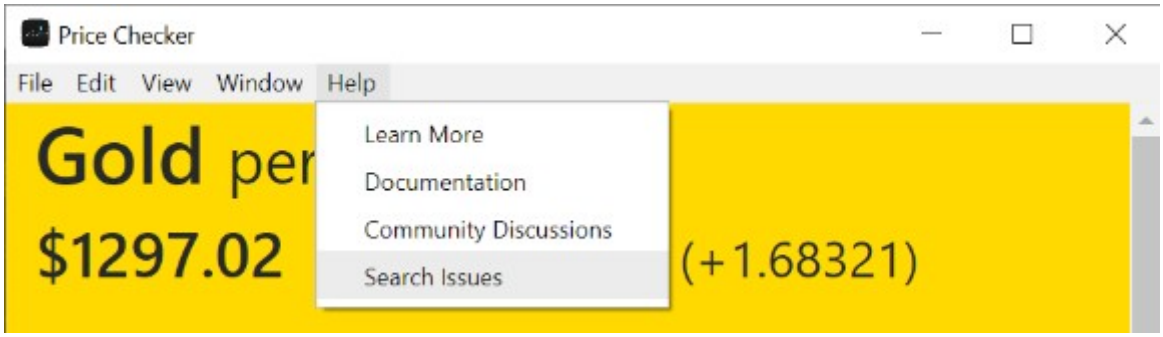

Figure 18. Default menu

However, Electron contains API for creating custom native menus. Index.html includes a reference to the menu.js file that handles the creation and customization of a custom native menu that is used as an application menu. The code below loads th necessary Electron modules into the script to create custom menus:

```
1. const {remote, shell, ipcRenderer} = require('electron');
2. const Menu = remote.Menu;
```

Remote is a module that allows the use of main process modules from the renderer process. In this case, it gives access to the *Menu* module which creates native application menus and context menus.

To construct a menu, a method called Menu.buildFromTemplate(template) is applied. Generally, a template is an array of options with fields that will become properties of the constructed menu items. For instance, to create an option to turn the application full screen the following code is required:

```
1. label: 'Toggle Full Screen',
2. accelerator: (function () {
3.     if (process.platform === 'darwin')
4.         return 'Ctrl+Command+F';
5.     else
6.         return 'F11';
7. })(),
8. click: function (item, focusedWindow) {
9.     if (focusedWindow)
```

```
10.       focusedWindow.setFullScreen(!focusedWindow.isFullScreen());
11.   }
```

Accelerator is an option that contains a key code to define keyboard shortcuts throughout the application. Once again, a conditional statement is used to specify the key code based on the OS.

Role is another option that can enhance native application experience. Roles allow menu items to have predefined behaviors that can be either universal to all systems or specific to a single OS.

Even though the *remote* module gives access to the main process, communication between main and renderer processes is limited. The code below uses *ipcRenderer* module to communicate asynchronously with the main process.

```
1. label: 'Help',
2. role: 'help',
3. submenu: [
4.     {
5.          label: About Price Checker,
6.         click: () => ipcRenderer.send('show-about')
7.     }
8. ]
```

Figure 19 shows the resulting custom menu with two top-level options each containing several sub options.

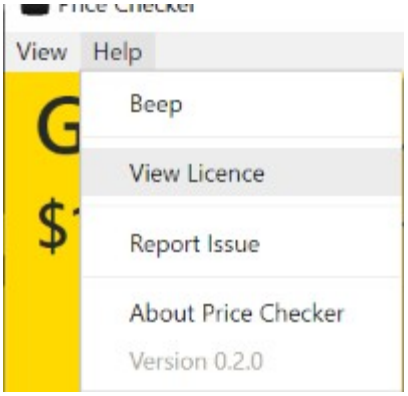

Figure 19. Custom menu

When the About Price Checker button in the Help section of the menu is clicked, the show-about event is sent to the main process. However, the main process has to listen to that event through the *ipcMain* module. The following command adds an event listener that loads the appropriate HTML file:

```
ipcMain.on('show-about', () => mainWindow.loadFile(__dirname + '/app/
menu/about.html'));
```

Once the event is triggered in the main process, the main window displays a new HTML file specified in the *ipcMain* function (Figure 20).

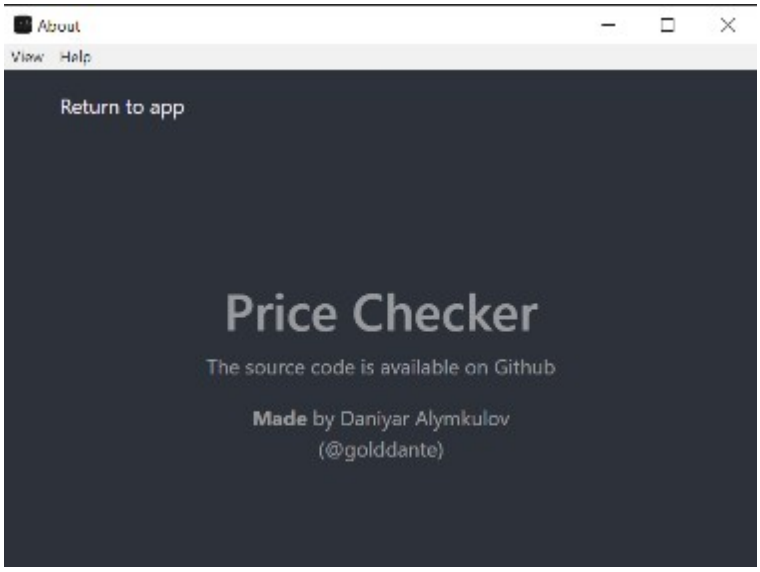

Figure 20. About Price Checker screen

Some of the text in the About Price Checker window are links which can be opened in a new browser window. That is done with the *shell* module that provides functions related to desktop integration by managing files and URLs using their default applications. The code below makes a link that opens a web page using the default browser of the platform:

```
1. <p>The source code is available on <a
2. onclick="shell.openExternal('https://github.com/golddante/
   electron-app-thesis')">GitHub</a>
3. <p>
```

The main window is ready at this point. Nevertheless, to fully showcase the capabilities of Electron, a second window can be created with a separate renderer process.

3.2.4 Second Window and IPC

Creating a second window is done in the main process just as main window was. It is used for adding new currencies to the main window. To do that, an additional entry was made to the application menu.

Figure 21 displays the newly created window. It contains nothing more but barebone HTML including an input field and a button.

Figure 21. Second window

Once the Submit button is pressed, a message is passed to a renderer process handling the main window. For a long time, Electron Framework was capable of relaying messages only between themain and renderer processes. That made the code longer and unnecessarily complicated. Recently, however, an additional function was introduced that made IPC among two renderer processes possible. The following function needs an ID of the receiver process since multiple renderers can exist simultaneously (GitHub Inc. 2019):

```
1. ipcRenderer.sendTo(1, 'get-add',
   document.querySelector('input').value);
2. ipcRenderer.send('close-add');
```

In this case, ID0 defaults to the main process and ID1 is assigned to the renderer process behind the main window.

3.2.5 Distribution

For distribution of the finished application, **Electron-builder** was used. According to Electron Userland (2019), it is a tool which is capable of packaging and building an application, among many other things, for a variety of formats, such as:

- Windows (exe, msi and appx)
- Linux (deb, rpm, freebsd, pacman, p5p, apk, etc.)
- OS X (dmg, pkg and mas)

Electron-builder doesn't work well with NPM due to several code signing issues. Therefore, **Yarn** is strongly recommended which is package manager similar to NPM. NPM can be used to install Yarn , surprisingly, with the following command:

```
npm install –save-dev yarn
```

Once yarn is working, Electron-builder can be installed with the command below:

```
yarn add electron-builder –dev
```

Both Electron-builder and Yarn are not needed for proper functioning of the application, and should therefore be placed in the development list of dependencies.

The configuration for Electron-builder is located in the package.json file. The code below is a simple config to make a distributable file for all major platforms. Unfortunately, it is not possible to make an OS X and Linux distributable under Windows and vice versa. In most cases, it is preferable to make a specific distributable on the target platform.

```
1. "build": {
2.     "appId": "com.electron.price-checker",
3.     "directories": {
4.       "output": "dist"
5.     },
6.     "mac": {
7.       "icon": null,
8.       "category": "public.app-category.finance",
9.       "target": "dmg"
10.     },
11.     "win": {
12.       "icon": null,
13.       "target": "nsis"
14.     },
15.     "linux": {
16.       "icon": null,
17.       "target": "deb"
```

```
18.     }
```

Electron-builder by default uses the application icon. Thus, there is no need to specify icon path in the config.

Then running the command below starts the make process:

```
yarn dist
```

After the make process is over, the resulting distributable files can be found inside the dist folder (Figure 22).

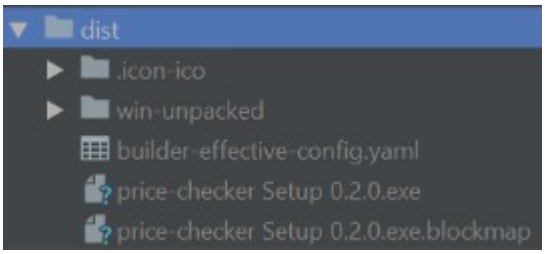

Figure 22. Distribution files

The .exe file is used to install the Price Checker application on Windows platforms. The installation process does not require any user input and starts automatically (Figure 23).

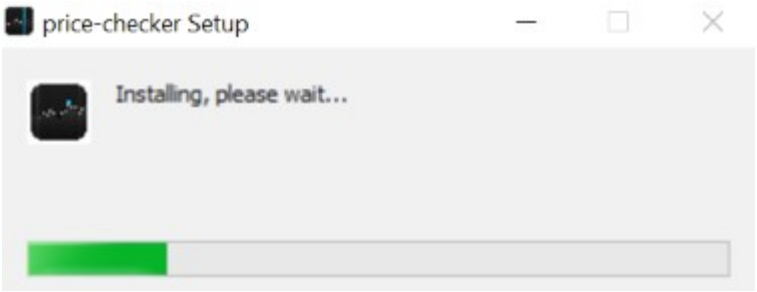

Figure 23. Installation process

Once the installation is done, the app can be found in the Windows application list (Figure 24).

Source code and installation binaries can be found on GitHub

3.3 Application Analysis

As stated in previous chapters, there are significant drawbacks in developing Electron applications. One of the ones is the size of the resulting apps. Since Electron has to pack both the Chromium engine and Node.js, even the most basic applications require a 100 MB of disk space minimum. In this case, the application takes 146 MB of disk space (Figure 24).

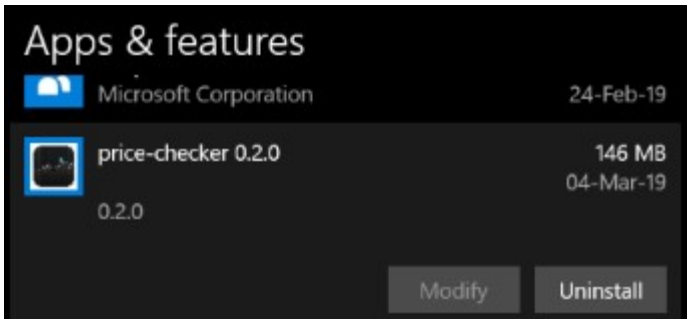

Figure 24. Electron app size

This code exists both on disk and in memory. Because Electron apps are separately packaged and executed, each application loads a separate instance of the libraries, even if the libraries are exactly the same version, the OS is unaware of that.

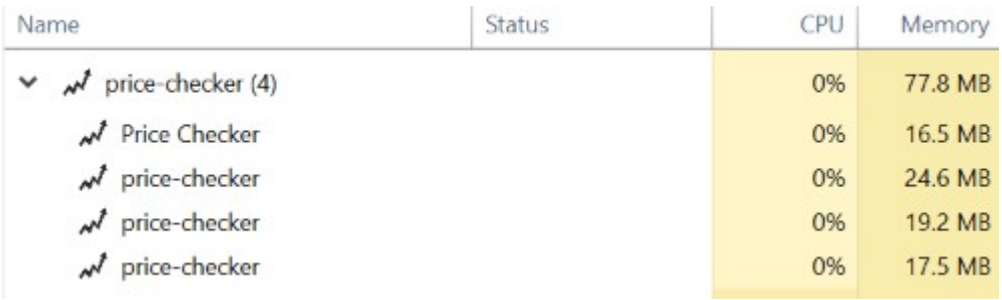

Figure 25. Electron memory consumption

Figure 25 demonstrates RAM consumption of this Electron application. This includes one main process, two renderer processes and a profiler. Both renderers together amount to about 45 MB. Other applications may have more processes and the memory usage can quickly spiral out of control. This overhead is not a big problem for large applications like VS Code or Atom. It does pose a larger issue for smaller apps that use Electron. Many developers are now using

Electron Framework to build desktop utilities and widgets. A typical desktop user may have a dozen of those open. If each of these has a copy of the whole Electron stack in memory, it wastes over a gigabyte of RAM and disk space for essentially nothing.

Another bottleneck of Electron is performance. It is hard to create cross-platform applications that would match native ones in terms of speed and stability. In the case of this thesis, the performance is decent enough to use comfortably by an average user. To evaluate the performance of this app, a profiler is used that comes with Chrome Development Tools. In fact, Google and Node.js teams collaborated to optimize Chrome DevTools to work seamlessly with Node applications (GitHub Inc. 2019).

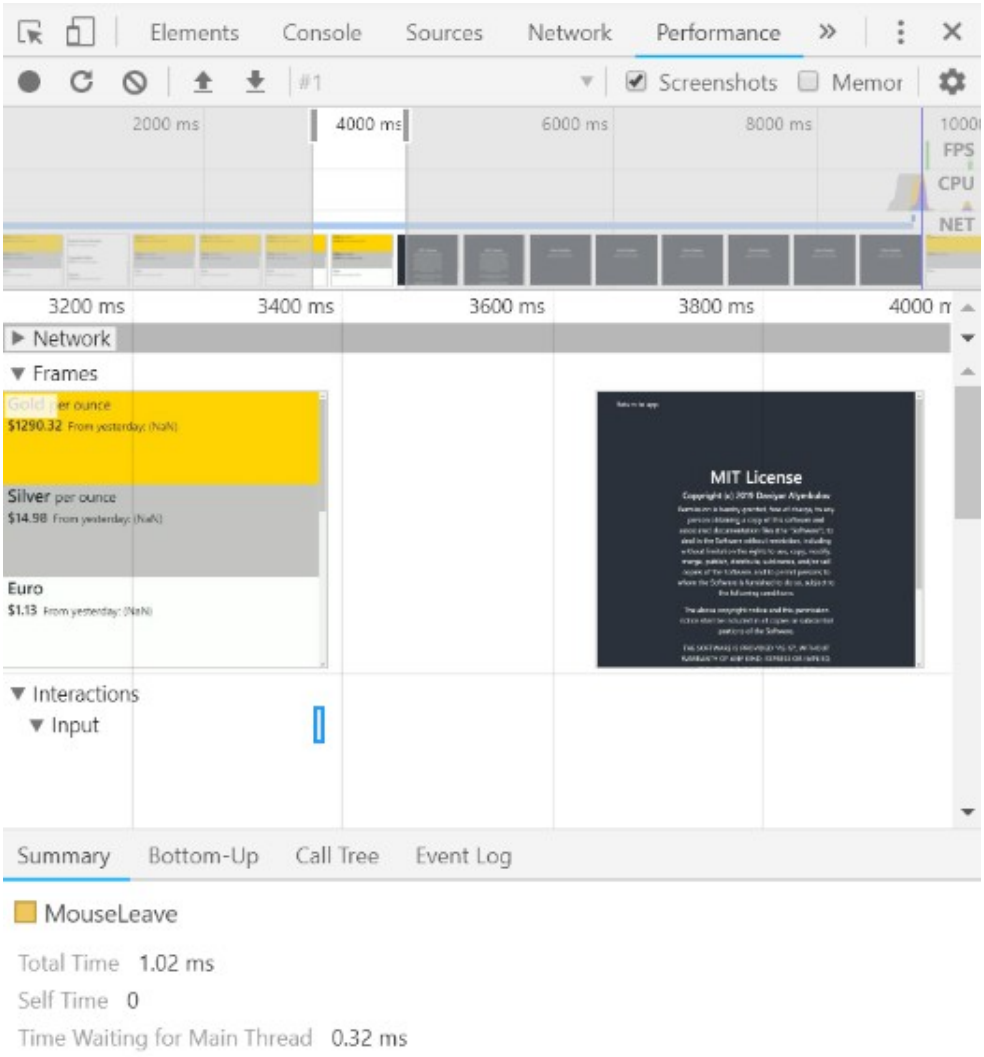

Figure 26. Chrome performance profiler

In Figure 26, the breakdown of the window change is illustrated. The blue bar on the bottom indicates the View License menu button press. It takes about 250 milliseconds to load and render the change of the main window. This result is big enough to notice, but not enough to interrupt the user's attention.

Another metric is FPS (frames per second). Figure 27 shows the FPS breakdown when the scrolling animation is triggered. Even though this metric is not completely reliable, given the limited number of elements to scroll through, the results indicate a consistent 60 frames per second, which is more than enough to an average user's eye.

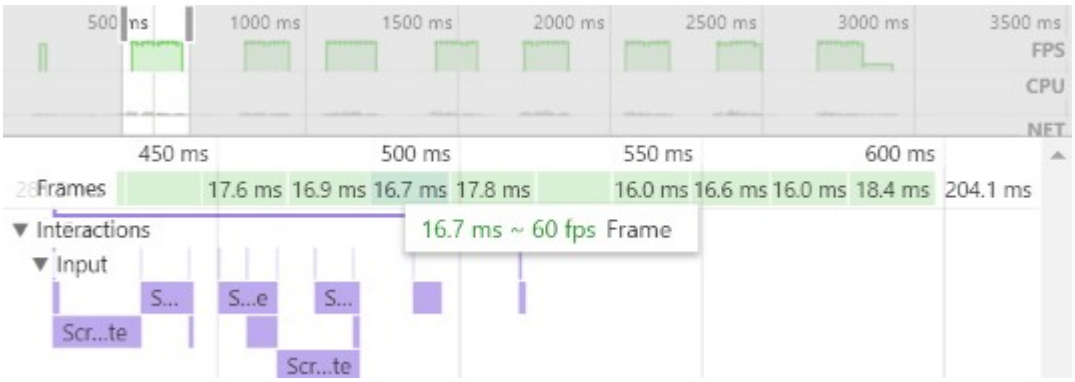

Figure 27. FPS counter

Being based on web technologies, Electron provides access to a huge number of community modules and extensions, not only for HTML or JS, but built specifically for Electron (GitHub Inc. 2019). On one hand, this enables rapid prototyping and additional functionality. On the other, the application may become too complex and bloated in size as a result of including too many external assets.

Lastly, platform-specific functionality is severely impaired and needs work, Windows in particular. APIs for macOS include in-app purchases, touch bar support and global dark theme. All of these are not available for Windows and Linux. One of the things that may be useful is Action Center integration for Windows 10 and quick actions for Linux. Other than platform-specific functions, Electron has enough components to make the application look native to the

operating system and not feel out of place. In this case study, only native menu API was used. In addition to that, many more are available, such as: desktop audio/video capturing, context menus, power monitoring, notifications, etc.

4 CONCLUSION

The purpose of this thesis was to showcase the Electron Framework and to establish a wide-ranging study between native .NET and cross-platform Electron frameworks. In the research part of the thesis, history, theory and architecture of .NET and Electron were discussed and compared. In order to showcase the Electron Framework, a cross-platform application was written using Electron. Due to the time and resource constraints, only an Electron app was written and no direct comparison between Electron and .NET application development was made. Nevertheless, the main object of demonstrating the Electron Framework was achieved.

When comparing Electron and .NET, it’s important to remember that Electron Framework is a young technology that is quickly gaining popularity in the web development community. In the increasingly mobile and cloud-oriented world, its basis on web technologies gives it a lot of support from big companies and developers. It also makes Electron the best choice for web and mobile developers who want to create desktop applications or maybe desktop versions of web applications.

.NET has much more time behind its shoulders and is specifically created by Microsoft to create highly optimized Windows apps and give greater control to the hands of the developers. Unlike Electron, which utilizes Chromium and Node.js, .NET is not tied to any external technology, and thus, can update and gain functionality on its own schedule. First-party documentation is also greater in .NET.

To conclude, both Electron and .NET have proven their value in the market and among the developers. They are not in competition necessarily, but rather shine

in different situations and complement each other. Developing applications for multiple platform applications is hard and requires ample time and resources, and thus, apps can survive a reasonable loss in performance.

REFERENCES

Agnew S., 2017. *5 Ways to Make HTTP Requests in Node.js.* WWW document. Available at: https://www.twilio.com/blog/2017/08/http-requests-in-node-js.html [Accessed 2 April 2019].

Bynens M. & Meurer B., 2018. *JavaScript engine fundamentals: Shapes and Inline Caches.* WWW document. Available at: https://mathiasbynens.be/notes/shapes-ics [Accessed 1 December 2018].

currencylayer, 2019. *Documentation for currencylayer API.* WWW document. Available at: https://currencylayer.com/documentation [Accessed 23 February 2019].

Ecma International, 2017. *C# Language Specification.* WWW document. Available at: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf [Accessed 2 February 2019].

Electron Userland, 2019. *electron-builder.* WWW document. Available at: https://www.electron.build/ [Accessed 11 January 2019].

Garbade M., 2018. *Native vs. cross-platform app development: pros and cons.* WWW document. Available at: https://codeburst.io/native-vs-cross-platform-app-development-pros-and-cons-49f397bb38ac [Accessed 3 April 2019].

GitHub Inc., 2019. *Application Debugging.* WWW document. Available at: https://electronjs.org/docs/tutorial/application-debugging#main-process [Accessed 9 April 2019].

GitHub Inc., 2019. *Electron Community.* WWW document. Available at: https://electronjs.org/community [Accessed 10 April 2019].

GitHub Inc., 2019. *Electron Documentation.* WWW document. Available at: https://electronjs.org/docs/api [Accessed 11 January 2019].

GitHub Inc., 2019. *ipcRenderer.* WWW document. Available at: https://electronjs.org/docs/api/ipc-renderer#ipcrenderersendtowebcontentsid-channel--arg1-arg2- [Accessed 5 April 2019].

Google, 2019. *Chromium.* WWW document. Available at: https://www.chromium.org/Home [Accessed 10 April 2019].

Harish M N., 2013. *Native vs Cross platform vs HTML5.* WWW document. Available at: https://www.slideshare.net/nagaharish_movva/native-vs-cross-platform-vs-html5 [Accessed 3 November 2018].

Hoffman C., 2018. *What's the Difference Between Chromium and Chrome?.* WWW document. Available at: https://www.howtogeek.com/202825/what%E2%80%99s-the-difference-between-chromium-and-chrome/ [Accessed 10 April 2019].

JavaTpoint, 2018. *C# Features.* WWW document. Available at: https://www.javatpoint.com/csharp-features [Accessed 19 February 2019].

Jayasuriya M., 2015. *Introduction to Web Design and Computer Principles.* WWW document. Available at: https://cs.nyu.edu/courses/fall15/CSCI-UA.0004-002/slides/slides_9_3.html#/ [Accessed 12 November 2018].

JetBrains, 2019. *WebStorm.* WWW document. Available at: https://www.jetbrains.com/webstorm/ [Accessed 13 February 2019].

Kavalaparambi S., 2017. *.NET Core Overview.* WWW document. Available at: https://devblogs.microsoft.com/premier-developer/net-core-overview/ [Accessed 1 February 2019].

Lord J., 2017. *Essential Electron.* WWW document. Available at: http://jlord.us/essential-electron/#what-is-electron [Accessed 1 December 2018].

Lynch A., 2017. *Beyond The Browser: From Web Apps To Desktop Apps.* WWW document. Available at: https://www.smashingmagazine.com/2017/03/beyond-browser-web-desktop-apps/ [Accessed 11 January 2019].

Maksimchik J., 2016. *Electron Awesome.* WWW document. Available at: https://slides.com/juliamaksimchik/electron-awesome#/ [Accessed 10 April 2019].

Mali A., 2016. *Web Page Designing.* WWW document. Available at: https://www.slideshare.net/AmitMali5/web-page-designing-58250788 [Accessed 23 November 2018].

Microsoft Corporation, 2015. *Introduction to the C# Language and the .NET Framework.* WWW document. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework [Accessed 23 February 2019].

Microsoft Corporation, 2017. *Overview of .NET Framework.* WWW document. Available at:

https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview [Accessed 12 February 2019].

Microsoft Corporation, 2019. *What is .NET?.* WWW document. Available at: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet [Accessed 1 February 2019].

Node.js Foundation, 2019. *About Node.js.* WWW document. Available at: https://nodejs.org/en/about/ [Accessed 3 April 2019].

Rickard C., 2016. *Choosing the right mobile app for your project: Native vs cross-platform vs hybrid.* WWW document. Available at: http://inoutput.io/articles/development/choosing-the-right-mobile-app-for-your-project-native-vs-cross-platform-vs-hybrid [Accessed 9 November 2018].

StatCounter, 2019. *StatCounter Global Stats.* WWW document. Available at: http://gs.statcounter.com/ [Accessed 1 February 2019].

Wikipedia, 2018. *Cascading Style Sheets.* WWW document. Available at: https://en.wikipedia.org/wiki/Cascading_Style_Sheets [Accessed 12 November 2018].

Wikipedia, 2019. *.NET Framework.* WWW document. Available at: https://en.wikipedia.org/wiki/.NET_Framework [Accessed 3 April 2019]