A Text Editor for Developers

William Wakeford

Supervisor: Dr Ian Mackie

Department of Informatics

BSc (Hons) Computer Science

Table of Contents

[Abstract 4](#_Toc165390365)

[A Text Editor for Developers 5](#_Toc165390366)

[1 Introduction 5](#_Toc165390367)

[1.1 What is a Text Editor? 5](#_Toc165390368)

[1.2 Types of Text Editors 5](#_Toc165390369)

[1.2.1 Basic text editor: 5](#_Toc165390370)

[1.2.2 Code editor: 5](#_Toc165390371)

[1.2.3 Word processors: 6](#_Toc165390372)

[1.2.4 Integrated Development Environments (IDEs): 7](#_Toc165390373)

[1.3 Objectives 7](#_Toc165390374)

[2 Motivation 8](#_Toc165390375)

[3 Background Research 8](#_Toc165390376)

[3.1 Target User Considerations 8](#_Toc165390377)

[3.2 Evaluation of Existing Solutions 9](#_Toc165390378)

[3.3 Conclusion 10](#_Toc165390379)

[4 Professional Considerations 10](#_Toc165390380)

[4.1 BCS Code of Conduct 10](#_Toc165390381)

[5 System Requirements 11](#_Toc165390382)

[5.1 Functional requirements 11](#_Toc165390383)

[5.2 Non-functional Requirements 12](#_Toc165390384)

[6 Choice of Technology 12](#_Toc165390385)

[6.1 Electron 12](#_Toc165390386)

[6.1.1 What is Electron? 12](#_Toc165390387)

[6.1.2 Chromium in Electron (Renderer) 13](#_Toc165390388)

[6.1.3 Node.js in Electron (Main) 13](#_Toc165390389)

[6.1.4 Inter-process Communication 13](#_Toc165390390)

[6.1.5 What is Electron Used For? 14](#_Toc165390391)

[6.1.6 Conclusion 14](#_Toc165390392)

[6.2 Comparison of Available Technologies 14](#_Toc165390393)

[6.2.1 Research 14](#_Toc165390394)

[6.2.2 Discussion 16](#_Toc165390395)

[6.2.3 Conclusion 17](#_Toc165390396)

[7 System Design 17](#_Toc165390397)

[7.1 File Structure of an Electron Application 17](#_Toc165390398)

[7.1.1 Overview of File Structure 17](#_Toc165390399)

[7.2 Files 18](#_Toc165390400)

[7.2.1 Main File (main.js) 18](#_Toc165390401)

[7.2.2 Renderer File (index.js) 19](#_Toc165390402)

[7.2.3 Preload File (preload.js) 20](#_Toc165390403)

[7.2.4 Node Modules Folder (node\_modules) 20](#_Toc165390404)

[7.2.5 HTML and CSS 20](#_Toc165390405)

[7.2.6 Package Files 21](#_Toc165390406)

[7.3 Modules 21](#_Toc165390407)

[7.3.1 Path 21](#_Toc165390408)

[7.3.2 File System 22](#_Toc165390409)

[7.3.4 Code Editor 22](#_Toc165390410)

[7.3.5 Terminal 23](#_Toc165390411)

[7.2 Version Control 24](#_Toc165390412)

[7.3 Screen Design 24](#_Toc165390413)

[References 25](#_Toc165390414)

Abstract

[The abstract should be one paragraph of between 150 and 250 words. It is not indented. Section titles, such as the word Abstract above, are not considered headings so they don’t use bold heading format. Instead, use the Section Title style. This style automatically starts your section on a new page, so you don’t have to add page breaks. To apply any text style in this document with just a tap, have a look at Styles on the Home tab of the ribbon.]

Keywords: [Tap here to add keywords.]

A Text Editor for Developers

# 1 Introduction

## 1.1 What is a Text Editor?

At the base level, a text editor is a software application which allows a user to create and modify text files. A text editor serves the purpose of a canvas for the user, whether that is for crafting code, writing a report, making a blog, or just jotting down notes. They are a fundamental tool in the digital world, used in vast number of settings.

## 1.2 Types of Text Editors

Text editors come in many different forms, each on catering for a specific need.

1.2.1 Basic text editor: A barebones application which only include essential features such as text entry, editing and saving. Examples of basic text editors are Notepad for Windows and TextEdit for macOS.

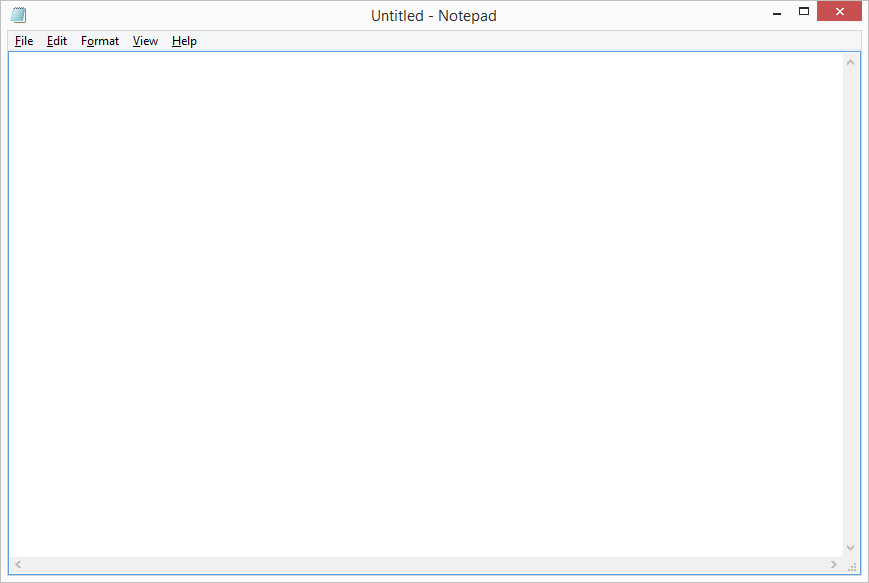


Figure 1: Screenshot of Notepad

Figure 1 shows a screenshot of Notepad, the default text editor for Windows. It has a basic layout and interface, allowing a user to open and modify a singular file. There are basic format options like changing the font and word wrapping.

1.2.2 Code editor: Designed for developers, these applications offer features such as syntax highlighting, code auto-completion, support for debugging, and embedded git. Some examples of code editors are Visual Studio Code (VSCode) and Sublime Text.



Figure 2: Screenshot of Sublime Text

Figure 2 shows a screenshot of Sublime Text, which includes the features described above. Sublime also allows a user to open folders, and switch between the current file with tabs.

1.2.3 Word processors: These are feature rich applications designed for creating, formatting, and editing documents with advanced styling options such as font changes, font colours, embedding images and adding pages.

A screenshot of a computer

Description automatically generated

Figure 3: Microsoft Word

Figure 3 shows a screenshot of Microsoft Word, which is a very popular word processor. Word includes a plethora of options for styling, formatting, and design. You can only edit files of type “docx” in Word, which means you cannot just open any text file, you have to create/open a file specifically designed for the software.

1.2.4 Integrated Development Environments (IDEs): An IDE is a comprehensive application catered for software development. They are similar to code editors; however, they are sometimes used for one single programming language, and include features such as debugging, project management and built-in compilers. Examples of IDEs are XCode and IntelliJ IDEA.

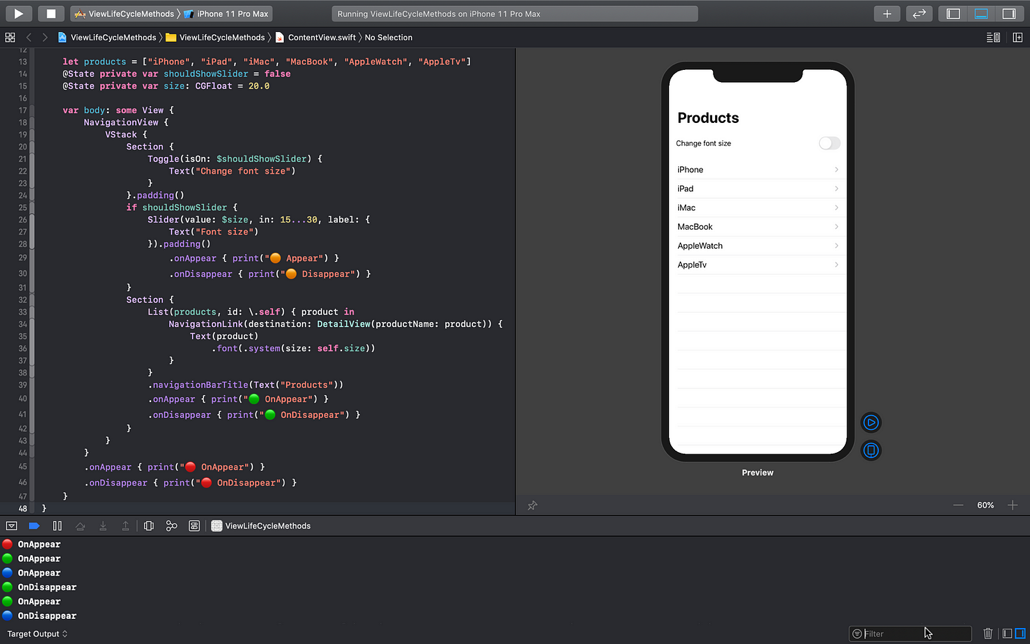


Figure 4: Screenshot of XCode

Figure 4 shows a screenshot of XCode, which is made for the programming language Swift. It includes extensive tools to help you specifically with the language.

## 1.3 Objectives

The primary aim of this project is to develop a code editor (see 1.2.2) which is tailored for developers. The envisioned outcome is to have a text editor which has a simple, user-friendly design, including tools and features that are helpful in writing code. The text editor should also be a great tool for basic text editing and will not be limited to developer use.

Specific Objectives:

1. Open, edit and save files and folders. For most text editors to function properly, they need to have functionality for opening, editing and saving files. I would also like to implement functionality for opening folders, allowing the user to edit multiple files inside a folder – increasing productivity.
2. User-friendly design. This text editor will have an interface with minimal complexity, ensuring a simple and easy editing experience for all skill levels.
3. Code editing features. A vital objective for this project is for it to include code editing features. This will include syntax highlighting, auto completion, line numbers, as well as the functionality to recognise the programming language of a file by its extension (i.e. “.py” for Python files).
4. Integrated terminal. The editor should have the option to open an integrated terminal, which will open at the folder/file location. This will allow a user to run the code they produce, as well as make system-side changes.
5. Cross-platform compatibility. The editor should be compatible across multiple operating systems, including Windows, MacOS and Linux. This ensures usability and accessibility.

By following these objectives, the project should deliver a code editor that meets the needs of developers, and also serves as a useful tool for all forms of text editing.

# 2 Motivation

My motivation for building a text editor mainly stems from the fact that they are an important part of my life as a computer science student and an aspiring software engineer. Having gone through numerous projects and tasks on many different text editors, it became apparent how important the software I am using is in order to increase the productivity and ease of work. In creating a text editor, it can help me understand how the software I use functions, and how I can better utilise them myself.

Due to the fact I use code editors often for my work, it helps me to understand what is desired by developers, and I can use that understanding to help me figure out what is most important in the project. Creating a text editor that aligns with what I believe will fulfil the needs and preferences of a developer is compelling and motivating to me.

# 3 Background Research

## 3.1 Target User Considerations

In order to develop this code editor, the user preferences and needs must be considered. By understanding the requirements of specific users, the editor can be tailored to provide the highest satisfaction possible.

The main target users will be developers of all levels. This means the editor should accommodate for a wide spectrum of developer types. Including tools like auto-completion and syntax highlighting will help entry level programmers understand how the language they are using works. On the other end of the spectrum, including a terminal will allow more experienced developers to take advantage of a tool that can enhance their productivity.

Another target user will be someone who doesn’t necessarily want to develop code, but just edit text. An easy to use interface will allow a user to take advantage of the editor for their basic needs, without feeling out of their depth.

## 3.2 Evaluation of Existing Solutions

I have researched the landscape of existing solutions, which all offer different features, functionality and user experience. This research is shown in Table 1.

Table 1: Existing Solutions Research

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Editor** | **Description** | **Positives** | **Negatives** |
| Visual Studio Code (VSCode) | Microsoft’s VSCode is a free, open-source code editor. It is rich with features and has support for a vast range of programming languages. It is a very popular choice among developers. | - Rich ecosystem. VSCode has a number of extensions and plugins  - User interface. The interface is clean and intuitive.  - Built-in Terminal  - Active community and support | - Learning curve. Using VSCode to its full potential may take a lot of time and energy, due to the large number of settings, extensions and configuration options available.  - Resource intensiveness. Working on large projects in VSCode can be very resource intensive. |
| Sublime Text | Sublime is a lightweight code editor known for its speed, efficiency and elegant design. | - Offers plugins and packages  - Speed and performance. Sublime offers fast response times even when handling large files/systems.  - Minimalist interface. Sublime has a clean interface, free from clutter and distractions. | - Limited features. Users may need to access third party plugins to access advanced functionality. |
| Atom | Atom is a free and open source code editor developed by GitHub. It is known for its extensive customization and community-driven development. | - Modern, intuitive user interface  - Collaborative editing environment. GitHub integration is catered for Atom so that it is easy for users to collaborate and use version control | - Resource intensive  - Performance issues. The editor can face performance issues when facing larger projects. |

## 3.3 Conclusion

Drawing conclusions from the research, I have gained valuable insight into the available solutions and the strengths/weaknesses they possess. I have found that it is important to have a number of important features, a clean and intuitive user interface, as well as good performance.

My proposed solution, stemming from the research of these solutions, is to create an editor that provides many features – such as syntax highlighting, auto complete and an integrated terminal – but also one that isn’t overwhelming and confusing for some users. In order to do this, I will need to find a middle ground, where the editor feels easy to use, has a tidy user interface and includes features that are desired by developers.

# 4 Professional Considerations

## 4.1 BCS Code of Conduct

The British Computer Society (BCS) code of conduct sets professional standards that govern the behaviour and responsibilities of individuals working in the field of computing. The code of conduct:

* “Sets out the professional standards required by BCS as a condition of membership.” [1]
* “Applies to all members, irrespective of their membership grade, the role they fulfil, or the jurisdiction where they are employed or discharge their contractual obligations.” [1]
* “Governs the conduct of the individual, not the nature of the business or ethics of any Relevant Authority.” [1]

# 5 System Requirements

## 5.1 Functional requirements

“Functional requirements define what a product must do and what its features and functions are.” [2]

1. Text editing features. For a text editor to function properly, one of the first things to consider will be the ability to edit text.
   1. Ability to create, edit and save text within files.
   2. Copy and paste text.
2. File operations. A user must be able to open a file, view and edit it in the application.
   1. Opening a file from the system.
   2. Saving a file to the system.
   3. Creating a new file in the system.
3. Folder operations. A user must be able to open a folder containing compatible files/folders.
   1. Open folders from the system.
   2. Use folder as a workspace.
4. Tab operations. A user must be able to switch between files using tabs.
   1. Open tabs from list of opened files.
   2. Close tabs.
   3. Switch between files using tabs.
5. Code editing features. To turn a text editor to a code editor, it must include basic code editing features.
   1. Syntax highlighting.
   2. Auto completion.
   3. Line numbers.
6. Terminal. The editor will need an integrated terminal to allow for developers to run their code through the application.

## 5.2 Non-functional Requirements

“Nonfunctional requirements describe the general properties of a system. They are also known as quality attributes.” [2]

1. Clean and easy to use UI.
   1. Clutter free.
   2. Neat file structure in explorer
2. Performance.
3. Compatibility. The editor should be compatible with different operating systems.
   1. Windows.
   2. MacOS.
   3. Linux.

# 6 Choice of Technology

To create a text editor that meets the needs and requirements of this one, technologies with many helpful tools and libraries will be required. The technology must support text manipulation and display, file and folder management, and more advanced features such as syntax highlighting and auto-completion. Therefore, the choice of technology is critical as it forms the foundation of which the text editor is built upon.

In the case of this project, I have carefully evaluated various technologies and frameworks before choosing Electron as the foundation technology.

## 6.1 Electron

### 6.1.1 What is Electron?

Electron is an open-source framework that allows development of desktop applications using web technologies such as JavaScript, HTML and CSS. I won’t be explaining in any detail how HTML, CSS and JavaScript work in this report; the HTML and CSS used is basic, if you want more understanding on these languages, feel free to read this blog post <https://blog.hubspot.com/marketing/web-design-html-css-javascript> [3]. It was originally created by GitHub for their code editor Atom, and has since become an extremely popular choice for developers looking to build cross-platform applications that can run on Windows, MacOS and Linux seamlessly. Electron is able to do this due to its ability to leverage the ecosystem of Node.js, along with the versatility of Chromium’s rendering capabilities. Electron also allows you to maintain the same JavaScript codebase across all platforms, which makes it far easier and drastically faster to develop a cross-platform application, which usually requires separate codebases for each platform.

### 6.1.2 Chromium in Electron (Renderer)

“Chromium is an open-source browser project that aims to build a safer, faster, and more stable way for all users to experience the web.” [4] Chromium is responsible for rendering the user interface and managing the web-related processes of an Electron app. As the project that underpins Google Chrome, it is known for its performance and compliance with web standards, making it a reliable foundation for web-based applications. This means that the HTML, CSS and JavaScript used to create a web page are displayed with the same speed expected from a leading web browser.

By utilising Chromium, Electron benefits from a heavily tested rendering engine, and allows developers to use languages and technologies that are familiar in the industry of web design.

### 6.1.3 Node.js in Electron (Main)

Node.js is another component that plays a vital role in Electron, as it gives applications access to low-level APIs. This means that Electron apps have the capability to read and write files to a system, listen to network requests and more. The renderer process (Chromium) is able to use Node.js modules as if it were a Node.js application, which makes it a powerful tool when needing to interact directly with the operating system.

One of the most impactful benefits of Electron using Node.js is that it grants the ability to use the npm ecosystem

#### 6.1.3.1 What is the npm Ecosystem?

Npm – [[docs.npmjs.com](http://docs.npmjs.com/)]

“npm is the world's largest software registry. Open source developers from every continent use npm to share and borrow packages, and many organizations use npm to manage private development as well.” [5] Electron can use the vast selection of npm packages to add features and functionality to applications without having to build everything from scratch.

### 6.1.4 Inter-process Communication

In Electron, the main process and the renderer process communicate with each other using inter-process communication (IPC). IPC allows the processes to send messages to each other. This means that, for example, the main process can access the OS file system and send data from that to the renderer. This is spoken about in more depth further into the report.

### 6.1.5 What is Electron Used For?

Electron is primarily used for building desktop applications that require a native look and feel, whilst also needing the flexibility of cross-platform support. It is useful for developers who already have experience in web development, as it allows them to use their existing skills to build a desktop application without having to learn new languages.

Some well-known applications built using Electron include VSCode, Slack, Discord and WhatsApp [5]. These applications showcase the capabilities of Electron and demonstrates that it can handle complex, high-performance requirements while providing a smooth user experience.

### 6.1.6 Conclusion

By using Electron as the technology to build the text editor, I can combine the ease and cross-compatibility of web development with the power of native application programming. Node.js will enable me to perform native tasks such as file operations and system commands, which is crucial for a text editor to work. Everything considered, Electron is the ideal choice for this project.

## 6.2 Comparison of Available Technologies

### 6.2.1 Research

Table 2 outlines some key considerations I have taken when comparing the available technologies. Understanding the benefits of all available resources is important in choosing the best fit for the application. By evaluating the options side by side, it can help determine which technology aligns best with the project’s goals.

Table 2: Comparison of available technologies

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature** | **Electron** | **Native Development** | **Java** | **Web Technologies** | **Cross-platform Frameworks (React Native, Flutter)** |
| **Development Language** | JavaScript, HTML, CSS | C++, Swift, .NET, etc. | Java | HTML, CSS, JavaScript | Dart (Flutter), JavaScript (React) |
| **Performance** | Good for most applications. Can struggle with large intensive tasks | High, performance is optimized for each platform | Good, can be optimized | Varies, dependant on browser technologies | High, close to native performance |
| **User Interface** | Native-like, customizable with web technologies | Fully native and can be fully optimized for OS | Not always native-like, can feel outdated | Fully web-based, limited by browser capabilities | Native-like, designed to feel integrated into the OS |
| **Cross-platform Support** | Yes, single codebase for Windows, MacOS and Linux | No, separate codebase needed for each platform | Yes, runs anywhere with JVM | Yes, runs in any modern browser | Yes, single codebase for Windows, MacOS and Linux |
| **System Access** | Full system access via Node.js | Full system access with native APIs | Full system access, but with JVM overhead | Limited by browser security restrictions | Full system access through bridge/compilation |
| **Community and Ecosystem** | Large, benefits from web development ecosystem | Large, well-established for each platform | Mature, less active for desktop applications | Extensive, encompasses entire web development sphere | Growing, strong communities and increasing support |
| **Typical Use Cases** | Desktop apps with moderate performance needs, rapid development cycle | High-performance applications, with deep OS integration | Enterprise applications, cross-platform tools | Web-based applications and platforms | Mostly mobile apps, potentially desktop apps |
| **Integration with Web Technologies** | Direct, uses web development technology for interface | Indirect, web views can be used but are not typical | Indirect, relies on Java for user interface | Direct | Indirect, primarily uses native components, but can integrate web views |

### 6.2.2 Discussion

From the information gathered, Electron seemed the most suitable choice for developing a text editor. Some of the standout factors for this was the fact that

1. Electron uses web development languages that I am already familiar with, speeding up the learning process.
2. Electrons vast range of libraries through Node.js means that a lot of features and functionality is made available to me.

The toughest decision was choosing between Electron and other cross-platform frameworks. This is especially the case with Flutter and React Native have similar features in areas, for example:

* Cross-platform capability. All three frameworks have a single codebase that can run on multiple different operating systems.
* Strong community and support. Electron, React Native and Flutter all have a strong community and support. This can speed up development tasks as helpful information is more accessible.
* Development efficiency: Each framework provides features that enhance development speed. For example, Electron allows the use of web technologies directly; React Native bridges web development with app features through JavaScript; and Flutter offers a hot reload capability that speeds up development cycles.

#### 6.2.2.1 Disadvantages of Electron

Electron is a powerful framework; however, it does come with some disadvantages. These include:

* Resource consumption: Electron apps can be resource-intensive, primarily because each app runs a full instance of the Chromium browser.
* App size: Both Chromium and Node.js being bundled into an Electron app causes the size of the app o be quite large.
* Security concerns: Electron can be susceptible to security vulnerabilities if not managed properly.

### 6.2.3 Conclusion

Choosing the right technology comes down to what the project needs, and which one suits those needs best. A cross-platform framework was the standout choice for the text editor, as they offer large ecosystems, use only one codebase and have full access to the operating system. Electron stood out to me as the framework of choice because of the fact it was built for desktop applications from the ground up, whereas other popular frameworks have been built for mobile app development initially. This means Electron is tailored towards desktop app development, making it a robust and mature environment. Another reason for Electron is the number of apps that have been made using it that are extremely successful, including VSCode, which proves it is a good choice for desktop apps.

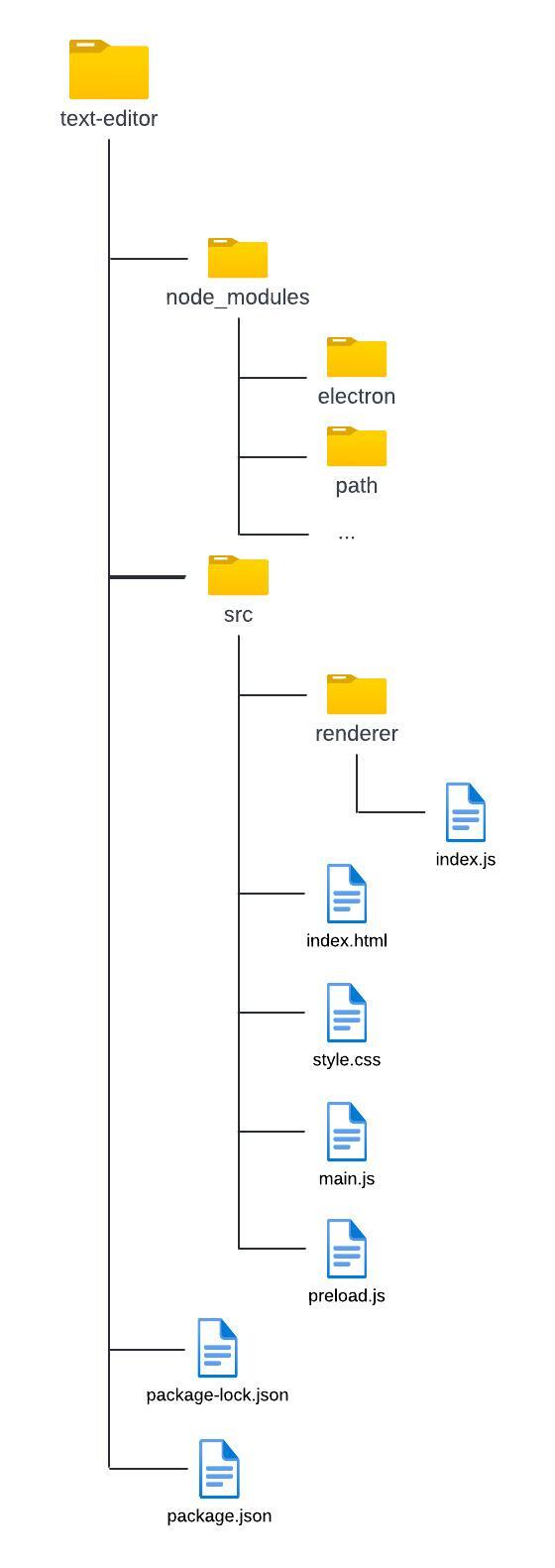
# 7 System Design

## 7.1 File Structure of an Electron Application

Electron uses a specific file architecture that is important to use in order to make the most of both the web technologies and the native system features. The structure creates a bridge, allowing communication between the main file and renderer file.

### 7.1.1 Overview of File Structure

An Electron project uses a fairly simple file structure and can be slightly altered depending on the project. Here is a diagram of how this project will be setup as an Electron app:



## 7.2 Files

### 7.2.1 Main File (main.js)

The main.js file is the entry point of an Electron app, where it initiates the main process. The main process controls the lifecycle of the app, creates windows and has full system access.

#### 7.2.1.1 Window Management

Every Electron app runs its application windows through ‘BrowserWindow’ instances, which are created in this file. Each instance of the BrowserWindow class creates an application window that loads a web page in a separate renderer process. The window’s ‘webContents’ object allows you to interact with the web content from the main process. This code displays how a BrowserWindow class is created, and the contents of it are retrieved.

const { BrowserWindow } = require('electron')

const win = new BrowserWindow({ width: 800, height: 1500 })

win.loadFile(‘renderer/index.js’)

const contents = win.webContents

console.log(contents)

When an instance of BrowserWindow is closed, the renderer process attached to it is also destroyed.

#### 7.2.1.2 Application Lifecycle

The main process manages the lifecycle of the application using the ‘app’ module. This allows you to modify the behaviour of the app, for example activating or closing BrowserWindow instances. This code shows how you can use the app module to control how an application is closed when on a MacOS system:

app.on("window-all-closed", () => {

if (!isMac) {

app.quit();

}

});

#### 7.2.1.3 Native APIs

The main process allows custom APIs to interact with the user’s operating system, which allows the Electron app to have access to things like an operating system’s menus and dialogs. This is crucial for opening, creating and saving files, a must for text editors.

### 7.2.2 Renderer File (index.js)

Each Electron window runs its own renderer process, which is responsible for running the web page. This process is the same as rendering a process on a web browser, so the user interfaces and functionality are all written in the same way they would be on the web.

An HTML file (index.html in this case) is used as the entry point to the renderer process. Styling is done through a CSS file. JavaScript code can be added in the HTML file through the <script> tag.

### 7.2.3 Preload File (preload.js)

“Preload scripts contain code that executes in a renderer process before its web content begins loading. These scripts run within the renderer context, but are granted more privileges by having access to Node.js APIs.” [7]

A preload script is attached to the main process through the windows webPreferences:

const win = new BrowserWindow({

webPreferences: {

preload: ‘path/to/preload.js’

}

})

Preload scripts are useful for increasing the security of the application, limiting direct renderer access to critical functions.

### 7.2.4 Node Modules Folder (node\_modules)

The ‘node\_modules’ folder is where all modules and libraries are stored for any Node.js project. The modules are typically installed to the node\_modules folder through npm, the node package manager. Having the folder means that all the project dependencies can be stored locally, rather than retrieving them from the internet every time.

The folder can become very large, depending on how many packages a project requires, however Node.js is designed to handle this well. Runtime performance of applications isn’t affected by the folder, however the speed of the initial build can be.

### 7.2.5 HTML and CSS

Both the HTML and CSS files will be simple due to the fact I will only use one window for the editor, with a minimal interface. The focus of this project is on the functionality over design, so more time will be spent on the functionality to ensure they are at the correct standard.

Having smaller HTML and CSS files can also increase the performance of the text editor, helping keep it lightweight and responsive. Essentially, there is not much point to adding unnecessary bulk to these files when it is not needed.

### 7.2.6 Package Files

The package.json and package-lock.json files are vital to an Electron app – they manage the project dependencies and allow for you to have a consistent environment across different setups.

#### 7.2.6.1 package,json

The package.json file contains:

* A list of dependencies
* Project metadata, such as the name, version and description
* Scripts for tasks such as building and testing
* Licensing information

#### 7.2.6.2 package-lock.json

The package-lock.json file is automatically generated when something is changed in the node\_modules folder. It holds information about the dependencies, for example the versions and the dependency trees.

## 7.3 Modules

Node modules will be essential for this project, as they handle specific functionality that would be extremely difficult to implement otherwise. This section will outline and discuss the modules that will be used in the project.

### 7.3.1 Path

“The path module in Node.js is a built-in module that provides utilities for working with file and directory paths. It helps in constructing, manipulating, and working with file and directory paths in a cross-platform manner, making it easier to write platform-independent code.” [8]

A text editor will be dealing with file and folder paths very regularly, in order to retrieve data from files and save them to the system. Therefore, the path module is essential to this project. An important method in this module is the parse() method, which turns a path into a JSON object. This allows the path to be split into different parts, for example the base, the directory and the extension.

### 7.3.2 File System

The fs (file system) module allows an application to interact with the files on a system. This module is crucial for the text editor as it enables the application to read the data inside files and write data to files.

Beyond this, the fs module also provides the capability to rename, delete, copy and move files on a system.

### 7.3.4 Code Editor

A code editor module is a library that provides text editing capabilities specifically designed for coding. They are built to handle the syntax of programming languages – offering features such as syntax highlighting, code completion, error detection, line numbers and more. This module will be the driving force for the editing area of the text editor, which makes it an important choice of which one to use.

#### 7.3.4.1 Available Code Editors

1. Monaco Editor: This code editor was developed by Microsoft, and is the editor used for VSCode. It is well known for its rich feature set, including IntelliSense (smart code completion), syntax highlighting and support for many languages.
2. Ace Editor: Ace is a web-based code editor, which supports over 120 languages and is highly customizable.
3. CodeMirror: This is a powerful web-based code editor used by companies like Adobe and Mozilla. It offers features like code folding, and support for over 100 languages.

For this project, I decided to go with Monaco Editor as the code editor. Some reasons for this include:

1. Monaco editor is known for its high performance, with the ability to handle very large files.
2. It is used on Visual Studio Code, which is one of the most powerful code editors at the moment.
3. The editor can be used in many different development contexts, making it a versatile choice.

One downside to using Monaco editor is that there aren’t a lot of resources regarding it apart from the official documentation. This can make it difficult to find out how to use it.

### 7.3.5 Terminal

There are two modules that will be used to create the terminal for the text editor: Xterm and Node-pty.

#### 7.3.5.1 Xterm

Xterm.js is a Node module that emulates a terminal in a browser window. It provides functionality of a terminal and integrates it into an HTML file. Here is a list of features it brings:

* Terminal emulation. Xterm emulates the behaviour of a desktop terminal, in our case enabling the renderer to run a terminal shell in the window.
* Customization: It supports custom themes, allowing the design of the terminal to match the text editor.
* Input and output handling: Inputs and outputs are handled/displayed just like a systems terminal would.
* Xterm can use web sockets to exchange data in real time.

#### 7.3.5.2 Node-pty

Node-pty is a Node module that enables interaction with terminals on a system. This provides a way for the app to interact with the systems terminal shell. This module works by spawning processes which control input and output to the system terminal. These processes are run inside a real terminal instance, which ensures that any terminal-specific behaviour is replicated.

#### 7.3.5.3 Building a Terminal

In order to integrate a terminal into the text editor, both modules play a vital role. In the renderer process (see 6.1.2), xterm will be used to create a terminal interface which is attached to an HTML element. This will render the terminal, display outputs and capture inputs.

The xterm terminal communicates with the node-pty module in the main process (see 6.1.3). The node-pty module then uses the process it has spawned to communicate with the system terminal, and so on. By combining these two modules, it will allow me to add a fully functional terminal into the text editor.

## 7.2 Version Control

Version control is an important component when developing software. For this project, I have used GitHub to record all changes to the code I have written. Not only does this help understand the evolution of the project, but also stores a backup of the code.

## 7.3 Screen Design

This text editor will only consist of one window, which means minima screen design is required. As stated in the requirements (See 5.2), the user interface have a minimalistic and clean design. The following screenshot is a basic screen design made using HTML and CSS, to display where sections should be placed. Note the colours are there just to highlight sections and are not used in the final project.

A screen shot of a computer

Description automatically generated

Figure 5: Screen Design

### 7.3.1 Explorer

### 7.3.2 Tabs

### 7.3.3 Text Area

References

[1] BCS (2022). BCS, THE CHARTERED INSTITUTE FOR IT CODE OF CONDUCT FOR BCS MEMBERS. [online] Available at: https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf.

[2] Altexsoft (2023). Functional and Non-functional Requirements: Specification an. [online] AltexSoft. Available at: <https://www.altexsoft.com/blog/functional-and-non-functional-requirements-specification-and-types/>.

[3] Kolowich, L. (2018). Web Design 101: How HTML, CSS, and JavaScript Work. [online] Hubspot.com. Available at: https://blog.hubspot.com/marketing/web-design-html-css-javascript.

[4] www.chromium.org. (n.d.). Home. [online] Available at: <https://www.chromium.org/chromium-projects/>.

[5] docs.npmjs.com. (n.d.). About npm | npm Docs. [online] Available at: <https://docs.npmjs.com/about-npm>.

[6] electronjs.org. (n.d.). App Showcase | Electron. [online] Available at: https://www.electronjs.org/apps.

[7] electronjs.org. (n.d.). Process Model | Electron. [online] Available at: https://www.electronjs.org/docs/latest/tutorial/process-model.

[8] DEV Community. (2023). Node.js path module. [online] Available at: https://dev.to/endeavourmonk/nodejs-path-module-16fm#:~:text=The%20path%20module%20in%20Node [Accessed 30 Apr. 2024].

‌

‌

‌