A Text Editor for Developers

William Wakeford

Supervisor: Dr Ian Mackie

Department of Informatics

BSc (Hons) Computer Science

Statement of Originality

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signed: William Wakeford

Date: 02/05/2024

Acknowledgements

I would like to thank Prof Ian Mackie for supervising this project, and for his help in finding creative ideas to improve it.

Table of Contents

Statement of Originality 2
Acknowledgements 3
Abstract 8
A Text Editor for Developers 9
1 Introduction 9
1.1 What is a Text Editor? 9
1.2 Types of Text Editors 9
1.2.1 Basic text editor: 9
1.2.2 Code editor: 9
1.2.3 Word processors: 10
1.2.4 Integrated Development Environments (IDEs): 11
1.3 Objectives 11
2 Motivation 12
3 Background Research 12
3.1 Target User Considerations 12
3.2 Evaluation of Existing Solutions 13
3.3 Conclusion 14
4 Professional Considerations 14
4.1 BCS Code of Conduct 14
5 System Requirements 15
5.1 Functional requirements 15
5.2 Non-functional Requirements 16
6 Choice of Technology 16
6.1 Electron 16
6.1.1 What is Electron? 16
6.1.2 Chromium in Electron (Renderer) 17

6.1.3 Node.js in Electron (Main) 17
6.1.4 Inter-process Communication 17
6.1.5 What is Electron Used For? 18
6.1.6 Conclusion 18
6.2 Comparison of Available Technologies 18
6.2.1 Research 18
6.2.2 Discussion 20
6.2.3 Conclusion 21
7 System Design 21
7.1 File Structure of an Electron Application 21
7.1.1 Overview of File Structure 21
7.2 Files 22
7.2.1 Main File (main.js) 22
7.2.2 Renderer File (index.js) 23
7.2.3 Preload File (preload.js) 24
7.2.4 Node Modules Folder (node_modules) 24
7.2.5 HTML and CSS 24
7.2.6 Package Files 25
7.3 Inter-Process Communication IPC 25
7.3.1 What is IPC? 25
7.3.2 How Does IPC Work? 26
7.3.3 Preload Script 26
7.4 Modules 27
7.4.1 Path 27
7.4.2 File System 27
7.4.4 Code Editor 27
7.4.5 Terminal 28

7.5 Version Control 29
7.6 Screen Design 30
7.6.1 Explorer 30
7.6.2 Tabs 31
7.6.3 Text Area 31
7.6.4 Terminal 31
7.6.5 Menu 32
8 System Implementation 32
8.1 Programming Technique 32
8.2 Inter Process Communication 33
8.3 Window Creation 33
8.4 File Handling 34
8.4.1 Main Process 34
8.4.2 Renderer Process 36
8.4.3 HTML and CSS 37
8.5 Folder Management 37
8.5.1 Main Process 37
8.5.3 Renderer Process 40
8.5.4 HTML and CSS 42
8.6 Tab Management 44
8.6.1 Active File Path 46
8.7 Editor 46
8.7.1 RequireJS 47
8.7.2 Using Monaco Editor 47
8.8 Terminal 48
8.8.1 Main Process 48
8.8.2 Renderer Process 49

8.8.3 HTML and CSS 51
8.9 Menu 52
9 System Evaluation 55
9.1 Testing 55
9.2 Future Work 56
9.2.1 Collaborative Coding Features 56
9.2.2 Auto Check File Contents 57
10 Conclusion 57
References 58

Abstract

This project presents the design and implementation of a text editor designed for the use of developers, utilizing the Electron framework. The primary objective of this project was to create a text editor that provides features such as syntax highlighting, auto-completion, and an integrated terminal to enhance the productivity and experience for developers. Through extensive research, the editor was developed to offer a blend of being feature-rich as well as having a clean, easy to use user interface.

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.

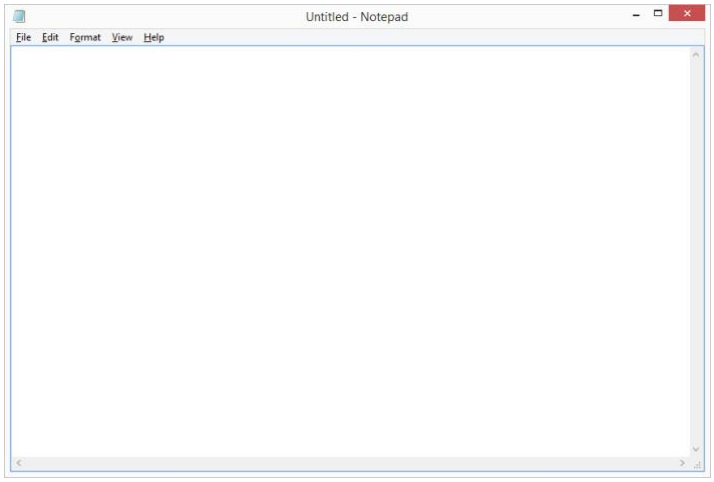


Figure 1: Screenshot of Notepad

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.

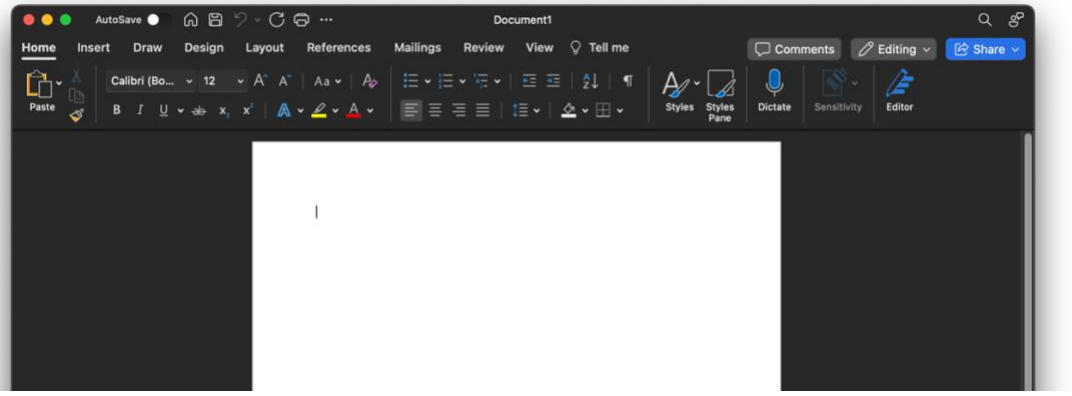


Figure 3: Microsoft Word

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.

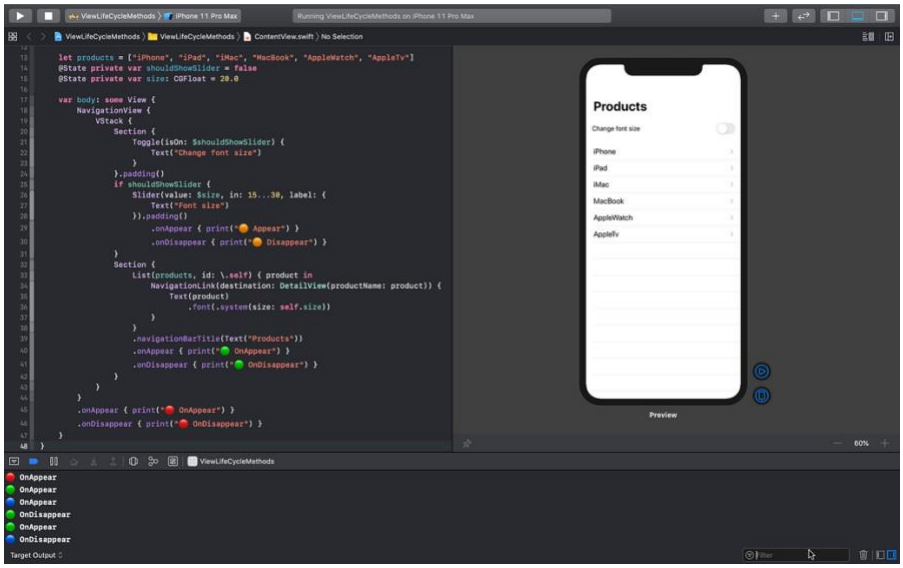


Figure 4: Screenshot of XCode

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	- Limited features. Users may need to access third party plugins to access advanced functionality.

		interface, free from clutter and distractions.	
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.
    a. Ability to create, edit and save text within files.
    b. Copy and paste text.
2. File operations. A user must be able to open a file, view and edit it in the application.
    a. Opening a file from the system.
    b. Saving a file to the system.
    c. Creating a new file in the system.
3. Folder operations. A user must be able to open a folder containing compatible files/folders.
    a. Open folders from the system.
    b. Use folder as a workspace.
4. Tab operations. A user must be able to switch between files using tabs.
    a. Open tabs from list of opened files.
    b. Close tabs.
    c. 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.
    a. Syntax highlighting.
    b. Auto completion.
    c. 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.
    a. Clutter free.
    b. Neat file structure in explorer
2. Performance.
3. Compatibility. The editor should be compatible with different operating systems.
    a. Windows.
    b. MacOS.
    c. 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]

"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	Large, well-established for each platform	Mature, less active for desktop applications	Extensive, encompasses entire web	Growing, strong communities and increasing support

	development ecosystem			development sphere	
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

a) Electron uses web development languages that I am already familiar with, speeding up the learning process.
b) 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.

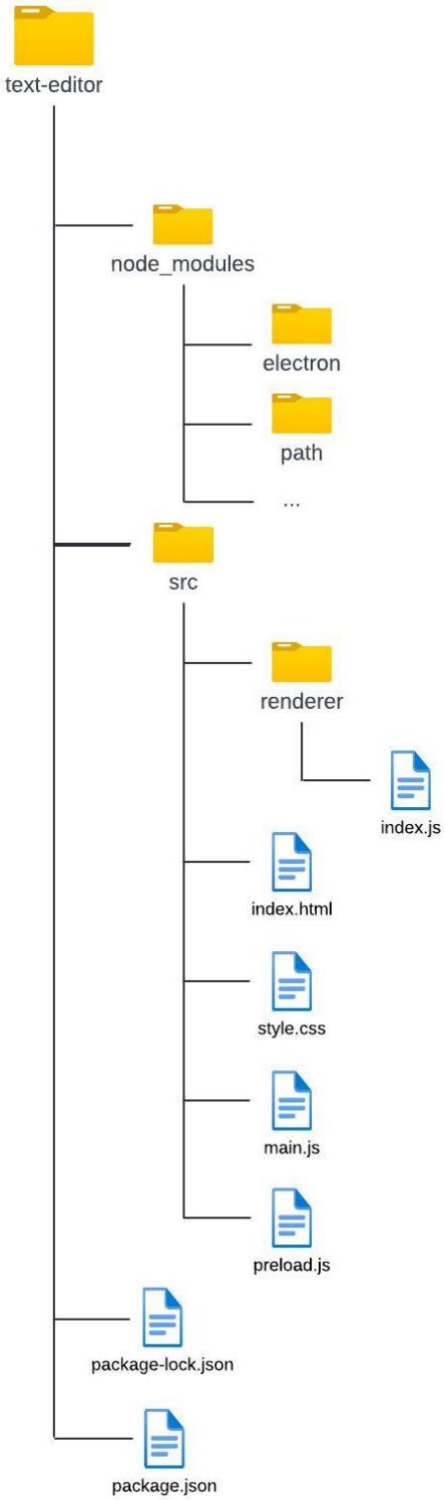
7 System Design

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 Inter-Process Communication IPC

7.3.1 What is IPC?

"Inter-process communication (IPC) is a key part of building feature-rich desktop applications in Electron. Because the main and renderer processes have different responsibilities in Electron's process model, IPC is the only way to perform many common tasks, such as calling a native API from your UI or triggering changes in your web contents from native menus." [8]

IPC is a vital part of building Electron application. It refers to a mechanism that allow different processes (main and renderer) to communicate with each other. The main process can access the native system, whereas the renderer process cannot. Therefore, the processes need to be able to communicate with one another in order to share data such as file contents.

7.3.2 How Does IPC Work?

IPC works by using a different “channel” in each process – ipcMain for the main process and ipcRenderer for the renderer process. With both these channels, you can send and receive data to/from the other.

To send a message from the renderer you would use:

```
ipcRenderer.send('send-data', data)
```

To receive a message from the main you would use:

```
ipcRenderer.on('receive-data', (data) => {
      console.log(data)
}
```

To send and receive messages from the main, you would use the same code, but replace ipcRenderer with ipcMain. The first parameter for both functions is the title of the event. These titles will be set in the preload.

7.3.3 Preload Script

The preload file will be where all the IPC functions are created and exposed. This bridges the gap between the main and renderer processes.

7.3.3.1 Security

Using the preload script for the IPC enhances the security by controlling how the renderer (Chromium environment) can interact with the main process (Node.js environment). This allows specific actions and functions to be performed securely, without exposing the whole Node.js API to the Chromium environment, which could be exploited.

The ipcRenderer is created in the preload script, and specific functions are created with it. These functions, for example, could be “send-file” or “open-terminal”. Only these specific functions will be able to occur between the IPCs, which stops any one with malicious intent from accessing the backend through functions they have made. An API called ContextBridge then exposes these functions to all processes, allowing communication between them.

7.4 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.

7.4.1 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." [9]

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.

7.4.2 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.

7.4.4 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.

7.4.4.1 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.

7.4.4.2 Monaco Editor

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.

7.4.5 Terminal

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

7.4.5.1 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.

7.4.5.2 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.

7.4.5.3 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.

7.5 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.

7.6 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.

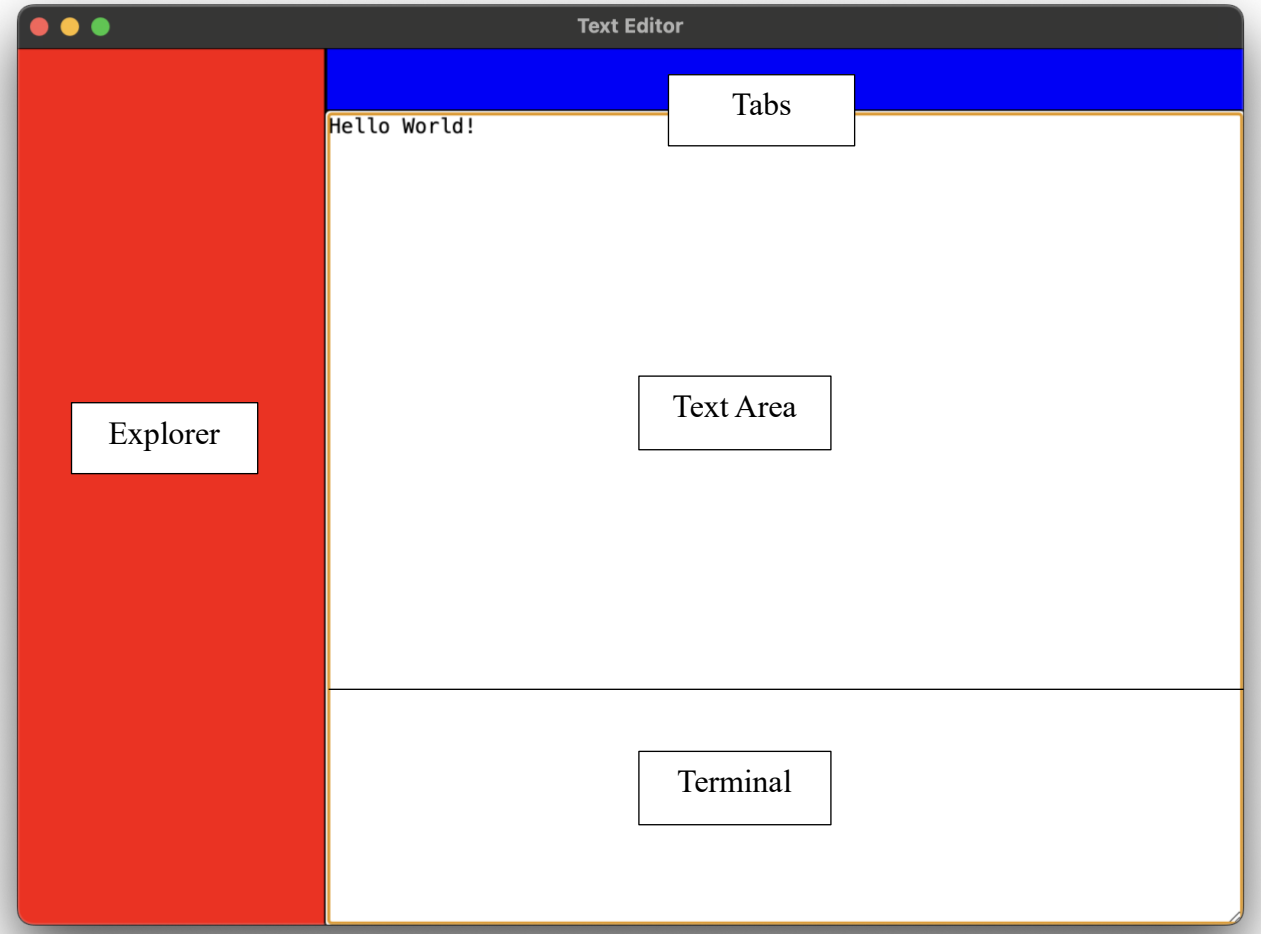


Figure 5: Initial Screen Design

7.6.1 Explorer

The explorer section will be where all the files and directories are displayed. These are the features to be included in the explorer area:

- A list of files/folders that have been opened through the systems menu.
- Upon clicking a folder, the files inside the folder we expand, and click the folder again will hide them.

- When clicking on a file, a tab for that file is created, and the document's data is opened in the text area section.
- When clicking on a file that is already open, the corresponding tab is opened.
- A scroll feature for when the number of files overflow the screen.

The explorer will have a title "Explorer", and will display the folder structure of the opened project to a user. Any buttons required will be placed at the top left-hand corner of the explorer.

7.6.2 Tabs

The tabs section will display different tabs, each corresponding to an opened file. Features of the tab area will include:

- A horizontal list of tabs which have been opened from the explorer.
- Each tab will be clickable in order to open the file.
- Each tab will have a close button, to close the file.
- The active tab will be highlighted.

The tab section will allow a user to keep track of what files they have currently open, as well as an indicator of which file they currently have opened.

7.6.3 Text Area

The text area will be where a user writes their text/code. The active tab will display the corresponding file's data in this section. This is where the Monaco Editor (See 7.3.4.2) will be displayed.

Each tab will store a Monaco Editor model, which will store the value inside the text area. This means that when a user switches tab without saving the file, the text they edited will still be stored – allowing them to go back to the previous tab.

7.6.4 Terminal

The terminal area will only be displayed when a user opens the terminal through the menu. When the terminal is closed, the text area will fill the space. A user will be able to open and close the terminal as they please, having it close allows more space for the text area.

7.6.5 Menu

The menu is the toolbar provided by the operating system. It is placed on the open window for Windows systems, and at the top of the screen on MacOS systems. The menu must have the following options:

- Open File
- Open Folder
- Save
- Save As
- Open Terminal
- Close Terminal

The menu will be the only place you can access these functions. Shortcuts should also allow a user to access these functions with ease, for example Ctrl/Cmd + S for Save.

8 System Implementation

8.1 Programming Technique

Programming rules including naming conventions and consistency have been set for this project. Each language have their own variable naming style to recognise which variables come from where. This is useful as a lot of variables are shared between files. The naming rules are as follows:

- JavaScript variables: camelCase for variables.
- HTML: lowercase. Multi-word names concatenate without any characters in between, for example: “mainheading”.
- IPC method names: lowercase. A hyphen separates multi-word names, for example: “save-file”.

There are multiple ways to create functions in JavaScript. The method used in this project is the “arrow function” method. An example of how this is formatted:

```
const function = (parameters) => {
      console.log(parameters);
}
```

The syntax rules for this are:

- Parameters should be passed in a small bracket.
- If there are no parameters, there must be an empty bracket.

8.2 Inter Process Communication

The preload script determines the channels that data can be sent and received by the renderers IPC. In this case, channels are just names for the type of data being sent/received. The channels include:

- Send:
    - “new-file”: request for main to create new file
    - “save-file”: send file data for a save function in the main
    - “save-data-as”: send file data for a save as function in the main
    - “terminal-data”: send terminal inputs to the main
- Receive:
    - “file”: file data received from main
    - “folder”: folder data received from main
    - “get-save”: a request from main to send file data for save
    - “get-save-as”: a request from main to send file data for save as
    - “open-terminal”: request from main to open the terminal
    - “close-terminal”: request from main to close the terminal
    - “terminal-output”: the data received from the terminal, after a user has entered an input

The requests from main are due to the fact that most functions are actioned from the menu, which is in the main process.

8.3 Window Creation

Upon window creation in the renderer, the HTML elements are retrieved, so they can be used and manipulated. In any code where you see “el.(element)”, this is a HTML element.

This code shows all the elements that are retrieved from the HTML file for use:

```
window.onload = () => {
  el = {
    newDocumentBtn: document.getElementById("newfile"),
    folderList: document.getElementById("folderlist"),
    explorer: document.getElementById("exploreritems"),
    tabList: document.getElementById("tablist"),
    editorArea: document.getElementById("editorarea"),
    editor: document.getElementById("editor"),
    terminal: document.getElementById("terminal"),
    terminalArea: document.getElementById("terminalarea"),
  };
```

Figure 6: Elements retreived from HTML file at window load

8.4 File Handling

8.4.1 Main Process

Files on the system side are handled by the main process – main.js. Files are sent and received as a JSON object, which contains:

- path: The path of the file. This is also a JSON object which includes:
    - root: root name
    - dir: directory name
    - base: filename with extension
    - ext: extension only
    - name: filename only
    - fullpath: directory + base
- data: The contents of the file.

Using this format for communicating files between the main and renderer process means that all the information required for each file is available in both processes. Here is an example of the code used to display this, located in the openFile function:

```
fs.readFile(filePath, "utf8", (err, data) => {
  if (err) throw err;

  let filePathObj = path.parse(filePath);
  filePathObj["fullpath"] = filePathObj.dir + "/" + filePathObj.base;

  if (isFolder) {
    createWindow();
    win.webContents.on("did-finish-load", () => {
      win.webContents.send("file", {
        path: filePathObj,
        data,
      });
    });
  } else {
    win.webContents.send("file", {
      path: filePathObj,
      data,
    });
  }
  openItems.push({ path: filePathObj, data });
});
```

Code Snippet 1: Part of the openFile function, displaying the format of a file being sent between processes

8.4.1.1 Opening files

When a user wants to open a file, Electron's dialog module allows interaction with the operating system's file explorer. The user will select the file they wish to open through the dialog. The file path of the chosen file is then retrieved by the dialog, and used by the fs module (see 7.4.2) to read the file data. The data and file path are put into the JSON object as shown above and sent to the renderer to be displayed.

8.4.1.2 Saving files

When a user saves a file, a request is sent to the renderer to get the save data. Once the renderer has responded with the data (the file path and content), it is saved to the users system using the fs module. The file is then sent back to the renderer, so it can store the newly saved version of the file in the explorer.

8.4.1.3 Save As

Users can use a Save As option to save a file as if it were new. This follows the same process as saving files but also creates a file on the system.

8.4.2 Renderer Process

The renderer is always listening for a file that has been sent from the main.

```
window.ipc.receive("file", (data) => {
  handleOpenFile(data);
});
```

Code Snippet 2: Renderer IPC receive file

This piece of code shows this; the IPC module is listening for a file , and once it has received one, the handleOpenFile function is called. The file is then handled, which involves:

- The file being added to a list of files stored locally in the renderer
- The file is sent to the addFileToList function, where:
    - A file item is created, which is an HTML button element that includes the file link and the base of the file path as text context.
    - An event listener is added to the button, which opens the file in a tab.
    - The file item is appended to it's parent HTML element. The parent element of a file is initially the explorer, however if the file is inside a folder, the parent element will be the folder it's in.
- If the file is already open, just the content of that file will be replaced.

Here is the handleOpenFile function:

```
const handleOpenFile = (file, parent = el.explorer) => {
  if (!fileInList(file.path)) {
    fileDataList.push(file);
    addFileToList(file, parent);
    initialContentMap[file.path.fullpath] = file.data;
  } else {
    replaceFileData(file);
  }
};
```

Code Snippet 3: handleOpenFile function

The initialContentMap holds the content of the file when it is first opened. This allows us to track any changes made to the file.

Once this function is complete, the file will be added to the explorer section of the text editor :

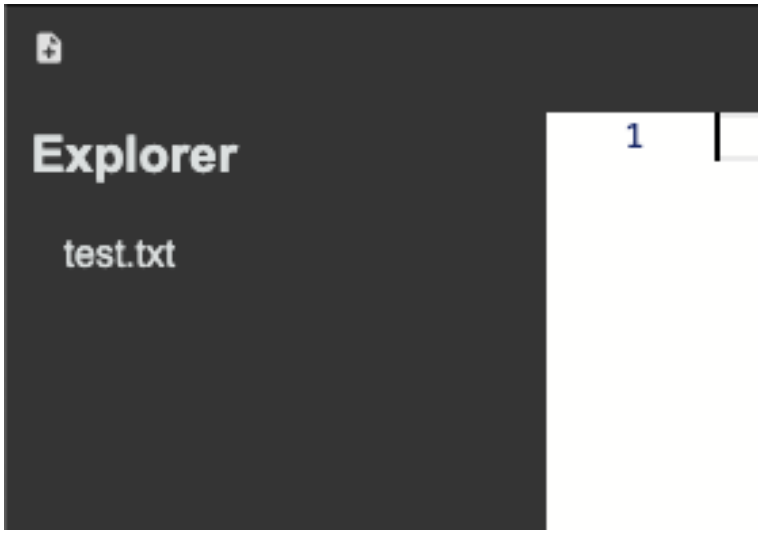


Figure 7: Explorer with file added

8.4.3 HTML and CSS

The HTML elements are created in JavaScript, then appended to the explorer element. Each file item:

- Includes a link which adds a tab when clicked.
- Includes the file path base as the text context
- If the file is inside a folder, it is hidden initially. This is so that a user can expand a folder to show the files inside.

CSS ensures that when the button is hovered over, it changes colour to highlight what the user is hovering over:

```
#exploreritems li button:hover  {
    background-color: var(--hover-color);
}
```

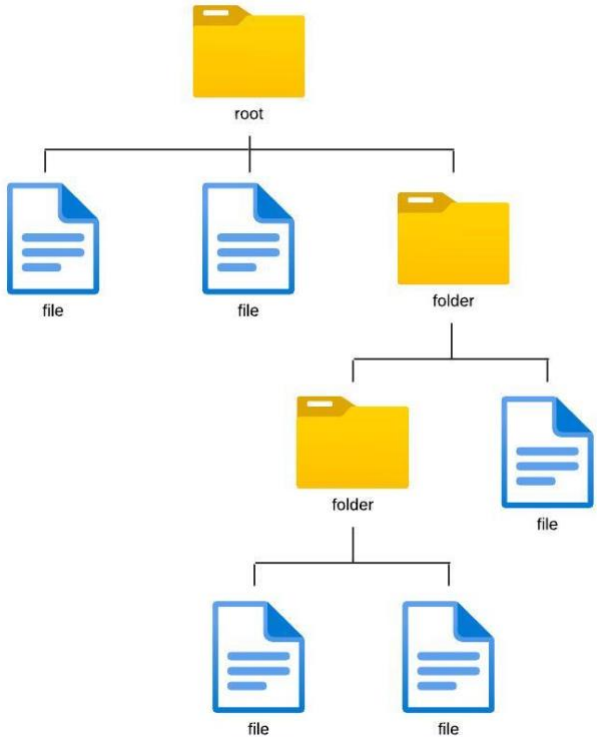
Code Snippet 4: CSS of file item hover

8.5 Folder Management

8.5.1 Main Process

The folder management is slightly different to the file management, as it requires retrieving multiple files and sometimes multiple folders with more files inside. The JSON object for folders is slightly different because of this.

- path is the root folder path – the folder that the user selects from the dialog.
- data is a tree of all files and folders within the root folder. It can be represented like this:



In order to create this object using this structure, I needed to create a recursive function that retrieves all the folder contents in the correct order. Here is the function used:

```
const getFolderContents = (folderPath) => {
  let contents = [];

  const files = fs.readdirSync(folderPath);

  // for each file in directory
  files.forEach((file) => {
    // exclude hidden files
    if (file.startsWith(".")) {
      return;
    }
    const fullPath = path.join(folderPath, file);
    const stats = fs.statSync(fullPath);
    const filePathObj = path.parse(fullPath);
    filePathObj["fullpath"] = filePathObj.dir + "/" + filePathObj.base;

    // if the file is a directory
    if (stats.isDirectory()) {
      // use recursion for contents inside subfolder
      const subFolderContents = getFolderContents(fullPath);

      // push folder to contents
      contents.push({
        type: "folder",
        path: filePathObj,
        data: subFolderContents,
      });
    } else {
      // get file contents
      const data = fs.readFileSync(fullPath, "utf-8");

      // push file to contents
      contents.push({ type: "file", path: filePathObj, data });
    }
  });

  return contents;
};
```

Code Snippet 5: getFolderContents function

Code Snippet 3:

1. The root directory is passed into the function as a parameter
2. “contents” is initialised as a list

3. Each item inside the directory is read and put into a list called “files”
4. Looping through each item in “files”:
    a. Using fs.statSync, we can determine whether the item is a file or a directory
    b. If the item is a directory, we use recursion to start this process from step 1 with the new root directory.
    c. If the item is not a directory, we push the file into the contents list
5. Return contents

This means that each directory has its own contents list, and that list can include files and more directories with their own contents list, and so on. Retrieving the folder contents this way allows all the information needed to be stored in one single object. It also makes it very readable, as you can visually see how the folders, subfolders and files are nested in the Chromium’s DevTools.

8.5.3 Renderer Process

Once a folder is received in the renderer process, another recursive function is needed to “unwrap” the contents. Each folder must be treated as a folder and each file must be treated as a file. This is critical for the application to be able to display the correct item in the correct position on the page.

```
const addFolder = (folder, parent = el.explorer) => {
  let folderPath = folder.path;
  let contents = folder.data;
  console.log(contents);

  let folderItem = document.createElement("li");
  folderItem.setAttribute("id", "folder");
  if (parent !== el.explorer) {
    folderItem.style.display = "none";
  }

  // set up folder link
  let folderLink = document.createElement("button");
  // add arrow icon
  let icon = document.createElement("i");
  icon.classList.add("bx", "bxs-chevron-right");
  let buttonText = document.createTextNode(folderPath.base);

  folderLink.appendChild(icon);
  folderLink.appendChild(buttonText);

  // add link to folder item
  folderItem.appendChild(folderLink);
  // add folder item to list
  parent.appendChild(folderItem);

  contents.forEach((item) => {
    // if item is a file, add as file
    if (item.type == "file") {
      handleOpenFile(item, folderItem);
    }

    // if item is a folder, use recursion to add items
    if (item.type == "folder") {
      addFolder(item, folderItem);
    }
  });
};
```

Code Snippet 6: addFolder function

The addFolder function displays how this is done.

1. The folder is passed as a parameter into the addFolder function, as well as the parent of the folder (default being the explorer)
2. The folder is set up as an HTML element
3. Looping through each item in the contents of the folder:
    a. If the item is a file, it is handled using the handleOpenFile method (Code Snippet)
    b. If the item is a folder, the function recursively calls itself, going back to step 1. This time, the parent parameter is the folder that was initially sent in.

Once this function is complete, each folder and file will be opened into the text editor's explorer section:

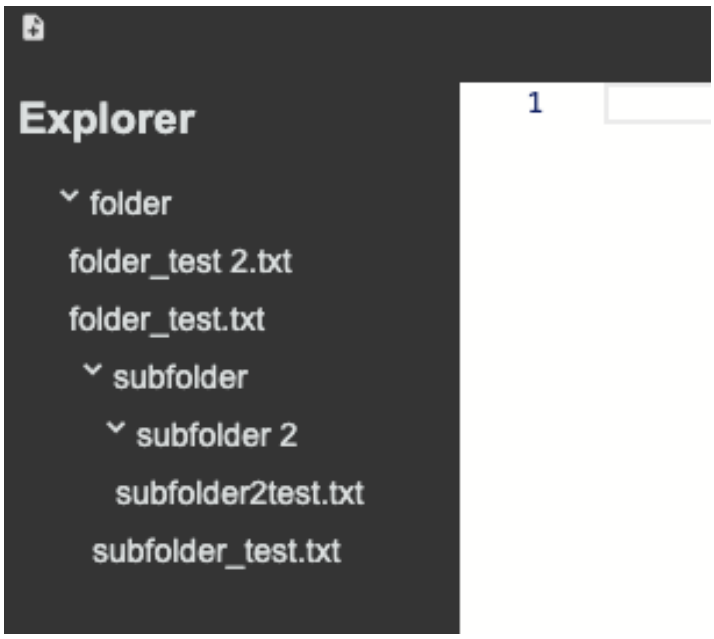


Figure 8: Explorer with folder added

8.5.4 HTML and CSS

Code Snippet 6 shows how folder elements are created for the HTML. A folder element:

- Includes a link to expand and close the folder's contents.
- Includes a chevron, which will be a visual indicator for whether the folder is expanded or not.
- Uses the folder name as the text context.

The CSS used for a folder item is as follows. It allows for the chevron to rotate when the folder expands/hides its content:

```css
#exploreritems #folder i {
    aspect-ratio: 1/1;
    padding-right: 5px;
    transition: transform 0.3s ease;
}

#exploreritems #folder .expanded i {
    transform: rotate(90deg);
}
```

Code Snippet 7: CSS for folder items

8.4.4.1 Issue with Event Bubbling

The folder event listeners had to be created in a separate function after they had been set up. This was because event bubbling caused folders to include the event listeners twice. "Event Bubbling is a concept in the DOM (Document Object Model). It happens when an element receives an event, and that event bubbles up (or you can say is transmitted or propagated) to its parent and ancestor elements in the DOM tree until it gets to the root element." [10]

There is a way to stop this from happening using the stopPropogation method of the event object, however as every folder needed the exact same event listener anyway, I decided to add it to just one element and use the bubbling to my advantage. I did this in the addFolderEventListeners function:

```
// adds event listener - hide files - to all folders
const addFolderEventListeners = () => {
  const folderElements = document.querySelectorAll("[id='folder']");
  console.log(folderElements);

  let root = folderElements[0];
  root.addEventListener("click", ({ target }) => {
    console.log(target);
    let sibling = target.nextElementSibling;

    while (sibling) {
      sibling.style.display = sibling.style.display == "block" ? "none" : "block";
      sibling = sibling.nextElementSibling;
    }

    target.classList.toggle("expanded");
  });
};
```

Code Snippet 8: addFolderEventListeners function

The “sibling” variable is each item inside the target element – each item inside the folder. The event listener toggles the items CSS display between “none” and “block”. This means a user can expand the folder contents when it is clicked, and hide the contents when it is clicked again.

8.6 Tab Management

The management of tabs is done purely in the renderer. It involves creating a tab element when a file item is clicked. A tab element:

- Includes a link that opens the file contents when clicked
- Includes a close button that closes the tab when clicked (to stop event bubbling (see 8.4.4.1), I used the stopPropogation method on the event. This is because we don’t want the event listener to bubble up to the parent item)

When a tab element is created, a Monaco Editor model is created for that tab. This will be covered later in the report. The addTab function shows how a tab is set up:

```
const addTab = (file) => {
  const filePath = file.path;

  if (isTabOpen(filePath)) {
    displayFile(filePath);
    return;
  }

  const tabItem = document.createElement("li");
  const tabLink = document.createElement("button");
  const closeButton = document.createElement("button");

  // set up tab link
  tabLink.textContent = filePath.base;
  tabLink.addEventListener("click", () => {
    displayFile(filePath);
  });
  tabLink.setAttribute("class", "tabbutton");
  tabLink.dataset.fullpath = filePath.fullpath;

  // set up close button
  closeButton.textContent = "X";
  closeButton.addEventListener("click", (e) => {
    e.stopPropagation();
    handleCloseTab(tabItem, filePath);
  });
  closeButton.setAttribute("class", "closebutton");

  // append tab and close button to tab item
  tabItem.appendChild(tabLink);
  tabItem.appendChild(closeButton);

  // append tab item to tab list
  el.tabList.appendChild(tabItem);

  // add tab to openTabs list
  openTabs.push(filePath);

  createModelForFile(file);
  // display file
  displayFile(filePath);
};
```

Code Snippet 9: addTab function

8.6.1 Active File Path

The filePathActive variable holds the file path of the currently opened tab. This is an extremely important variable, as it lets the system know what file is being manipulated. If a user wants to save a file, the filePathActive variable will be used to determine which file is to be saved. This means the variable must always hold the correct file path.

In order for a user to see what tab they are currently working on, the tab item needs to be highlighted. To do this, I created a function that changes the class of the open tab to "highlighted":

```
const highlightTab = (filePath) => {
  let tabs = document.querySelectorAll("#tablist li button");

  // remove current highlighted tab
  tabs.forEach((tab) => {
    tab.classList.remove("highlighted");
  });

  tabs.forEach((tab) => {
    if (tab.dataset.fullpath === filePath.fullpath) {
      tab.classList.add("highlighted");
    }
  });
};
```

Code Snippet 10: highlightTab function

The CSS for a highlighted tab is as follows:

```
#tablist .highlighted {
    background-color: var(--hover-color);
}
```

Code Snippet 11: CSS for highlighted tab

8.7 Editor

For the editor area, the Monaco Editor (See 7.4.4) is used. This provides the functionality for code editing features, such as syntax highlighting, auto-complete, line numbers and more.

8.7.1 RequireJS

In order to include Monaco in the editor, I had to use a module called RequireJS. This allowed me to require the Monaco editor directly into the renderer, the only place it is needed. It packages the Monaco module into a JavaScript file recognised by the renderer, so it can be included as if the code is built into the renderer file.

8.7.2 Using Monaco Editor

The Monaco Editor is created when the renderer window is loaded, and it initialised with no value and an undefined language – making it an empty editor.

```
require(["vs/editor/editor.main"], () => {
  editor = monaco.editor.create(el.editor, {
    value: "",
    language: undefined,
  });
});
```

Code Snippet 12: Editor creation

When a tab is created, a new editor model is made, which automatically finds the language of the file and adds the file contents as the value.

```
const createModelForFile = (file) => {
  // retrieve file extension
  const filePath = file.path;
  const extension = filePath.fullpath.split(".").pop();
  const language = getLanguageId(extension);
  monaco.editor.createModel(file.data, language, monaco.Uri.parse(file.path.fullpath));
};
```

Code Snippet 13: createModelForFile function

The getLanguageId function is a function I created to find the language of the file and return it as an id that Monaco can use.

```
const getLanguageId = (extension) => {
  const languages = monaco.languages.getLanguages();
  console.log(languages);
  let languageId = null;

  languages.forEach((language) => {
    if (language.extensions && language.extensions.includes("." + extension)) {
      languageId = language.id;
    }
  });

  return languageId;
};
```

Code Snippet 14: getLanguageId function

When the tab is closed, the model for that tab is disposed. If a user had made changes to the file without saving, they are asked if they are sure they would like to close the tab without saving.

8.8 Terminal

Adding the terminal functionality to the text editor involved different steps for the main and renderer process. In the main process, the node-pty module connects to the systems terminal. In the renderer process, the xterm module displays the output of from the main and sends user inputs to the main.

8.8.1 Main Process

To initialise the terminal, the operating system that the user was using had to be determined in order to find the correct shell.

```
var shell = os.platform() === "win32" ? "powershell.exe" : "zsh";
```

Code Snippet 15: Shell variable

For Windows - PowerShell is used, for Unix-systems, zsh is used.

The terminal process had to be spawned with node-pty, which connects the module to the systems terminal.

```
const newTerminal = () => {
  win.webContents.send("open-terminal");

  ptyProcess = pty.spawn(shell, [], {
    name: "xterm-color",
    cols: 80,
    rows: 10,
    cwd: process.env.HOME,
    env: process.env,
  });

  ptyProcess.on("data", (data) => {
    win.webContents.send("terminal-output", data);
  });
};
```

Code Snippet 16: newTerminal function

Once the process is spawned, it sends all output data to the renderer using the IPC. All inputs from the renderer are received here:

```
ipcMain.on("terminal-data", (_event, data) => {
  ptyProcess.write(data);
});
```

Code Snippet 17: Receive terminal data from renderer

This means the main and renderer process can “talk” to each other through the terminal channels on the IPC.

8.8.2 Renderer Process

As xterm is only used in the renderer process, I have used RequireJs to include it (See 8.6.1). When a user opens a terminal, an xterm Terminal object is created:

```
const openTerminal = () => {
  el.terminalArea.style.display = "block";

  require(["xterm"], (xterm) => {
    term = new xterm.Terminal({
      cols: 80,
      rows: 10,
    });
    term.options = {
      theme: {
        background: "#3B3B3B",
      },
      fontSize: 12,
    };

    term.open(el.terminal);
    term.onData((data) => {
      window.ipc.send("terminal-data", data);
    });
  });
  window.dispatchEvent(new Event("resize"));
};
```

Code Snippet 18: openTerminal function

This creates a section for the output and input to be displayed in the text editor. It is attached to the terminal HTML element. The terminal sends input data to the main process through the “terminal-data” channel in the IPC. The renderer waits for terminal output data from the main and writes it to the terminal, so it is displayed to the user:

```
window.ipc.receive("terminal-output", (data) => {
  term.write(data);
});
```

Code Snippet 19: Terminal output received from main

8.8.3 HTML and CSS

The terminal element initially has a display of “none”, so that there is no empty element cluttering the user interface. When a user opens a terminal, the display is set to “block”, to display the terminal.

Without terminal:

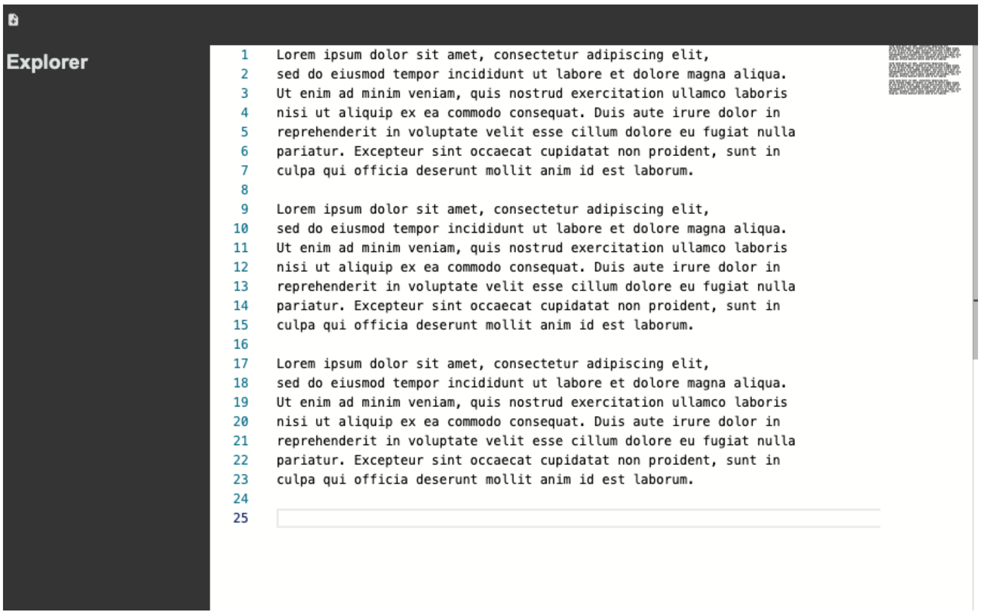


Figure 9: Interface without terminal

With terminal:

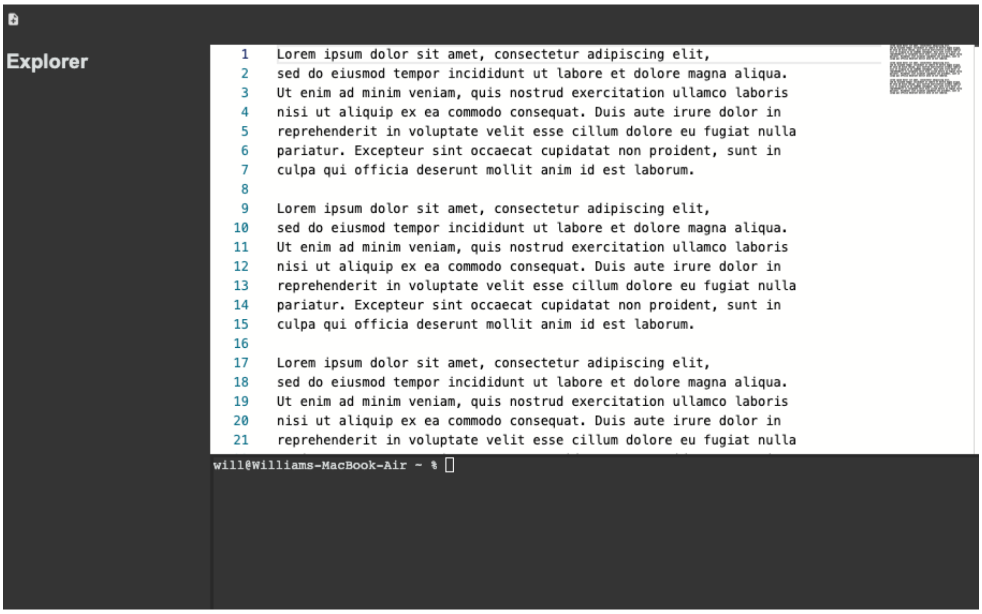


Figure 10: Interface with terminal

8.9 Menu

A user will have these custom options from the menu:

- File
  - Open File
  - Open Folder
  - Save
  - Save As
- Terminal
  - New Terminal
  - Close Terminal

These menu buttons perform their corresponding functions spoken about previously in this report.

The menu is created in the main process. A menu template is created, which is then set in the application in the app.whenReady() function. This is the menu template:

```
const menu = [
  ...(isMac
    ? [
        {
          label: app.name,
          submenu: [
            {
              label: "About",
            },
          ],
        },
      ]
    : []),
  {
    label: "File",
    submenu: [
      {
        label: "Open File...",
        click: () => {
          openFile();
        },
      },
      {
```

```
      label: "Open Folder...",
      accelerator: isMac ? "Cmd+O" : "Ctrl+O",
      click: () => {
        openFolder();
      },
    },
    {
      type: "separator",
    },
    {
      label: "Save",
      accelerator: isMac ? "Cmd+S" : "Ctrl+S",
      click: () => {
        saveFile();
      },
    },
    {
      label: "Save-As",
      accelerator: isMac ? "Cmd+Shift+S" : "Ctrl+Shift+S",
      click: () => {
        saveFileAs();
      },
    },
  ],
},
{
  role: "editMenu",
},
{
  role: "viewMenu",
},
{
  label: "Terminal",
  submenu: [
    {
      label: "New Terminal",
      click: () => {
        if (ptyProcess) {
          closeTerminal();
        }
```

```
        newTerminal();
      },
    },
    {
      label: "Close Terminal",
      click: () =>{
        closeTerminal();
      },
    },
  ],
},
{
  role: "windowMenu",
},
];
```

Each menu item includes:

- Label: the label the user sees.
- A “click” function: This tells the menu what to do when the option is clicked
- Accelerator: This defines the shortcut for the menu option.

The roles are predefined menus, usually include the default menu options for the label.

And here it is being added to the application:

```
app.whenReady().then(() => {
  createWindow();

  // implement menu
  const mainMenu = Menu.buildFromTemplate(menu);
  Menu.setApplicationMenu(mainMenu);

  app.on("activate", () => {
    if (BrowserWindow.getAllWindows().length === 0) {
      createWindow();
    }
  });
});
```

Code Snippet 20: Setting the menu

This is how the menu shows up on a MacOS system:

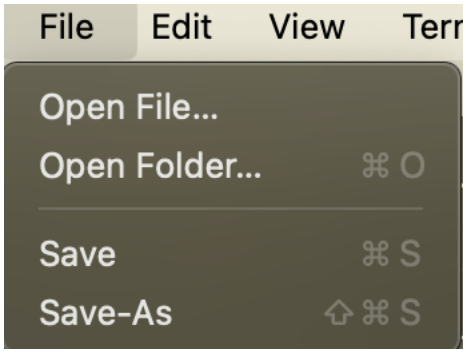


Figure 11: File Menu

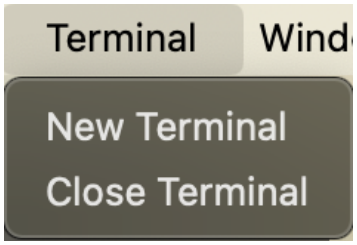


Figure 12: Terminal Menu

9 System Evaluation

9.1 Testing

I tested the functionality of the text editor myself through multiple test cases. This tested opening files, saving files, using the terminal and how well the editor worked.

Table 3: Testing

Test Case No.	Description	Expected result	Pass/fail
1	Open a text file	Contents of text file are displayed after clicking on the file item in explorer	Pass
2	Open a folder	All items inside folder are added to	Pass

		explorer, and each item is fully functional	
3	Save file	File is saved to specified location with the new contents	Pass
4	Terminal usage	Terminal works and actions are performed properly	Pass
5	Open a Python file	File is opened, when opening the tab for the file, the syntax highlighting is Python specific	Pass
6	Close tab without saving added contents	Asked if user is sure before closing	Pass

The application passed all tests conducted.

9.2 Future Work

A text editor has near endless possibilities to add in terms of improvements. Text editors in the industry are constantly getting updated with new features. There are some things that, with more time, I would like to add to this editor in order to improve it.

9.2.1 Collaborative Coding Features

Including a feature that would allow users to share the code they share in some way would be a great improvement to the editor, allowing teams to work on projects together. One method of collaborative code editing I code bring to the text editor would be for users to edit the same code in real-time. In order to achieve this, here is a basic overview of what I would need to do:

1. Choose a synchronisation method. There are different methods used for real-time synchronisation, for example Operational Transformation (OT), which allows users to apply edits independently and the system handles any conflicts.
2. Implement networking: use a socket or API to connect users to the same file. For example, WebSocket or REST API.
3. Display user presence in the editor's user interface.

These are very broad steps but show how it would be possible to add this feature.

9.2.2 Auto Check File Contents

A feature that watches for file/folder changes outside of the text editor while working on a project would be very useful to have. In order to implement this, I would need to:

1. Set up file watching: Use fs.watch() to listen for any changes to the open files.
2. Handle changes: If changes are made to the watched file, handle them in the renderer

10 Conclusion

In this project, I have developed a text editor aimed at enhancing the experience and productivity for developers. The project had many core requirements, which have all been fulfilled. Throughout this project, there have been many hurdles and challenges that I have faced, however all of these have improved my knowledge and understanding of the technologies I have used.

The text editor I have developed offers many features, however there is always more work that can be done to further improve it. In the future, I would like to further improve the text editor with the features mentioned in Sec. 9.2, as well as improve the user interface. There are endless features to add, making this text editor an exciting base for a potentially massive project.

Overall, I am happy with the result – a functional text editor with robust and impressive features, that has achieved the objectives set out.

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] electronjs.org. (n.d.). Inter-Process Communication | Electron. [online] Available at: https://www.electronjs.org/docs/latest/tutorial/ipc.

[9] 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].

[10] freeCodeCamp.org. (2022). Event Bubbling in JavaScript – How Event Propagation Works with Examples. [online] Available at: https://freecodecamp.org/news/event-bubbling-in-javascript/ [Accessed 2 May 2024].