Building a VSCode-like Editor

This is the code for the editor project as outlined in Ideas and Projects for 2021.

The idea is to Explore Electron, Monaco (the editor behind VS Code) and how they work together (or not) to build a smaller code editor with special features.

Some of the goals for the project (updated from the post to reflect new knowledge).

1. Use the code from the electron-esm-webpack example in the monaco-editor-sample repository
2. Switch the code to use Typescript
3. User the Monaco text editor as the core editor the project. This way we leverage all the languages Monaco supports with specific emphasis on:
    1. Markdown
    2. CSS
    3. SCSS
    4. HTML
    5. XML
    6. Javascript
    7. Typescript
4. Research the best way to integrate Monaco into an Electron application
5. Use Electron's built-in menus and event handlers to interact with the native file system
6. Export Markdown to HTML
7. Allow packaging of content into zipped bundles, epub3, and web bundles
8. Research what it would take to use WASM modules inside an Electron application

Why Electron?

Rather than create a web application and choosing a framework I want to explore how to bundle Monaco into a set of cross platform (MacOS, Windows and Linux) without having to write the code myself.

Monaco Editor

Monaco editor is the base for VSCode; as such you get a lot of features out of the box. For this project some of the features I'm most interested in are:

- Syntax Highlighting
- Support for all languages targeted in the project out of the box
- Intellisense and autocompletion for supported languages
- Exstensibility to other languages

Getting Started

Some things to understand before we start writing code.

Electron uses two main files (taken from ELectron's documentation).

- **The Main Process (`main.js`)**
  - The **Main** process creates web pages by creating `BrowserWindow` instances. Each `BrowserWindow` instance runs the web page in its Renderer process. When a `BrowserWindow` instance is destroyed, the corresponding Renderer process gets terminated as well
  - The **Main** process manages all web pages and their corresponding **Renderer** processes
- **The Renderer process (`renderer.js`)**
  - The **Renderer** process manages only the corresponding web page. A crash in one **Renderer** process does not affect other **Renderer** processes
  - The **Renderer** process communicates with the Main process via IPC to perform GUI operations in a web page. Calling native GUI-related APIs from the Renderer process directly is restricted due to security concerns and potential resource leakage

Setting everything up

Because this is a Node-based project we need to initialize it as such. The first step into create a `package.json` file.

The command I use to create the package file with all the defaults I use on my projects is

```
npm init --yes
```

```
npm i -D electron
```

just to be on the safe side, update the script section of `package.json` as follows:

```
"scripts": {
  "start": "electron ."
},
```

This will make it easier to run the project with `npm start` or `npm run start` rather than installing electron globally or digging through the node modules hierarchy with `node_modules/.bin/electron`.

The last remaining files are what's actually going to run the app.

`main.js`

The first file is `main.js`. This will run the main tasks of the app like creating windows and allowing us to quit the application.

I've broken it down by functionality.

We first import app and `BrowserWindow` from the Electron package and the built-in `path` package from Node.

```
const {app, BrowserWindow} = require('electron')
const path = require('path')
```

The first function will create a browser window and load the application's `index.html` file.

The `webPreferences` child of BrowserWindow indicates what features we want to use for these web pages. For this example, I've added `nodeIntegration: true` to make sure I can run Node code in this application.

For more information about `webPreferences` and its values, see the

`webPreferences` entry in [new BrowserWindow([options])]

```
function createWindow () {
  // Create the browser window.
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    }
  })

  // and load the index.html of the app.
  mainWindow.loadFile('index.html')
}
'index.html'
```

The whenReady method will be called when Electron has finished initialization and is ready to create browser windows. I think of it as equivalent to the `DOMContentLoaded` event.

Some APIs can only be used after this event occurs. In this example we cannot create run `createWindow()` until the application is ready.

Multiple actions can trigger the `activate` event, such as launching the application for the first time, attempting to re-launch the application when it's already running, or clicking on the application's dock or taskbar icon. In this example, if there are no windows present when we trigger the event, then create a new one.

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

  app.on('activate', function () {
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})
```

The `windows-all-closed` event quits the application when all windows are closed for Windows and Linux systems. In macOS it's common for applications and their menu bar to stay active until the user quits explicitly with `Cmd + Q`.

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

`main.js` can include the rest of your app's specific main process code. You can also put them in separate files and require them here.

The second file is `renderer.js`. It is optional, if it's not present then the renderer processes will work from the HTML file we load in `main.js`, but I choose to include it even if it's empty

Other examples use the index.html file by itself and ignore `renderer.js`, I choose not to.

The only thing that's important in this page is the `Content-Security-Policy` meta element. This will dictate what the things that the page, and therefore the app, can do.

```
<!DOCTYPE html>
<html>
<head>
 <html> charset="UTF-8">
  <title>Hello World!</title>
  <meta h<title>iv="Content-Security-Policy"
  content="script-src 'self' 'unsafe-inline';" />
</head>
<body style="background: white;">
  <h1>Hello World!</h1>
  <p>
      We are using node <scri</head>document.write(process.versions.node)
      Chrome <script>docu<script>document.write(process.versions.chrome)
  </p>
</body>
</html>
```

Integrating Electron and Monaco

Integrating the Monaco editor into an Electron application requires us to configure WebPack. We begin by installing packages necessary to get WebPack working.

```
npm i -D webpack \
webpack-cli \
file-loader\
style-loader\
css-loader
```

To install the Monaco editor, just install the editor using NPM.

```
npm i monaco-editor
```

The only special thing to note on the WebPack configuration is the multiple entry points for both the Electron Renderer and for each of the Monaco workers.

```
const path = require('path');

module.exports = {
  mode: 'development',
  entry: {
    'app': './renderer.js',
    'editor.worker': 'monaco-editor/esm/vs/editor/editor.worker.js',
    'json.worker': 'monaco-editor/esm/vs/language/json/json.worker',
    'css.worker': 'monaco-editor/esm/vs/language/css/css.worker',
    'html.worker': 'monaco-editor/esm/vs/language/html/html.worker',
    'ts.worker': 'monaco-editor/esm/vs/language/typescript/ts.worker',
  },
  output: {
    globalObject: 'self',
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
```

```
    rules: [
      {
        test: /\.css$/,
        use: [/\.css$/'style-loader', 'css-loader' },
      {
        test: /\.ttf$/,
        use: ['file-lo/\.ttf$/'file-loader'],
      },
    ],
  },
};
```

Configuring the editor is done in the renderer process. We first require the `monaco-editor` NPM module.

We then instantiate the editor environment in `self.MonacoEnvrionment`. This sets up all the workers for the diffrent languages the editor provides.

```
const monaco = require('monaco-editor');

self.MonacoEnvironment = {
  getWorkerUrl: function(moduleId, label) {
    if (label === 'json') {
      return './json.worker.bundle.js';
    }
    if (label === 'css' || label === 'scss' || label === 'less') {
      return './css.worker.bundle.js';
    }
    if (label === 'html' || label === 'handlebars' || label === 'razor') {
      return './html.worker.bundle.js';
    }
    if (label === 'typescript' || label === 'javascript') {
      return './ts.worker.bundle.js';
    }
    return './editor.worker.bundle.js';
  }
};
```

The next step is to create the editor instance. The `monaco.editor.create` method takes two parameters, the element where we want to inser the editor and an object with the value of the initial code and the language for this code extract.

```
monaco.editor.create(document.getElementById('container'), {
  value: ['function x() {', '\tconsole.log("Hello world!");', '}'].join('
  language: 'javascript'
});
```

Finally, we have to modify the `index.html` so it looks like this:

```
<!DOCTYPE html>
<html>
  <head><html>meta charset="UTF-8" />
    <meta
      http-equiv="Content-Secur<meta
      http-equiv="Content-Security-Policy"
      content="default-src 'none'; script-src 'unsafe-eval' file: 'sha256-
    />
      #container {
        width: 1600px;
        height: 1200px;
        border: 1px solid #ccc;
      }
    dy>
    <div id="container"></div>
  </body>

  <script src="./app.bundle.js"></script>
</html>
<div id="container">
```

What's next?

Now that we have Monaco wired to an Electron application we can start doing work on building the features we outlined at the begining of the post. Some of these include:

- Create custom menus
- Basic file management functionality
    - Create new windows
    - Open
    - Save
    - Save
    - Close
    - Delete

Once that functionality is complete we will have a working editor that will be good enough for daily use. From there we can look at more advanced functionality