ELECTRON SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Electron Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books 7

About the Author 9

Acknowledgements 10

Introduction 11

Chapter 1 Getting Started 12

Project overview 12

Prerequisites 12

Hardware and software requirements 12

Windows tooling quick start 13

Installing Electron 14

Summary 14

Chapter 2 Basic Building Blocks 15

Quick intro 15

Why plain JavaScript? 15

The problem 15

Creating the project 16

Our app's entry point: main.js 18

Replacing the default menu 22

Adding submenus 23

Invoking a new window 25

Recap: main.js 31

Creating: doc.html 34

Exiting the application 36

Closing secondary windows 37

Secondary window garbage collection....................37
Adding a shortcut key....................38
Optimizing the menu....................39
Enabling Developer Tools....................43
The code so far—main.js....................45
Summary....................50
Chapter 3 Expanding the App....................52
Quick intro....................52
The core logic....................52
Retransmitting the response: main.js....................55
Receiving the response: main.html....................55
Clearing the document list....................58
Add styling resources....................60
Styling the app....................64
Styling the list....................70
Statically rendered chip components....................73
Dynamically rendered chip components....................75
Checking dates....................86
Deleting documents....................87
Searching documents....................88
Add Document window facelift....................90
Data validation assumptions....................96
Adjusting the secondary window....................96
Ordering items by expiration date....................100
Recap of the code....................107
Summary....................125

Chapter 4 Packaging the App 126

App distribution...126

Electron Packager 126

Wrap-up and final thoughts...131

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

About the Author

Ed Freitas is a consultant on Software Development applied to Customer Success, mostly related to Financial Process Automation, Accounts Payable Processing, and Data Extraction.

He really likes technology and enjoys playing soccer, running, traveling, life hacking, learning, and spending time with his family.

You can reach him at: https://edfreitas.me

Acknowledgements

Many thanks to all the people who contributed to this book, including the amazing Syncfusion team that helped this book become a reality—especially Tres Watkins, Darren West, and Graham High.

The Manuscript Manager and Technical Editor thoroughly reviewed the book's organization, code quality, and overall accuracy: Darren West from Syncfusion, and James McCaffrey from Microsoft Research. Thank you both.

This book is dedicated to *Mi Chelin* and *Lala*, who inspire me every day and light up my path ahead—God bless you all, always.

Introduction

Wouldn't it be awesome if web apps could also run on the desktop—on Windows, Linux, and macOS—allowing web developers to reuse their HTML, CSS, and JavaScript skills? With Electron, this is possible.

Think of all the time that you, as a web developer, have spent learning web technologies. With Electron, those skills can also be applied to desktop development.

Electron is a framework built by the folks at GitHub (now part of Microsoft), which allows you to practically "convert" web applications into fully featured desktop applications, without sacrificing any web skills or technology.

Electron is a powerful and easy-to-learn platform that has enabled the rapid development of powerhouse apps such as Visual Studio Code, Slack, Atom, and countless others.

Whether you are inspired to create the world's next awesome desktop application, or you simply want to port your web app to run on the desktop, Electron is the one framework for you.

Before Electron, there was no easy way to port a web application over to the desktop without rewriting code—and depending on how many different host operating systems the application needed to be ported to, you would have ended up with multiple versions of the app, using different desktop UI frameworks—one for Windows, another for Linux, and so on. This is the importance and essence of Electron, and the problem it solves.

Before we start writing any code, I'll first describe the project we'll be building throughout this book, the prerequisites, the hardware and software requirements, and how to install Electron.

So, without further ado, let's get started.

Chapter 1 Getting Started

Project overview

The application we'll be building throughout this book will keep track of important personal documents that have an expiration date, such as a passport, a driver's license, or a credit card.

This type of application is handy to have, so we know when we need to renew these important documents before they expire.

Prerequisites

To make the most of this book, it's important that you have some basic web development knowledge, including HTML, CSS, and JavaScript.

No specific JavaScript framework knowledge is required, as all the code will be written using vanilla JavaScript (without using any specific JavaScript framework).

The Succinctly Series has you covered if you need to refresh your web development basics—most of these technologies are already explained by some of the other books in the series, so feel free to check out the awesome list of books available within the Succinctly library.

Hardware and software requirements

One drawback to using cross-platform tools and frameworks, such as Electron, is that there are many choices of hardware and software combinations to choose from.

Electron runs on any modern operating system, so even though I'll be doing everything on Windows 10, you should be able to follow along easily using a Mac or even Linux, since the code itself won't change.

Electron makes heavy use of Node.js, which is a JavaScrip-based, event-loop command runner built on top of Chrome's V8 engine. Node.js is a fundamental requirement. Besides Node.js, we'll also need Electron itself, and a CSS library.

For editing, I'll be using Visual Studio Code, which is easy to use, and it is available for free. Also known as VS Code, it's been designed from the ground up to work with the technologies we'll be using, and as it turns out, it's written in Electron.

You may also use any other editor of your choice, such as Atom or Sublime.

Windows tooling quick start

Although it's not mandatory, it's good to have the latest version of Git installed. Depending on your code editor or IDE of choice, it might be possible to avoid typing Git commands, and we probably won't be using Git directly.

Nevertheless, Git might be used by some of the tools we'll be using, so it doesn't hurt to have it installed—besides, it's always good to commit your code.

If you open your browser and navigate to the Git website using this link, you'll be able to easily find where to download Git from. In my case, it looks as follows.

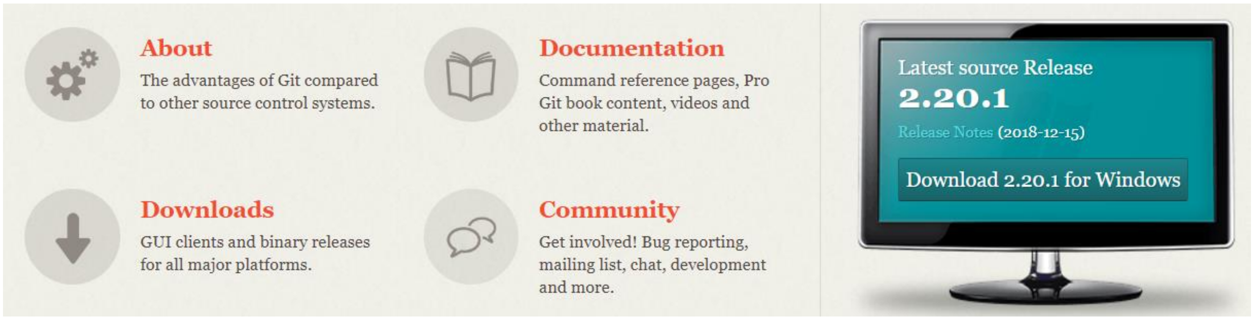

About
The advantages of Git compared to other source control systems.
Documentation
Command reference pages, Pro Git book content, videos and other material.
Downloads
GUI clients and binary releases for all major platforms.
Community
Get involved! Bug reporting, mailing list, chat, development and more.
Latest source Release
2.20.1
Release Notes (2018-12-15)
Download 2.20.1 for Windows

Figure 1-a: Where to Download Git

One of the advantages of installing Git is that it also installs a much better command prompt, called Git Bash, which I recommend using whenever possible.

You should be able to click **Downloads for Windows**, and then download Git and install with the default options. The process is incredibly simple and intuitive to follow.

To get an automated Git installation on Windows, you can use the Git Chocolatey package.

Once you've installed Git, the next tool you'll need to install is Node.js. I recommend installing the latest LTS (Long Term Support) version, which is clearly indicated on the Node.js website as follows.

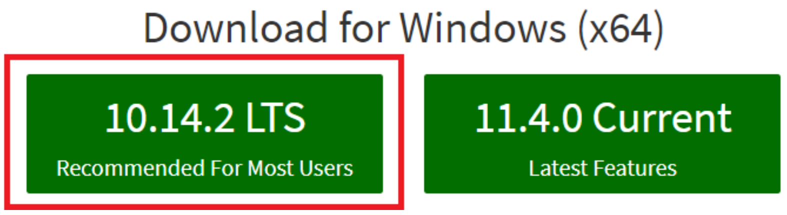


Figure 1-b: Where to Download Node.js

Once it's downloaded, run the installer with the default options and follow the installation wizard—the process is straightforward and incredibly simple. However, if you would like to know more about it, the Node.js installation process is covered in depth within Ionic Succinctly.

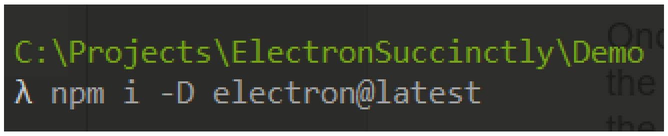
Installing Electron

Now that we have Git and Node.js installed, we need to install Electron. To do that, open the command line, and go to the folder where you want to keep the source code of the demo application, which we will be building throughout this book. Then, type in the following command.

Listing 1-a: Installing Electron Command

```
npm i -D electron@latest
```

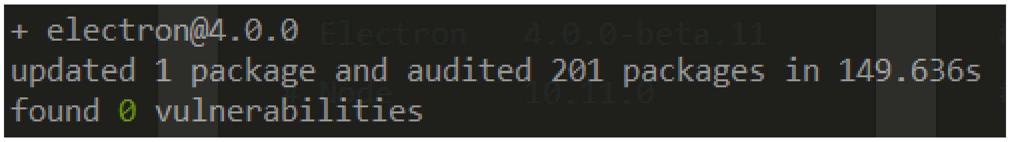
In my case, it looks like this:



```
C:\Projects\ElectronSuccinctly\Demo
λ npm i -D electron@latest
```

Figure 1-c: Installing Electron through the Command Line

Once the command has been executed and the installation process is finished, you should see some results.



```
+ electron@4.0.0
updated 1 package and audited 201 packages in 149.636s
found 0 vulnerabilities
```

Figure 1-d: Post-Installation Electron Results

With Electron installed, we are ready to start coding.

Summary

In this very brief chapter, we’ve quickly introduced Electron and explained how to set it up. In the next chapter, we’ll explore all the steps required to build the foundation of our Electron desktop app, using plain JavaScript—this way we can be framework agnostic.

Chapter 2 Basic Building Blocks

Quick intro

In this chapter, we’ll focus on how to build the foundation of the desktop application we’ll be writing throughout this book.

We’ll look at the basics of Electron, the benefits of using plain JavaScript for creating the logic of our desktop app, and describe the problem we’ll be solving.

Why plain JavaScript?

As you probably know, one of the biggest challenges that developers face today is how to choose the right JavaScript framework. There are many amazing JavaScript frameworks out there, but the learning curve can be steep, depending on which one you choose.

The fundamental idea behind Electron is that cross-platform desktop applications should be written using web development techniques and languages—which in today’s world, usually means working with a JavaScript framework, although Electron does not enforce the usage of any framework in particular.

JavaScript frameworks are great and provide a lot of amazing features, such as dependency injection, two-way data binding, reusable components, routing, and more. However, this also adds an extra layer of complexity, which might not be needed for creating a basic Electron app and understanding fundamental concepts.

For instance, routing in a desktop app might not be needed, as desktop apps could have multiple windows. It might be easier, and more logical, for users accustomed to working with desktop applications, to use windows and menu navigation rather than routing, like with web apps—even though the underlying language used for both is JavaScript.

So, writing Electron apps in plain JavaScript offers advantages, such as being able to use familiar windows and menu navigation, which is common in desktop applications. This also provides a much easier way to understand how Electron works—without getting too preoccupied with how to integrate a JavaScript framework into an Electron app.

The problem

Important documents can be lifesavers. Suppose your work suddenly requires you to travel abroad, and you realize that your passport has expired. So, one of your colleagues is sent off to do the job and your passport expiration issue doesn’t go down very well with your manager.

Alternatively, suppose you have your passport in order, but your company's credit card has expired. Since you don't notice it until you're asked for the credit card when checking out, you're left paying the bill that you thought you were going to pay with the company's card.

These are simple examples of why it's good to know in advance when an important document is going to expire—this way, you can plan when to renew it with sufficient time.

Throughout this book, we'll build an Electron app using plain JavaScript that will help keep track of documents that have an expiration date.

We'll focus on the Electron aspects of the application, so you can understand how Electron works. This will allow us to build the standard desktop application functionality.

Later, if you wish, you can expand the application by adding any other features you consider necessary—such as saving the data on a database. The application will not save any data; it will only display it on the screen.

Creating the project

For the deployment of our application, we'll use called Electron Packager, which will allow us to package and deploy the application to Windows, Mac, and Linux. However, the deployment doesn't come until we've finalized our application, so first we need to create our project.

You can open Visual Studio Code and the command line—in my case, I'll use this command line tool I love, but feel free to use one of your choice.

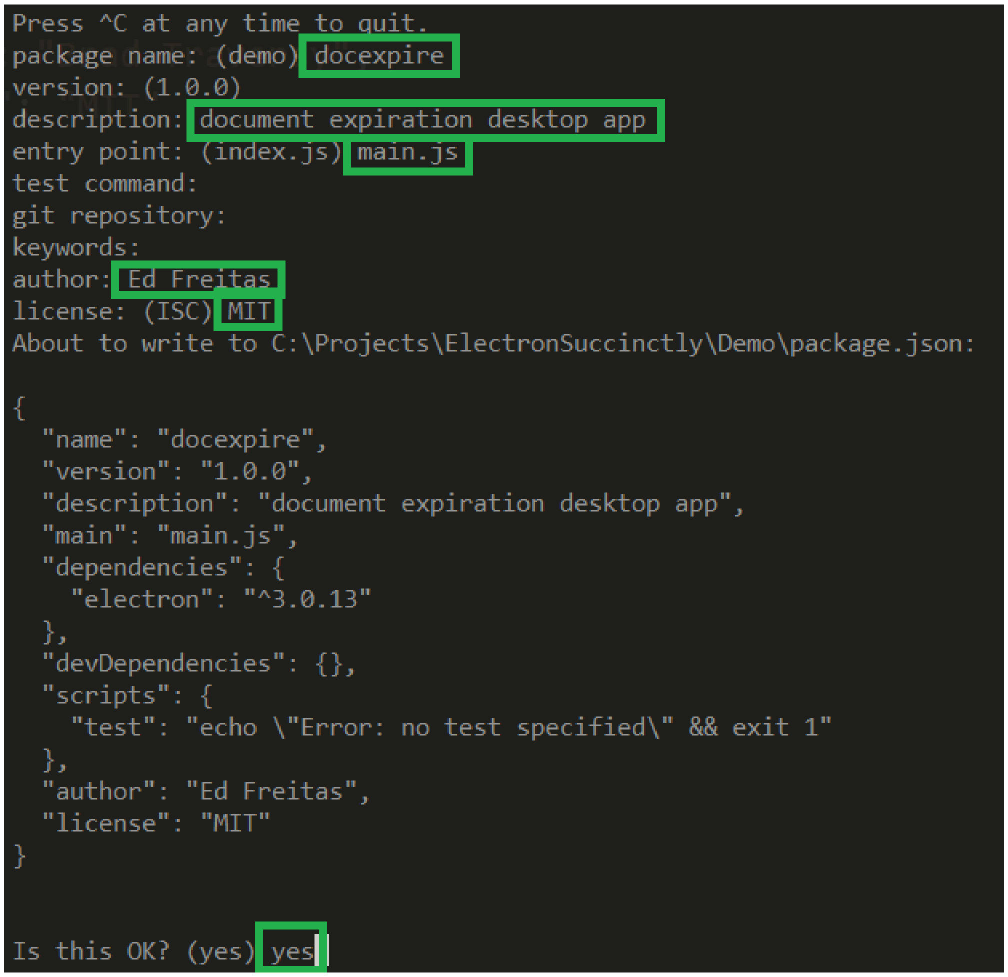
Using the command line, go into the project folder and execute the following command.

Listing 2-a: The "npm init" Command

```
npm init
```

This command is used to create the main project file (**package.json**), which we will use to include Electron and other dependencies we might need for creating the app.

Follow the steps and answer the questions—in my case, it looks like this:



```
Press ^C at any time to quit.
package name: (demo) docexpire
version: (1.0.0)
description: document expiration desktop app
entry point: (index.js) main.js
test command:
git repository:
keywords:
author: Ed Freitas
license: (ISC) MIT
About to write to C:\Projects\ElectronSuccinctly\Demo\package.json:

{
  "name": "docexpire",
  "version": "1.0.0",
  "description": "document expiration desktop app",
  "main": "main.js",
  "dependencies": {
    "electron": "^3.0.13"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ed Freitas",
  "license": "MIT"
}


Is this OK? (yes) yes
```

Figure 2-a: Creating the package.json File

As you might have noticed, under the **`dependencies`** section, Electron has been automatically added to our project's package.json file, because we had already installed Electron under the project's folder using another **`npm`** command.

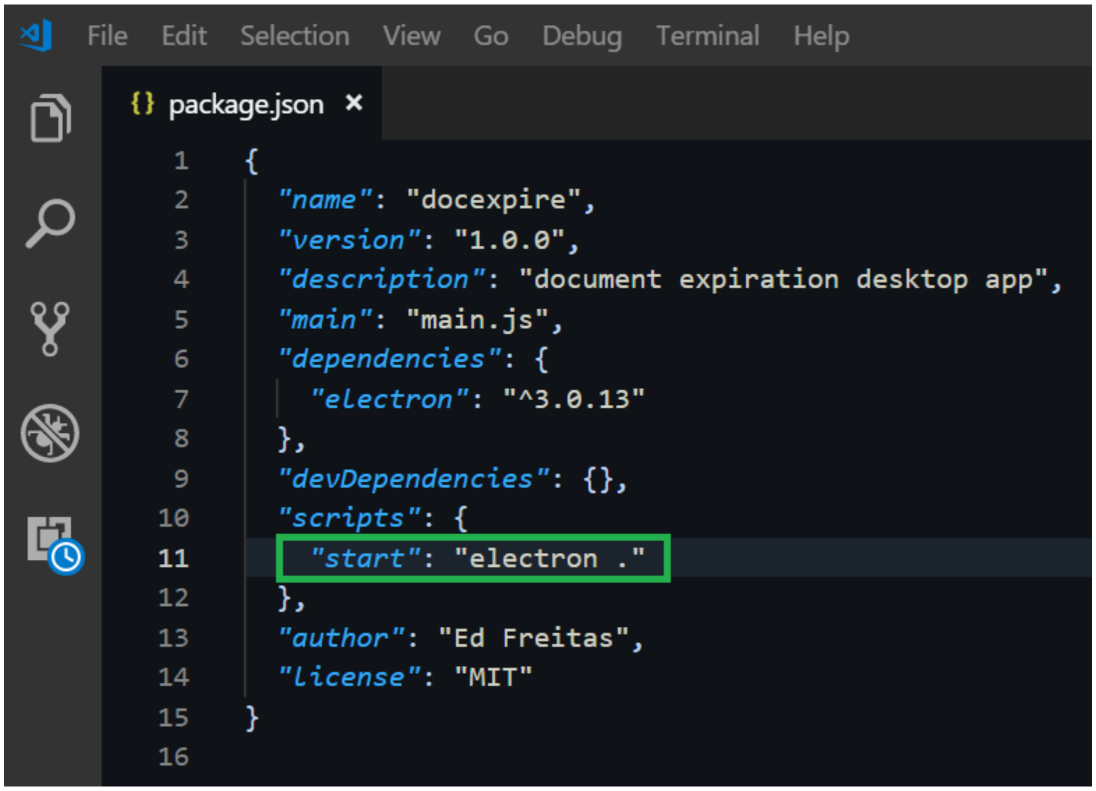
If for some reason you missed installing Electron previously, then simply run the following command to install it now.

Listing 2-b: The "npm install --save" Command

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

This command will add the Electron dependency to the **package.json** file.

Then using Visual Studio Code, open the **package.json** file. We'll need to modify the **`scripts`** section by removing the **`test`** subsection and replacing it with a **`start`** subsection, which will allow us to start Electron, using the **`start`** command.



```
{
  "name": "docexpire",
  "version": "1.0.0",
  "description": "document expiration desktop app",
  "main": "main.js",
  "dependencies": {
    "electron": "^3.0.13"
  },
  "devDependencies": {},
  "scripts": {
    "start": "electron ."
  },
  "author": "Ed Freitas",
  "license": "MIT"
}
```

Figure 2-b: The "start" Script

This is all we need to do with the **package.json** file—now, we're ready to start coding.

Our app's entry point: main.js

You might have noticed that when the **`npm init`** command was executed, we were asked to give the name of the app's main entry point. I decided to call it **main.js**; however, we could have given it any other name.

I chose this name simply to make it easier for desktop developers coming from a C# or C++ background, to easily correlate this *entry point concept* with the name of the **`main`** method, which is used to start programs written in these languages.

With Visual Studio Code opened, go ahead and create the main.js file under your project folder. This is where we will place the main code for our application.

So, the first thing we need to do within our main.js file is to bring in the **`Electron`** module and a couple of Node.js modules, which are complementary to working with Electron—these are the **`url`** and **`path`** modules.

Listing 2-c: The Main Required Modules (main.js)

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

const url = require('url');

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

Let's define the **app**, **BrowserWindow**, and **menu** constants that our application will use.

Listing 2-d: The app, BrowserWindow and menu Constants (main.js)

```
const {app, BrowserWindow, Menu} = electron;
```

Next, let's define the variable that we will use for our application's main window.

Listing 2-e: The Main Window Variable (main.js)

```
let main;
```

These three simple steps are all we need to start developing the skeleton of our application.

So, let's work on the main window logic—to do that, we first need to subscribe to the **ready** event using the **on** function with lambda syntax, which indicates that the application is ready. We can do this as follows.

Listing 2-f: The Application Ready Event (main.js)

```
app.on('ready', () => {

});
```

As you can see, we pass an arrow function to the **ready** event, which will execute the code necessary for creating the browser window.

Next, inside the **ready** event, let's create the **main** window object and pass it the name of the HTML file that will contain its user interface. We can do this as follows.

Listing 2-g: The Complete Application Ready Event (main.js)

```
app.on('ready', () => {

    // Create the new window

    main = new BrowserWindow({});

    // Load the html into the window

    main.loadURL(url.format({

        pathname: path.join(__dirname, 'main.html'),

        protocol: 'file',

        slashes: true

    }));

});
```

Let's quickly review what we've done. First, we create the browser window instance that will be responsible for executing and rendering the main user interface of our application—represented by main.html—which we will create shortly.

Second, we pass to the browser window object the location of the main.html file—this is what the **`loadURL`** method does.

Although the code for the **`loadURL`** method looks very complex, all it is essentially doing is passing the file path of main HTML file that will be loaded.

Listing 2-h: Example of a File Path

```
// Example of the file path that will be loaded in main.js

// file://dirname/main.html
```

We now have the basic skeleton for our application. But, before we can try it out and see how it looks, let's quickly create the main.html file and add the following boilerplate code.

Listing 2-i: Basic main.html Code

```
<!DOCTYPE html>
```

```
<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport"

        content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>DocExpire</title>

</head>

<body>

    <h1>DocExpire</h1>

</body>

</html>
```

Save the changes, and on the command line, type **`npm start`** and press **Enter**. Now you should be able to see the application running.

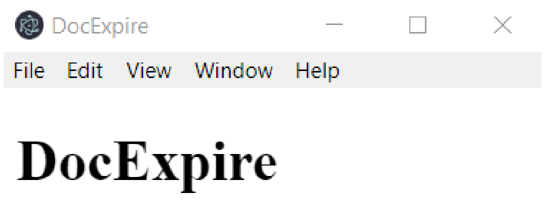
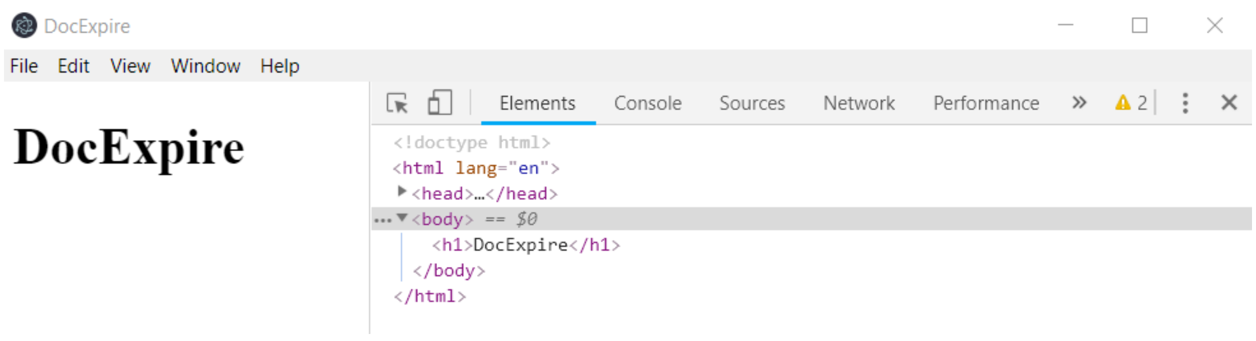


Figure 2-c: The Application Running

Awesome—we've now executed our first "Hello World" Electron app.

Notice though, that because we haven't created our own menu items, it's using the default menu items that are provided out of the box, including the Chrome Developer Tools, accessible through the View menu.



```
<!doctype html>
<html lang="en">
▶<head>…</head>
…▼<body> == $0
    <h1>DocExpire</h1>
  </body>
</html>
```

Figure 2-d: The Application Running with Chrome Developer Tools Opened

The next thing we'll do is to replace the default menu with our own.

Replacing the default menu

We can replace the default menu by creating a menu template, which in Electron is an array of objects. We can do this as follows in main.js, just before the **`ready`** event.

Listing 2-j: Our Menu Template (main.js)

```
const menuTemplate = [
    {
        label: 'File'
    }
];
```

For now, we are simply creating a File menu to quickly test it out. This defines the menu template, but we still need to add it to our application; we can do this as follows inside the **`ready`** event.

Listing 2-k: Building and Adding the Menu (main.js)

```
app.on('ready', () => {
    // … Previous code goes here …
    // Build the menu from the template
    const menu = Menu.buildFromTemplate(mainMenu);


    // Insert the menu into the application
    Menu.setApplicationMenu(menu);
});
```

As you can see, the menu is built by invoking the **`buildFromTemplate`** method, and then it's added to the application with the **`setApplicationMenu`** method.

Let's run **`npm start`** from the command line to see if our menu replaces the default menu options.

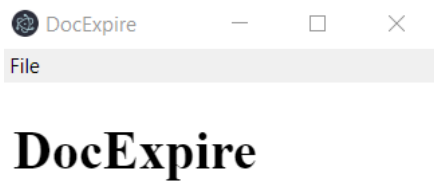


Figure 2-f: The Application using the Menu Template

If you are using a Mac, you might have noticed that the **File** menu is not called **File**, but instead is called, **Electron**.

This is because menus are rendered differently on a Mac than on other platforms. So if you are using a Mac and see **Electron** instead of **File**, don't worry—we'll later explore how to address this.

Awesome—that's how easy it is to replace the default menu and add our own.

Adding submenus

We've just added a very basic menu template; however, that's not enough for our application's functionality.

So how can we add a submenu to our existing menu? If all we had to do to add a menu was to define an array of objects, then adding a submenu is no different—a submenu is just an array of objects inside a menu option.

With this said, let's define some submenu options for the File menu that we will use in our application. We can do this as follows.

Listing 2-I: Adding Submenus (main.js)

```
const mainMenu = [
    {
        label: 'File',
        submenu:[
            {
                label: 'Add Document'
            },
            {
                label: 'Clear Documents'
            },
            {
                // For cloud syncing - for the future
                label: 'User Details'
            },
            {
                label: 'Exit App'
            }
        ]
    }
];
```

For now, we've just added the label for the menu and submenus—later, we'll finish off the menu template, but for now, this is enough.

Notice how we have added four submenus—the first option is to add a new document, the second option is to clear all the documents, the third is to add user details, and the last option is to exit the application.

By "document," I'm referring to a document that has an expiration date—remember, we are building an application that is going to keep track of the expiration date of important documents.

The option to add user details could be used in the future, so that a list of documents is associated with an email address, which means that we could use the same application for different users. Each user could have its own list of documents, and this list could be synced between machines running the application, with something like Cloud Firebase.

Invoking a new window

There's no reason why we couldn't include all our application's logic within the scope of one window. However, most desktop applications (regardless of the platform) rely on more than one main window. Therefore, a key element of developing Electron applications is being able to use multiple windows and benefitting from intra-window messaging.

Let's define a `click` event for the **Add Document** menu option, which will allow us to open a new window, which we can use to add the details of a new document.

Listing 2-m: The Add Document Click Event (main.js)

```
const mainMenu = [

    {

        label: 'File',

        submenu:[

            {

                label: 'Add Document',

                click() {

                    addDocumentWindow();

                }

            },

            // … Existing code …
        ]

    }

];
```

On the **`click`** event, we are invoking the **`addDocumentWindow`** function, which we have not defined.

Let's go ahead and define this function—just before the **`ready`** event, and after the **`mainMenu`** template—within main.js.

Listing 2-n: The addDocumentWindow Function (main.js)

```
function addDocumentWindow() {

    // the addDocumentWindow functionality

}
```

Here comes the interesting bit: what do you think will go inside this function? If we are going to create a new window, then it's pretty much going to be the same logic we originally had inside the **`ready`** event, before we replaced the default menu.

In other words, we already know how to do this—we can add this logic just before the **`ready`** event.

Listing 2-o: The addDocumentWindow Function (main.js)

```
let addDocWin;

// Create Add Document window

function addDocumentWindow() {

    // Create the new window

    addDocWin = new BrowserWindow({});

    // Load the html into the window

    addDocWin.loadURL(url.format({
        pathname: path.join(__dirname, 'doc.html'),

        protocol: 'file',

        slashes: true

    }));

}
```

This would create a new window. Notice that we are repeating the same code we wrote for most of the **`ready`** event. We can refactor this code as follows.

Listing 2-p: The Code Refactored (main.js)

```
function addWindow(name) {

    // Create the new window

    let win = new BrowserWindow({});

    // Load the html into the window

    win.loadURL(url.format({

        pathname: path.join(__dirname, name + '.html'),

        protocol: 'file',

        slashes: true

    }));


    return win;

}
```

By refactoring the code this way, we can reuse this function to create the Main window and the window to add a new document, by passing the **`name`** of the HTML file that the window will display and returning the handler to the window—**`win`**.

Now, let's rewrite the **`addDocumentWindow`**, so we can create the Add Document window like we originally intended to.

Listing 2-q: The New AddDocumentWindow Function (main.js)

```
function addDocumentWindow() {

    addDocWin = addWindow('doc');

}
```

Basically, this function just calls **`addWindow`** and passes the name of the HTML file for which the window will be created.

To make our code even cleaner, we can move the declaration of the **`addDocWin`** variable to the same line where we have the **`main`** variable declared.

Listing 2-r: The main and addDocWin Variables (main.js)

```
let main, addDocWin;
```

With all this refactoring done, the **`ready`** event code now looks as follows.

Listing 2-s: The Refactored ready Event (main.js)

```
app.on('ready', () => {

    // Create the new window

    main = addWindow('main');


    // Build the menu from the template

    const menu = Menu.buildFromTemplate(mainMenu);


    // Insert the menu into the application

    Menu.setApplicationMenu(menu);

});
```

We can now run **npm start** and test the application. Click the **Add Document** menu option and check if the new window opens.

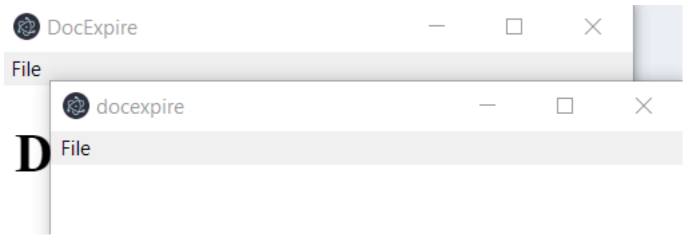


Figure 2-g: The Application Running (Two Windows Opened)

Notice that when you click the **Add Document** option, an almost identical second window opens. This second window has the same width, height, and menu as the main window—however, notice that the title on the second window is in lowercase, and that there is no content within the second window—this is because we have not yet created the doc.html file.

Let's add some parameters when we open the second window—this way, we can customize the **`width`**, the **`height`**, and the **`title`** bar text.

We can do this by adjusting the **`addWindow`** function as follows.

Listing 2-t: The Adjusted addWindow Function (main.js)

```
function addWindow(name, params) {
    // Create the new window
    let win = new BrowserWindow(params);
    // Load the html into the window
    win.loadURL(url.format({
        pathname: path.join(__dirname, name + '.html'),
        protocol: 'file',
        slashes: true
    }));

    return win;
}
```

As you can see, we've added a **`params`** parameter. So, how can we adjust the **`width`**, **`height`**, and **`title`** using **`params`**? We can do that by using **`params`** as an object rather than an individual property.

Let's modify the **`addDocumentWindow`** function and the **`ready`** event, as follows.

Listing 2-u: The Adjusted addDocumentWindow Function and ready Event (main.js)

```
function addDocumentWindow() {
    addDocWin = addWindow('doc',
        {width: 200, height: 300, title: 'Add Document'});
}

app.on('ready', () => {
    // Create the new window
    main = addWindow('main', {});

    // Build the menu from the template
    const menu = Menu.buildFromTemplate(mainMenu);

    // Insert the menu into the application
    Menu.setApplicationMenu(menu);
});
```

Now, run **`npm start`** to test the application. Click the **Add Document** menu option to see how the application looks now.

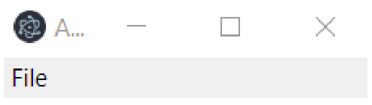


Figure 2-h: The Second Window Opened

Notice how the second window opened with the defined **`width`**, **`height`**, and **`title`** bar, and appeared centered to the main window.

The title is barely visible; it can be better seen by adjusting the **`width`** and **`height`**.

Recap: main.js

With some code and explanations behind, let’s recap all the code we’ve written so far in main.js.

Listing 2-v: main.js

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

const url = require('url');

const path = require('path');


const {app, BrowserWindow, Menu} = electron;


let main, addDocWin;


// Menu template

const mainMenu = [

    {

        label: 'File',

        submenu:[

            {

                label: 'Add Document',

                click() {

                    addDocumentWindow();

                }
            },
            {
                label: 'Clear Documents'
            },
            {
                // For cloud syncing - for the future
                label: 'User Details'
            },
            {
                label: 'Exit App'
            }
        ]
    }
];

// Create a new window
function addWindow(name, params) {
    // Create the new window
    let win = new BrowserWindow(params);
    // Load the html into the window
    win.loadURL(url.format({
        pathname: path.join(__dirname, name + '.html'),
        protocol: 'file',
        slashes: true
    }));


    return win;
}


function addDocumentWindow() {
    addDocWin = addWindow('doc',
        {width: 200, height: 300, title: 'Add Document'});
}


// Application ready event
app.on('ready', () => {
    // Create the new window
    main = addWindow('main', {});


    // Build the menu from the template
    const menu = Menu.buildFromTemplate(mainMenu);


    // Insert the menu into the application
    Menu.setApplicationMenu(menu);
});
```

Creating: doc.html

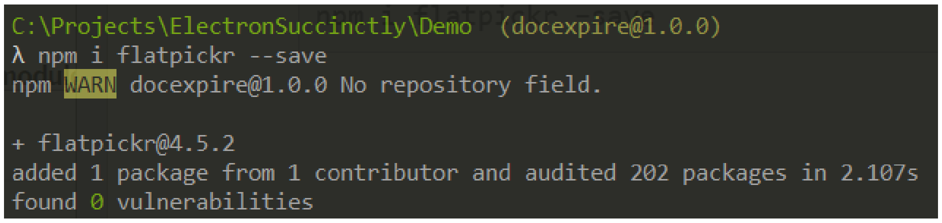
Now that we know how to create the Add Document window, we need to be able to add some HTML markup to it—this way we can start to give our app its character and "look and feel."

As our application is all about keeping track of documents and their expiration dates, I'm going to leave it up to you, to use a datetime picker component—I won't use this myself, but I thought I would mention it.

If you would like to get this component installed and later experiment with it, just run the following command from the application project folder.

Listing 2-w: Installing a Datetime Picker Component

```
npm i flatpickr -save
```



```
C:\Projects\ElectronSuccinctly\Demo  (docexpire@1.0.0)
λ npm i flatpickr --save
npm WARN docexpire@1.0.0 No repository field.

+ flatpickr@4.5.2
added 1 package from 1 contributor and audited 202 packages in 2.107s
found 0 vulnerabilities
```

Figure 2-i: The Datetime Picker Component Installed

Let's define the basic HTML markup for our doc.html file. If you haven't yet created the doc.html file, go ahead and create it in Visual Studio Code or the editor you are using.

Then, paste the following markup and save the doc.html file.

Listing 2-x: Basic doc.html Markup

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport"

        content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <form>
        <div>
            <label>Document name:</label>
            <input type="text" id="name" autofocus>
            <label>Expiration date:</label>
            <input type="text" id="date">
        </div>
        <button type="submit">Save</button>
    </form>
</body>
</html>
```

If you now run the **`npm start`** command to test the application, and then click the **Add Document** menu option, you should see the following.

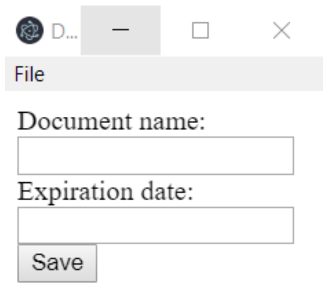


Figure 2-j: Executing doc.html

Notice that we haven't added any styling or functionality to this form yet, so it's just markup. We'll later add Google's Material Design to make our markup look more appealing.

Although we have slightly deviated from the main.js logic that we've been working on since the beginning of this chapter, I wanted to highlight how we can easily create a secondary window—which we'll then link back to the logic within main.js—using an ipcRenderer object.

Exiting the application

With the **`File`** submenu options defined, let's quickly see how we can add the logic required to exit the application.

Listing 2-y: Exiting the Application (main.js)

```
const mainMenu = [
    {
        label: 'File',
        submenu:[
            // … Existing code …
            {
                label: 'Exit App',
                click() {
                    app.quit();
                }
            }
        ]
    }
];
```

As you can see, exiting the application is as simple as adding a **`click`** event and invoking the **`app.quit`** method.

Now, save the changes to main.js, and then run **`npm start`** to give it a shot.

With the application opened, go to **File** > **Exit App**, and the application should properly close.

Closing secondary windows

When developing Electron applications, keep in mind that secondary windows that are opened will not be closed when the main window is closed.

Say your application is running, and both the main window and the Add Document window are opened, but you close the main window. What will happen is that the secondary window will remain opened.

The ideal behavior for any desktop application—what users normally expect—is that once the main window is closed, any secondary windows should also be closed, and the application terminated.

To achieve this, we need to add a bit of extra code to the **`ready`** event, as follows.

Listing 2-z: Terminating the Application (main.js)

```
app.on('ready', () => {
    // … Existing code …

    // Terminating the application
    main.on('closed', () => {
        app.quit();
    });
});
```

As you can see, all we are doing is to listen for the **`closed`** event from the **`main`** window to get triggered—when that happens, an arrow function is executed, with the instruction to finalize the application.

If you now run **`npm start`**, and then click on the **Add Document** menu option and try to close the main window, the secondary window should also be closed, and the application terminated.

Secondary window garbage collection

It's great that we can now successfully close all windows and terminate the app when closing the main window.

However, any variable that holds a reference to a secondary window will keep holding that memory space—therefore, it's good to do some garbage collection for secondary windows.

We can do this by adding the following code to the **`addDocumentWindow`** function.

Listing 2-aa: Secondary Windows Garbage Collection (main.js)

```
function addDocumentWindow() {
    // … Existing code …

    // Garbage collection
    addDocWin.on('close', () => {
        addDocWin = null;
    });
}
```

As you can see, we set the secondary window variable to **`null`** when the window's **`close`** event is triggered—this will release the memory space allocated for this window.

Although this isn't required, it's a recommended practice that optimizes how our Electron application runs.

Adding a shortcut key

Besides being able to close the application through the menu option, I'd also like to be able to close it using a shortcut key (for example, Ctrl+E). What we can do is to add an *accelerator*.

Adding an accelerator is quite easy, but we need to make sure that we remain operating-system agnostic—which is the whole purpose of writing an Electron app in the first place. Therefore, on a Mac, instead of Ctrl+E, we'll have to use Command+E.

To achieve this, we'll have to ask Electron to check on which operating system it is running. We can achieve this by invoking the process.platform property from the Node.js API.

Remember that Electron internally uses Node.js, so we can invoke Node.js properties and methods within our Electron app.

If we are on a Mac, we can use the Command+E shortcut key to close the app, and on any other platform—including Windows—we can use Ctrl+E.

We can evaluate if we are on a Mac or another platform by using a ternary operator and checking if the value of **`process.platform`** is equal to **`darwin`**.

Listing 2-ab: The Exit App Accelerator inside mainMenu (main.js)

```
const mainMenu = [
    {
        label: 'File',
        submenu:[
            // … Existing code …
            {
                label: 'Exit App',
                accelerator: process.platform == 'darwin' ?
                    'Command+E' : 'Ctrl+E',
                click() {
                    app.quit();
                }
            }
        ]
    }
];
```

If you save the changes to main.js and run **`npm start`**, you should be able to exit the application using the shortcut key defined.

Optimizing the menu

I'm running Windows, but if you are following on a Mac, when the application has been running, you've probably noticed that the **File** menu is called **Electron** instead. As I explained previously, this is because menus are rendered differently on a Mac than they are on other platforms.

This can be easily solved by adding an “almost empty” object to the beginning of the **`mainMenu`** array. We can do this as follows.

Listing 2-ac: The Menu Template for a Mac (main.js)

```
const mainMenu = [
    // Adding this empty object solves the File menu issue on a Mac
    { label: '' },
    {
        label: 'File',
        submenu:[
            // … Existing code …
        ]
    }
];
```

Notice that all we have done is to add **`{ label: '' }`** before defining the **`File`** menu option.

If you are testing this application on a Mac, you’ll be happy; however, if you are using Windows or another platform and execute the **`npm start`** command, you’ll notice that an empty option has been added to the menu bar.

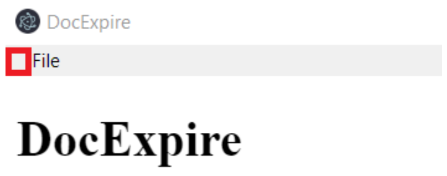


Figure 2-k: Empty Menu Option on Windows

So, we’ve managed to address the issue on a Mac, but created a problem on other platforms. Also, this solution of adding an empty element to the menu template array is not very elegant.

The way to address this, is to check the value of the **`process.platform`** property from the Node.js API, and only then, add the empty object when we are on a Mac. We can do this as follows.

Listing 2-ad: The menuMac Function (main.js)

```
function menuMac() {

    if (process.platform == 'darwin') {

        mainMenu.unshift({ label: '' });

    }

}
```

I’ve created a new function called **`menuMac`**, which can be placed on main.js after the **`mainMenu`** template declaration.

Basically, the **`unshift`** array method adds an element to the beginning of the array—in this case, we are adding the object **`{ label: '' }`** to **`mainMenu`** when the system detects that we are on a Mac, instead of hardcoding it within the array itself.

Don’t forget to remove **`{ label: '' }`** from the from **`mainMenu`**. The menu template should now look as follows.

Listing 2-ae: The Menu Template (main.js)

```
const mainMenu = [

    {

        label: 'File',

        submenu:[

            // … Existing code …

        ]

    }

];
```

For this to work, we’ll need to invoke the **`menuMac`** function from the **`ready`** event. Our updated **`ready`** event should look as follows.

Listing 2-af: The Updated ready Event (main.js)

```
app.on('ready', () => {
    // … Existing code …


    menuMac();


    // Build the menu from the template
    const menu = Menu.buildFromTemplate(mainMenu);


    // Insert the menu into the application
    Menu.setApplicationMenu(menu);


    // … Existing code …
});
```

I'm quite happy with this code, but whenever I'm writing code, I like to keep things as clean as possible—which means that if there's anything that can be optimized, I'll do it.

In this case, there are three lines of code within the `ready` event that are related to displaying the menu. It's a good idea to refactor those three lines into a separate function—let's do that.

Listing 2-ag: The displayMenu Function (main.js)

```
function displayMenu() {
    menuMac();


    // Build the menu from the template
    const menu = Menu.buildFromTemplate(mainMenu);

    // Insert the menu into the application
    Menu.setApplicationMenu(menu);
}
```

I’ve refactored the menu-related code into a new function called **`displayMenu`**, which we can now invoke from the **`ready`** event. Let’s adjust the **`ready`** event accordingly.

Listing 2-ah: The Updated ready Event (main.js)

```
app.on('ready', () => {
    // … Existing code …


    displayMenu();


    // … Existing code …
});
```

If you run the application, you’ll see that visually, nothing has changed.

Enabling Developer Tools

By adding our own menu template, we’ve lost access to the Chrome Developer Tools menu option—which can still be handy to have around during development, but not in production.

So, let’s create a function that can allow us to enable the Chrome Developer Tools when we are not running our application in a production environment. Let’s add this new function just before **`displayMenu`**.

Listing 2-ai: Function to Enable the Chrome Developer Tools (main.js)

```
function enableDevTools() {

    if (process.env.NODE_ENV === undefined ||

        process.env.NODE_ENV !== 'production') {

        mainMenu.push({

            label: 'DevTools',

            submenu: [

                {

                    label: 'Toggle DevTools',

                    accelerator: process.platform == 'darwin' ?

                        'Command+T' : 'Ctrl+T',

                    click(i, fw) {

                        fw.toggleDevTools();

                    }

                }

            ]

        });

    }

}
```

We've inserted an element—only when we our application is not in production—that inserts the **DevTools** menu option into the **`mainMenu`** array by using the JavaScript **`push`** method.

Notice how we've also added a keyboard accelerator, with the option to toggle (hide/show) the **DevTools** menu option, using the **`toggleDevTools`** method.

Now, we need to invoke the **`enableDevTools`** function from **`displayMenu`**—we can do this as follows.

Listing 2-aj: The Updated display (main.js)

```
function displayMenu() {

    menuMac();

    enableDevTools();


    // … Existing code …

}
```

If you now run the application with **`npm start`**, you should see the **DevTools** option on the menu bar.

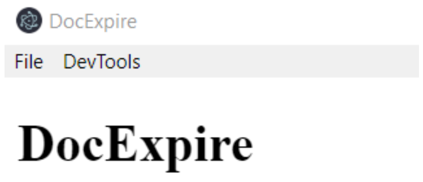


Figure 2-l: The Application with the DevTools Menu Option

The code so far—main.js

We’ve covered a lot of ground to get the basic structure of our Electron application off the ground. Let’s have a quick look at how our main.js file looks now.

Listing 2-ak: The Code So Far (main.js)

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

const url = require('url');

const path = require('path');


const {app, BrowserWindow, Menu} = electron;
```

```
let main, addDocWin;


// Menu template

const mainMenu = [

    {

        label: 'File',

        submenu:[

            {

                label: 'Add Document',

                click() {

                    addDocumentWindow();

                }

            },

            {

                label: 'Clear Documents'

            },

            {

                // For cloud syncing - for the future

                label: 'User Details'

            },

            {

                label: 'Exit App',

                accelerator: process.platform ==

                    'darwin' ? 'Command+E' : 'Ctrl+E',
                click() {
                    app.quit();
                }
            }
        ]
    }
];

function menuMac() {
    if (process.platform == 'darwin') {
        mainMenu.unshift({ label: '' });
    }
}

function enableDevTools() {
    if (process.env.NODE_ENV === undefined ||
        process.env.NODE_ENV !== 'production') {
        mainMenu.push({
            label: 'DevTools',
            submenu: [
                {
                    label: 'Toggle DevTools',
                    accelerator: process.platform ==
                        'darwin' ? 'Command+T' : 'Ctrl+T',
                    click(i, fw) {
                        fw.toggleDevTools();
                    }
                }
            ]
        });
    }
}

function displayMenu() {
    menuMac();
    enableDevTools();

    // Build the menu from the template
    const menu = Menu.buildFromTemplate(mainMenu);

    // Insert the menu into the application
    Menu.setApplicationMenu(menu);
}

// Create a new window
function addWindow(name, params) {
    // Create the new window
    let win = new BrowserWindow(params);
    // Load the html into the window
    win.loadURL(url.format({
        pathname: path.join(__dirname, name + '.html'),
        protocol: 'file',
        slashes: true
    }));

    return win;
}

function addDocumentWindow() {
    addDocWin = addWindow('doc', {width: 200, height: 300,
        title: 'Add Document'});

    // Garbage collection
    addDocWin.on('close', () => {
        addDocWin = null;
    });
}

// Application ready event
app.on('ready', () => {
    // Create the new window
    main = addWindow('main', {});
    displayMenu();

    // Terminating the application
    main.on('closed', () => {
        app.quit();
    });
});
```

Summary

Throughout this chapter, we’ve explored how to create the basic structure that our Electron application requires. We are now ready to start with the actual functionality of the application. We have everything we need from a functional point of view, so we can focus exclusively on our application’s logic—this is what we’ll do in the next chapter.

Chapter 3 Expanding the App

Quick intro

With the basics covered, let’s now shift our attention and add specific application logic—we’ll see how we can combine that with the code we have written so far.

To correctly exchange information between windows of the same Electron application, we’ll need to use an ipcRenderer (inter-process communication), which basically sends an event with a payload from one window to another—which we can then catch with ipcMain.

You can think of **`ipcRenderer`** and **`ipcMain`** like WebSockets for Electron. We’ll be using them when adding or updating a new document and reflecting those changes on the main window.

The core logic

In the previous chapter, we created the basic template for the doc.html file with a couple of fields, but we didn’t add any logic.

Before we add any code, let’s have a look how the flow of information will look between the various parts of application, using **`ipcRenderer`** and **`ipcMain`**.

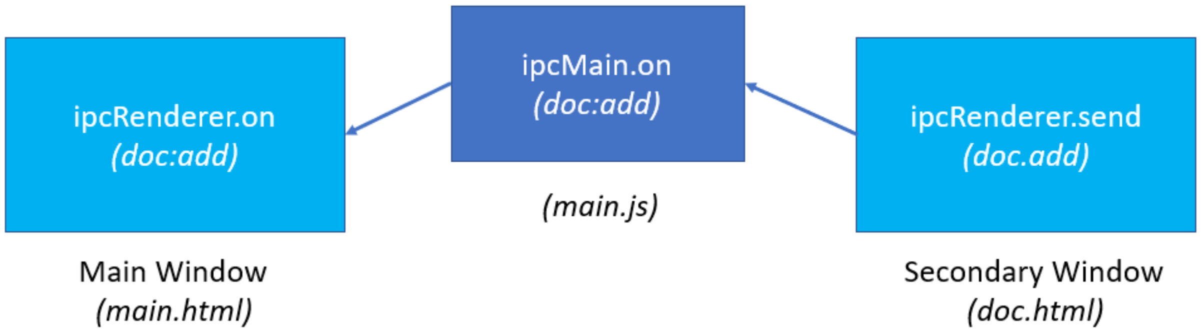


Figure 3-a: The Flow of Information Between Windows with ipcMain and ipcRenderer

What will happen is that the **`ipcRenderer.send`** method on the secondary window (doc.html) will send a message with the document added, and this event will be captured by the **`ipcMain.on`** event in main.js. Then, within **`ipcMain.on`**, the document is added and its payload is sent to the main window (main.html), and is captured by the **`ipcRenderer.on`** event present there.

So, as you can see, the **`ipcRenderer`** and **`ipcMain`** modules serve as an internal messaging transfer system between the various windows on an Electron application.

With the flow of information clear, we can now add the necessary logic to doc.html; we can add a new document and send the added data back to the main window (main.html).

With Visual Studio Code, open doc.html and add a **`<script>…</script>`** section, just before the **`</form>`** tag, so we can write some vanilla JavaScript to get the elements from the DOM, and send them using the **`ipcRenderer.send`** method.

Listing 3-a: The Initial Logic for doc.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport"

        content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Document</title>

</head>

<body>

    <form>

        <div>

            <label>Document name:</label>

            <input type="text" id="name" autofocus>

            <label>Expiration date:</label>

            <input type="text" id="date">

        </div>

        <button type="submit">Save</button>

    </form>
    <script>
        const electron = require('electron');
        const {ipcRenderer} = electron;

        const form = document.querySelector('form');
        form.addEventListener('submit', docSubmit);

        function docSubmit(e) {
            e.preventDefault();
            const name = document.querySelector('#name').value;
            const date = document.querySelector('#date').value;
            ipcRenderer.send('doc:add', name + '|' + date);
        }
    </script>
</body>
</html>
```

On the first two lines, we reference and declare the essential modules and variables we need, such as **`electron`** and **`ipcRenderer`**.

Then, on the next two lines, we select the form using the standard JavaScript **`querySelector`** method, and add an event listener to it when the form is submitted.

The **`docSubmit`** method is executed when the form's **`submit`** method is triggered. Within the **`docSubmit`** method, the first thing we do is to prevent the default form action—this is why the **`preventDefault`** method is called.

We then get the values of the **`name`** and **`date`** fields by using the **`querySelector`** method on each field. Finally, we send those values to main.js using the **`ipcRenderer.send`** method.

Retransmitting the response: main.js

Now that we've added some logic to doc.html and we've managed to send across the information captured from the form, we need to be able to retransmit it to main.html—and for that, we need to capture the message and payload on main.js.

We can do this with the following code, which we can place just before the **`ready`** event.

Listing 3-b: Retransmitting the Response (main.js)

```
ipcMain.on('doc:add', (e, item) => {

    main.webContents.send('doc:add', item);

    addDocWin.close();

});
```

Notice that instead of using **`ipcRenderer`**, we now use **`ipcMain.on`** and listen for the **`doc:add`** event. When that occurs, an arrow function with two parameters gets executed—the second parameter being the document (**`item`**) that was transmitted.

That **`item`** gets then retransmitted to the main window (main.html), which is done by invoking the **`main.webContents.send`** method. Once the **`item`** has been sent, the secondary window (**`addDocWin`**) is closed.

Let's now add the logic to main.html to receive the message and payload retransmitted.

Receiving the response: main.html

Open the main.html file and add a **`<script>…</script>`** section, just before the **`</body>`** tag—this is where we will add our code. The main.html file now looks as follows.

Listing 3-c: Receiving the Response (main.html)

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">
    <meta name="viewport"

        content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>DocExpire</title>

</head>

<body>

    <h1>DocExpire</h1>

    <ul></ul>

    <script>

        const electron = require('electron');

        const {ipcRenderer} = electron;


        const ul = document.querySelector('ul');


        ipcRenderer.on('doc:add', (e, item) => {

            const doc = document.createElement('li');

            const docText = document.createTextNode(item);


            doc.appendChild(docText);

            ul.appendChild(doc);

        });

    </script>

</body>

</html>
```

Just like with the doc.html code, on the first two lines we reference and declare the essential modules and variables we need, such as **`electron`** and **`ipcRenderer`**.

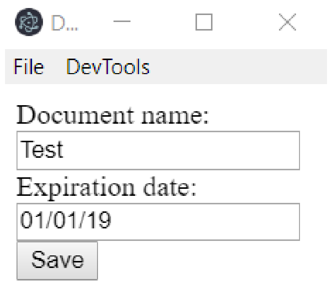
Then, we reference the **`ul`** HTML element by using the **`querySelector`** method. Notice that on the HTML markup, we've added a **`ul`** element, which represents an unordered list of elements—this will contain the document names added through the logic present within doc.html.

After referencing the **`ul`** element, we call **`ipcRenderer.on`** and listen to the **`doc:add`** event.

Once the **`doc:add`** event gets triggered, we create a **`li`** element node and assign to it the **`name`** and **`date`** of the document that was created on the secondary window.

With the command line opened, let's run **`npm start`** and test the application. When the application is opened, click on the **File** menu, and then on the **Add Document** option.

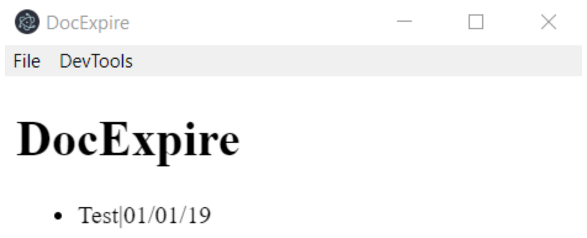
Then, with the secondary window opened, add the following information, and click **Save**.



Document name:
Test
Expiration date:
01/01/19

Figure 3-b: Adding a Test Document

The the secondary window will be automatically closed, and on the main window (main.html), we should see the document added to the list. Let's see if that's the case.



DocExpire

- Test|01/01/19

Figure 3-c: The Test Document Added

The document has been successfully added to the list. This means that the **`ipcMain`** and **`ipcRenderer`** logic has worked seamlessly.

If you would like to test further, feel free to add some extra sample documents, which should also be added to the list on the main window.

Clearing the document list

Now that we have added sample documents to the list, it's nice that we have an option to clear that list. To do that, let's switch back to the main.js file and add some code to the menu template for the Clear Documents option.

Listing 3-d: The Clear Documents Option (main.js)

```
const mainMenu = [
    // … Existing code …
    {
        label: 'Clear Documents',
        click() {
            main.webContents.send('doc:clear');
        }
    },
    // … Existing code …
];
```

Notice that all we've done is to add a click event to the Clear Documents option. Inside the event, we execute the **`main.webContents.send`** method and specify **`doc:clear`** as a parameter, which represents the action.

On the main.html file, let's add the **`doc:clear`** code that will be responsible for clearing the document list—we can do this as follows.

Listing 3-e: The Clear Documents Code (main.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>DocExpire</title>
</head>
<body>
    <h1>DocExpire</h1>
    <ul></ul>
    <script>
        // … Existing code …

        ipcRenderer.on('doc:clear', () => {
            ul.innerHTML = '';
        });
    </script>
</body>
</html>
```

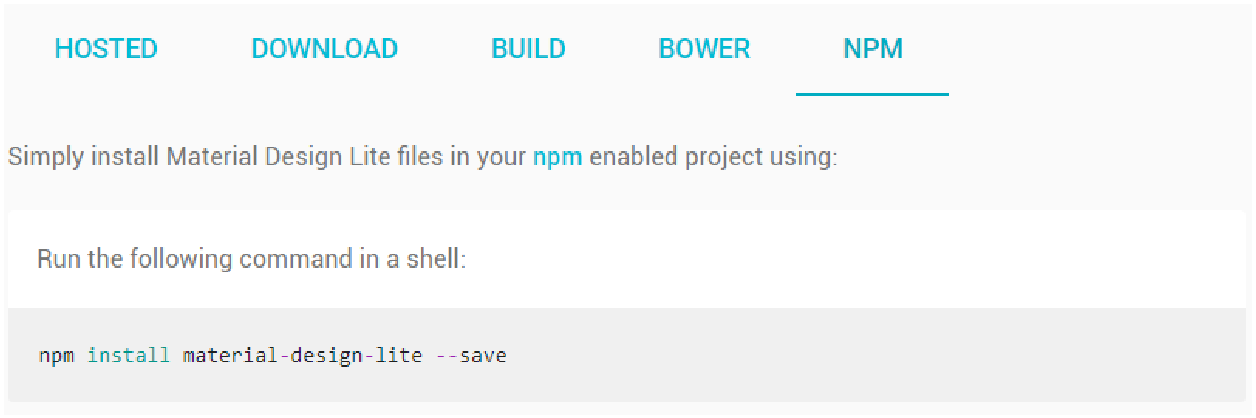
All we need to do to clear the list of documents is to set the **`innerHTML`** property of the **`ul`** element to an empty string.

If you close and run again the application by executing **`npm start`** from the command line, you should be able to check if this works by adding a few items to the list, and then clicking the **Clear Documents** option.

Add styling resources

So far, everything that we've managed to do works, but it's not particularly appealing to the eye. Let's change that by adding a bit of style—we can do that by using the Material Design Lite CCS framework, which is based on Google's Material Design.

So, go to the Material Design Lite website, click on the **Getting Started** option, then click on **NPM**.



Simply install Material Design Lite files in your npm enabled project using:

Run the following command in a shell:

```
npm install material-design-lite --save
```

Figure 3-d: The Material Design Lite NPM Option

Then, open the command line and enter the command indicated on the Material Design Lite website.

Listing 3-f: Install Material Design Lite via NPM

```
npm install material-design-lite --save
```

The reason we have installed the Material Design Lite library files rather than using the hosted version, is that we are building a desktop application, and the app's resources should reside locally, which is not necessarily the case when building a web app.

So, let's add the reference to the Material Design Lite library. We can do this as follows, just after the **`<title>…</title>`** tags—you'll see this highlighted in **bold** in the following code.

Listing 3-g: Adding the (Partially Local) Material Design Library (main.html)

```
<!DOCTYPE html>

<html lang="en">

<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>DocExpire</title>
    <link rel="stylesheet"
      href="./node_modules/material-design-lite/material.min.css">
    <script
      src="./node_modules/material-design-lite/material.min.js"></script>
    <link rel="stylesheet"
      href="https://fonts.googleapis.com/icon?family=Material+Icons">
</head>
```

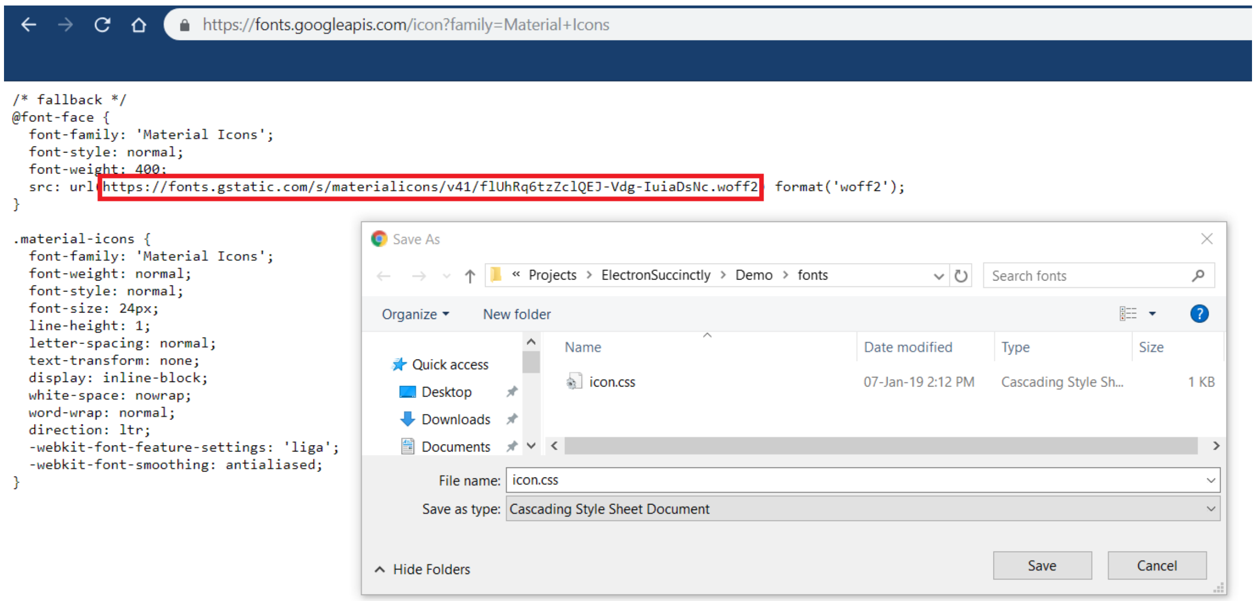
Notice how we are referencing the local Material Design Lite CSS (**`material.min.css`**) and JavaScript (**`material.min.js`**) files that were installed using NPM. However, there is still a reference to hosted Google Fonts, which is a requirement for Material Design Lite.

As we are building a desktop application, I suggest that we have all our resources locally, rather than having to depend on a hosted resource. So, to do that, open your browser, paste in the following URL, and then press **Enter**.

Listing 3-h: The Google Fonts URL

```
https://fonts.googleapis.com/icon?family=Material+Icons
```

This will display the icons.css file in our browser, which we can then download to a local project folder.



https://fonts.googleapis.com/icon?family=Material+Icons

```
/* fallback */
@font-face {
  font-family: 'Material Icons';
  font-style: normal;
  font-weight: 400;
  src: url(https://fonts.gstatic.com/s/materialicons/v41/flUhRq6tzZclQEJ-Vdg-IuiaDsNc.woff2) format('woff2');
}

.material-icons {
  font-family: 'Material Icons';
  font-weight: normal;
  font-style: normal;
  font-size: 24px;
  line-height: 1;
  letter-spacing: normal;
  text-transform: none;
  display: inline-block;
  white-space: nowrap;
  word-wrap: normal;
  direction: ltr;
  -webkit-font-feature-settings: 'liga';
  -webkit-font-smoothing: antialiased;
}
```

Save As

« Projects › ElectronSuccinctly › Demo › fonts

Name	Date modified	Type	Size
icon.css	07-Jan-19 2:12 PM	Cascading Style Sh...	1 KB

File name: icon.css

Save as type: Cascading Style Sheet Document

Figure 3-e: Saving the Google Fonts Icon File Locally

Next, copy the URL of the font style itself—which I've outlined in red in Figure 3-e, so we can also save a copy of the font locally, to the same folder where the icon.css file will be saved—in my case, to the **fonts** folder, under the project's main folder.

You can save this file with the default name given by the browser—which in my case is: **flUhRq6tzZclQEJ-Vdg-IuiaDsNc.woff2**. Please note that the **woff** extension stands for Web Open Font Format.

Let's now update the icon.css file in our **fonts** folder, to use the local copy of **flUhRq6tzZclQEJ-Vdg-IuiaDsNc.woff2** instead of the hosted one—highlighted in bold in the following code listing.

Our local icon.css file should now look as follows.

Listing 3-i: The Local icon.css File with Updated (Local) Fonts

```
/* fallback */

@font-face {

  font-family: 'Material Icons';

  font-style: normal;

  font-weight: 400;

  src: url(flUhRq6tzZclQEJ-Vdg-IuiaDsNc.woff2) format('woff2');
}


.material-icons {
  font-family: 'Material Icons';
  font-weight: normal;
  font-style: normal;
  font-size: 24px;
  line-height: 1;
  letter-spacing: normal;
  text-transform: none;
  display: inline-block;
  white-space: nowrap;
  word-wrap: normal;
  direction: ltr;
  -webkit-font-feature-settings: 'liga';
  -webkit-font-smoothing: antialiased;
}
```

All we need to do now is to replace the hosted reference to the Google Fonts resource on our main.html file, with a local reference to icon.css instead—this is highlighted in bold in the following code.

Listing 3-j: Adding the (Fully Local) Material Design Library (main.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <!-- Previous code goes here -->

    <link rel="stylesheet" href="./fonts/icon.css">

</head>
```

Awesome—we now have almost all the styling we need locally on our project folder, and we do not need to rely on any hosted resources.

There's one more CSS file that I'd like to use in our application (found here), which we can also download to our local **fonts** folder.

Styling the app

Now that we have the styling resources that we need in place, let's add a bit of style to our application.

First, let's add the styles.css to our main.html file that we just downloaded to our **fonts** folder—I've highlighted this in bold in the following code.

Listing 3-k: Adding styles.css (main.html)

```
<!DOCTYPE html>

<html lang="en">

<head>

    <!-- Previous code goes here -->

    <link rel="stylesheet" href="./fonts/styles.css">

</head>
```

Next, also within main.html, let's replace the code corresponding to the **`<h1>…</h1>`** tags with the following code.

Listing 3-l: The Main App Bar (main.html)

```
<header class="demo-header mdl-layout__header mdl-layout__header--scroll
```

```
    mdl-color--grey-100 mdl-color-text--grey-800">

    <div class="mdl-layout__header-row">

        <span class="mdl-layout-title">Documents to Expire</span>

        <div class="mdl-layout-spacer"></div>

        <div class="mdl-textfield mdl-js-textfield

            mdl-textfield--expandable">

            <label class="mdl-button mdl-js-button

                mdl-button--icon" for="search">

                <i class="material-icons">search</i>

            </label>

            <div class="mdl-textfield__expandable-holder">

                <input class="mdl-textfield__input"

                    type="text" id="search">

                <label class="mdl-textfield__label" for="search">

                    Enter your query...

                </label>

            </div>

        </div>

    </div>

</header>

<div class="demo-ribbon"></div>

<main class="demo-main mdl-layout__content">

    <div class="demo-container mdl-grid">

    </div>
```

```
</main>
```

What we are doing here is adding some Material Design classes that will allow us to have a top application bar with a search option, and a content section, which we will use to display results.

Now, let’s wrap this code inside the following **`<div>`**.

Listing 3-m: The Main App Bar Wrapped Around a Styled <div> (main.html)

```
<div class="demo-layout mdl-layout mdl-layout--fixed-header
    mdl-js-layout mdl-color--grey-100">
    <!-- Previous code goes here -->
</div>
```

Our updated main.html file should now look as follows.

Listing 3-n: The Updated main.html File

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>DocExpire</title>
    <link rel="stylesheet"
        href="./node_modules/material-design-lite/material.min.css">
    <script
```

```
        src="./node_modules/material-design-lite/material.min.js">
    </script>
    <link rel="stylesheet" href="./fonts/icon.css">
    <link rel="stylesheet" href="./fonts/styles.css">
</head>
<body>
    <div class="demo-layout mdl-layout mdl-layout--fixed-header
        mdl-js-layout mdl-color--grey-100">
        <header class="demo-header mdl-layout__header
            mdl-layout__header--scroll
            mdl-color--grey-100 mdl-color-text--grey-800">
          <div class="mdl-layout__header-row">
            <span class="mdl-layout-title">Documents to Expire</span>
            <div class="mdl-layout-spacer"></div>
            <div class="mdl-textfield mdl-js-textfield
                mdl-textfield--expandable">
              <label class="mdl-button mdl-js-button
                  mdl-button--icon" for="search">
                <i class="material-icons">search</i>
              </label>
              <div class="mdl-textfield__expandable-holder">
                <input class="mdl-textfield__input"
                   type="text" id="search">
                <label class="mdl-textfield__label" for="search">
```

```
               Enter your query...
            </label>
          </div>
        </div>
      </div>
    </header>
    <div class="demo-ribbon"></div>
    <main class="demo-main mdl-layout__content">
        <div class="demo-container mdl-grid">
        </div>
    </main>
    <ul></ul>
</div>
<script>
    const electron = require('electron');
    const {ipcRenderer} = electron;

    const ul = document.querySelector('ul');

    ipcRenderer.on('doc:add', (e, item) => {
        const doc = document.createElement('li');
        const docText = document.createTextNode(item);

        doc.appendChild(docText);
```

```
            ul.appendChild(doc);
        });


        ipcRenderer.on('doc:clear', () => {
            ul.innerHTML = '';
        });
    </script>
</body>
</html>
```

If we now run the **`npm start`** command through the command line, we should see the following changes to our app’s user interface.

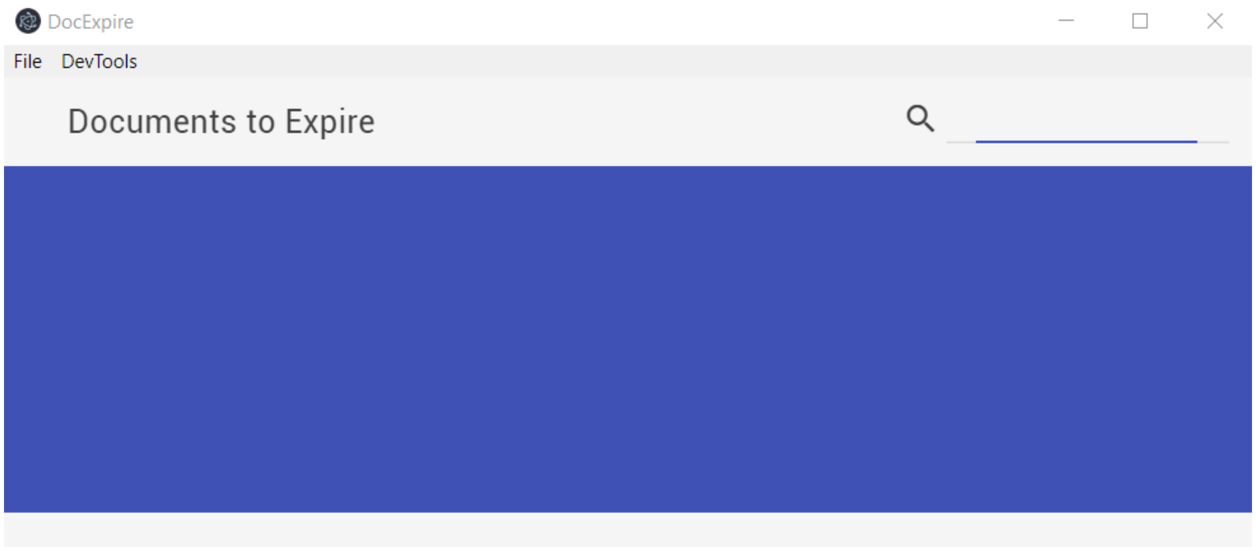


Figure 3-f: Our App’s Cool Looking User Interface, using Material Design Lite

How cool is that? By sprinkling a bit of Material Designed-flavored CSS and some HTML marku, we’ve taken a bare bones and not-so-great-looking application into something that looks much nicer.

Styling the list

Although our app has a bit of style now, we are far from done. If we try now to add a document by clicking on the **Add Document** option from the **File** menu—we'll end up with something that looks like this.

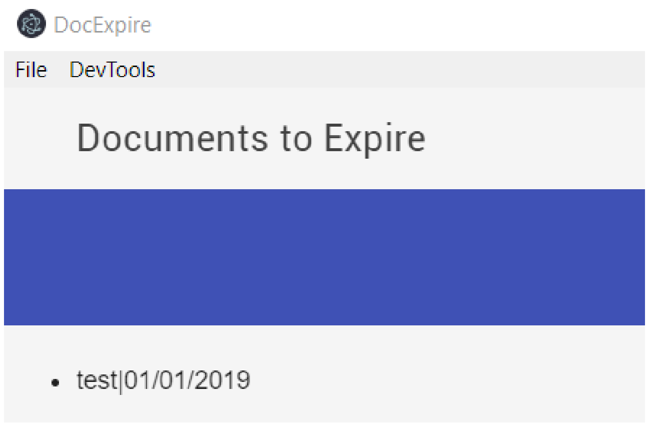


Figure 3-g: A New Document Added to the UI

As you can see, the item added doesn't seem to fit with the new user interface's look and feel—let's fix that.

First, let's add a quick CSS fix for centering HTML elements. We can add this to the **`<header>`** section of the HTML markup as follows.

Listing 3-o: Adding a Style for Centering HTML Elements

```
<style>
    .mdl-grid.center-items {
       justify-content: center;
    }
</style>
```

All we are doing is adding a CSS class that we can use to center HTML content.

Next, let's adjust the HTML content between the **`<body>…<script>`** tags as follows.

Listing 3-p: Adjusting the Main Markup

```
<div class="demo-layout mdl-layout mdl-layout--fixed-header

    mdl-js-layout mdl-color--grey-100">

    <header class="demo-header mdl-layout__header

        mdl-layout__header--scroll mdl-color--grey-100

        mdl-color-text--grey-800">

        <div class="mdl-layout__header-row">

            <span class="mdl-layout-title">Documents to Expire</span>

            <div class="mdl-layout-spacer"></div>

            <div class="mdl-textfield mdl-js-textfield

                mdl-textfield--expandable">

                <label class="mdl-button mdl-js-button

                    mdl-button--icon" for="search">

                    <i class="material-icons">search</i>

                </label>

                <div class="mdl-textfield__expandable-holder">

                    <input class="mdl-textfield__input" type="text"

                        id="search" onkeyup="searchItems()">

                    <label class="mdl-textfield__label" for="search">

                        Enter your query...

                    </label>

                </div>

            </div>

        </div>
    </header>

    <div class="demo-ribbon"></div>

    <main class="demo-main mdl-layout__content">

        <div class="demo-container mdl-grid">

            <div class="mdl-cell mdl-cell--12-col

                mdl-cell--hide-tablet mdl-cell--hide-phone">

            </div>

            <div class="demo-content mdl-color--white

                mdl-shadow--4dp content mdl-color-text--grey-800

                mdl-cell mdl-cell--12-col">

                <div class="mdl-grid center-items">

                    <div class="mdl-cell mdl-cell--12-col">

                        <ul id="ulist" class="demo-list-icon mdl-list">

                        </ul>

                    </div>

                </div>

            </div>

        </div>

    </main>

</div>
```

Notice that the **`header`** section is basically the same as it was, with the exception of a **`onkeyup`** event on the **`search`** field, which is now hooked up to a **`searchItems`** function that we'll create later.

Next, notice the main middle section **`<main>..</main>`** that is basically a placeholder for the list of documents—all nicely enhanced with Material Design Lite styles and classes.

I've given the `<ul>` element an `id`, which will make it easier to reference it in the code that will follow, within the `<ul>..</ul>` tags, which is where our documents will be displayed, after being created.

For further details about Material Design Lite components and styles, and how to use them, please refer to the official documentation here.

Before we jump into the code that explains how we are going to dynamically add each document between the `<ul>…</ul>` tags, let me show you how the finished main window UI will look.

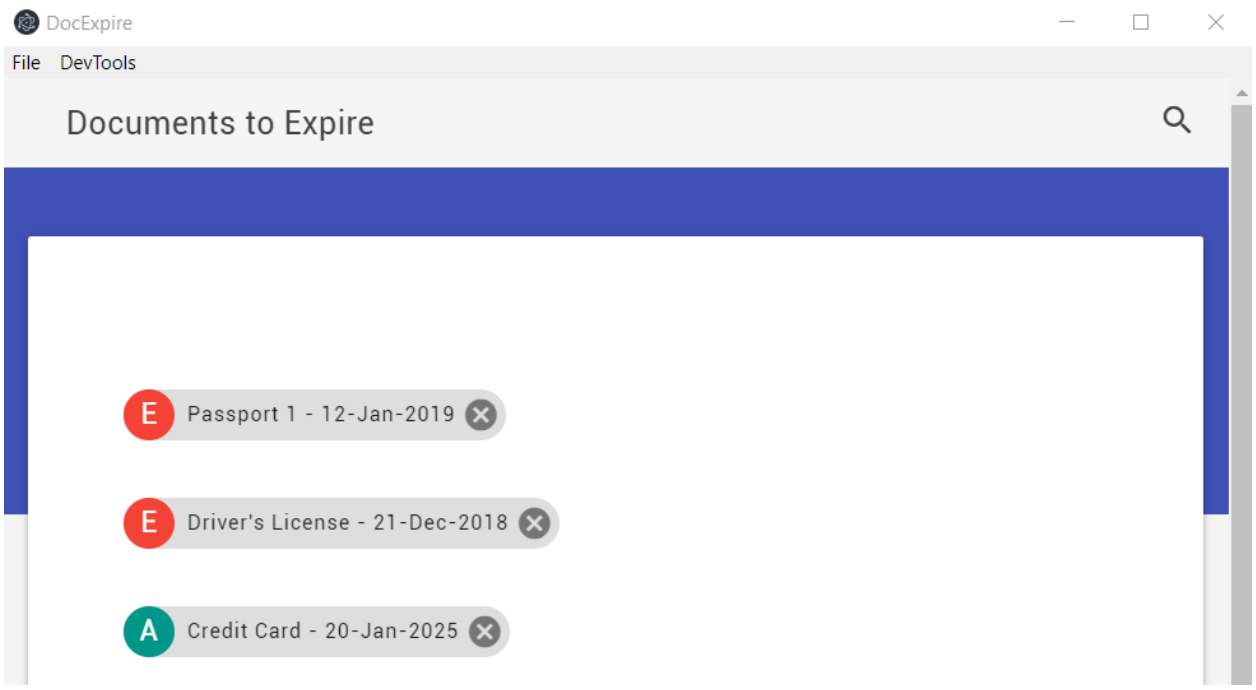


Figure 3-h: The Finished Main Window UI

Notice that all the HTML markup previously described is used to render the main content displayed in the main window's UI. However, we haven't yet written any markup or code that allows us to display the three elements shown within the white area seen in Figure 3-h.

The markup for those three elements are going to be dynamically rendered using vanilla JavaScript code—without any specific framework.

Statically rendered chip components

The idea behind any of these elements—which are inspired by Material Design Lite chip components—is that each one represents an important document that has either expired or will expire in the future, which is the whole purpose of the application.

Notice how items that have an expired date are displayed with a red **E**, meaning *expired*; and items that have the green **A**, meaning *active* (unless you are reading this book on or after January 20, 2025).

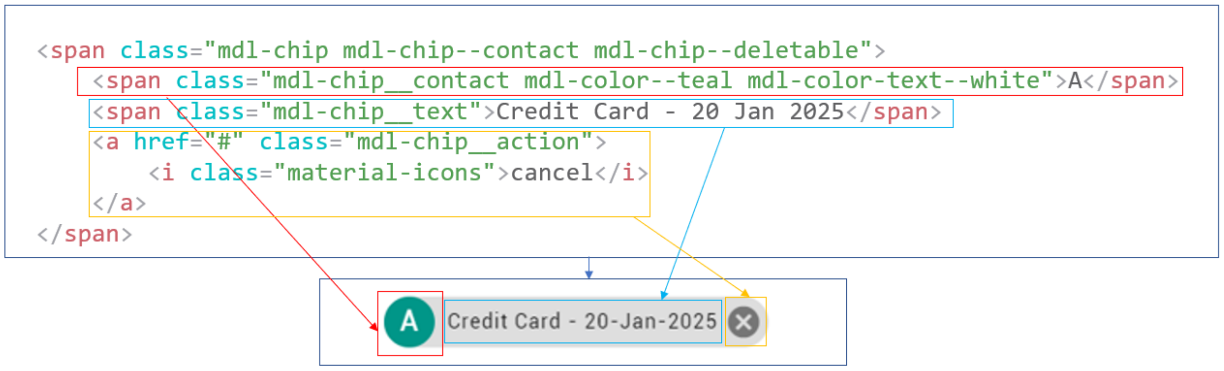
Before we can understand how to render these elements dynamically using plain JavaScript, let's show how we would render one of them using normal HTML markup.

Listing 3-q: Chips HTML Markup

```
<span class="mdl-chip mdl-chip--contact mdl-chip--deletable">
    <span class="mdl-chip__contact mdl-color--teal
        mdl-color-text--white">A
    </span>
    <span class="mdl-chip__text">Passport 1 - 20 Jan 2025</span>
    <a href="#" class="mdl-chip__action">
        <i class="material-icons">cancel</i>
    </a>
</span>
```

We can see that the three elements that make up the document are contained within the **`<span>…</span>`** tags, and use mainly the **`mdl-chip`** CSS class from the Material Design Lite library.

The following diagram shows how this item is formed and which HTML elements compose each of its individual parts.



```
<span class="mdl-chip mdl-chip--contact mdl-chip--deletable">
   <span class="mdl-chip__contact mdl-color--teal mdl-color-text--white">A</span>
   <span class="mdl-chip__text">Credit Card - 20 Jan 2025</span>
   <a href="#" class="mdl-chip__action">
      <i class="material-icons">cancel</i>
   </a>
</span>
```

Figure 3-i: The Chip Element HTML Parts

Dynamically rendered chip components

Now, let’s now have a look at the main.html JavaScript code that is responsible for rendering these elements dynamically—we’ll dissect this into smaller parts, so we can understand each step involved.

Listing 3-r: Full main.html Code

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

const {ipcRenderer} = electron;


const ul = document.querySelector('ul');


function createItemDom(item) {

    let iparts = item.split('|');

    let id = iparts[0].replace(' ', '_').replace('/', '-');


    const doc = document.createElement('li');

    doc.id = 'li_' + id;

    doc.className = 'mdl-list__item';


    const docText = document.createElement('span');

    docText.className = 'mdl-chip mdl-chip--contact mdl-chip--deletable';


    let exp = checkExpDate(iparts[1]);

    const sp1 = document.createElement('span');

    sp1.className = (exp == 'A') ?

        'mdl-chip__contact mdl-color--teal mdl-color-text--white' :
```

```
    'mdl-chip__contact mdl-color--red mdl-color-text--white';
const sp1Txt = document.createTextNode(exp);

const sp2 = document.createElement('span');
sp2.className = 'mdl-chip__text';
const sp2Txt = document.createTextNode(iparts[0] +
    ' - ' + iparts[1]);

const ar = document.createElement('a');
ar.className = 'mdl-chip__action';
ar.id = 'a_' + id;

const ir = document.createElement('i');
ir.className = 'material-icons';
const irTxt = document.createTextNode('cancel');

ir.appendChild(irTxt);
ar.appendChild(ir);

sp1.appendChild(sp1Txt);
docText.appendChild(sp1);

sp2.appendChild(sp2Txt);
docText.appendChild(sp2);
```

```
    docText.appendChild(ar);

    doc.appendChild(docText);
    ul.appendChild(doc);

    document.getElementById(ar.id).addEventListener('click', deleteItem);
}

function deleteItem(e) {
    let doc = e.target.parentNode.parentNode.parentNode;
    doc.innerHTML = '';
    doc.parentNode.removeChild(doc);
}

function checkExpDate(dt) {
    let currentDate = Date.now();
    let date = Date.parse(dt);
    return (!isNaN(date) && currentDate - date < 0) ? 'A' : 'E';
}

function searchItems() {
    let input, filter, ul, li, a, i, txtValue;
    input = document.getElementById('search');
    filter = input.value.toUpperCase();
    ul = document.getElementById('ulist');
    li = ul.getElementsByTagName('li');

    for (i = 0; i < li.length; i ++) {
        a = li[i].getElementsByTagName('a')[0];
        txtValue = (a.textContent || a.innerText) &&
            (a.parentNode.textContent ||
             a.parentNode.innerText).replace('cancel', '');
        if (txtValue.toUpperCase().indexOf(filter) > -1) {
            li[i].style.display = "";
        } else {
            li[i].style.display = "none";
        }
    }
}

ipcRenderer.on('doc:add', (e, item) => {
    createItemDom(item);
});

ipcRenderer.on('doc:clear', () => {
    ul.innerHTML = '';
});
```

To get a better understanding, let's look at what each part of this code does—we'll start with the main constants.

Listing 3-s: Part 1 - main.html – Declaration Code

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

const {ipcRenderer} = electron;



const ul = document.querySelector('ul');
```

This first part of the code simply declares the essential required modules and variables: **`electron`**, **`ipcRenderer`**, and, **`ul`**, which refers to the list where the documents will be rendered.

Listing 3-t: Part 2 – main.html – Full createItemDom Code

```
function createItemDom(item) {

    let iparts = item.split('|');

    let id = iparts[0].replace(' ', '_').replace('/', '-');


    const doc = document.createElement('li');

    doc.id = 'li_' + id;

    doc.className = 'mdl-list__item';


    const docText = document.createElement('span');

    docText.className = 'mdl-chip mdl-chip--contact mdl-chip--deletable';


    let exp = checkExpDate(iparts[1]);

    const sp1 = document.createElement('span');
```

```
sp1.className = (exp == 'A') ?
    'mdl-chip__contact mdl-color--teal mdl-color-text--white' :
    'mdl-chip__contact mdl-color--red mdl-color-text--white';
const sp1Txt = document.createTextNode(exp);

const sp2 = document.createElement('span');
sp2.className = 'mdl-chip__text';
const sp2Txt = document.createTextNode(iparts[0] +
    ' - ' + iparts[1]);

const ar = document.createElement('a');
ar.className = 'mdl-chip__action';
ar.id = 'a_' + id;

const ir = document.createElement('i');
ir.className = 'material-icons';
const irTxt = document.createTextNode('cancel');

ir.appendChild(irTxt);
ar.appendChild(ir);

sp1.appendChild(sp1Txt);
docText.appendChild(sp1);
```

```
    sp2.appendChild(sp2Txt);

    docText.appendChild(sp2);


    docText.appendChild(ar);


    doc.appendChild(docText);

    ul.appendChild(doc);


    document.getElementById(ar.id).addEventListener('click', deleteItem);

}
```

The **`createItemDom`** function is what does the magic of dynamically rendering the “expiring” or “about to expire” documents.

Listing 3-u: Part 3 – main.html – Partial createItemDom Code

```
let iparts = item.split('|');

let id = iparts[0].replace(' ', '_').replace('/', '-');
```

The first line basically takes the **`item`** parameter—which is literally the data that was saved from the Add Document window, the name of the document, and the expiration date of the document—concatenated with a pipe (|) character and split it into two parts.

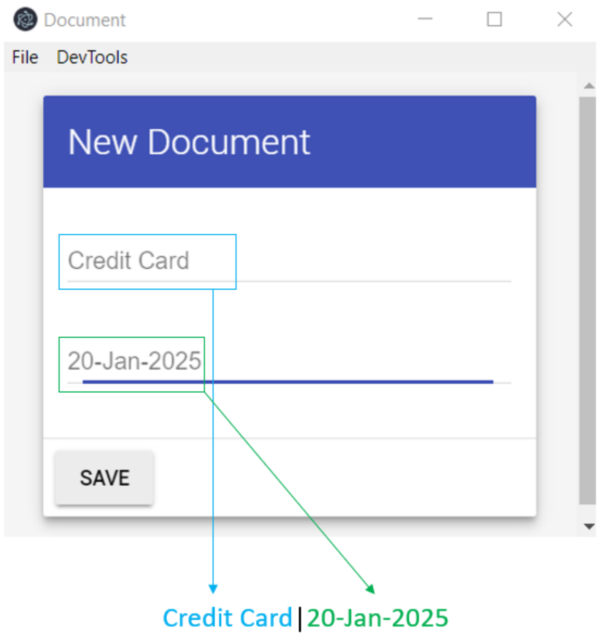


Figure 3-j: The Concatenated Document Data

The second line basically takes the **name** of the document and replaces any spaces ( ) and slashes (/) found with underscores (_) and dashes (-), in a very basic form of data validation. This will be used to create the **id** of the HTML element, which is dynamically created from the details received from the Add Document window.

Listing 3-v: Part 4 – main.html – Partial createItemDom Code

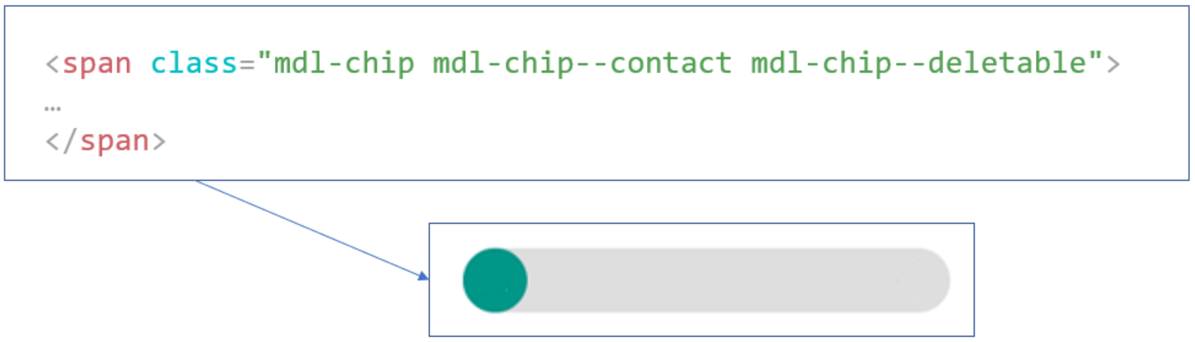
```
const doc = document.createElement('li');

doc.id = 'li_' + id;

doc.className = 'mdl-list__item';
```

These lines basically create the **<li>…</li>** tags that wrap the chip elements described in *Figure 3-i*. In HTML, each **<li>** represents an element of a **<ul>**.

Listing 3-w: Part 5 – main.html – Partial createItemDom Code

```
const docText = document.createElement('span');

docText.className = 'mdl-chip mdl-chip--contact mdl-chip--deletable';
```

In the two previous lines, we create the main **`<span>`** that wraps the element. We can see the main **`<span>`** created by those two lines of code, in the following screenshot.



```
<span class="mdl-chip mdl-chip--contact mdl-chip--deletable">
…
</span>
```

Figure 3-k: The Main <span> of the Chip Element

Let's continue analyzing the code. Next, we are going to create the **A**ctive or **E**xpired **`<span>`** of the chip element, as shown in the following code.

Listing 3-x: Part 6 – main.html – Partial createItemDom Code

```
let exp = checkExpDate(iparts[1]);
const sp1 = document.createElement('span');
sp1.className = (exp == 'A') ?
    'mdl-chip__contact mdl-color--teal mdl-color-text--white' :
    'mdl-chip__contact mdl-color--red mdl-color-text--white';
const sp1Txt = document.createTextNode(exp);
```

The first line executes the **`checkExpDate`** function, which checks if the date received from the Add Document window has expired or is still active, by comparing it with the current date. We'll come back to the implementation details of the **`checkExpDate`** function later—but in short, it returns an **`A`** for active and **`E`** for expired.

The **`document.createElement`** function creates the element on the DOM—in this case, the **`<span>`**.

Next, depending on value returned from the **`checkExpDate`** function, we assign the appropriate Material Design Lite CSS class, using a ternary operator conditional statement:

```
sp1.className = (exp == 'A') ?
```

```
            'mdl-chip__contact mdl-color--teal mdl-color-text--white' :
            'mdl-chip__contact mdl-color--red mdl-color-text--white';
```

For instance, if the value returned by the **`checkExpDate`** function is **`A`**, then the **`<span>`** would look as follows, and these CSS classes would be applied: **`mdl-chip__contact mdl-color--teal mdl-color-text--white`**.

Figure 3-l: The Active <span> Element

On the other hand, if the value returned by the **`checkExpDate`** function is **`E`**, then the **`<span>`** would look as follows, and these CSS classes would be applied: **`mdl-chip__contact mdl-color--red mdl-color-text--white`**.

Figure 3-m: The Expired <span> Element

Finally, the last line of code adds the text value returned by the **`checkExpDate`** function, either **`A`** or **`E`** to the **`<span>`**.

The next lines of code add the main text information to the chip element, which contains the name of the document and the expiration date of the document added through the Add Document window.

Listing 3-y: Part 7 – main.html – Partial createItemDom Code

```
const sp2 = document.createElement('span');
sp2.className = 'mdl-chip__text';
const sp2Txt = document.createTextNode(iparts[0] +
    ' - ' + iparts[1]);
```

As you can see, a **`<span>`** that will contain the text is created, the required CSS class is added, and finally, the text itself is concatenated: first, the name of the document (represented by **`iparts[0]`**) and then the expiration date (indicated by **`iparts[1]`**).

The next part of the code basically creates the Cancel button, which is going to be used to delete the document, via the **`<li>…</li>`** element from the **`<ul>…</ul>`** list of elements, as can be seen in the following figure.

Figure 3-n: The cancel Button

The code first creates a **`<a>`** element, and then adds an **`<i>`** child element under it.

Listing 3-z: Part 8 – main.html – Partial createItemDom Code

```
const ar = document.createElement('a');

ar.className = 'mdl-chip__action';

ar.id = 'a_' + id;


const ir = document.createElement('i');

ir.className = 'material-icons';

const irTxt = document.createTextNode('cancel');
```

The last part of the code for the **`createItemDom`** function is responsible for adding the created HTML elements to the DOM. This is shown in the comments in the following code.

Listing 3-aa: Part 9 – main.html – Partial createItemDom Code

```
// Adds the <i> element to its parent, the <a> element

ir.appendChild(irTxt);

ar.appendChild(ir);


// Adds the ‘A’ or ‘E’ <span> element to its parent, the main <span>

sp1.appendChild(sp1Txt);

docText.appendChild(sp1);


// Adds the Name of Expiry Date <span> to its parent, the main <span>
```

```
sp2.appendChild(sp2Txt);

docText.appendChild(sp2);


// Adds the <a> element to its parent, the main <span>

docText.appendChild(ar);

// Adds the main <span>, Chip Element to the <li> element

doc.appendChild(docText);


// Adds the <li> element to its parent, the <ul> list of elements

ul.appendChild(doc);


// Adds the event handler when for the cancel button for each element

document.getElementById(ar.id).addEventListener('click', deleteItem);
```

As you can see, the comment for each section is very descriptive and explains how each element is added to the DOM. The order in which it is done is very important.

Checking dates

A key aspect of being able to render these documents in our application's UI accordingly is being able to know if a document's date is still valid, or if it has expired. This is what the **`checkExpDate`** function does—let's have a look at its code.

Listing 3-ab: Part 10 - main.html – checkExpDate Function

```
function checkExpDate(dt) {

    let currentDate = Date.now();

    let date = Date.parse(dt);
    return (!isNaN(date) && currentDate - date < 0) ? 'A' : 'E';

}
```

The code is very simple. First, we first get the **`currentDate`** by calling the **`Date.now`** method. Then, we parse the value of the date supplied (**`dt`**) when the document was added through the **Add Document** window, using the **`Date.parse`** method

The final part is to check if the parsed **`date`** is a number (**`!isNaN`**)—and verifying that the **`currentDate`**, compared to the document's date (**`dt`**), is in the future—if so, the document is considered active—if not, it is considered expired.

Deleting documents

So far, we have created all the code required to render documents on the main window's UI (main.html) once those documents have been added through the Add Document window.

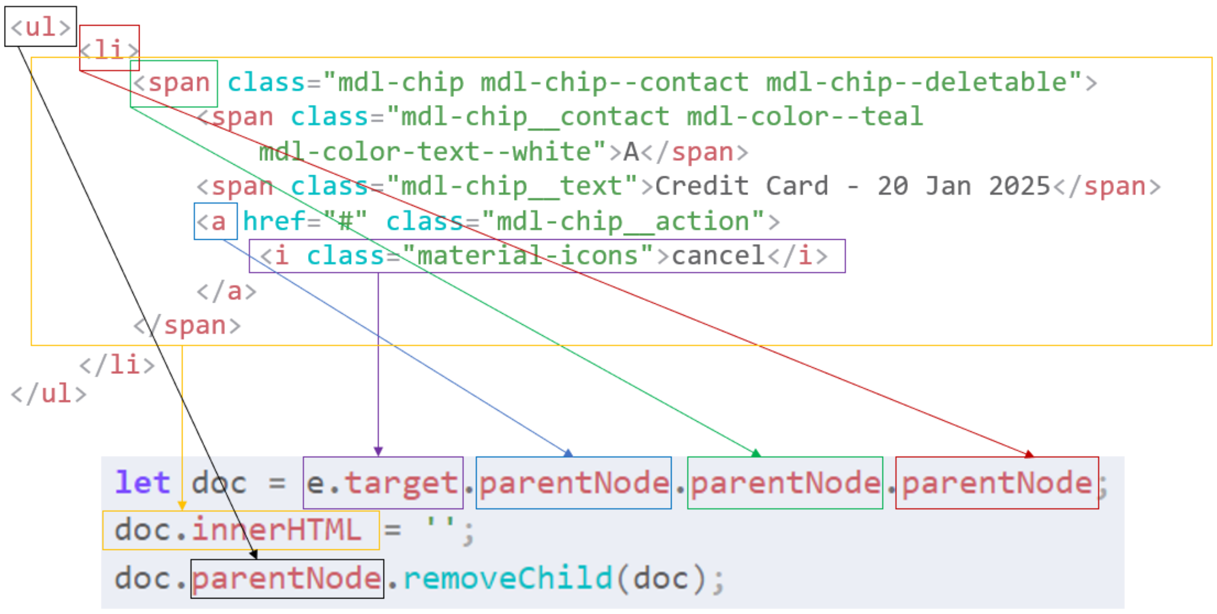
However, we still haven't looked at the code for the **`deleteItem`** method that was added as an event handler to the Cancel button of a document. Let's have a look at the code.

Listing 3-ac: Part 11 - main.html – deleteItem Function

```
function deleteItem(e) {

    let doc = e.target.parentNode.parentNode.parentNode;

    doc.innerHTML = '';

    doc.parentNode.removeChild(doc);

}
```

The method receives an **`e`** parameter, which refers to the DOM object that contains the Cancel button icon.

The following diagram shows the relationship between the lines of code of the **`deleteItem`** function and the actual HTML markup.



```
<ul>
  <li>
    <span class="mdl-chip mdl-chip--contact mdl-chip--deletable">
      <span class="mdl-chip__contact mdl-color--teal
          mdl-color-text--white">A</span>
      <span class="mdl-chip__text">Credit Card - 20 Jan 2025</span>
      <a href="#" class="mdl-chip__action">
        <i class="material-icons">cancel</i>
      </a>
    </span>
  </li>
</ul>
```

```
let doc = e.target.parentNode.parentNode.parentNode;
doc.innerHTML = '';
doc.parentNode.removeChild(doc);
```

Figure 3-o: The Relationship Between the HTML Markup and the deleteItem Code

As we can see, the **`e.target.parentNode.parentNode.parentNode`** object refers to the **`<li>`** item, whose content we delete by setting its **`innerHTML`** property to an empty string.

Then, we remove the **`<li>`** element from the list by referencing the **`<ul>`** object, using the **`doc.parentNode`** property and invoking the **`removeChild`** method.

Searching documents

The final part of the main.html code that we still haven't talked about is the search functionality—let's have a look at that now.

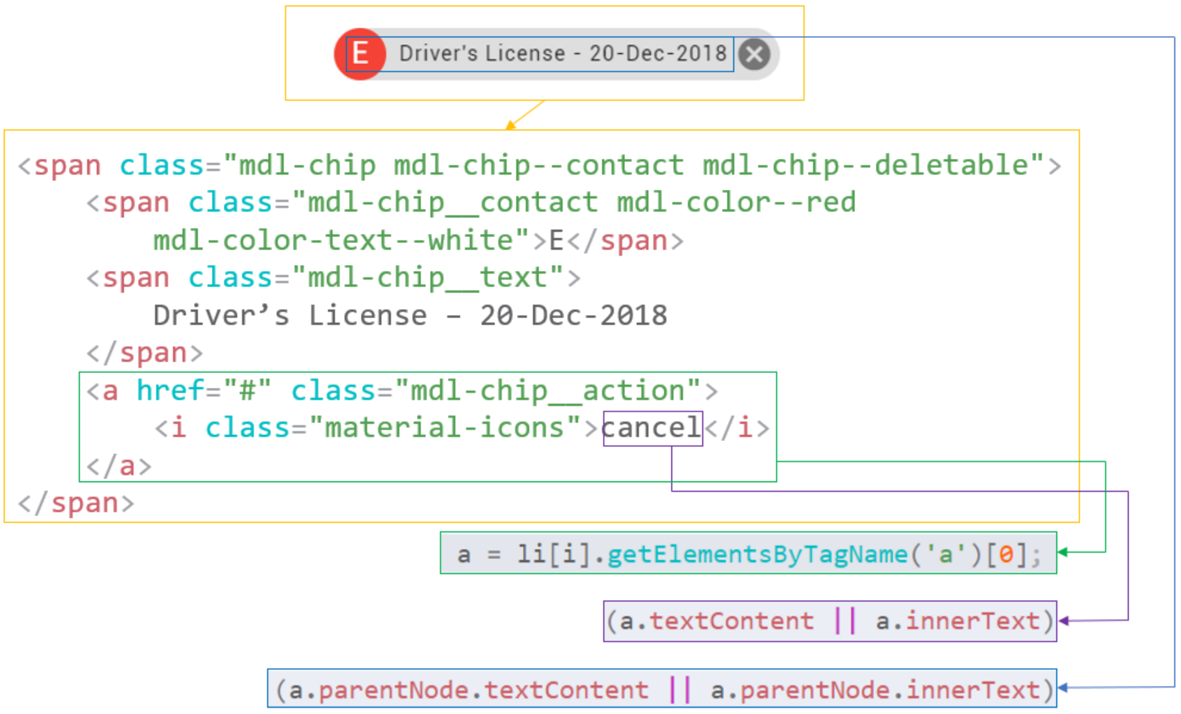
Listing 3-ad: Part 12 - main.html – searchItems Function

```
function searchItems() {

    let input, filter, ul, li, a, i, txtValue;

    input = document.getElementById('search');

    filter = input.value.toUpperCase();

    ul = document.getElementById('ulist');

    li = ul.getElementsByTagName('li');
    for (i = 0; i < li.length; i ++) {
        a = li[i].getElementsByTagName('a')[0];
        txtValue = (a.textContent || a.innerText) &&
            (a.parentNode.textContent ||
             a.parentNode.innerText).replace('cancel', '');
        if (txtValue.toUpperCase().indexOf(filter) > -1) {
            li[i].style.display = "";
        } else {
            li[i].style.display = "none";
        }
    }
}
```

What's going on here? First, we declare the variables we need by referencing key elements within the HTML markup, such as the **`search`** box and **`ulist`**, which refers to the list of **`<li>`** items contained within **`<ul>…</ul>`**.

Then, we loop through the list of all the **`<li>`** items and get the **`<a>`** child element of each **`<li>`** item—this is done by invoking **`li[i].getElementsByTagName('a')[0]`**.

To better understand how the code of the **`searchItems`** function works, let's have a look at the following diagram, which illustrates the parts of the code that relate to the HTML elements.



```
<span class="mdl-chip mdl-chip--contact mdl-chip--deletable">
    <span class="mdl-chip__contact mdl-color--red
        mdl-color-text--white">E</span>
    <span class="mdl-chip__text">
        Driver’s License - 20-Dec-2018
    </span>
    <a href="#" class="mdl-chip__action">
        <i class="material-icons">cancel</i>
    </a>
</span>
```

```
a = li[i].getElementsByTagName('a')[0];
(a.textContent || a.innerText)
(a.parentNode.textContent || a.parentNode.innerText)
```

Figure 3-p: The Relationship Between the HTML Markup and the searchItems Code

As we can see in Figure 3-p, we are able to retrieve the displayed text of the items we are looking for by evaluating `(a.textContent || a.innerText)` and `(a.parentNode.textContent || a.parentNode.innerText)`, and removing any references to the word **`cancel`** in the resultant string—which is assigned to the variable **`txtValue`**.

The uppercase value of the variable **`txtValue`** is then compared against the uppercase value entered through the **`search`** field. This is done so that any elements that don’t match the **`search`** value entered, are hidden from the **`<ul>`** list.

This is achieved by setting the **`style.display`** property of any non-matching **`<li>`** elements to **`none`**—which means any **`<li>`** element that doesn’t match the **`search`** query is therefore hidden, and not displayed.

So, all the elements of the list are always there—the **`searchItems`** function simply hides those that don’t match the **`search`** query.

Add Document window facelift

We now have all the code for our main window (main.html) ready. Let’s now make some visual improvements to the **Add Document** window, doc.html.

I going to place the finished code for the doc.html file in the following code listing, which we can then review in parts.

Listing 3-ae: Finished doc.html

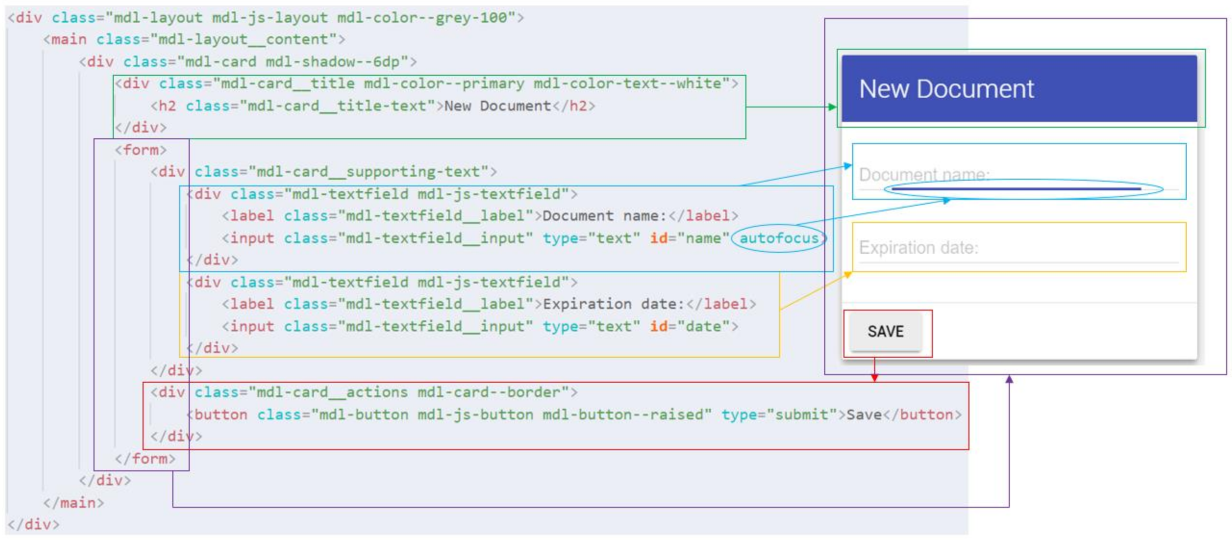
```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport"

        content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Document</title>

    <link rel="stylesheet"

        href="./node_modules/material-design-lite/material.min.css">

    <script

        src="./node_modules/material-design-lite/material.min.js">

    </script>

    <link rel="stylesheet" href="./fonts/icon.css">

    <link rel="stylesheet" href="./fonts/styles.css">


    <style>

        .mdl-layout {

            align-items: center;

            justify-content: center;

        }

        .mdl-layout__content {
            padding: 24px;

            flex: none;

        }

    </style>

</head>

<body>

    <div class="mdl-layout mdl-js-layout mdl-color--grey-100">

        <main class="mdl-layout__content">

            <div class="mdl-card mdl-shadow--6dp">

                <div class="mdl-card__title

                    mdl-color--primary mdl-color-text--white">

                    <h2 class="mdl-card__title-text">New Document</h2>

                </div>

                <form>

                    <div class="mdl-card__supporting-text">

                        <div class="mdl-textfield mdl-js-textfield">

                            <label class="mdl-textfield__label">

                                Document name:

                            </label>

                            <input

                                class="mdl-textfield__input"

                                type="text" id="name" autofocus>

                        </div>

                        <div class="mdl-textfield mdl-js-textfield">
                            <label
                                 class="mdl-textfield__label">
                                 Expiration date:
                            </label>
                            <input class="mdl-textfield__input"
                                type="text" id="date">
                        </div>
                    </div>
                    <div class="mdl-card__actions mdl-card--border">
                        <button class="mdl-button mdl-js-button
                            mdl-button--raised" type="submit">
                            Save
                        </button>
                    </div>
                </form>
            </div>
        </main>
    </div>
    <script>
        const electron = require('electron');
        const {ipcRenderer} = electron;

        const form = document.querySelector('form');
        form.addEventListener('submit', docSubmit);
```

```
        function docSubmit(e) {
            e.preventDefault();
            const name = document.querySelector('#name').value;
            const date = document.querySelector('#date').value;
            ipcRenderer.send('doc:add', name + '|' + date);
        }
    </script>
</body>
</html>
```

As you have probably noticed, the **`<header>…</header>`** section is pretty much the same as the one from main.html.

The interesting part of the code is actually just after the **`<body>`** tag, where I've added some Material Design Lite CSS styling to a few **`<div>`** sections that make up our document form.

The following diagram, which is easier to follow, illustrates the relationship between the HTML markup and the finished form on the screen.



```
<div class="mdl-layout mdl-js-layout mdl-color--grey-100">
    <main class="mdl-layout__content">
        <div class="mdl-card mdl-shadow--6dp">
            <div class="mdl-card__title mdl-color--primary mdl-color-text--white">
                <h2 class="mdl-card__title-text">New Document</h2>
            </div>
            <form>
                <div class="mdl-card__supporting-text">
                    <div class="mdl-textfield mdl-js-textfield">
                        <label class="mdl-textfield__label">Document name:</label>
                        <input class="mdl-textfield__input" type="text" id="name" autofocus
                    </div>
                    <div class="mdl-textfield mdl-js-textfield">
                        <label class="mdl-textfield__label">Expiration date:</label>
                        <input class="mdl-textfield__input" type="text" id="date">
                    </div>
                </div>
                <div class="mdl-card__actions mdl-card--border">
                    <button class="mdl-button mdl-js-button mdl-button--raised" type="submit">Save</button>
                </div>
            </form>
        </div>
    </main>
</div>
```

Figure 3-q: The Relationship Between the HTML Markup and the Finished Form

As you can see, each **`<div>`** correlates with a section of the finished form. For more information about the Material Design Lite classes used to style each section, please refer to the official documentation here.

With the HTML markup part covered, let's move on to the JavaScript code. Let 's break it into parts to understand it better.

Listing 3-af: Partial doc.html Code

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

const {ipcRenderer} = electron;



const form = document.querySelector('form');
```

This part of the code is very easy to understand. All we are doing is referencing the **`electron`** and **`ipcRenderer`** modules—essential for the messaging lifecycle within the application.

Next, we declare a **`form`** constant that refers to the **`form`** DOM element within the HTML markup. We'll need to subscribe to its **`submit`** event so we can intercept the data and send it over to the **`ipcMain`** receiver on main.js, which will relay it to the **`ipcRenderer`** on main.html, as described in Figure 3-a.

Following that, we add an event listener to the **`form`** object for the **`submit`** event, which will trigger the execution of the **`docSubmit`** function.

Listing 3-ag: Partial doc.html Code

```
form.addEventListener('submit', docSubmit);
```

Next, let's have a look at the **`docSubmit`** function.

Listing 3-ag: Partial doc.html Code

```
function docSubmit(e) {

    e.preventDefault();

    const name = document.querySelector('#name').value;

    const date = document.querySelector('#date').value;
    ipcRenderer.send('doc:add', name + '|' + date);
}
```

The code is very simple. We start off by invoking the **`preventDefault`** method, which basically means that if the event does not get explicitly handled, its default action should not be taken as it normally would be.

In other words, the HTML form element's default **`submit`** action—which is to send the **`form`** details to a web server—will not be executed. Instead, the code that follows the **`preventDefault`** method will be executed.

The code that follows the **`preventDefault`** method takes the values of the document **`name`** and **`date`** expiration fields and concatenates them into a single string, which is sent to the listener on main.js, by invoking **`ipcRenderer.send`**.

Data validation assumptions

This concludes the code for doc.html. Notice, though, that I haven't added any code to validate the data entered, for neither the **`name`** of the document or the expiration **`date`**.

For the sake of simplicity, we will assume that the **`name`** of the document will just contain alphanumeric characters and spaces, and the expiration **`date`** will be a date formatted using this convention: *dd-MMM-yyyy* (for example, **`25-Mar-2034`**).

However, you are welcome to expand the doc.html code to add additional logic to validate the **`name`** of the document and expiration **`date`**.

Adjusting the secondary window

We've covered a lot of ground so far in this long chapter, but we're still not entirely done yet. Before we can close off this chapter, let's execute the app by running **`npm start`** from the command line to see what the app looks like.

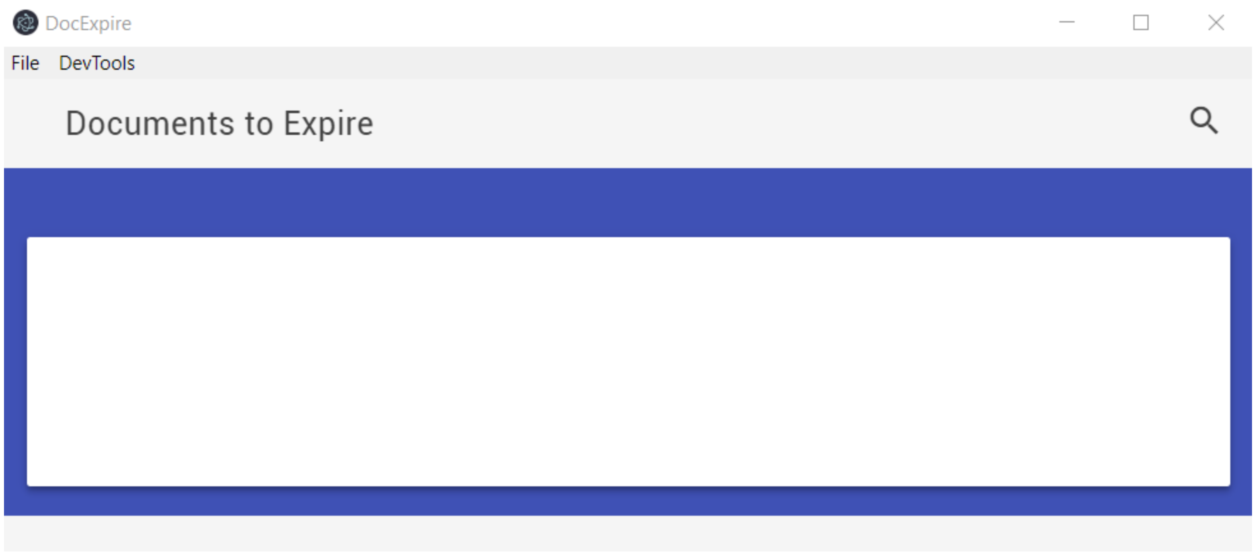


Figure 3-r: The App's Main Window

My app's main window looks like this. I'll now click on the **Add Document** menu option to see how it looks. Here's what I see.

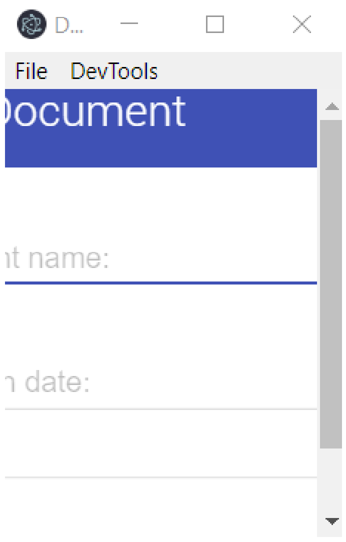


Figure 3-s: The App's Secondary Window

That's interesting, as it seems that the recent UI changes to doc.html don't fit nicely with the dimensions of the secondary window. How can we fix this?

The fix is quite easy—all we need to do is to change the dimensions of the secondary window when it is opened. Let's have a look at the existing code that's responsible for doing this.

Listing 3-ah: addDocumentWindow Function Code – main.js

```
function addDocumentWindow() {

    addDocWin = addWindow('doc',

        {width: 200, height: 300, title: 'Add Document'});


    // Garbage collection

    addDocWin.on('close', () => {

        addDocWin = null;

    });

}
```

All we need to do inside the **`addDocumentWindow`** function of main.js, is to simply adjust the values of the **`width`** and **`height`** parameters, when invoking the **`addWindow`** method. So, let's adjust them accordingly—the changes are highlighted in bold in the following code.

Listing 3-ai: Adjusted addDocumentWindow Function Code – main.js

```
function addDocumentWindow() {

    addDocWin = addWindow('doc',

        {width: 400, height: 400, title: 'Add Document'});


    // Garbage collection

    addDocWin.on('close', () => {

        addDocWin = null;

    });

}
```

If I now run the **`npm start`** command again to execute the app, I'm able to see a change in the dimensions when clicking on the **Add Document** menu option.

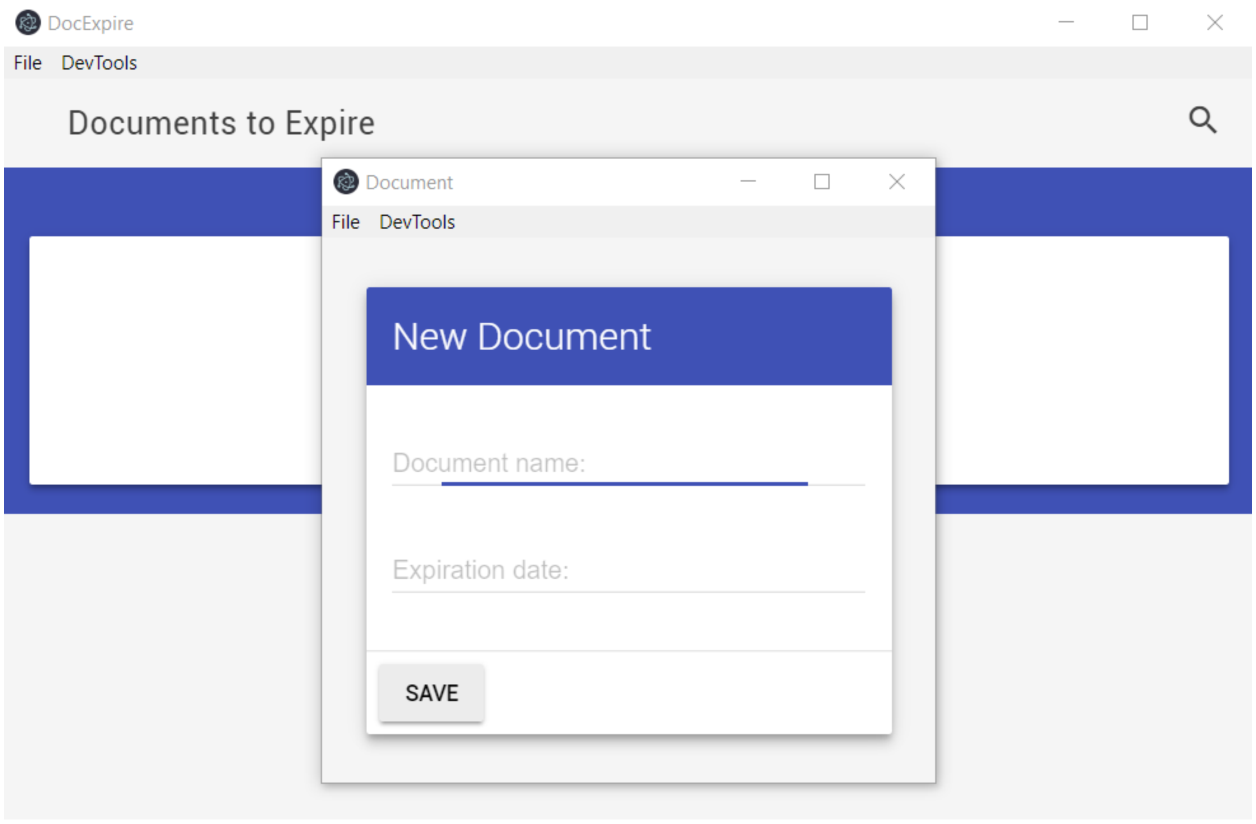


Figure 3-t: The App's Secondary Window Adjusted

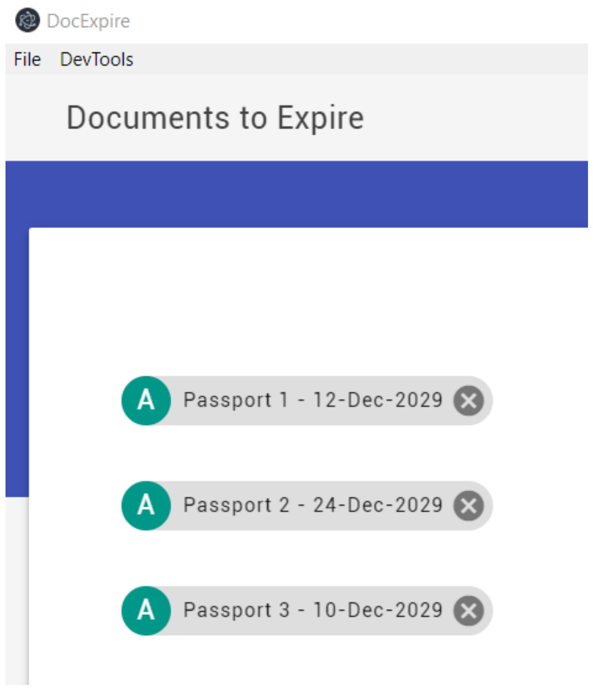
Awesome—that looks so much better. Now, let's add some documents to the list. I'll add the following items, one at a time, through the **Add Document** window, in this order:

Passport 1 – 12-Dec-2029

Passport 2 – 24-Dec-2029

Passport 3 – 10-Dec-2029

This is how the list on the main window looks afterwards.



Documents to Expire

Passport 1 - 12-Dec-2029

Passport 2 - 24-Dec-2029

Passport 3 - 10-Dec-2029

Figure 3-u: Items Added to the Main Window

Nothing unexpected—they have been added to the list in the same sequence we have added them, which is fine, given that this is the logic we have defined within our code.

Personally, what I'd like to see, is the items listed according to their expiration **`date`**. So, the item that expires first, should be listed on the top of the list. and so on.

Ordering items by expiration date

Ideally, we want to see every item added to the list ordered by the expiration **`date`**. To do this, we need to make some changes to our code. We need to make two small, but important changes to the **`createItemDom`** function. I've highlighted these two changes in bold in the following code.

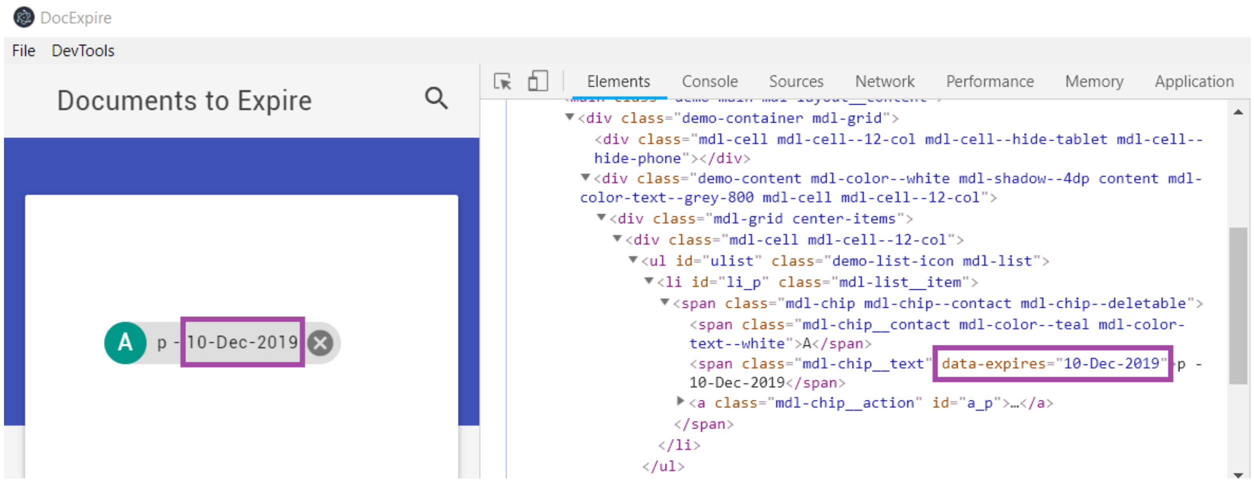
Listing 3-aj: Adjusted createItemDom Function Code – main.html

```
function createItemDom(item) {

    let iparts = item.split('|');

    let id = iparts[0].replace(' ', '_').replace('/', '-');
```

```
const doc = document.createElement('li');

doc.id = 'li_' + id;

doc.className = 'mdl-list__item';


const docText = document.createElement('span');

docText.className = 'mdl-chip mdl-chip--contact mdl-chip--deletable';


let exp = checkExpDate(iparts[1]);

const sp1 = document.createElement('span');

sp1.className = (exp == 'A') ?

   'mdl-chip__contact mdl-color--teal mdl-color-text--white' :

   'mdl-chip__contact mdl-color--red mdl-color-text--white';

const sp1Txt = document.createTextNode(exp);


const sp2 = document.createElement('span');

sp2.className = 'mdl-chip__text';

sp2.setAttribute('data-expires', iparts[1]);

const sp2Txt = document.createTextNode(iparts[0] +

    ' - ' + iparts[1]);


const ar = document.createElement('a');

ar.className = 'mdl-chip__action';

ar.id = 'a_' + id;
```

```
    const ir = document.createElement('i');

    ir.className = 'material-icons';

    const irTxt = document.createTextNode('cancel');


    ir.appendChild(irTxt);

    ar.appendChild(ir);


    sp1.appendChild(sp1Txt);

    docText.appendChild(sp1);


    sp2.appendChild(sp2Txt);

    docText.appendChild(sp2);


    docText.appendChild(ar);


    doc.appendChild(docText);


    sortList(ul, doc, Date.parse(iparts[1]));


    document.getElementById(ar.id).addEventListener('click', deleteItem);

}
```

Were you able to spot the changes? First, I added a custom **`data-expires`** attribute to the **`<span>`** element that contains the description of the document being added to the list. We can see it when we inspect the **`<span>`** element using Chrome Dev Tools.



```
▼<div class="demo-container mdl-grid">
   <div class="mdl-cell mdl-cell--12-col mdl-cell--hide-tablet mdl-cell--
   hide-phone"></div>
  ▼<div class="demo-content mdl-color--white mdl-shadow--4dp content mdl-
  color-text--grey-800 mdl-cell mdl-cell--12-col">
    ▼<div class="mdl-grid center-items">
      ▼<div class="mdl-cell mdl-cell--12-col">
        ▼<ul id="ulist" class="demo-list-icon mdl-list">
          ▼<li id="li_p" class="mdl-list__item">
            ▼<span class="mdl-chip mdl-chip--contact mdl-chip--deletable">
               <span class="mdl-chip__contact mdl-color--teal mdl-color-
               text--white">A</span>
               <span class="mdl-chip__text" data-expires="10-Dec-2019">p -
               10-Dec-2019</span>
             ▶<a class="mdl-chip__action" id="a_p">…</a>
             </span>
           </li>
         </ul>
```

Figure 3-v: Inspecting the data-expires Attribute using Chrome Dev Tools

Secondly, I replaced the following instruction:

```
// Adds the <li> element to its parent, the <ul> list of elements

ul.appendChild(doc);
```

With this one:

```
sortList(ul, doc, Date.parse(iparts[1]));
```

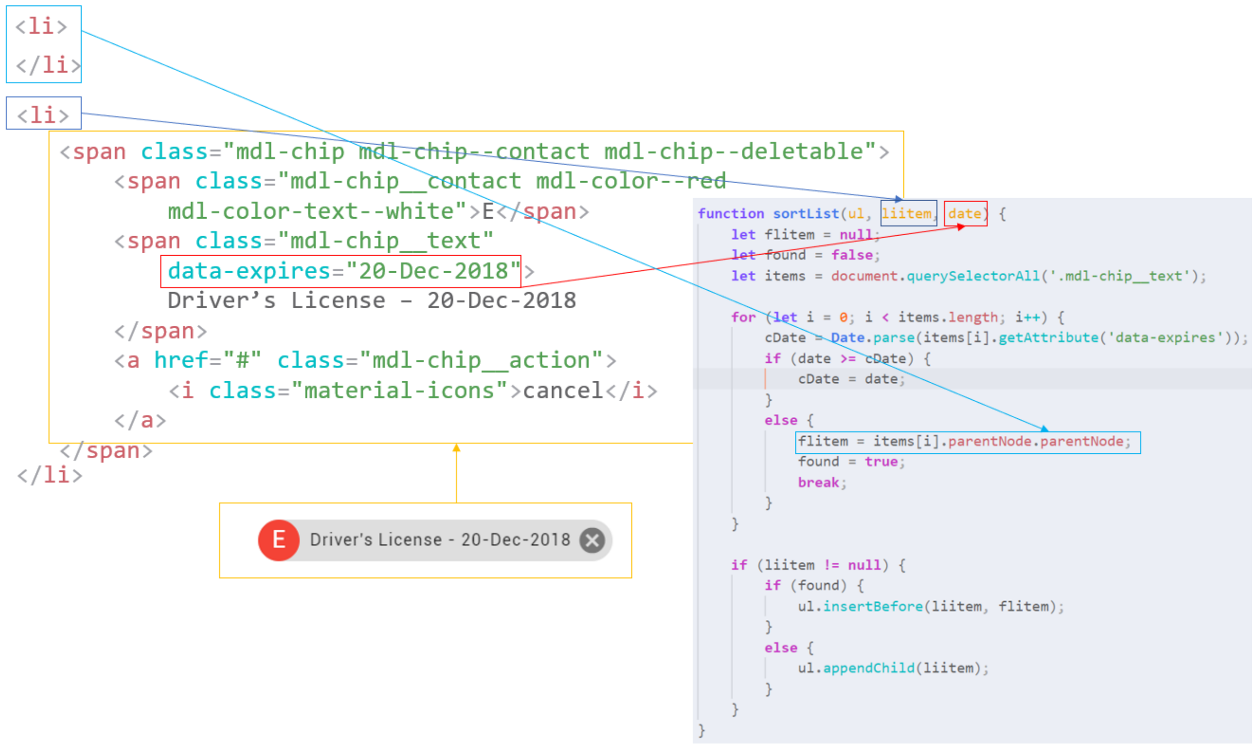
This code is responsible for inserting the element on the list in the position it corresponds to, given the expiration **`date`** mentioned on the **`data-expires`** attribute.

This is achieved by executing the logic contained within the **`sortList`** function that follows.

Listing 3-ak: sortList Function Code – main.html

```
function sortList(ul, liitem, date) {

    let flitem = null;

    let found = false;

    let items = document.querySelectorAll('.mdl-chip__text');


    for (let i = 0; i < items.length; i++) {

        cDate = Date.parse(items[i].getAttribute('data-expires'));
        if (date >= cDate) {

            cDate = date;

        }

        else {

            flitem = items[i].parentNode.parentNode;

            found = true;

            break;

        }

    }


    if (liitem != null) {

        if (found) {

            ul.insertBefore(liitem, flitem);

        }

        else {

            ul.appendChild(liitem);

        }

    }

}
```

Let’s have a look at the following diagram to better understand the relationship between the code of this function and the HTML markup.



```
<li>
</li>
<li>
    <span class="mdl-chip mdl-chip--contact mdl-chip--deletable">
        <span class="mdl-chip__contact mdl-color--red
            mdl-color-text--white">E</span>
        <span class="mdl-chip__text"
            data-expires="20-Dec-2018">
            Driver's License - 20-Dec-2018
        </span>
        <a href="#" class="mdl-chip__action">
            <i class="material-icons">cancel</i>
        </a>
    </span>
</li>
```

```
function sortList(ul, liitem, date) {
    let flitem = null;
    let found = false;
    let items = document.querySelectorAll('.mdl-chip__text');

    for (let i = 0; i < items.length; i++) {
        cDate = Date.parse(items[i].getAttribute('data-expires'));
        if (date >= cDate) {
            cDate = date;
        }
        else {
            flitem = items[i].parentNode.parentNode;
            found = true;
            break;
        }
    }

    if (liitem != null) {
        if (found) {
            ul.insertBefore(liitem, flitem);
        }
        else {
            ul.appendChild(liitem);
        }
    }
}
```

Figure 3-w: sortList Code-Markup Relationship

The **`sortList`** function receives three parameters—the first is the **`<ul>`** list to which the new document is going to be inserted.

The second parameter is the **`<li>`** item that is going to be inserted—in the preceding figure, we can see it highlighted in a **dark blue** + **yellow** box.

The third parameter is the expiration **`date`** of the document that is going to be inserted (**dark blue** + **yellow** box). This **`date`** will be compared against the dates (**`cDate`**) of each of the items already existing within the **`<ul>`** list.

If the expiration date of the document being inserted (**`date`**) is before the expiration date of the current **`<li>`** item (**light blue** box) within the loop (**`cDate`**), then we obtain a reference to the current **`<li>`** item. This is done so that the new item can be inserted before the current one, which is done by invoking the **`insertBefore`** method from the **`ul`** element.

Before we can test this, we need to do two more things. One is to ensure that all elements added to the list have a **`deleteItem`** event handler attached. We can do this by writing the following "fail-safe" function within our main.html code.

Listing 3-al: addDeleteEvents Function Code – main.html

```
function addDeleteEvents() {
    let items = document.querySelectorAll('.mdl-chip__action');

    for (let i = 0; i < items.length; i++) {

        items[i].addEventListener('click', deleteItem);

    }

}
```

This function simply ensure that all documents added to the list have an event handler attached that can execute the **`deleteItem`** method, which is used for removing a specific item from the list.

The second thing is to wire this up within the **`ipcRendered.on`** method for the **`doc:add`** event. We can do this as follows:

Listing 3-am: Updated ipcRenderer.on Method – main.html

```
ipcRenderer.on('doc:add', (e, item) => {

    createItemDom(item);

    sortList(document.getElementById('ulist'));

    addDeleteEvents();

});
```

As you can see, all we've done is added the calls to the **`sortList`** and **`addDeleteEvents`** functions after invoking **`createItemDom`**.

To test this, run **`npm start`** from the command line to execute the program, and enter the following documents in this order, by using the **Add Document** window:

Passport 1 – 12-Dec-2029

Passport 2 – 24-Dec-2029

Passport 3 – 10-Dec-2029

Passport 4 – 02-Dec-2019

Passport 5 – 09-Dec-2029

Passport 6 – 09-Dec-2019

Passport 7 – 31-May-2020

You should then see the documents added to the main window, as follows.

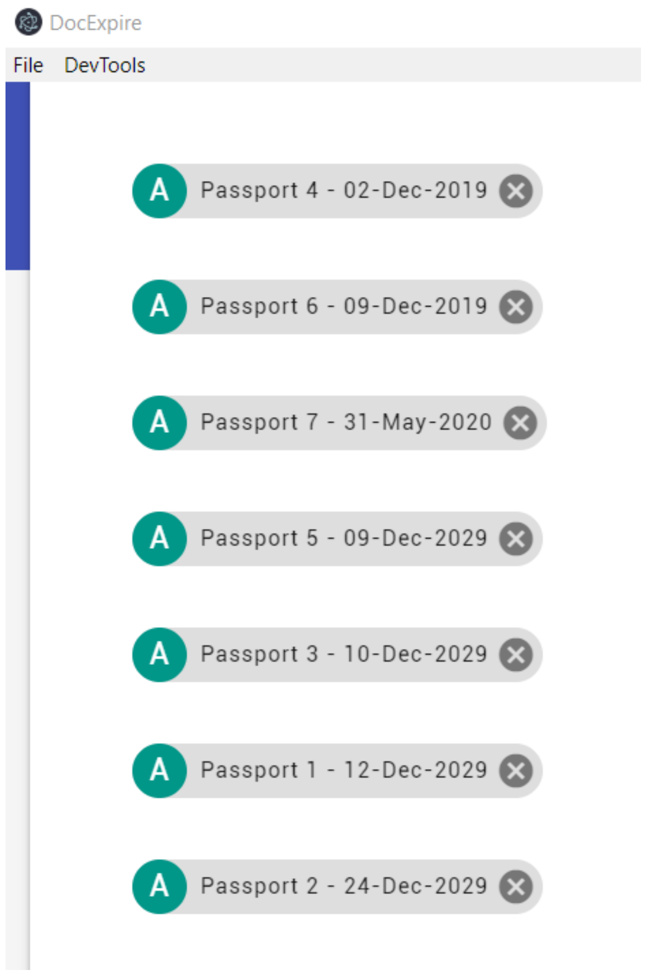


Figure 3-x: sortList Code-Markup Relationship

Notice how the documents have been added to the main window according to their expiration **`date`**, which is what we were aiming to achieve.

Recap of the code

We've made a lot of changes throughout this chapter, so it's a good idea to review the final code for the three files that make up our app's code: main.js, main.html (main window), and doc.html (secondary window).

Listing 3-an: Finished main.js

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

const url = require('url');

const path = require('path');


const {app, BrowserWindow, Menu, ipcMain} = electron;


let main, addDocWin;


// Menu template

const mainMenu = [

    {

        label: 'File',

        submenu:[

            {

                label: 'Add Document',

                click() {

                    addDocumentWindow();

                }

            },

            {

                label: 'Clear Documents',

                click() {

                    main.webContents.send('doc:clear');
                }
            },
            {
                // For cloud syncing - for the future
                label: 'User Details'
            },
            {
                label: 'Exit App',
                accelerator: process.platform ==
                    'darwin' ? 'Command+E' : 'Ctrl+E',
                click() {
                    app.quit();
                }
            }
        ]
    }
];

function menuMac() {
    if (process.platform == 'darwin') {
        mainMenu.unshift({ label: '' });
    }
}
```

```
function enableDevTools() {
    if (process.env.NODE_ENV === undefined ||
        process.env.NODE_ENV !== 'production') {
        mainMenu.push({
            label: 'DevTools',
            submenu: [
                {
                    label: 'Toggle DevTools',
                    accelerator: process.platform ==
                        'darwin' ? 'Command+T' : 'Ctrl+T',
                    click(i, fw) {
                        fw.toggleDevTools();
                    }
                }
            ]
        });
    }
}

function displayMenu() {
    menuMac();
    enableDevTools();

    // Build the menu from the template
    const menu = Menu.buildFromTemplate(mainMenu);


    // Insert the menu into the application

    Menu.setApplicationMenu(menu);

}


// Create a new window

function addWindow(name, params) {

    // Create the new window

    let win = new BrowserWindow(params);

    // Load the html into the window

    win.loadURL(url.format({

        pathname: path.join(__dirname, name + '.html'),

        protocol: 'file',

        slashes: true

    }));


    return win;

}


function addDocumentWindow() {

    addDocWin = addWindow('doc',

       {width: 400, height: 400, title: 'Add Document'});
    // Garbage collection
    addDocWin.on('close', () => {
        addDocWin = null;
    });
}

ipcMain.on('doc:add', (e, item) => {
    console.log(item);
    main.webContents.send('doc:add', item);
    addDocWin.close();
});

// Application ready event
app.on('ready', () => {
    // Create the new window
    main = addWindow('main', {});

    displayMenu();

    // Terminating the application
    main.on('closed', () => {
        app.quit();
    });
});
```

Listing 3-ao: Finished main.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport"

        content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>DocExpire</title>

    <link rel="stylesheet"

        href="./node_modules/material-design-lite/material.min.css">

    <script

        src="./node_modules/material-design-lite/material.min.js">

    </script>

    <link rel="stylesheet" href="./fonts/icon.css">

    <link rel="stylesheet" href="./fonts/styles.css">


    <style>

        .mdl-grid.center-items {

            justify-content: center;

        }

    </style>

</head>
```

```
<body>
    <div class="demo-layout mdl-layout mdl-layout--fixed-header
        mdl-js-layout mdl-color--grey-100">
        <header class="demo-header mdl-layout__header
            mdl-layout__header--scroll mdl-color--grey-100
            mdl-color-text--grey-800">
          <div class="mdl-layout__header-row">
            <span class="mdl-layout-title">Documents to Expire</span>
            <div class="mdl-layout-spacer"></div>
            <div class="mdl-textfield mdl-js-textfield
                mdl-textfield--expandable">
              <label class="mdl-button mdl-js-button
                  mdl-button--icon" for="search">
                <i class="material-icons">search</i>
              </label>
              <div class="mdl-textfield__expandable-holder">
                <input class="mdl-textfield__input"
                    type="text" id="search" onkeyup="searchItems()">
                <label class="mdl-textfield__label" for="search">
                    Enter your query...
                </label>
              </div>
            </div>
          </div>
        </header>

        <div class="demo-ribbon"></div>

        <main class="demo-main mdl-layout__content">

            <div class="demo-container mdl-grid">

                <div class="mdl-cell mdl-cell--12-col

                    mdl-cell--hide-tablet mdl-cell--hide-phone">

                </div>

                <div class="demo-content mdl-color--white

                    mdl-shadow--4dp content mdl-color-text--grey-800

                    mdl-cell mdl-cell--12-col">

                    <div class="mdl-grid center-items">

                        <div class="mdl-cell mdl-cell--12-col">

                            <ul id="ulist"

                                class="demo-list-icon mdl-list">

                            </ul>

                        </div>

                    </div>

                </div>

            </div>

        </main>

    </div>

    <script>

        const electron = require('electron');

        const {ipcRenderer} = electron;
```

```
const ul = document.querySelector('ul');

function createItemDom(item) {
    let iparts = item.split('|');
    let id = iparts[0].replace(' ', '_').replace('/', '-');

    const doc = document.createElement('li');
    doc.id = 'li_' + id;
    doc.className = 'mdl-list__item';

    const docText = document.createElement('span');
    docText.className =
        'mdl-chip mdl-chip--contact mdl-chip--deletable';

    let exp = checkExpDate(iparts[1]);
    const sp1 = document.createElement('span');
    sp1.className = (exp == 'A') ?
    'mdl-chip__contact mdl-color--teal mdl-color-text--white' :
    'mdl-chip__contact mdl-color--red mdl-color-text--white';
    const sp1Txt = document.createTextNode(exp);

    const sp2 = document.createElement('span');
    sp2.className = 'mdl-chip__text';
```

```
sp2.setAttribute('data-expires', iparts[1]);

const sp2Txt = document.
    createTextNode(iparts[0] + ' - ' + iparts[1]);


const ar = document.createElement('a');

ar.className = 'mdl-chip__action';

ar.id = 'a_' + id;


const ir = document.createElement('i');

ir.className = 'material-icons';

const irTxt = document.createTextNode('cancel');


ir.appendChild(irTxt);

ar.appendChild(ir);


sp1.appendChild(sp1Txt);

docText.appendChild(sp1);


sp2.appendChild(sp2Txt);

docText.appendChild(sp2);


docText.appendChild(ar);


doc.appendChild(docText);
```

```
        sortList(ul, doc, Date.parse(iparts[1]));

        document.getElementById(ar.id).
            addEventListener('click', deleteItem);
    }

    function deleteItem(e) {
        let doc = e.target.parentNode.parentNode.parentNode;
        doc.innerHTML = '';
        doc.parentNode.removeChild(doc);
    }

    function checkExpDate(dt) {
        let currentDate = Date.now();
        let date = Date.parse(dt);
        return (!isNaN(date) && currentDate - date < 0) ? 'A' : 'E';
    }

    function searchItems() {
        let input, filter, ul, li, a, i, txtValue;
        input = document.getElementById('search');
        filter = input.value.toUpperCase();
        ul = document.getElementById('ulist');
```

```
            li = ul.getElementsByTagName('li');

            for (i = 0; i < li.length; i ++) {
                a = li[i].getElementsByTagName('a')[0];
                txtValue = (a.textContent || a.innerText) &&
                    (a.parentNode.textContent ||
                     a.parentNode.innerText).replace('cancel', '');
                if (txtValue.toUpperCase().indexOf(filter) > -1) {
                    li[i].style.display = "";
                } else {
                    li[i].style.display = "none";
                }
            }
        }

        function sortList(ul, liitem, date) {
            let flitem = null;
            let found = false;
            let items = document.querySelectorAll('.mdl-chip__text');

            for (let i = 0; i < items.length; i++) {
                cDate = Date.parse(items[i].
                    getAttribute('data-expires'));
                if (date >= cDate) {
```

```
                cDate = date;
            }
            else {
                flitem = items[i].parentNode.parentNode;
                found = true;
                break;
            }
        }

        if (liitem != null) {
            if (found) {
                ul.insertBefore(liitem, flitem);
            }
            else {
                ul.appendChild(liitem);
            }
        }
    }

    function addDeleteEvents() {
        let items = document.querySelectorAll('.mdl-chip__action');
        for (let i = 0; i < items.length; i++) {
            items[i].addEventListener('click', deleteItem);
        }
        }


        ipcRenderer.on('doc:add', (e, item) => {

            createItemDom(item);

            sortList(document.getElementById('ulist'));

            addDeleteEvents();

        });


        ipcRenderer.on('doc:clear', () => {

            ul.innerHTML = '';

        });

    </script>

</body>

</html>
```

Listing 3-ap: Finished doc.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport"

        content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>

    <link rel="stylesheet"

        href="./node_modules/material-design-lite/material.min.css">

    <script

        src="./node_modules/material-design-lite/material.min.js">

    </script>

    <link rel="stylesheet" href="./fonts/icon.css">

    <link rel="stylesheet" href="./fonts/styles.css">


    <style>

        .mdl-layout {

            align-items: center;

            justify-content: center;

        }

        .mdl-layout__content {

            padding: 24px;

            flex: none;

        }

    </style>

</head>

<body>

    <div class="mdl-layout mdl-js-layout mdl-color--grey-100">

        <main class="mdl-layout__content">

            <div class="mdl-card mdl-shadow--6dp">
```

```
                <div class="mdl-card__title
                    mdl-color--primary mdl-color-text--white">
                    <h2 class="mdl-card__title-text">New Document</h2>
                </div>
                <form>
                    <div class="mdl-card__supporting-text">
                        <div class="mdl-textfield mdl-js-textfield">
                            <label class="mdl-textfield__label">
                                Document name:
                            </label>
                            <input class="mdl-textfield__input"
                                type="text" id="name" autofocus>
                        </div>
                        <div class="mdl-textfield mdl-js-textfield">
                            <label class="mdl
                                textfield__label">Expiration date:
                            </label>
                            <input class="mdl-textfield__input"
                                type="text" id="date">
                        </div>
                    </div>
                    <div class="mdl-card__actions mdl-card--border">
                        <button class="mdl-button mdl-js-button
                            mdl-button--raised" type="submit">
                            Save
                        </button>
                    </div>
                </form>
            </div>
        </main>
    </div>
    <script>
        const electron = require('electron');
        const {ipcRenderer} = electron;

        const form = document.querySelector('form');
        form.addEventListener('submit', docSubmit);

        function docSubmit(e) {
            e.preventDefault();
            const name = document.querySelector('#name').value;
            const date = document.querySelector('#date').value;
            ipcRenderer.send('doc:add', name + '|' + date);
        }
    </script>
</body>
</html>
```

The full project, along with the package.json file (which includes the references to the project dependencies) can be downloaded here.

If you wish, you may download the full project, unzip it to a specific folder, and then within that folder, run **npm install** from the command line to install all the project dependencies and Node.js modules required.

Summary

It’s been a long but interesting chapter—we now have our basic Electron application ready. It really doesn’t do much more than add items to a list in an ordered way, which might not seem very exciting.

However, the main objective was to explore how this technology works with a relatively simple example. We’ve see how messages between different windows are sent across, and how the UI is rendered—without worrying about how to integrate external JavaScript frameworks, which would have added unnecessary complexity to this journey.

In the next and final chapter, we are going to explore how to package our application for Windows and conclude with some thoughts on how you can broaden your view going forward with Electron.

Chapter 4 Packaging the App

App distribution

We are ready to bundle our application and put it out into the world. According to the official Electron documentation, there are a few ways to distribute an Electron application, such as:

- Packaging the app into a file,
- Rebranding with Electron binaries
- Rebranding by building electron from source
- Using packaging tools

In our case, we'll be exploring how to use one of the existing packaging tools, called Electron Packager.

Electron Packager

We are now ready to package our application so we can deploy it—we will be covering how to package the application for Windows. To do that, we are going to use a Node.js module called Electron Packager. We can install this module from our project folder by executing the following command.

Listing 4-a: Command to Install Electron Packager

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

This will save this module as a development dependency within our package.json file.

Let's open up the package.json file and add the following packaging script for Windows, which we can add under the scripts section, as follows.

Listing 4-b: Packaging Script for Windows – package.json

```
"scripts": {

    "start": "electron .",

    "package-win": "electron-packager . --overwrite --asar=true --
platform=win32 --arch=ia32 --icon=fonts/icons/win/icon.ico --prune=true -
-out=release-builds --version-string.CompanyName=CE --version-
string.FileDescription=CE --version-string.ProductName=\"DocExpire\""

}
```

In bold, we can see a reference to the icon file that we will use during packaging—which we need to place within our **project folder\fonts\icons\win** folder. Also, notice the name of the application, which I have called **DocExpire**.

We have the **fonts** folder within our project folder, but we don't have the **icons\win** subfolder, so we must manually create it. I've chosen the following icon for the application—feel free to use any other that you like. Make sure you use the .ico version of the icon.

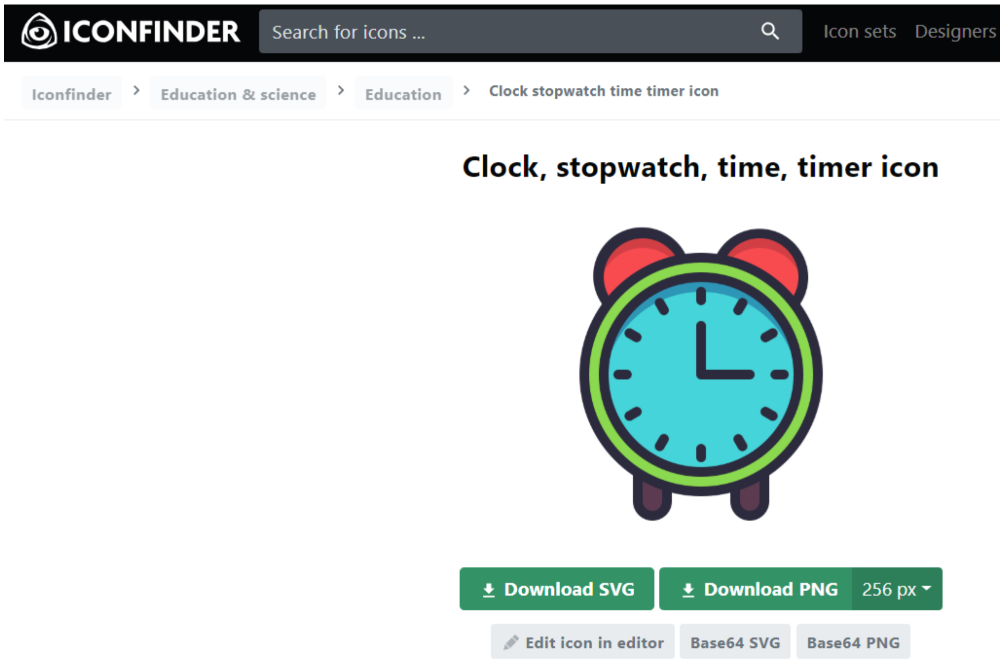


Figure 4-a: Our Application Icon

Let's now place the downloaded .ico file in our **project folder\fonts\icons\win** folder. Once downloaded, rename it to **icon.ico**.

With this in place, let's now go back to the command line and execute the windows packaging script we have just added.

But before we do that, let's double-check how our package.json file looks. In the following figure, it's highlighted in bold in the packaging script section.

Listing 4-c: package.json

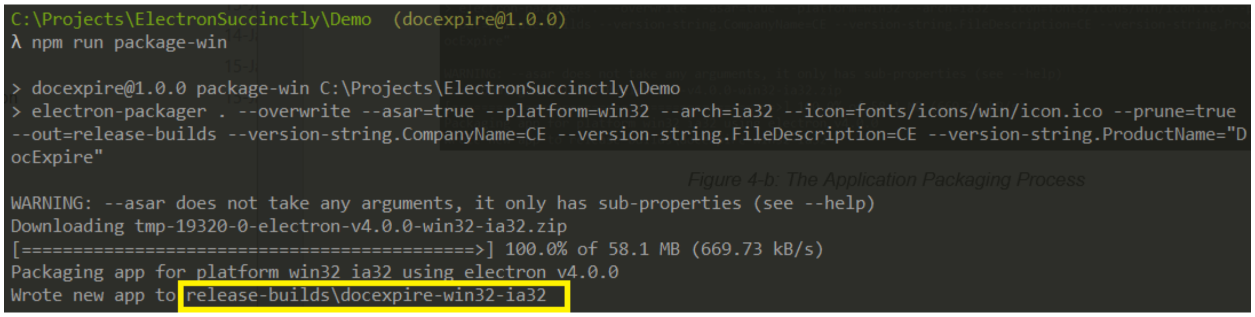
```
{
  "name": "docexpire",
  "version": "1.0.0",
  "description": "document expiration desktop app",
  "main": "main.js",
  "dependencies": {
    "material-design-lite": "^1.3.0"
  },
  "devDependencies": {
    "electron": "^4.0.0",
    "electron-packager": "^13.0.1"
  },
  "scripts": {
    "start": "electron .",
    "package-win": "electron-packager . --overwrite --asar=true --
platform=win32 --arch=ia32 --icon=fonts/icons/win/icon.ico --prune=true -
-out=release-builds --version-string.CompanyName=CE --version-
string.FileDescription=CE --version-string.ProductName=\"DocExpire\""
  },
  "author": "Ed Freitas",
  "license": "MIT"
}
```

So, let's give it a try and run the script within the project folder using the command line. We can execute it as follows.

Listing 4-d: Windows Packing Script

```
npm run package-win
```

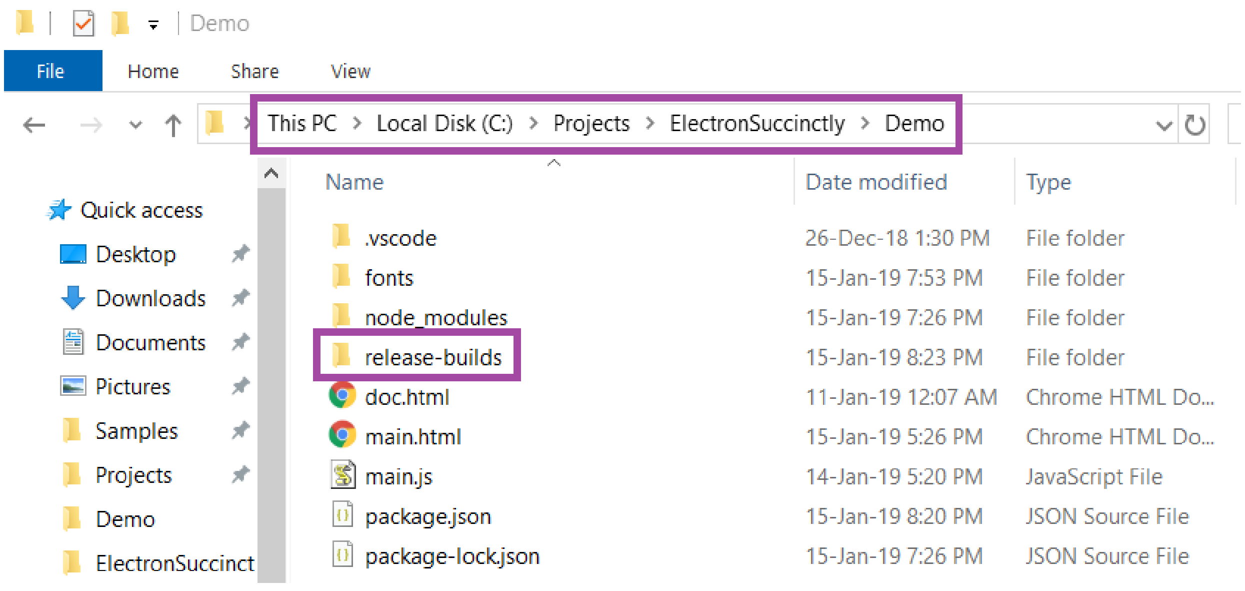
Notice that **`package-win`** is the name we have given to the packaging script. This is the result from my machine.



```
C:\Projects\ElectronSuccinctly\Demo  (docexpire@1.0.0)
λ npm run package-win

> docexpire@1.0.0 package-win C:\Projects\ElectronSuccinctly\Demo
> electron-packager . --overwrite --asar=true --platform=win32 --arch=ia32 --icon=fonts/icons/win/icon.ico --prune=true
--out=release-builds --version-string.CompanyName=CE --version-string.FileDescription=CE --version-string.ProductName="D
ocExpire"

WARNING: --asar does not take any arguments, it only has sub-properties (see --help)
Downloading tmp-19320-0-electron-v4.0.0-win32-ia32.zip
[============================================>] 100.0% of 58.1 MB (669.73 kB/s)
Packaging app for platform win32 ia32 using electron v4.0.0
Wrote new app to release-builds\docexpire-win32-ia32
```

Figure 4-b: The Application Packaging Process

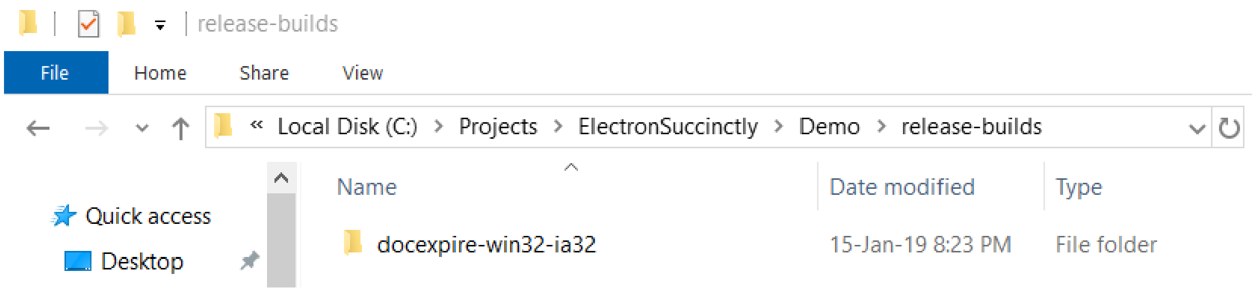
The packaged application is built to the **release-builds** folder under our project folder. We can verify this ourselves by checking our project folder.



Name	Date modified	Type
.vscode	26-Dec-18 1:30 PM	File folder
fonts	15-Jan-19 7:53 PM	File folder
node_modules	15-Jan-19 7:26 PM	File folder
release-builds	15-Jan-19 8:23 PM	File folder
doc.html	11-Jan-19 12:07 AM	Chrome HTML Do...
main.html	15-Jan-19 5:26 PM	Chrome HTML Do...
main.js	14-Jan-19 5:20 PM	JavaScript File
package.json	15-Jan-19 8:20 PM	JSON Source File
package-lock.json	15-Jan-19 7:26 PM	JSON Source File

Figure 4-c: Application release-builds Folder

Let's explore what we have within the **release-builds** folder to see what the Electron Packager was able to do for us.



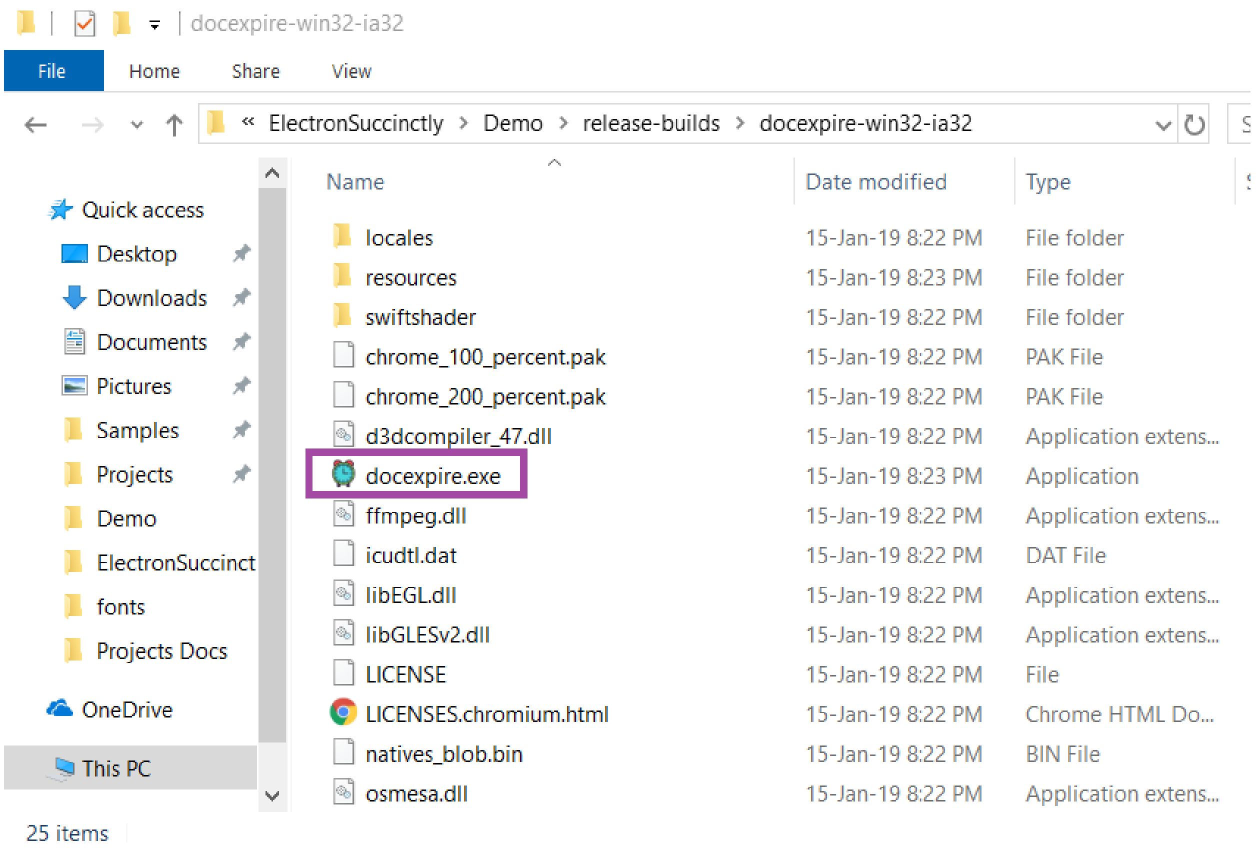
release-builds

File
Home
Share
View
Local Disk (C:) › Projects › ElectronSuccinctly › Demo › release-builds
Quick access
Desktop

Name	Date modified	Type
docexpire-win32-ia32	15-Jan-19 8:23 PM	File folder

Figure 4-d: The docexpire-win32-ia32 Folder

We can see that it has created a folder that should contain all our application files. Let's go inside the folder to see if that's the case.



docexpire-win32-ia32

Name	Date modified	Type
locales	15-Jan-19 8:22 PM	File folder
resources	15-Jan-19 8:23 PM	File folder
swiftshader	15-Jan-19 8:22 PM	File folder
chrome_100_percent.pak	15-Jan-19 8:22 PM	PAK File
chrome_200_percent.pak	15-Jan-19 8:22 PM	PAK File
d3dcompiler_47.dll	15-Jan-19 8:22 PM	Application extens...
docexpire.exe	15-Jan-19 8:23 PM	Application
ffmpeg.dll	15-Jan-19 8:22 PM	Application extens...
icudtl.dat	15-Jan-19 8:22 PM	DAT File
libEGL.dll	15-Jan-19 8:22 PM	Application extens...
libGLESv2.dll	15-Jan-19 8:22 PM	Application extens...
LICENSE	15-Jan-19 8:22 PM	File
LICENSES.chromium.html	15-Jan-19 8:22 PM	Chrome HTML Do...
natives_blob.bin	15-Jan-19 8:22 PM	BIN File
osmesa.dll	15-Jan-19 8:22 PM	Application extens...

Figure 4-e: Our Application Binaries

We can see that within this folder, Electron Packager has created all the binary runtimes our application will require, including the actual docexpire.exe file, which is the actual program we have created.

All we need to do to deploy the application once it has been built by Electron Packager is to copy the contents of this folder to any other machines with a compatible operating system version, and we are good to go.

If you are not using Windows, I highly recommend checking out this tutorial for more details on how to create a packaging script for macOS and Linux.

Wrap-up and final thoughts

Although we've covered quite a bit of ground throughout this book, we've barely scratched the surface of all the possibilities Electron can offer to web developers for creating desktop applications, with the technologies they know and love.

When I originally started working on this book, I was thinking of doing a follow-up on what was covered in one of my previous books, Ionic Succinctly. However, as I started exploring the concept of creating a desktop application using the Ionic Framework, I realized that while a book based on using the Ionic Framework could appeal to lovers of Ionic, it might not appeal to developers with backgrounds in other frameworks, such as Angular, Vue, React, and anything in between.

Based on this reasoning, I decided to start from scratch and write *Electron Succinctly* using a totally different approach, which was to be framework-agnostic, by focusing on Electron inter-process communication (IPC) and understanding the fundamental aspects of the platform.

Unlike with my previous books, where the focus was more on the code, I thought that given the broad choices web developers are faced with in terms of frameworks and libraries, they could be free to choose which framework to use when developing with Electron. Therefore, the focus shifted to using vanilla JavaScript and clearly explaining the interaction between IPC, Electron windows, markup, and JavaScript logic. I sought to mimic the experience you would expect when developing for the desktop, while using basic web technologies.

Having said this, the main takeaway for me about Electron is that it's an open canvas with endless possibilities, and is only limited by your imagination and the effort you are willing to put into it. It's like writing for the web, but contained within a desktop environment.

If you are an enthusiastic (*insert the name of your favorite JavaScript framework or library here*) developer, take the concepts from this book and just envision what type of desktop wonders you'll be able to write, simply by using the web stack you already know, and with the bits and pieces from Electron you hopefully learned from this manuscript. I invite you to continue this journey.

I hope you have enjoyed reading this book and exploring the concepts outlined here, as much as I have enjoyed writing it.

Until we meet again, keep learning, making wonders, and being the best you can possibly be. Thank you for reading.