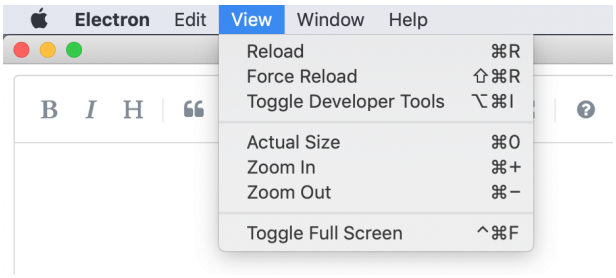
Integrating the application menu

As you already know, your application is essentially an HTML5 stack running inside Chromium, and Electron provides all necessary integration with the underlying operating system, whether that's macOS, Windows, or Linux.

The concept of application menus is slightly different across platforms. macOS, for instance, provides a single application menu that reflects the active application and displays the corresponding menu items. The Windows system tends to provide a separate menu for each instance of the application window. Finally, Linux systems usually vary based on the window manager's implementations.

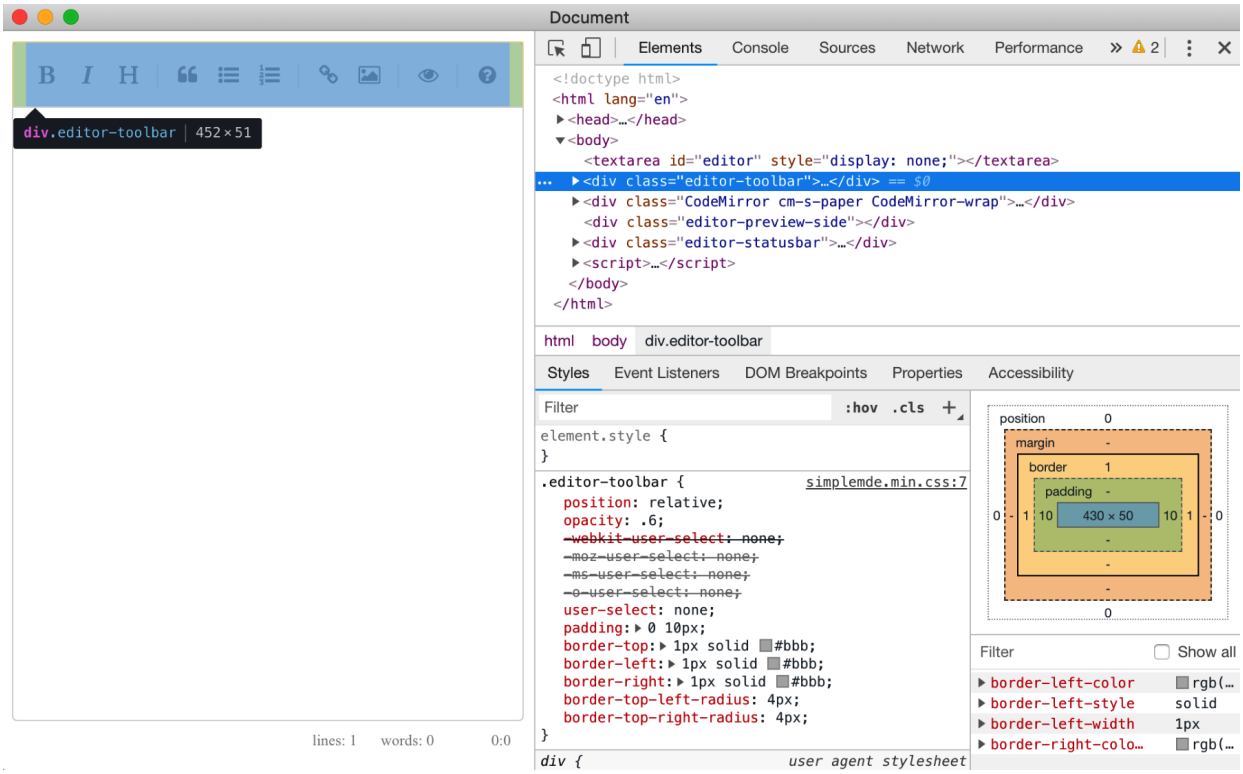
Handling every case would be quite cumbersome for developers; that is why the Electron framework provides a unified interface for building application menus from the JSON definition and takes care of integration details.

Let's take a macOS application menu as an example. As soon as you launch your application, Electron provides a set of predefined menu items. For development, one of the most popular menu items is View as it provides access to application reloading and Chrome Developer Tools:



To see the Developer Tools in action, run the application with `npm start` and click the View | Toggle Developer Tools menu item.

Note that you instantly get access to the whole set of debugging capabilities for the running application. Later on, you are probably going to use this feature a lot during development. In the following screenshot, you can see what the Chrome Developer Tools look like when you've invoked the menu item:



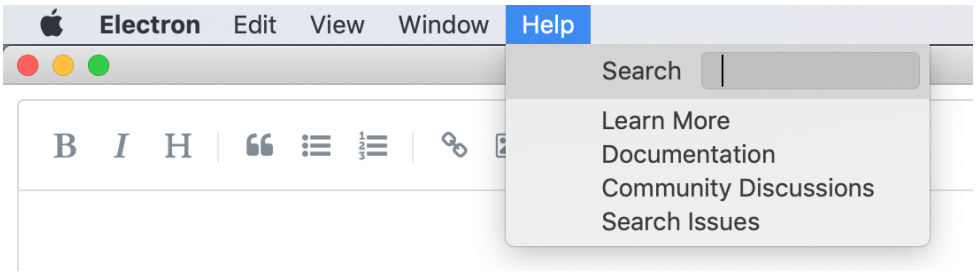
Now, let's see what it takes to create such menus from within application code. We are going to perform the following actions with the system menu component:

- Create a custom menu item
- Define the *roles* menu item
- Provide menu separators
- Support keyboard accelerators
- Support platform-specific menus

The first thing we need to address is how to create a custom menu item and render it at runtime.

Creating a custom menu item

Check out the Help menu item:



Like the other menu items, if you don't provide a custom application menu template, the Electron shell does this for you at runtime. Let's change that and provide a simple About Editor Component menu item that opens the home page of the `SimpleMDE` markdown editor component we are using for our application:

1. First of all, create a new file called `menu.js` in the project's root folder.

> **Tip**
>
> It's good practice to put menus into a separate file so that each time your application needs changing or improving, you can find the menu items quickly.

Here, you need to import the `Menu` and `shell` objects from the Electron framework. The `Menu` object provides an API that we can use to build an application menu from a JSON template. The `shell` object is going to help us invoke a browser window with a URL address that we can use to navigate:

Copy

```
const { Menu, shell } = require('electron');
```

2. Next, we need a template for our application menu that's in JSON format. Append the following code to the end of the `menu.js` file so that it holds a simple menu template:

Copy

```
const template = [
  {
    role: 'help',
    submenu: [
      {
        label: 'About Editor Component',
        click() {
            shell.openExternal('https://simplemde.com/');
        }
      }
    ]
  }
];
```

> **Note**
>
> Note that the root object of the JSON template must be an array since we define the whole application menu with multiple top-level menu items.

As you can see, there is an object with the `role` property set to `help`. This defines a top-level menu item called `Help`. We are going to focus on what `role` means in a minute, so for now take it as it is. After that, we create a `submenu` array to hold submenu items and declare an `About Editor Component` array with a `click` handler in order to invoke an external browser.

This is a minimal template, just to show you how to assemble a custom application menu. To compile our first template into a real menu, we need to call the `Menu.buildFromTemplate` function, which converts our JSON content into an Electron `Menu` object:

Copy

```
const menu = Menu.buildFromTemplate(template);

module.exports = menu;
```

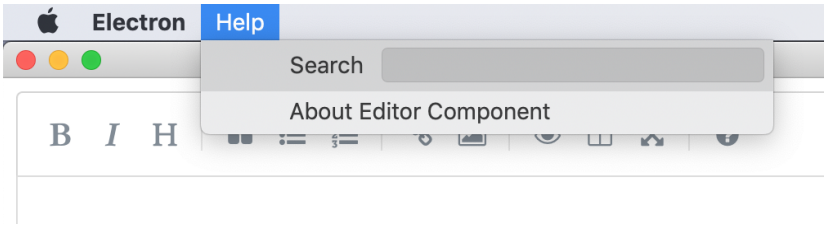
We build a new instance of the menu and export it through the `module.exports` call. Module exporting is a Node.js feature that allows us to import the `Menu` instance to other files. In our case, we need to export the menu from the `menu.js` file and import it to `index.js`, which is where the central part of our program lives.

3 Switch to the `index.js` file and update its content so that it looks as follows:

Copy

```
const { app, BrowserWindow, Menu } = require('electron');
const menu = require('./menu');

let window;

app.on('ready', () => {
    window = new BrowserWindow({
      width: 800,
      height: 600,
      webPreferences: {
        nodeIntegration: true
      }
    });
    window.loadFile('index.html');
});

Menu.setApplicationMenu(menu);
```

Most of the files should be familiar to you. We import the `menu` object from the `menu.js` file that we created earlier. Then, we build the main application window and load the `index.html` file into it. Finally, we set a new application menu based on our custom template:



4. Now, save the changes if you haven't done so already and launch the application. Given that we just redefined the whole application menu, you should see only two menu items: Electron and Help. The Electron menu is something you get out of the box when running on macOS, and the Help menu is what we defined in our code earlier.

5. Click the Help menu and ensure that you can see the About Editor Component entry. If you click the About.. menu entry, your system browser should open with the https://simplemde.com/ (https://simplemde.com/) address loaded.

Now that you can create menu items, let's take a look at the different menu item roles.

Defining menu item roles

The Electron framework supports a set of standard actions that you can associate with menu items. Instead of providing a label text, click handlers, and other settings, you can pick one of the `role` presets, and the Electron shell will handle it on the fly. Using menu presets saves a lot of time and effort as you don't need to type a lot of code to replicate standard and system entries.

Let's learn how to run Chrome's Developer Tools from our custom menu, without writing a single line of code in JavaScript:

1. Switch back to the menu template in the `menu.js` file and insert the following block to create a new `Debugging` menu:

Copy

```
const template = [
  {
    role: 'help',
    submenu: [
      {
        label: 'About Editor Component',
        click() {
            shell.openExternal('https://simplemde.com/');
        }
      }
    ]
  },
  {
    label: 'Debugging',
    submenu: [
      {
        role: 'toggleDevTools'
      }
    ]
```

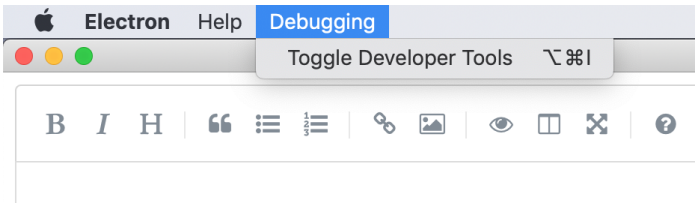
Note how we set only a single attribute, that is, `role`, to the value of `toggleDevTools` in the `submenu` array. `toggleDevTools` is one of the numerous predefined roles that the Electron framework supports. With a single role reference, your application usually gets a label, keyboard shortcut, and a click handler. In some cases, you may get even a complex menu structure with child items, such as when you use a `Help` role.

2 Run the application to see the `toggleDevTools` role in action:

Copy

```
npm start
```

Note that you now have two custom top-level menus. One of those is Debugging, which contains the Toggle Developer Tools menu item. Once you click it, you should get the standard Chrome Developer Tools on your screen:



3 Changing the title of the predefined role item is easy. Just add the `label` attribute, as shown in the following code:

Copy

```
    {
        label: 'Debugging',
        submenu: [
          {
            label: 'Dev Tools',
            role: 'toggleDevTools'
          }
        ]
    }
```

4. Now, if you run the application once again, the title of the menu item will be Dev Tools, but the behavior is still the same—it opens Chrome's Developer Tools when it's clicked.

Note

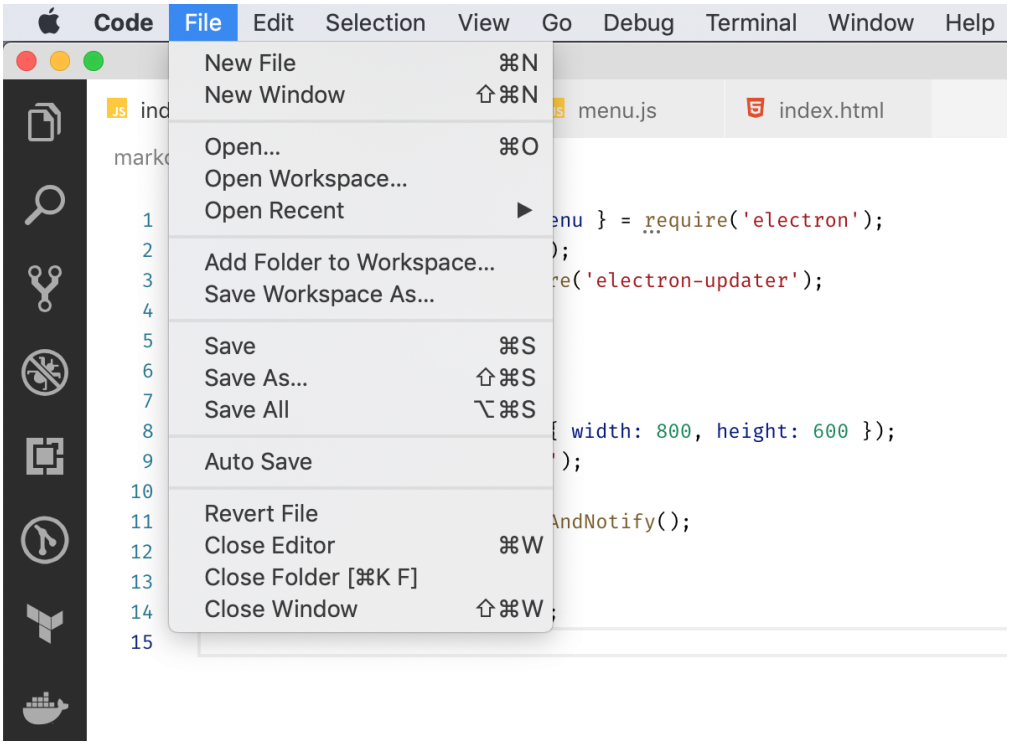
You can find out more about supported `role` values at https://electronjs.org/docs/api/menu-item#roles (https://electronjs.org/docs/api/menu-item#roles).

A typical application may contain lots of menu items. In the next section, we are going to learn how to gather actions into groups and use menu separators.

Providing menu separators

Let's stop for a moment. Traditionally, in large applications, developers collect menu items into logical groups so that it is much easier for end users to remember and use them.

The following is an example of the File menu from Visual Studio Code, which you are probably using right now to edit project files:



The keyboard shortcuts may differ, depending on the platform you are using, but the structure should be the same with all operating systems.

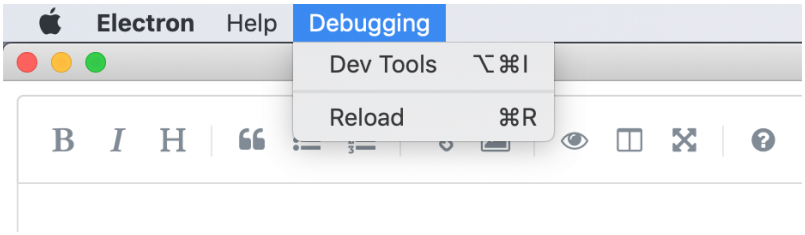
Note how developers group multiple items into separate areas. If you want to separate two menu items, follow these steps:

1. You can use an extra entry that has the `type` attribute set to `separator`. This instructs Electron to render a horizontal line to separate items visually.
2. Update the code for your `Debugging` menu so that it looks as follows:

Copy

```
{
    label: 'Debugging',
    submenu: [
      {
        label: 'Dev Tools',
        role: 'toggleDevTools'
      },

      { type: 'separator' },
      { role: 'reload' }
    ]
}
```

3. Restart the application. Inside the Debugging menu item, you should see two entries: Dev Tools and Reload:



Notice How the horizontal line separates both entries. This is our `separator` role in action, and you can use as many separators as you like in your menus.

Now, let's learn how Electron handles keyboard shortcuts, also known as accelerators, and key combinations.

Supporting keyboard accelerators

Accelerators are strings that can contain multiple modifiers and a single key code, combined by the + character, and are used to define keyboard shortcuts throughout your application.

Traditionally, menu items in applications provide support for keyboard shortcuts. Nowadays, everyone is used to using the *Cmd* + *S* or *Ctrl* + *S* combinations to save a file, *Cmd* + *P* or *Ctrl* + *P* to print a document, and so on.

Electron provides support for keyboard shortcuts, or *accelerators*, that you can use either globally or with a particular menu item. To create a new keyboard shortcut, you need to add a new attribute called `accelerator` to your menu item and specify the key combination in plain text.

In the previous examples, when you created a menu item separator, we introduced an additional menu item called Reload. This reloads the embedded browser with each click and allows you to see the updated HTML code. The `reload` role covers this functionality, but the item has no keyboard shortcut by default. Let's fix this by adding an *Alt* + *R* shortcut:

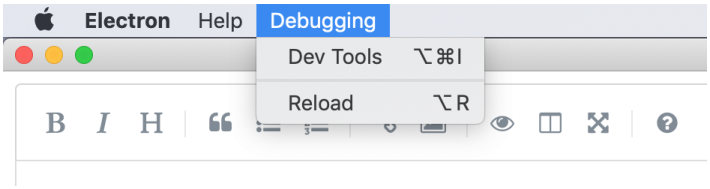
1. Edit the `menu.js` file and add the object, as shown in the following code:

Copy

```
{
  role: 'reload',
  accelerator: 'Alt+R'
}
```

2. Save the file and restart the application once again.

This time, the Reload menu item has shortcut details listed next to the label. If you are using macOS, for instance, it will be a special Alt symbol, but for Windows and Linux, it may be just the word Alt:



Note that, for many predefined menu roles, the Electron framework provides the most commonly used combinations out of the box.

> **Note**
>
> You can find out more about accelerators and their use cases at https://electronjs.org/docs/api/accelerator (https://electronjs.org/docs/api/accelerator).

The next thing we need to address is menus that are specific to a particular platform.

Supporting platform-specific menus

While Electron provides a unified and convenient way to build application menus across platforms, there are still scenarios where you may want to tune the behavior or appearance of certain items based on the platform your users use.

An excellent example of a platform-specific rendering is a macOS deployment. If you are a macOS user, you already know that each application has a specific item that always goes first in the application menu. This menu item always has the same label as the application name, and it provides some application-specific facilities, such as quitting the running instance, navigating to preferences, often showing the `About` link, and so on.

Let's create a macOS-specific menu item that allows your users to see the `About` dialog and also quit the application:

1. First of all, we need to fetch the name of the application somehow. You can do that by importing the `app` object from the Electron framework:

Copy

```
const { app, Menu, shell } = require('electron');
```

The `app` object includes the `getName` method, which fetches the application name from the `package.json` file.

Of course, you can hardcode the name as a string, but it is much more convenient to get the value dynamically at runtime from the package configuration file. This allows us to keep a single centralized place for the application name and makes our code reusable across multiple applications.

Node.js exposes a global object called `process`, which provides access to environment variables. This object can also provide information about the current platform architecture. We are going to check this against the `darwin` value to detect the macOS platform.

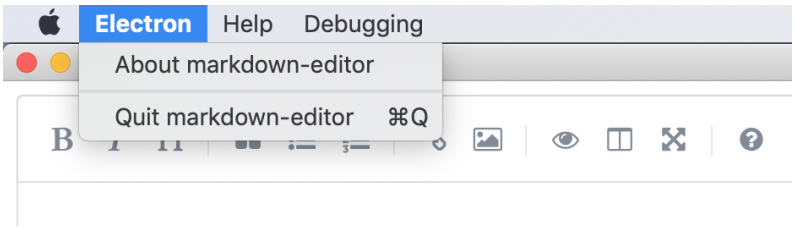
2 Append the following code right after the `template` declaration:

Copy

```
if (process.platform === 'darwin') {
  template.unshift({
    label: app.getName(),
    submenu: [
      { role: 'about' },
      { type: 'separator' },
      { role: 'quit' }
    ]
  })
}
```

As you can see, we check for the `darwin` string. In the case of an application running on macOS, a new menu entry is inserted at the beginning of the application menu.

For the time being, it is going to show Electron every time you run the `npm start` command, but don't worry—we are going to change that shortly:



The following options are available when you're checking for process architecture:

- aix
- darwin
- freebsd
- linux
- openbsd
- sunos

win32

Typically, you are going to check for `darwin` (macOS), `linux` (Ubuntu and other Linux systems), and `win32` (Windows platforms).

> **Note**
>
> For more details regarding `process.platform`, please refer to the following Node.js documentation: https://nodejs.org/api/process.html#process_process_platform (https://nodejs.org/api/process.html#process_process_platform).

Configuring the application name in the menu

You may have already noticed the Electron label in the main application menu. This has happened because we launched a generic Electron shell to run and test our application with the `npm start` command. As you may recall, we defined the `start` command like so:

Copy

```
{
  "name": "markdown-editor",
  "version": "1.1.0",
  "main": "index.js",

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

  "devDependencies": {
    "electron": "^7.0.0",
    "electron-builder": "^21.2.0"
  },
  "dependencies": {
    "simplemde": "^1.11.2"
  }
}
```

But when you package the application for distribution, it is going to have its own version of Electron embedded in it. In that case, the name of your application renders as expected.

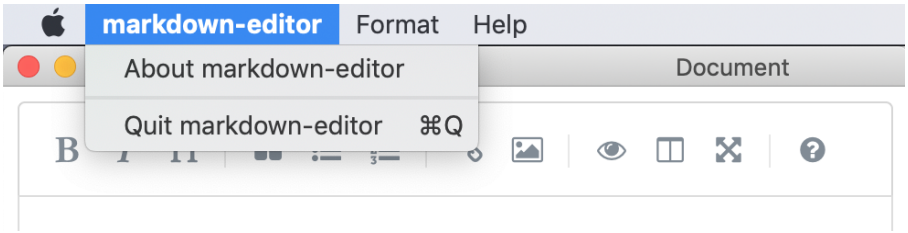
Let's test the package with the macOS build:

1. Append the `build:macos` command to the `scripts` section of the `package.json` file:

Copy

```
{
  "scripts": {
    "start": "electron .",
    "build:macos": "electron-builder --macos --dir"
  }
}
```

2. Now, execute the `npm run build:macos` command in the Terminal to create a quick package for local development and testing.
3. Next, go to the `dist/mac` folder and run the `markdown-editor` application by double-clicking on its icon:



> **Note**
>
> Note that the application menu now shows the correct value. Here, the application is called markdown-editor.

4. The code in the `menu.js` file now takes the following values from the `package.json` settings:

Copy

```
{
  "name": "markdown-editor",
  "version": "1.0.0"
}
```

The same behavior applies to the application version. When you run your project in testing mode, the About box will show the Electron framework version. For the packaged application, however, you should see the correct value.

Hiding menu items

There's one more important topic we should touch on when it comes to the conditional visibility of menu items. Besides platform-specific entries, developers usually provide utility functions that are relevant only for local development and debugging.

Let's take *Chrome Developer Tools* as an example. This is an extremely convenient set of utilities that help you debug code and inspect the layout at runtime. However, you don't want your end users accessing the code when they're using the application in real life. In most cases, it is going to be harmful rather than useful. That's why we're going to learn how to use particular menu items for development but hide them in production mode.

It may be a good idea to clean up the menu a bit first. Perform the following steps to do so:

1. Remove the `Debugging` menu from the template and only leave the `Help` entry, as shown in the following code:

Copy

```
const template = [
  {
    role: 'help',
    submenu: [
      {
        label: 'About Editor Component',
        click() {
          shell.openExternal('https://simplemde.com/');
        }
      }
    ]
  }
];

const menu = Menu.buildFromTemplate(template);

module.exports = menu;
```

2. Run the project with `npm start` and ensure there is no `Debugging` item in the application menu.

We have already used the `process` object from Node.js to detect the platform. `process` also provides access to environment variables by utilizing the `process.env` object. Each property of this object is a runtime environment variable.

Let's assume that we would like to use extra menus when the `DEBUG` environment variable is provided. In this case, the application needs to check for `process.env.DEBUG` .

3. Take a look at the following code to get a better understanding of how to check for environment variables:

Copy

```
if (process.env.DEBUG) {
  template.push({
    label: 'Debugging',
    submenu: [
      {
        label: 'Dev Tools',
        role: 'toggleDevTools'
      },

      { type: 'separator' },
      {
        role: 'reload',
        accelerator: 'Alt+R'
      }
    ]
  });
}
```

As you can see, once you have defined the `DEBUG` environment variable, the application pushes an extra `Debugging` item to the main application menu. This process is similar to the one we used earlier to add an extra menu item for macOS platforms.

4 Now, let's modify our start script so that we always start in debugging mode for local development and testing:

Copy

```
{
  "name": "markdown-editor",
  "version": "1.1.0",
  "description": "",
  "main": "index.js",

  "scripts": {
    "start": "DEBUG=true electron ."
  }
}
```

Tip

On Windows, you will need to use the `set DEBUG=true & electron` command since the Windows Command Prompt uses `set` to define environment variables.

You can use environment variables with production applications too. However, while you can add some debugging capabilities, please don't hide any security-sensitive features behind these flags.

With the help of environment variables, you can enable or disable certain features in your application. This is excellent since it allows you to have better debugging and testing utilities without confusing your application users with technical and low-level functionalities.

In the next section, we are going to learn how Node.js and Chrome processes can communicate and how menu items can help us send messages between both.

Sending messages between processes

Let's take a closer look at keyboard handling with our editor. By default, the `SimpleMDE` component provides support for most common editing shortcuts, such as the following:

- *Cmd* + *B* (Mac) or *Ctrl* + *B* (PC) to toggle the bold feature
- *Cmd* + *H* (Mac) or Ctrl + *H* (PC) to toggle the heading feature
- *Cmd* + *I* (Mac) or Ctrl + *I* (PC) to toggle the italics feature

Note

Note, however, that these commands are supported by the web component itself, not by the Electron shell. You can find out more about supported keyboard shortcuts at https://github.com/sparksuite/simplemde-markdown-editor#keyboard-shortcuts (https://github.com/sparksuite/simplemde-markdown-editor#keyboard-shortcuts).

The application menu isn't part of the web page. Therefore, we need a way to handle clicks and let the web page know that something has happened, or to trigger some code in JavaScript.

As you already know, the Electron framework is a combination of Chromium (rendering process) and Node.js (`main` process). Those processes are running side by side but isolated, and the only way to communicate between both processes is by sending messages.

This is why we are going to build the following data flow. The users of your application should get the Edit menu with the Bold item. Every time the Bold menu item is clicked, the Node.js (`main` process) handles the keyboard event and sends the message to the web page (rendering process) that the user wants to toggle the Bold feature for. Through JavaScript, the web page invokes the underlying functionality in the markdown editor component it uses.

Introducing editor-event

Let's introduce `editor-event` so that we can handle messages from Node.js . We need to import an `ipcRenderer` object from the Electron framework and listen to any channel. In this case, it is going to be `editor-event` . For the sake of simplicity, let's output the message's content to the browser console:

Copy

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

  ipcRenderer.on('editor-event', (event, arg) => {
    console.log(arg);
  });
</script>
```

The preceding code listens to the `editor-event` channel and writes the message to the browser console's output.

Sending confirmation messages to the main process

You can also send messages back to the `main` process with the `send` function:

Copy

```
ipcRenderer.send('<channel-name>', arg);
```

As an exercise, let's send a confirmation back to the main process. Electron provides convenient access to the sender of the message via the `event` argument. This allows us to have generic message handlers wired with multiple channels.

The Node.js part of the application is going to listen to the `editor-reply` channel to receive feedback from the web page.

1. Update the code of the index.html page to reflect the following example:

Copy

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

  ipcRenderer.on('editor-event', (event, arg) => {
    console.log(arg);
    // send message back to main process
    event.sender.send('editor-reply', `Received ${arg}`);
  });
</script>
```

2. At the renderer side, we need to create a reply handler. First, we need to import the `ipcMain` project from the Electron framework. Update the `menu.js` file and add the following import to the top of the file:

Copy

```
const { ipcMain } = require('electron');
```

3. Next, write the handler, similar to what we did for the web page scripts:

Copy

```
ipcMain.on('editor-reply', (event, arg) => {
  console.log(`Received reply from web page: ${arg}`);
});
```

To keep things simple and understandable, we also put the content of the message in the output.

Now, it's time to see the messages go from the renderer to the main process.

4 For testing purposes, append the following code to the bottom of the script in the `index.html` page:

Copy

```
ipcRenderer.send('editor-reply', 'Page Loaded');
```

5 The whole script block should look as follows:

Copy

```
<script>
      var editor = new SimpleMDE({
        element: document.getElementById('editor')
      });

      const { ipcRenderer } = require('electron');

      ipcRenderer.on('editor-event', (event, arg) => {
        console.log(arg);
        event.sender.send('editor-reply', `Received ${arg}`);
      });

      ipcRenderer.send('editor-reply', 'Page Loaded');
</script>
```

As you can see, as soon as the page is rendered to the users, the script sends the `Page Loaded` message to the main process while utilizing the `editor-reply` channel. We enabled logging to the console for all reply messages once you run your application with the `npm start` script, the command's output should contain the following text:

Copy

```
> DEBUG=true electron .

Received reply from web page: hello world
```

This message means that your first messaging channel works from the renderer process to the main one.

Sending messages to the renderer process

Now, we can send messages from the main process back to the renderer. According to our initial scenario, we are going to handle application menu clicks and let the renderer process know about user interactions.

To send messages to the renderer process, we need to know what window we should address. Electron supports multiple windows with different content, and our code needs to know or figure out which window contains the editor component. For the sake of simplicity, let's access the focused window object since we have only a single-window application right now:

1. Import the `BrowserWindow` object from the Electron framework:

Copy

```
const { BrowserWindow } = require('electron');
```

The format of the call is as follows:

Copy

```
const window = BrowserWindow.getFocusedWindow();
window.webContents.send('<channel>', args);
```

At this point, we have communication handlers from both areas, that is, the browser and Node.js. It is time to wire everything with a menu item.

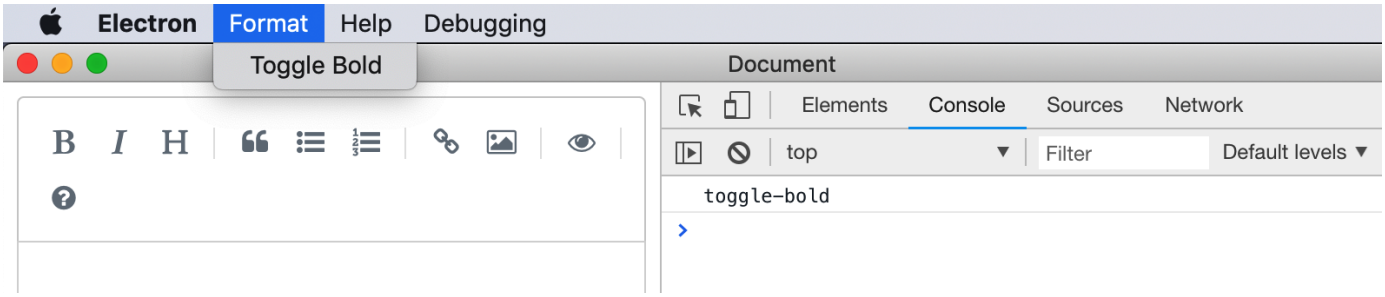
2. Update your `menu.js` file and provide a `Toggle Bold` entry that sends a `toggle-bold` message using our newly introduced `editor-event` channel. Refer to the following code for implementation details:

Copy

```
const template = [
  {
    label: 'Format',
    submenu: [
      {
        label: 'Toggle Bold',
        click() {
          const window = BrowserWindow.getFocusedWindow();
          window.webContents.send(
            'editor-event',
            'toggle-bold'
          );
        }
      }
    ]
  }
];
```

Let's check whether the messaging process works as expected.

3. Run the application with the `npm start` command, or restart it, and toggle the Developer Tools.
4. Note that you also have the Format menu, which contains the Toggle bold subitem. Click it and see what happens in the browser console output in the Developer Tools:



5. The Terminal output should contain the following text:

Copy

```
> DEBUG=true electron .

Received reply from web page: Page Loaded
Received reply from web page: Received toggle-bold
```

This is a great result! As soon as we click on the application menu button, the main process finds the focused window and sends the `toggle-bold` message. The renderer process handles the message in Javascript and posts it to the browser console. After that, it replies to the message, and the main process receives and outputs the response in the Terminal window.

Wiring the toggle bold menu

Finally, let's wire the command with the `toggle-bold` functionality:

1. The markdown editor component we are using for this application provides multiple functions that developers can invoke from code. One of those functions is `toggleBold()`. Our code can check the content of the message, and if it's the `toggle-bold` one, it will run the corresponding component function:

Copy

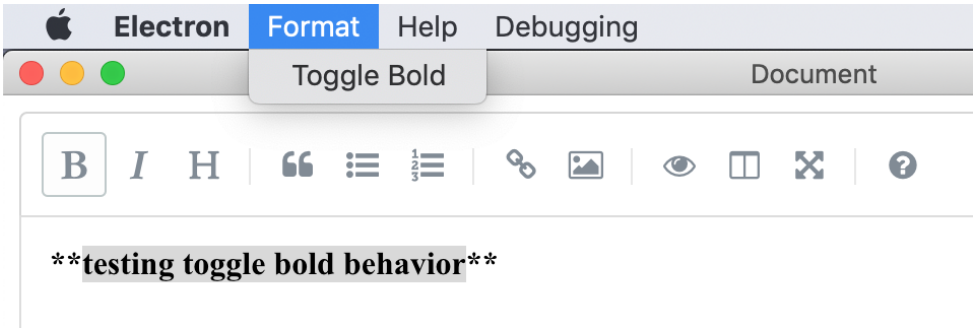
```
if (arg === 'toggle-bold') {
  editor.toggleBold();
}
```

2. The whole script section should look as follows:

Copy

```
<script>
      var editor = new SimpleMDE({
        element: document.getElementById('editor')
      });

      const { ipcRenderer } = require('electron');

      ipcRenderer.on('editor-event', (event, arg) => {
        console.log(arg);
        event.sender.send('editor-reply', `Received ${arg}`);

        if (arg === 'toggle-bold') {
          editor.toggleBold();
        }
      });

      ipcRenderer.send('editor-reply', 'Page Loaded');
</script>
```

3 Restart the application once again, type something into the editor, and then select the text. Next, click the Format | Toggle Bold menu item and see what happens. The text you previously selected will be emboldened and the markdown editor will render special `**` symbols around the selection, as shown in the following screenshot:



Congratulations! You have got cross-process messaging up and running in your Electron application.

You have also integrated the Electron application menu with the web component hosted inside the application. This employs specific messages that allow Javascript code to trigger formatting features.

As an exercise, try to provide support for more formatting features, such as `italic` and `strikethrough`, styles. The markdown editor functions of interest are `editor.toggleItalic()` and `editor.toggleStrikethrough()`.

Note

The editor component supports many other useful functions. For a list of available methods and properties, please refer to the corresponding documentation: https://github.com/sparksuite/simplemde-markdown-editor#toolbar-icons (https://github.com/sparksuite/simplemde-markdown-editor#toolbar-icons).

Saving files to a local system

In this section, we are going to provide support for saving files to the local filesystem, as well as handling global keyboard shortcuts.

Depending on the platform, you may want to support either *Cmd* + *S* for macOS or *Ctrl* + *S* for Windows or Linux desktops.

Let's start by switching back to the `menu.js` file and registering a new global shortcut. The Electron framework is going to handle it regardless of the focused window. It can handle globally registered shortcuts even if no window is present. This is often used when the application provides support for the *minimize to tray* feature:

1. Update the `menu.js` file and import the `globalShortcut` object from the Electron framework:

Copy

```
const { globalShortcut } = require('electron');
```

This object allows you to access shortcut registration utilities. Check out the following code, which shows you how to register a universal shortcut that addresses every platform:

Copy

```
app.on('ready', () => {
  globalShortcut.register('CommandOrControl+S', () => {
    console.log('Saving the file');
  });
});
```

Please note that the shortcut is called `CommandOrControl+S`. This means that, if your application is running on macOS, then Electron is going for listen to *Cmd* + *S* clicks. In any other case, it accepts the *Ctrl* + *S* click. How convenient!

2. Now, run or restart the application and, depending on the platform you are using right now, press either *Cmd* + *S* or *Ctrl* + *S* a few times.
3. Switch to the Terminal window and check the application's output. You should see the initial message we created earlier, as well as a Saving the file string for each of your clicks:

Copy

```
Received reply from web page: Page Loaded
Saving the file
Saving the file
Saving the file
```

This proves that the code is working and our Electron application is able to handle global shortcuts. Next, we need to get the content of the markdown editor somehow and save it to a file.

Work through the following these steps to practice with the event bus:

1. Node.js is going to send a message to the browser window and notify it that we are about to save a file.
2. The rendering process should extract the raw text value of the user content and send it back to the main process via another message.

3. Finally, the Node.js side is going to receive the data, invoke the system dialog to save the file, and write some content to the local disk.
4. You already know how to send messages. We used the `editor-event` channel to send `toggle-bold` commands to the renderer process. Feel free to reuse the same channel to send an extra `save` command, as shown in the following code:

Copy

```
app.on('ready', () => {
  globalShortcut.register('CommandOrControl+S', () => {
    console.log('Saving the file');

    const window = BrowserWindow.getFocusedWindow();
    window.webContents.send('editor-event', 'save');
  });
});
```

On the renderer process side, we also have an event listener. Now, we need an additional condition handler.

5. As soon as the `save` message arrives, we call `editor.getValue()` to get the actual text inside the markdown editor and send it back using the `save` channel name:

Copy

```
if (arg === 'save') {
  event.sender.send('save', editor.getValue());
}
```

6. Like all the previous implementations, the client-side handler should look as follows:

Copy

```
const { ipcRenderer } = require('electron');

ipcRenderer.on('editor-event', (event, arg) => {
  console.log(arg);
  event.sender.send('editor-reply', `Received ${arg}`);

  if (arg === 'toggle-bold') {
    editor.toggleBold();
  }

  if (arg === 'save') {
    event.sender.send('save', editor.value());
  }
});
```

7 Now, switch back to the `menu.js` file and place the listener for the `save` event that the renderer process should now be raising:

Copy

```
ipcMain.on('save', (event, arg) => {
  console.log(`Saving content of the file`);
  console.log(arg);
});
```

As you can see, this isn't doing much. For the sake of simplicity, it is just putting received data into the Terminal output so that we can verify that the messaging is working as expected.

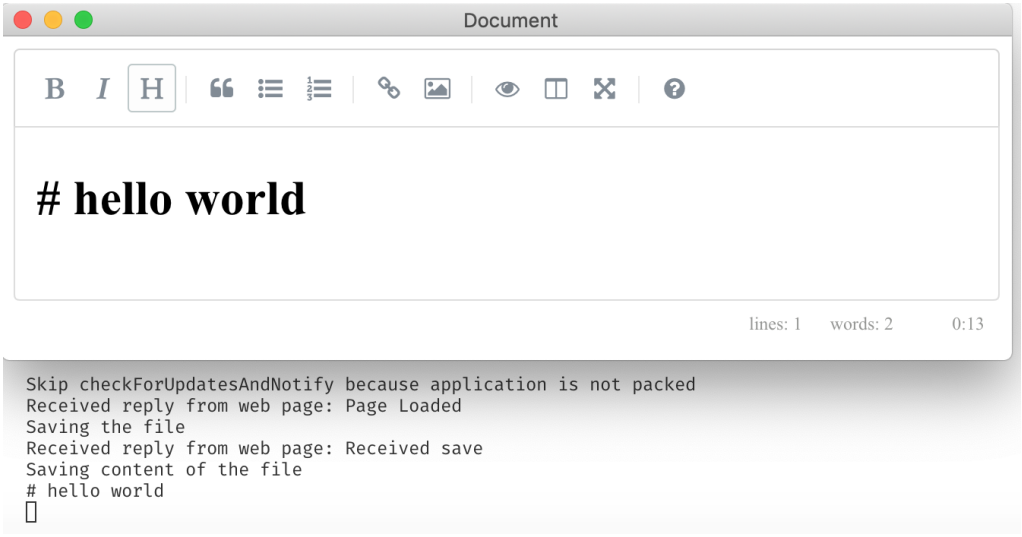
8 Before we start testing the data flow, we need to verify that our messaging implementation in `menu.js` looks as follows:

Copy

```
app.on('ready', () => {
  globalShortcut.register('CommandOrControl+S', () => {
    console.log('Saving the file');

    const window = BrowserWindow.getFocusedWindow();
    window.webContents.send('editor-event', 'save');
  });
});

ipcMain.on('save', (event, arg) => {
  console.log(`Saving content of the file`);
  console.log(arg);
});

ipcMain.on('editor-reply', (event, arg) => {
  console.log(`Received reply from web page: ${arg}`);
});
```

This should help us understand where all the strings in the Terminal window are coming from.

9 Restart the application and type `hello world`. Then, click the H button to turn the text into a `Heading` element:



As soon as you check the Terminal window while the application is running, you should see the following output from all the message handlers we set up earlier:

Copy

```
Received reply from web page: Page Loaded
Saving the file
Received reply from web page: Received save
Saving content of the file
# hello world
```

Note that you can also see the entirety of the text content. Try editing the text some more and press *Cmd* + *S* or *Ctrl* + *S* from time to time. Ensure that the latest text value ends up in the Terminal output.

Now, it's time to save the file to the local disk.

Using the save dialog

The Electron framework provides support for saving, opening, confirmation, and many more. These dialogs are native to each platform. We are going to use the macOS platform to see the native *save dialog* that macOS users are familiar with. The same code running on Windows machines triggers Windows-like dialogs.

Let's start by importing a `dialog` object into the `menu.js` file from the Electron framework:

Copy

```
const {
  app,
  Menu,
  shell,
  ipcMain,
  BrowserWindow,
  globalShortcut,
  dialog
} = require('electron');
```

You can now use the `showSaveDialog` method, which requires a parent window object reference and a set of options before it can customize the behavior of the dialog.

In our case, we are going to set the `title` of the dialog and restrict the format to `.md`, which is a *markdown* file extension:

Copy

```
ipcMain.on('save', (event, arg) => {
  console.log(`Saving content of the file`);
  console.log(arg);

  const window = BrowserWindow.getFocusedWindow();
  const options = {
    title: 'Save markdown file',
    filters: [
      {
        name: 'MyFile',
        extensions: ['md']
      }
    ]
  };

  dialog.showSaveDialog(window, options);
});
```

Note

You can find out more about dialogs, and a list of available options, in the following Electron documentation: https://electronjs.org/docs/api/dialog (https://electronjs.org/docs/api/dialog).

`showSaveDialog` receives the third parameter, that is, the callback function that gets invoked if the user closes the dialog with the `Save` or `Cancel` button. The first callback parameter provides you with the path of the file to use when saving content.

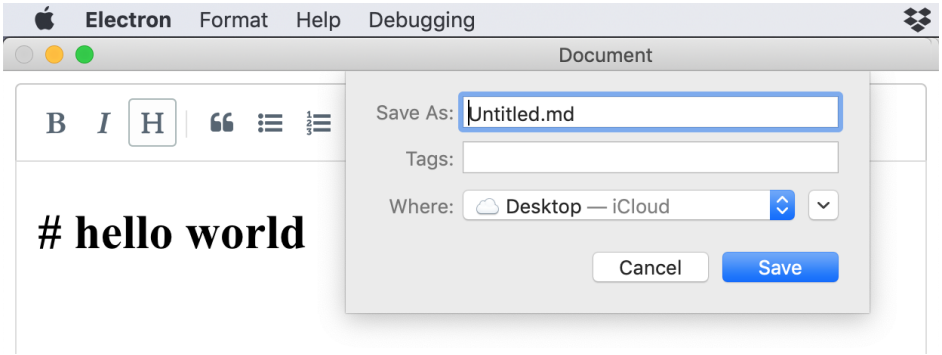
Let's see how the whole thing works.

1. Add the console.log the path to output the file name to the terminal window:

Copy

```
dialog.showSaveDialog(window, options, filename => {
  console.log(filename);
});
```

2. Restart your application, type `# hello world`, and press *Cmd* + *S* or *Ctrl* + *S*. You should see the native Save dialog, as shown in the following screenshot:



3. Change the name to `test` so that the final filename is `test.md` and click the Save button.
4. Switch to the Terminal window and check out the output. It should contain the full path to the file that you have provided via the Save dialog. In this case, for the macOS platform, it should look as follows:

Copy

```
/Users/<username>/Desktop/test.md
```

Sometimes, you may see the following message in the Terminal if you are a macOS user:

Copy

```
objc[4988]: Class FIFinderSyncExtensionHost is implemented in both
/System/Library/PrivateFrameworks/FinderKit.framework/Versions/
A/FinderKit (0x7fff9c38e210) and
/System/Library/PrivateFrameworks/FileProvider.framework/
OverrideBundles/FinderSyncCollaborationFileProviderOverride.bundle/
Contents/MacOS/FinderSyncCollaborationFileProviderOverride
(0x11ad85dc8).
One of the two will be used. Which one is undefined.
```

This is a known issue and should be fixed in future versions of macOS and Electron. Don't pay attention to this for the time being.

At this point, we have our keyboard combinations working and the application showing the Save dialog and passing the resulting file path to the main process. Now, we need to save the file.

5 To deal with files, we need to import the `fs` object from the Node.js filesystem utils:

Copy

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

We are mainly interested in the `writeFileSync` function, which receives the path to the file and the data and invokes the callback as soon as writing finishes.

6 The callback returns `String` or `undefined`, the path of the file that was chosen by the user if a callback was provided, or if the dialog was canceled, it returns `undefined`. This is why the null-check is very important.

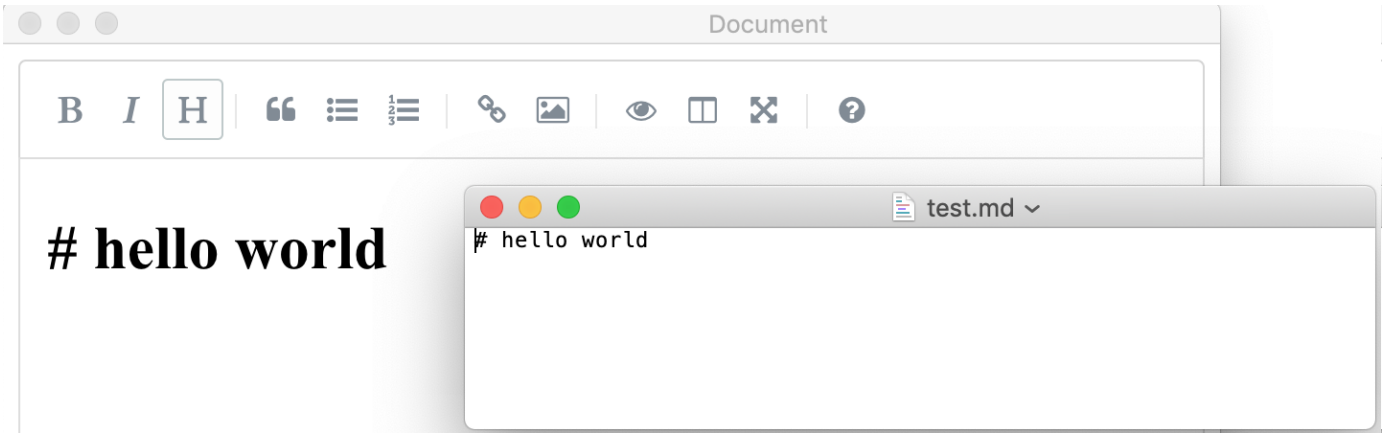
7 Check if the `filename` value has been provided and save the file using the `fs.writeFileSync` method, as shown in the following code:

Copy

```
dialog.showSaveDialog(window, options, filename => {
  if (filename) {
    console.log(`Saving content to the file: ${filename}`);
    fs.writeFileSync(filename, arg);
  }
});
```

8 Restart the application and repeat the previous steps. Type in some text, press the shortcut, and pick the location and name for the file.

9 This time, however, the file should appear in your filesystem. You can find it using the File browser and open it with the text editor. It should contain the content that you previously typed in:



10 That's all we need to do. The final implementation of the `save` event handler is as follows:

Copy

```
ipcMain.on('save', (event, arg) => {
  console.log(`Saving content of the file`);
  console.log(arg);

  const window = BrowserWindow.getFocusedWindow();
  const options = {
    title: 'Save markdown file',
    filters: [
      {
        name: 'MyFile',
        extensions: ['md']
      }
    ]
  };

  dialog.showSaveDialog(window, options, filename => {
    if (filename) {
      console.log(`Saving content to the file: ${filename}`);
      fs.writeFileSync(filename, arg);
```

In this section, we achieved the following:

- We sent the `save` event to the client-side (browser).
- The browser code handles the event, fetches the current value of the text editor, and sends it back to the Node.js side.
- The Node.js side handles the event and invokes the system save dialog.
- Once the user defines a file name and clicks Save, the content gets saved to the local filesystem.

Congratulations—you are now able to invoke system-level Save dialogs from your applications! Now, let's learn how to load files from a local system.

Loading files from a local system

Now that you have got the Open File functionality and registered the global keyboard shortcut for it, let's see what it takes to load a file from the local filesystem back into the editor component:

1. Let's start by updating the `menu.js` file and registering a second global shortcut for *Cmd* + *O* or *Ctrl* + *O*, depending on the user's desktop platform:

Copy

```
globalShortcut.register('CommandOrControl+O', () => {
    // show open dialog
});
```

We have already imported the `dialog` object from the Electron framework. You can use it to invoke the system's Open dialog as well.

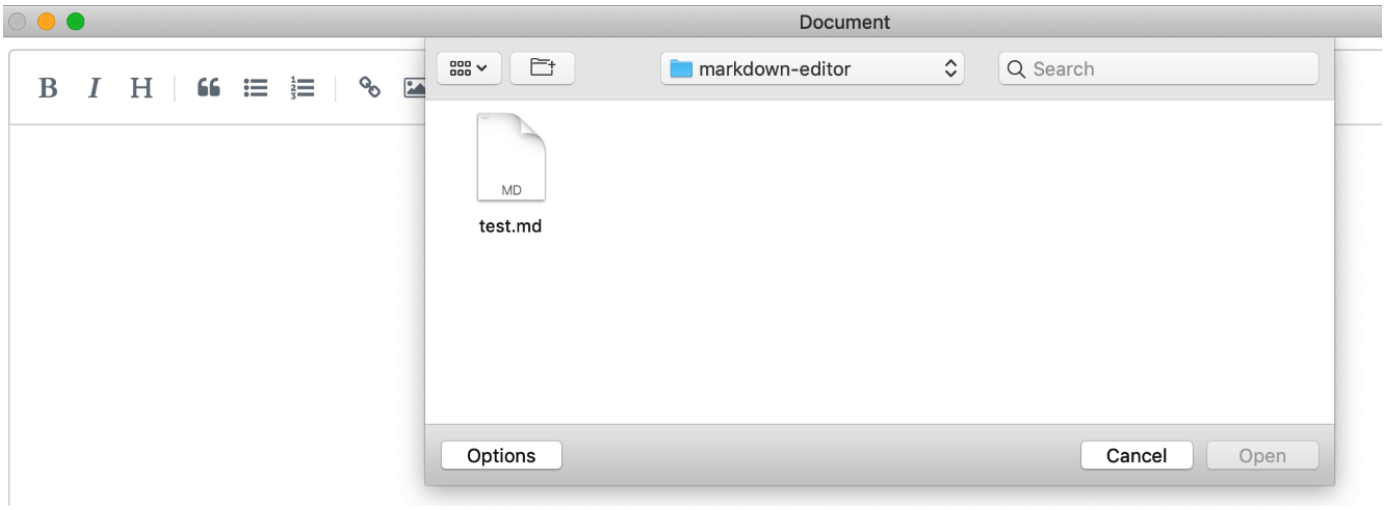
2. Update the `menu.js` file according to the following code:

Copy

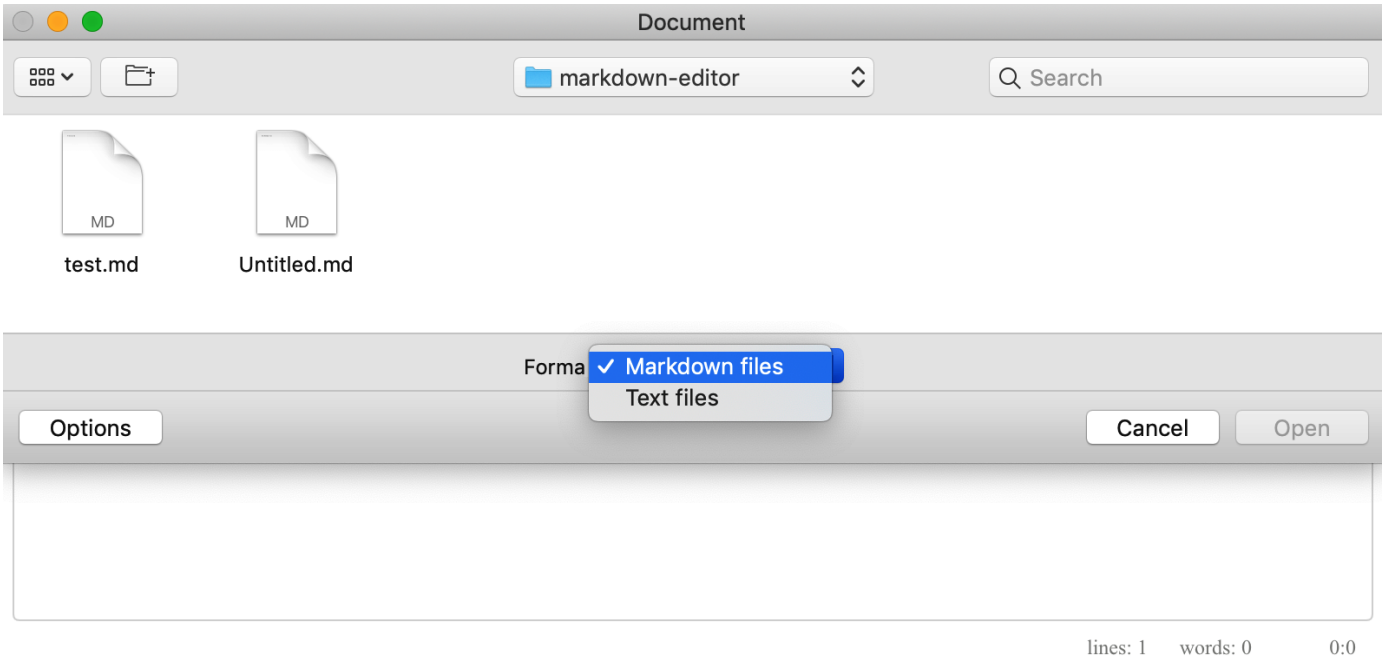
```
globalShortcut.register('CommandOrControl+O', () => {
    const window = BrowserWindow.getFocusedWindow();

    const options = {
      title: 'Pick a markdown file',
      filters: [
        { name: 'Markdown files', extensions: ['md'] },
        { name: 'Text files', extensions: ['txt'] }
      ]
    };

    dialog.showOpenDialog(window, options);
  });
```

Note that, this time, we are providing more than one file filter. This allows users to open multiple file formats in a grouped fashion. For the sake of simplicity, we are allowing our users to open markdown and plain text files.

3. Run the application and press *Cmd* + *O* or *Ctrl* + *O*, depending on the platform you are using for development. Note that the system dialog appears and allows us to select markdown files by default:



4. You can also switch to the Text files group by means of the native Open dialog:



5 Now, let's get back to the `menu.js` file. Similar to the Save dialog, the Open dialog supports a callback function that provides us with information about selected files. The user can also close the dialog without picking anything, so you should always validate the results.

6 Given the nature of our editor application, we are only providing support for editing one file at a time. That 's why you only need to pick the first file if the user performs multi-selection, as follows:

Copy

```
dialog.showOpenDialog(window, options, paths => {
  if (paths && paths.length > 0) {
    // read file and send to the renderer process
  }
});
```

7 Finally, we use the `fs` object that we imported from Node.js earlier to support the Save dialog. This time, however, we are looking for the `fs.readFileSync` method.

8 As soon as we've read the file, we need to emit the cross-process event via the `load` channel so that the rendering process can listen and perform additional actions.

9 Update the `dialog.showOpenDialog` call so that it looks as follows:

Copy

```
  dialog.showOpenDialog(window, options, paths => {
    if (paths && paths.length > 0) {
      const content = fs.readFileSync(paths[0]).toString();
      window.webContents.send('load', content);
    }
  });
```

10 Before we move on to the rendering side, please ensure that the implementation of your new global shortcut looks as follows:

Copy

```
  globalShortcut.register('CommandOrControl+O', () => {
    const window = BrowserWindow.getFocusedWindow();
    const options = {
      title: 'Pick a markdown file',
      filters: [
        { name: 'Markdown files', extensions: ['md'] },
        { name: 'Text files', extensions: ['txt'] }
      ]
    };
    dialog.showOpenDialog(window, options, paths => {
      if (paths && paths.length > 0) {
        const content = fs.readFileSync(paths[0]).toString();
        window.webContents.send('load', content);
      }
    });
  });
```

11 Open the `index.html` file for editing and scroll to the scripts section, where we already have some process communication handling in place.

12 Add a new handler that listens to the `load` channel and the corresponding messages coming from the renderer process:

Copy

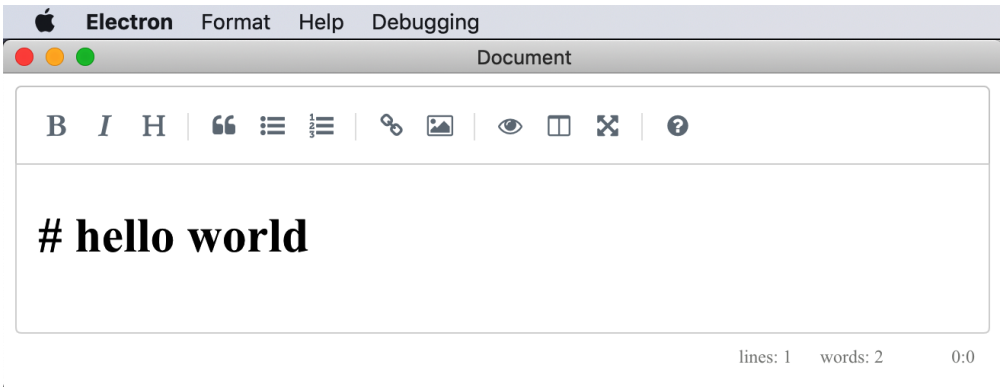
```
  ipcRenderer.on('load', (event, content) => {
    if (content) {
      // do something with content
    }
  });
```

13 As you can see, we're validating the input to ensure that the text content is indeed there and using the `editor.value(<text>)` method to replace the markdown editor content with new text:

Copy

```
ipcRenderer.on('load', (event, content) => {
  if (content) {
    editor.value(content);
  }
});
```

14 This is all we need to implement for the Open File feature. Run or restart your Electron application, press *Cmd* + *O* or *Ctrl* + *O*, and select a markdown file:



You should now see the content of the file on the screen. As soon as we call the `value()` function, the `SimpleMDE` component will reformat everything according to the markdown rules.

Creating a file menu

Given that we have two file management features, that is, Open and Save, now is an excellent time to introduce a dedicated application menu entry so that users can use a mouse to perform these operations.

Before we proceed with the application menu templates, let's refactor our file handling a bit to make the code more reusable. Don't forget that we need to call the dialogs from the menu item click handlers as well. Let's get started:

1 Move the code that's responsible for saving to a new `saveFile` function, as shown in the following code:

Copy

```
function saveFile() {
  console.log('Saving the file');

  const window = BrowserWindow.getFocusedWindow();
  window.webContents.send('editor-event', 'save');
}
```

2 Refactor and move the file loading code to the `loadFile` function:

Copy

```
function loadFile() {
  const window = BrowserWindow.getFocusedWindow();
  const options = {
    title: 'Pick a markdown file',
    filters: [
      { name: 'Markdown files', extensions: ['md'] },
      { name: 'Text files', extensions: ['txt'] }
    ]
  };
  dialog.showOpenDialog(window, options, paths => {
    if (paths && paths.length > 0) {
      const content = fs.readFileSync(paths[0]).toString();
      window.webContents.send('load', content);
    }
  });
}
```

3 Now, our `app.ready` event handler should be concise and readable:

Copy

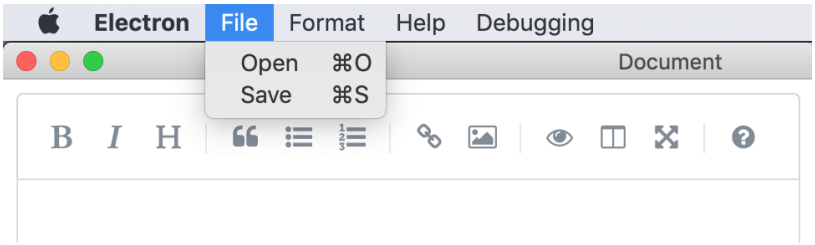
```
app.on('ready', () => {
  globalShortcut.register('CommandOrControl+S', () => {
    saveFile();
  });

  globalShortcut.register('CommandOrControl+O', () => {
    loadFile();
  });
});
```

4 Now, let's build a `File` menu template. This shouldn't be difficult as we have already touched on this. Update the `template` constant in the `menu.js` file, as shown in the following code:

Copy

```
const template = [
  {
    label: 'File',
    submenu: [
      {
        label: 'Open',
        accelerator: 'CommandOrControl+O',
        click() {
          loadFile();
        }
      },
      {
        label: 'Save',
        accelerator: 'CommandOrControl+S',
        click() {
          saveFile();
        }
      }
    ]
```

[5] Note that, if you are running on macOS, the menu item is going to show macOS-related keyboard accelerators, that is, *Cmd* + *O* or *Cmd* + *S*, in the menu. For Linux and Windows, you should see *Ctrl* + *O* or *Ctrl* + *S*, respectively:



Try clicking the menu items or pressing the corresponding keyboard combinations. You can now use the mouse and the keyboard to manage your files.

Congratulations on integrating menu and keyboard shortcuts. We have achieved the following milestones:

- We can access the local filesystem
- We can read and write files
- We can use the `Save` and `Load` dialogs
- We can wire keyboard shortcuts (accelerators)

Our end users will probably expect our application to support drag and drop functionality as well. This is something we are going to address in the next section.

Previous Section (/book/mobile/9781838552206/2/ch02lvl1sec12/fitting-the-screen-size)

Next Section (/book/mobile/9781838552206/2/ch02lvl1sec14/adding-drag-and-drop-support)