

Electron

Projects

Build over 9 cross-platform desktop applications from scratch



Packt

www.packt.com

Denys Vuika

Electron Projects

Build over 9 cross-platform desktop applications from scratch

Denys Vuika



BIRMINGHAM - MUMBAI

Electron Projects

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Pavan Ramchandani

Acquisition Editor: Karan Gupta

Content Development Editor: Divya Vijayan

Senior Editor: Mohammed Yusuf Imaratwale

Technical Editor: Jinesh Topiwala

Copy Editor: Safis Editing

Project Coordinator: Kinjal Bari

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Jyoti Chauhan

First published: November 2019

Production reference: 1281119

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83855-220-6

www.packt.com

To my dear wife, Iuliia, who always inspires me and supports me in anything I try.

– Denys Vuika



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

About the author

Denys Vuika is an application platform developer and tech lead at Alfresco Software Inc. He is a full stack developer and a constant open source contributor. He has more than 16 years of programming experience, including 10 years of frontend development with AngularJS, Angular, ASP.NET, React.js, and other modern web technologies, and more than three years of experience of Node.js development. Denys works with web technologies on a daily basis. He has a good understanding of cloud development and the containerization of web applications.

He is a frequent Medium blogger and is the author of the *Developing with Angular* book on Angular, JavaScript, and TypeScript development. He also maintains a series of Angular-based open source projects.

About the reviewer

Mats Lindblad has worked as a developer for more than 20 years. His experience covers everything from the backend with Java and PHP to the frontend using React, Angular, and Vue. And, of course, Electron!

He's dabbled with Rust and C/C++, but is more passionate about frontend technologies and the magic they enable.

He runs his own consultancy company.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Building Your First Electron Application	6
Technical requirements	6
What is Electron?	7
Preparing a development environment	8
Installing Visual Studio Code	8
Installing Visual Studio Code for Ubuntu	9
Setting up the environment for macOS	9
Installing Git on macOS	9
Installing Node.js on macOS	10
Setting up the environment for Ubuntu Linux	11
Installing Git on Ubuntu	11
Installing Node.js on Ubuntu	11
Setting up the environment for Windows	12
Installing Git on Windows	12
Installing Node.js on Windows	14
Verifying the installation	15
Creating a simple application	15
Packaging for multiple platforms	20
Packaging for macOS	21
Packaging for Ubuntu	24
Packaging for Windows	26
Summary	29
Chapter 2: Building a Markdown Editor	30
Technical requirements	30
Configuring a new project	31
Integrating the editor component	34
Fitting the screen size	37
Integrating the application menu	38
Creating a custom menu item	40
Defining menu item roles	43
Providing menu separators	45
Supporting keyboard accelerators	46
Supporting platform-specific menus	48
Configuring the application name in the menu	49
Hiding menu items	51
Sending messages between processes	53
Introducing editor-event	54
Sending confirmation messages to the main process	54

Sending messages to the renderer process	56
Wiring the toggle bold menu	58
Saving files to a local system	59
Using the save dialog	63
Loading files from a local system	68
Creating a file menu	72
Adding drag and drop support	74
Supporting automatic updates	77
Testing automatic updates	85
Changing the title of the application	88
Summary	89
Chapter 3: Integrating with Angular, React, and Vue	90
Technical requirements	91
Building an Electron application with Angular	91
Generating our Angular project scaffold	92
Integrating the Angular project with Electron	95
Configuring Live Reloading	99
Why test in the browser?	102
Setting up production builds	102
Setting up conditional loading	104
Using Angular Material components	106
Modifications made by installing Angular Material	107
Adding the Material Toolbar component	108
Angular routing	110
Building an Electron application with React	117
Generating a React project	118
Live reloading	122
Setting up production builds	125
Setting up conditional loading	127
Using the Blueprint UI toolkit	128
Adding an application menu	129
Adding routing	131
Final touches	133
Building an Electron application with Vue.js	135
Creating a Vue configuration file	139
Live reloading	141
Production builds	143
Setting up conditional loading	144
Adding routing	146
Configuring Vue Material	148
Creating an application toolbar	149
Summary	152
Chapter 4: Building a Screenshot Snipping Tool	153
Technical requirements	154

Preparing the project	154
Configuring frameless windows	155
Additional options for macOS	158
Using the hidden titleBarStyle	158
Using the hiddenInset titleBarStyle	159
Using the customButtonsOnHover titleBarStyle	159
Transparent windows	160
Making application windows draggable	165
Adding a snip toolbar button	166
Using the desktopCapturer API	167
Calculating the primary display size	169
Generating and saving a thumbnail image	170
Resizing and cropping the image	172
Testing the application's behavior	175
Integrating with the system tray	176
Hiding the main application window on startup	178
Registering global keyboard shortcuts	179
Summary	181
Chapter 5: Making a 2D Game	182
Technical requirements	183
Configuring a game project	183
Running a Hello World example	187
Rendering background images	190
Preventing window resizing	192
Rendering a sprite	192
Scaling sprites	193
Handling keyboard input	195
Flipping sprites based on their direction	197
Controlling sprite coordinates	198
Controlling sprite speed	201
Summary	203
Chapter 6: Building a Music Player	204
Technical requirements	204
Creating a project scaffold	205
Exploring the music player component	207
Downloading music files	209
Providing basic player setup	212
Using AmplitudeJS elements	213
Implementing the global play button	214
Implementing the global pause button	215
Implementing the global play/pause button	216
Styling buttons	216
Exploring the playback control buttons	219

Stop button	220
Mute and unmute buttons	222
Volume buttons	224
Implementing a song progress bar	227
Displaying music metadata	229
Improving the user interface	234
Reviewing the final structure	237
Summary	240
Chapter 7: Analytics, Bug Tracking, and Licensing	241
Technical requirements	242
Understanding analytics and tracking	242
Creating your own solution or using an existing service	243
Creating your own analytics services	244
Using third-party analytics services	244
Using Nucleus for Electron applications	245
Creating a new Nucleus account	246
Creating a new project with tracking support	250
Installing the Nucleus Electron library	252
Inspecting real-time analytics data	254
Identifying users	259
Disabling tracking per user request	259
Verifying real-time user statistics	260
Supporting offline mode	261
Handling application updates	262
Loading global server settings	264
License checking and policies	266
Creating a new policy and license	267
Checking licenses in the application	271
Summary	272
Chapter 8: Building a Group Chat Application with Firebase	273
Technical requirements	274
Creating an Angular project	275
Configuring the Electron Shell	277
Creating a Firebase account	280
Creating a Firebase application	284
Configuring Angular Material components	287
Adding a Browser Animations module	287
Configuring the default theme	288
Adding the Material Icons library	288
Adding a navigation bar	288
Testing the application with the material toolbar	290
Building a login dialog	290

Implementing the Material interface	292
Supporting error handling	294
Preparing the chat component placeholder	295
Connecting the login dialog to Firebase Authentication	297
Enabling the sign-in provider	297
Creating demo accounts	300
Integrating the Login dialog with Firebase	302
Configuring the Realtime Database	306
Creating demo groups	308
Rendering the group list	310
Testing real-time updates	312
Implementing the group messages page	314
Displaying group messages	316
Improving query performance	320
Sending group messages	321
Updating the message list interface	324
Ideas for further enhancements	325
Verifying the Electron Shell	325
Summary	326
Chapter 9: Building an eBook Editor and Generator	327
Technical requirements	328
Creating the project structure	328
Generating a new React application	328
Installing the editor component	330
Testing the web application	335
Integrating with the Electron shell	337
Updating the code to use React Hooks	339
Controlling keyboard shortcuts	340
Loading files	342
Saving files	345
Integrating with the application menu	348
Setting up the book generator	351
Installing Docker	352
Running the Pandoc container	356
Sending documents to the main (Node.js) process	359
Invoking Docker commands from Electron	362
Sending the markdown text to the Node.js process	362
Saving the markdown text to the local drive	363
Generating PDF books	366
Generating ePub books	369
Summary	372
Chapter 10: Building a Digital Wallet for Desktops	373
Technical requirements	374

Generating the project scaffold with React	374
Integrating the Ant Design library	376
Setting up a personal Ethereum blockchain	379
Configuring the Ethereum JavaScript API	384
Displaying Ethereum Node information	386
Getting node information	386
Rendering node information in the header	387
Integrating with the application menu	389
Rendering a list of accounts	390
Showing our account balance	395
Transferring Ether to another account	397
Packaging the application for distribution	404
Summary	407
Other Books You May Enjoy	408
Index	411

Preface

The goal of this book is to provide you with practical experience and guide you through setting up, configuring, building, and distributing Electron applications. You are going to build multiple projects, address common challenges and pitfalls, and integrate with modern JavaScript frameworks and underlying toolchains.

Who this book is for

The target audience of this book is beginner or experienced web developers. Readers should have a basic understanding of HTML, CSS, and JavaScript. Familiarity with one of the modern web frameworks and libraries such as React, Angular and Vue.js is recommended.

No prior knowledge of desktop development is required to get started.

What this book covers

Chapter 1, *Building Your First Electron Application*, prepares the environment and gets you started with Electron development.

Chapter 2, *Building a Markdown Editor*, gets you familiar with the main building blocks of the typical Electron-based application.

Chapter 3, *Integrating with Angular, React, and Vue*, covers modern frontend Javascript frameworks, such as Angular, React.js, and Vue.js, and explains how to integrate them with Electron apps to build cross-platform desktop applications that can share their code base with their website counterparts.

Chapter 4, *Building a Screenshot Snipping Tool*, covers working with the native image capturing API in Electron, system tray integration, and keyboard handling.

Chapter 5, *Making a 2D Game*, covers integrating a JavaScript-based game engine and handling game loops, loading external resources, and practicing communication between Main and Renderer processes.

Chapter 6, *Building a Music Player*, covers building a simple desktop music player with playlist support and custom album art.

Chapter 7, *Analytics, Bug Tracking, and Licensing*, gives essential information for developers that want to monitor Electron application in production, track errors and crashes, analyze a real-time user base, and more.

Chapter 8, *Building a Group Chat Application with Firebase*, covers creating an Electron application with group chat features, integrating with Google Firebase services for mobile apps, configuring Google Authentication, and storing application data in the cloud.

Chapter 9, *Building an eBook Editor and Generator*, covers creating a simple cross-platform book editor, utilizing Docker to generate PDF and ePub books, and previewing the resulting PDF files in the separate Electron windows.

Chapter 10, *Building a Digital Wallet for Desktops*, covers developing a simple digital wallet application that integrates with external services and connecting to locally running servers.

Release cycle note

Please note that we are using Electron 7 in the book. The Electron project has moved to a 12-week release cadence starting on May 13, 2019. You can see the official announcement on the Electron blog post at <https://electronjs.org/blog/12-week-cadence>.

Having shorter release cycles means you are getting features, bug fixes, and security fixes faster. However, that also means there will most probably be newer versions of the Electron released once this book is out.

The good news is the Electron team is going to support the last three major versions. Be sure to check the schedule and more details by visiting <https://electronjs.org/docs/tutorial/support#supported-versions>. Besides that, it is relatively easy to upgrade your application project to the latest version of the Electron. You can do that by running the following command:

```
npm install electron@latest
```

It is highly recommended to watch the Electron team blog for more details on each release: <https://electronjs.org/blog>.

To get the most out of this book

It is recommended that readers get familiar with the Node.js runtime (<https://nodejs.org/en/>) and basic commands such as `npm install`.

Readers who prefer the Angular ecosystem may want to read the details on the Angular CLI (<https://cli.angular.io/>) and its commands.

For the React development, getting familiar with the Create React App (<https://github.com/facebook/create-react-app>) tool is recommended.

If you want to use the Vue.js framework for Electron development, the Vue CLI (<https://cli.vuejs.org/>) application documentation should give a lot of details and examples.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Electron-Projects>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Just type code or visual studio code into the search box—you should get the link to the corresponding package."

A block of code is set as follows:

```
win = new BrowserWindow({
  width: 800,
  height: 600,
  webPreferences: {
    nodeIntegration: true
  }
  frame: false
});
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
function createWindow() {
  win = new BrowserWindow({ titleBarStyle: 'hidden' });

  win.loadURL(`http://localhost:3000`);

  win.on('closed', () => {
    win = null;
  });
}
```

Any command-line input or output is written as follows:

```
git --version
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "If you click on the icon, you should get the **Quit** menu entry."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Building Your First Electron Application

The goal of this book is to provide you with practical experience and guide you through setting up, configuring, building, and distributing Electron applications. You are going to learn how to build multiple projects, address common challenges and pitfalls, and how to integrate with modern JavaScript frameworks and underlying toolchains.

In this chapter, you are going to get a brief overview of the Electron framework, its history, and its architecture. You will learn how to get all the prerequisites installed for multiple platforms, get your first Electron project up and running with Node.js and NPM, and learn how the apps are packaged for various platforms.

By the end of this chapter, you should have a basic project template so that you can complete future practical tasks.

In this chapter, we will cover the following topics:

- What is Electron?
- Preparing a development environment
- Creating a simple application
- Packaging for multiple platforms

Technical requirements

To get started with Electron application development, you will need a standard laptop or desktop that's running macOS, Windows, or Linux.

Before we dive into Electron development, we need to prepare the prerequisites for your platform of choice. We are going to focus on all the major platforms, that is, macOS, Ubuntu (Linux), and Windows.

To get started, we need the following software:

- **Git**, a version control system
- **Node.js** with **Node Package Manager (NPM)**
- **Visual Studio Code**, a free and open-source code editor

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter01>.

What is Electron?

Electron is an open-source framework for building cross-platform applications with the modern web technology stack: HTML, CSS, and JavaScript.

It is developed and maintained by GitHub Inc. and has had an active community of contributors since it appeared and was released on July 15, 2013 (its first commit appeared in April 2013) as part of the Atom editor, a free and open-source code editor for Linux, Windows, and macOS. Initially, it was called Atom Shell until GitHub renamed it Electron and started shipping it as a separate project.

The secret sauce of Electron is a combination of Chromium, an open-source project behind the Google Chrome browser and Google Chrome OS, and Node.js, a JavaScript runtime built on Chrome's V8 JavaScript engine.

Electron uses Chromium for the frontend and Node.js for the backend. It provides a rich set of **application programming interfaces (APIs)** that allow developers to build cross-platform applications that share the same HTML, CSS, and JavaScript code. Also, Electron provides us with access to operating system resources and specific platform features and supports thousands of JavaScript libraries and utilities that you can use with the Node.js portion of the application.

Since its release, the Electron framework has won the hearts of all web and desktop developers. Many popular applications have been built with Electron that you may have used in the past or are using daily, such as Skype, Slack, WhatsApp, Discord, Signal, Visual Studio Code, Microsoft Teams, Keybase, and many others. Please check out the official list of Electron-based applications that are featured online at <https://electronjs.org/apps>—it has more than 700 entries and counting.

Preparing a development environment

In this section, we are going to look at the setup process for each operating system. You can skip the sections that do not correspond to your current platform. Note, however, that you may still need to have multiple operating systems available in case you ever want to test how the packaging and deployment of your applications work across all platforms.

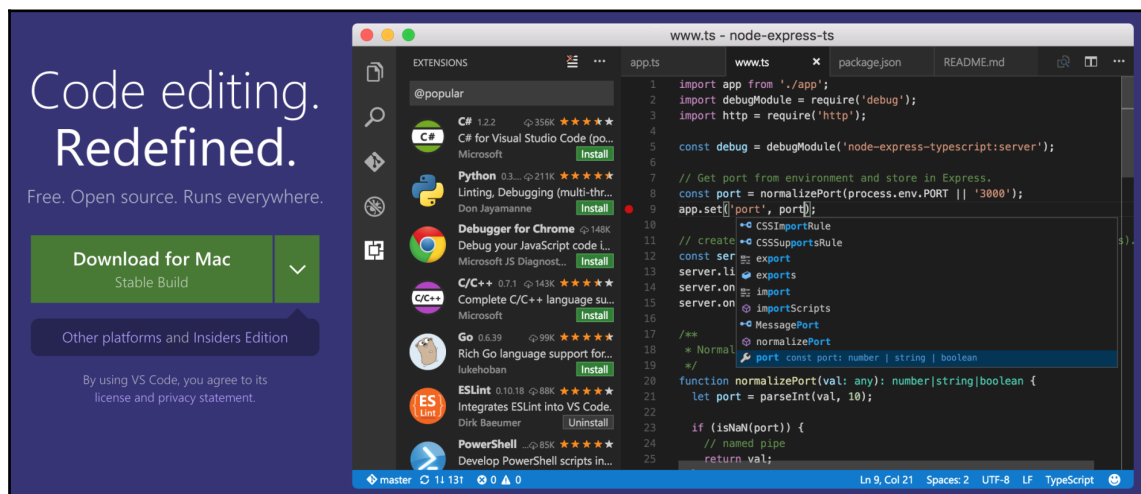
Keep in mind that most of the code for the application projects in this book are universal; the blocks or steps that are different for a particular system will be highlighted and explained.

Installing Visual Studio Code

We are going to be using Visual Studio Code for all the projects and examples in this book. It's a free, open-source, and cross-platform code editor based on Electron. However, feel free to use Atom, Sublime, Vim, or any other editor of your choice.

Setting up Visual Studio Code is very simple and, thanks to Electron's support, the process doesn't differ much across platforms. Let's get started:

1. Navigate to <https://code.visualstudio.com/>. An installation package will be suggested to you regarding your current operating system. It is also possible to choose from a list of available distributions:



2. Click the big **Download** button to get a `.dmg` installer for macOS, an `.msi` file for Windows, or a `.deb` package for Debian-based Linux distributions.
3. Run the corresponding file and follow the on screen instructions. You don't need to customize anything during the setup process.

If you are using or plan to use Ubuntu Linux for development purposes, the process is slightly easier than it is for other operating systems.

Installing Visual Studio Code for Ubuntu

If you are using Ubuntu Linux as your primary development machine, you can download Visual Studio Code from the Ubuntu Software Center. Just type `code` or `visual studio code` into the search box—you should get the link to the corresponding package.



Note that there is also an Insiders Version for Visual Studio Code. This version is updated daily and is for experienced developers that want to see the latest features. If you are getting started with Visual Studio Code, then it is better to stick to the regular version as it is usually more stable than the Insiders edition.

Setting up the environment for macOS

This section describes how to install and configure the required software for macOS. Feel free to skip this section if you are using a Linux or Windows platform.

Installing Git on macOS

Git comes preinstalled with all macOS versions. To verify this, launch the Terminal application and run the following command:

```
git --version
```

The output should be similar to the following:

```
git version 2.17.2 (Apple Git-113)
```



Please note that it isn't critical if your system's version of Git doesn't match the one in this example.

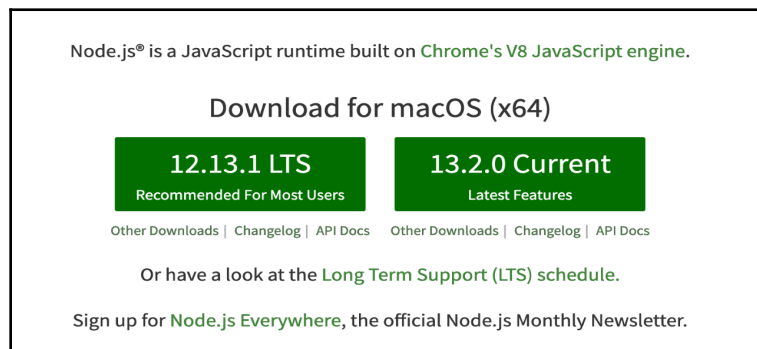
Installing Node.js on macOS

Next, let's install Node.js and NPM using the following steps. You can find the necessary installation packages by navigating to <https://nodejs.org>:



Note that Node.js typically comes in two flavors: the **Long-Term Support (LTS)** version, which is suitable for most users, and the **Current** version, which provides the most cutting-edge features and enhancements.

1. First, we need to download and install Node.js. The website automatically detects your browser and platform and suggests the appropriate packages for you to download. For macOS, you are going to see the **Download for macOS (x64)** label and two big download buttons, as shown in the following screenshot:



2. Choose any version and click the corresponding button to get the relevant installer package. The installation process for the macOS platform is pretty straightforward. Keep all the default settings; proceed with the wizard's steps until the setup is over.
3. In the Terminal application, run the following commands to verify that you have Node.js and NPM installed on your machine:

```
node -v  
npm -v
```

4. The system's output should be similar to the following, though their versions may vary:

```
v12.13.0  
6.12.1
```

Congratulations! You've successfully installed Node.js on your macOS.

Setting up the environment for Ubuntu Linux

In this section, we are going to use the latest Ubuntu desktop version, 18.10, though previous LTS versions should also run fine. You can skip this section and jump to the Windows or macOS setup section if you don't use Linux. However, you may find this section useful once you start testing cross-platform deployment for your Electron applications.

Installing Git on Ubuntu

You can check whether you have Git installed by running the following command:

```
git --version
```

Typically, Git is not present on fresh installations of Ubuntu. To get it, run the following command:

```
sudo apt install -y git
```



Please note that you need to enter the administrator password to proceed.

Installing Node.js on Ubuntu

Ubuntu usually doesn't ship with the Node.js and NPM tools out of the box. You need to install them separately.

To install Node.js, follow these steps:

1. Run the following command:

```
sudo apt install -y nodejs
```

2. Now, we need to verify that Node.js has been installed. You can check the version that you've installed in the Terminal application by using the following command:

```
node --version
```

The system's output, which will be the version's value, will be **v8.11.4** or higher.

To install NPM, follow these steps:

1. Use the following command:

```
sudo apt install -y npm
```

2. The fastest way to check that NPM has been installed is to check its version. You can do so by using the following command:

```
npm --version
```

The version number should be **5.8.0** or higher.

Setting up the environment for Windows

In this section, we are going to walk through the installation process for Windows 10.

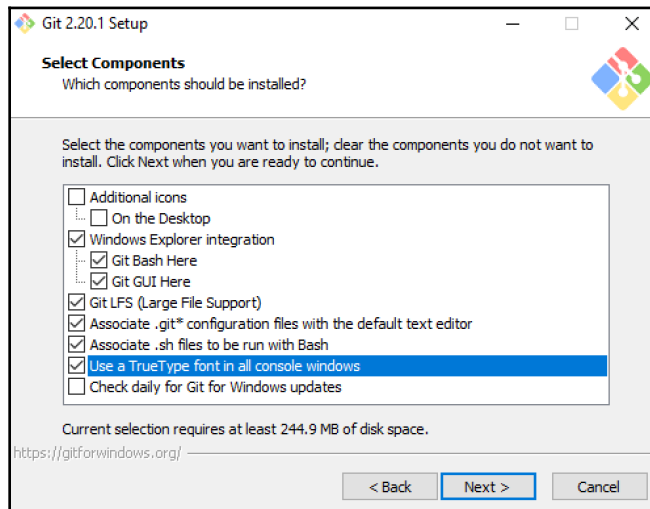
Installing Git on Windows

I recommend installing Git after Visual Studio Code because the Git setup wizard allows you to enable integration between the two.

The process of installing Git on Windows 10 is slightly different compared to macOS and Ubuntu. Follow these steps to install it:

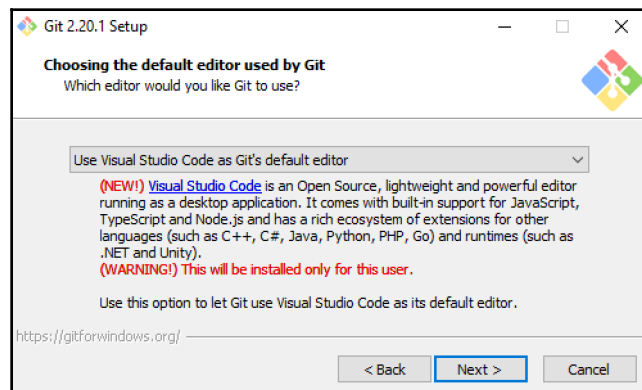
1. Navigate to <https://git-scm.com>. The website will detect your platform and suggest a proper distribution.
2. Click the button that says **Download 2.20.1 for Windows**, wait for the file to download, and run the installer package.
3. The Git installer for Windows sets all the respective defaults for you. Just proceed with the questions until you reach the **Select Components** dialog.

- I suggest checking the **Use a TrueType font in all console windows** option, as shown in the following screenshot:



This is optional, but it helps improve readability slightly.

- Follow the installation steps and use the predefined settings until you get to the **Choosing the default editor used by Git** dialog.
- If you have already installed Visual Studio Code, I strongly recommend selecting the **Use Visual Studio Code as Git's default editor** option from the drop-down list, as shown in the following screenshot:



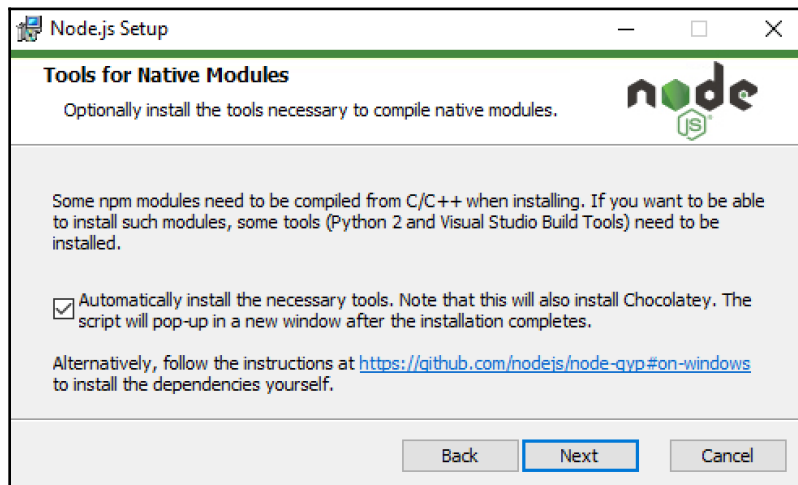
- Accept all the defaults in the subsequent dialogs until the setup is over.

Installing Node.js on Windows

Once Visual Studio Code and Git are ready, we can install Node.js and NPM on Windows:

1. Navigate to <https://nodejs.org/en> and get the corresponding installer. Note that the website detects your platform for you and suggests the corresponding installer package. For Windows, you are going to see the **Download for Windows (x64)** label and two buttons where you can select either the LTS, that is, the stable LTS version, or a current one, with the most recent cutting-edge features.
2. Download and run the installation file. Proceed with the setup wizard and use the default settings—these are usually pretty reasonable.

Optionally, in the **Tools for Native Modules** dialog, you can allow the automatic installation of a set of the tools so that you can compile native modules. This is shown in the following screenshot:



The option for the native module instructs the setup wizard to download and configure all the necessary tooling after the installation of Node.js is over.



Please note that the extra tools require about 3 GB of additional space on your disk and may take a few minutes to install. However, I recommend installing those tools as you may come across third-party modules and libraries for the system's integration that require those tools.

Alternatively, you can always download the latest copy of the Node.js installer again, go through the setup wizard steps, and check the **Tools for Native Modules** option if you forgot to do that previously.

Verifying the installation

Launch the Command Prompt utility and execute the following two commands to ensure that both Node.js and NPM are present on your machine:

```
node --version
npm --version
```

You should receive the following system output:

```
v12.13.0
6.12.1
```



Please note that your versions may vary, depending on the last published packages you downloaded. At the time of writing, it is essential to get any output for each command to prove the tool is there, rather than a version value.

In this section, we covered the installation of Node.js and NPM for the Windows, macOS, and Linux systems so that you can start creating a simple application project. In the next section, we are going to walk through the minimal configuration process to help you get started.

Creating a simple application

Let's walk through a typical *hello world* application with Electron, package it, and see it running on all platforms. Let's get started:

1. Somewhere in your projects folder, create a new directory called `my-first-app` and navigate to it, as shown in the following code:

```
mkdir my-first-app
cd my-first-app
```

2. Now, we need to initialize our new project with the NPM tool by using the following command:

```
npm init
```



The tool asks a series of questions, such as the name of the project, a user-friendly description, version, author information, and a license. Feel free to enter any details you want. For our first example project, you can specify any values you want.

Next time, if you want to set up quickly with a single command and with reasonable defaults, you can use the same command with the `-y` switch. This switch, as shown in the following code, instructs NPM to accept all the questions and use predefined values:

```
npm init -y
```

This tool generates the following file, called `package.json`:

```
{
  "name": "my-first-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Feel free to update the description, author, license, or other fields as necessary.

Please note that the value of the `main` field is `index.js`. This means that the primary entry point for NPM commands and your Electron app is going to be that file. We are going to create it shortly, but, first, let's install the Electron framework library for our project:

1. Run the following command:

```
npm i -D electron
```

2. If you take a look at the `package.json` file, you may notice a new section called `devDependencies`, which has an Electron library. Its version may vary, depending on Electron's release frequency:

```
{
  "devDependencies": {
    "electron": "^7.0.0"
  }
}
```

3. Now, it's time to get back to the `index.js` file. Create it in the root folder of your project, next to the `package.json` file.
4. Let's take a closer look at the minimum code you need to run an Electron window. The following snippet demonstrates the steps we have to perform in the `index.js` file:

```
// 1. import electron objects
const { app, BrowserWindow } = require('electron');

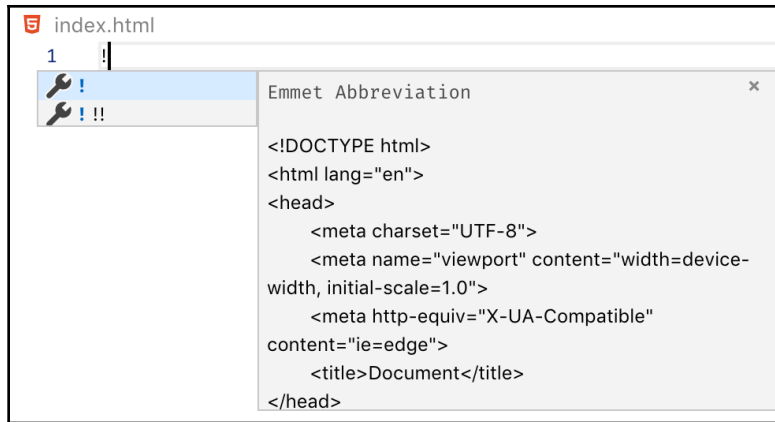
// 2. reserve a reference to window object
let window;

// 3. wait till application started
app.on('ready', () => {
  // 4. create a new window
  window = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    }
  });

  // 5. load window content
  window.loadFile('index.html');
});
```

First, you need to import the required objects and classes from the `electron` namespace. Then, reserve a reference to an object or the `BrowserWindow` type that you are going to instantiate and display to your users. After that, you need to wait for the application to become ready and create a small window that's **800 x 600** in size. Finally, load and display the content of the `index.html` file, which contains the main content of your Electron application.

5. Now, we need to define the main application's content in the form of an HTML page. Create a new `index.html` file next to the `package.json` and `index.js` files. With Visual Studio Code, it is effortless to generate an initial web page. Just type an exclamation mark, `!`—the code editor will auto suggest a template for you to use:



6. Press the *Tab* or *Enter* key. Visual Studio Code will generate and fill in the HTML page's content in the place of your cursor. It even moves the cursor inside the body element so that you can continue working on the 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="Content-Security-Policy"
      content="script-src 'self' 'unsafe-inline';"
    />
    <title>Document</title>
  </head>
  <body>
  </body>
</html>
```

7. Let's put the traditional Hello World example between the body tags so that we can inspect the Electron component's versions:

```
<h1>Hello World!</h1>
We are using node <script>document.write(process.versions.node)
</script>,
Chrome <script>document.write(process.versions.chrome)
</script>,
and Electron <script>document.write(process.versions.electron)
</script>.
```

We are going to see the versions of Node.js that are powering our application, our embedded version of Chrome, and, of course, the Electron library version.

8. The final step of the initial setup is to update the package scripts. Update the `package.json` file and add the `start` entry for the `scripts` section to invoke the Electron binary app against our project folder, as shown in the following code:

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

To launch, develop, and test the application, we only need to run `npm start` from the command line. If you need to add more parameters, you can always update the script once again—there's no need to memorize long commands.

9. The content of your `package.json` file should look like this:

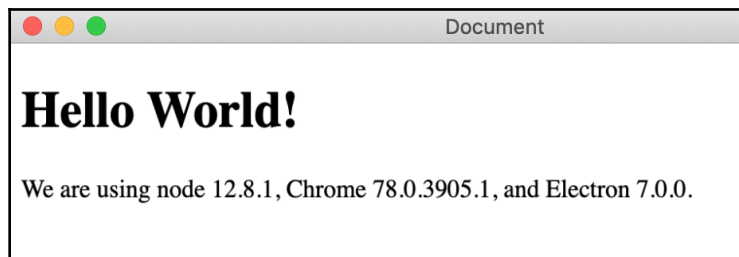
```
{  
  "name": "my-first-app",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "electron ."  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "electron": "^7.0.0"  
  }  
}
```

You are now ready to launch your first Electron application. Let's get started:

1. In the application menu of Visual Studio Code, select **View** and then **Terminal** to access the embedded Terminal tool.
2. Run the `start` command, as shown in the following code:

```
npm start
```

Congratulations! You now have your first Electron application up and running:



If you want to stop the application, press `Ctrl + C` in the Terminal window.

Now that we have a new application up and running, let's understand how our application will be packaged for different platforms.

Packaging for multiple platforms

Please note that *running on all platforms* does not necessarily mean that you can test and run all the installation packages on a single platform. This means you cannot, for example, launch Windows installers on Linux, or macOS installers on Windows. You may need to have access to either real machines with their respective platforms, or virtual machines, running with VirtualBox, Parallels, or any other modern virtualization software.

There are many community tools that you can use to build and package Electron applications for production. We are going to use `electron-builder` (<https://www.electron.build/>) for this purpose.

According to its documentation, `electron-builder` is as follows:

*A complete solution to package and build a ready for distribution—Electron app for macOS, Windows, and Linux with **auto update** support out of the box.*



The list of supported features is outstanding; it is recommended that you take a look at the documentation of `electron-builder` if you want to find out more.

With this tool, for example, you can create distribution packages for all platforms when developing only on macOS, or any other platform.

Before we continue, let's install it for our project with the following command:

```
npm i -D electron-builder
```

Now, let's look at how we can set up the packaging scripts, depending on your target platform. We are going to package our Electron application for macOS, Ubuntu Linux, and Windows 10 with a minimal set of configuration parameters.

Packaging for macOS

If you intend to publish our application to the App Store, you should provide an application ID and category settings. Open the project's `package.json` file for editing, and append the following section to the end of the file:

```
{
  "build": {
    "appId": "com.my.app.id",
    "mac": {
      "category": "public.app-category.utilities"
    }
  }
}
```

Feel free to customize the values and provide the relevant information later. For now, you can leave those values as they are.

There are two ways you can build your application: through the development and production modes. Let's start with the development script, which allows you to quickly run and see that your application is working as expected:

1. Update the `package.json` file and add the `build:macos` entry to the `scripts` section, as shown in the following code:

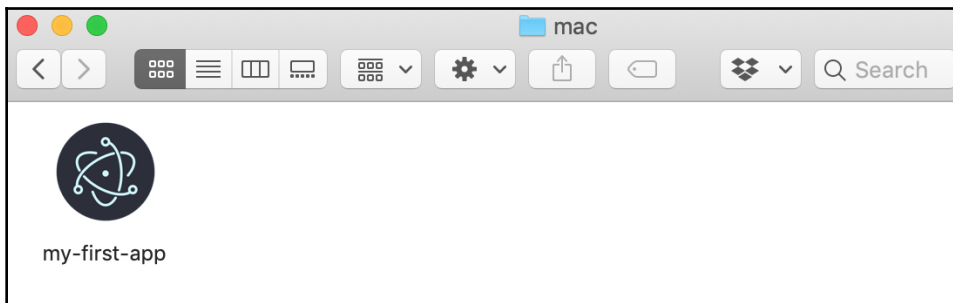
```
{
  "scripts": {
    "start": "electron .",
    "build:macos": "electron-builder --macos --dir"
  }
}
```

Just like the `npm start` command we used earlier, you can customize all the parameters in a single place. You only need to remember and document a simple command `npm run build:macos`.

2. To build the application for development, open the Terminal window in VS Code and run the `build:macos` script, as follows:

```
npm run build:macos
```

3. After a few seconds, you will see the build's output in the `dist/mac` folder:

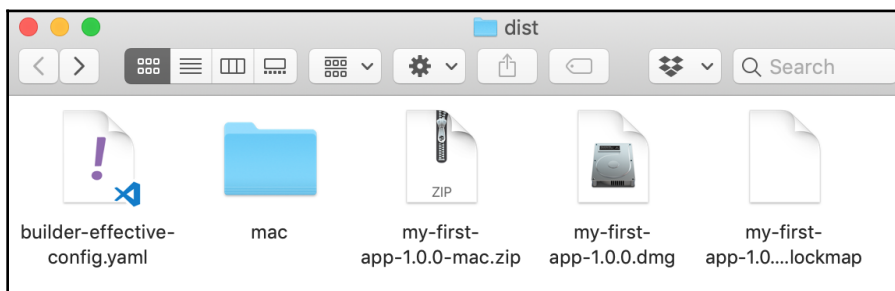


4. Double-click on the icon to run your simple Electron application locally.
5. Let's also add the necessary script so that we can create production or distribution packages. Append the `dist:macos` entry to the `scripts` section, as shown in the following code:

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

Now, you have two scripts that handle running and packaging on your macOS machine.

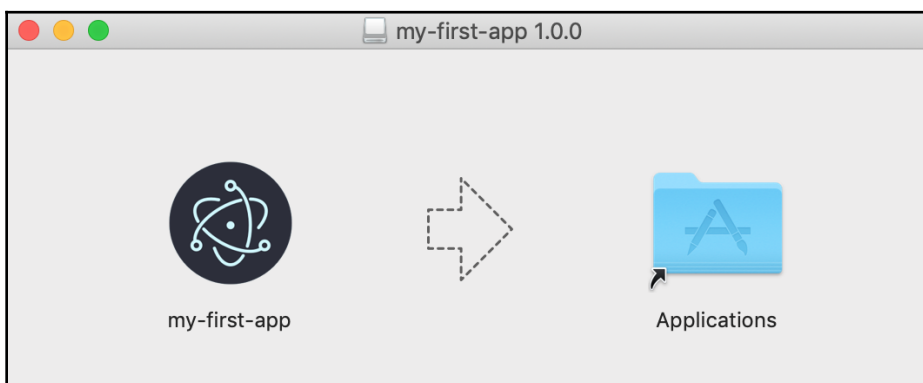
Running the `dist:macos` script takes a bit longer than the `build:macos` one. After running the script, you get several different packages in the `dist` folder of your project: `my-first-app-1.0.0.dmg`, a typical macOS installer; `my-first-app-1.0.0-mac.zip`, an archived installer so that you can distribute it easily; and, of course, `mac/my-first-app`, which includes the ready-to-launch application:



Try running the `.dmg` file; you should see the typical macOS installer:



Please refer to the electron-builder documentation for ideas and tips on how to customize it: <https://www.electron.build/configuration/dmg>.



Congratulations—you've got your first cross-platform Electron application installer up and running on macOS!

Packaging for Ubuntu

The process of packaging your application for Ubuntu doesn't differ much from that of macOS. Let's get started:

1. You need to provide an application identifier and a category in the `linux` section of the `package.json` file:

```
{
  "build": {
    "appId": "com.my.app.id",
    "linux": {
      "category": "Utility"
    }
  }
}
```



Please note that you can declare settings for Linux alongside those for other platforms, which is handy when you're developing for multiple operating systems or switching between them. The same applies to the scripts section. In this book, we are going to use different script names so that you can merge them into a single configuration.

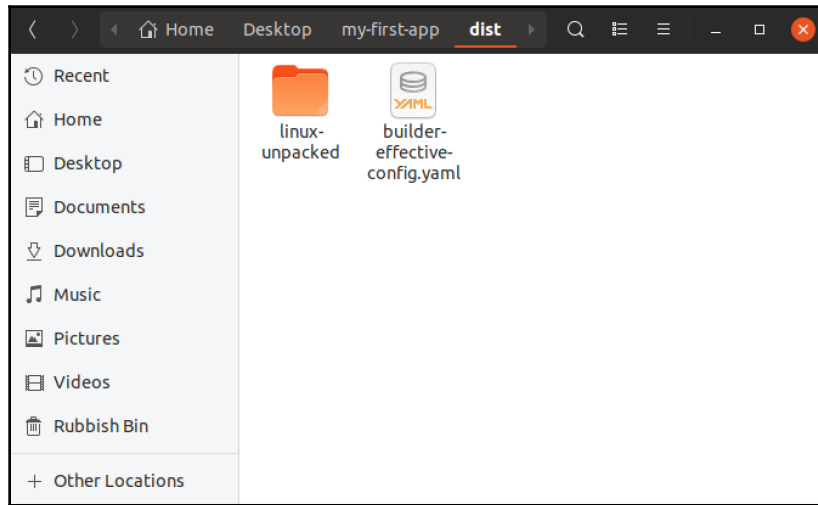
2. Update your `package.json` file and append the following scripts to it so that you can build your application in development mode and distribution mode:

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

3. Let's ensure that we can build the application for local testing. Run the first script in the Terminal window:

```
npm run build:linux
```

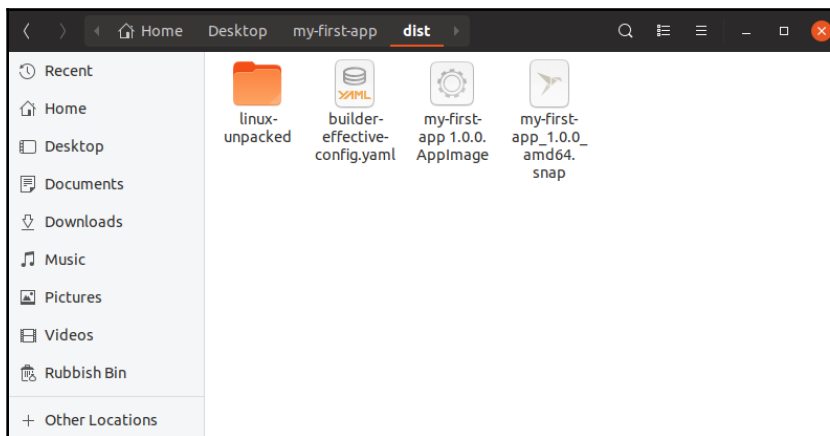
4. In the project's root, you should see the `dist/linux-unpacked` folder, which contains several build artifacts:



5. Now, let's see what you get when you're building packages for distribution. Run the second command, as shown in the following code:

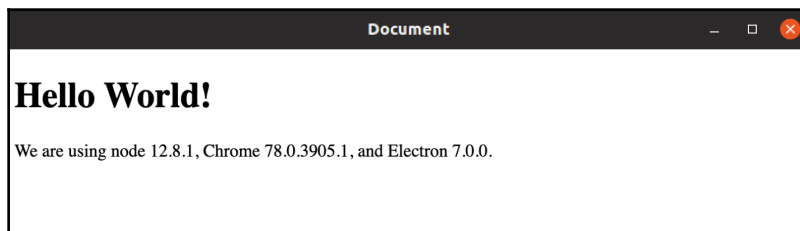
```
npm run dist:linux
```

6. This time, you are going to get multiple packages in the `dist` folder, as shown in the following screenshot:



The files that will be in your output folder are as follows:

- `my-first-app 1.0.0.AppImage`: The AppImage format is a universal software packaging format for all GNU/Linux distros.
 - `my-first-app_1.0.0_amd64.snap`: This is a snap file, another popular format for sandboxed applications.
 - `linux-unpacked/my-first-app`: This is the unpacked build for local testing and custom distributions.
7. For now, double-click on `my-first-app 1.0.0.AppImage` to run the app. If you get the **Would you like to integrate my-first-app with your system** dialog, click **No**.
 8. This will be your final output:



Congratulations—you've got your first cross-platform Electron application package up and running on Ubuntu Linux!

Packaging for Windows

Now that you know how to set up build scripts for macOS and Ubuntu Linux, configuring for Windows shouldn't be a problem for you.

As I mentioned earlier, it is possible and also recommended to keep the configuration files for all platforms in a single code repository, inside the `package.json` file. The build scripts for Windows are shown in the following code:

```
{
  "scripts": {
    "start": "electron .",
    "build:windows": "electron-builder --win --dir",
    "dist:windows": "electron-builder --win"
  }
}
```

Both scripts should be familiar to you. The `build:windows` script creates an unpacked local build for development and testing purposes, while the `dist:windows` script prepares the application for distribution.

Let's try to build and run the development version of the application:

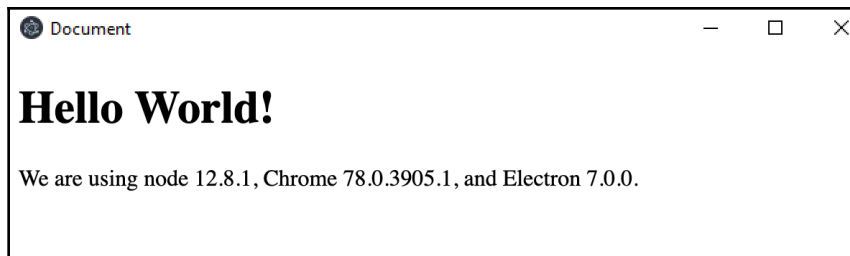
1. Open the Terminal window in Visual Studio Code, or a Command Prompt tool, and run the following script:

```
npm run build:windows
```



Note that you can build Windows packages with macOS or Ubuntu Linux if you have the Wine tool installed, but I recommend having a virtual machine nearby for testing purposes. It should also be possible to build for Linux on a Windows machine, but you may want to have a real Linux machine for application testing purposes.

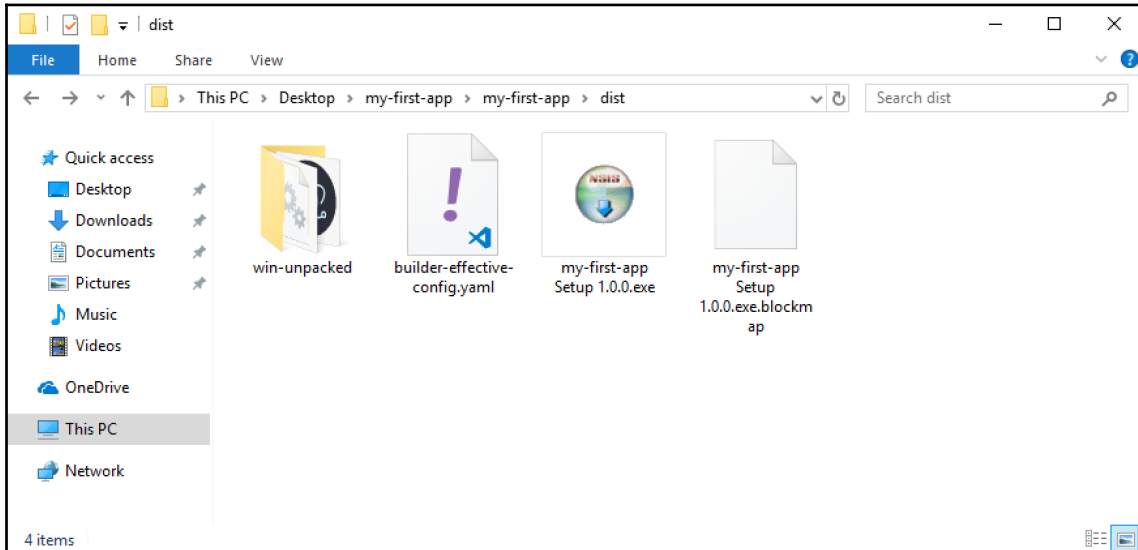
2. Once the script exits, you should see the prebuilt application, that is, `my-first-app.exe`, in the `dist/win-unpacked` folder.
3. Double-click on the `my-first-app.exe` file to run the application:



4. We need to use the `build:windows` script to create a distribution package for testing purposes. Let's check that we can build packages for redistribution:

```
npm run dist:windows
```

5. Check the `dist` folder once again. You should see the `my-first-app Setup 1.0.0.exe` installer file alongside the `win-unpacked` folder:



6. Now, double-click the installer file. The setup wizard should set up the application and automatically launch it.

Congratulations—you've got your first cross-platform Electron application package up and running on Windows 10!

Summary

In this chapter, we walked through a brief history of Electron and learned how to configure a development environment on popular platforms such as macOS, Windows, and Ubuntu Linux. You also looked at the various configuration options you can implement for your Electron applications so that you can build, distribute, and run them on the corresponding platforms.

As you can see, not just the applications that are built with the Electron framework are cross-platform—the development process is nearly identical too, thanks to Node.js and NPM. You can work on a single platform and even build distribution packages for other platforms, though you usually need to have access to real or virtual machines to run and test apps.

In the next chapter, we are going to focus on application development and our first project implementation. We are going to build a markdown editor project so that you can understand how a web application can be integrated with the desktop shell.

2

Building a Markdown Editor

In this chapter, we are going to build a minimal Markdown Editor application. This mini exercise is going to help you get an idea of how to build a web application that integrates with the Electron shell on desktops.

You are about to walk through the process of integrating a third-party editor component, learning how to support application menus, and establishing communication channels between the rendering (browser) and the main (Node.js) processes. We are doing this so that you become confident with Electron and can build more complex projects.

As part of this chapter, we will also create a new GitHub repository to store application releases, publish multiple versions of the Markdown Editor to GitHub, configure automatic updates, and see them in action.

In this chapter, we will cover the following topics:

- Configuring a new project
- Integrating the editor component
- Fitting the screen size
- Integrating the application menu
- Adding drag and drop support
- Supporting automatic updates
- Changing the title of the application

Technical requirements

To get started with this chapter, you will need a standard laptop or desktop running macOS, Windows, or Linux.

The software that you need to have installed for this chapter is as follows:

- Git, a version control system
- Node.js with **node package manager (NPM)**
- Visual Studio Code, a free and open-source code editor

You can find the code files for this chapter in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter02>.

Configuring a new project

Let's start our journey by configuring a new Electron project and naming it `markdown-editor` since we are building a markdown editor application. You can create a corresponding folder with the following commands:

```
mkdir markdown-editor
cd markdown-editor
```

As you may recall from *Chapter 1, Building Your First Electron Application*, we need to initialize a new project with the `npm init` command. You should also install `electron`, the core library that provides an application shell. In addition, your project needs an `electron-builder` library, which allows you to publish and distribute features for multiple platforms. Let's get started:

1. Run the following commands to set up a new project:

```
npm init -y
npm i -D electron
npm i -D electron-builder
```

The `npm init` command should generate a `package.json` file with the following content:

```
{
  "name": "markdown-editor",
  "version": "1.0.0",
  "main": "index.js",
  "devDependencies": {
    "electron": "^7.0.0",
    "electron-builder": "^21.2.0"
  }
}
```

The `-D` switch means that the libraries should be installed in the `devDependencies` section.

2. Now, create an `index.js` file with a bare minimum amount of JavaScript code in it so that you can run an empty application:

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

let window;

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

3. Finally, create the `index.html` file, which is the HTML template for our new project, next to `index.js`. For now, just put in a dummy string as the content for the body element:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
  <meta
    http-equiv="Content-Security-Policy"
    content="script-src 'self' 'unsafe-inline';"
  />
  <title>Document</title>
</head>
<body>
  <h1>Editor</h1>
</body>
</html>
```

At this point in time, our focus is on quickly setting up a project structure that we can turn into a markdown editor application.

4. The last piece of the puzzle lies in supporting the `npm start` script so that we can run and test our application without having to know all the command parameters and switches off by heart. Let's update the `package.json` file and extend the `scripts` section, as shown in the following code:

```
{
  "name": "markdown-editor",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "electron ."
  },
  "devDependencies": {
    "electron": "^7.0.0",
    "electron-builder": "^21.2.0"
  }
}
```



Note that your versions of the libraries may vary.

5. We are ready to create our Electron application. For all the updates to the files that we are going to create throughout this chapter, the testing process will run the following command:

`npm start`

Press `Ctrl + C` to stop the running application.

We will look at more project configuration options and live reloading in Chapter 3, *Integrating with Angular, React, and Vue*. For the time being, you should just be stopping the application with `Ctrl + C` and starting it with the `npm start` or `npm run start` command every time you change the code and want to see it live.

Now that the project is up and running, let's switch to the user interface and integrate the editor component with our Electron application.

Integrating the editor component

For our project, we don't need to build everything from scratch, including the components that edit and format text in markdown format. There are lots of free, open source components you can use to save time and focus on building the application and delivering value to your users, rather than spending months reinventing the wheel.

For the sake of simplicity, we are going to use the `SimpleMDE` component, which stands for **Simple Markdown Editor**. You can find more details about the project on its home page: <https://simplemde.com/>. The project is open source and has an MIT license. Follow these steps to incorporate the component:

1. Similarly to how we installed the Electron framework itself, you can use NPM commands to get `SimpleMDE` into your project:

```
npm install simplemde
```



Don't forget to stop the application before installing a new library.

Like any other typical JavaScript component, the `SimpleMDE` component comes with a JavaScript file and a CSS stylesheet so that we can integrate with the web page.

2. Append the following lines to the bottom of the `head` block in the `index.html` file:

```
<head>
  <link rel="stylesheet" href="./node_modules/simplemde/dist/
                                simplemde.min.css">
  <script src="./node_modules/simplemde/dist/
                                simplemde.min.js"></script>
</head>
```

Note how we are referencing the `node_modules` folder from the `index.html` page. Now, to update the `SimpleMDE` component to a newer version, you just need to run `npm install simplemde` once again. There's no need to update the web page each time; as soon as you build and run the application, it will use the updated libraries.

3. Now, let's run the newly installed component inside the Electron shell. According to the component's requirements, we need an empty `textarea` element defined on the page and a script block that turns that element into a markdown editor at runtime. Take a look at the following code, which shows a basic implementation of this:

```
<textarea id="editor"></textarea>

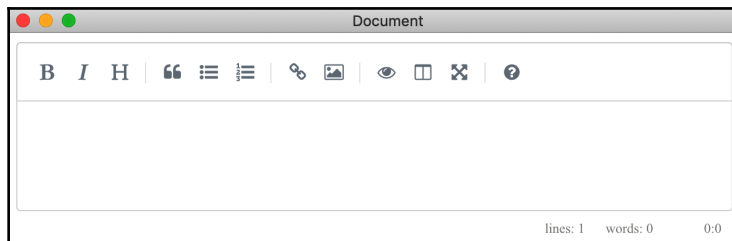
<script>
  var editor = new SimpleMDE({
    element: document.getElementById('editor')
  });
</script>
```

4. At this point, the content of the HTML page for your application should look as follows:

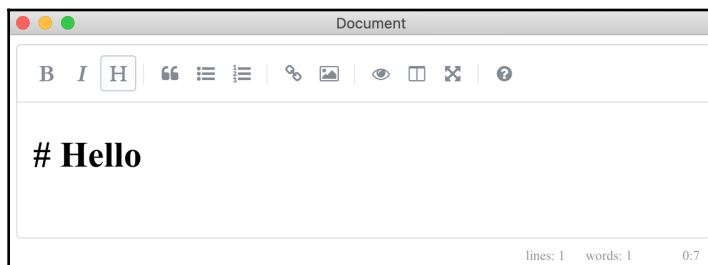
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
  <meta
    http-equiv="Content-Security-Policy"
    content="script-src 'self' 'unsafe-inline';"
  />
  <title>Document</title>
  <link rel="stylesheet" href="./node_modules/simplemde
    /dist/simplemde.min.css">
  <script src="./node_modules/simplemde/dist
    /simplemde.min.js"></script>
</head>
<body>
  <textarea id="editor"></textarea>
  <script>
    var editor = new SimpleMDE({
      element: document.getElementById('editor')
    });
  </script>
</body>
</html>
```

5. Save your changes and run the application.

6. You should see a window with the markdown editor component in the middle, a toolbar with a set of default buttons used to format the text, and word and line counter labels at the bottom:



The SimpleMDE component provides a nice live formatting feature so that your end users can see the final formatting alongside the markdown syntax. To see that in action, type something into the editor and press the **H** button on the toolbar. This turns the block into a Heading or `<h1>` element:



Feel free to experiment with the controls and see how they affect the text. Check the formatting options and **Preview** mode, which allows you to see what the final document looks like when it's rendered to HTML markup.



There are many other options and features that you can enable or customize. Be sure to check out the documentation for SimpleMDE here: <https://github.com/sparksuite/simplemde-markdown-editor>. The out-of-the-box experience is pretty basic, and you may want to toggle additional settings later on.

One of the most important things you need to address is keeping your web application inside the Electron window. Let's see what it takes to match the content to the screen size.

Fitting the screen size

If you keep experimenting with your application at runtime, you may notice that the editor component doesn't fit the whole application area once you start resizing the window or maximizing it. To address this, we need to add some CSS styles to tell the component it needs to fit the parent width and height.

Please note that, at the lower level, SimpleMDE wraps another great component called CodeMirror.



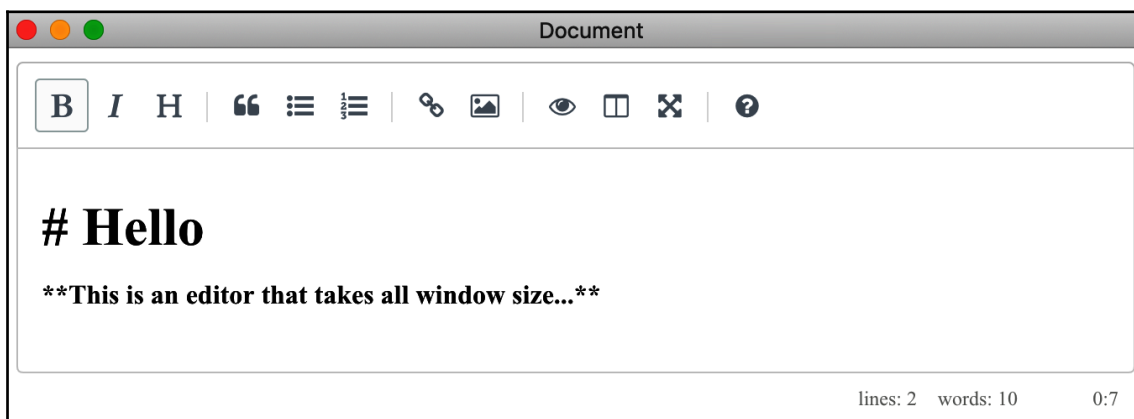
CodeMirror is a versatile text editor that's implemented in JavaScript for the browser. It is specialized for editing code and comes with a number of language modes and addons that implement more advanced editing functionality.

Here, we are going to add `flex` layout features to the whole body of the HTML base and add some styling support for the CodeMirror part, which is part of SimpleMDE. Let's get started:

1. Update the styles in the `index.html` file according to the following code:

```
<style>
  html, body {
    height: 100%;
    display: flex;
    flex: 1;
    flex-direction: column;
  }
  .CodeMirror {
    flex: 1;
  }
</style>
```

2. Run the application and try resizing the window to make it wider or taller. Notice that, now, the markdown editor area perfectly fits the entire page area:



Now, let's move on to integrating the application menu.

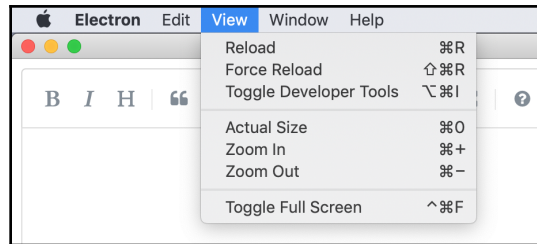
Integrating the application menu

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.

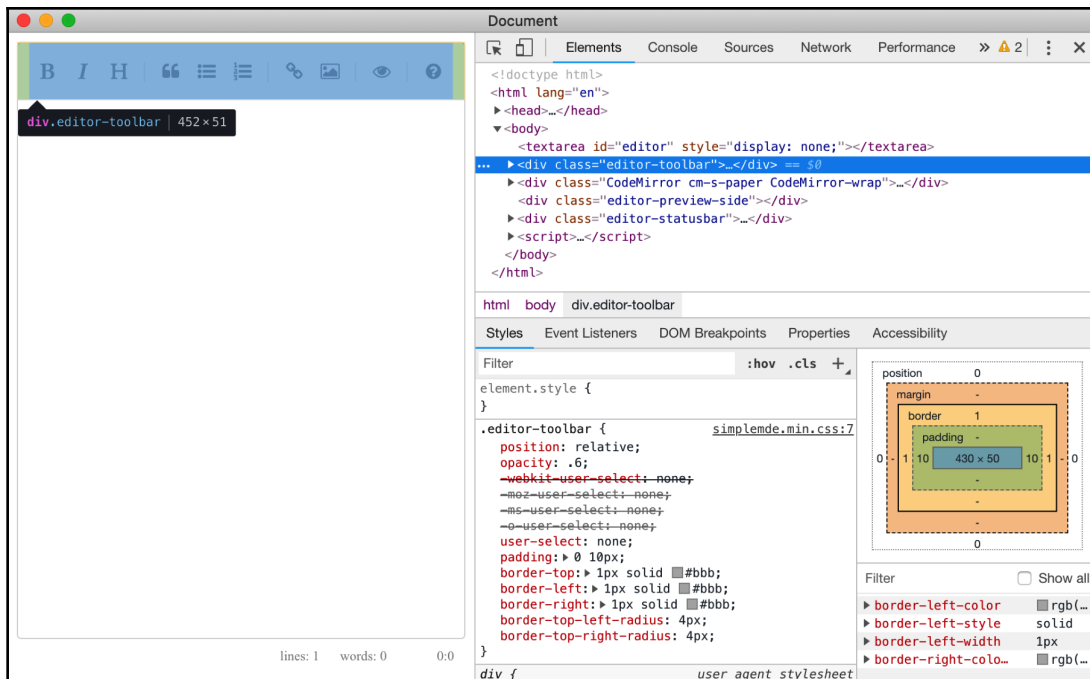
Handling every case would be quite cumbersome for developers; that is why the Electron framework provides a unified interface for building application menus from the JSON definition and takes care of integration details.

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:



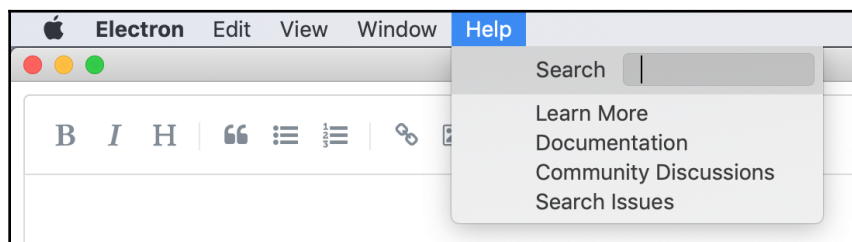
Now, let's see what it takes to create such menus from within application code. We are going to perform the following actions with the system menu component:

- 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.



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:

```
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:

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



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:

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

module.exports = menu;
```

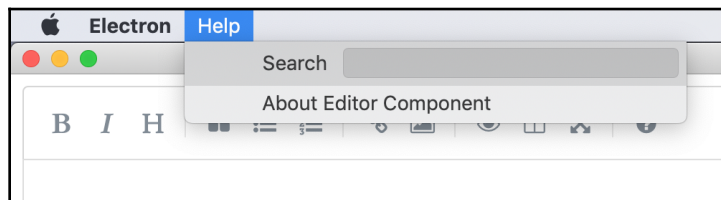

We build a new instance of the menu and export it through the `module.exports` call. Module exporting is a Node.js feature that allows us to import the `Menu` instance to other files. In our case, we need to export the menu from the `menu.js` file and import it to `index.js`, which is where the central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

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

```
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/` 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:

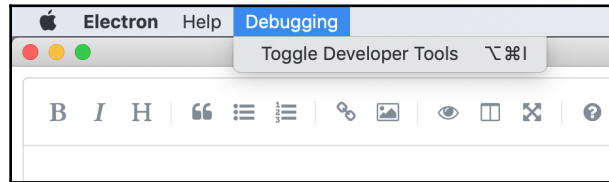
```
const template = [
  {
    role: 'help',
    submenu: [
      {
        label: 'About Editor Component',
        click() {
          shell.openExternal('https://simplemde.com/');
        }
      }
    ]
  },
  {
    label: 'Debugging',
    submenu: [
      {
        role: 'toggleDevTools'
      }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

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:

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:

```
{
  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.



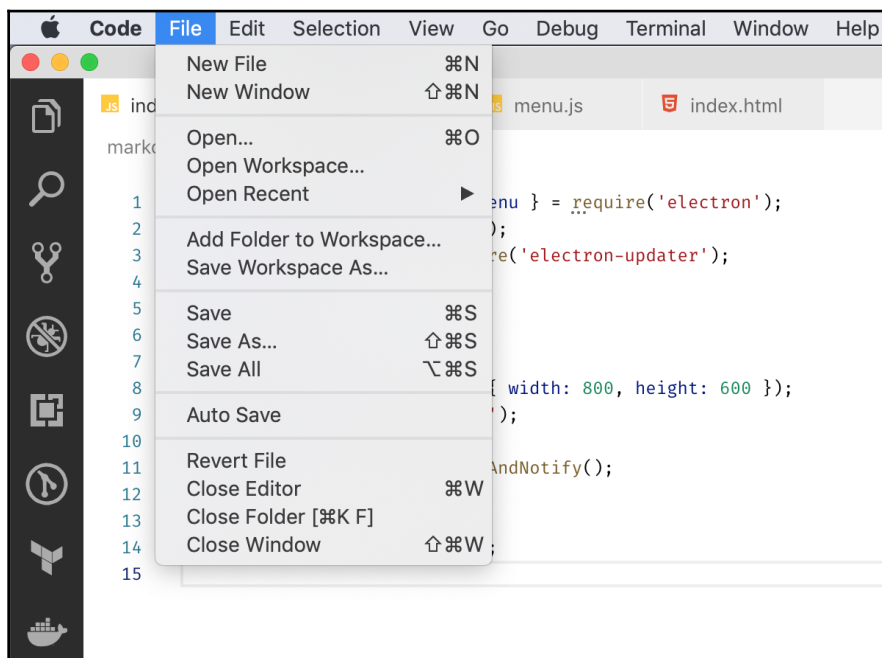
You can find out more about supported `role` values at <https://electronjs.org/docs/api/menu-item#roles>.

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:



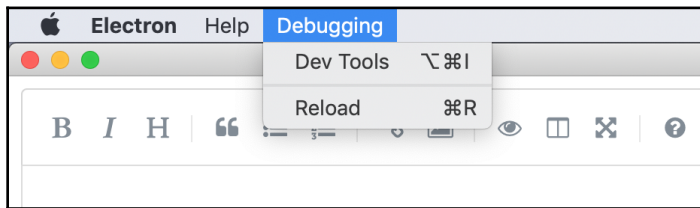
The keyboard shortcuts may differ, depending on the platform you are using, but the structure should be the same with all operating systems.

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:

```
{
  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.

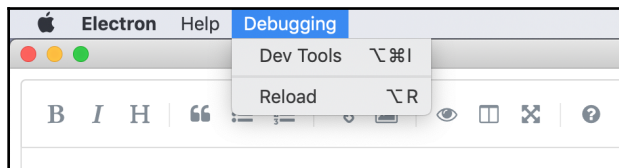
In the previous examples, when you created a menu item separator, we introduced an additional menu item called **Reload**. This reloads the embedded browser with each click and allows you to see the updated HTML code. The `reload` role covers this functionality, but the item has no keyboard shortcut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

```
{
  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.



You can find out more about accelerators and their use cases at <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:

```
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.

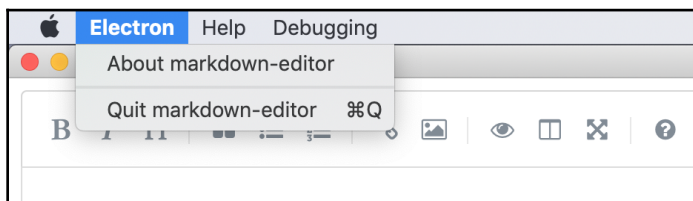
Node.js exposes a global object called `process`, which provides access to environment variables. This object can also provide information about the current platform architecture. We are going to check this against the `darwin` value to detect the macOS platform.

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

```
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).



For more details regarding `process.platform`, please refer to the following Node.js documentation: 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:

```
{
  "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"  
}  
}
```

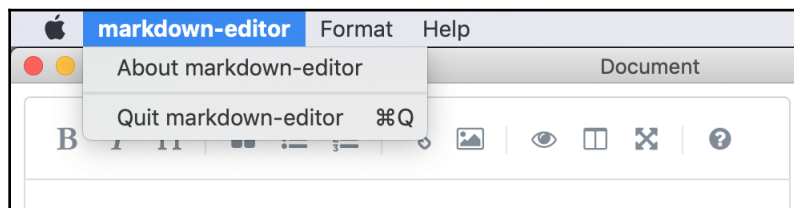
But when you package the application for distribution, it is going to have its own version of Electron embedded in it. In that case, the name of your application renders as expected.

Let's test the package with the macOS build:

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

```
{  
  "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 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:

```
{
  "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:

```
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:

```
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:

```
{
  "name": "markdown-editor",
  "version": "1.1.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "DEBUG=true electron ."
  }
}
```



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, 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>.

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:

```
<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:

```
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:

```
<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:

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

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

```
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:

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

5. The whole script block should look as follows:

```
<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:

```
> 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:

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

The format of the call is as follows:

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

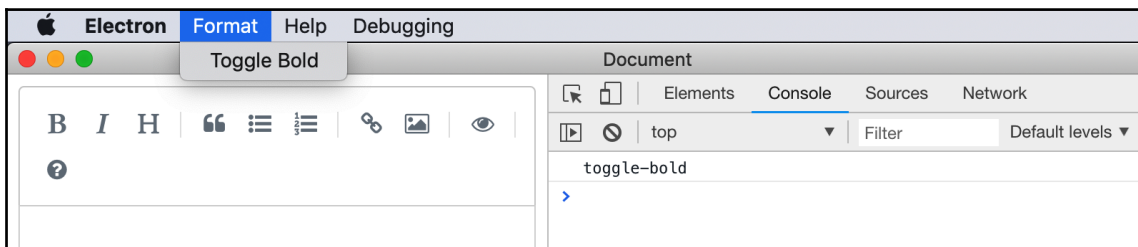
At this point, we have communication handlers from both areas, that is, the browser and Node.js. It is time to wire everything with a menu item.

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:

```
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:

```
> 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:

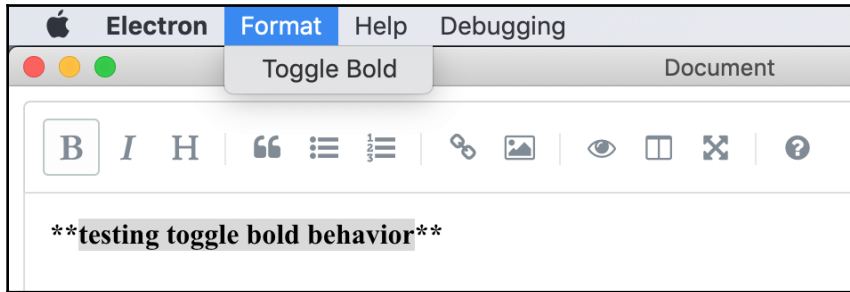
1. The markdown editor component we are using for this application provides multiple functions that developers can invoke from code. One of those functions is `toggleBold()`. Our code can check the content of the message, and if it's the `toggle-bold` one, it will run the corresponding component function:

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

2. The whole script section should look as follows:

```
<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>
```

- 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()`.



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>.

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:

```
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:

```
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 to 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:

```
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:

```
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:

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

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

```
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:

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

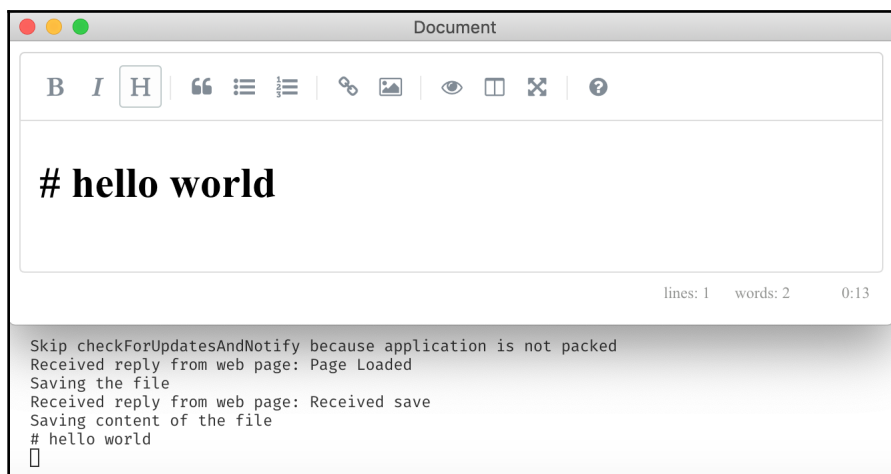
As you can see, this isn't doing much. For the sake of simplicity, it is just putting received data into the Terminal output so that we can verify that the messaging is working as expected.

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

```
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.

- 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:

```
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:

```
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:

```
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);
});
```



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

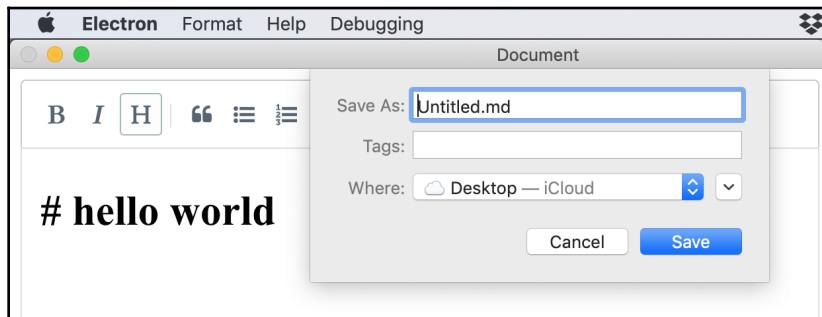
`showSaveDialog` receives the third parameter, that is, the callback function that gets invoked if the user closes the dialog with the `Save` or `Cancel` button. The first callback parameter provides you with the path of the file to use when saving content.

Let's see how the whole thing works.

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

```
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:

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

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

```
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:

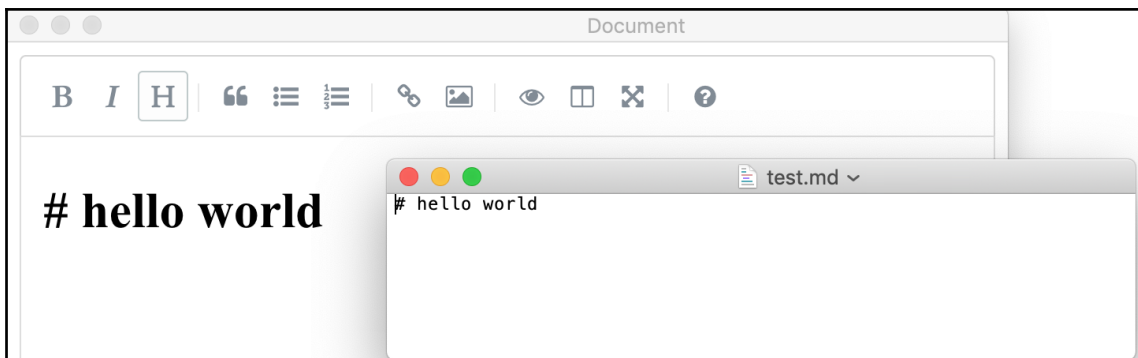
```
const fs = require('fs');
```

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:

```
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:

```
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:

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

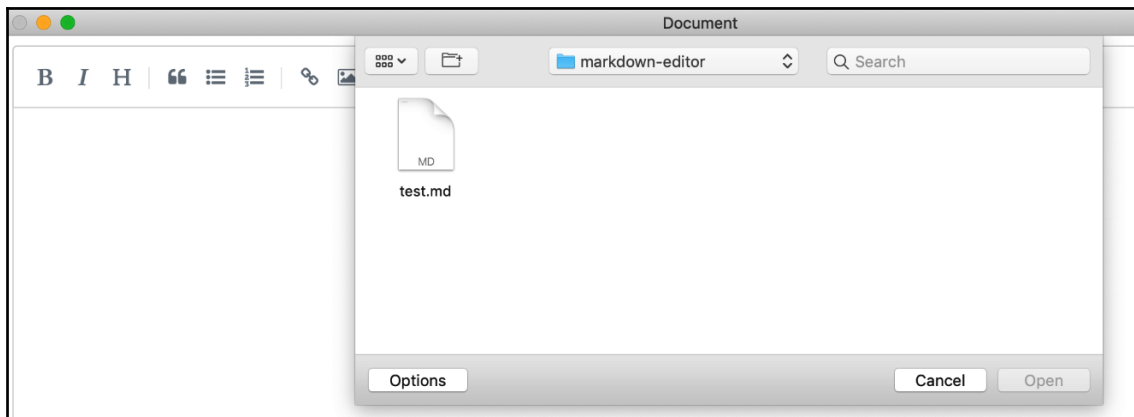
We have already imported the `dialog` object from the Electron framework. You can use it to invoke the system's **Open** dialog as well.

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

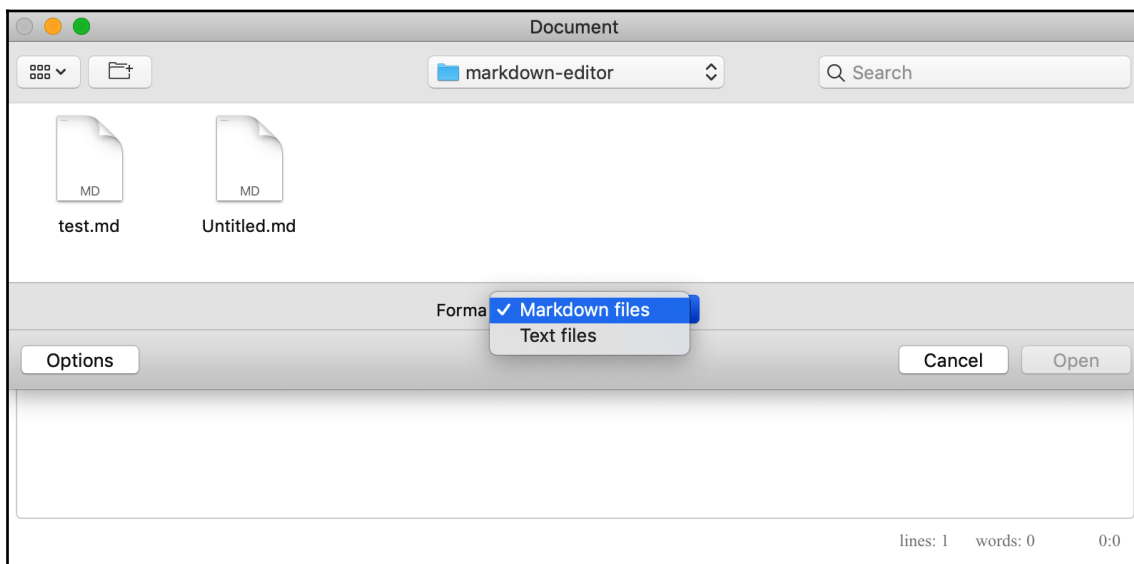
```
globalShortcut.register('CommandOrControl+O', () => {  
  const window = BrowserWindow.getFocusedWindow();  
  
  const options = {  
    title: 'Pick a markdown file',  
    filters: [  
      { name: 'Markdown files', extensions: ['.md'] },  
      { name: 'Text files', extensions: ['.txt'] }  
    ]  
  };  
  
  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:

```
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:

```
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:

```
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);
    }
  });
});
```

```
    }  
  });  
});
```

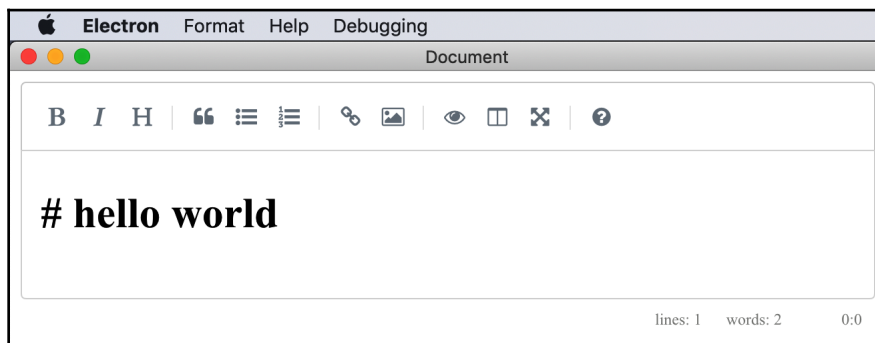
11. Open the `index.html` file for editing and scroll to the scripts section, where we already have some process communication handling in place.
12. Add a new handler that listens to the `load` channel and the corresponding messages coming from the renderer process:

```
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:

```
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:

```
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:

```
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:

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

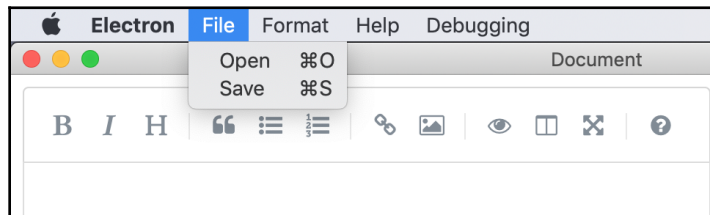
  globalShortcut.register('CommandOrControl+O', () => {
```

```
    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:

```
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.

Adding drag and drop support

Another nice feature you can provide for your small markdown editor application is the ability to drag and drop files onto the window. Users of your application should have the ability to drop a markdown file onto the editor's surface and have the content of the file immediately available to them. This scenario also helps us address some extra features of the Electron framework that you may use later. Let's get started:

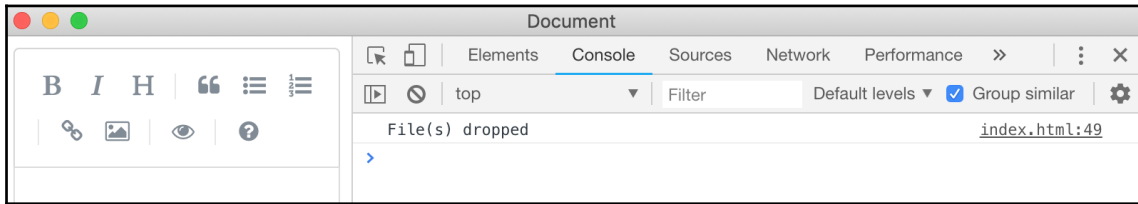
1. The easiest way to enable drop support for an entire web page running in Electron is to set the `ondrop` event handler for the body element:

```
<body ondrop="dropHandler(event);">
  <!-- page content -->
</body>
```

2. For now, the drop handler implementation can be as simple as putting a message into the browser console's output. The most important part here is to prevent the default behavior and tell other DOM elements that we are now responsible for drop operations:

```
<script>
function dropHandler(event) {
  console.log('File(s) dropped');
  event.preventDefault();
}
</script>
```

3. Run the Chrome Developer Tools with the **Console** tab open and drag and drop a file from your system onto the markdown editor area:



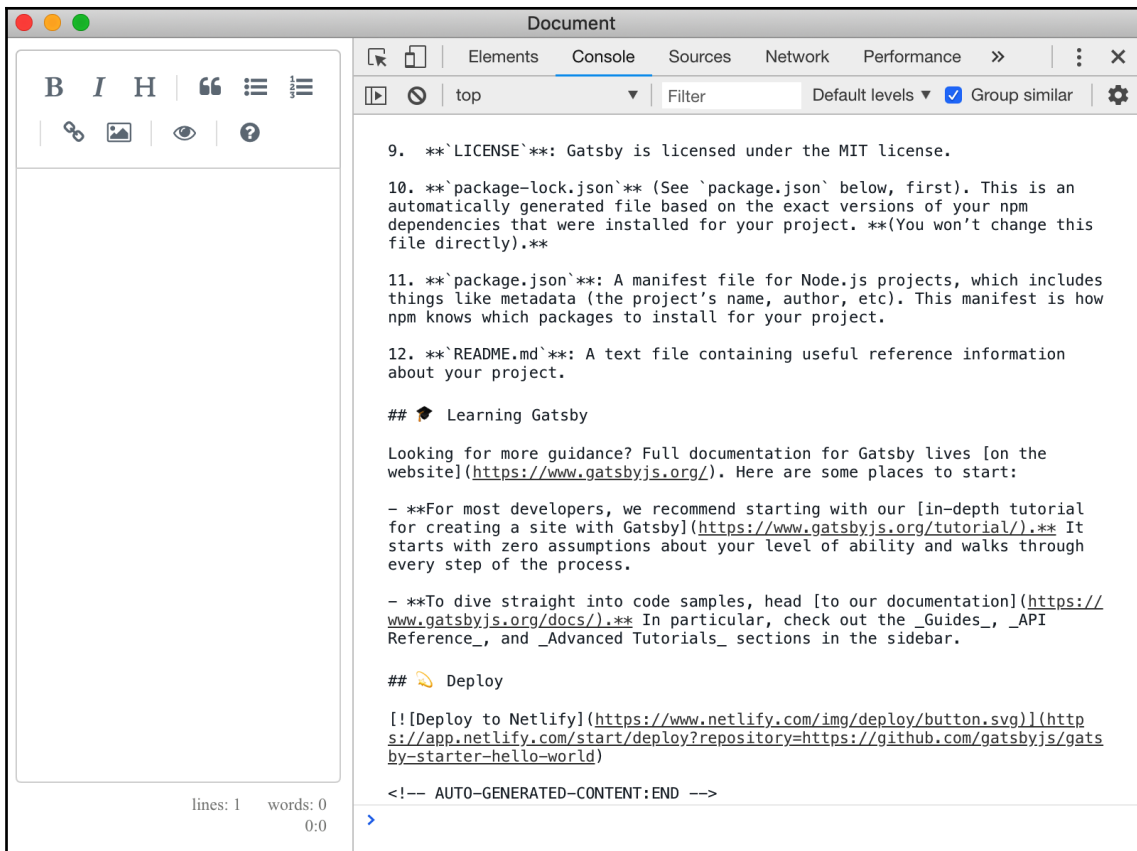
You can find out more about drag and drop handling in HTML5 at https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API/File_drag_and_drop.

4. For the next step, let's have some code that reads the content of the file that the user drags and drops, and shows the text in the browser console. Please refer to the following listing to see what the code should look like:

```
function dropHandler(event) {
  event.preventDefault();

  if (event.dataTransfer.items) {
    if (event.dataTransfer.items[0].kind === 'file') {
      var file = event.dataTransfer.items[0].getAsFile();
      if (file.type === 'text/markdown') {
        var reader = new FileReader();
        reader.onload = e => {
          console.log(e.target.result);
        };
        reader.readAsText(file);
      }
    }
  }
}
```

5. Notice that we are filtering out dropped files by the mime type. It should be equal to the `text/markdown` value, meaning that you need to use files with the `.md` extension. Also, we only take the first file entry if the user drops multiples.
6. Run the application, open Chrome Developer tools, and drop a markdown file onto the editor. It can be anything. In our example, it's a `README.md` file:



As you can see, the text of the markdown file should be present in the browser's console output.

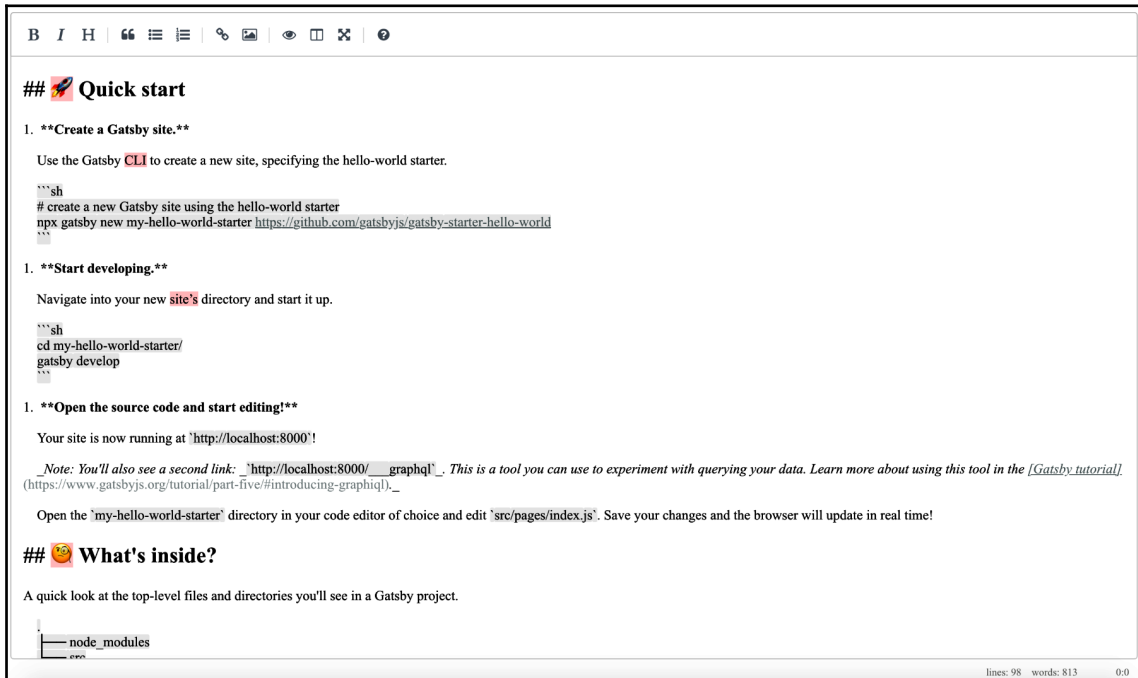
7. The final part of the implementation is straightforward. We already have a reference to the SimpleMDE editor instance, so the only thing we need to do is call the `codemirror` function in order to set the new text value, as follows:

```

var reader = new FileReader();
reader.onload = e => {
  // console.log(e.target.result);
  editor.codemirror.setValue(e.target.result);
};

```

8. Try the application once again. You should see the text from the dropped file appear directly inside the markdown editor:



Our implementation had succeeded, so feel free to clean the code from the `console.log` calls. Now, let's learn how we can support automatic updates with our markdown editor application.

Supporting automatic updates

The `electron-builder` project that we are using with our Electron application also provides support for automatic updates. In this section, we will learn how to set up a GitHub repository so that we can store and distribute application updates.

Our Markdown Editor application is going to check for new versions on each start-up and notify users if a new version is available. Let's set up automatic updates for Electron applications:

1. First, let's create a new GitHub repository and call it `electron-updates`. Initialize it with the **README** file to save time cloning and setting up the initial content:




Please select **Public** mode for the new GitHub repository. This is going to simplify the entire configuration and update process significantly.

Create a new repository

A repository contains all project files, including the revision history.

Owner

Repository name *

 DenysVuika ▾


 /

electron-updates ✓


Great repository names are short and memorable. Need inspiration? How about **furry-enigma**?

Description (optional)

testing automatic updates

☒  **Public**

Anyone can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.


☒ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾

 |

Add a license: **None** ▾



Create repository



It is possible to use private GitHub repositories too. However, private updates require authentication tokens and should only be used for edge cases, according to the documentation.

2. Next, we need to generate a separate authentication token to allow our application to access GitHub and fetch updates:

OAuth Apps

GitHub Apps

Personal access tokens

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Token description

publish-electron-app

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

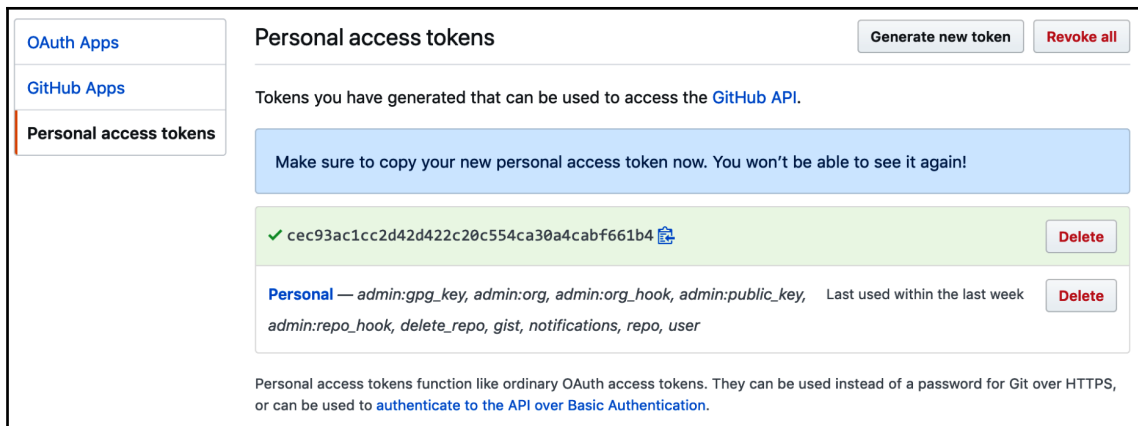
<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations



Follow the procedure documented at <https://help.github.com/en/articles/creating-a-personal-access-token-for-the-command-line> to do this. You can generate a new token by going to <https://github.com/settings/tokens/new>.

Please note that the access token you create with GitHub's web interface should have the scope/permission repository set.

3. Once you get the token, save it somewhere—you are going to use it from the command line as a `GH_TOKEN` environment variable:



In my case, for demonstration purposes, the token is a value like this:

```
cec93ac1cc2d42d422c20c554ca30a4cabf661b4
```

Please note, however, that the token is exclusive and is equivalent to a password. Never share it with others and don't push it to the source code. For the rest of the examples in this chapter, we are only going to use the access token from the command line in the form of an environment variable.

4. Install the `electron-updater` dependency to enable support for automatic update checks in our markdown editor project:

```
npm i electron-updater
```

5. Update the `package.json` file and append the build and publish settings:

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

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

  "build": {
    "appId": "com.my.markdown-editor",
    "publish": {
      "provider": "github",
      "owner": "<username>",
      "repo": "electron-updates"
    }
  }
}
```

```
    }  
  }  
}
```

6. Use your GitHub account name as the `owner` property value and `electron-updates` as the `repo` value. This is how we call our GitHub project upon creation.

Now, let's learn how to publish the macOS distribution:

1. Update the `scripts` section of your `package.json` file according to the following code:

```
{  
  "scripts": {  
    "publish:github": "build --mac -p always"  
  }  
}
```



For more details on automatic update configuration, please refer to the corresponding documentation online: <https://www.electron.build/auto-update>.

2. Don't run the `publish` command yet; we still need to wire the automatic update checks with the code.
3. Switch to the `index.js` file and import the `autoUpdater` object from the `electron-updater` library:

```
const { autoUpdater } = require('electron-updater');
```

4. Checking for a new version of the application is extremely easy. All you need to do is call the `checkForUpdatesAndNotify` method of the `autoUpdater` object—the Electron library will handle the rest of the functionality.
5. Update the `ready` event in the `index.js` file, as follows:

```
app.on('ready', () => {  
  window = new BrowserWindow({  
    ...  
  });  
  window.loadFile('index.html');  
  autoUpdater.checkForUpdatesAndNotify();  
});
```


Here, we're creating a window, loading the `index.html` file to display the user interface, and then initiating the update check. The updater will perform the check against the GitHub repository and release it in the background so that our users can keep using the application without interruptions.

6. The final content of your `index.js` file should look as follows:

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

let window;

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

  autoUpdater.checkForUpdatesAndNotify();
});

Menu.setApplicationMenu(menu);
```

7. Now, you can run the following command to publish your first application version to GitHub:

```
GH_TOKEN=cec93ac1cc2d42d422c20c554ca30a4cabf661b4
npm run publish:github
```

8. Don't forget to provide your token value for the `GH_TOKEN` environment variable. There may be many output messages in the Terminal window. The tool is going to compile the application, sign it, upload it to your GitHub repository, and issue the release draft.
9. The end of the log should look similar to the following:

```
building target=macOS zip arch=x64 file=dist/
markdown-editor-1.0.0-mac.zip
building target=DMG arch=x64 file=dist/
markdown-editor-1.0.0.dmg
building block map blockMapFile=dist/
markdown-editor-1.0.0.dmg.blockmap
```

```
publishing publisher=Github (owner: DenysVuika, project:
electron-updates, version: 1.0.0)
uploading file=markdown-editor-1.0.0.dmg.blockmap provider=Github
uploading file=markdown-editor-1.0.0.dmg provider=Github
creating Github release reason=release doesn't exist tag=v1.0.0
version=1.0.0 [===== ] 38% 25.6s | markdown-editor-1.0.0.dmg
to Github
building embedded block map file=dist/markdown-editor-1.0.0-mac.zip
[===== ] 40% 24.4s | markdown-editor-1.0.0.dmg
to Github
uploading file=markdown-editor-1.0.0-mac.zip provider=Github
[=====] 100% 0.0s | markdown-editor-1.0.0.dmg
to Github
[=====] 100% 0.0s | markdown-editor-1.0.0-mac.zip
to Github
```



Note the primary steps in the execution: **building**, **uploading**, and **creating GitHub release**. If there are no errors in the output, then the publishing went well and as expected.

10. Navigate to your GitHub repository and switch to the **Releases** section. You should see a new **Release Draft** there with a few files that we are going to distribute:

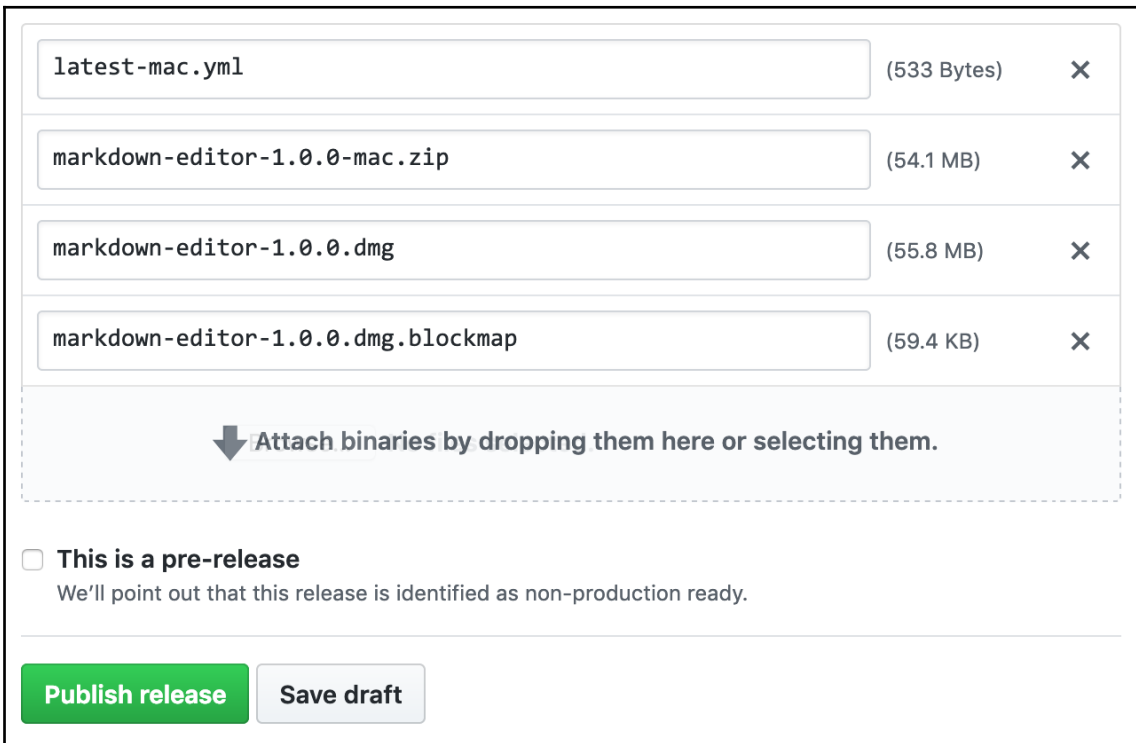
The screenshot shows the GitHub interface for a repository. At the top, there are navigation links: Code, Issues (0), Pull requests (0), Projects (0), Wiki, Insights, and Settings. Below these, the 'Releases' tab is active, and a 'Draft a new release' button is visible. The main content area shows a draft release for version '1.0.0'. It includes the user's profile picture and name 'DenysVuika' with the text 'drafted this 4 minutes ago'. Under the 'Assets' section, four files are listed:

Asset Name	Size
latest-mac.yml	533 Bytes
markdown-editor-1.0.0-mac.zip	54.1 MB
markdown-editor-1.0.0.dmg	55.8 MB
markdown-editor-1.0.0.dmg.blockmap	59.4 KB

As you may have recalled, we have configured the macOS target for building and packaging. This is why multiple different download links refer to the macOS platform. As soon as you enable other targets, you should see more entries on the release draft page, including Windows installers and Linux packages.

You can publish multiple times to the save release version draft. The version number depends on the `version` field value inside your `package.json` file.

11. Once you are happy with the first version, you can click the **Edit** button, write some details about the current release, and press the **Publish release** button:



latest-mac.yml	(533 Bytes)	×
markdown-editor-1.0.0-mac.zip	(54.1 MB)	×
markdown-editor-1.0.0.dmg	(55.8 MB)	×
markdown-editor-1.0.0.dmg.blockmap	(59.4 KB)	×

↓ Attach binaries by dropping them here or selecting them.

☐ This is a pre-release
We'll point out that this release is identified as non-production ready.

Publish release Save draft

As soon as you publish the release, the application becomes available for all users. Now, let's see automatic updates in action.

Testing automatic updates

Testing the whole auto-update process takes a few steps since you need to install one version of the app, publish a new one, and then see what happens. Let's see those steps in practice:

1. Go to the release page and download the installer package. For macOS, this is going to be in `.dmg` format:



2. Install the app and run it to ensure it works as expected. Close the application for now; we are going to return to it shortly.
3. Update the `package.json` file and set the `version` attribute to `1.1.0`. Alternatively, you can run the following command to update the file:

```
npm version minor
```

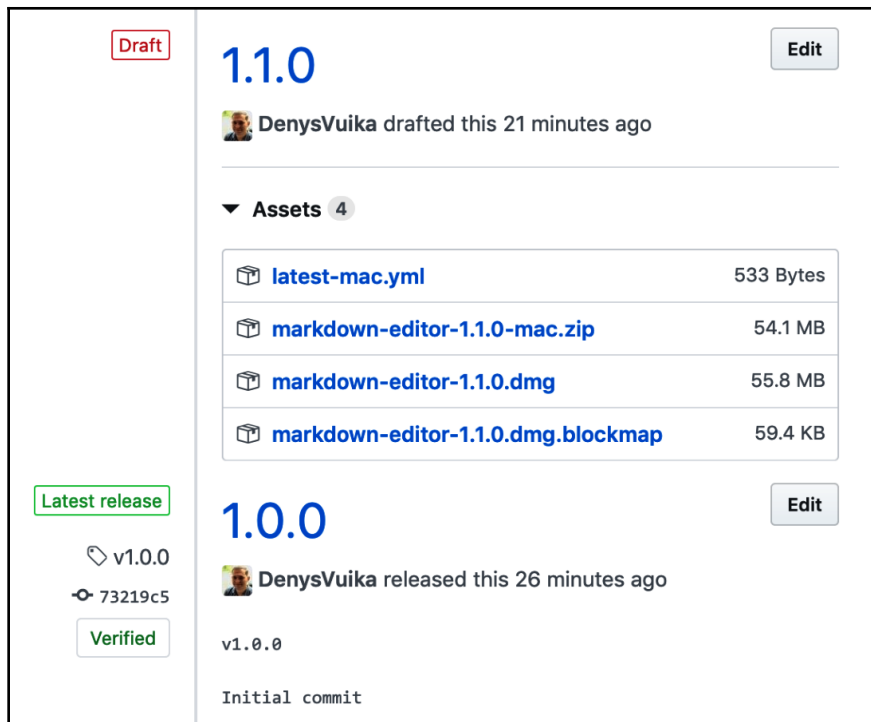
4. The output should be as follows:

```
v1.1.0
```

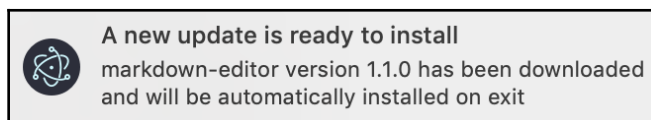
5. Run the `publish` command once again to create a new release draft:

```
GH_TOKEN=<YOUR-TOKEN> npm run publish:github
```

6. Now, you should have two releases on GitHub, including a new draft for version `1.1.0`:

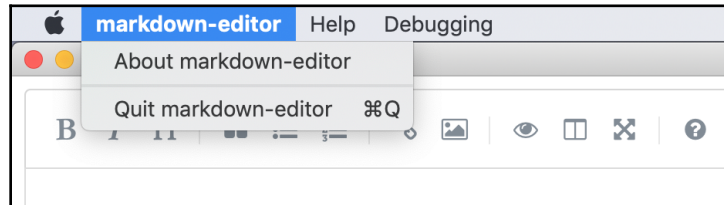


7. Perform the same steps you performed earlier and publish the new release. Then, run the application you downloaded and installed earlier.
8. In a few seconds, after startup, the automatic updater will raise a system notification, saying that a new version of the application has been downloaded and ready to install:

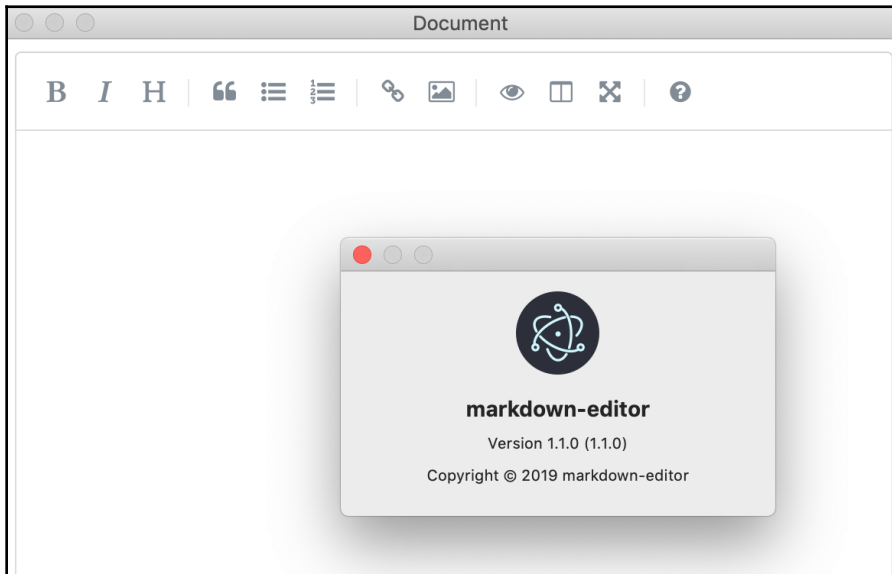


9. Quit the application and run it once again. At this point, you should be using the latest version, that is, 1.1.0 (at the time of writing).

10. You can use the standard Electron framework out of the box to check that your application version is the latest one:



11. Note the version value; it is now **1.1.0**:



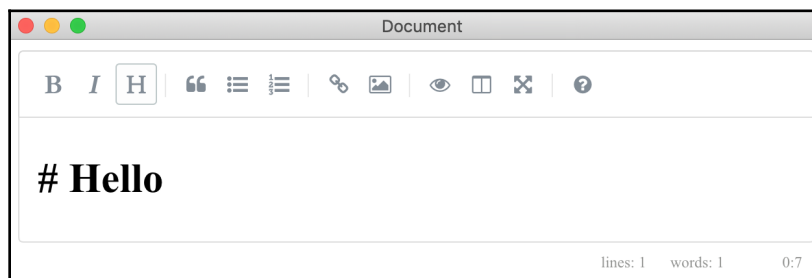
Well done and congratulations on setting up publishing and automatic updates for your application!

As an exercise, try to configure building and publishing for other platforms. Be sure to test the installation and upgrade process with Windows or Ubuntu Linux if you have real or virtual machines nearby.

In the next section, we are going to provide a proper title for our application.

Changing the title of the application

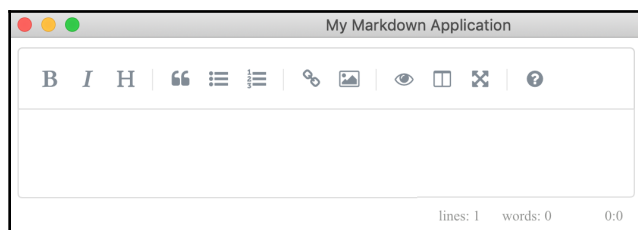
Throughout the whole process of application development, you may have noticed that our window is called **Document**, as shown in the following screenshot:



This is not an issue with the Electron framework; the title of the page comes from the `<title>` tag inside the `index.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>Document</title>
    <link
      rel="stylesheet"
      href="./node_modules/simplemde/dist/simplemde.min.css"
    />
```

Change the value of the `title` to something more meaningful, for example, **My Markdown Application**, and restart the application. You should see the new title, as follows:



Feel free to provide a different value for the name. Usually, it is the same value you are going to have in the `package.json` file, inside the `name` property.

Summary

In this chapter, we have successfully created a minimalistic markdown editor. We have walked through the process of integrating third-party editor components, wiring keyboard combinations, and performing messaging between the browser and Node.js parts of the Electron application. You should now have a better understanding of application deployments and automatic updates, as well as simple release management via the GitHub repository.

Then, you learned how to build a basic desktop application with system menu integration and access to the local filesystem. This is essentially the bare bones of a typical Electron project you are going to work on in the future. However, the Electron framework provides you with a wrapper around your web application. You still need to decide whether to use plain JavaScript, HTML, and CSS or employ an existing web framework to move faster. This is something we are going to look at in the next chapter.

3

Integrating with Angular, React, and Vue

With the rapid evolution of web technologies and frameworks, we no longer need to build our web applications from scratch. There's a vast variety of component libraries, feature libraries, extensions, and frameworks that combine the most reusable building blocks for our needs.

In this chapter, we are going to focus on three essential frameworks that most modern developers use, as follows:

- Angular, which is backed by Google
- React, which is backed by Facebook
- Vue.js, which is backed by Evan You and its sponsors

You are about to go through the process of setting up three different projects for each web framework. As part of this practical exercise, you will learn how to configure the *live reloading* feature, integrate UI toolkits and component libraries, and set up application routing.

In this chapter, we will cover the following topics:

- Building an Electron application with Angular
- Building an Electron application with React
- Building an Electron application with Vue.js

Let's get started!

Technical requirements

To get started with this chapter, you will need a standard laptop or desktop running macOS, Windows, or Linux.

The software that you'll need to have installed to complete this chapter is as follows:

- Git, a version control system
- Node.js with **node package manager (NPM)**
- Visual Studio Code, a free and open-source code editor

You can find the code files for this chapter in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter03>.

Let's start with the Angular framework.

Building an Electron application with Angular

This section assumes that you already have experience with the Angular framework. To find out more, please refer to the Angular Quickstart section at <https://angular.io/guide/quickstart>.

To get the application up and running fast, we are going to use the Angular CLI. The Angular CLI is a project that's maintained by the Angular team. It's described in the official documentation (<https://angular.io/cli#cli-overview-and-command-reference>) as follows:

The Angular CLI is a command-line interface tool that you use to initialize, develop, scaffold, and maintain Angular applications. You can use the tool directly in a command shell, or indirectly through an interactive UI such as Angular Console.

You can install the latest version of the Angular CLI through the NPM package manager. Typically, developers install it as a global tool so that they can generate a new project in any folder using the command-line tool or Terminal application.

To find out more about the Angular CLI, and globally get a list of all supported commands with a detailed explanation of what they do, please go to <https://angular.io/cli>.

Run the following command to install the CLI:

```
npm i -g @angular/cli@latest
```

The output should look similar to the following:

```
/usr/local/bin/ng -> /usr/local/lib/node_modules/@angular/cli/bin/ng  
+ @angular/cli@7.3.8
```

The NPM package manager downloads and installs the Angular CLI and all its dependencies. Upon completion, it also registers a new global `ng` command that you can use anywhere.

Now, let's create our Angular project scaffold, which we will use with the Electron shell.

Generating our Angular project scaffold

In this section, we are going to learn how to set up a new project that follows Angular's development practices. Let's get started:

1. Run the following commands to generate a new Angular project called `integrate-angular`:

```
ng new integrate-angular  
cd integrate-angular
```

The Angular CLI tool usually asks a series of questions to clarify what extra features you want to have in the resulting application.

2. If you're asked about routing support, type `Y` and press *Enter*:

```
Would you like to add Angular routing? (y/N)  
Y
```

3. Next, if the tool asks you about the stylesheet format, choose **SCSS**, as shown in the following code:

```
Which stylesheet format would you like to use? (Use arrow keys)  
SCSS
```

4. The output of the preceding code is as follows:

```
[?] Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS [ https://sass-lang.com/documentation/syntax#scss ]
CREATE integrate-angular/README.md (1033 bytes)
CREATE integrate-angular/.editorconfig (246 bytes)
CREATE integrate-angular/.gitignore (631 bytes)
CREATE integrate-angular/angular.json (3769 bytes)
CREATE integrate-angular/package.json (1291 bytes)
CREATE integrate-angular/tsconfig.json (543 bytes)
CREATE integrate-angular/tslint.json (1988 bytes)
CREATE integrate-angular/browserslist (429 bytes)
CREATE integrate-angular/karma.conf.js (1029 bytes)
CREATE integrate-angular/tsconfig.app.json (270 bytes)
CREATE integrate-angular/tsconfig.spec.json (270 bytes)
CREATE integrate-angular/src/favicon.ico (5430 bytes)
CREATE integrate-angular/src/index.html (303 bytes)
CREATE integrate-angular/src/main.ts (372 bytes)
CREATE integrate-angular/src/polyfills.ts (2838 bytes)
CREATE integrate-angular/src/styles.scss (80 bytes)
CREATE integrate-angular/src/test.ts (642 bytes)
CREATE integrate-angular/src/assets/.gitkeep (0 bytes)
CREATE integrate-angular/src/environments/environment.prod.ts (51 bytes)
CREATE integrate-angular/src/environments/environment.ts (662 bytes)
CREATE integrate-angular/src/app/app-routing.module.ts (246 bytes)
CREATE integrate-angular/src/app/app.module.ts (393 bytes)
CREATE integrate-angular/src/app/app.component.scss (0 bytes)
CREATE integrate-angular/src/app/app.component.html (1152 bytes)
CREATE integrate-angular/src/app/app.component.spec.ts (1128 bytes)
CREATE integrate-angular/src/app/app.component.ts (222 bytes)
CREATE integrate-angular/e2e/protractor.conf.js (810 bytes)
Successfully initialized git.
```

Note how the Angular CLI generates a set of files for you. The tool provides you with various *ignore* rules for NPM and Git inside the `.gitignore` file, configuration files for Typescript and the Karma test runner, and even a set of unit and end-to-end tests as part of the initial scaffold.

5. Check out what the `package.json` file looks like, especially its general information and the `scripts` section:

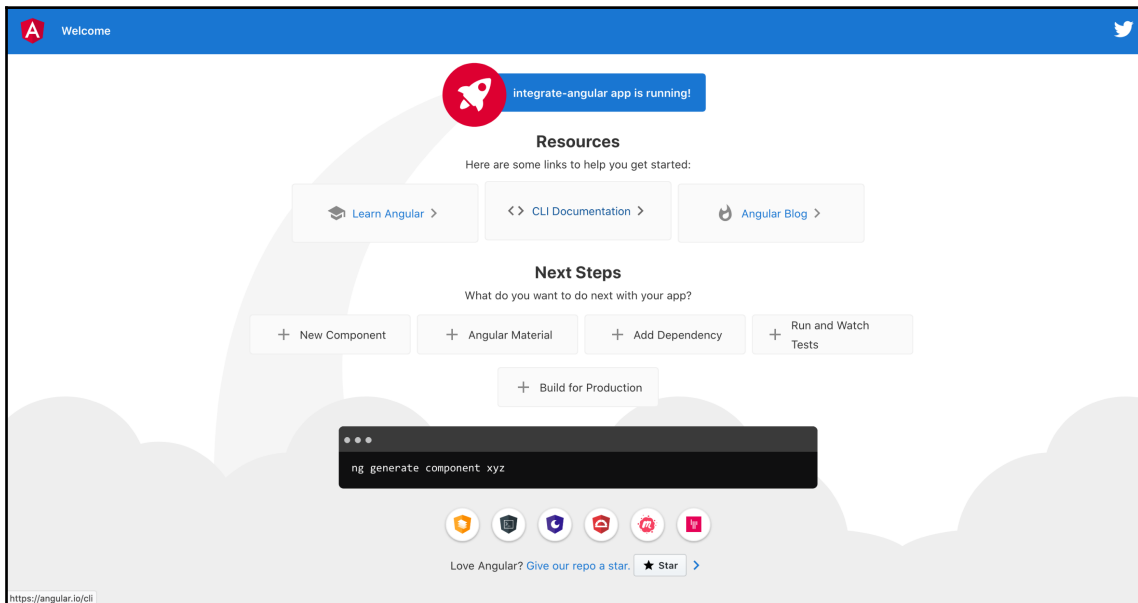
```
{
  "name": "integrate-angular",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
}
```

6. The Angular CLI also performs dependency library installation, so all you need to do is run the following command to get your web application up and running:

npm start

Historically, every web app that we generate with the help of the Angular CLI runs on port 4200 by default. You can quickly change the port in the future, but for now let's stick to the defaults.

7. Launch your preferred browser and navigate to `http://localhost:4200`. You should see a landing page called **Welcome to integrate-angular!** that the Angular CLI has created for you:



Now, let's configure the Electron shell so that it works with Angular code.

Integrating the Angular project with Electron

Now that you have an Angular project scaffold, let's integrate it with the Electron shell:

1. Open the project in Visual Studio Code. You can do that from the Terminal using the following command:

```
code .
```

First, though it's not mandatory, let's change the application's title.

2. Open the `src/index.html` file and change the content of the `title` tag to Angular with Electron, or any title of your choice:

```
<title>Angular with Electron</title>
```

3. Now, we need to update the base application path to `./`:

```
<base href="./" />
```

This makes all resources relative to the `index.html` file. This is what we need when we're running an Angular application from within the Electron shell.

4. This results in the following output:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Angular with Electron</title>
    <base href="./" />
    <meta name="viewport" content="width=device-width,
      initial-scale=1" />
    <link rel="icon" type="image/x-icon" href="favicon.ico" />
  </head>
  <body>
    <app-root></app-root>
  </body>
</html>
```

5. Switch to the Command Prompt or a Terminal window. We need to install the `electron` library into the project. You can do this with the following command:

```
npm i electron -D
```

6. We need to provide a `main.js` file and register it with the `package.json` file so that it acts as the main entry point for the Electron shell that will be loaded upon startup. Every time Electron starts, it checks for the entry and uses that file.
7. Open the `package.json` file so that you can edit it. Update it so that the code looks as follows:

```
{
  "name": "integrate-angular",
  "version": "0.0.0",
  "main": "main.js",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
}
```

8. Now, create a `main.js` file in the project root folder and add the following content to it:

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

let win;

function createWindow() {
  win = new BrowserWindow({ width: 800, height: 600 });

  win.loadFile('index.html');

  win.on('closed', () => {
    win = null;
  });
}

app.on('ready', createWindow);

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

app.on('activate', () => {
  if (win === null) {
```

```
        createWindow();  
    }  
});
```



To find out more, please refer to <https://electronjs.org/docs/tutorial/first-app>. The Electron team has provided a great set of examples and code blocks that you can copy and paste into your application.

The preceding code is a minimal implementation of an Electron window. We are going to use similar snippets throughout this book.

9. The first thing that we need to change in the `main.js` code is where we can find the `index.html` file. If we compile an Angular project in production mode, we'll get the final application artifacts in the `dist` subfolder. Change the `win.loadURL` call to reflect that:

```
win.loadURL(`file://${__dirname}/dist/index.html`)
```

For experienced developers, if you have multiple projects in the workspace, you may also need to specify the project folder name within the output, as follows globally:

```
win.loadURL(`file://${__dirname}/dist/integrate-angular/index.html`);
```

10. Before we finish the configuration, switch back to the `package.json` file and rename the `start` script to `serve`. Then, add a new `start` script entry to invoke the Electron application:

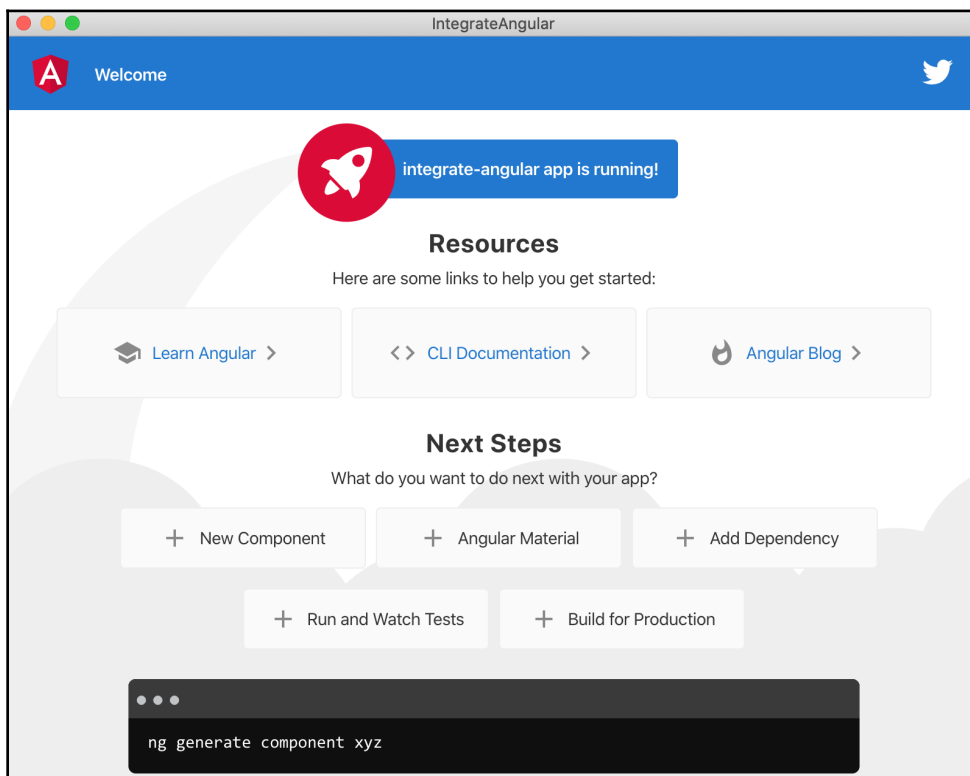
```
{  
  "name": "integrate-angular",  
  "version": "0.0.0",  
  "main": "main.js",  
  "scripts": {  
    "ng": "ng",  
    "serve": "ng serve",  
    "start": "electron .",  
    "build": "ng build",  
    "test": "ng test",  
    "lint": "ng lint",  
    "e2e": "ng e2e"  
  },  
}
```


11. Launch the Terminal window in VS Code, or use any other Command Prompt tool, and execute the following scripts:

```
npm run build
npm start
```

The first command builds the application in production mode. This provides you with a `dist` folder that contains optimized scripts, styles, and HTML files. The second command launches an Electron shell with your application inside it so that you can test or debug your features.

12. The application window should look as follows:



Now, the project is ready to be worked on. However, as you may have noticed, to test a change in the code or user interface, you have to perform the following actions:

1. Stop the Electron app.
2. Stop the running web server.

3. Change the code.
4. Start the web server.
5. Start the Electron app.

Let's learn how to improve our setup so that we can automatically rebuild and reload the application when code changes are made.

Configuring Live Reloading

Live Reloading (or Hot Reloading) is a feature that allows you to reload your browser every time code changes. Developers often use this feature when they're building and testing web applications. Given that Electron contains a browser instance, the Live Reloading feature can apply to desktop applications as well.

You already know how to serve an Angular CLI application locally and that you can navigate to `http://localhost:4200` to use it in the browser. The solution for local development is to make your Electron application use the same address to load the main `index.html` content, instead of the prebuilt file. Let's get started:

1. To see this in action, update the `main.js` file according to the following code:

```
// win.loadURL(`file://${__dirname}/dist/integrate-angular
//   /index.html`);

win.loadURL(`http://localhost:4200`);
```

Now, it's time to see the live reloading feature in action.

2. Run the `serve` command and ensure that the application is up and running:

```
npm run serve
```

You should see the following output if everything is OK:

```
Hash: 580d684324c23500227d
Time: 10770ms
chunk {es2015-polyfills} es2015-polyfills.js, es2015-polyfills.js.map
(es2015-polyfills) 284 kB [initial] [rendered]
chunk {main} main.js, main.js.map (main) 11.5 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills)
  236 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.08 kB
[entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 16.7 kB
```

```
[initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.76 MB
[initial] [rendered]
i [wdm]: Compiled successfully.
```

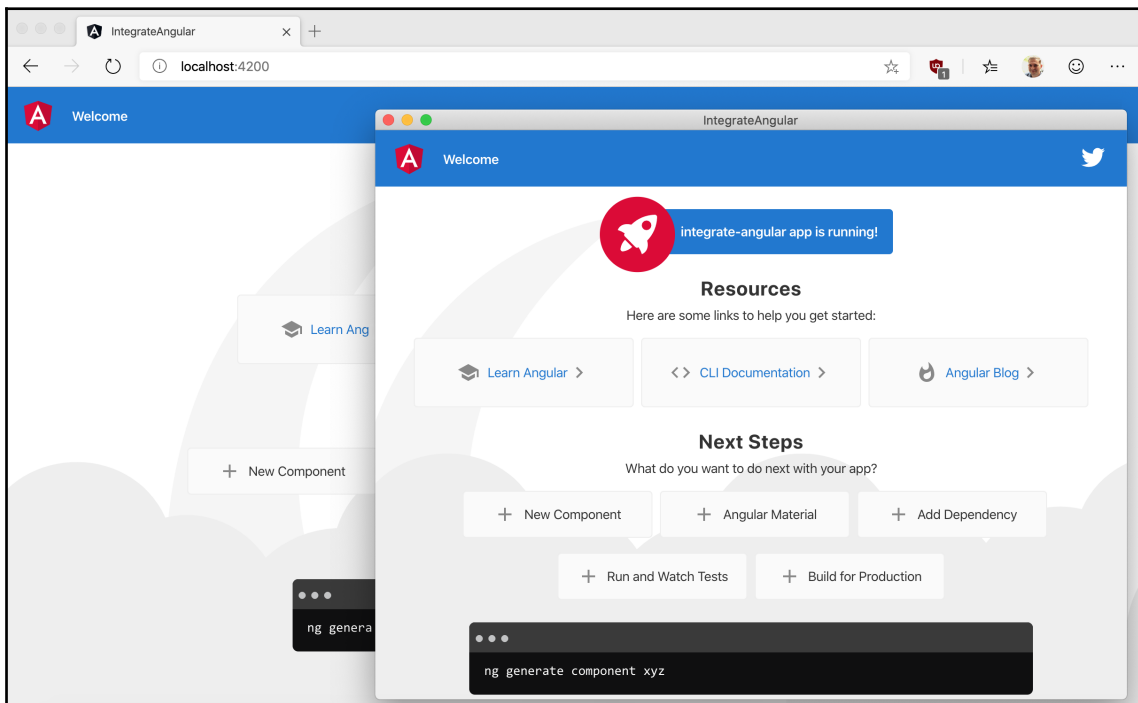
Remember the number of files that Angular CLI builds; we are going to get back to that in a minute.

3. Now, switch to another Terminal window and start the Electron app in parallel to it:

```
npm start
```

Note that the web server should still be running.

4. The good news is that you can test your application against your desktop and browser at the same time. Feel free to open `http://localhost:4200` in your browser. You should see the same code running in both windows, and that both of them are reloaded when changes are made to them:



5. Now, with the browser tab open and the Electron application running, go to the `src/app/app.component.ts` file and change the title by updating its content with the following code:

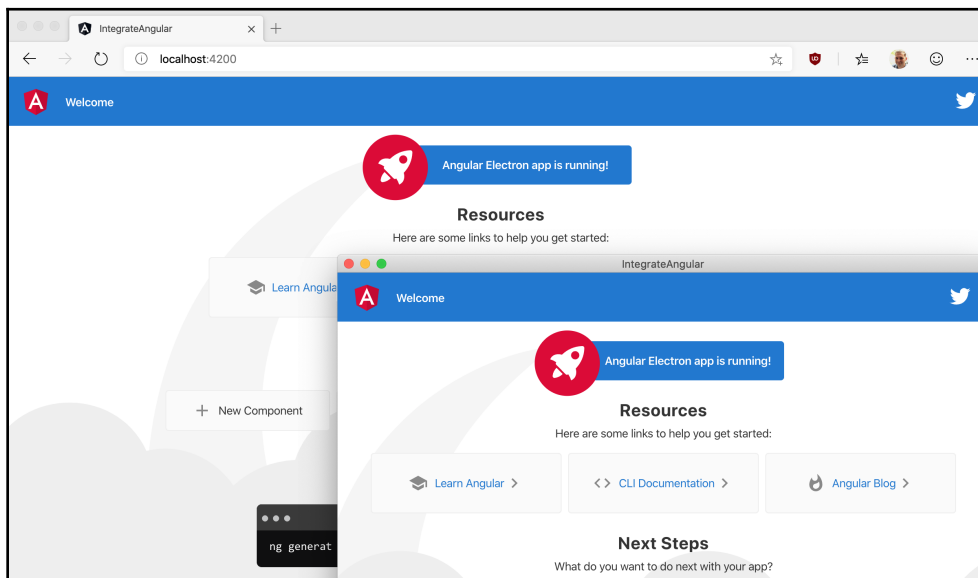
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'Angular Electron';
}
```

6. Take a look at the Terminal instance where the web server is running. Notice that the `main.js` and `main.js.map` files recompile when an update occurs:

```
Hash: 042ed91436c7c2fe2749 - Time: 2046ms
5 unchanged chunks
chunk {main} main.js, main.js.map (main) 11.5 kB [initial] [rendered]
i ｢wdm｣: Compiled successfully.
```

7. If you have a browser or an Electron shell—or both—running, the applications will automatically reload. You should see the updated title:



Why test in the browser?

Often, applications that run in the Electron shell share their code with the web client's implementations. Let's take Slack or Skype as an example. You can use a web client with your browser or download the desktop client based on Electron. The desktop client usually provides better integration with the underlying operating system in terms of file management, downloads, system notifications, and the system tray.

It is good practice to build the web client with a desktop in mind, so you may want to check how well the same code base behaves in the browser. This is why it's very convenient to run both a desktop shell and a web tab.

Setting up production builds

When developing locally, you may want to use live reloading (or hot reloading). However, when packaging for distribution, the application needs to have access to the production output.

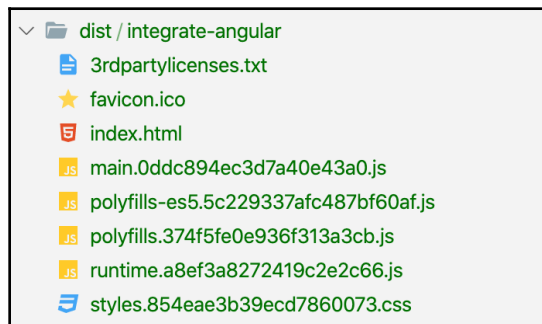
The Angular CLI allows us to quickly compile a web application in production mode. You can do this by running the following command:

```
ng build --prod
```

According to the Angular documentation, the `—prod` switch does the following:

When true, it sets the build configuration to the production target. All builds make use of bundling and limited tree-shaking. A production build also runs limited dead code elimination.

This means that you get a minimal and highly optimized output, as follows:



You will probably run this command often when preparing production builds. It would be a good idea to have a shortcut command to avoid typing out all the parameters. The best way to achieve this is to have an entry in the `package.json` file. Let's create one now:

1. Let's update the `package.json` file and so we have `build.prod` script that contains various flags that will save us time in future:

```
"build.prod": "ng build --prod",
```

2. Don't forget that, for Electron, we also have to change the *base path* property inside the `index.html` file:

```
<base href= "./" />
```

3. The `ng build` command in the Angular CLI provides support for that scenario as well. You can use the `—baseHref` parameter to provide a custom value:

```
"build.prod": "ng build --prod --baseHref= ./",
```

4. Excluding the `dependencies` and `devDependencies` sections, your `package.json` file should now look as follows:

```
{
  "name": "integrate-angular",
  "version": "0.0.0",
  "main": "main.js",
  "scripts": {
    "ng": "ng",
    "serve": "ng serve",
    "start": "electron .",
    "build": "ng build",
    "build.prod": "ng build --prod --baseHref= ./",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
}
```

5. Let's test the whole setup. Switch to your Terminal or console window and run the following command:

```
npm run build.prod
```

6. Check out the `index.html` file in the `dist` folder; it should now contain a custom *base path* value:

```
integrate-angular ▸ dist ▸ integrate-angular ▸ index.html ▸ ...
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <title>Angular with Electron</title>
6      <base href="." />
7      <meta name="viewport" content="width=device-width, initial-scale=1" />
8      <link rel="icon" type="image/x-icon" href="favicon.ico" />
9      <link rel="stylesheet" href="styles.3ff695c00d717f2d2a11.css"></head>
10   <body>
11     <app-root></app-root>
12     <script type="text/javascript" src="runtime.26209474bfa8dc87a77c.js">
13   </script><script type="text/javascript" src="es2015-polyfills.1e04665e16f944715fd2.js" nomodule></script>
14   <script type="text/javascript" src="polyfills.8bbb231b43165d65d357.js"></script>
15   <script type="text/javascript" src="main.11e22453b6987e6500ed.js"></script>
16   </body>
17 </html>
```

In the next section, we are going to learn how to set up conditional loading support.

Setting up conditional loading

Let's review the application startup process:

1. If you switch to the `package.json` file, the start script will look as follows:

```
"start": "electron ."
```

As you already know, Node.js provides us with access to environment variables so that the application can perform different behaviors, depending on external parameters.

2. Supporting the `DEBUG` parameter is a standard practice:

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

For Windows users, the start script should be slightly different:

```
{
  "start": "SET DEBUG=true && electron ."
}
```

3. In this case, our set of scripts will look similar to the following:

```
{
  "scripts": {
    "ng": "ng",
    "serve": "ng serve",
    "start": "DEBUG=true electron .",
    "build": "ng build",
    "build.prod": "ng build --prod --baseHref=.",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
}
```

4. In the `main.js` file, you can now check the `process.env` object, which contains the `DEBUG` value, and perform conditional loading.
5. Here, we need to run this from `http://localhost:4200` for development mode and use `dist/index.html` when in production:

```
if (process.env.DEBUG) {
  // load from running server on port 4200
} else {
  // load production build from the "dist" folder
}
```

6. Update the `main.js` file and make the corresponding changes to the `createWindow` function:

```
function createWindow() {
  win = new BrowserWindow({ width: 800, height: 600 });

  if (process.env.DEBUG) {
    win.loadURL(`http://localhost:4200`);
  } else {
    win.loadURL(`file://${__dirname}/dist/integrate-angular/index.html`);
  }

  win.on('closed', () => {
    win = null;
  });
}
```


7. Now, we are all set. Run `npm run serve` in one Terminal window, wait until the server starts, and then run `npm start` in another window:

```
npm run serve
npm start
```

Since we are running the application with the `DEBUG` parameter, the Electron shell is going to display content from `http://localhost:4200` with live reloading support. This is what we should expect when we're working on the application locally.

Now, it's time to learn how to use the UI toolkit with our Angular project.

Using Angular Material components

In most cases, you aren't going to write every UI component from scratch. There are many libraries that can save you time and allow you to focus on what matters most—the business logic of your application.

There are many component libraries out there. Among the most popular are the following:

- Angular Material, by Google (<https://material.angular.io/>)
- PrimeNG, by PrimeTek (<https://www.primefaces.org/primeng/#/>)

In this section, we are going to use Angular Material by Google, but you can always experiment with additional component libraries later on.

Installing Angular Material into your project is effortless. The `ng add` command is supported by the Angular CLI and is provided by Angular Material, which makes the process straightforward. Let's learn how to install it:

1. Run the following command:

```
ng add @angular/material
```

2. For our current experiment, select the **Indigo/Pink** theme if the Angular CLI asks about prebuilt themes:

```
Q: Choose a prebuilt theme name, or "custom" for a custom theme:
A: Indigo/Pink
```

3. I also suggest enabling HammerJS support for gestures, as well as the browser animations module:

Q: Set up HammerJS for gesture recognition?

A: Y

Q: Set up browser animations for Angular Material?

A: Y

4. Once you've answered all the questions, the Angular CLI will install the corresponding dependencies and even modify a few files to integrate the library with your project:

```
UPDATE src/main.ts (391 bytes)
UPDATE src/app/app.module.ts (502 bytes)
UPDATE angular.json (4158 bytes)
UPDATE src/index.html (522 bytes)
UPDATE src/styles.scss (181 bytes)
```

You can take a look at the files the Angular CLI generated for you and check out their content if you wish. In the next section, we are going to discuss the project's structure and the modifications that the Angular CLI makes now that we've installed Angular Material.

Modifications made by installing Angular Material

Let's quickly review what modifications the Angular CLI makes to install the Angular Material library. This is quite common when we install third-party libraries that don't support `ng add` schematics. In this section, we'll take a look at what we should do when we see a new component library for Angular. Let's take a look at these modifications:

1. The `angular.json` file gets an extra reference in the `styles` array:

```
"styles":
[
  "./node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
  "src/styles.scss"
],
```

2. The `index.html` page now contains two extra fonts, Roboto and Material Icons. Check out <https://material.io/tools/icons> to see what icons are available for your application when you install Angular Material components such as MatIcon:

```
<link href="https://fonts.googleapis.com/css?
family=Roboto:300,400,500" rel="stylesheet">
```

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
      rel="stylesheet">
```

3. The `main.ts` file gets an additional import for the `hammerjs` library. This is a prerequisite for various Angular Material components when we need to deal with various touch and gesture operations.
4. The `styles.scss` files has been updated with some minor enhancements to the application style and to the new default font that points to Roboto:

```
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue",
      sans-serif; }
```

5. Finally, the `src/app/app.module.ts` file now imports and uses `BrowserAnimationsModule`.

As you can see, you have just managed to update numerous files by running a single `ng add` command. In the future, you may see more and more libraries supporting this form of installation since it's straightforward and convenient compared to the manual setup option.

Now, let's look at how to use components from the Angular Material package.

Adding the Material Toolbar component

The simplest thing to start with is the Toolbar component. You can find details about the API, as well as examples of how to use it, here: <https://material.angular.io/components/toolbar>.

Let's learn how to easily integrate the Material Toolbar component:

1. Open the `src/app/app.module.ts` file for editing and append the `MatToolbarModule` import to the top of the file, in the import section:
2. However, importing a type from the Angular Material library isn't enough. You also need to register with the application module's imports:

```
@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
```

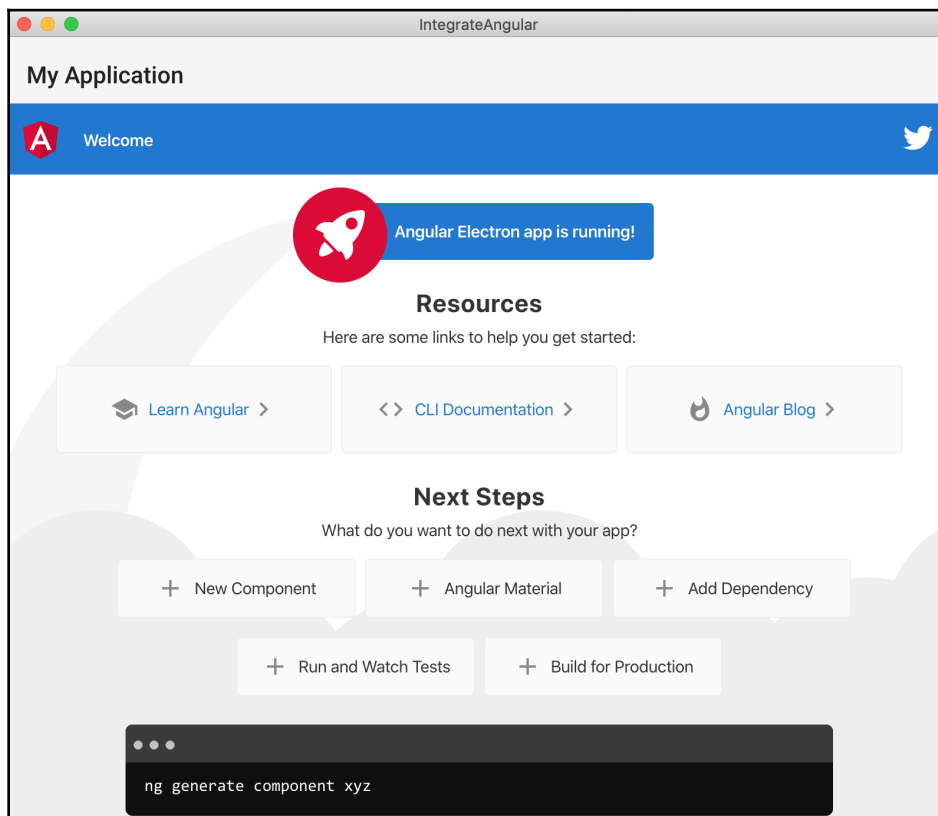
```
MatToolbarModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule {}
```

Now, you can use the `<mat-toolbar>` component in your HTML templates. Let's see how this works:

1. Switch to the `src/app/app.component.html` file and put the following code at the top of the file:

```
<mat-toolbar>
<span>My Application</span>
</mat-toolbar>
```

2. At runtime, your Electron application should now look as follows:



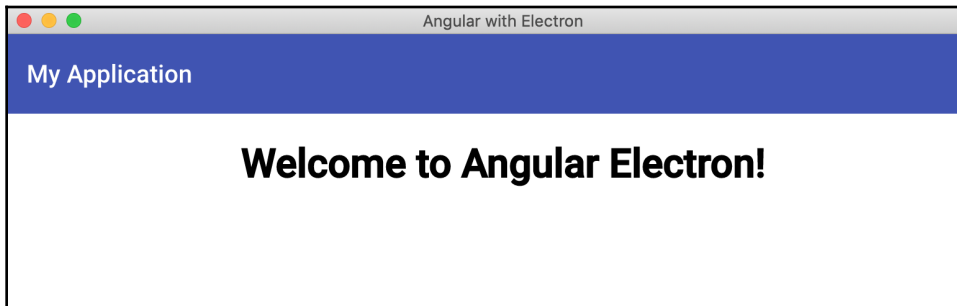
3. You can also clean up the HTML template a bit. The main things we need to retain are our Toolbar, the **Welcome** label, and the Router Outlet component:

```
<mat-toolbar color="primary">
  <span>My Application</span>
</mat-toolbar>

<div style="text-align:center">
  <h1>Welcome to {{ title }}!</h1>
</div>

<router-outlet></router-outlet>
```

4. Restart the application if you aren't using Live Reloading. The main page will now look as follows:



Most applications consist of multiple views. You're probably going to have different pages, such as *Settings*, *About*, and more. For this situation, Angular provides support for routing or linking parts of the URL address to particular components.

Angular routing

We are about to implement two routes to see routing in action. The first route is our main page, which shows the *Welcome* screen. The second page shows the *About* screen. We are also going to need some links or buttons to switch between the two pages.

Since the Angular CLI provides the `ng generate component` command, which creates a new component scaffold with all the necessary files and project modifications, let's generate an *About* page first:

1. Run the following command:

```
ng generate component about
```

Pay attention to the command's output. As a result, you get a Typescript file with the component; a `spec.ts` file, which contains a unit testing placeholder; and an HTML template, which contains the SCSS stylesheet. The Angular CLI also updates the main application module in the `app.module.ts` file to include the newly generated component in the application's structure.

2. The output should be similar to the following:

```
CREATE src/app/about/about.component.scss (0 bytes)
CREATE src/app/about/about.component.html (24 bytes)
CREATE src/app/about/about.component.spec.ts (621 bytes)
CREATE src/app/about/about.component.ts (266 bytes)
UPDATE src/app/app.module.ts (651 bytes)
```

Once our new About component is ready, we can register an application route to display it to the user.

3. Update the `src/app/app-routing.module.ts` file and add a new entry to the routes array, as shown in the following code:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AboutComponent } from './about/about.component';

const routes: Routes = [
  {
    path: 'about',
    component: AboutComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

The preceding code introduces a new URL fragment, `/about`, which displays the About component when we visit that link. We are going to see this in action shortly, but first let's add some buttons to allow users to switch between screens.

4. Update the `app.module.ts` file and import the `MatButtonModule` and `MatIconModule` modules from Angular Material. Don't forget to provide them inside the module's `imports` section as well:

```
import { MatButtonModule } from '@angular/material/button';
import { MatIconModule } from '@angular/material/icon';

@NgModule({
  declarations: [AppComponent, AboutComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserModuleAnimationsModule,
    MatToolbarModule,
    MatButtonModule,
    MatIconModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

5. The Angular Material library provides different kinds of buttons and button styling. For the sake of simplicity, let's use the Icon Button component with the `help_outline` image from Material Icons.
6. Update your HTML code according to the following listing:

```
<mat-toolbar color="primary">
  <span>My Application</span>

  <span class="spacer"></span>

  <button mat-icon-button>
    <mat-icon>help_outline</mat-icon>
  </button>
</mat-toolbar>

<!--The content below is only a placeholder and can be
  replaced.-->
<div style="text-align:center">
  <h1>Welcome to {{ title }}!</h1>
</div>

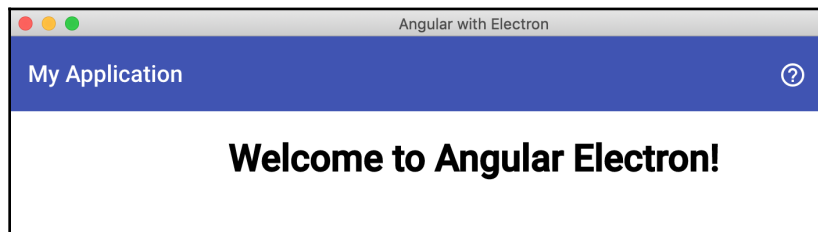
<router-outlet></router-outlet>
```

Note the use of an extra `span` element within the `spacer` class. We need it to take up space in the middle of the toolbar and move the buttons to the right corner while leaving the application title on the left. This is a typical layout for application toolbars or menus.

7. To make our `spacer` work as expected, we also need to update the `app.component.scss` file and declare the following style:

```
.spacer {  
  flex: 1 1 auto;  
}
```

8. If you run the application now, you should see a nice-looking application toolbar with a `?` button on the right-hand side:



You may have noticed, however, that the button on the toolbar does nothing so far. The application title label also does nothing. Traditionally, the application's title redirects the user to the main page, that is, the home page.

The Angular Router component allows us to map buttons to particular routes and hides all the complexity related to navigation. You can use the `routerLink` attribute with your components to inform the router that this button expects navigation to occur.

Let's learn how to use attributes to configure routing in our application:

1. Update your HTML template according to the following code:

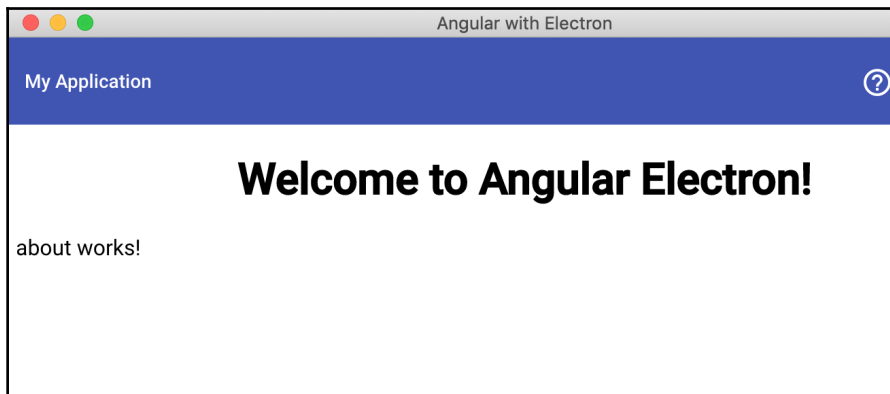
```
<mat-toolbar color="primary">  
  <button mat-button routerLink="/">My Application</button>  
  <span class="spacer"></span>  
  
  <button mat-icon-button routerLink="/about">  
    <mat-icon>help_outline</mat-icon>  
  </button>  
</mat-toolbar>  
  
<div style="text-align:center">
```



```
<h1>Welcome to {{ title }}!</h1>  
</div>
```

```
<router-outlet></router-outlet>
```

2. Switch to your running application and try to click on the ? button again. Notice that the main content area underneath the **Welcome** label changes to **about works!**. This string is part of the auto-generated About component, and it proves that the About route works for us:



3. Click the **My Application** label to get back to the home page.
4. Notice, however, that extra content gets displayed within the `<router-outlet>` tags:

```
<router-outlet></router-outlet>
```

The **Welcome to Angular Electron** heading is placed above the router outlet so that you can see it on every page. Note that you have static content that should be visible on every page, such as the toolbar component, and some dynamic content that changes according to the user's actions.

As an exercise, let's move the heading into a separate `Home` component so that we can see the `About` and `Home` content take up all the space:

1. Generate a new component that will store the content of the `Home` page:

```
ng generate component home
```

Like the previous component, the Angular CLI generates output so that you can see what files have been affected by your change. Also, if you are using a source control program, such as Git, you can always roll-back these changes.

The output from using the preceding command is as follows:

```
CREATE src/app/home/home.component.scss (0 bytes)
CREATE src/app/home/home.component.html (23 bytes)
CREATE src/app/home/home.component.spec.ts (614 bytes)
CREATE src/app/home/home.component.ts (262 bytes)
UPDATE src/app/app.module.ts (878 bytes)
```

2. Proceed to the `app.routing-module.ts` file and prepend the `Home` route to the `routes` collection:

```
import { AboutComponent } from './about/about.component';
import { HomeComponent } from './home/home.component';

const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'about',
    component: AboutComponent
  }
];
```

Note that we are using an empty path value this time. This means that the `HomeComponent` will be displayed on the default application path, for example, at `http://localhost:4200`.

3. Now, we need to move the **Welcome** label to the new location in the `home.component.ts` file. Update the `src/app/home/home.component.html` template according to the following code:

```
<div style="text-align:center">
  <h1>Welcome to {{ title }}!</h1>
</div>

<p>
  home works!
</p>
```

4. The component template relies on the `title` property, so don't forget to move that property from the `app.component.ts` file to the `home.component.ts` file as well:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit {
  title = 'Angular Electron';

  constructor() {}

  ngOnInit() {}
}
```

5. Finally, we can clean up the main application component and leave only the toolbar and the router outlet, which are our placeholders for the active route's content:

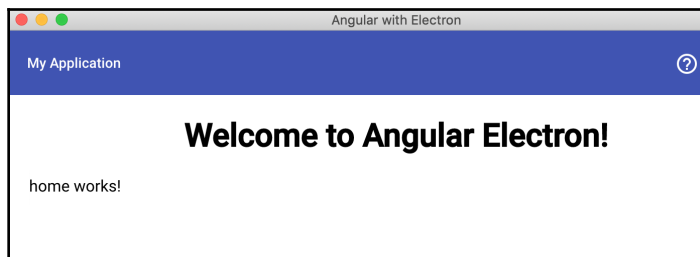
```
<mat-toolbar color="primary">
  <button mat-button routerLink="/">My Application</button>

  <span class="spacer"></span>

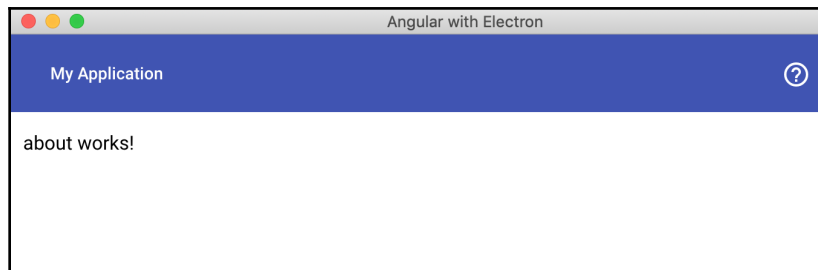
  <button mat-icon-button routerLink="/about">
    <mat-icon>help_outline</mat-icon>
  </button>
</mat-toolbar>

<router-outlet></router-outlet>
```

6. At runtime, you should see the same **Welcome** heading and a **home works!** string:



This time, however, all the text on the screen comes from the `Home` component that we loaded for the default route. Click on the `?` button once again to ensure that you can see the `About` screen taking up the content area:



At this point, you are ready to build multi-page applications and display different areas by navigating users via the Angular Routing system.

The application we have just built can share the same code base as the web client.

Now, you know how to build Electron applications with the Angular framework and Google's ecosystem of components and libraries. In the next section, we are going to walk through the process of setting up React projects and using the Facebook ecosystem.

Building an Electron application with React

React is a highly popular view library for building web applications. It is maintained by Facebook and has a huge community of developers. There are lots of component libraries, tutorials, blog posts, and other informational resources available in the internet.

If you decide to build an Electron application with the React library, you are going to find a lot of reusable code and resources that will be very useful and save you time.

Now, let's see what it takes to integrate the React library with an Electron application.

Generating a React project

Perform the following steps to learn how to create a React project:

1. Similarly to the Angular CLI, React has its own application scaffold generator, `Create React App`. You can generate a new project by running the following command:

```
npx create-react-app integrate-react
```

2. You should see the following output:

```
Success! Created integrate-react at <path>/integrate-react
Inside that directory, you can run several commands:
```

```
yarn start
Starts the development server.
```

```
yarn build
Bundles the app into static files for production.
```

```
yarn test
Starts the test runner.
```

```
yarn eject
Removes this tool and copies build dependencies,
configuration files
and scripts into the app directory. If you do this, you can't
go back!
```

We suggest that you begin by typing:

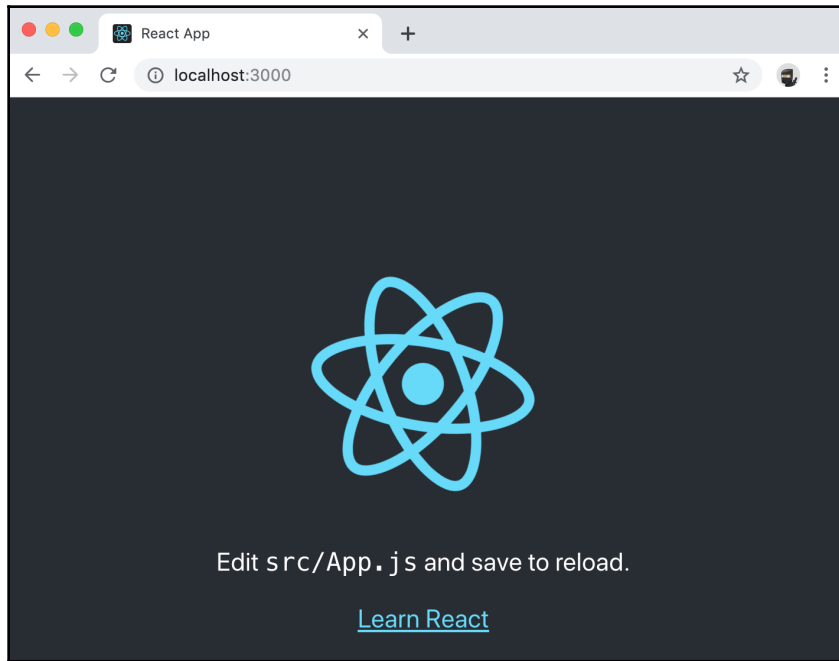
```
cd integrate-react
yarn start
```

Note that `create-react-app` uses the Yarn package manager. However, you can still use the same commands with `npm run` if you wish; for example, `npm run build` instead of `yarn build`, and so on. Alternatively, you can install Yarn and use its commands. You can find out more about Yarn at <https://yarnpkg.com>.

3. Run the application with the `start` command to see if it's working:

```
cd integrate-react/
npm start
```

4. Run your preferred browser and navigate to `http://localhost:3000`. You should see the following screen with the animated React logo:



5. Press `Ctrl + C` to stop the server. I always recommend checking the `package.json` scripts after generating a project. They provide us with a clear understanding of what actions the project supports out of the box.

When generated with the `Create React App` application, the `scripts` section of the `package.json` file looks as follows:

```
{
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
}
```

6. The next step is to install the `electron` library as a development dependency. The following command should be familiar to you from the previous chapters:

```
npm i -D electron
```

7. Now, we need to make modifications to the `package.json` file. First of all, add the `main` entry so that it's pointing to the `main.js` file:

```
{  
  "main": "main.js"  
}
```

8. Next, we need to set the base path of the application so that it's relative to the `index.html` file. In Angular projects, you did this by modifying the `index.html` file. React, however, supports the `homepage` field in the `package.json` file.



You can find out more about React and relative paths at <https://facebook.github.io/create-react-app/docs/deployment#building-for-relative-paths>.

9. This will make sure that all asset paths are relative to `index.html`. Now, you will be able to move your app from `http://mywebsite.com` to `http://mywebsite.com/relativepath` or even `http://mywebsite.com/relative/path` without having to rebuild it.
10. Modify the `package.json` file according to the following code:

```
{  
  "homepage": ".",  
  "scripts": {  
    "serve": "react-scripts start",  
    "start": "electron .",  
    "build": "react-scripts build",  
    "test": "react-scripts test",  
    "eject": "react-scripts eject"  
  },  
}
```

As you can see, we have added a `homepage` attribute, renamed the `start` script to `serve`, and introduced a new `start` one to launch the Electron shell.

11. The content of a minimal `main.js` file implementation is pretty much standard for every framework:

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

let win;

function createWindow() {
  win = new BrowserWindow({ width: 800, height: 600 });

  win.loadFile('index.html');

  win.on('closed', () => {
    win = null;
  });
}

app.on('ready', createWindow);

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

app.on('activate', () => {
  if (win === null) {
    createWindow();
  }
});
```

12. When you build a React application, the production output goes into the `build` folder. This is why we need to load the `index.html` file from there:

```
function createWindow() {
  win = new BrowserWindow({ width: 800, height: 600 });

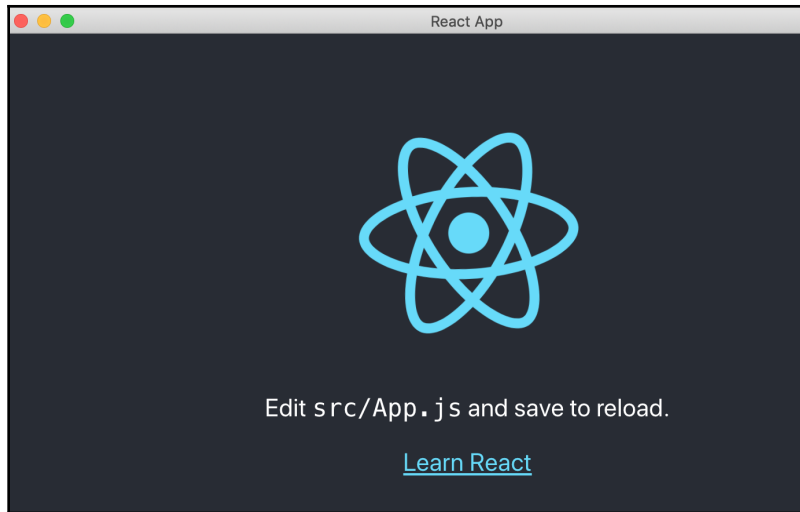
  // win.loadFile('index.html');
  win.loadURL(`file://${__dirname}/build/index.html`);

  win.on('closed', () => {
    win = null;
  });
}
```


13. Before we move on, let's verify that the application builds and runs fine. In the Terminal window, run the following commands, one by one:

```
npm run build
npm start
```

14. You should see the default application up and running on your desktop inside an Electron shell, as shown in the following screenshot:



You have successfully created an initial React-based Electron application. Feel free to make a backup so that you can use it as a template for future applications with the same stack.

Now, we need to wire the Electron shell with the locally running web server so that we can test our application while we develop it. This is called **live reloading**, and we are going to configure it in the next section.

Live reloading

Instead of stopping and restarting the application each time you need to verify a code change, you can enable the live reloading feature and have the browser or Electron window automatically refresh as you save the changes.

When you start the application with the `npm run serve` script (or `npm start` by default), you will see the following output:

You can now view integrate-react in the browser.

Local: `http://localhost:3000/`
On Your Network: `http://192.168.0.10:3000/`

Note that you need to use port 3000 in this case. Let's update the `createWindow` function accordingly:

1. Open the `main.js` file and update the `createWindow` function according to the following code:

```
function createWindow() {  
  win = new BrowserWindow({ width: 800, height: 600 });  
  
  // win.loadURL(`file://${__dirname}/build/index.html`);  
  win.loadURL(`http://localhost:3000`);  
  
  win.on('closed', () => {  
    win = null;  
  });  
}
```

To verify the application, you are going to need two Terminal windows.

2. In the first Terminal window, run the `serve` command, as follows:

`npm run serve`

3. Give the web server a few seconds to start and run the following command in the second Terminal:

`npm start`

You should see an Electron window showing the React home page.

4. Now, go to the `src/App.js` file and edit its content to see the live reload feature in action. For instance, insert the **React Electron** label:

```
import React, { Component } from 'react';  
import logo from './logo.svg';  
import './App.css';  
  
class App extends Component {  
  render() {
```

```
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1>React Electron</h1>
          <p>
            Edit <code>src/App.js</code> and save to reload.
          </p>
          <a
            className="App-link"
            href="https://reactjs.org"
            target="_blank"
            rel="noopener noreferrer"
          >
            Learn React
          </a>
        </header>
      </div>
    );
  }
}

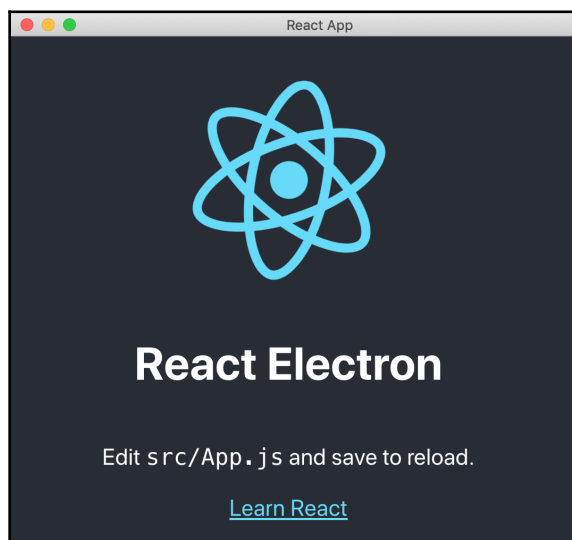
export default App;
```

Please note that a typical React application requires a single element tag so that it can wrap the rest of the components, similar to the following structure:

```
class App extends Component {
  render() {
    return (
      <div>...</div>
    );
  }
}
```

All the examples in this chapter assume that you have a root element in the `App` component.

5. Switch to the Electron window and notice how it updates on the fly with the new text:



You have successfully configured live reloading for your Electron application. This should help you develop and test your project fast as you'll be able to see changes almost instantly.

When development is over and you need to distribute the application, live reloading is not needed. Instead, you need to have a production build output with all the static assets for the application. This is what we are going to address in the next section.

Setting up production builds

Unlike the Angular CLI, you don't need extra scripts to perform a production build with the *Create React App* tool. You should already have the `build` command in your `package.json` file for that:

```
{
  "scripts": {
    "serve": "react-scripts start",
    "start": "electron .",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
}
```

Whenever you need to perform a production build, use the following command in the Terminal:

```
npm run build
```

In this case, the output should look as follows:

```
Creating an optimized production build...
Compiled successfully.
```

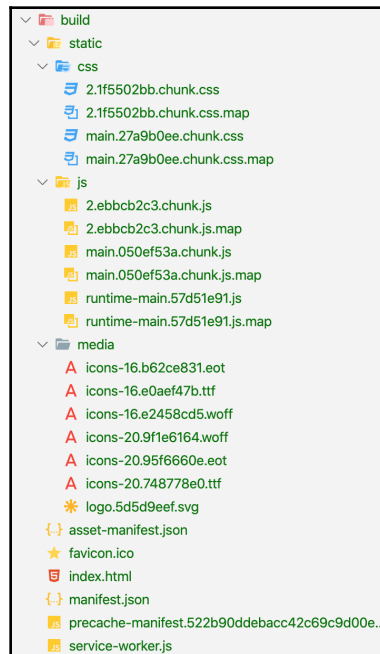
File sizes after gzip:

```
36.83 KB build/static/js/2.6efc73d3.chunk.js
763 B   build/static/js/runtime~main.d653cc00.js
725 B (+15 B) build/static/js/main.dff8d9a2.chunk.js
540 B   build/static/css/main.0e186509.chunk.css
```

The project was built assuming it is hosted at `./`.

You can control this with the `homepage` field in your `package.json`.

As you can see, all the production assets reside in the `build` folder. Now, you can distribute or host the resulting web application without the need for Node.js or any other tool:



Many developers prefer to host web applications in a local server to greatly reduce the time they need to spend testing the end result. You may also want to follow that path and only have the build folder before you package your Electron application for the final testing and publishing stages.

Now, let's learn how to configure the use of local web servers for development purposes.

Setting up conditional loading

If you have read the *Building an Electron application with Angular* section, the process of setting up conditional loading is going to be familiar to you.

In this section, we are going to focus on running the web server with the application in the way that Create React App does. Then, we will render our locally running application from within the Electron shell when developing, debugging, and testing on the local machine. For production mode, however, the application is going to use production output.

Let's configure the package scripts so that we don't have to use as many parameters during the development and testing phase:

1. Update the `scripts` section in the `package.json` file according to the following code:

```
{
  "scripts": {
    "serve": "react-scripts start",
    "start": "DEBUG=true electron .",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
}
```

For Windows users, the `start` script should be as follows:

```
{
  "start": "SET DEBUG=true && electron ."
}
```

2. The format of the call is simple: depending on the value, you either use a web server on port 3000 or load the precompiled `index.html` file:

```
if (process.env.DEBUG) {  
  // load from running server on port 3000  
} else {  
  // load production build from the "build" folder  
}
```

3. Now, update the `main.js` file and make the following changes to the `createWindow` function:

```
function createWindow() {  
  win = new BrowserWindow({ width: 800, height: 600 });  
  
  if (process.env.DEBUG) {  
    win.loadURL(`http://localhost:3000`);  
  } else {  
    win.loadURL(`file://${__dirname}/build/index.html`);  
  }  
  
  win.on('closed', () => {  
    win = null;  
  });  
}
```

Note that, traditionally, Angular CLI applications run on port 4200; the **Create React App** tool prefers port 3000.

4. Now, it's time to test how things work. Run the following two commands in separate Terminal windows:

```
npm run serve  
npm start
```

Since we are running the application with the `DEBUG` parameter, the Electron shell is going to display content from `http://localhost:3000` with live reloading support.

Now, let's move on to the user interface and integrate the Blueprint component library.

Using the Blueprint UI toolkit

For Angular, it is recommended that you use an Angular Material library of components for any UI toolkit. For React.js, I strongly recommend starting with the Blueprint toolkit, which you can find at <https://blueprintjs.com/>.

Blueprint is an open source project that's developed by Palantir. It's a React-based UI toolkit for the web and has been optimized for building complex data-dense interfaces for desktop applications.

Run the following command to install the library for your project:

```
npm i @blueprintjs/core
```

Now we have the Blueprint library installed, let's start using it in our application.

Adding an application menu

To see how easy it is to consume Blueprint components, let's add an application menu or application toolbar:

1. First, append the following code to the `index.css` file:

```
@import '~normalize.css';
@import '~@blueprintjs/core/lib/css/blueprint.css';
@import '~@blueprintjs/icons/lib/css/blueprint-icons.css';
```

2. Then, go to the `src/App.js` file and import the following types from the `@blueprintjs/core` package:

```
import { Navbar, Button, Alignment } from '@blueprintjs/core';
```

3. This allows you to use the `Navbar` and `Button` components, as well as the `Alignment` enumerable, in your JSX templates. Update the code according to the following listing:

```
<Navbar>
  <Navbar.Group align={Alignment.LEFT}>
    <Navbar.Heading>Blueprint</Navbar.Heading>
    <Navbar.Divider />
    <Button className="bp3-minimal" icon="home" text="Home" />
    <Button className="bp3-minimal" icon="document" text="Files" />
  </Navbar.Group>
</Navbar>
```


4. At runtime, you should see a nice-looking toolbar with a Blueprint label and two buttons, Home and Files:



If you like dark themes, you will be pleased to know that Blueprint supports them. Given that the default React template is dark, let's update the Navbar to match the theme:

1. All you need to do is add the `bp3-dark` class name to the navbar:

```
<Navbar className="bp3-dark">
  <Navbar.Group align={Alignment.LEFT}>
    <Navbar.Heading>Blueprint</Navbar.Heading>
    <Navbar.Divider />
    <Button className="bp3-minimal" icon="home" text="Home" />
    <Button className="bp3-minimal" icon="document" text="Files" />
  </Navbar.Group>
</Navbar>
```

2. Check out your application. Notice that the application menu bar now has a dark theme:



Now, let's see what it takes to enable routing features for a React application and Electron shell.

Adding routing

The tool we are going to use is called React Router, maintained by React Training (<https://reacttraining.com/react-router/web>).

Integrating routing support is straightforward. Let's learn how to do this for our Electron project:

1. Stop the application if it's running and install the library using the following command:

```
npm install react-router-dom
```

2. Now, we are going to have some simple Screens or Views backed by two routes: Index and Files. Update the `App.js` file with the following functional components:

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

function Index() {
  return <h2>Home</h2>;
}

function Files() {
  return <h2>Files</h2>;
}
```

3. We have just imported a few types from the `react-router-dom` package. One of these is a `Link` component, which allows us to navigate to a particular route. For example, you can create a `Hyperlink` element that points to the `/files/` URL fragment using the following syntax:

```
<Link to="/files/">Files</Link>
```

4. First, update the navbar code according to the following listing:

```
<Navbar className="bp3-dark">
  <Navbar.Group align={Alignment.LEFT}>
    <Navbar.Heading>Blueprint</Navbar.Heading>
    <Navbar.Divider />
    <Button className="bp3-minimal" icon="home">
      <Link to="/">Home</Link>
    </Button>
    <Button className="bp3-minimal" icon="document">
      <Link to="/files/">Files</Link>
    </Button>
  </Navbar.Group>
</Navbar>
```

5. Now, modify its main content, as follows:

```
<header className="App-header">
  <img src={logo} className="App-logo" alt="logo" />

  <Route path="/" exact component={Index} />
  <Route path="/files/" component={Files} />
</header>
```

6. Run the application and try clicking the `Home` and `Files` links in the application menu component. Notice how the content of the page reflects the route you click:



At this point, you have a nice Electron application based on React with routing and UI library support up and running. Feel free to make a backup of this project in its current state so that you can use it for similar applications in the future.

Let's see what else we can do to improve the look and feel of the application.

Final touches

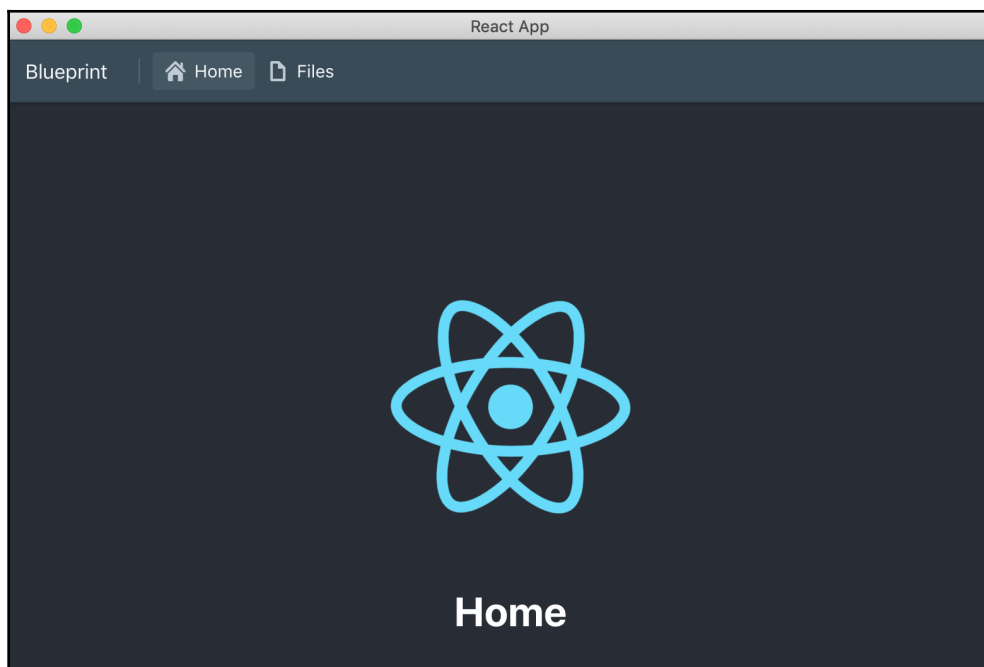
You may have noticed that the toolbar buttons don't look very good now that we've wired them with the `Link` component. You don't need to have button components here since `Blueprint` allows us to style other elements, such as hyperlinks, so that we can make them look like buttons.

To use `Link` components without button wrappers, we need to update the code for the application component so that it looks as follows:

```
<Link
  className="bp3-button bp3-minimal bp3-icon-home"
  to="/">
  Home
</Link>

<Link
  className="bp3-button bp3-minimal bp3-icon-document"
  to="/files/">
  Files
</Link>
```

Now, the application toolbar should look much better:



Congratulations—you have an initial React application project wired with the Electron shell! You have also integrated an external UI toolkit to save time creating components and focus on the business logic.

In the next section, we are going to build a similar application with another highly popular framework: Vue.js. You are going to explore the differences in project configuration and libraries for the user interface and decide which application stack to use for production.

Building an Electron application with Vue.js

We've already looked at how to use Electron applications with the Angular and React frameworks. Now, it's time to look at another popular framework: Vue.js. This framework will help you boost your productivity when building desktop applications with web technologies.

Similar to the Angular CLI and Create React App, Vue.js has its own CLI tool too. You can install it and generate a new application called `integrate-vue` by following these steps:

1. Run the following two commands:

```
npm install -g @vue/cli
vue create integrate-vue
```



If you are a Windows user, please refer to the Vue documentation on how to set up commands: <https://cli.vuejs.org/guide/creating-a-project.html#vue-create>.

2. The tool is going to ask you a few questions so that it can figure out the final configuration of the project. For **preset**, please select the **default** option. You can use the **Manually select features** option later:

```
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
  Manually select features
```

3. There are two different Node package managers that the community uses nowadays: NPM and Yarn. Vue.js is going to ask you to pick your favorite one. We will use it later to install dependencies:

```
? Pick the package manager to use when installing dependencies:
(Use arrow keys)
  Use Yarn
> Use NPM
```

4. We are going to use NPM, so select the corresponding entry and press *Enter*.
5. Depending on the tool's version, you may see a lot of output during the process of project generation. Make sure you see a successful-completion output, similar to the one shown in the following code:

```
Successfully created project integrate-vue.  
Get started with the following commands:
```

```
$ cd integrate-vue  
$ npm run serve
```

6. Now, open to the `integrate-vue` folder in the Visual Studio Code
7. Let's inspect the scripts that are available in the `package.json` file out of the box:

```
{  
  "name": "integrate-vue",  
  "version": "0.1.0",  
  "private": true,  
  "scripts": {  
    "serve": "vue-cli-service serve",  
    "build": "vue-cli-service build",  
    "lint": "vue-cli-service lint"  
  },  
}
```

Here, we have three main scripts so that we can serve the web application locally, build it for distribution, and perform code quality checks and linting.

8. Let's start the application and verify that the project compiles and runs as expected:

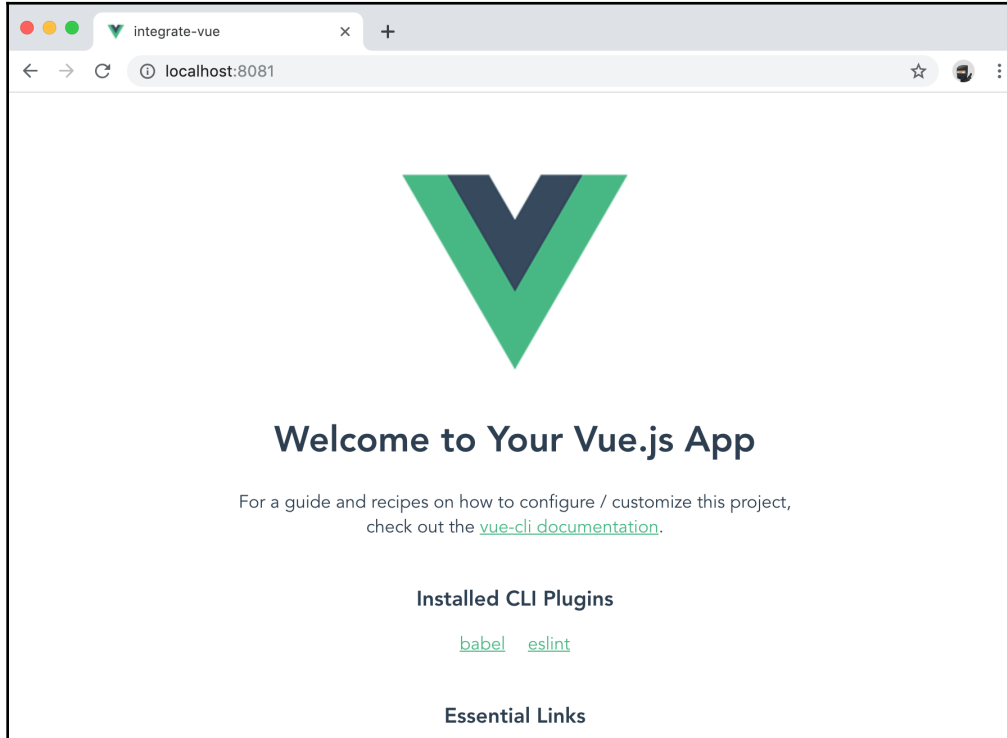
```
npm run serve
```

9. Your output should look similar to the following. Please take note of the URL address and port number of the application:

```
App running at:  
- Local: http://localhost:8080/  
- Network: http://192.168.0.10:8080/
```

```
Note that the development build is not optimized.  
To create a production build, run npm run build.
```

10. Use your favorite browser and navigate to `http://localhost:8080`. Verify that you can see a standard home page for all the projects you generated with the Vue CLI:



Now, it's time to install the `Electron` dependency:

1. Use the following command to grab the most recent version that's available on the NPM registry:

```
npm i -D electron
```

2. The next step is to configure the `main` entry in the `package.json` file so that it points to the `main.js` file. This is how `Electron` finds the main entry point and execute it on startup:

```
{
  "name": "integrate-vue",
  "version": "0.1.0",
  "private": true,
  "main": "main.js",
```



```
"scripts": {  
  "serve": "vue-cli-service serve",  
  "build": "vue-cli-service build",  
  "lint": "vue-cli-service lint"  
},  
}
```

3. Finally, as you may already know from the previous chapters, we need to put a minimal `main.js` implementation in the project's root folder:

```
const { app, BrowserWindow } = require('electron');  
  
let win;  
  
function createWindow() {  
  win = new BrowserWindow({ width: 800, height: 600 });  
  
  win.loadFile('index.html');  
  
  win.on('closed', () => {  
    win = null;  
  });  
}  
  
app.on('ready', createWindow);  
  
app.on('window-all-closed', () => {  
  if (process.platform !== 'darwin') {  
    app.quit();  
  }  
});  
  
app.on('activate', () => {  
  if (win === null) {  
    createWindow();  
  }  
});
```

4. The Vue CLI generates a start script that runs a local web server for testing and development purposes. Given that our primary focus is on Electron and desktop development, I recommend renaming the existing `start` script to `serve` and using a new `start` implementation that launches the Electron shell:

```
{  
  "scripts": {  
    "serve": "vue-cli-service serve",  
    "start": "electron .",  
  },  
}
```

```
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
}
```

5. As soon as you run the `npm run build` command, the Vue CLI is going to perform a production build with the output artifacts that live in the `dist` folder. This means that our Electron shell needs to load the `dist/index.html` file at runtime when we launch it locally:

```
function createWindow() {
  win = new BrowserWindow({ width: 800, height: 600 });

  // win.loadFile('index.html');
  win.loadURL(`file://${__dirname}/dist/index.html`);

  win.on('closed', () => {
    win = null;
  });
}
```

Now, let's take a quick look at the Vue configuration files and how we can change the base path of our application.

Creating a Vue configuration file

The next step is to create a Vue configuration file, `vue.config.js`. Perform the following steps to do so:



According to the official documentation (<https://cli.vuejs.org/config/#vue-config-js>), `vue.config.js` is an optional config file that will be automatically loaded by `@vue/cli-service` if it's present in your project root (next to `package.json`). You can also use the `vue` field in `package.json`, but note that you will be limited to JSON-compatible values only.

1. Let's start with the basic configuration stub, as shown in the following code:

```
// vue.config.js
module.exports = {
  // options...
}
```



You can find out more about supported parameters in the official documentation: <https://cli.vuejs.org/config/#global-cli-config>.

2. The field we are looking for is called `publicPath`. We need to switch the value so that the relative paths work correctly in the Electron shell:

```
// vue.config.js
module.exports = {
  publicPath: './'
};
```

3. Now, run the production build with the following command:

```
npm run build
```

4. The output should be similar to the following:

```
⌚ Building for production...
DONE Compiled successfully in 6389ms

File                                Size                                Gzipped
dist/js/chunk-vendors.ecd76ec1.js   82.81 KiB                          29.94 KiB
dist/js/app.60bf8267.js             4.60 KiB                           1.65 KiB
dist/css/app.e2713bb0.css           0.33 KiB                           0.23 KiB

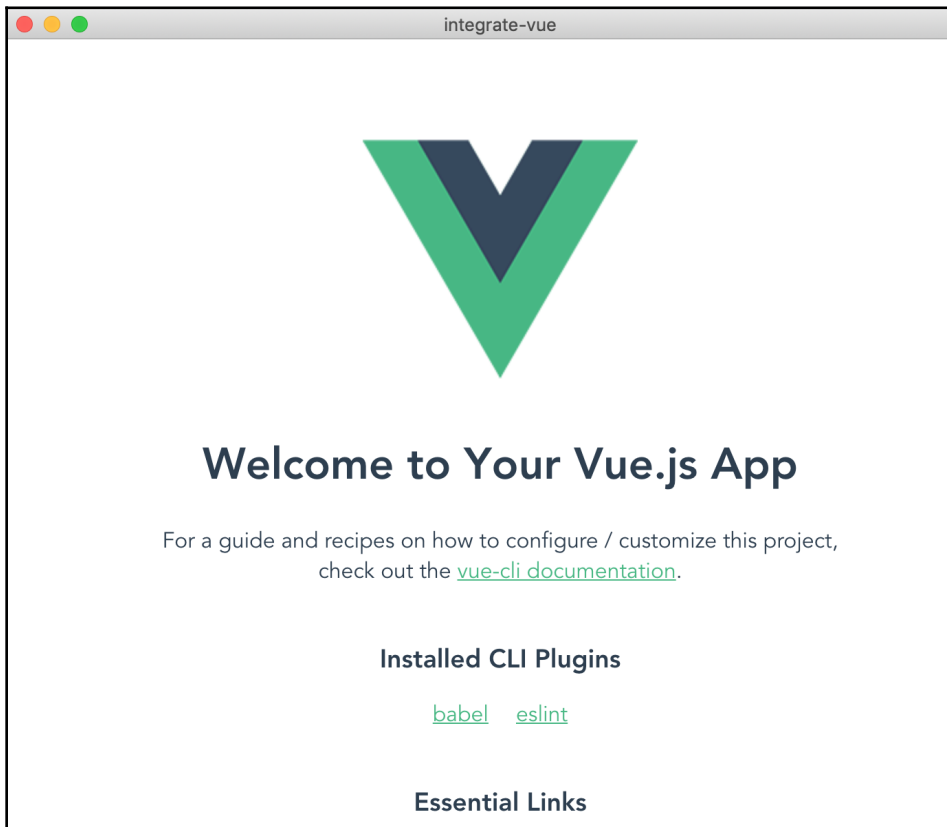
Images and other types of assets omitted.

DONE Build complete. The dist directory is ready to be deployed.
INFO Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html
```

5. To verify that the application loads as expected, run the start script:

```
npm start
```

6. After a few seconds, you should see the Electron application with a traditional UI example that the Vue CLI generates for you:



You're doing great! Now, it's time to simplify our lives as developers by enabling the live reloading feature.

Live reloading

As you may already know, the CLI tool for each popular framework provides facilities so that we can serve the web application locally for testing and development purposes. In the case of Vue, we execute the `npm run serve` script and usually get the following output:

```
App running at:
- Local: http://localhost:8081
- Network: http://192.168.0.10:8081
```

The traditional ports that are used across different frameworks are as follows:

- Angular: 4200
- React: 3000
- Vue: 8080

However, for demonstration purposes, port 8080 is busy with another application on my machine. This is why the development server takes the next free port. In my case, this is 8081. The tool is going to increment the port number until it reaches a free one, so it is essential to pay attention to the console's output when running the application. For now, let's use the default port, that is, 8080: `vue-cli-service serve`

```
vue-cli-service servefunction createWindow() {
  win = new BrowserWindow({ width: 800, height: 600 });

  // win.loadURL(`file://${__dirname}/dist/index.html`);
  win.loadURL('http://localhost:8080');

  win.on('closed', () => {
    win = null;
  });
}
```

For the next step, you are going to need two Terminal windows. In the first one, start the local development server, as follows:

```
npm run serve
```

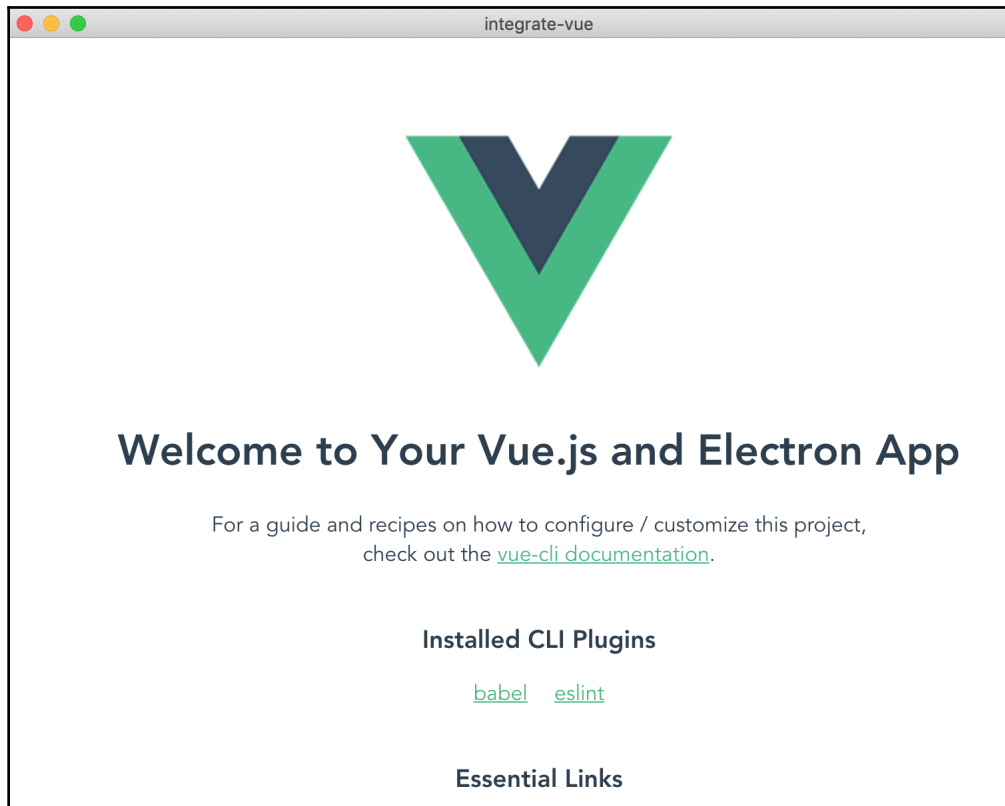
While the server is running, launch the Electron application with the following command:

```
npm start
```

You should get the same window that we got with the Vue UI. This time, however, the web server is watching for changes under the hood and instructing the web client to reload. This also applies to our Electron window. To see the live reload feature in action, switch to the `src/App.js` file and update the label:

```
<template>
  <div id="app">
    
    <HelloWorld msg="Welcome to Your Vue.js and Electron App"/>
  </div>
</template>
```

As soon as you save the file, the application reflects these changes on the screen:



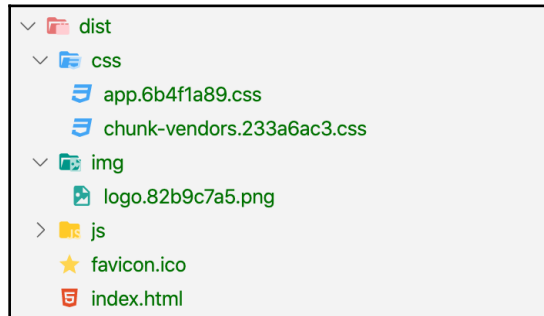
You now have a basic project template for an Electron application that is powered by Vue.js. Now, it's time to prepare our application for production compilation.

Production builds

Many CLI tools for web frameworks have the concept of production and development builds. The Vue CLI provides you with a script that produces highly optimized and minimized artifacts for distribution. Just run the `build` command, as shown in the following code:

```
npm run build
```

You can find the final bits you'll need in the `dist` folder. The `dist` folder contains everything you need in order to distribute or serve your web application:



Note that you don't need Vue.js or Node.js to run the application on a web server.

Let's move on and learn how to set up conditional loading so that our desktop application uses a web server for development but references the production output when distributing.

Setting up conditional loading

We are going to use the `DEBUG` environment variable as an indicator that the application needs to connect to a local development server. Let's learn how to enable this in application scripts:

1. Update the `package.json` file and set the following variable as the default option:

```
{
  "scripts": {
    "serve": "vue-cli-service serve",
    "start": "DEBUG=true electron .",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
}
```

For Windows users, the start script should look as follows:

```
{
  "start": "SET DEBUG=true && electron ."
}
```

2. Once you've defined an environment variable, the format of the check inside the `main.js` file should be similar to the following:

```
if (process.env.DEBUG) {
  // load from running server on port 3000
} else {
  // load production build from the "dist" folder
}
```

3. Update the `main.js` file and make the following changes to the `createWindow` function:

```
function createWindow() {
  win = new BrowserWindow({ width: 800, height: 600 });

  if (process.env.DEBUG) {
    win.loadURL(`http://localhost:8080`);
  } else {
    win.loadURL(`file://${__dirname}/build/index.html`);
  }

  win.on('closed', () => {
    win = null;
  });
}
```

Note that, depending on what applications or development servers are running on your machine, the port may differ. Update the code according to the output you receive when you run the `npm run serve` command.

Now, you can run against the development server with live reloads when working on application code and have static production build assets when preparing the application for distribution.

Adding routing

The routing feature is an essential part of any modern web application. Routing allows us to switch between different screens and have dedicated URL addresses for particular features.

The Vue CLI allows us to utilize the `vue add` command so that we can add additional plugins. Let's use it to install the `router` library:

1. Run the following command in the project root:

```
vue add router
```

2. When asked about **history mode**, type `Y` as your answer:

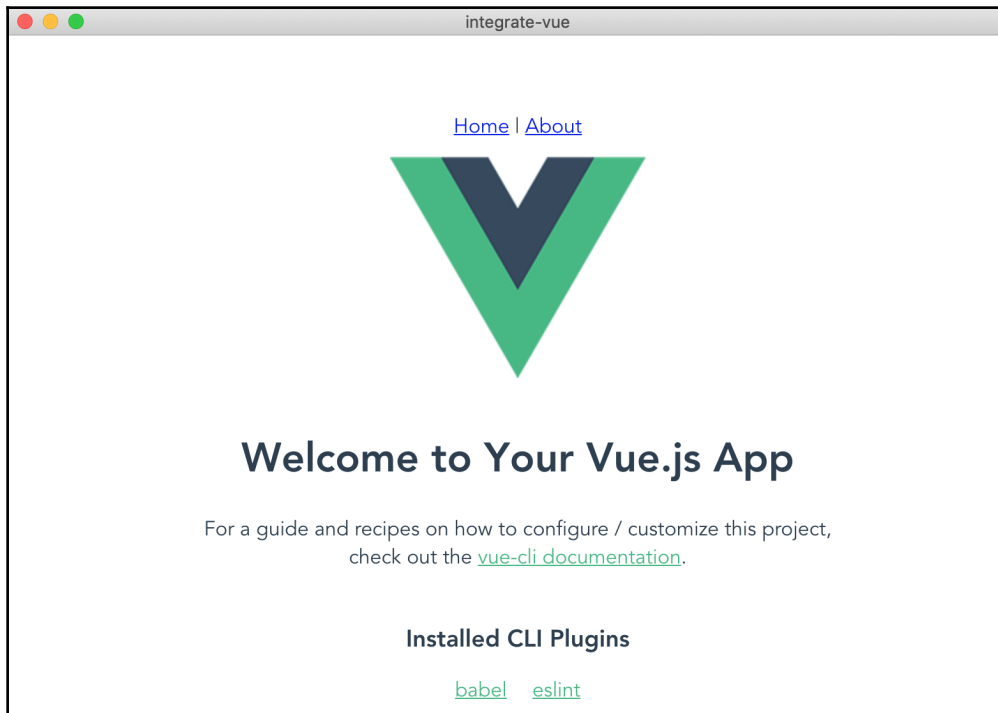
```
? Use history mode for router? (Requires proper server setup for
index fallback in production) (Y/n)
A: Y
```

3. Now, you should have multiple modified files, as shown in the following code:

```
✓ Successfully invoked generator for plugin: core:router
The following files have been updated / added:

src/router.js
src/views/About.vue
src/views/Home.vue
package-lock.json
package.json
src/App.vue
src/main.js
```

4. Start the application and pay attention to the bottom of the screen. It now contains two new links: `Home` and `About`:



5. Now, click the **About** link to see what happens. The Electron application should switch to the `About` component:



Now, you are able to build multiple views for your application. Let's learn how we can use UI toolkits to save time.

Configuring Vue Material

There are many libraries out there for Vue.js. In this section, we are going to use the *Vue Material* toolkit. You can find out more at <https://vuematerial.io>. Let's get started:

1. Run the following NPM command to install the `vue-material` library in the project:

```
npm install vue-material
```

2. As part of Vue Material's integration, you may want to set up the Roboto font as well. Update the `public/index.html` file with the following code:

```
<link
  rel="stylesheet"
  href="//fonts.googleapis.com/css?family=Roboto:400,500,700,400italic|Material+Icons"
/>
```

3. The easiest way to enable all Vue Material components is to import all the scope that's inside the `src/main.js` file. Later, you can optimize the imports and declare the components you are using:

```
import Vue from 'vue';
import App from './App.vue';
import router from './router';

Vue.config.productionTip = false;

import VueMaterial from 'vue-material';
import 'vue-material/dist/vue-material.min.css';
import 'vue-material/dist/theme/default.css';

Vue.use(VueMaterial);

new Vue({
  router,
  render: h => h(App)
}).$mount('#app');
```

4. You should already have the essential configuration for routes since it was generated as part of the router plugin's installation. To integrate the routing component with Vue Material, please update the code inside the `src/router.js` file according to the next listing:

```
Vue.use(Router);

Vue.component('router-link', Vue.options.components.RouterLink);
Vue.component('router-view', Vue.options.components.RouterView);
```

Now that you have routing support in your Electron application, it's time to integrate the application toolbar.

Creating an application toolbar

As a simple exercise, let's create an application toolbar that can redirect us to different routes:

1. In the `src/App.vue` file, declare the `md-app-toolbar` component, as follows:

```
<md-app>
  <md-app-toolbar class="md-primary">
    <span class="md-title">My Title</span>
  </md-app-toolbar>
</md-app>
```

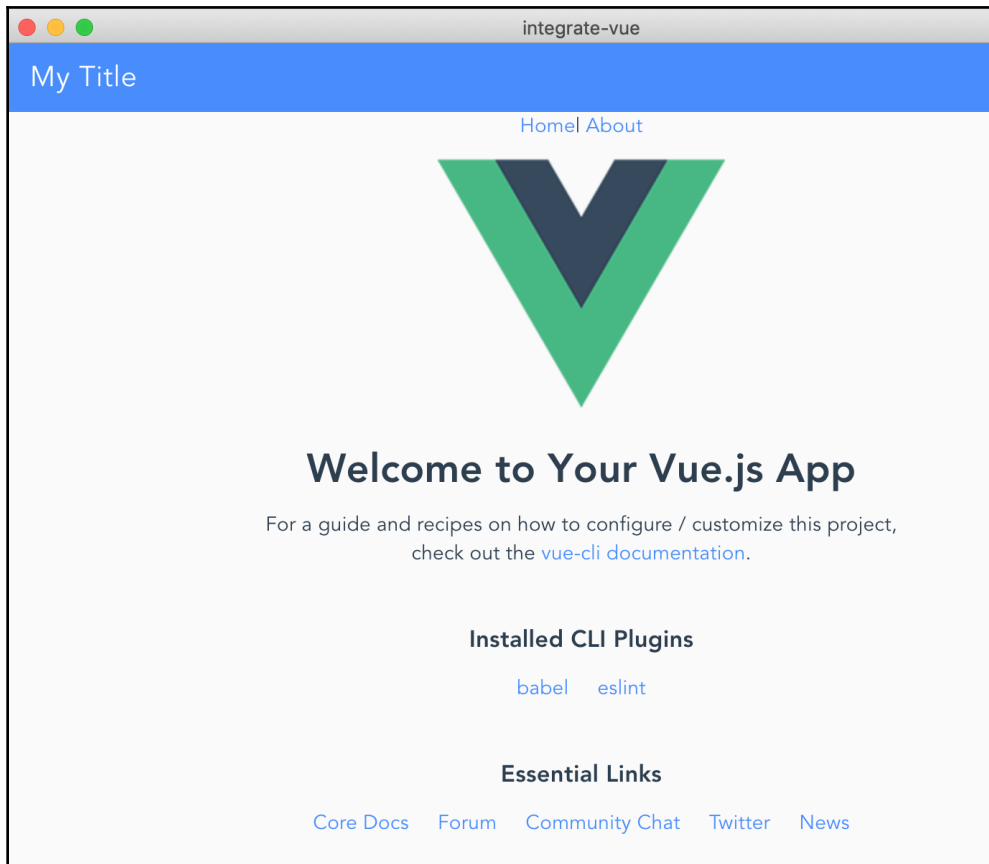
2. Also, remove the margin from the `#app` style. Now, the file should look as follows:

```
<template>
  <div id="app">
    <md-app>
      <md-app-toolbar class="md-primary">
        <span class="md-title">My Title</span>
      </md-app-toolbar>
    </md-app>
    <div id="nav">
      <router-link to="/">Home</router-link>|
      <router-link to="/about">About</router-link>
    </div>
    <router-view/>
  </div>
</template>

<style>
#app {
```

```
font-family: "Avenir", Helvetica, Arial, sans-serif;  
-webkit-font-smoothing: antialiased;  
-moz-osx-font-smoothing: grayscale;  
text-align: center;  
color: #2c3e50;  
}  
</style>
```

3. Run the application if you haven't done so already. The main screen should now contain a nice blue application toolbar, as shown in the following screenshot:



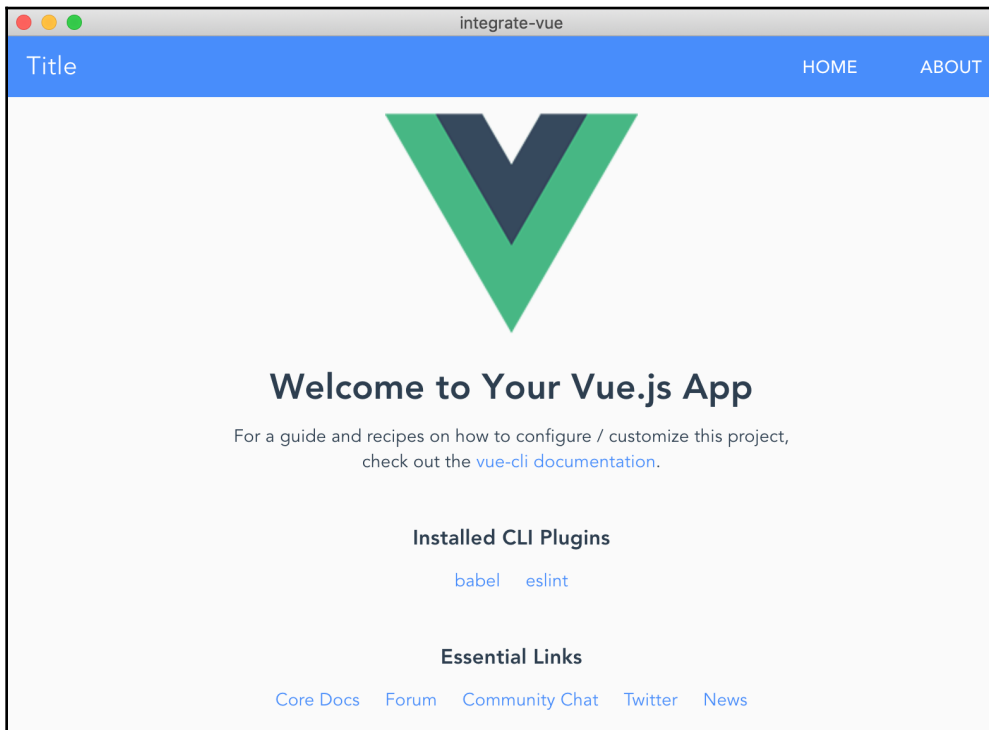
4. Now, let's insert two buttons and redirect them to the Home and About routes when they're clicked:

```
<template>  
  <div id="app">
```

```
<md-app>
  <md-app-toolbar class="md-primary">
    <h3 class="md-title" style="flex: 1; text-align:
      left;">Title</h3>
    <md-button to="/">Home</md-button>
    <md-button to="/about" class="md-primary">About</md-button>
  </md-app-toolbar>
</md-app>
<router-view/>
</div>
</template>
```

Note that we also have a `spacer` element so that we can move the buttons to the right-hand side of the screen while leaving the `Title` element on the left. This pattern is widely used when developers build application toolbars.

5. Switch back to your Electron application and check out the buttons. They should switch your screens according to the rules we have just added:



At this point, you are ready to build complex applications that involve multiple screens and routing.

Summary

In this chapter, you successfully configured three Electron projects based on popular web frameworks: Angular, React, and Vue.js. Now, you can work efficiently with the *live reloading* feature, which provides instant feedback on changes you make.

You also have a better understanding of how to integrate different UI toolkit libraries so that you can save time when you're developing primitives and focus on your application's business logic instead.

Keep the routing feature in mind when you're building applications. This feature allows you to switch Views in your Electron application quickly and follow the separation-of-concerns design principle.

In the next chapter, we are going to build a screenshot snipping tool so that you can see various Electron development tasks at work in the real world.

4

Building a Screenshot Snipping Tool

In this chapter, we are going to build a small screenshot snipping tool using the latest Electron framework, React.js, and the Blueprint UI toolkit for React.

As part of the practical exercise in this chapter, you are going to generate screenshots using the Electron API and manage the application's window state and visibility. This will help you understand how to control application windows in Electron. You will also be working with `desktopCapture` features and generating thumbnails from code. Finally, you will learn how to integrate with the System Tray and invoke its functionality with the global keyboard shortcut.

In this chapter, we will learn about the following topics:

- Preparing the project
- Configuring frameless windows
- Transparent windows
- Making application windows draggable
- Adding a *Snip* toolbar button
- Using the `desktopCapturer` API
- Calculating the primary display size
- Generating and saving a thumbnail image
- Resizing and cropping the image
- Testing the application's behavior
- Integrating with the system tray
- Hiding the main application window on startup
- Registering global keyboard shortcuts

Technical requirements

To get started with this chapter, you will need a standard laptop or desktop running macOS, Windows, or Linux.

The minimal set of software you need to have installed for this chapter is as follows:

- Git, a version control system
- Node.js with NPM
- Visual Studio Code, a free and open source code editor

You can find the code files for this chapter in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter04>.

Preparing the project

We aren't going to repeat the project setup procedure. By now, you should already know how to bootstrap an Electron application using multiple frameworks, or even plain JavaScript. If you want a recap, please refer to *Chapter 3, Integrating with Angular, React, and Vue*.

Let's use the React library to create a React app utility so that we can build a screenshot snipping tool.



Note that if you are using Electron 5.0.0 or later, you need to enable Node.js integration explicitly. In later versions of Electron, Node.js integration is disabled due to security reasons.

As you already know, Electron applications can also display remote websites. This gives a remote web page access to your local resources and potentially allows them to perform malicious activities. That is why Node.js integration is disabled. For fully offline applications, we need to enable Node.js support explicitly.

We are going to be using local resources and assets, so you should use the `webPreferences` object to allow `window.require` and other Node.js features to be used in the web renderer process. You can enable all the necessary features by using the following code:

```
webPreferences: {  
  nodeIntegration: true  
}
```

Please refer to the following code and update the `createWindow` function in your `main.js` file accordingly:

```
function createWindow() {  
  win = new BrowserWindow({  
    transparent: true,  
    frame: false,  
    webPreferences: {  
      nodeIntegration: true  
    }  
  });  
  
  win.loadURL(`http://localhost:3000`);  
  
  win.on('closed', () => {  
    win = null;  
  });  
}
```

There are many other options you can toggle or tune. You can find the full list at <https://electronjs.org/docs/api/browser-window>.

Now that we have prepared our project, let's move on to the first step of creating our application. Let's see what it takes to create a frameless window with Electron.

Configuring frameless windows

For the screenshot snipping tool, we need a minimal window, known as a Chrome, so that we can select a portion of the screen to make a screenshot. Due to this, we need to use an Electron feature called **frameless windows**, which allows you to *open a window without toolbars, borders, or other graphical **chromes***.



Refer to the following resource to find out more: <https://electronjs.org/docs/api/frameless-window>.

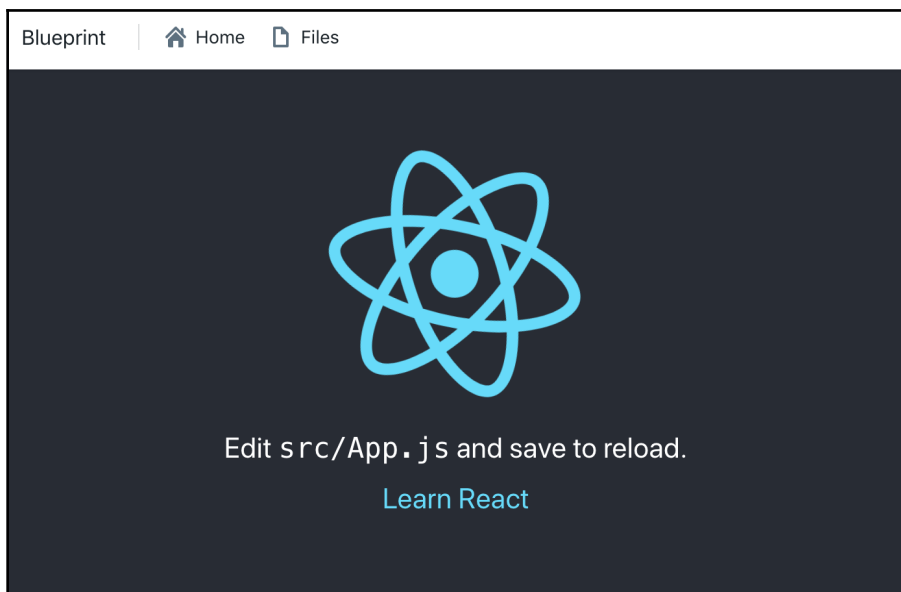
We are only going to touch on the basics that you will need to implement for the application. For now, let's learn how to create a basic frameless window:

1. Update the code of the `main.js` file so that it looks as follows:

```
win = new BrowserWindow({  
  width: 800,
```

```
    height: 600,  
    webPreferences: {  
      nodeIntegration: true  
    }  
    frame: false  
  });
```

2. If you run the application now, you will see that it is missing the traditional menu bar and traffic light buttons (minimize, maximize, and close):

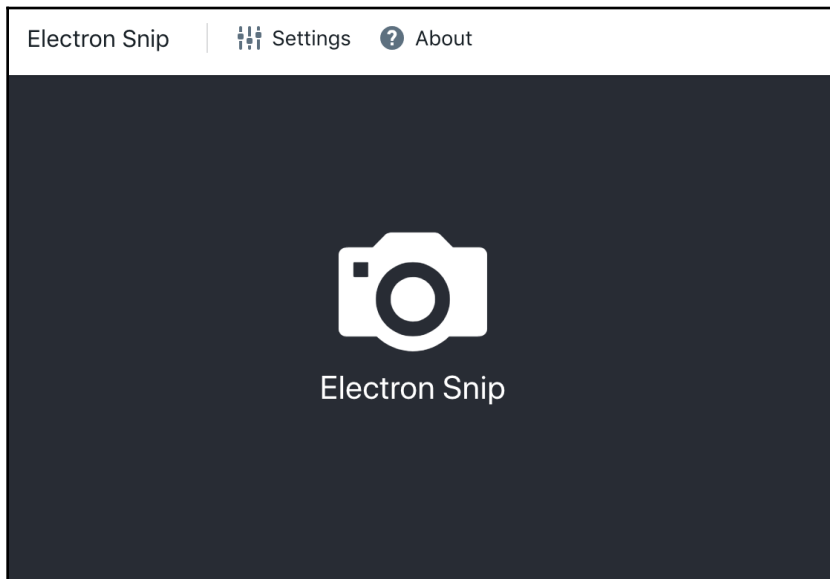


3. Feel free to change the toolbar buttons to something more meaningful for the type of application you're creating. For example, we can do this for the Settings and About routes. Use the following code to do so:

```
import React from 'react';  
import './App.css';  
import { Navbar, Button, Alignment, Icon } from '@blueprintjs/core';  
function App() {  
  return (  
    <div className="App">  
      <Navbar>  
        <Navbar.Group align={Alignment.LEFT}>  
          <Navbar.Heading>Electron Snip</Navbar.Heading>  
          <Navbar.Divider />  
          <Button className="bp3-minimal" icon="settings">
```

```
      text="Settings" />
    <Button className="bp3-minimal" icon="help" text="About" />
  </Navbar.Group>
</Navbar>
<main className="App-main">
  <Icon icon="camera" iconSize={100} />
  <p>Electron Snip</p>
</main>
</div>
);
}
export default App;
```

4. You are probably running the live reloading configuration right now since we discussed it in the previous chapters. Switch to the running application instance and ensure that the toolbar now displays the expected buttons:



As you can see, we can't drag the application window across the screen at the moment. Don't worry, though, as we are going to address this shortly. First, let's take a look at the extra options we have when we're working with macOS.

Additional options for macOS

When running on macOS, you can use the `titleBarStyle` property. Here's what the official documentation says about this feature:

*Instead of setting **frame** to **false**, which disables both the title bar and window controls, you may want to have the title bar hidden and your content extend to the full window size, yet still preserve the window controls (**traffic lights**) for standard window actions. You can find out more here: <https://electronjs.org/docs/api/frameless-window#alternatives-on-macos>.*

Let's take a look at how each of these features works in practice.

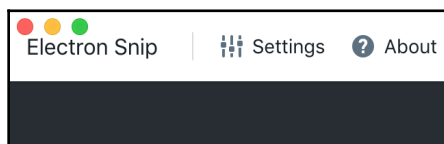
Using the hidden titleBarStyle

When you set the `titleBarStyle` property to `hidden`, you instruct Electron to hide the title bar but leave the traffic light controls in the top-left corner. This allows us to continue controlling the look and feel of the application window but preserve the behavior behind the control buttons.

Update the `createWindow` function in the `main.js` file as follows:

```
function createWindow() {  
  win = new BrowserWindow({ titleBarStyle: 'hidden' });  
  
  win.loadURL(`http://localhost:3000`);  
  
  win.on('closed', () => {  
    win = null;  
  });  
}
```

At this point, you need to restart the application as live reloading won't help. Once the Electron shell has been restarted, take a closer look at the control button area and notice the missing title bar:



Imagine that instead of the navigation bar, you have a beautiful border picture or CSS style.

Using the `hiddenInset` `titleBarStyle`

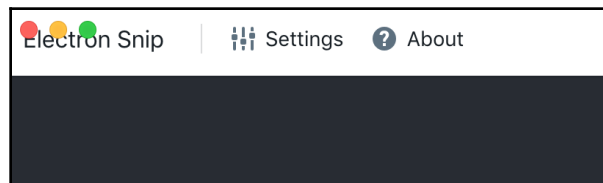
The second option you have is the `hiddenInset` value for the `titleBarStyle` property. The only difference to the `hidden` style is that, with `hiddenInset`, the buttons have an inset-like style, but the overall behavior is still the same:

```
function createWindow() {
  win = new BrowserWindow({ titleBarStyle: 'hiddenInset' });

  win.loadURL(`http://localhost:3000`);

  win.on('closed', () => {
    win = null;
  });
}
```

Restart your Electron application. When you check the positioning and style of the window control buttons, you will see something like this:



As you can see, this time, the buttons are above the application title in the toolbar.

Using the `customButtonsOnHover` `titleBarStyle`

Last but not least, we have the `customButtonsOnHover` value, which you can exclusively use with the `titleBarStyle` property for macOS and only when running in frameless mode:

```
function createWindow() {
  win = new BrowserWindow({
    titleBarStyle: 'customButtonsOnHover',
    frame: false
  });

  win.loadURL(`http://localhost:3000`);
}
```

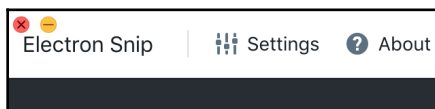
```
win.on('closed', () => {  
  win = null;  
});  
}
```

This option is very nice and convenient when you want your custom window to look completely non-standard. The window control buttons, also known as traffic light buttons, stay hidden by default, but your users can still get them by moving their mouse cursor over the top left corner. Let's see how that works:

1. Restart or launch your application one more time and pay attention to where the buttons should be:



2. The buttons are invisible by default. Now, move the mouse cursor over that region and see what happens:



To find out more about macOS support for the title bar styles, please refer to the following resource: <https://electronjs.org/docs/api/frameless-window#alternatives-on-macos>.

The next important thing we need to look at is transparent windows.

Transparent windows

Given that we are working on the screenshot snipping tool, it is vital to be able to select an area of the screen to create a screenshot. Traditionally, such tools offer a transparent and resizable window so that users can imagine the result more clearly.

You can enable window transparency by using the `transparent` property like so:

```
function createWindow() {  
  win = new BrowserWindow({  
    transparent: true,  
    frame: false  
  });  
  
  win.loadURL(`http://localhost:3000`);  
  
  win.on('closed', () => {  
    win = null;  
  });  
}
```

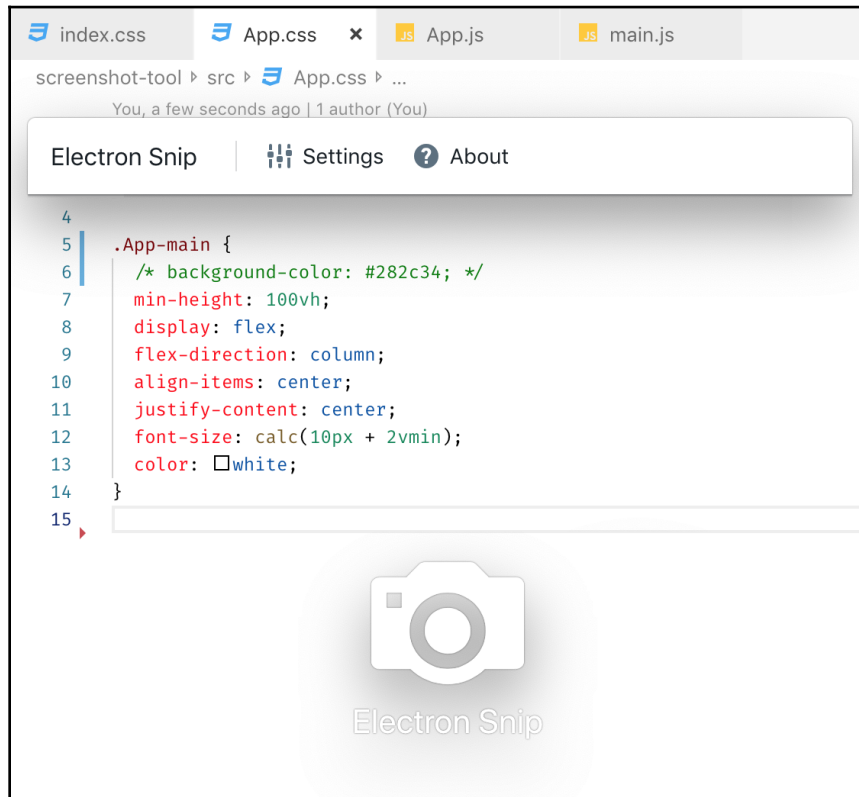


Please note that transparency mode has its own platform-specific limitations. You can find all the details here: <https://electronjs.org/docs/api/frameless-window#limitations>.

However, if you try to run your application right now, it won't be transparent. This is because of the default background color that Create React App generates for the initial application scaffolds. You can easily change that by updating the `App.css` file and commenting out the `background-color` style, like so:

```
.App {  
  text-align: center;  
}  
  
.App-main {  
  /* background-color: #282c34; */  
  min-height: 100vh;  
  display: flex;  
  flex-direction: column;  
  align-items: center;  
  justify-content: center;  
  font-size: calc(10px + 2vmin);  
  color: white;  
}
```


This time, as you can see, the application body is genuinely transparent; you can see the content of Visual Studio Code or any other application in the background:



The only non-transparent element that remains is the toolbar. This happens because of our custom background color. This is perfectly fine for our scenario as we intend to use that area to drag the application around.

Let's polish the application's look a bit by setting a distinctive application border style and centering the application icon:

1. First, update the `App.css` file according to the following code:

```
.App {  
  text-align: center;  
  height: 100vh;  
}  
.App-main {  
  height: 100%;
```

```
display: flex;
flex-direction: column;
align-items: center;
justify-content: center;
font-size: calc(10px + 2vmin);
}
```

The preceding code does content centering for us.

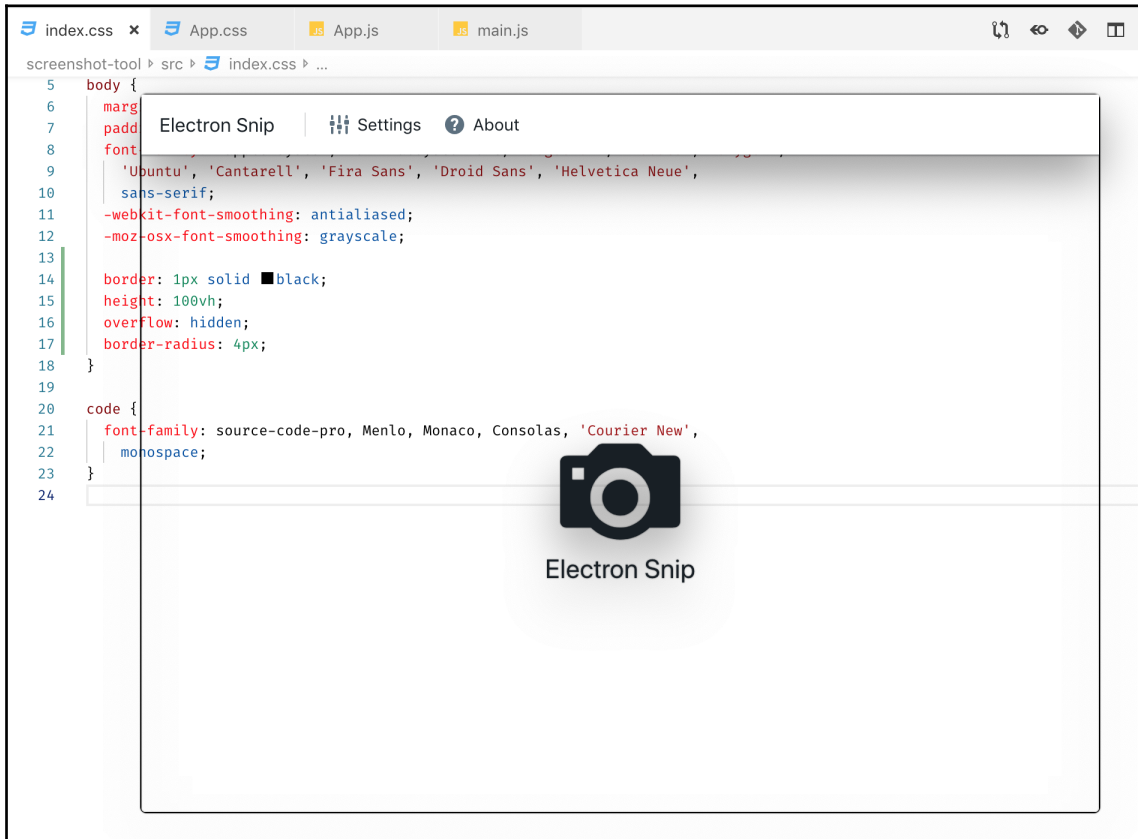
2. Next, switch to the `index.css` file to add a border around the application's body element:

```
@import '~normalize.css';
@import '~@blueprintjs/core/lib/css/blueprint.css';
@import '~@blueprintjs/icons/lib/css/blueprint-icons.css';
body {
  margin: 0;
  padding: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI',
    'Roboto', 'Oxygen', 'Ubuntu', 'Cantarell',
    'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;

  border: 1px solid black;
  height: 100vh;
  overflow: hidden;
  border-radius: 4px;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas,
    'Courier New',
    monospace;
}
```

3. Now, the users of your application can see the boundaries of the application and imagine the resulting screenshot area:



As you may recall, the main application window of our project is still static. We don't have an application title to drag it around as we have made it frameless. Instead, let's make the whole window draggable.

Making application windows draggable

You may have already noticed that you cannot drag the application window as soon as it becomes frameless. That is the default behavior of Electron windows, but you can easily change it by applying the `-webkit-app-region: drag` style to the `body` element of the HTML document:

```
<body style="-webkit-app-region: drag">
</body>
```

Note, however, that once you apply the `-webkit-app-region: drag` style, the whole application area becomes draggable, including all the buttons and input elements on the page. You can whitelist certain areas or HTML elements by utilizing the `-webkit-app-region: no-drag` CSS value. In this case, the marked element is excluded from the drag feature.

Let's whitelist all the buttons from the dragging area so that the users of our application can click them:

1. Update the `App.css` file and add the `button` rule:

```
.App {
  text-align: center;
  height: 100vh;
}

.App-main {
  height: 100%;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
}

button {
  -webkit-app-region: no-drag;
}
```

2. Run the application, click somewhere on the screen, and try to drag the window.

Notice that you can drag the window by pressing any element except buttons. This is what we expect from our application's behavior.

Now, we need to create a button so that we can invoke the image snipping functionality. This is what we are going to address in the next section.

Adding a snip toolbar button

Now that we have a draggable window and non-draggable menu buttons, let's add a button so that we can take a screenshot:

1. Update the `App.js` code and append a new button component with the icon value of `camera` and with the text `Snip`, as shown in the following example:

```
<Navbar>
  <Navbar.Group align={Alignment.LEFT}>
    <Navbar.Heading>Electron Snip</Navbar.Heading>
    <Navbar.Divider />
    <Button className="bp3-minimal" icon="settings"
      text="Settings" />
    <Button className="bp3-minimal" icon="help" text="About" />
    <Button className="bp3-minimal" icon="camera"
      text="Snip"/>
  </Navbar.Group>
</Navbar>
```

2. We also need to add an `onSnipClick` function stub. This is going to handle our `Snip` button clicks.
3. For now, let's create a simple `console.log` call to ensure that the handler works at runtime. We are going to get back to the real implementation shortly:

```
const onSnipClick = () => {
  console.log('todo: making screenshot');
};
```

4. The complete implementation of the `App.js` file should now look as follows:

```
import React from 'react';
import './App.css';
import { Navbar, Button, Alignment, Icon } from '@blueprintjs/core';
function App() {
  const onSnipClick = () => {
    console.log('todo: making screenshot');
  };
  return (
    <div className="App">
      <Navbar>
        <Navbar.Group align={Alignment.LEFT}>
          <Navbar.Heading>Electron Snip</Navbar.Heading>
          <Navbar.Divider />
          <Button className="bp3-minimal" icon="settings"
            text="Settings" />

```

```
        <Button className="bp3-minimal" icon="help" text="About" />
        <Button
          className="bp3-minimal"
          icon="camera"
          text="Snip"
          onClick={onSnipClick}
        />
      </Navbar.Group>
    </Navbar>

    <main className="App-main">
      <Icon icon="camera" iconSize={100} />
      <p>Electron Snip</p>
    </main>
  </div>
);
}

export default App;
```

Now, we are going to focus on the `onSnipClick` function's implementation.

Using the desktopCapturer API

First of all, you need to be familiar with the `desktopCapturer` API that Electron provides. According to the official documentation, this API allows you to do the following:

Access information about media sources that can be used to capture audio and video from the desktop using the `navigator.mediaDevices.getUserMedia` (<https://developer.mozilla.org/en/docs/Web/API/MediaDevices/getUserMedia>) API.

Now, we are going to walk through the basics and introduce the `Snip` button click event, which has access to the capturing sources:

1. Update the `onSnipClick` function implementation according to the following code:

```
const onSnipClick = async () => {
  const { desktopCapturer, remote } = window.require('electron');
  const screen = remote.screen;
  try {
    const sources = await desktopCapturer.getSources({ types:
      ['screen'] });
    const entireScreenSource = sources.find(
      source => source.name === 'Entire Screen'
```

```
    );  
    if (entireScreenSource) {  
      console.log(entireScreenSource);  
    }  
  } catch (err) {  
    console.error(err);  
  }  
};
```

Please note that your users may be using more than one monitor when they use your application. In such cases, Electron may find multiple different sources. Instead of `Entire Screen`, you may have `Screen 1`, `Screen 2`, and so on.

For the sake of simplicity, let's make use of the first screen.

2. Update the code so that we can take multiple screens into account and check for `Screen 1`:

```
const entireScreenSource = sources.find(  
  source => source.name === 'Entire Screen'  
  || source.name === 'Screen 1'  
);
```

As you can see, if Electron fails to find the `Entire screen` source, it is going to check for `Screen 1` as well.

In a real-world application, you may want to provide some sort of dialog or settings page to allow users to configure these sources. For example, we could present the user with a list of available sources and allow them to make some of them *defaults*.



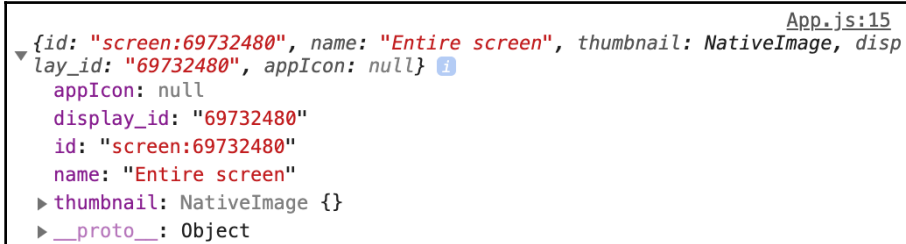
You can always find more details and examples in the following documentation article: <https://electronjs.org/docs/api/desktop-capturer>.

Your users may have different types of monitors with various resolutions and aspect ratios. A good example is a MacBook with a Retina display. This is why we need to calculate the primary display size for the resulting thumbnail. Let's see how we can do this.

Calculating the primary display size

Here, we are requesting a screen source for capturing. Since more than one result is possible, we will select the capturing source named `Entire Screen`. This is one of the ways we can get access to the current screen content so that we can record a video, audio, or an image thumbnail.

In our initial implementation, we retrieved the `Entire Screen` source and logged it to the browser console to show you its structure:

A screenshot of a browser's developer console. The log shows a JavaScript object representing a screen source. The object has properties: id, name, thumbnail, display_id, appIcon, and __proto__. The values are: id: "screen:69732480", name: "Entire screen", thumbnail: NativeImage, display_id: "69732480", appIcon: null. The log is truncated with an ellipsis. The file name and line number "App.js:15" are visible in the top right corner of the console window.

```
App.js:15
{id: "screen:69732480", name: "Entire screen", thumbnail: NativeImage, display_id: "69732480", appIcon: null}
  appIcon: null
  display_id: "69732480"
  id: "screen:69732480"
  name: "Entire screen"
  ▶ thumbnail: NativeImage {}
  ▶ __proto__: Object
```

One of the convenient features that the screen capturing source object provides is Thumbnail generation. You can generate a preview thumbnail from the source with custom parameters. The `Source` object is going to generate a `NativeImage` instance that we can manipulate further. This is what we are going to reuse for our tool.

Also, the Electron framework allows you to access the screen's details. When working with multi-display operating systems, you probably want to get the primary display. In any case, we are going to import the `screen` and calculate the maximum possible square size of the potential thumbnail image, as follows:

```
const screenSize = screen.getPrimaryDisplay().workAreaSize;

const maxDimension = Math.max(
  screenSize.width,
  screenSize.height
);

const sources = await desktopCapturer.getSources({
  types: ['screen'],
  thumbnailSize: {
    width: maxDimension * window.devicePixelRatio,
    height: maxDimension * window.devicePixelRatio
  }
});
```


As you may have noticed, we also took the `window.devicePixelRatio` property value into account, which supports HiDPI or Retina displays.



You can find out more about the difference between rendering on a standard display versus a HiDPI or Retina display at <https://developer.mozilla.org/en-US/docs/Web/API/Window/devicePixelRatio>.

Now it's time to generate our first screenshot and save it as a thumbnail image.

Generating and saving a thumbnail image

When we requested the capturing source, we provided a thumbnail size that's equal to the size of the main screen. Now you can use the `NativeImage` methods to perform any necessary image conversion and manipulation.

Many useful APIs are exposed by the `NativeImage` class. You can find out more at <https://electronjs.org/docs/api/native-image#class-nativeimage>.

For now, you need the following functions to finish the application:

- `toPNG`: This converts image data into png format.
- `resize`: This manipulates the size of the resulting image.
- `crop`: This cuts out a portion of the image.

You can convert the screen thumbnail into a png image using the following code:

```
const image = entireScreenSource.thumbnail.toPNG();
```

Follow these steps to get started:

1. First, let's import the `os`, `path`, and `fs` classes from Node.js so that we can generate temporary file names. Now, we need to use the `writeFile` function to store the file locally. We also need `shell` from Electron so that we can invoke shell commands:

```
const { desktopCapturer, remote, shell } =  
  window.require('electron');  
const screen = remote.screen;  
const path = window.require('path');  
const os = window.require('os');  
const fs = window.require('fs');
```

2. Next, update your code so that it looks as follows. Here, we are generating a temporary file, saving the PNG image, and using `shell.openExternal` to launch the file outside the Electron shell:

```
try {
  const screenSize = screen.getPrimaryDisplay().workAreaSize;
  const maxDimension = Math.max(screenSize.width,
    screenSize.height);
  const sources = await desktopCapturer.getSources({
    types: ['screen'],
    thumbnailSize: {
      width: maxDimension * window.devicePixelRatio,
      height: maxDimension * window.devicePixelRatio
    }
  });

  const entireScreenSource = sources.find(
    source => source.name === 'Entire Screen' || source.name ===
      'Screen 1'
  );
  if (entireScreenSource) {
    const outputPath = path.join(os.tmpdir(), 'screenshot.png');
    const image = entireScreenSource.thumbnail.toPNG();
    fs.writeFile(outputPath, image, err => {
      if (err) return console.error(err);
      shell.openExternal(`file://${outputPath}`);
    });
  }
} catch (err) {
  console.error(err);
}
```

3. Your operating system will automatically fetch the default application for previewing files. For macOS, for example, you should expect a standard preview to appear.
4. Update the code so that it can handle the missing screen source:

```
if (entireScreenSource) {
  // ...
} else {
  window.alert('Screen source not found.');
```

Now, let's look at how we can manipulate the resulting image. It's time to learn how to resize and crop the image.

Resizing and cropping the image

Now that we have a thumbnail image, we need to perform two additional steps before saving it to local storage:

1. The first thing we need to do is resize the image so that it fits the screen size. As you may recall, we make a square image based on the maximum dimension, which is based on either the screen's height or the screen's width. It is easy to resize the `NativeImage` instance because you already have a dedicated `resize` method for this very purpose.
2. The second step is to crop the image. When we take a screenshot of the whole screen area, our users may only want a portion of the screen or a frameless and transparent window. Therefore, we need to crop it and leave only the rectangle based on the window boundaries.

The `NativeImage` class allows us to perform method chaining. This allows us to call multiple methods before we convert the final result into PNG format:

```
const image = entireScreenSource.thumbnail
  .resize({
    width: screenSize.width,
    height: screenSize.height
  })
  .crop({
    x: window.screenLeft,
    y: window.screenTop,
    width: window.innerWidth,
    height: window.outerHeight
  })
  .toPNG();
```

We can improve our code slightly by getting access to the application window from the web renderer side. This means that we can access window boundaries, as well as manipulate the window's state. Let's look at an example:

```
const { remote } = window.require('electron');

const win = remote.getCurrentWindow();
const windowRect = win.getBounds();
```

As you can see, we are accessing the `remote` object and fetching the current application window. After that, it's easy to get the bounds rectangle and pass it to the `crop` method, as shown in the following code:

```
const image = entireScreenSource.thumbnail
  .resize({
    width: screenSize.width,
    height: screenSize.height
  })
  .crop(windowRect)
  .toPNG();
```

Finally, when taking a screenshot, we have to hide the window and then show it again. This ensures that the window isn't part of the resulting image.

Your web renderer already has access to the active application window, so you can simply call the `win.hide()` or `win.show()` methods to control its visibility.

Please refer to the full implementation details that are provided in the following code:

```
const onSnipClick = async () => {
  const { desktopCapturer, screen, shell, remote } = window.require(
    'electron'
  );
  const path = window.require('path');
  const os = window.require('os');
  const fs = window.require('fs');
  const win = remote.getCurrentWindow();
  const windowRect = win.getBounds();

  win.hide();

  try {
    const screenSize = screen.getPrimaryDisplay().workAreaSize;
    const maxDimension = Math.max(screenSize.width, screenSize.height);

    const sources = await desktopCapturer.getSources({
      types: ['screen'],
      thumbnailSize: {
        width: maxDimension * window.devicePixelRatio,
        height: maxDimension * window.devicePixelRatio
      }
    });

    const entireScreenSource = sources.find(
      source => source.name === 'Entire Screen' || source.name ===
        'Screen 1'
    );
```

```
);

if (entireScreenSource) {
  const outputPath = path.join(os.tmpdir(), 'screenshot.png');

  const image = entireScreenSource.thumbnail
    .resize({
      width: screenSize.width,
      height: screenSize.height
    })
    .crop(windowRect)
    .toPNG();

  fs.writeFile(outputPath, image, err => {
    win.show();

    if (err) return console.error(err);
    shell.openExternal(`file://${outputPath}`);
  });
} else {
  window.alert('Screen source not found.');
```

```
} catch (err) {
  console.error(err);
}
};
```

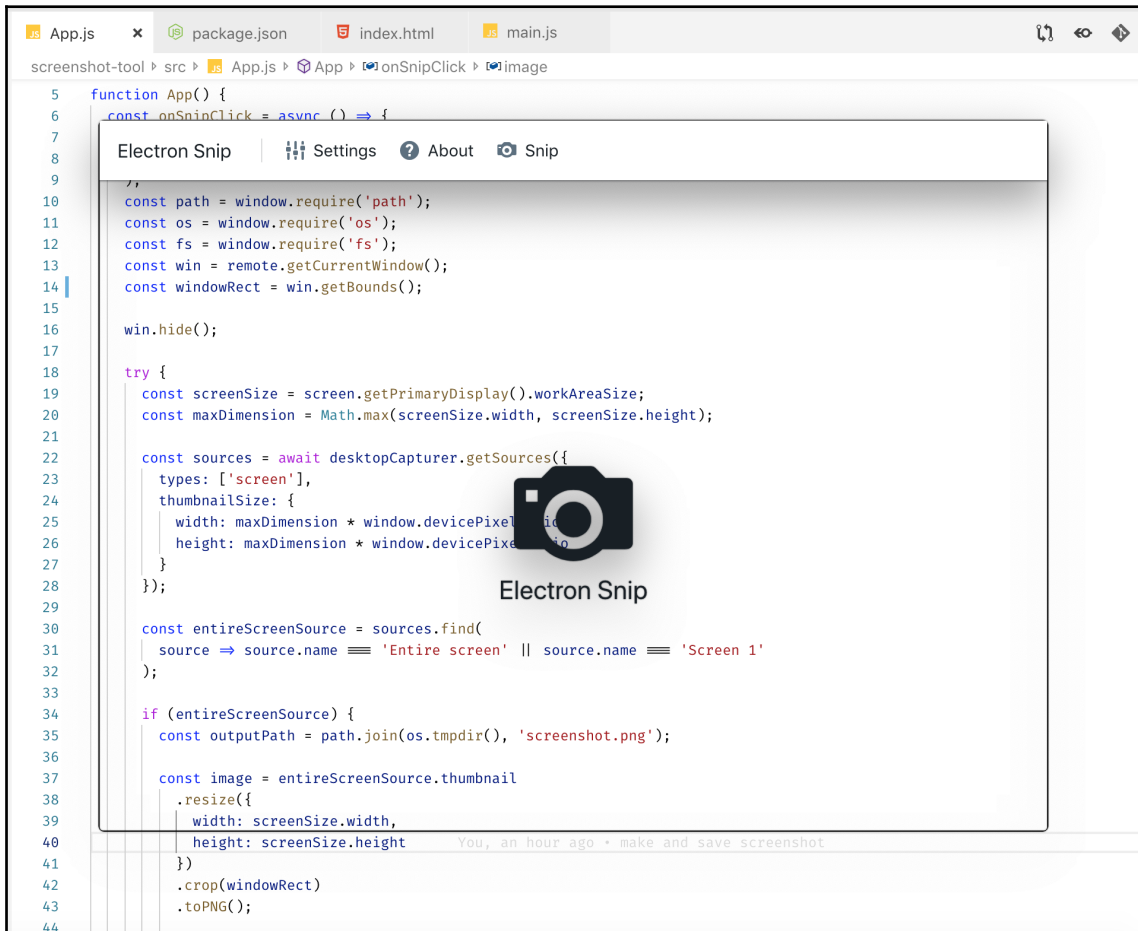
Note that we call `win.hide()` to hide the window before taking a screenshot and then enable visibility by calling `win.show()` when writing the resized and cropped result to the disk.

We have made very good progress. Now, it's time to test the application to see if everything we've built so far works as expected.

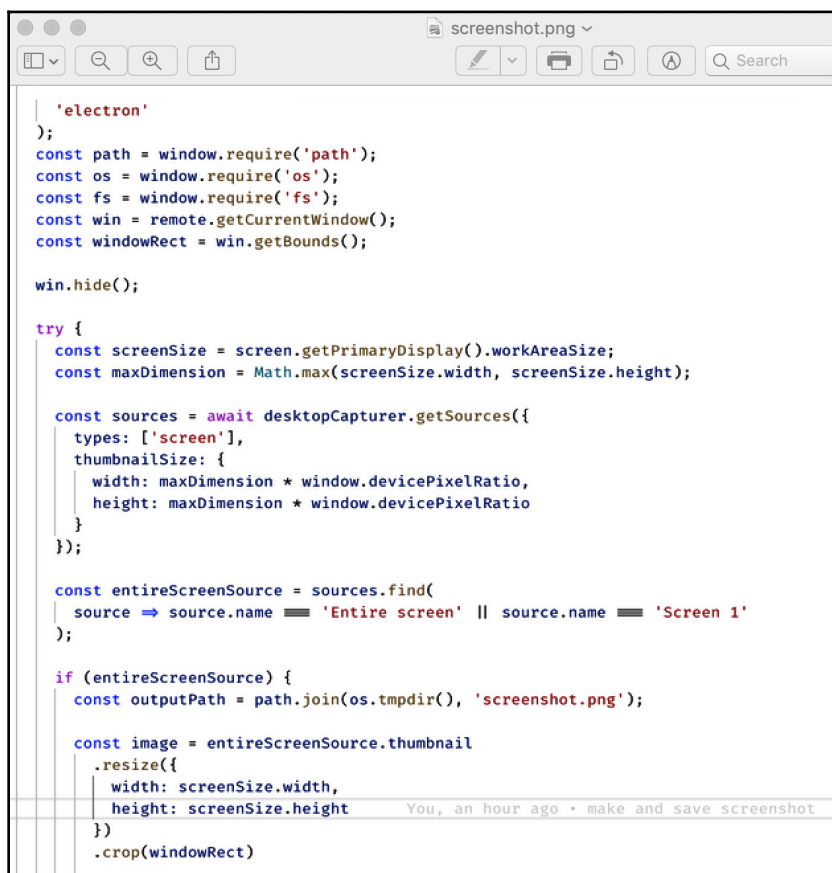
Testing the application's behavior

Now, let's test our screenshot tool:

1. Run the Electron application and drag the window around to select a portion of the screen. In this case, we are using the Visual Studio Code window as the source:



2. When you are ready, click the **Snip** button. Notice how the window disappears for a moment, and then you get the default system viewer showing an image:



3. Users can then use the default **Save As** functionality of the preview tool to save the file.

Congratulations on getting your first screenshot tool up and running! Now, let's improve it by adding support for the system tray and keyboard shortcuts.

Integrating with the system tray

In most cases, the users of your application may only use it when required and then minimize or close the app.

You can significantly improve user experience by keeping the application up and running in the background and displaying it in the system tray area. Another essential feature is to have a global keyboard shortcut so that the users of your application can quickly invoke it without needing to use the mouse.

Let's start by integrating the system tray:

1. First, you need to import the `Menu` and `Tray` objects from the `Electron` framework, the `Tray` integration, and also the `path` from `Node.js` to resolve the path to the `Tray` icon image:

```
const { Menu, Tray } = require('electron');
const path = require('path');
let tray;
```

2. Next, create a folder called `assets` and put a small 16 x 16 image in `png` format inside it. For the sake of simplicity, let's call it `icon.png`.
3. The following code shows how we can create a basic `Tray` entry with a custom image:

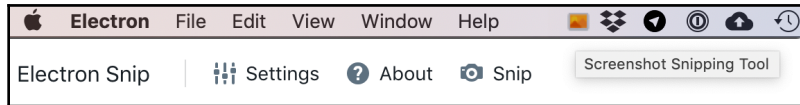
```
function createTray() {
  const iconPath = path.join(__dirname, 'assets/icon.png');
  tray = new Tray(iconPath);
  const contextMenu = Menu.buildFromTemplate([
    {
      label: 'Quit',
      type: 'normal',
      click() {
        app.quit();
      }
    }
  ]);

  tray.setToolTip('Screenshot Snipping Tool');
  tray.setContextMenu(contextMenu);
}
```

4. Here, we have created a function called `createTray` that builds and sets up the `Tray` component. Now, you need to call this function from within the `ready` handler:

```
app.on('ready', () => {
  createTray();
  createWindow();
});
```


5. Run or restart your Electron application and check out the system tray area. We are using macOS in this example. Notice the custom tooltip content if you hover the mouse cursor over the icon:



6. If you click on the icon, you should get the **Quit** menu entry, which is the one we have just defined in the `createMenu` function:



Now, we can make some minor enhancements to the application's user experience. Let's hide the application window on startup.

Hiding the main application window on startup

Let's try to hide the main application window on startup and show it only when it's invoked from the system tray menu:

1. The `BrowserWindow` class provides the `show` and `hide` methods, which we can use to control the visibility of the window instance. This means that we can call the `hide` method when building the window object at startup:

```
function createWindow() {  
  win = new BrowserWindow({  
    transparent: true,  
    frame: false,  
    webPreferences: {  
      nodeIntegration: true  
    }  
  })  
};  
  
win.hide();  
win.loadURL(`http://localhost:3000`);  
win.on('closed', () => {  
  win = null;  
});
```

```
    });  
  }
```

2. Finally, you can create an additional Show menu entry to invoke `win.show` and display the main application window to the user:

```
function createTray() {  
  const iconPath = path.join(__dirname, 'assets/icon.png');  
  tray = new Tray(iconPath);  
  const contextMenu = Menu.buildFromTemplate([  
    {  
      label: 'Show',  
      type: 'normal',  
      click() {  
        win.show();  
      }  
    },  
    {  
      label: 'Quit',  
      type: 'normal',  
      click() {  
        app.quit();  
      }  
    }  
  ]);  
  
  tray.setToolTip('Screenshot Snipping Tool');  
  tray.setContextMenu(contextMenu);  
}
```

Don't start the application just yet. Let's register some global keyboard shortcuts while we're here.

Registering global keyboard shortcuts

Now that the minimal system tray menu is up and running, let's provide keyboard support. You can use any key combination of your choice; for example, try *Cmd + Alt + Shift + S* for macOS or *Ctrl + Alt + Shift + S* for Linux and Windows. Let's get started:

1. First, import `globalShortcut` from the Electron framework, as shown in the following code:

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

2. As you already know, you can provide and render keyboard combinations by utilizing the `accelerator` property of the menu item:

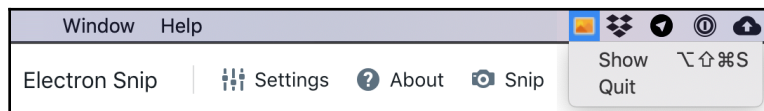
```
function createTray() {
  const iconPath = path.join(__dirname, 'assets/icon.png');
  tray = new Tray(iconPath);
  const contextMenu = Menu.buildFromTemplate([
    {
      label: 'Show',
      type: 'normal',
      accelerator: 'CommandOrControl+Alt+Shift+S',
      click() {
        win.show();
      }
    },
    {
      label: 'Quit',
      type: 'normal',
      click() {
        app.quit();
      }
    }
  ]);

  tray.setToolTip('Screenshot Snipping Tool');
  tray.setContextMenu(contextMenu);
}
```

3. However, the preceding code doesn't invoke the window when the application is minimized; it mainly serves as a hint that tells your users what combination they should use. We still need to register a global handler, as shown in the following code:

```
globalShortcut.register('CommandOrControl+Alt+Shift+S', () => {
  if (win) {
    win.show();
  }
});
```

4. As you can see, this was very easy to implement. Once the global shortcut handler has been invoked, we can call `win.show()` to display the main application window so that that end user can take a desktop screenshot:



5. Finally, disable the window after taking a screenshot, as shown in the following code:

```
fs.writeFile(outputPath, image, err => {  
  // win.show();  
  
  if (err) return console.error(err);  
  shell.openExternal(`file://${outputPath}`);  
});
```

Congratulations on reaching the end of this chapter! Feel free to extend and improve your desktop snipping tool as you see fit.

Summary

In this chapter, you have managed to create a lightweight screenshot snipping tool with the help of Rect and Electron and can apply this knowledge to other projects.

Now, you have an understanding of the desktop capturing API in Electron and you know how to detect multiple screens and how to work with pixel ratios, and you also know how to make screenshots of the entire desktop and control application transparency and visibility. We have also covered the Tray API and global keyboard shortcuts so that we can invoke our application. Feel free to extend the project with more features.

In the next chapter, we are going to build a simple 2D game to study graphics and gaming with Electron applications.

5

Making a 2D Game

In this chapter, we are going to build a simple 2D game that can run on the desktops of all major platforms by utilizing the Electron framework. We aren't going to build the game engine, though. To save time and to focus on the result, I am going to use Phaser. Phaser is a fast, free, and fun open source framework for Canvas and WebGL that powers browser and mobile games. You can check out Phaser at <https://www.phaser.io/>.

First, I am going to guide you through the process of creating a new game project scaffold. We will be packaging simple Phaser examples as a desktop application in order to render images and manipulate game sprites.

A sprite is a computer graphics term for a two-dimensional bitmap that is integrated into a larger scene, most often in a 2D video game. Essentially, a sprite is a game object that we can move, flip, manipulate, and so on.

We will also learn about handling keyboard events. By the end of this chapter, you will know how to get started with game development for multiple platforms.

In this chapter, we will cover the following topics:

- Configuring a game project
- Running a Hello World example
- Rendering background images
- Preventing window resizing
- Rendering a sprite
- Scaling sprites
- Handling keyboard input
- Flipping sprites based on their direction
- Controlling sprite coordinates
- Controlling sprite speed

Technical requirements

To get started with this chapter, you will need a standard laptop or desktop running macOS, Windows, or Linux.

The minimal amount of software you need to have installed for this chapter is as follows:

- Git, a version control system
- Node.js with NPM
- Visual Studio Code, a free and open source code editor

You can find the code files for this chapter in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter05>.

Configuring a game project

We need to configure at least a basic Electron application. Choose a destination for the project files and follow these steps to get started:

1. Let's start by creating a new folder called `game` so that we can store the game project's files and assets:

```
mkdir game
cd game
```

2. Next, we'll initialize the project and install the Electron and Phaser libraries:

```
npm init -y
npm i electron
npm i phaser
```

3. As you already know, we need to have a `start` script in the `scripts` section of the `package.json` file. Also, don't forget to update the `main` entry point.
4. Your file should look as follows:

```
{
  "name": "game",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "start": "electron ."
  },
}
```

```
"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
  "electron": "^7.0.0",
  "phaser": "^3.20.1"
}
}
```

5. Finally, you need to place the `main.js` file in the project's root folder. The minimum amount of content that we need so that the game can run is shown in the following code:

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

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    }
  });
  win.loadFile('index.html');
}

app.on('ready', createWindow);
```

While you can declare the JavaScript code directly in the `index.html` file, I strongly recommend storing the game's content in a separate `game.js` file.

6. The minimum amount of content that we need for the `game.js` file is shown in the following code:

```
var config = {
  type: Phaser.AUTO,
  width: 800,
  height: 600,
  backgroundColor: '#03187D',
  scene: {
    preload: preload,
    create: create,
    update: update
  }
};
var game = new Phaser.Game(config);
```

```
function preload() {}  
function create() {}  
function update() {}
```

As you can see, the initial configuration is self-explanatory. We create a window that's 800 x 600 pixels in size with a predefined background color and a few function references.

There are three common functions that you are often going to use when you start a new game, that is, `preload`, `create`, and `update`. Let's go over these now:

- The `preload` function is called when the game is about to start. This is useful when you want to render a beautiful *Loading* screen with a progress bar. That is also where you can load all of the game's assets.
- The `create` function is the primary builder of your game. All of the initialization logic happens here, for example, setting the background, creating game characters, and configuring keyboard and mouse input.
- Last but not least, the `update` function is called every time the game needs to update its state. This is the most frequently called function and is usually invoked numerous times per second.

We will look into the main functions in more detail shortly, but first, let's finish the project's setup by following these steps:

1. To finish the project's setup, we need to place the `index.html` file in the project's root folder and import the `phaser.min.js` and `game.js` files:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="UTF-8" />  
    <title>Hello World!</title>  
  </head>  
  <body>  
    <script src="node_modules/phaser/dist/phaser.min.js"></script>  
    <script src="game.js"></script>  
  </body>  
</html>
```

2. Also, I suggest creating a dedicated CSS stylesheet file for the game. We need it so that we can remove all of the document margins and disable the scrollbars.
3. Create a `game.css` file next to the `game.js` file and put the following content inside it:

```
body {  
  margin: 0;
```



```
    overflow: hidden;
  }
```

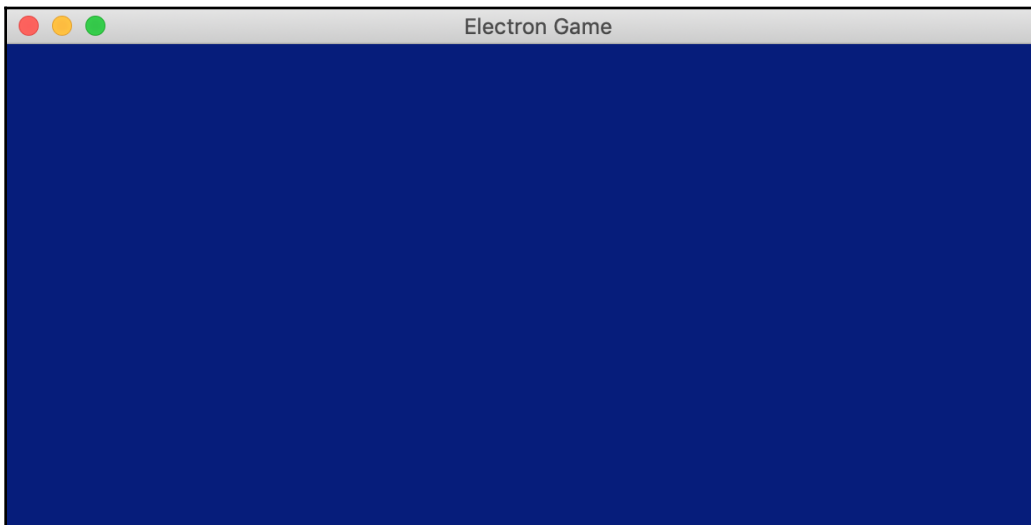
4. Update the game window title and import the stylesheet. The content of the `index.html` file should now look as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Electron Game</title>
    <link rel="stylesheet" href="game.css" />
  </head>
  <body>
    <script src="node_modules/phaser/dist/phaser.min.js"></script>
    <script src="game.js"></script>
  </body>
</html>
```

5. To see our first game project in action, run the following command in a Terminal window or Command Prompt:

npm start

6. Once the application is started up, you should see a window with a dark blue surface, as shown in the following screenshot, which means that our Phaser game is up and running and that we can render the background as expected:



Our game doesn't do much yet, but it's a good template for all of your future game projects. To practice with this more, let's look at an official Phaser example and wrap it with our project's scaffold.

Running a Hello World example

Let's take the official `Hello World` example and turn it into a desktop application powered by Electron. Follow these steps to do so:

1. First, create the configuration file shown in the following code. It is almost the same one that we had earlier, but with an extra `physics` configuration section:

```
var config = {
  type: Phaser.AUTO,
  width: 800,
  height: 600,
  backgroundColor: '#03187D',
  physics: {
    default: 'arcade',
    arcade: {
      gravity: { y: 200 }
    }
  },
  scene: {
    preload: preload,
    create: create,
    update: update
  }
};
```

2. Next, let's implement the `preload` function and load some image resources:

```
function preload() {
  this.load.setBaseURL('http://labs.phaser.io');
  this.load.image('sky', 'assets/skies/space3.png');
  this.load.image('logo', 'assets/sprites/phaser3-logo.png');
  this.load.image('red', 'assets/particles/red.png');
}
```

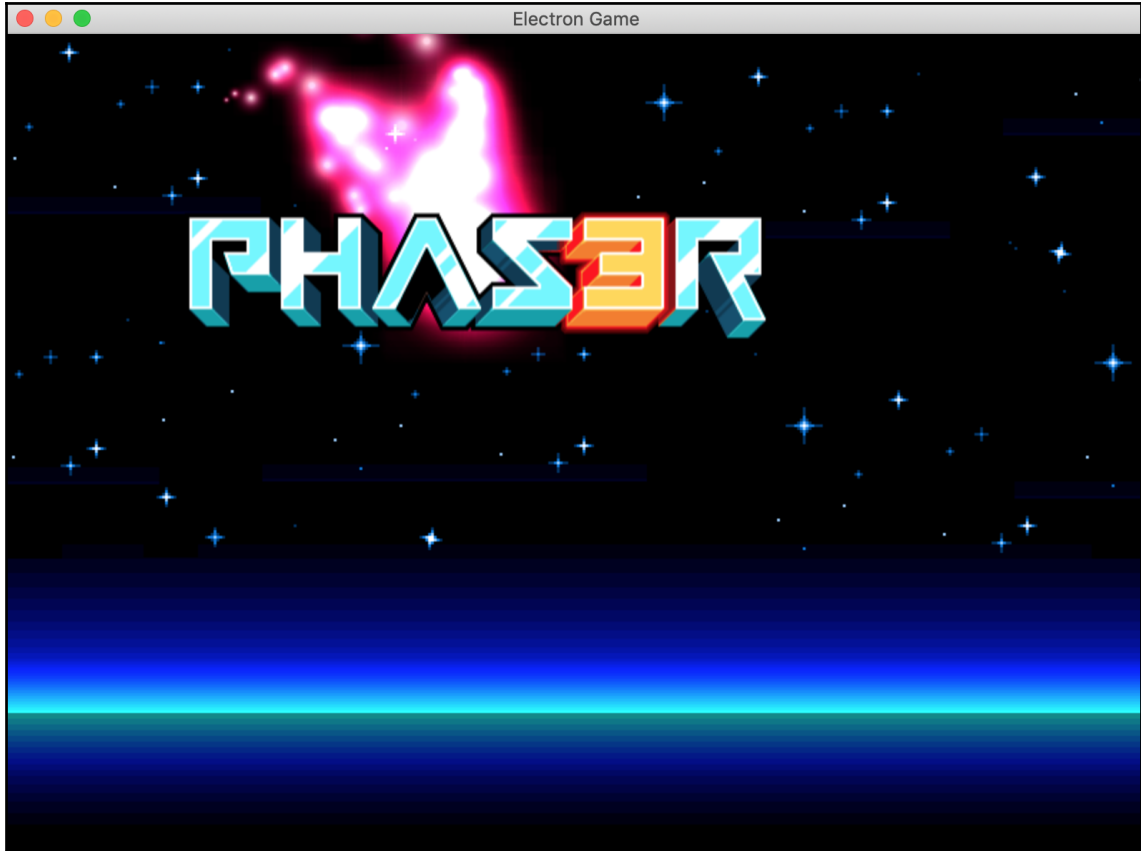
There are two essential points of interest in the preceding code. Notice that we can set the base URL for all the game assets. This can be a local or a remote address, depending on your scenario. Sometimes, you may want to store your assets remotely so that you can, for example, update the game server and apply changes to all the clients. For the current demo, we're instructing the game to fetch all the resources from the `http://labs.phaser.io` web address.

Another point of interest is how we load the game assets. The Phaser framework allows us to load an image file and give it a unique key that can be used in the game's code. This is very convenient as it allows us to change the asset image in a single place without changing its key in multiple places.

3. Now, update the `create` function according to the following code:

```
function create() {  
    this.add.image(400, 300, 'sky');  
    var particles = this.add.particles('red');  
  
    var emitter = particles.createEmitter({  
        speed: 100,  
        scale: { start: 1, end: 0 },  
        blendMode: 'ADD'  
    });  
  
    var logo = this.physics.add.image(400, 100, 'logo');  
  
    logo.setVelocity(100, 200);  
    logo.setBounce(1, 1);  
    logo.setCollideWorldBounds(true);  
  
    emitter.startFollow(logo);  
}
```

4. Restart the application. This time, you should see a *Phaser* logo bouncing around the custom background. Apart from this, the particle emitter adds some special effects to the logo sprite:



Based on our previous experience, let's change the game so that it loads our custom background and renders some different sprites. I suggest that we find a space background image and use a spaceship image on top of it. It would also be nice to have keyboard input handling so that users can control where the ship flies.

Rendering background images

We are going to store all our game assets locally, and our game is going to be running fully offline. Let's look at how we can render the background images:

1. Create a new `assets` folder in the project root so that you have somewhere to store your files.
2. After that, find and download a beautiful space picture, like the one shown in the following screenshot:



You can find the preceding background image in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/blob/master/Chapter05/assets/background.jpg>.

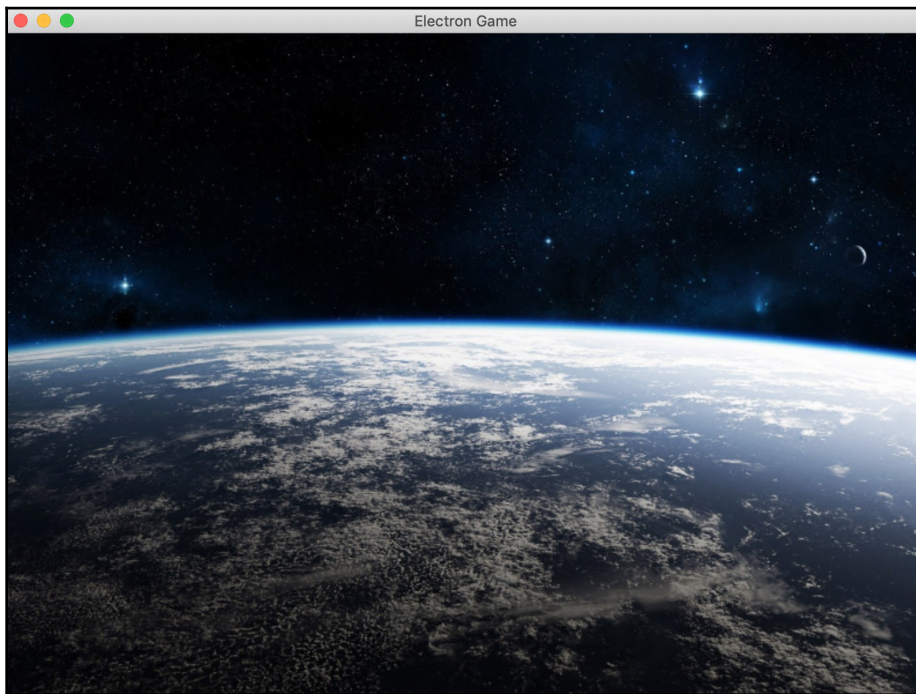
3. Save it as an `assets/background.jpg` file.
4. Update the `preload` function and set the base URL to the local application folder. Then, fetch the background image. Here, we're giving it the `background` key. Use the following code to reference this image:

```
function preload() {  
  this.load.setBaseURL('.');  
  this.load.image('background', 'assets/background.jpg');  
}
```

5. To display this image as the game's background, we need to perform some more image manipulation. The original image may not necessarily be 800 x 600 pixels size, or we may want to have different window sizes. In any case, our image needs to fit the entire window and should be scaled.
6. Adding to the game scene and scaling the background image is what our update function is going to do:

```
function create() {  
  const image = this.add.image(  
    this.cameras.main.width / 2,  
    this.cameras.main.height / 2,  
    'background'  
  );  
  let scaleX = this.cameras.main.width / image.width;  
  let scaleY = this.cameras.main.height / image.height;  
  let scale = Math.max(scaleX, scaleY);  
  image.setScale(scale).setScrollFactor(0);  
}
```

7. We have already defined the preload and create function implementations. Before we take a look at the update function, let's take a look at the application:



The application looks amazing. The image fits the main content area, but for the sake of simplicity, let's see how we can switch off resizing for our Electron shell.

Preventing window resizing

So far, we've created an Electron window that's 800 x 600 pixels in size. We also initialized a *Phaser* game with the same size parameters. If you don't want to deal with scaling and resizing and want to restrict the size of the screen, you can do so in the `main.js` file:

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

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    },
    resizable: false
  });

  win.loadFile('index.html');
}

app.on('ready', createWindow);
```

You can always disable this option later and adopt the game in multiple screen sizes. Now, let's create a spaceship sprite.

Rendering a sprite

Let's learn how to can render a sprite:

1. Find an image of a spaceship and upload it to the `assets` folder. In my example, I'm using the `phaser-ship.png` file.
2. Now, load the image into the game as a `ship` asset:

```
function preload() {
  this.load.setBaseURL('.');
  this.load.image('background', 'assets/background.jpg');
  this.load.image('ship', 'assets/phaser-ship.png');
}
```

3. Next, create a global variable called `ship`. This is going to contain our spaceship sprite:

```
const game = new Phaser.Game(config);  
let ship;
```

4. Now, assign the `ship` variable from within the `create` function. We need to reuse the variable inside the `update` function:

```
// Create ship  
ship = this.add.sprite(100, 100, 'ship');
```

Notice how we passed the initial coordinates and the asset key to add a new sprite to the game. You can control the position of the sprite on the screen when it first appears. In our case, it has coordinates of 100 pixels from the left and 100 pixels from the top.

You can also scale the image if you want the sprite to be bigger or smaller. This is something we are going to address in the next section.

Scaling sprites

Just like the background, the original image for our spaceship may be either too big or too small. In my case, it is tiny, and I need to scale it a bit. Let's see how we can do that:

1. To set custom scaling in the form of a float number, use the `sprite.setScale(x, y)` method:

```
// Create ship  
ship = this.add.sprite(100, 100, 'ship');  
ship.setScale(4, 4);
```

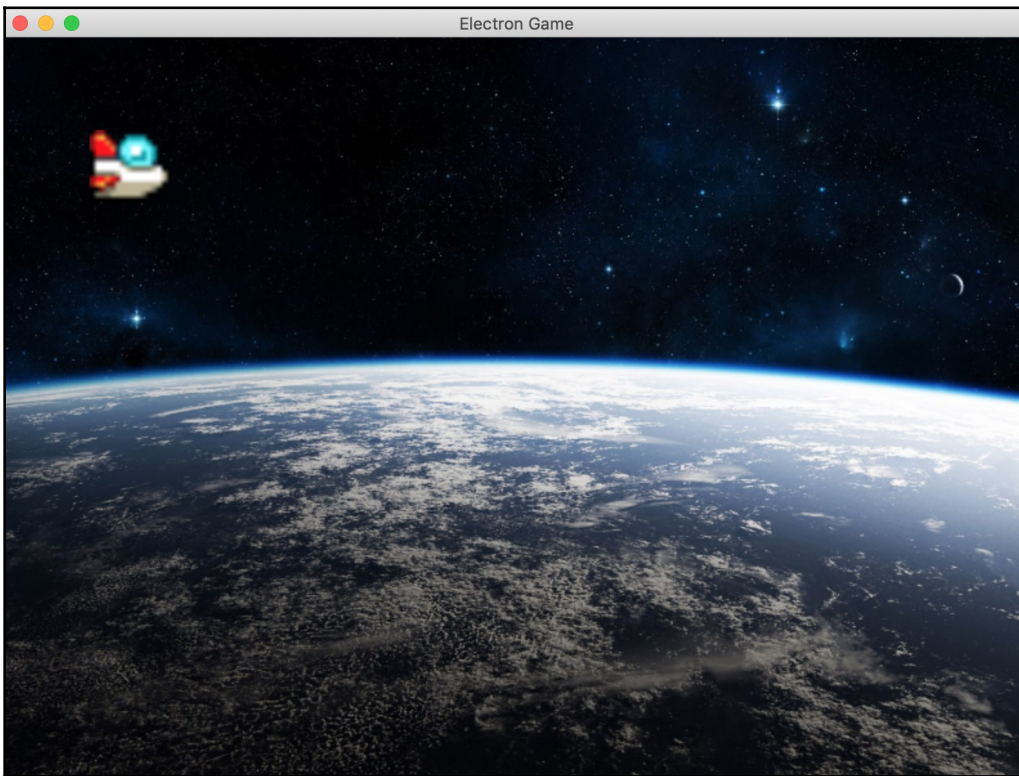
2. Along with the global variables, the `create` implementation of our game should look as follows:

```
const game = new Phaser.Game(config);  
let ship;  
  
function create() {  
  // Create background  
  const image = this.add.image(  
    this.cameras.main.width / 2,  
    this.cameras.main.height / 2,  
    'background'  
  );  
  let scaleX = this.cameras.main.width / image.width;
```



```
let scaleY = this.cameras.main.height / image.height;  
let scale = Math.max(scaleX, scaleY);  
image.setScale(scale).setScrollFactor(0);  
  
// Create ship  
ship = this.add.sprite(100, 100, 'ship');  
ship.setScale(4, 4);  
}
```

3. Now, restart the application. You should see a spaceship image displayed at the position 100:100 on the screen. It has also been scaled to be four times bigger than the original image:



Typically, you want your assets to be the size that the game requires them to be so that you don't waste CPU and RAM on scaling things at runtime. However, you should still know how to scale images; that is why we used a smaller ship and then resized it.

Now, let's look at how we can handle keyboard input to move the ship around the screen.

Handling keyboard input

In this section, we are going to provide keyboard support for our minigame. Our users should be able to use the cursor keys to move the ship in all directions. To access the state of the keyboard keys, we need a global variable to hold the state of the pressed keys. Let's learn how to do this:

1. Let's call it `cursors` and put it under the `game` instance:

```
const game = new Phaser.Game(config);  
let cursors;
```

2. The `create` function allows you to access the `input.keyboard` object. You can use this object to retrieve a reference to the cursor keys' state:

```
// Create cursors  
cursors = this.input.keyboard.createCursorKeys();
```

According to the official documentation, this creates a new object called `cursors`. It contains four objects: `up`, `down`, `left`, and `right`. These are all `Phaser.Key` objects, so anything you can do with a `Key` object you can do with these.

3. Now, we can check the state of the keys in the `update` method. As you may recall, the `update` method usually gets called multiple times per second, so it is a perfect place to check the state of the input controls and update the game accordingly:

```
function update() {  
  if (cursors.right.isDown) {  
    ship.x += 2;  
  }  
}
```

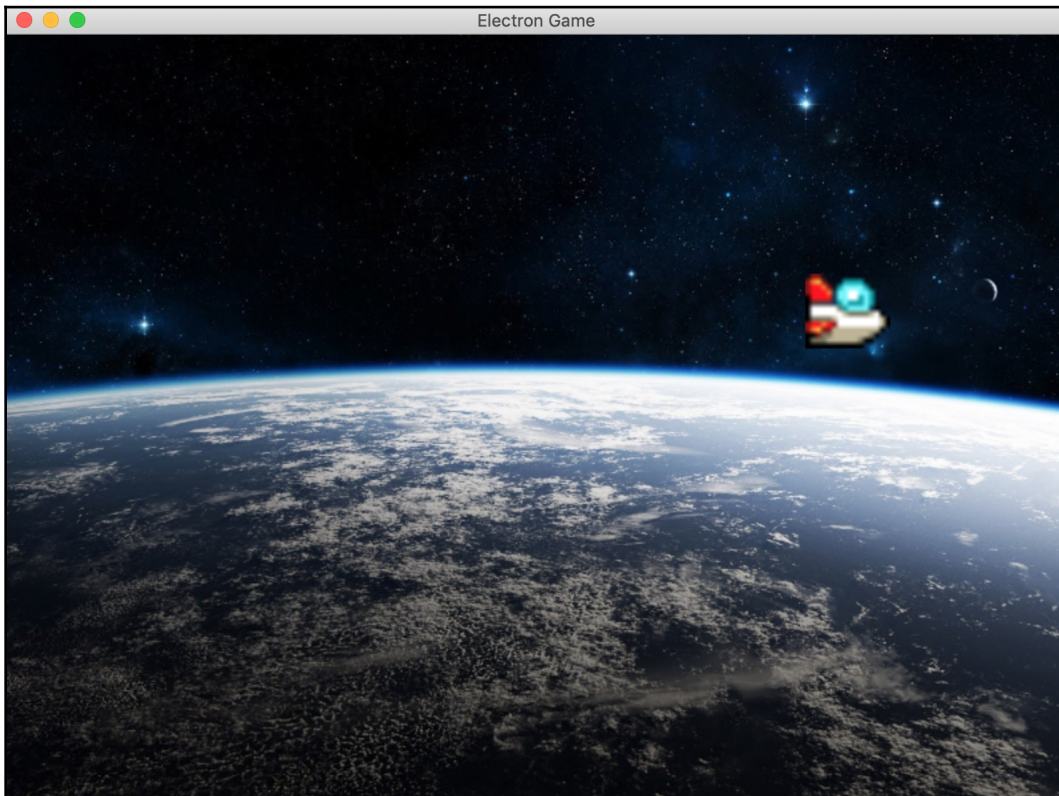
4. Every time the `update` method is invoked, we check the state of the `right` key and increment the horizontal position of the ship by 2 if the key is in the pressed, or down, state. In the same way, if the `left` key is pressed, we decrease the horizontal position of the ship by 2, as shown in the following code:

```
function update() {  
  if (cursors.right.isDown) {  
    ship.x += 2;  
  } else if (cursors.left.isDown) {  
    ship.x -= 2;  
  }  
}
```

5. Now, we can navigate our spaceship horizontally, and it shouldn't difficult for you to support the vertical axes as well. Please refer to the following code snippet to get an idea of how to perform vertical navigation:

```
function update() {  
  if (cursors.right.isDown) {  
    ship.x += 2;  
  } else if (cursors.left.isDown) {  
    ship.x -= 2;  
  } else if (cursors.up.isDown) {  
    ship.y -= 2;  
  } else if (cursors.down.isDown) {  
    ship.y += 2;  
  }  
}
```

6. Now, restart your Electron game and try using the keyboard keys. Notice how the ship moves in all four directions:



In this section, you have successfully implemented keyboard support for your game project. We received the cursor key's state and changed the coordinates of the ship according to user actions. Now, let's make the ship's behavior more natural and make it face the direction it's moving in.

Flipping sprites based on their direction

When navigating the ship, you should notice that it is always facing right. This is the expected behavior since the original image that we used for the sprite is facing right.

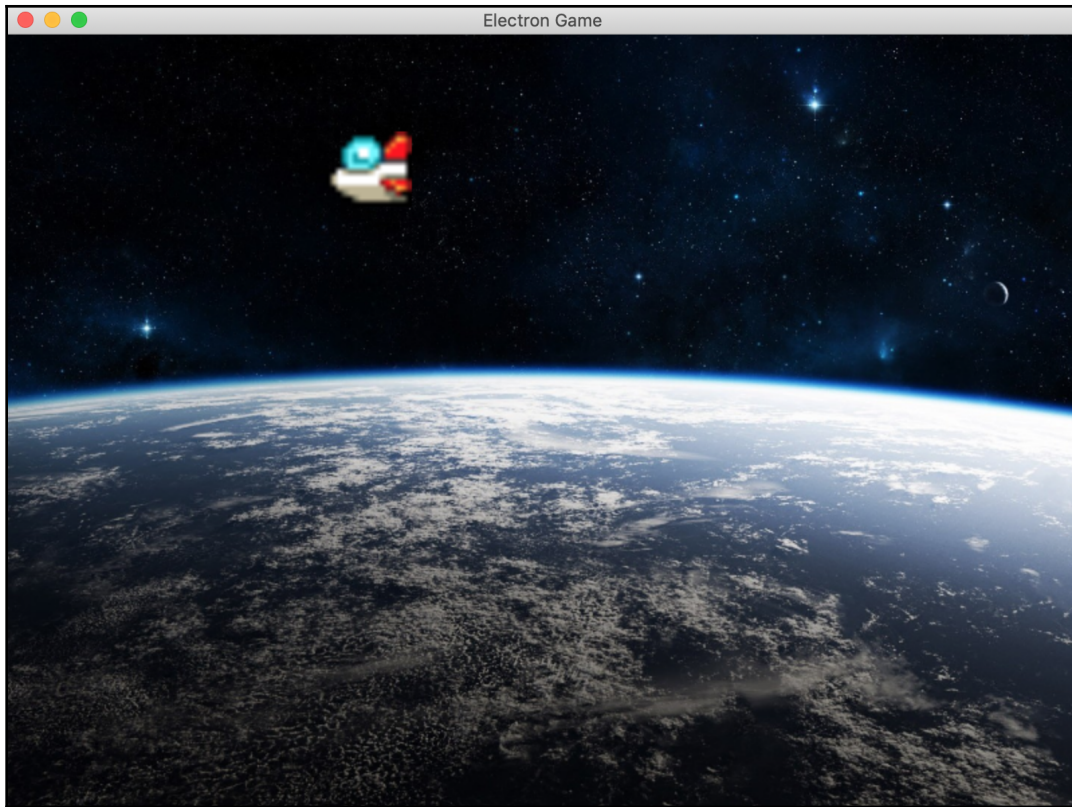
In real life, however, you may want the ship to face the direction that it's moving in. Luckily, the Photon framework supports flipping sprite images, thereby allowing us to invert the ship either horizontally or vertically using the following command:

```
sprite.flipX = true;
```

Now, let's update the code so that we can flip the image based on the keyboard state:

```
function update() {  
    // RIGHT button  
    if (cursors.right.isDown) {  
        ship.x += 2;  
        ship.flipX = false;  
    }  
    // LEFT button  
    else if (cursors.left.isDown) {  
        ship.x -= 2;  
        ship.flipX = true;  
    }  
    // UP button  
    else if (cursors.up.isDown) {  
        ship.y -= 2;  
    }  
    // DOWN button  
    else if (cursors.down.isDown) {  
        ship.y += 2;  
    }  
}
```

Restart the game application and try moving the ship left and right. Notice how the image changes to reflect the direction it's moving in:



In this section, we have made the ship's behavior more natural by flipping the image according to the direction it's moving in. Next, we need to prevent the ship from going off screen.

Controlling sprite coordinates

Let's prevent the sprite from going off screen. Here, we are going to render the ship in another part of the screen. Follow these steps to do so:

1. First, let's set the screen size to dedicated constants so that we can use it in our code:

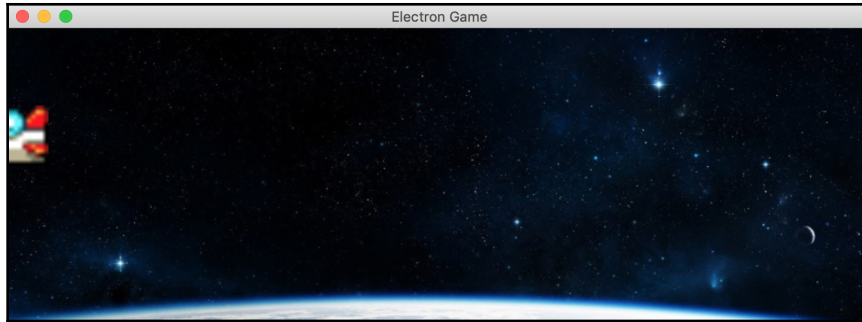
```
const screenWidth = 800;
const screenHeight = 600;
var config = {
  type: Phaser.AUTO,
```

```
    width: screenWidth,
    height: screenHeight,
    backgroundColor: '#03187D',
    scene: {
      preload: preload,
      create: create,
      update: update
    }
  };
```

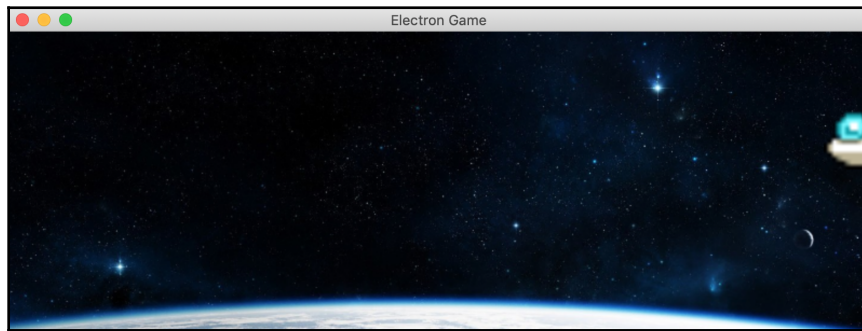
2. Now, on each update, you can check whether the new coordinates of the ship are off screen and change the value to point to another place, or maybe even prevent the ship from moving if you want the ship to stay where it is:

```
function update() {
  // RIGHT button
  if (cursors.right.isDown) {
    ship.x += 2;
    ship.flipX = false;
  }
  // LEFT button
  else if (cursors.left.isDown) {
    ship.x -= 2;
    if (ship.x <= 0) {
      ship.x = screenWidth;
    }
    ship.flipX = true;
  }
  // UP button
  else if (cursors.up.isDown) {
    ship.y -= 2;
  }
  // DOWN button
  else if (cursors.down.isDown) {
    ship.y += 2;
  }
}
```

3. As you can see, each time the horizontal or x coordinate becomes less than zero, that is, the image is going off the left-hand side of the screen, we reassign it to the value of the screen's width, that is, the right-hand side of the game screen.
4. Restart the game and press the *Left* arrow key until the ship reaches the left-hand side of the game screen:



5. Now, try moving the ship further off of the screen. Notice how it appears from the right-hand side and keeps moving to the center:



6. Now, let's do the same for the vertical axis:

```
function update() {  
  // RIGHT button  
  if (cursors.right.isDown) {  
    ship.x += 2;  
    if (ship.x >= screenWidth) {  
      ship.x = 0;  
    }  
    ship.flipX = false;  
  }  
  // LEFT button  
  else if (cursors.left.isDown) {  
    ship.x -= 2;  
    if (ship.x <= 0) {  
      ship.x = screenWidth;  
    }  
    ship.flipX = true;  
  }  
}
```

```
// UP button
else if (cursors.up.isDown) {
    ship.y -= 2;
    if (ship.y <= 0) {
        ship.y = screenHeight;
    }
}
// DOWN button
else if (cursors.down.isDown) {
    ship.y += 2;
    if (ship.y >= screenHeight) {
        ship.y = 0;
    }
}
}
```

When the ship reaches the bottom edge of the screen, we move it to the top, and vice versa. Now, it's impossible for the ship to leave the screen; the ship always appears on the opposite edge of the screen.

In this section, we have learned how to control the sprite's coordinates on the screen. Now, let's learn how to make the ship move faster by controlling its speed.

Controlling sprite speed

In the `update` calls, we have been incrementing the position of the ship sprite by 2. In real life, however, you may want to store that value as a global constant or a centralized setting. In this case, changing the overall speed means that we need to update a single constant or variable, instead of refactoring the whole game.

We have already moved the screen size into constants; let's do the same with the speed:

1. Introduce a new constant called `shipSpeed` and set its value to 2:

```
const screenWidth = 800;
const screenHeight = 600;
const shipSpeed = 2;
```

2. Now, update all of the existing code and use the `shipSpeed` constant in all of the places you need to increment or decrement the position of the ship, as shown in the following code:

```
function update() {
    // RIGHT button
    if (cursors.right.isDown) {
```



```
    ship.x += shipSpeed;
    if (ship.x >= screenWidth) {
        ship.x = 0;
    }
    ship.flipX = false;
}
// LEFT button
else if (cursors.left.isDown) {
    ship.x -= shipSpeed;
    if (ship.x <= 0) {
        ship.x = screenWidth;
    }
    ship.flipX = true;
}
// UP button
else if (cursors.up.isDown) {
    ship.y -= shipSpeed;
    if (ship.y <= 0) {
        ship.y = screenHeight;
    }
}
// DOWN button
else if (cursors.down.isDown) {
    ship.y += shipSpeed;
    if (ship.y >= screenHeight) {
        ship.y = 0;
    }
}
}
```

As you can see, you now have a single place that holds the speed of the ship. Any changes that are made to the `shipSpeed` constant will be reflected in all of the code blocks since we're not refactoring the code in multiple places.

3. Now, change the speed to 4 and see what happens:

```
const shipSpeed = 4;
```

Notice how the ship is now two times faster than before when it moves in all four directions. You can experiment more and either increase or decrease the speed of the ship until you find a value that provides the best and smooth movement behavior.



It is good practice to use constants or variables to control various aspects of the game in a single place. This helps us to avoid code duplication and the need for refactoring.

Summary

Congratulations on finishing this chapter, which was related to game development and the Electron framework. Together, we have created a simple game project that you can extend to make a genuine cross-platform game. In this chapter, you managed to load and render window backgrounds, draw and manipulate game sprites, and control keyboard input. This is an excellent foundation that you can build your knowledge on.

If you liked working with the Phaser framework, make sure you look through the official learning materials and guides at <http://phaser.io/learn>. You can find the complete project's source code in this book's assets, inside the `game` folder.

In the next chapter, we are going to build a music player that provides playback control, metadata, and cover albums.

6

Building a Music Player

Now that you know how to create basic Electron applications, let's craft something interesting that involves multimedia components and your hardware.

In this chapter, we are going to build a simple music player application with playback controls, sound options, metadata, and album art rendering.

You are going to learn how to use web technologies to build a cross-platform music player for desktop applications and create a good foundation project that you can make further enhancements to. By the end of this chapter, we will have a minimal music player application with cover album art and song metadata support.

In this chapter, we will cover the following topics:

- Creating a project scaffold
- Exploring the music player component
- Exploring the playback control buttons
- Implementing a song progress bar
- Displaying music metadata
- Improving the user interface
- Reviewing the final structure

Let's start by configuring a new project scaffold for our music application.

Technical requirements

To get started with this chapter, you will need a standard laptop or desktop running macOS, Windows, or Linux.

The software you will need to have installed to be able to complete this chapter is as follows:

- Git, a version control system
- Node.js with NPM
- Visual Studio Code, a free and open source code editor

You can find the code files for this chapter in this book's GitHub repository at <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter06>.

Creating a project scaffold

As in all the previous chapters, let's start by creating a project scaffold. We are going to call our project `music-player`; it is going to use a pure JavaScript and HTML5 stack without any additional frameworks. You can always wire a framework of your choice later.

Let's create a new project using the Terminal window or Command Prompt:

1. Navigate to your `projects` or home folder.
2. Run the following commands in your Terminal window or Command Prompt:

```
mkdir music-player
cd music-player
```

The preceding commands create a new directory for your music player application.

3. Now it's time to initialize the project and generate the `package.json` file. Use the following commands to set up an `npm` project:

```
npm init -y
echo node_modules > .gitignore
npm i -D electron
```

As you can see, besides setting up a new project with NPM, we've also generated a minimal `.gitignore` file in case we ever decide to use GitHub or GitLab repositories to store our project code. We have also installed the Electron framework dependency.

4. Update the `main` and `scripts` sections according to the following code:

```
{
  "name": "music-player",
```

```
"version": "1.0.0",
"description": "",
"main": "main.js",
"scripts": {
  "start": "electron ."
},
"keywords": [],
"author": "",
"license": "ISC",
"devDependencies": {
  "electron": "^7.0.0"
}
}
```

The `main.js` application file is pretty much the same template we've used in every chapter so far. For now, let's start with a non-resizable window that's 800x600 in size with Node.js integration enabled.

5. Use the following code to create a new `main.js` file in the project root:

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

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    },
    resizable: false
  });

  win.loadFile('index.html');
}

app.on('ready', createWindow);
```

6. The `index.html` content that we'll use to render our main application window will look as follows:

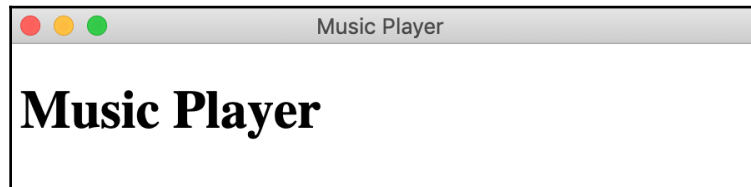
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
  </head>
  <body>
    <h1>Music Player</h1>
```

```
</body>  
</html>
```

7. Let's try out the application to see if everything works as expected. Run the following command in the Command Prompt:

```
npm start
```

8. Once the Electron application has started up, you should see the following screen:



At this point, we have a basic application template for our upcoming music player project. Feel free to back it up and use it in the future to save yourself some time when you're building similar projects.

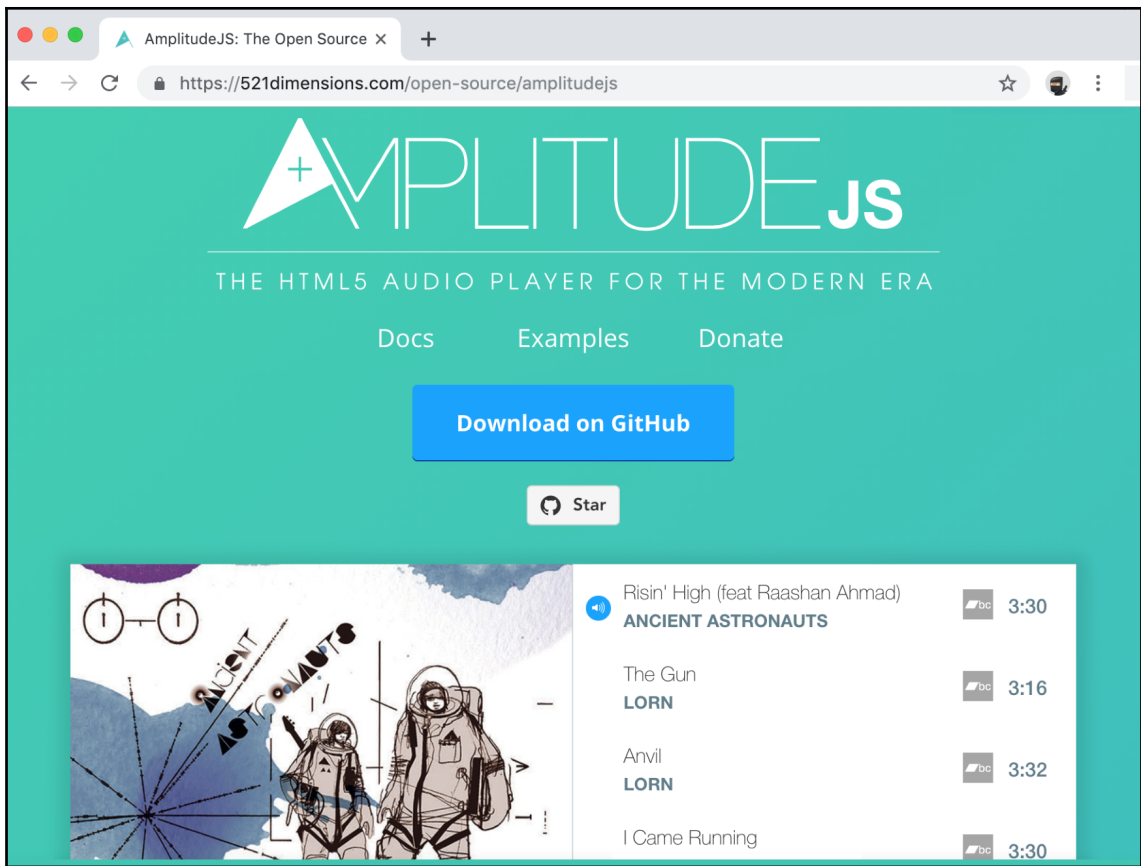
Now, let's move on to the most significant part of our journey—choosing a music component library. This will form the foundation of our project.

Exploring the music player component

Modern browsers provide full support for music and video playback through HTML5 and JavaScript. However, it may take some time to implement all the specifications and study all the APIs.

Also, building cross-browser music components is not a trivial task. That's why I strongly recommend that you look for existing third-party components—freeware or commercial—that already encapsulate the features we need to use.

For our project, we are going to use AmplitudeJS, the HTML5 audio player for the modern era. It requires no external dependencies:



Follow these steps to add a music player component:

1. First, install the AmplitudeJS JavaScript library using the following NPM command:

```
npm i amplitudejs
```

Although you can write JavaScript in the HTML document by using `script` tags, it's good practice to keep the scripts separate from the presentation layer.

2. Let's separate the concerns and introduce a new file called `player.js`. This file is going to hold all the code that is related to the music player's implementation and playback. Create the `player.js` file with the following stub content:

```
// player.js
// todo: player configuration
```

3. Now, update the `index.html` file and include the newly created `player.js` file at the bottom of the `body`, after the `amplitude.js` import.
4. The file's content should look as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
  </head>
  <body>
    <h1>Music Player</h1>
    <script src="./node_modules/amplitudejs/dist/amplitude.js">
    </script>
    <script src="./player.js"></script>
  </body>
</html>
```

At this point, we have an Electron application that loads and initializes the Amplitude library at startup. We also loaded our custom script file, `player.js`, after the `amplitude.js` file. This allows us to access all the multimedia APIs that are exposed by the third-party component library.

Now, it's time to grab some music files to test the application.

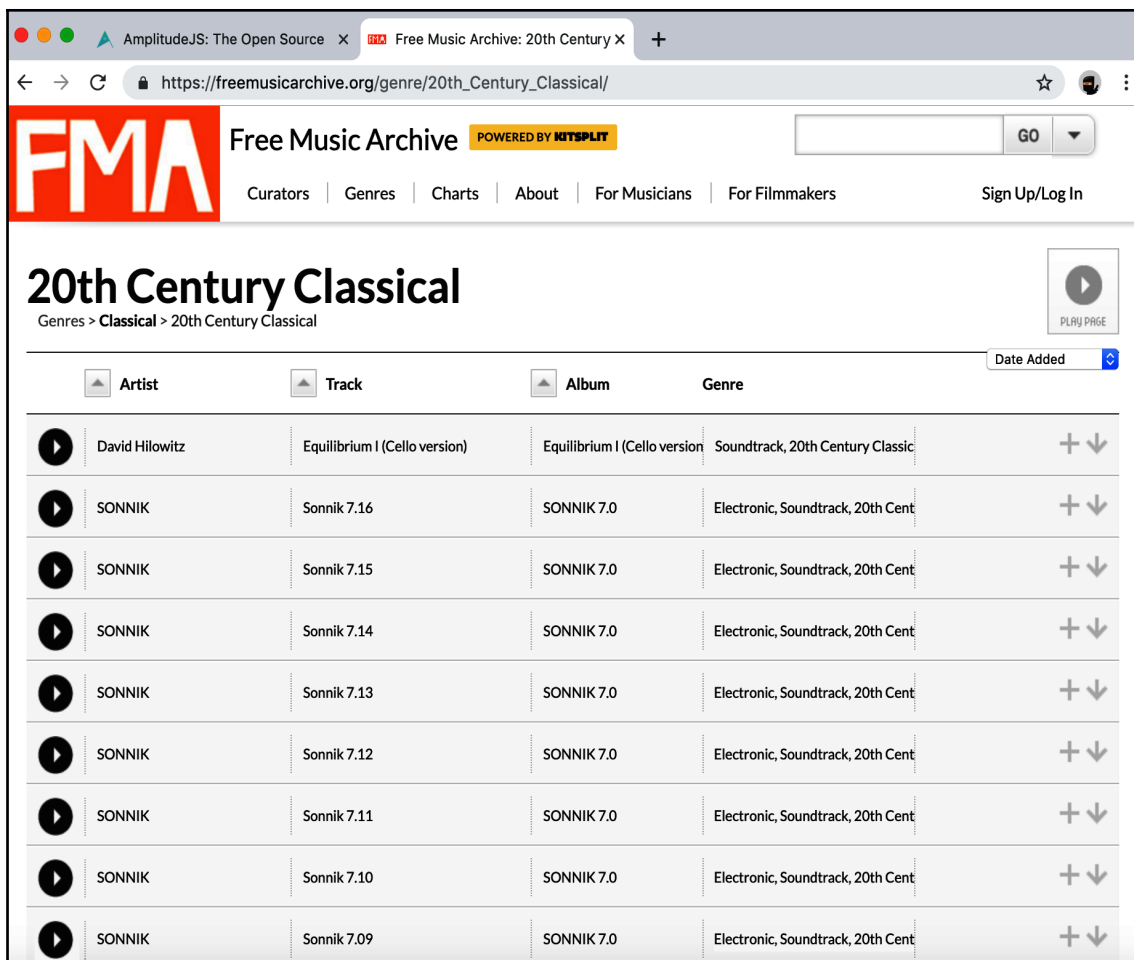
Downloading music files

To develop and test the application, you are going to need at least one music file containing metadata such as author, album title, and cover image.

We are going to use the **Free Music Archive** website to grab some files and metadata. However, feel free to use your own content once you've finished this chapter.

Follow these steps to get started:

1. Navigate to the **Free Music Archive** website (<http://freemusicarchive.org>) as follows:

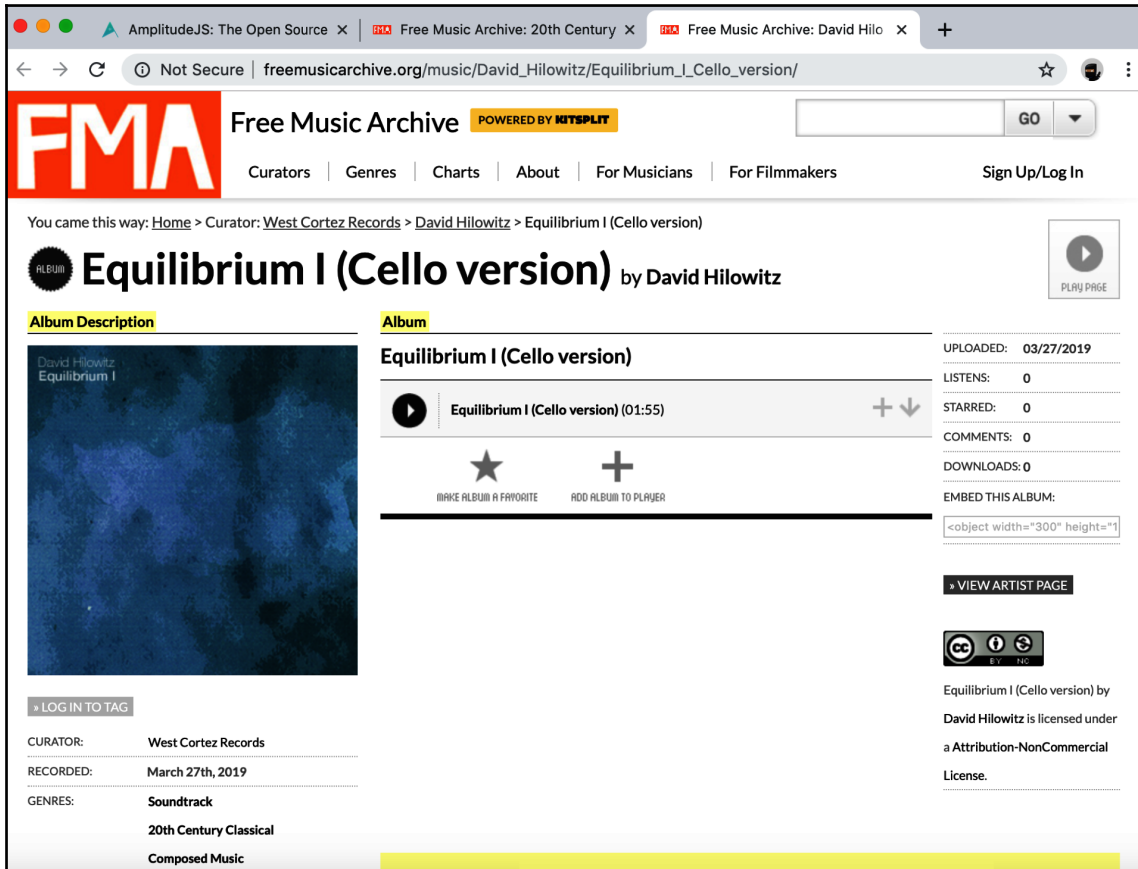


- Before you start downloading files, create a separate `music` directory. This is where we will store all of our media files for our application.
- Switch to your Terminal window or Command Prompt and run the following commands:

```
mkdir music
open .
```

The preceding commands create a folder called `music` and open the Finder or Explorer so that we can observe its contents.

- Next, find the **Equilibrium I (Cello version) by David Hilowitz** file. It should be available at http://freemusicarchive.org/music/David_Hilowitz/Equilibrium_I_Cello_version/:



- Download the music file; it should be named `David_Hilowitz_-_Equilibrium_I_Cello_version.mp3`.
- Also, right-click the album cover image and save it locally; we are going to use this later.



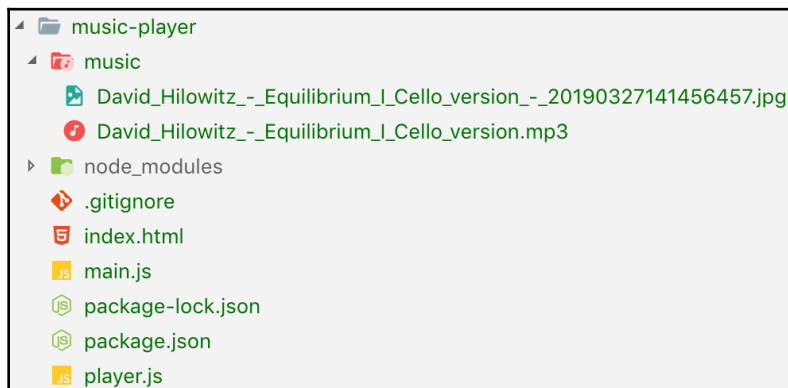
You can find all the music files for this chapter, along with the complete project code, in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter06/music>.

At this point, we have created a separate folder so that we can store all of our multimedia resources. You also have an experimental music file and an image to use as an album cover. In the next section, we are going to learn how to set up the player component and play the music file we just retrieved.

Providing basic player setup

In the previous section, we prepared a folder structure and obtained some files to use within our application. Now, let's learn how to initialize a music player component and play the sound files we have.

At this point, your project structure should look similar to the following:



Switch to the `player.js` file and configure a new instance of the player component by calling the `Amplitude.init` method with an object of configuration settings, as shown in the following code:

```
Amplitude.init({
  songs: [
    {
      name: 'Equilibrium I (Cello version)',
      artist: 'David Hilowitz',
      album: 'Equilibrium I (Cello version)',
      url: './music/David_Hilowitz_-_Equilibrium_I_Cello_version.mp3',
      cover_art_url:
        './music/David_Hilowitz_-_Equilibrium_I_Cello_version-
        20190327141456457.jpg'
    }
  ]
});
```

In the preceding example, we are using the following properties:

- `songs`: A list of songs to play. For now, we are using a single entry.
- `name`: The song's name.
- `artist`: The song's artist or band.
- `album`: Name of the album.
- `url`: The link to the music content. Here, we are using a local path, but you can point it to the remote web address if you wish.
- `cover_art_url`: The link to the album cover image. Similar to the `url` property, it can point to either a local or remote location.

Note that these are the basic metadata properties we are going to use when building the music player application. You can, however, store much more metadata content and use it on demand. We are going to address this later in this chapter.

In this section, we are going to address the following aspects:

- Using AmplitudeJS elements
- Implementing the global *play* button
- Implementing the global *pause* button
- Implementing the global *play/pause* button

Now, let's look at the elements that are available in AmplitudeJS and how we can use them.

Using AmplitudeJS elements

The significant benefits of using the AmplitudeJS library include the absence of external dependencies and no user interface to enforce their look and feel. Instead, AmplitudeJS relies on the CSS classes that are prefixed with `amplitude-`, which you can add to any HTML element. You can have any nested hierarchy of elements and a fully custom visual look that fits your application's theme.



Please take a look at the following article to get a good understanding of what elements are available for your applications: <https://521dimensions.com/open-source/amplitudejs/docs/elements/index.html>.

Implementing the global play button

To turn an HTML element into a `Play` button, you need to use the `amplitude-play` CSS class like so:

```
<span class="amplitude-play"></span>
```

As you can see, adding the `amplitude-play` class turns an element into a clickable object that starts music playback. Let's use this class with a button element in our application:

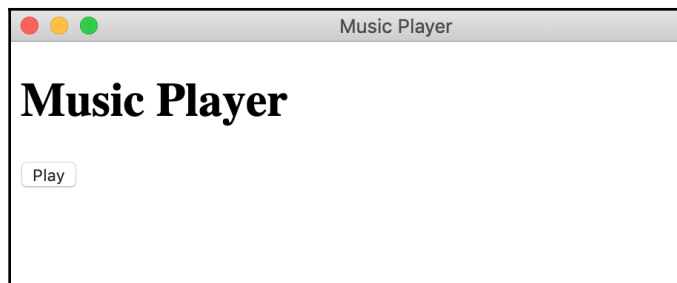
1. Declare the button element in the `index.html` file as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
  </head>
  <body>
    <h1>Music Player</h1>
    <div>
      <button class="amplitude-play">Play</button>
    </div>
    <script
src="./node_modules/amplitudejs/dist/amplitude.js"></script>
    <script src="./player.js"></script>
  </body>
</html>
```

2. Try out the application by running the following command from the Terminal window or Command Prompt:

```
npm start
```

3. As you can see, at startup, you now have a **Play** button under the dummy **Music Player** heading:



Now, if you click the **Play** button, you should hear the music playing.

We don't have any other music control buttons, so the only way to stop the player is to quit the application. Don't worry; we are going to address this shortly. Next, we are going to create a **Pause** button so that we can stop the music.

Implementing the global pause button

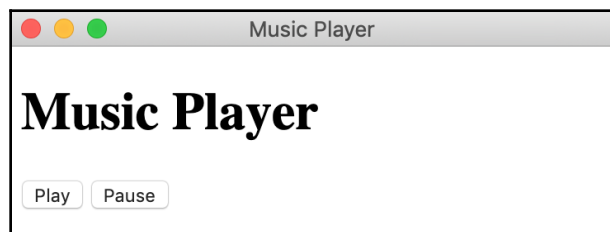
Similar to the global `Play` button, you can define an element to handle the `Pause` functionality as well. You should use the `amplitude-pause` CSS class for this. The usage format, according to the official documentation, is as follows:

```
<span class="amplitude-pause"></span>
```

You can turn any clickable element or complex web component into a `Pause` button. For the sake of simplicity, let's use the standard `HTML` `button` element. Update the `index.html` file according to the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
  </head>
  <body>
    <h1>Music Player</h1>
    <div>
      <button class="amplitude-play">Play</button>
      <button class="amplitude-pause">Pause</button>
    </div>
    <script src="./node_modules/amplitudejs/dist/amplitude.js"></script>
    <script src="./player.js"></script>
  </body>
</html>
```

Now, you should have two buttons rendered next to one another:



This time, you can pause music playback with the **Pause** button and resume or play the file with the **Play** button. All you did here was set the CSS class for the HTML `button` element.

Let's move on and slightly rework our buttons to improve our user experience.

Implementing the global play/pause button

In real life, you aren't going to have two separate buttons for playback. Typically, users prefer having a single button that both starts and pauses the song.

Luckily, AmplitudeJS provides support for this scenario as well. Here, we can use `amplitude-play-pause` to make an HTML element toggle the playback. The usage format is pretty much the same as it was for the `Play` and `Pause` functionality:

```
<span class="amplitude-play-pause"></span>
```

Comment out or remove the `Play` and `Pause` buttons and use a new unified button, as shown in the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
  </head>
  <body>
    <h1>Music Player</h1>
    <div>
      <button class="amplitude-play-pause">Play / Pause</button>
    </div>
    <script src="./node_modules/amplitudejs/dist/amplitude.js"></script>
    <script src="./player.js"></script>
  </body>
</html>
```

Before we dive into other music controls, let's check out how we can style our elements to make them look much better than default HTML styles.

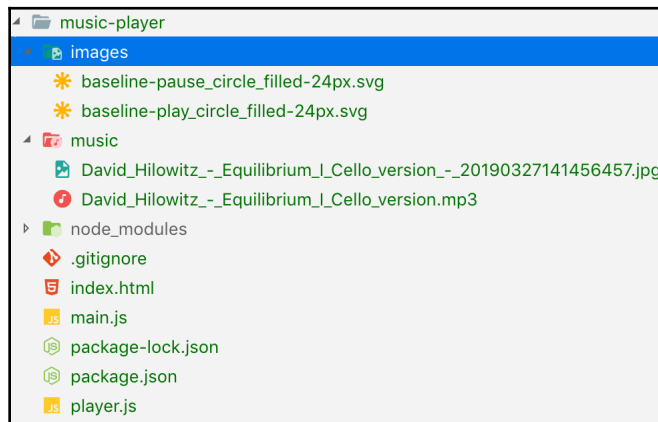
Styling buttons

In this section, we are going to render SVG icons for the `Play` and `Pause` states instead of the HTML `button`. The easiest and quickest way to get the corresponding resources is by using the Google Material Icons library.

Material Icons is an open source set of icons that have been created and maintained by Google. You can find these resources at <https://material.io/tools/icons>.

Let's learn how to use a button with a custom style and SVG images by following these steps:

1. First, create a folder called `images` in the project root. Download the SVG versions of the Play and Pause buttons and save them in the `images` folder.
2. At this point, your project structure should look similar to the following:



3. Next, replace the button element in the `index.html` file with the div one. We don't need it to be a button anymore as it is going to be an image-based component. Update your `index.html` code and use the div element like so:

```
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
    <link rel="stylesheet" href="player.css" />
  </head>
  <body>
    <h1>Music Player</h1>
    <div class="controls">
      <div class="amplitude-play-pause"></div>
    </div>
    <script src="./node_modules/amplitudejs/dist/amplitude.js">
    </script>
    <script src="./player.js"></script>
  </body>
</html>
```


4. As you can see, we also added an import of the `player.css` file. Having a separate file for the player styles is good practice, so let's create it next to the `player.js` script file with the following content:

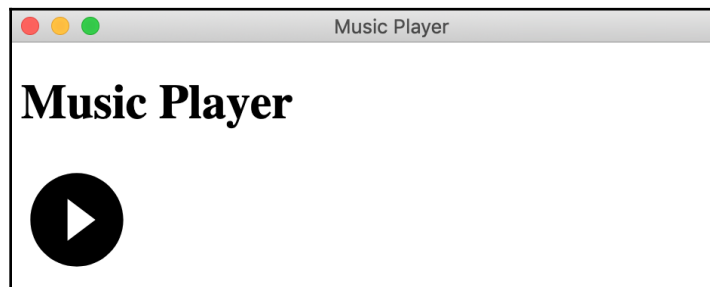
```
.controls .amplitude-play-pause {
  width: 74px;
  height: 74px;
  cursor: pointer;
}

.controls .amplitude-play-pause.amplitude-paused {
  background: url('../images/baseline-play_circle_filled-24px.svg');
  background-size: cover;
}
```

The button is going to be 74x74 pixels in size and have a `pointer` cursor to emulate a button effect.

Another helpful behavior of AmplitudeJS is the extra CSS classes that are attached to the `amplitude-` elements, depending on the player's state. For example, the element gets the `amplitude-paused` class appended to it when we pause the playback, while it gets the `amplitude-playing` class appended to it when we start or resume it. This is very convenient if you want to style the buttons differently.

5. Now, if you start or restart your application, you should see a cute `Play` button:

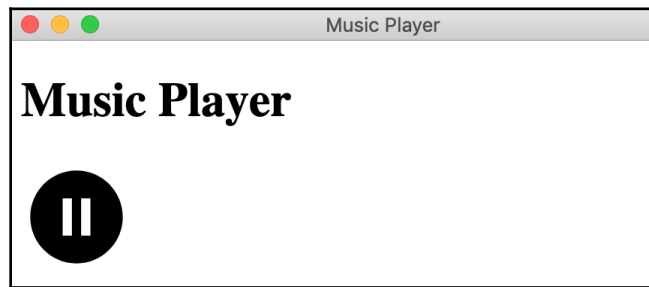


6. Now, let's add a separate style for the `Playing` mode:

```
.controls .amplitude-play-pause {
  width: 74px;
  height: 74px;
  cursor: pointer;
}
```

```
.controls .amplitude-play-pause.amplitude-paused {  
  background: url('./images/baseline-play_circle_filled-24px.svg');  
  background-size: cover;  
}  
  
.controls .amplitude-play-pause.amplitude-playing {  
  background: url('./images/baseline-pause_circle_filled-24px.svg');  
  background-size: cover;  
}
```

7. This time, the button in your application automatically changes to reflect the playback state, like so:



Now, you have a basic understanding of how to use music controls on a web page. Here, we've added play and pause buttons, provided custom styles, and even improved the user experience by merging two buttons into a single one.

In the next section, we are going to implement a traditional set of buttons to control playback.

Exploring the playback control buttons

We have made excellent progress with our music player application so far. You now have a play/pause button that works and allows you to listen to a song from within the Electron application. Of course, a single button isn't enough for a music player. In this section, we are going to implement a traditional set of buttons to control the music playback.

For comfortable user experience, we need to have at least the following buttons:

- Play/pause
- Stop

- Mute/unmute
- Volume up/down

You have already implemented the Play / Pause button, so let's move on to the Stop button.

Stop button

So far, our users can start the playback and pause it. However, there is no way for us to stop and reset the song's progress so that we can listen to the same song from the beginning, for example. Let's look at how we can add the Stop button:

1. Create the new Stop button using the following Amplitude CSS class:

```
<span class="amplitude-stop"></span>
```

2. Download the stop icon from Material Design Icons (<https://material.io/tools/icons/?icon=stopstyle=baseline>) and place it into the images folder, as we did earlier.
3. After that, update the index.html file and append the button element with the corresponding class to the controls element, which we are using to hold all the playback controls:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
    <link rel="stylesheet" href="player.css" />
  </head>
  <body>
    <h1>Music Player</h1>

    <div class="controls">
      <div class="amplitude-play-pause"></div>
      <div class="amplitude-stop"></div>
    </div>

    <script
src="./node_modules/amplitudejs/dist/amplitude.js"></script>
    <script src="./player.js"></script>
  </body>
</html>
```

4. Finally, we need to get back to the `player.css` file and provide a new style for the `.amplitude-stop` class so that we can render our button.
5. Append the following code to your stylesheet:

```
.controls .amplitude-stop {  
  width: 48px;  
  height: 48px;  
  cursor: pointer;  
  display: inline-block;  
  background: url('../images/baseline-stop-24px.svg');  
  background-size: cover;  
}
```

6. You can restart the application or press `Cmd + R` (`Ctrl + R`) to reload the window. We now have two buttons available, as shown in the following screenshot:



Note that when you click the Stop button, the Play / Pause functionality is also updated.

Given that we are going to add more buttons, let's optimize the CSS to avoid code repetition:

1. Create a separate style addressing all the child `div` elements inside the `.controls` parent.
2. Add the following code to the bottom of our `player.css` file:

```
.controls > div {  
  width: 48px;  
  height: 48px;  
  cursor: pointer;  
  display: inline-block;  
}
```

3. Now, refactor the `player.css` file so that it looks as follows:

```
.controls > div {
  width: 48px;
  height: 48px;
  cursor: pointer;
  display: inline-block;
}

.controls .amplitude-play-pause {
  width: 74px;
  height: 74px;
  cursor: pointer;
  display: inline-block;
}
```

As you can see, all our buttons are going to be 48x48 pixels in size, except for the Play / Pause button, which we make intentionally bigger. All the buttons have the same cursor and display settings.

Now it's time to move on and introduce the mute and unmute buttons.

Mute and unmute buttons

Similar to the Play / Pause button, we are going to introduce the Mute / Unmute button so that our users have more control over the music playback functionality. Let's get started:

1. Download the following icons from the **Material Design Icons** website:

- `volume_mute` (https://material.io/tools/icons/?icon=volume_mute&style=baseline)
- `volume_off` (https://material.io/tools/icons/?icon=volume_off&style=baseline)

2. You can enable the mute functionality for any HTML element using the following code:

```
<span class="amplitude-mute"></span>
```

3. When you click on the element and the player becomes muted, the span gets a secondary class, that is, `amplitude-muted`.

You can also address `amplitude-not-muted` if you want some extra styling. Follow these steps to add the buttons:

1. Update the `index.html` file so that it looks as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
    <link rel="stylesheet" href="player.css" />
  </head>
  <body>
    <h1>Music Player</h1>

    <div class="controls">
      <div class="amplitude-play-pause"></div>
      <div class="amplitude-stop"></div>
      <div class="amplitude-mute"></div>
    </div>

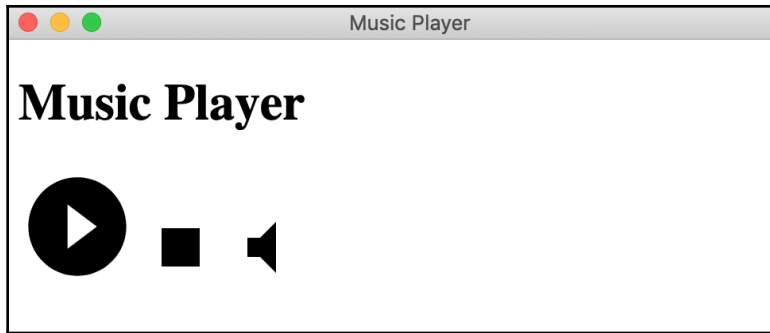
    <script src="./node_modules/amplitudejs/dist/amplitude.js">
    </script>
    <script src="./player.js"></script>
  </body>
</html>
```

2. Due to all the base classes we defined earlier, we only need to add the following content to the `player.css` file:

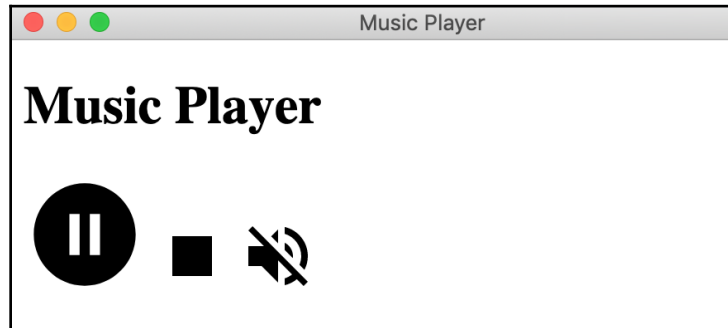
```
.controls .amplitude-mute {
  background: url('./images/baseline-volume_mute-24px.svg');
  background-size: cover;
}

.controls .amplitude-mute.amplitude-muted {
  background: url('./images/baseline-volume_off-24px.svg');
  background-size: cover;
}
```

3. Refresh the window or restart the application to see the changes in action. Your player should now display three buttons, including the `Mute` one we just created, as shown in the following screenshot:



4. Click the `Play` button to start the playback and then click the `Mute` button. Notice that you don't hear any sound. The button updates its style to show the muted icon:



Now, it's time to move on to the volume buttons. These buttons will allow our users to tune the output so that it's either louder or quieter.

Volume buttons

The Amplitude library provides you with two separate CSS classes so that you can convert your HTML elements into volume controls. You can use the following code to create the `Volume Up` control:

```
<span class="amplitude-volume-up"></span>
```

Similar to the preceding code, you can turn any element into the `Volume Down` control by attaching the following class to it:

```
<span class="amplitude-volume-down"></span>
```

Let's get two more images for our buttons. Follow these steps to do so:

1. Get the following icons from the **Google Material Icons** website:
 - `volume_down` (https://material.io/tools/icons/?icon=volume_down&style=baseline)
 - `volume_up` (https://material.io/tools/icons/?icon=volume_up&style=baseline)
2. Update the content of the `controls` element file according to the following code:

```
<div class="controls">
  <div class="amplitude-play-pause"></div>
  <div class="amplitude-stop"></div>
  <div class="amplitude-mute"></div>
  <div class="amplitude-volume-down"></div>
  <div class="amplitude-volume-up"></div>
</div>
```

Similar to the `Play` and `Mute` buttons, we have to define separate CSS styles for each state.

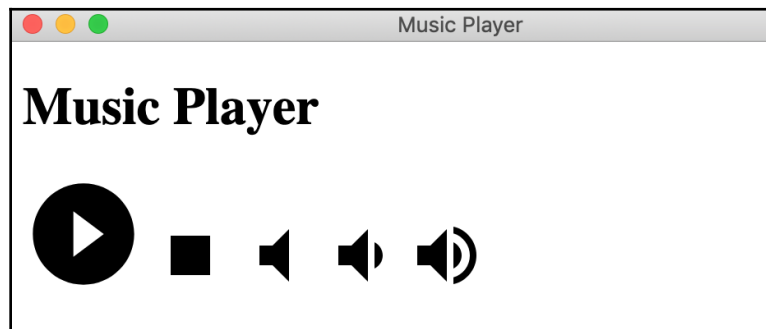
3. Update the `player.css` file and add the corresponding code:

```
.controls .amplitude-volume-up {
  background: url('../images/baseline-volume_up-24px.svg');
  background-size: cover;
}

.controls .amplitude-volume-down {
  background: url('../images/baseline-volume_down-24px.svg');
  background-size: cover;
}
```

These style classes are pretty similar. They receive most of the settings from the parent element rules and only provide the path to the background image.

4. Check out the application. This time, you are going to see five buttons in a row:



Note that clicking on the `Volume Down` button multiple times triggers the `Mute` button. It changes its state as soon as you reach a volume level of zero. By default, each click of the button increases or decreases the volume by 5%. You can change this step value when initializing the `Amplitude` object, but the end user's experience may become overwhelmed.

Instead of two buttons, let's have a range slider so that our end users can change the desired level of volume much faster than when they were using separate buttons:

1. Luckily, `AmplitudeJS` already provides support for ranges in the following format:

```
<input type="range" class="amplitude-volume-slider"/>
```

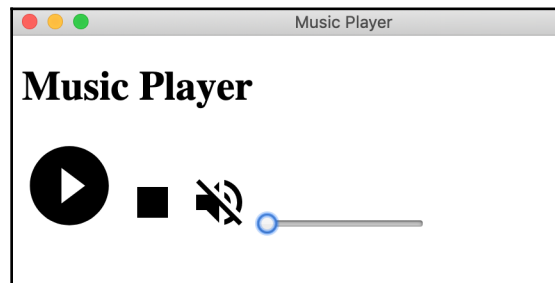
2. For the sake of simplicity, let's comment out the `Volume Up` and `Volume Down` buttons and append the `input range` element, as follows:

```
<div class="controls">
  <div class="amplitude-play-pause"></div>
  <div class="amplitude-stop"></div>
  <div class="amplitude-mute"></div>
  <!-- <div class="amplitude-volume-down"></div> -->
  <!-- <div class="amplitude-volume-up"></div> -->
  <input type="range" class="amplitude-volume-slider" />
</div>
```

3. Restart the application or reload the main window. You should see a range element next to the Mute button. By default, the slider is in the middle, that is, at the 50% level, as shown in the following screenshot:



4. Another useful feature is that other elements react to the player state too. This happens because Amplitude updates all the corresponding CSS classes, and the browser immediately reacts to that. For example, as soon as the range slider reaches zero, the mute button changes its state:



So far, we can start and stop the music, reset its progress, and even control its volume. Now it's time to display the progress of the song to our users. In the next section, you will learn how to declare and use the Progress Bar element, which comes with HTML and is supported by the Amplitude library.

Implementing a song progress bar

One of the traditional music components in any user interface is the progress element. The progress bar component usually displays the current position of the song that we're listening to. In this section, we are going to provide support for the HTML progress element and wire it with the Amplitude library.

In this case, you should be using the `progress` HTML element. It should be in the following format:

```
<progress class="amplitude-song-played-progress"></progress>
```

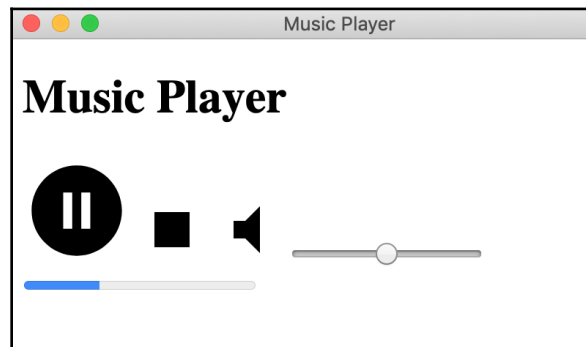
Let's create a separate `div` to hold our `progress` element. For the sake of simplicity, let's leave the style of the element as it is. You can always make it more beautiful later. Follow these steps to do so:

1. Update the `index.html` file according to the following code:

```
<div class="controls">
  <div class="amplitude-play-pause"></div>
  <div class="amplitude-stop"></div>
  <div class="amplitude-mute"></div>
  <!-- <div class="amplitude-volume-down"></div> -->
  <!-- <div class="amplitude-volume-up"></div> -->
  <input type="range" class="amplitude-volume-slider" />
</div>

<div>
  <progress class="amplitude-song-played-progress"></progress>
</div>
```

2. Once again, restart the application or use `Cmd/Ctrl + R` to reload the main window.
3. Start the playback and pay attention to the progress element; it changes its value while the song plays:



We are making excellent progress. Now that we know how to add a song progress bar, it's time to render metadata for our music.

Displaying music metadata

Metadata is a piece of additional information about the song you play. Metadata can contain the name of the album, the year of its release, rating, and many other blocks of useful information.

In this section, we are going to display the following metadata when playing a music file:

- Elapsed time
- Remaining time
- Cover image
- Song name
- Artist name

Before we start, let's learn how to deal with the song metadata. You can extract the values as follows:

```
<span data-amplitude-song-info="<PROPERTY>"></span>
```

`<PROPERTY>` corresponds to one of the metadata properties for the song object that we defined earlier in the `player.js` file:

```
Amplitude.init({
  songs: [
    {
      name: 'Equilibrium I (Cello version)',
      artist: 'David Hilowitz',
      album: 'Equilibrium I (Cello version)',
      url: './music/David_Hilowitz_-_Equilibrium_I_Cello_version.mp3',
      cover_art_url:
        './music/David_Hilowitz_-_Equilibrium_I_Cello_version-_
        _20190327141456457.jpg'
    }
  ]
});
```

This means you can use the following class to display the `name` field in the metadata:

```
<span data-amplitude-song-info="name"></span>
```

Besides custom fields, `Amplitude` provides you with a set of time-related elements.

To display the elapsed time, we are going to use the following CSS class:

```
<span class="amplitude-current-minutes"></span>
```

This gives us a value in minutes. However, for better precision, you may also want to use seconds:

```
<span class="amplitude-current-seconds"></span>
```

Similar to the elapsed time, you can render the duration of the entire song in minutes:

```
<span class="amplitude-duration-minutes"></span>
```

Again, we also need to show seconds using the following format:

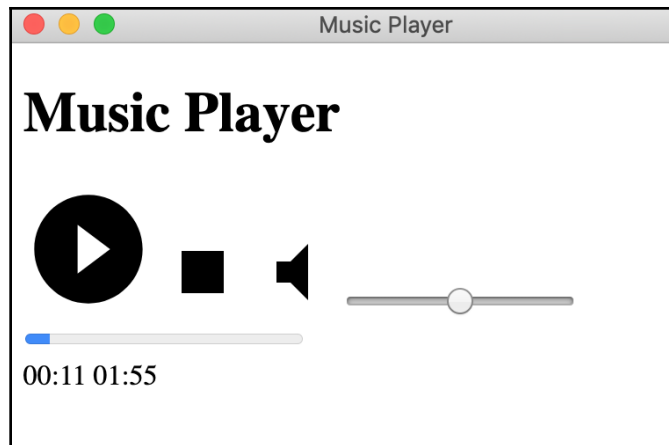
```
<span class="amplitude-duration-seconds"></span>
```

Let's combine all of these elements and display the timing values. Follow these steps to do so:

1. Update the `body` element of your `index.html` page and append the following block of code to it:

```
<div>
  <span class="amplitude-current-minutes"></span>:
  <span class="amplitude-current-seconds"></span>
  <span class="amplitude-duration-minutes"></span>:
  <span class="amplitude-duration-seconds"></span>
</div>
```

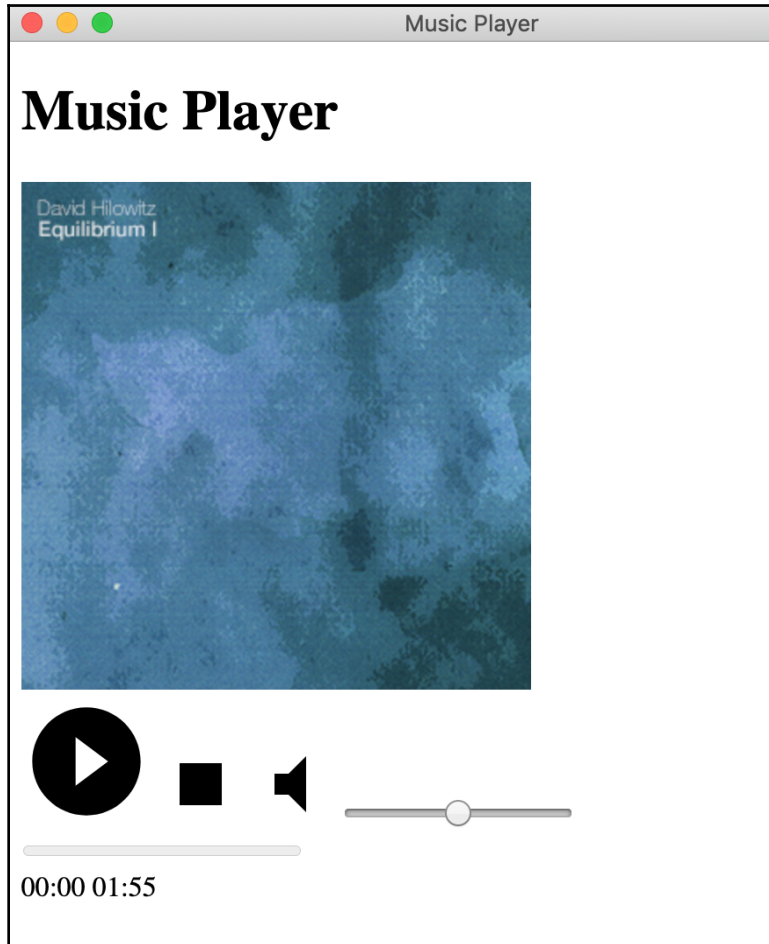
2. Restart the application, start the playback, and check out the timer values. As you can see, we have played 00:11 of the song, which has a total length of 01:55:



3. Next, we need to show the cover art for the album. Amplitude allows you to render the metadata value as the source of the `image` element. All you need to do is declare the `data-` attribute, as shown in the following code:

```
<img data-amplitude-song-info="cover_art_url" />
```

4. As you may recall, we downloaded the cover art picture and declared it in the song configuration file `player.js`. Now, you should see the image at runtime:



5. Finally, let's put the album's name at the top of the cover image:

```
<div>
  <span data-amplitude-song-info="album"></span>
</div>
```

6. The final HTML markup should look as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
    <link rel="stylesheet" href="player.css" />
  </head>
  <body>
    <h1>Music Player</h1>

    <div>
      <span data-amplitude-song-info="album"></span>
    </div>

    <img data-amplitude-song-info="cover_art_url" />

    <div>
      <span data-amplitude-song-info="name"></span>
      by
      <span data-amplitude-song-info="artist"></span>
    </div>

    <div class="controls">
      <div class="amplitude-play-pause"></div>
      <div class="amplitude-stop"></div>
      <div class="amplitude-mute"></div>
      <input type="range" class="amplitude-volume-slider" />
    </div>

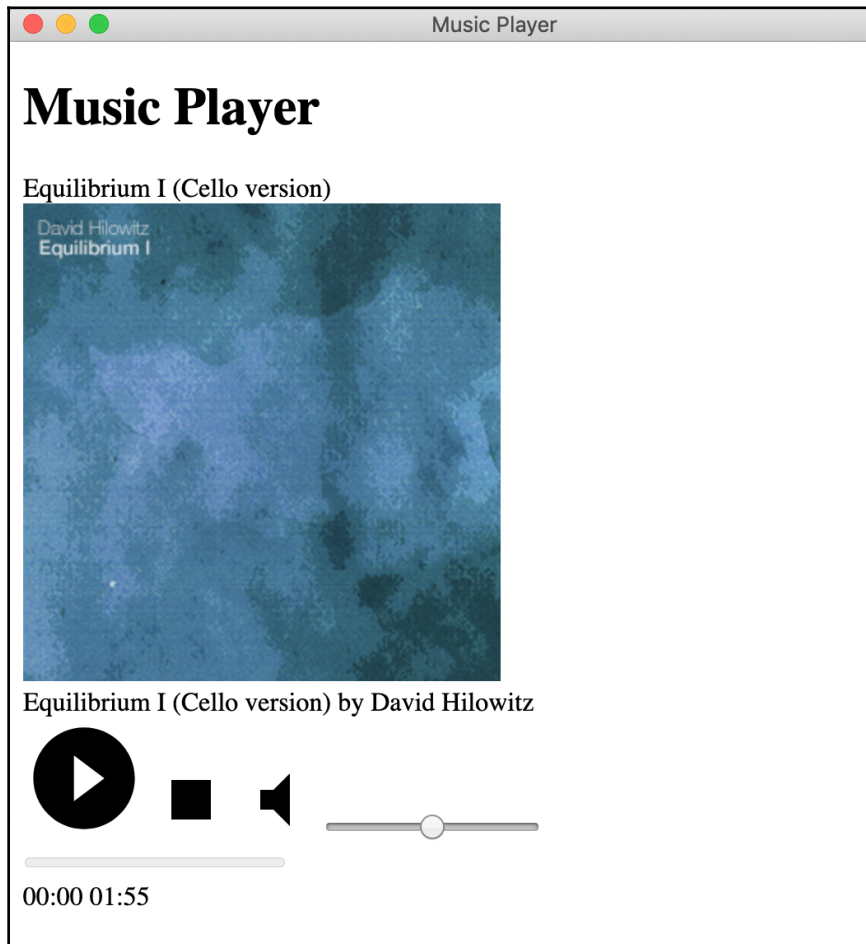
    <div>
      <progress class="amplitude-song-played-progress"></progress>
    </div>

    <div>
      <span class="amplitude-current-minutes"></span>:<span
        class="amplitude-current-seconds"
      ></span>
      <span class="amplitude-duration-minutes"></span>:<span
        class="amplitude-duration-seconds"
      ></span>
```

```
</div>

<script src="./node_modules/amplitudejs/dist/amplitude.js">
</script>
<script src="./player.js"></script>
</body>
</html>
```

7. Our minimalistic music player application now appears as follows:



We have made brilliant progress, and as you can imagine, there are many more features we can add to the application at this point. However, let's not forget about the user interface and polish our application a bit.

Improving the user interface

At this point, you have a draft implementation of the user interface, along with a set of playback control and visualization components. Now is a perfect time to take a break and revisit the look and feel of our application.

In this section, we are going to polish the interface and make it more appealing to our users.

First of all, let's change the default size of the player. You may have already noticed that, in its current form, it takes up less space than the size of the window. Follow these steps to get started:

1. Switch to the `main.js` file and set its size to 480x500. This should be enough for now. You can use the following code for reference:

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

function createWindow() {
  const win = new BrowserWindow({
    width: 480,
    height: 500,
    webPreferences: {
      nodeIntegration: true
    },
    resizable: false
  });

  win.loadFile('index.html');
}

app.on('ready', createWindow);
```

2. Next, remove the `h1` element from the **Music Player** label as you no longer it. Now, we need to tune the cover image.
3. Declare a new `cover-image` class in the `player.css` stylesheet and use the following set of rules to make the image fit the available space nicely:

```
.cover-image {
  object-fit: contain;
  width: 100%;
  height: 100%;
  max-height: 300px;
}
```

The style isn't going to work until you update the `index.html` file and assign it to the `img` element, which holds the album cover.

4. Add the `cover-image` class, as shown in the following code:

```
<img class="cover-image" data-amplitude-song-info="cover_art_url"
/>
```

5. Next, you should wrap up all the time-related elements in the `div` element using the `song-progress-container` CSS class. Update the code so that it looks as follows:

```
<div class="song-progress-container">
  <div>
    <span class="amplitude-current-minutes"></span>:<span
      class="amplitude-current-seconds"
    ></span>
  </div>

  <progress class="amplitude-song-played-progress"></progress>

  <div>
    <span class="amplitude-duration-minutes"></span>:<span
      class="amplitude-duration-seconds"
    ></span>
  </div>
</div>
```

6. Append the following styles to the `player.css` file:

```
div.song-progress-container {
  display: grid;
  grid-template-columns: 1fr 10fr 1fr;
}

progress.amplitude-song-played-progress {
  width: 100%;
}
```

7. Now, gather all the metadata fields inside the `div` contain with the `song-info-container` class. This allows us to apply styles to all the fields within it:

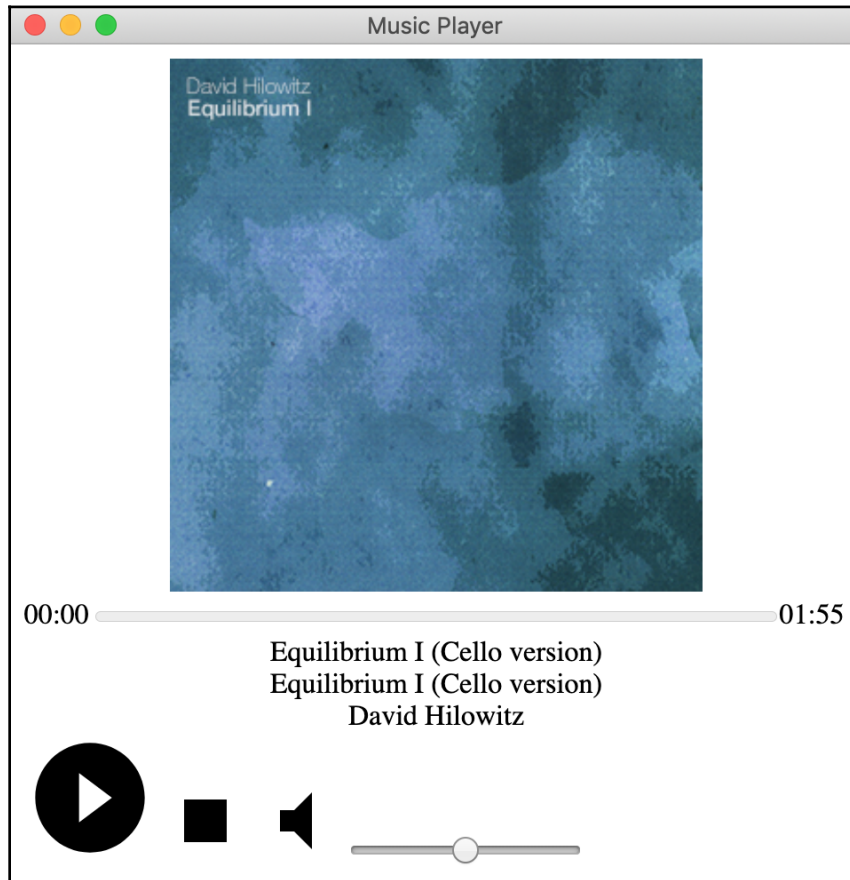
```
<div class="song-info-container">
  <div data-amplitude-song-info="name"></div>
  <div data-amplitude-song-info="album"></div>
  <div data-amplitude-song-info="artist"></div>
</div>
```

8. Append the `song-info-container` style implementation to the `player.css` file.

9. For now, we only need to center the text horizontally:

```
.song-info-container {  
  text-align: center;  
}
```

10. Now, our lovely music player application looks like this:



Now we can focus on the features we've implemented and add much more functionality. Don't forget to read the AmplitudeJS documentation and check out all available CSS elements: <https://521dimensions.com/open-source/amplitudejs/docs>.

In the next section, we are going to review the final structure of our application.

Reviewing the final structure

Now, we have a nice-looking music player application based on Electron and the Amplitude library. In this section, we are going to review the final state of the code and verify that we have done everything correctly.

Please refer to the following `index.html` file in case your application's layout looks different or you think you may have missed a certain step:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Music Player</title>
    <link rel="stylesheet" href="player.css" />
  </head>
  <body>
    <img class="cover-image" data-amplitude-song-info="cover_art_url" />

    <div class="song-progress-container">
      <div>
        <span class="amplitude-current-minutes"></span>:<span
          class="amplitude-current-seconds"
        ></span>
      </div>

      <progress class="amplitude-song-played-progress"></progress>

      <div>
        <span class="amplitude-duration-minutes"></span>:<span
          class="amplitude-duration-seconds"
        ></span>
      </div>
    </div>

    <div class="song-info-container">
      <div data-amplitude-song-info="name"></div>
      <div data-amplitude-song-info="album"></div>
      <div data-amplitude-song-info="artist"></div>
    </div>

    <div class="controls">
      <div class="amplitude-play-pause"></div>
      <div class="amplitude-stop"></div>
      <div class="amplitude-mute"></div>
      <input type="range" class="amplitude-volume-slider" />
    </div>
```

```
<script src="./node_modules/amplitudejs/dist/amplitude.js"></script>
<script src="./player.js"></script>
</body>
</html>
```

The full CSS class implementation in the `player.css` file should look similar to the following:

```
.cover-image {
  object-fit: contain;
  width: 100%;
  height: 100%;
  max-height: 300px;
}

div.song-progress-container {
  display: grid;
  grid-template-columns: 1fr 10fr 1fr;
  grid-gap: 10px;
}

progress.amplitude-song-played-progress {
  width: 100%;
}

.song-info-container {
  text-align: center;
}

.controls > div {
  width: 48px;
  height: 48px;
  cursor: pointer;
  display: inline-block;
}

.controls .amplitude-play-pause {
  width: 74px;
  height: 74px;
  cursor: pointer;
  display: inline-block;
}

.controls .amplitude-play-pause.amplitude-paused {
  background: url('./images/baseline-play_circle_filled-24px.svg');
  background-size: cover;
}
```

```
.controls .amplitude-play-pause.amplitude-playing {
  background: url('../images/baseline-pause_circle_filled-24px.svg');
  background-size: cover;
}

.controls .amplitude-stop {
  background: url('../images/baseline-stop-24px.svg');
  background-size: cover;
}

.controls .amplitude-mute {
  background: url('../images/baseline-volume_mute-24px.svg');
  background-size: cover;
}

.controls .amplitude-mute.amplitude-muted {
  background: url('../images/baseline-volume_off-24px.svg');
  background-size: cover;
}

.controls .amplitude-volume-up {
  background: url('../images/baseline-volume_up-24px.svg');
  background-size: cover;
}

.controls .amplitude-volume-down {
  background: url('../images/baseline-volume_down-24px.svg');
  background-size: cover;
}
```

As you can see, we didn't need to use too much code to make the player work. It's mainly just a few HTML elements and a set of CSS styles so that we can wire them with Amplitude or make them look better.

The application may need more work to be done to it so that it's truly useful as a player. Please feel free to provide support so that you can switch between different songs, support playlists, fetch album covers from online sources, and much more.

Summary

In this chapter, we looked at building a minimalistic cross-platform music player with the help of the Electron framework and the AmplitudeJS library.

You have acquired the skills you need in order to build music applications, display metadata, and handle cover album art. You also know how to style web components so that they look like music playback buttons.

We have successfully met our goal; now you know how to set up a new Electron application with multimedia support that's backed by third-party libraries. You can also create user interfaces based on CSS class names and allow external libraries to turn HTML elements into music playback or visualization components.

We also used a third-party library to play the sounds files and build a user interface without much effort. Then, we created a song entry with cover album art and metadata. We successfully rendered the image and song information onto the screen.

We hope this chapter has given you many ideas on how you can extend your Electron-based music player even further.

Another essential part of every desktop application life cycle is analytics, especially when it comes to bug tracking and user feedback. In the next chapter, we are going to learn how to integrate real-time analytics into our Electron applications.

7

Analytics, Bug Tracking, and Licensing

This is a pretty big list of features, so let's get started with deciding on whether we need. This chapter provides essential information for developers who want to monitor Electron applications in production, track errors and crashes, analyze a real-time user base, and much more. In this chapter, you are going to walk through the processes of integrating with third-party analytics services, raising custom events, and wiring your Electron applications with license checks. We are also going to send notifications to installed copies of our application across all major desktop platforms.

By the end of this chapter, you will have an Electron project with tracking support integrated. You will also be able to generate some statistics and tracking information for demonstration purposes. As part of the practical exercise in this chapter, you are going to integrate with the third party service and get the Nucleus service subscription, which is free for the first month, so that you can collect and inspect the analytics data. The estimated project build time is three hours.

In this chapter, we will cover the following topics:

- Understanding analytics and tracking
- Creating your own solution or using an existing service
- Using Nucleus for Electron applications
- Creating a new Nucleus account
- Creating a new project with tracking support
- Installing the Nucleus Electron library
- Inspecting real-time analytics data
- Disabling tracking per user request
- Verifying real-time user statistics
- Supporting offline mode
- Handling application updates

- Loading global server settings
- License checking and policies

This is a pretty big list of features, so let's get started with deciding on whether we need analytics and whether we need to build all the features ourselves.

Technical requirements

To get started with this chapter, you will need a standard laptop or desktop running macOS, Windows, or Linux.

The software that you'll need to have installed for this chapter is as follows:

- Git, a version control system
- Node.js with NPM
- Visual Studio Code, a free and open source code editor

You can find the code files for this chapter in this book's GitHub repository <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter07>.

Understanding analytics and tracking

As soon as you release the first version of your Electron-based application into production, you may need to gather some statistics regarding how your customers use the application. Then, you can use that information to improve the product and boost the downloads or sales of your application if it's commercial. Some examples of how we can use this information are as follows:

- **Improving distribution and sales:** Imagine that, according to analytics, most of your user base is coming from the macOS world. You may want to provide more macOS integration or focus on strengthening support for other platforms, for example. The same goes for geographical locations. You may see the need to improve advertising for a particular set of countries that lack downloads or purchases.

- **Improving product quality:** The critical aspect of any successful application release is having a bug tracking system. Bugs sometimes happen, and it's not always easy for our end users to raise issues or provide all the necessary technical details. Upon releasing a new version of your application to the public, you definitely want to see error details, as well as the number of systems that have been affected by any errors that have occurred.
- **License management:** Tracking application licenses is another crucial part of your product life cycle. You may want to gather statistics on how many licenses have been sold alongside the analytical statistics. This helps when you are selling your application or you have different feature tiers or add-ons that users can purchase separately.

Next, we are going to discuss whether we should start building our homegrown analytics and tracking solution or use existing services. In addition to this, you will learn how to use **Nucleus** (<https://nucleus.sh/>) in order to integrate analytics, bug tracking, and license management for your Electron applications with little effort.

Creating your own solution or using an existing service

You should start thinking about using analytics and bug tracking in the early stages of development so that they fit into your code and application architecture nicely.

In most cases, it's just about calling a particular function or API to inform your server about an event; for example, *application loaded*, *authentication failed*, or something crashed in the code.

The main thing to consider is whether you need to build analytics features yourself or whether there are existing solutions that can boost you in the early stages.

Let's take a look at the pros and cons of the following:

- Creating and using your own analytics services
- Using existing services that have been provided by third parties

Creating your own analytics services

In the early stages of application development, you may be tempted to build a complete analytics service from scratch. Note, however, that this is a time-consuming process that requires a lot of effort.

The following are the pros of creating your own analytics services:

- Full control over the data
- Control over the API and the services
- Backend server ownership

The following are the cons of creating your own analytics services:

- Having to maintain the databases
- Having to secure the data yourself
- Having to maintain the backend servers
- Having to support online availability

As you can see, owning the complete solution comes with considerable responsibility in terms of storing and securing data, as well as investing in the hardware and hosting services for the backend servers.

Now, let's take a look at the other side of the coin.

Using third-party analytics services

There are already a lot of different analytics services you can reuse for a fair subscription price. Let's go over some of the pros and cons.

The following are the pros of using third-party analytics services:

- No need to maintain the hardware
- Databases are maintained by the vendor
- Security is maintained by the vendor
- Online availability and scaling

The following are the cons of using third-party analytics services:

- Your data is stored externally
- Your application relies on the availability of the vendor services

- Reports and feature sets may be limited
- Pricing may change over time

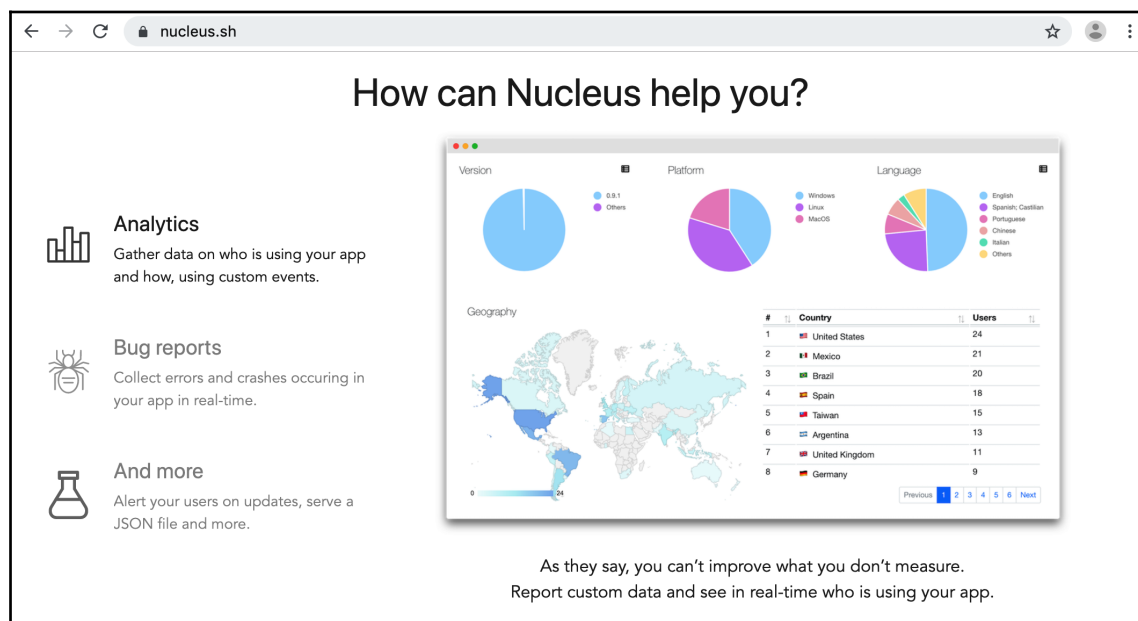
With third-party analytics, you save time, effort, and money in the early stages. This allows you to move fast and focus on the business side of your application.

As soon as your application installation base starts growing, you can always switch to a homegrown solution with dedicated support and a good development team. Meanwhile, however, I strongly recommend getting started with a third-party service.

Now that we have seen all the pros and cons of third-party services, let's learn how to use one such service. In the next section, we are going to integrate Nucleus into our Electron application.

Using Nucleus for Electron applications

Nucleus is an analytics platform for Electron applications. You can use it to install an extra library and emit tracking events from your application code. Nucleus aggregates information from all the client instances, processes the data, and stores it in its own servers. By doing this, you can access reports, view statistics, and even send notifications to the running clients, as shown in the following screenshot:



There is a set of browser characteristics that an application can collect and send to the analytics services. It can also be based on the information that doesn't identify a person, but this still helps us to improve our projects. According to the official documentation, the service collects the following information from each client:

- Time of the requests
- The hashed identifier of the machine (cannot be used to identify the user outside of Nucleus' context)
- Browser locale (language)
- The country where the request was made (from Cloudflare)
- Operating system family (Mac, Windows, or Linux)
- OS version
- Your app's version
- Nucleus module version
- RAM that's available on the device

The following information is not collected:

- IP addresses
- The user's Chromium browser user agent
- The city or region of the request
- Screen resolution



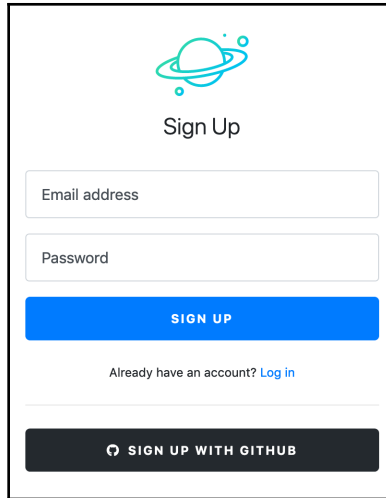
Check out the official transparency report to view the changes and updates that are constantly made to the list of tracking rules: <https://nucleus.sh/transparency>.

Nucleus is a subscription-based project. You need to have an account to use its features. Luckily, you can start a trial period for testing purposes. Let's learn how to create an account and start the 30-day trial.

Creating a new Nucleus account

In this section, we are going to walk through the account registration process. Follow these steps:

1. Click on the **Sign Up** button on the main page or navigate to <https://nucleus.sh/signup>:

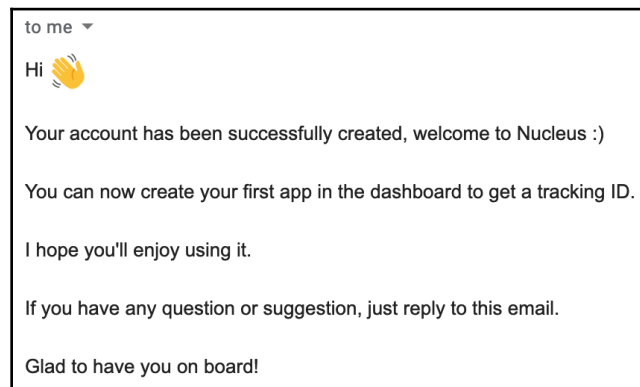
A screenshot of a web form titled "Sign Up" with a teal planet logo. It contains two input fields: "Email address" and "Password". Below these is a blue "SIGN UP" button. Under the button is a link: "Already have an account? [Log in](#)". At the bottom is a dark grey button with the GitHub logo and the text "SIGN UP WITH GITHUB".

Here, you have two options. You can either provide an email address and password to create a new account or sign up with your existing GitHub account.



GitHub authentication works similar to other popular social sign-in providers you may have come across on numerous websites, including Facebook, Twitter, and Google. With GitHub authentication, the system will use the email address you have associated with your GitHub account. You won't need a password as your authenticated GitHub session will be your proof of identity.

2. Select one of the available options and register a new account. You should receive a welcome email shortly after registration. In the following screenshot, you can see my Gmail confirmation letter:



3. Proceed to the **Sign In** screen. Here, you are going to see a list of available subscription plans. Select a plan that suits your needs. Each plan has a 30-day trial period, and I recommend picking the **Hobby** plan for now. You can always cancel it or continue using the subscription if you like the service. The available plans can be seen in the following screenshot:

The screenshot displays a 'Select a plan' interface with three columns representing different subscription tiers. The 'Hobby' plan is highlighted with a blue border. Each plan includes a price, a 30-day free trial, and a list of features. At the bottom, there is a billing section with a card number input field, a 'Continue to Hobby' button, and a note about secure storage with Stripe.

Plan	Price	Trial	Users	Bug Reports	Apps	Retention
Hobby	\$19 / per month	30 days free trial	200 users per day	5000 monthly bug reports	Track up to 3 apps	1-year data retention
Startup	\$49 / per month	30 days free trial	1000 users per day	Unlimited bug reports	Real-time map	Track up to 10 apps
Business	\$249 / per month	30 days free trial	Unlimited users	Unlimited bug reports	Track unlimited apps	5-year data retention

Billing will start at the end of the trial period if you haven't canceled.

Card number MM / YY CVC

Continue to Hobby →

Card stored securely with [Stripe](#)

4. Upon logging in, you should see an **Account** page and a dialog suggesting that you create your first application. You need an application so that you can get the tracking ID that's unique to your application.
5. The form is pretty minimal, so let's fill it in to register for a new application. Use the default application details shown in the following screenshot and click **Create**:

The screenshot shows the 'Account' section of the Nucleus interface. It features a form to create a new app with fields for 'App name' (containing 'My App') and 'App version' (containing '0.0.1'). A blue 'Create' button is positioned to the right of the version field. Below the form, a white box titled 'Create your first app' contains instructions: 'First, let's create an app and enter the current version so we can obtain a tracking ID.' and a 'Close' button.

- Now, you will find yourself on your application's analytics page. There's no data here yet. For now, you are provided with some instructions regarding how to configure an Electron application with your tracking ID number:

The screenshot displays the Nucleus Analytics dashboard. On the left is a teal sidebar with the Nucleus logo and a menu including 'Analytics' (selected), 'Users', 'Events list' (marked 'new!'), 'Live view' (marked 'new!'), 'Errors', 'Other', and 'Account'. The main content area has a light gray background and contains the following text:

There is nothing to show yet.

Start by installing and integrating the [library](#) in your app.

This application ID is : **5d55bc992fcb6700d70b1eff**

```
> npm install --save electron-nucleus
```

```
// Import the Nucleus Library and init with your app id
const Nucleus = require('electron-nucleus')('5d55bc992fcb6700d70b1eff')

// Optional: report an event
Nucleus.track('BUTTON_CLICKED')
```

At the bottom, a link says: 'Need help or something is missing? → hello@nucleus.sh'

Now, you have successfully created an account with Nucleus and have a 30-day trial period to get familiar with its features and decide whether you want to continue with the subscription or focus on your own solutions.

In Nucleus, analytics is based on the concept of projects. A project is essentially a single application that sends the analytics data. You can have multiple projects per account.

Now, it's time to create a new Electron project with tracking support.

Creating a new project with tracking support

Let's set up a new project called `analytics-tracking`, as follows:

1. Create a new folder so that you can store all of the project files:

```
mkdir analytics-tracking
cd analytics-tracking
```

2. Perform a quick setup of the repository using the following commands:

```
npm init -y
echo node_modules > .gitignore
npm i -D electron
```

3. Drop an `index.html` file into the project root along with a primary template, as shown in the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
    <title>Electron Analytics</title>
  </head>
  <body>
  </body>
</html>
```

4. Create a `main.js` file in the project root with a minimal set of instructions regarding how to create a new Electron window:

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

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
```

```
        height: 600,  
        webPreferences: {  
            nodeIntegration: true  
        },  
        resizable: false  
    });  
  
    win.loadFile('index.html');  
}  
  
app.on('ready', createWindow);
```



For the sake of simplicity, we're creating a non-resizable window that's 800x600 pixels in size with Node.js integration enabled by default. Feel free to add any other settings to the code if you need them.

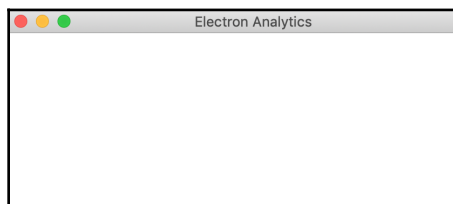
5. Update the `package.json` file:

```
{  
  "name": "analytics-tracking",  
  "version": "1.0.0",  
  "description": "",  
  "main": "main.js",  
  "scripts": {  
    "start": "electron ."  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "electron": "^7.1.1"  
  }  
}
```

6. At this point, you have a barebones setup for a new Electron project so that you're ready to perform analytics and tracking integration. You can check that everything runs as expected using the following command:

```
npm start
```

7. You are going to see a blank Electron window entitled **Electron Analytics**, as shown in the following screenshot:



That is the expected behavior so far.

Now, you have a basic application project that you can use for analytics testing. You can copy the configuration and use it as a template for future experiments. For now, let's move on and integrate the Nucleus library into this project.

Installing the Nucleus Electron library

Configuring Nucleus support is easy since all the required APIs are published as a single Node.js library. Follow these steps to install the library:

1. In the root folder of your project, run the following command to install the `electron-nucleus` library from NPM:
2. To integrate the library, you need to have the tracking ID number that you retrieved from the Nucleus website earlier. All you need to do to get your tracking ID number is run the following code somewhere in the `main.js` file:

```
npm i electron-nucleus
```

```
const Nucleus = require('electron-nucleus')('Your App ID', {  
  onlyMainProcess: true  
});
```

You can also raise a custom tracking event, as follows:

```
Nucleus.track(<NAME>, <DATA>);
```

`NAME` is the name of your event and can be anything that gives you meaning when you're inspecting analytics. `DATA` is an optional JSON object that you can use to pass details about the event; this can be error details or tracking details, for example.

3. Update the `main.js` file according to the following code:

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

const Nucleus = require('electron-nucleus')('Your App ID', {
  onlyMainProcess: true
});

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    },
    resizable: false
  });

  win.loadFile('index.html');

  // Optional: report an event
  Nucleus.track('APP_LAUNCHED');
}

app.on('ready', createWindow);
```



Don't forget to use your tracking ID instead of the one shown in this example. You can find your tracking ID value online at any time.

Note that, besides Nucleus library initialization, we can also raise an `APP_LAUNCHED` event within the `createWindow` function. This event is invoked every time the application starts and is a simple example of how we can use custom events in our applications. We are going to see how this works shortly.

4. Run the following command to test your application:

```
npm start
```

Nothing extra should happen from a visual standpoint. You should still see the blank Electron window with the **Electron Analytics** title. This time, however, your application should send tracking data to the Nucleus server.

At this point, you have an Electron application that has the Nucleus library installed. Each time the application starts, the Nucleus library sends a notification to the server. We also send an extra event called `APP_LAUNCHED` so that we can view the default and custom events in action.

Leave the application running for now. In the next section, we are going to see what the analytics data looks like in real life.

Inspecting real-time analytics data

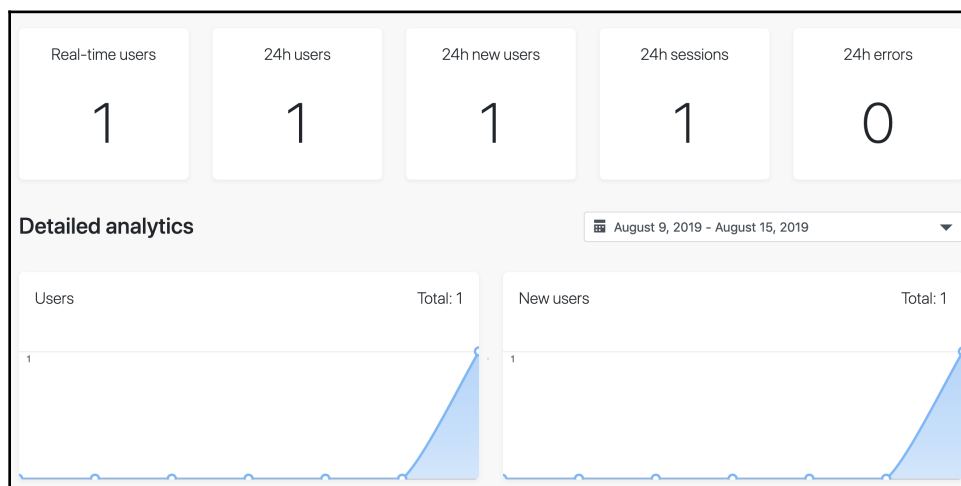
Your Electron application is up and running, so now is an excellent time to see real-time analytics in action. Let's get started:

1. Navigate to your Nucleus account and select the application you created earlier.



If you have only one application, it is going to be automatically displayed next time you log in.

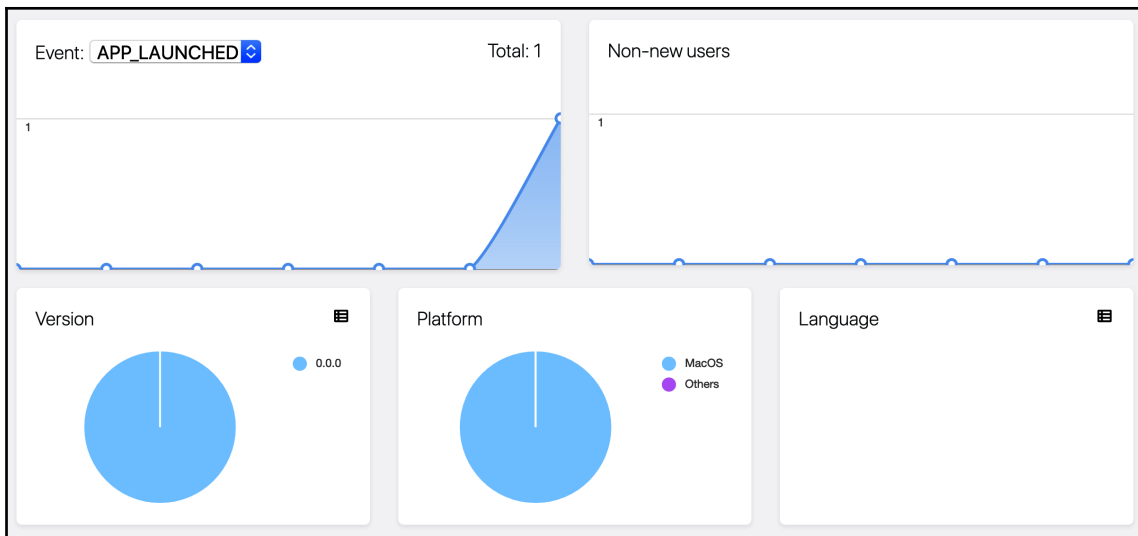
2. The first thing you are going to see is the real-time user statistics, as shown in the following screenshot:



As you can see, the service reflects our running application and displays one real-time user in the chart.

This analysis provides the following information:

- The number of real-time users
 - The number of users that used your application in the last 24 hours
 - The number of new users that used your application in the last 24 hours
 - The number of sessions that were opened in the last 24 hours (the users launched your application multiple times)
 - The number of errors the applications raised in the last 24 hours
3. Next comes the detailed analytics. You will be able to beautiful graphs of existing users compared to new users over time. This service allows you to select different time ranges as well.
 4. Below this detailed analytics information, you will be able to see the events our application instances raise:



As you can see, the service already contains data about the `APP_LAUNCHED` event we raised in our code. Here, we can check all kinds of events, as well as the graph of calls over time.

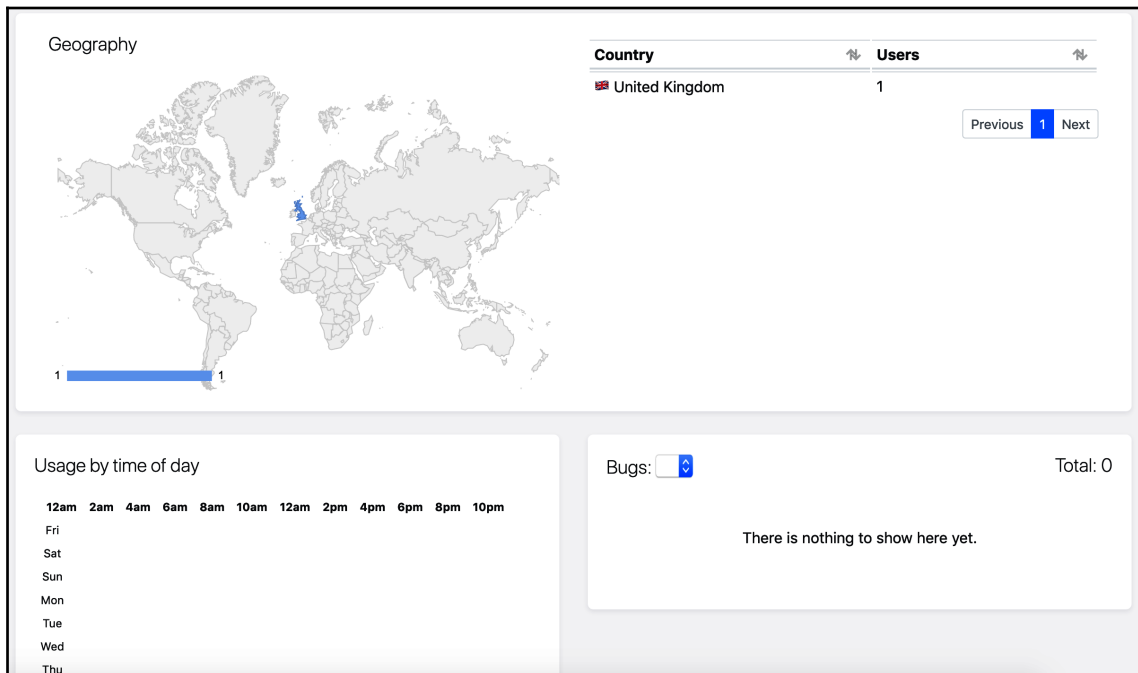
Other important information you can inspect here is as follows:

- The version of your application (always 0.0.0 when you're running in debug mode; this can also be detected from your application properties).
- The platform. As you can see, I'm using macOS to run the demo.

- The system languages that are being used by the users of our application (this helps us detect whether we need to localize the user interface based on our audience).
5. Note that you can also provide custom values for the application's version and language. You should do this in case the Nucleus library fails to auto-detect them or if you want to use a custom mechanism to detect and track that data. You may want to pass these custom options when you create a new instance of Nucleus in the `main.js` file:

```
const Nucleus = require("electron-nucleus")("<App Id>", {  
  version: '1.0.0',  
  language: 'en'  
});
```

6. Other valuable indicators are geographical location and usage in terms of the time of the day:

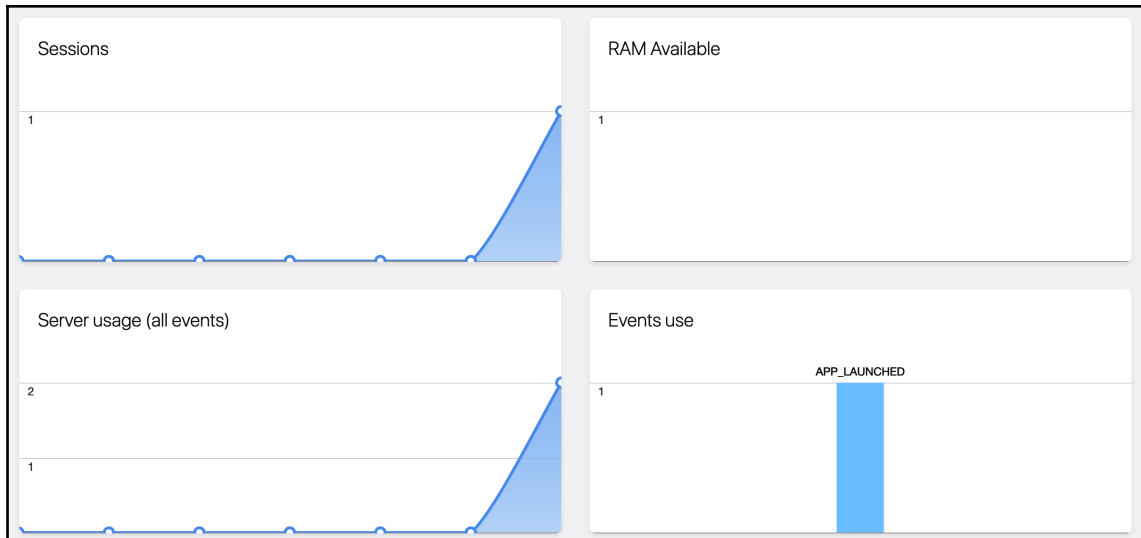


Nucleus provides a nice-looking chart showing all the countries of origin for the users of your application. This provides a good overview of your user base, as well as insight into where your application is popular.

7. If you scroll a little bit further, you should see the following sets of charts:

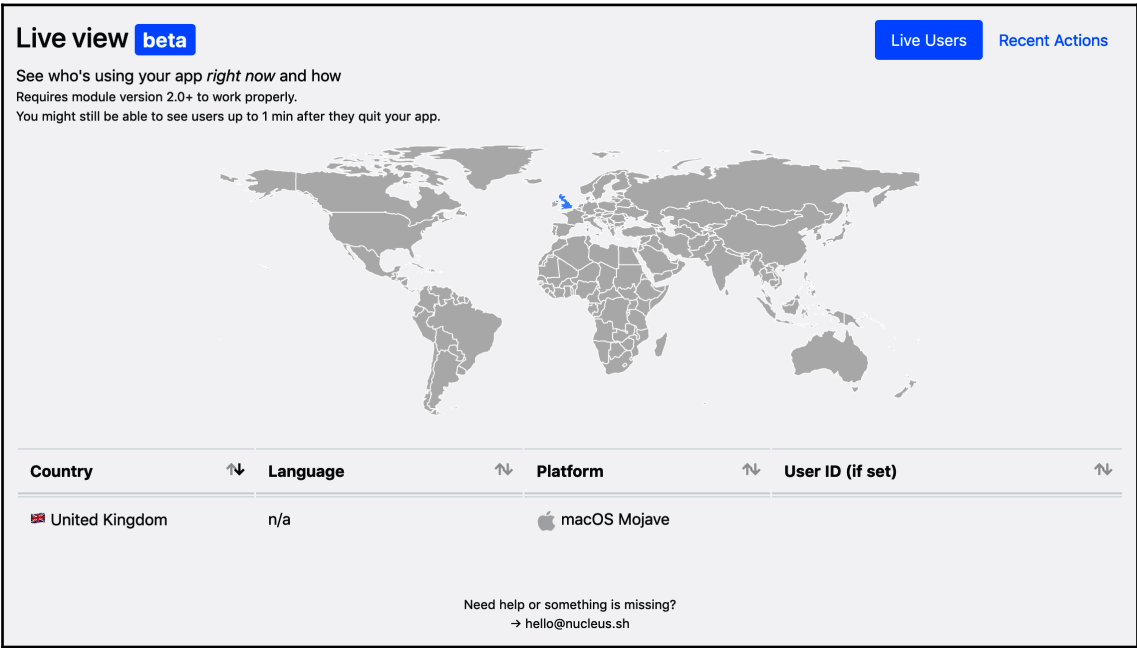
- **Sessions**
- **RAM Available**
- **Server usage (all events)**
- **Events use**

These can be seen in the following screenshot:



You may see all of these in use at some point, especially **RAM Available**, since this helps us detect memory leaks or a lack of RAM on our user's machines.

8. In the middle of the left sidebar, there's also a beta version of the **Live View** chart, as shown in the following screenshot:



This chart shows you the users who are actively using your application on a geographical map, alongside information about them, including the following:

- **Country**
- **Language**
- **Platform**
- **User ID**

As you can see, right now, there's a single user from those **United Kingdom** using the application. The application is running on the **macOS Mojave** operating system, and no dedicated user ID has been provided.

Identifying users

The Nucleus service allows us to identify our users if we need that level of tracking. When working with analytics data, you may want to know the user's or machine's information.

You can do this when you create a new instance of the Nucleus object. Check out the following example:

```
const Nucleus = require("electron-nucleus")("<App Id>", {
  userId: '<unique-identifier>'
});
```

In the preceding code, `<unique-identifier>` is any value that you can generate on the first application run, such as the GUID. Alternatively, you can have the user provide it, in which case you'll be provided with something such as a login or email address.

You can also specify the user ID at runtime if you don't know the value beforehand. In this case, we would use the following code:

```
Nucleus.setUserId('<unique-identifier>')
```

As you have seen, you have access to a wide range of very useful charts and analysis capabilities. All of this rich visualization may help you make the right decisions regarding how you evolve and maintain your project.

However, there are cases where you will need to allow users to opt out of tracking. Let's take a look at how we can toggle tracking features programmatically.

Disabling tracking per user request

Many countries have laws that require you to get a user's consent before you enable tracking.

You may also want to provide some dialog on the first run that asks users whether they want to provide anonymous feedback to help you improve the service. If the user rejects this feedback option, you should probably disable Nucleus integration at the application level.

The Nucleus library provides specific APIs that allow developers to switch tracking features off and on. If the user explicitly states (via the user interface) that anonymous feedback should be disabled, use the following code:

```
Nucleus.disableTracking()
```

You can save the user's decision somewhere in the configuration file as a flag and run the code on each application startup.

Depending on the scenario, on each application upgrade, you may also want to ask users whether they wish to enable tracking or keep it disabled. If the user decides to enable automatic feedback, you can use the following function to enable Nucleus integration once more:

```
Nucleus.enableTracking()
```

Alternatively, you can set the `disableTracking` flag when creating a new instance of the Nucleus object at application startup:

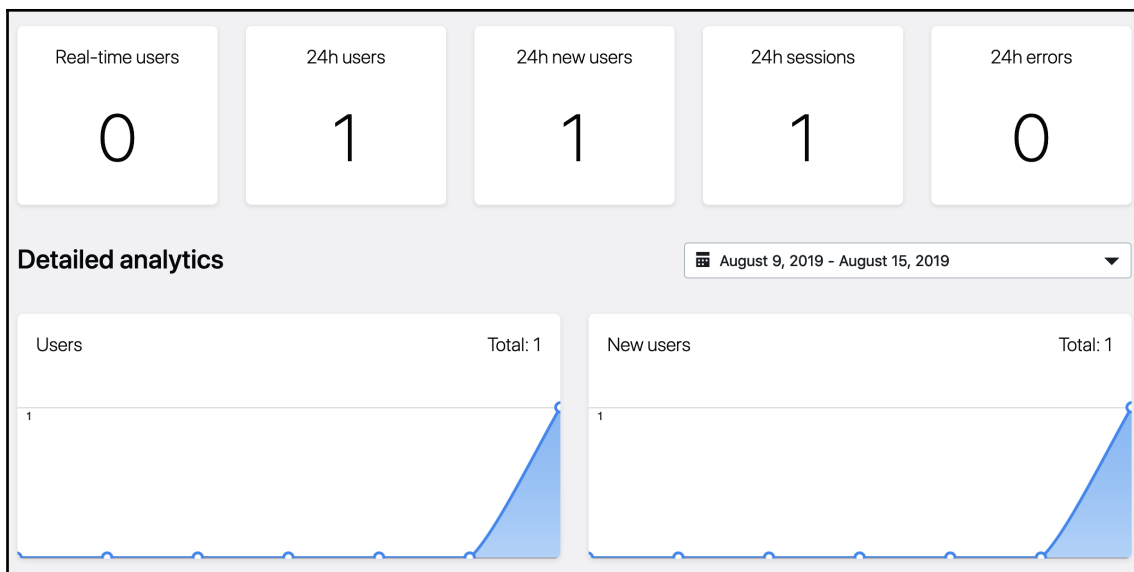
```
const Nucleus = require("electron-nucleus")("<App Id>", {  
  disableTracking: false  
});
```

We still have our Electron application running in the background. Let's see what happens to the analytics data when we shut the application down.

Verifying real-time user statistics

In this section, we are going to verify that our real-time user statistics are getting updated on the fly. Follow these steps to do so:

1. Shut down your Electron application and wait for about a minute. It may take some time for these changes to be propagated. Sometimes, you need to wait for a minute, but sometimes waiting a few seconds will be enough. Be patient and give the backend server time to organize the data.
2. Switch back to the **Analytics** tab and reload the page. You should see the updated data. In our case, the **Real-time users** chart shows zero, as shown in the following screenshot:



This makes perfect sense as we have just unloaded the application. Note, however, that our session is still present in other charts:

- **24h users: 1**
- **24h new users: 1**
- **24h sessions: 1**

Now, you can track real-time usage of your Electron application. However, there may be cases when your users are offline or have no internet connectivity. In the next section, we are going to provide support for offline mode.

Supporting offline mode

By default, the Nucleus library expects your users to have an internet connection to send tracking events. However, you may be wondering what happens if users run your application when they're offline, for example, during a flight or when they're on the tube.

The good news is that you can turn on offline support for Nucleus-enabled Electron applications. This allows us to store the application events locally, cached to disk, and send them to the analytics server once the application is back online.

Use the `persist` property to enable or disable caching for events:

```
const Nucleus = require("electron-nucleus")("<App Id>", {
  persist: true
});
```

As you can see, enabling offline mode support for your application's analytics is not difficult with the Nucleus library.

Next, let's learn how to handle application updates in a centralized manner.

Handling application updates


At this point, you should have at least one application with a tracking ID in your Nucleus account.

In this chapter, you have an app called **My App** whose current version is **0.0.1**, as shown in the following screenshot:

Account

Create apps here to obtain tracking IDs.

Create new app

Name	Tracking ID	Current version	Actions
 My App	5d55bc992fcb6700d70b1eff	0.0.1	Edit Delete

Your application can track version changes and execute certain pieces of code each time an update happens.

The API is in the following format:

```
Nucleus.onUpdate = version => {
  // do something with the version
}
```

For the sake of simplicity, let's raise a standard JavaScript alert message on each version update:

1. Update the `main.js` file according to the following code:

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

const Nucleus = require('electron-nucleus')('Your App ID', {
  onlyMainProcess: true,
  version: '0.0.1'
});

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    },
    resizable: false
  });

  win.loadFile('index.html');

  // Optional: report an event
  Nucleus.track('APP_LAUNCHED');

  Nucleus.onUpdate = version => {
    win.webContents.executeJavaScript(`
      alert('There is a new version available: ${version}');
    `);
  };

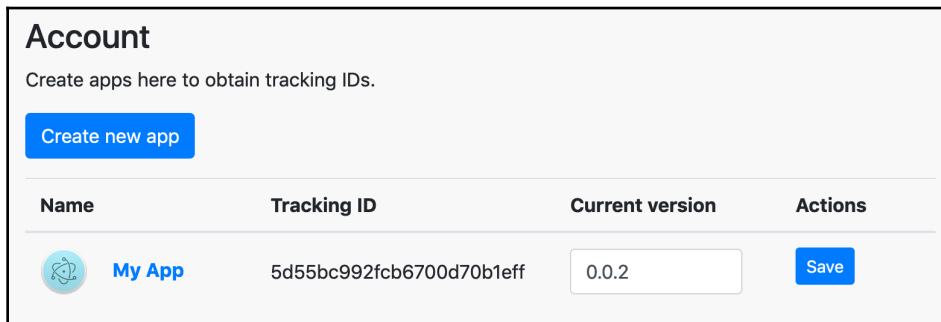
  app.on('ready', createWindow);
```



Note that we have provided an explicit version of the application this time.


2. Save these changes and launch the application using the `npm start` command.
3. Wait a few seconds before navigating to **Analytics** and reloading the page. You should see a counter increase in the **Real-time users** chart. If you don't see these changes, wait a little bit longer and reload the page from time to time.
4. Go to **Account** and click the **Edit** button for the **My App** entry.

5. Change the version value to 0.0.2, as shown in the following screenshot:



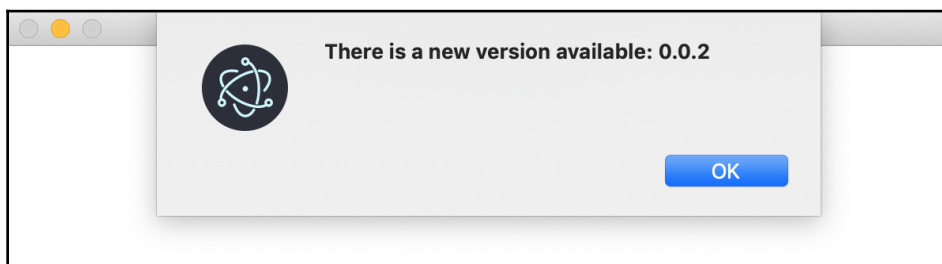
Account
Create apps here to obtain tracking IDs.

Create new app

Name	Tracking ID	Current version	Actions
 My App	5d55bc992fcb6700d70b1eff	<input type="text" value="0.0.2"/>	Save

As you can see, we can only change the version field. The form allows you to notify every client application that a new version of the application has been registered.

6. Click the **Save** button and switch back to your Electron application.
7. You should see a simple dialog saying that there is a new version available:

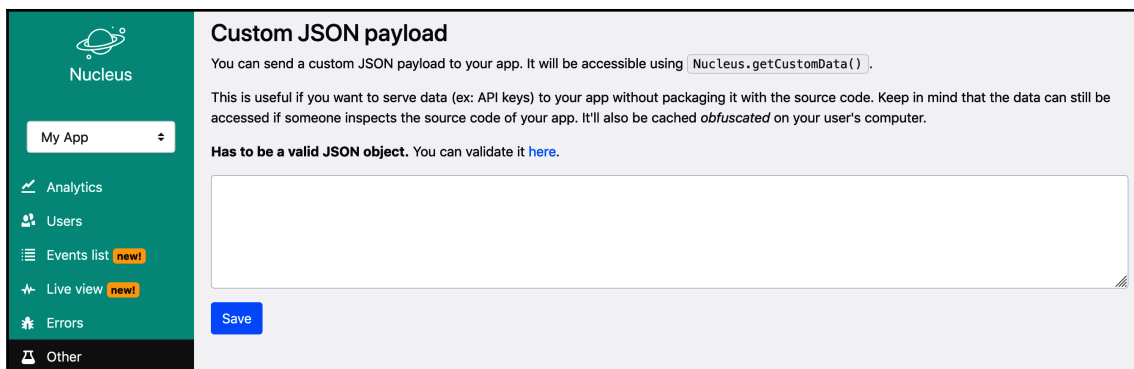


Note that you can build a more sophisticated user interface with extra buttons to take a look at the release notes, for example, or navigate to your website.

In the next section, we are going to learn how to introduce global server settings.

Loading global server settings

Another exciting feature of the Nucleus service I would like to talk about is **Custom JSON payload**. You can find it in the **Other** section of your Nucleus account:



From here, you can create a JSON document on the server side that each of your Electron application instances can fetch upon startup and use for configuration or business logic purposes.

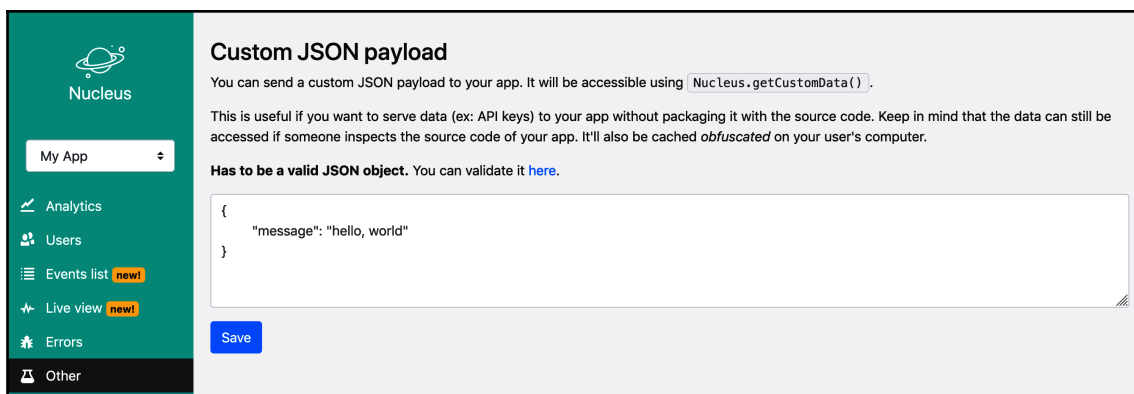
Imagine, for instance, having global settings that you would like to be able to change over time, or even API keys. There's a great variety of scenarios where your applications can benefit from having access to dynamically changing data.

Let's try to create a simple configuration document and deliver it to the client instance:

1. Navigate to the **Other** section in your Nucleus web account and fill in the following JSON content:

```
{
  "message": "hello, world"
}
```

2. Click the **Save** button. Your page should now appear as follows:



3. Switch back to the application code and append the following code to the `createWindow` function:

```
// Fetch global settings
Nucleus.getCustomData((err, data) => {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
```

As you can see, we are using the `Nucleus.getCustomData` API to fetch the JSON object from the server. Your callback function is always going to receive two parameters: `err`, which provides any error details, and `data`, which provides the content of your server-side document.

4. We are going to redirect both parameters to the console output so that we can see what is coming as a response from the server.
5. Run the application with the `npm start` command. As soon as your Electron application starts, switch back to Command Prompt and check out the program's output.
6. You should see the following output:

```
$ electron .
{ message: 'hello, world' }
```

You have successfully integrated with the server-side settings. Feel free to provide the code that relies on the JSON document coming from the server.



As an experiment, try to modify the values and then restart your client. Note that the application fetches the updated data and that you don't need to issue a full release to update a few parameters.

Now, we are ready to check out licensing support.

License checking and policies

Last but not least, we are going to see the license checking feature in action.

The Nucleus service provides you with a minimalistic license checking mechanism that you can use with your Electron applications.



If you are looking for more sophisticated solutions, please check out the **Keygen** (<https://keygen.sh/>) service, a "dead-simple software licensing API built for developers."

The Nucleus service allows you to manage a list of licenses in the **Other / License Policies** section of your online account and then let the client application check their status.

Based on this check, you can either enable or disable certain features or notify users that the license has expired. You can even provide an in-app way to buy a new one or upgrade to a different tier if your application supports that.

Creating a new policy and license

Let's take a look at a license and perform a simple application-level check. Follow these steps to do so:

1. Navigate to the **Other** section of your Nucleus account. The initial screen should appear as follows:

ID	Validity	Version	Machines allowed	Price	Action
<input type="text" value="Policy id"/>	<div>Always</div>	<div>All versions</div>	<div>Unlimited</div>	<input type="text" value="19.99"/>	<button>Create</button>

Need help or something is missing?
→ hello@nucleus.sh

You can provide the following values for a new license policy:

- **Policy ID:** The unique identifier of the license policy.
- **Validity:** The life term of the license. Select either **Always**, **1 week**, **1 month**, **90 days**, or **1 year**.
- **Version:** The version of the application to use. Select either **All versions** or **Only current**.

- **Machines allowed:** The number of machines that can be used with the given license policy. You can pick **Unlimited** or either **1**, **2**, **5**, or **10**.
 - **Price:** The associated price.
2. Fill in the details for the new license policy using the following values:
 - **Policy ID:** a409f54f-b799-48e6-99ec-4d46bc4101a6
 - **Validity:** always
 - **Version:** all
 - **Machines allowed:** unlimited
 - **Price:** 9.99
 3. Click the **Create** button to save your changes and create a new policy. As soon as your license policy is created, you should see the following screen:

License policies

Policies define the different types of licenses that your app offers.

ID	Validity	Version	Machines allowed	Price	Action
a409f54f-b799-48e6-99ec-4d46bc4101a6	always	all	unlimited	9.99	Update Delete

[Create a policy](#)

Licenses

Here you can manage licenses associated to your app.

[Manually create a license](#)

4. Click **Manually create a license**. Feel free to provide your email address in the **New License** form. Notice the dropdown where you can pick the **License policy** you wish to use if you have more than one:

Licenses

Here you can manage licenses associated to your app.

User email

License policy


Create license



Use your real email address here – you're going to need it for the steps that follow.

5. Click on the **Create license** button. As soon as you click this button, the system sends an email confirmation with the license number. If you have provided a valid email address, you are going to get the following notification email for the **My App** application:

Your My App license Inbox x

**Nucleus**
to me ▾

Hello!

Thank you for obtaining a license for My App.

Your license is: **a479bd4d7fc1c57b8650**

Bests.


↩ Reply➡ Forward

6. Notice that the web page view changes. Now, you should have a list of available licenses, along with their details:

Licenses

Here you can manage licenses associated to your app.

Show entries Search:

License	User email	Policy	Version	Created	Expire
 556e3bb5e9f5e230d884	denys.vuika@gmail.com	a409f54f-b799-48e6-99ec-4d46bc4101a6	0.0.1	2019-08-19T20:19:18.938Z	never

Showing 1 to 1 of 1 entries Previous **1** Next


[Manually create a license](#)

7. You can control individual licenses by clicking the *plus* (+) icon next to one of the license's names and either temporarily disable the license or delete it completely:

Licenses

Here you can manage licenses associated to your app.

Show entries Search:

License	User email	Policy	Version	Created	Expire
 556e3bb5e9f5e230d884	denys.vuika@gmail.com	a409f54f-b799-48e6-99ec-4d46bc4101a6	0.0.1	2019-08-19T20:19:18.938Z	never

Actions Disable Delete

Showing 1 to 1 of 1 entries Previous **1** Next

[Manually create a license](#)

Now, let's integrate a license check into our Electron application.

Checking licenses in the application

So far, we have generated a new license, and we've received the license ID in our confirmation email. Now, it's time to validate this ID in the application code:

1. Switch to the `main.js` file and append the following code to the `createWindow` function:

```
Nucleus.checkLicense('556e3bb5e9f5e230d884', (err, license) => {  
  if (err) return console.error(err);  
  
  if (license.valid) {  
    console.log('License is valid :) Using policy ' +  
      license.policy);  
  } else {  
    console.log('License is invalid :(');  
  }  
});
```

Here, we're using the `Nucleus.checkLicense` function to get a callback with two parameters: an `err`, which provides any error details, and `license`, which provides details about the license.

For the sake of simplicity, we're going to redirect the check to the console output.

2. Run the application with the `npm start` script and check the console's output. You should see the following line in your console:

```
License is valid :) Using policy a409f54f-  
b799-48e6-99ec-4d46bc4101a6
```

3. For testing purposes, try to change the license ID to some other value and restart the application once again. This time, the console's output should be as follows:

```
License is invalid :(
```

Congratulations on integrating license checking into your Electron application.

In a real-life project, you will probably see a dialog asking for a license key. After doing so, you should store it somewhere and provide the different behaviors of the application based on the license's validation checks.



Please note that we have only touched on the basics of licensing and policies. You may want to harden storage and security, as well as encryption for your locally stored license values. Check out the **Keygen** (<https://keygen.sh/>) service if you want to find out more about sophisticated services and APIs.

Now, let's summarize what we have learned in this chapter.

Summary

In this chapter, you have successfully configured an Electron application with the online tracking and license checking features.

You are now able to use third-party analytics services with your projects, maintain and load global settings from the server, and perform license generation and validation.

In the next chapter, we are going to talk about chat applications and build a group chat with Google Firebase support.

8

Building a Group Chat Application with Firebase

So far, we have been building offline Electron applications. For offline-first applications, we store all the HTML, CSS, and JavaScript content locally and then embed all the files into the resulting package. Then, you can redistribute the package and release new versions of the application in case you need to change the code to bring new features to your application. We also store all the application data locally.

Another popular way of developing Electron applications is storing resources remotely on a server and using your application as a thin client. In this case, you can reduce the number of feature releases and give your application an *evergreen*. This means that you deploy new changes to the server and all the client applications automatically get new content on the next restart, for instance, or even in real time.

In this chapter, we are going to build a Slack-like chat application with the *group chat* feature. We are going to store all the data in the remote server and render group and message lists in real time. You are also going to learn how to store new messages on the server. The estimated build time for this project is three hours.

In this chapter, we will cover the following topics:

- Creating an Angular project
- Creating a Firebase account
- Creating a Firebase application
- Configuring Angular Material components
- Building a login dialog
- Connecting the login dialog to Firebase Authentication
- Configuring the Realtime database
- Rendering the group list
- Implementing the group messages page

- Displaying group messages
- Sending group messages
- Verifying the Electron Shell

We have a pretty big agenda, so let's get started and build our chat application. First, however, we are going to define the stack we are about to use.

Technical requirements

Before we dive into coding, let's decide on the stack we wish to use with the application. In this chapter, we are going to use the following:

- The Angular framework, for the overall application
- Angular components (also known as Angular Material) so that we can build user interfaces with the Google Material Design specification
- Firebase, for authentication, real-time databases, and hosting

Let's provide a quick overview of each item and discuss why we need it for this project:

- **Angular:** We plan to build a simple chat application with the remote backend. This means you are going to need the HTTP client, routing to support multiple pages, such as the Login and Chat windows, and many other smaller pieces. To achieve our goal fast, we need to focus on the implementation of our application features, rather than building the whole ecosystem from scratch. This is why we aren't going to start with plain JavaScript and use the Angular CLI tool. We are about to generate a ready to use web application that we can package as an Electron project.



We described how we can get started with Electron and the Angular framework in *Chapter 3, Integrating with Angular, React, and Vue*. Feel free to jump to that chapter and revisit the steps that are provided there.

- **Angular Material:** Our chat application is going to need some UI components. We need at least a Login dialog, a Chat window, the input components, and many other blocks. To save time, we shall be using Angular Material components as an easy and natural way to build a typical Angular application. The Angular Material library already contains a great variety of components. Ensure that you check out the main website if you wish to view any documentation, guides, and examples: <https://material.angular.io/>.

Next, we need to pick our data storage solution. In this chapter, we are going to be using Google Firebase.

- **Google Firebase:** We need to store our chat data somewhere in the cloud. For this purpose, we are going to use Firebase. Firebase is a very popular mobile and web application development platform. It provides a wide variety of services that you can use to boost your applications, including the following:
 - A real-time database that allows us to sync data across devices and platforms
 - An authentication service with multiple protocols and integrations
 - Hosting
 - Push notifications
 - Analytics



You can find out more about Firebase at <https://firebase.google.com/>.

Please note that Google Firebase is not the only solution out there. Check out the following link for the top 10 Firebase alternatives: <https://blog.back4app.com/2018/01/12/firebase-alternatives/>.

It is effortless to get started with Google Firebase, and we are going to walk through that process in this chapter. First, though, let's create a new Angular project.

You can find the code files for this chapter in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter08>.

Creating an Angular project

In this chapter, we are going to craft a basic scaffold for our project that suits the needs of our chat application. Let's get started:

1. Create a new project folder.
2. Run the following command to initialize a new project:

```
ng new chat-app
```

3. Regarding the question about Angular routing support, answer **Yes** or type **y**:

```
Would you like to add Angular routing? (y/N)
y
```

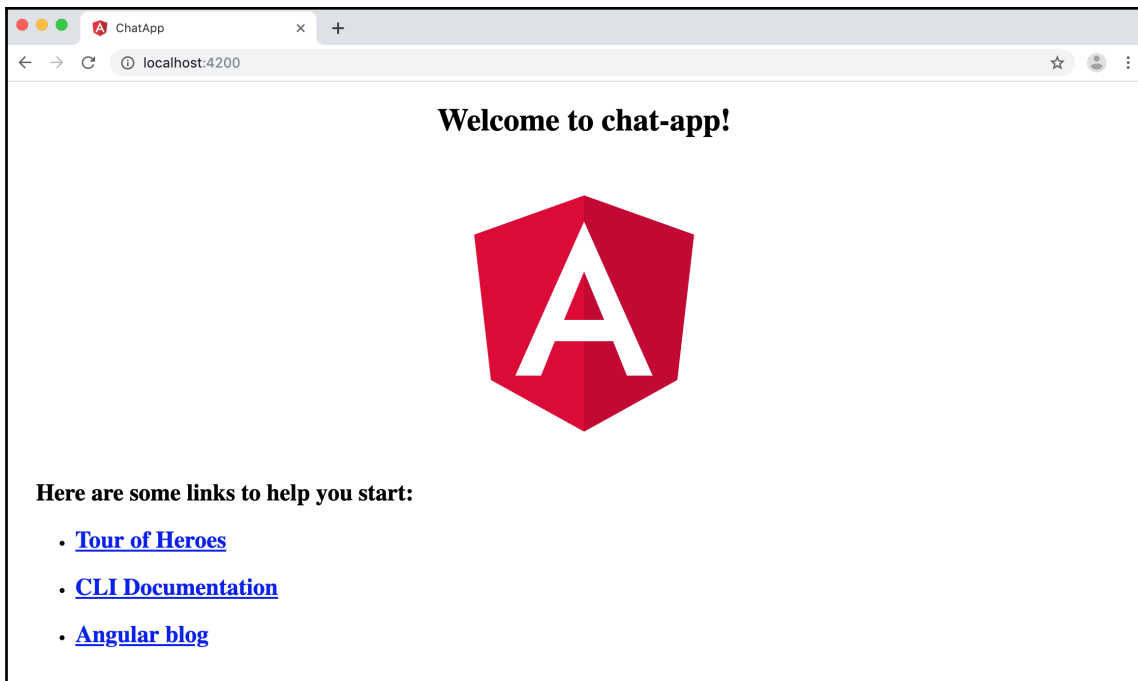
4. Regarding the question about stylesheet format, pick **SCSS**:

```
Which stylesheet format would you like to use?
SCSS
```

5. In the Angular CLI, you will see the following output:

```
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS [ https://sass-lang.com/documentation/syntax#scss ]
CREATE chat-app/README.md (1024 bytes)
CREATE chat-app/.editorconfig (246 bytes)
CREATE chat-app/.gitignore (631 bytes)
CREATE chat-app/angular.json (3697 bytes)
CREATE chat-app/package.json (1282 bytes)
CREATE chat-app/tsconfig.json (543 bytes)
CREATE chat-app/tslint.json (1988 bytes)
CREATE chat-app/browserslist (429 bytes)
CREATE chat-app/karma.conf.js (1020 bytes)
CREATE chat-app/tsconfig.app.json (270 bytes)
CREATE chat-app/tsconfig.spec.json (270 bytes)
CREATE chat-app/src/favicon.ico (5430 bytes)
CREATE chat-app/src/index.html (294 bytes)
CREATE chat-app/src/main.ts (372 bytes)
CREATE chat-app/src/polyfills.ts (2838 bytes)
CREATE chat-app/src/styles.scss (80 bytes)
CREATE chat-app/src/test.ts (642 bytes)
CREATE chat-app/src/assets/.gitkeep (0 bytes)
CREATE chat-app/src/environments/environment.prod.ts (51 bytes)
CREATE chat-app/src/environments/environment.ts (662 bytes)
CREATE chat-app/src/app/app-routing.module.ts (246 bytes)
CREATE chat-app/src/app/app.module.ts (393 bytes)
CREATE chat-app/src/app/app.component.scss (0 bytes)
CREATE chat-app/src/app/app.component.html (1152 bytes)
CREATE chat-app/src/app/app.component.spec.ts (1101 bytes)
CREATE chat-app/src/app/app.component.ts (213 bytes)
CREATE chat-app/e2e/protractor.conf.js (810 bytes)
Directory is already under version control. Skipping initialization of git.
```

6. Try out the app using the `ng serve --open` command:



We are doing well so far. Now, it's time to configure the Electron Shell.

Configuring the Electron Shell

To configure the Electron Shell's integration, we need to make a few changes to the project files. Let's get started:

1. Update the `src/index.html` file so that it contains the following code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>ChatApp</title>
    <base href="." />

    <meta name="viewport" content="width=device-width,
      initial-scale=1" />
    <link rel="icon" type="image/x-icon" href="favicon.ico" />
  </head>
  <body>
```

```
<app-root></app-root>
</body>
</html>
```

2. Install the Electron library with the following command:

```
npm i electron -D
```

3. Update the `package.json` file:

```
{
  "name": "chat-app",
  "version": "0.0.0",
  "main": "main.js",
  "scripts": {
    "ng": "ng",
    "serve": "ng serve",
    "start": "electron .",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  // other content
}
```

We are going to use the `npm run serve` command to run the Angular application and `npm run start` or `npm start` to launch the Electron application.



More details about script configuration are provided in Chapter 3, *Integrating with Angular, React, and Vue*. Make sure that you check out the examples regarding how to set up production builds.

4. Put the `main.js` file, along with the following code, into the project's root folder:

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

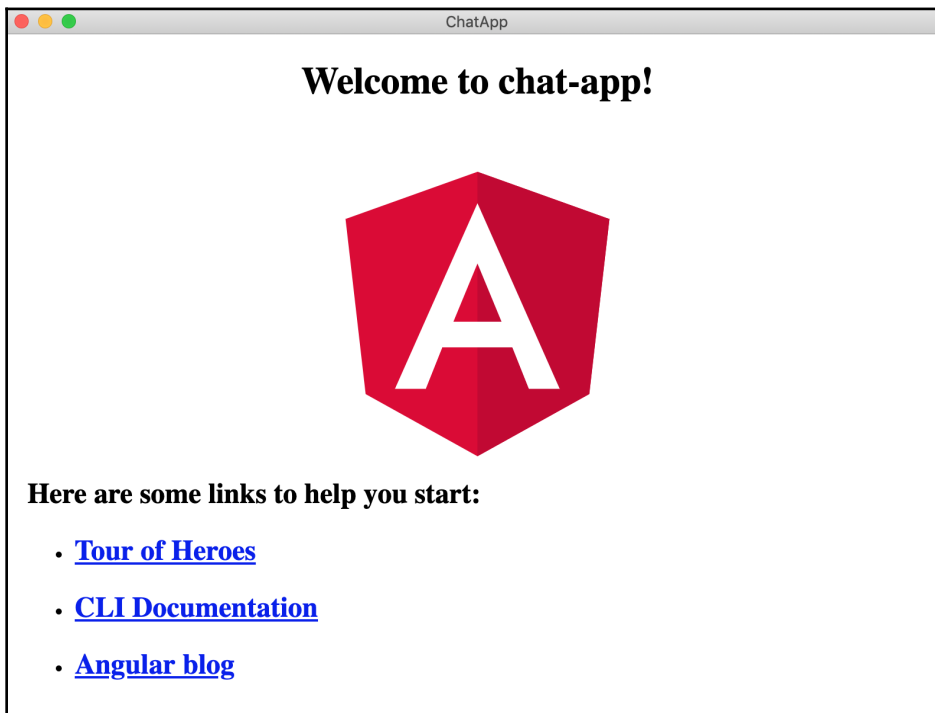
function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    },
  },
  {
    resizable: false
  })
}
```

```
});  
  
win.loadURL(`http://localhost:4200`);  
}  
  
app.on('ready', createWindow);
```

5. Run the following commands in two separate console instances:

```
# first console  
npm run serve  
  
# second console  
npm start
```

6. As a result, you should see the window of the application, as shown in the following screenshot:

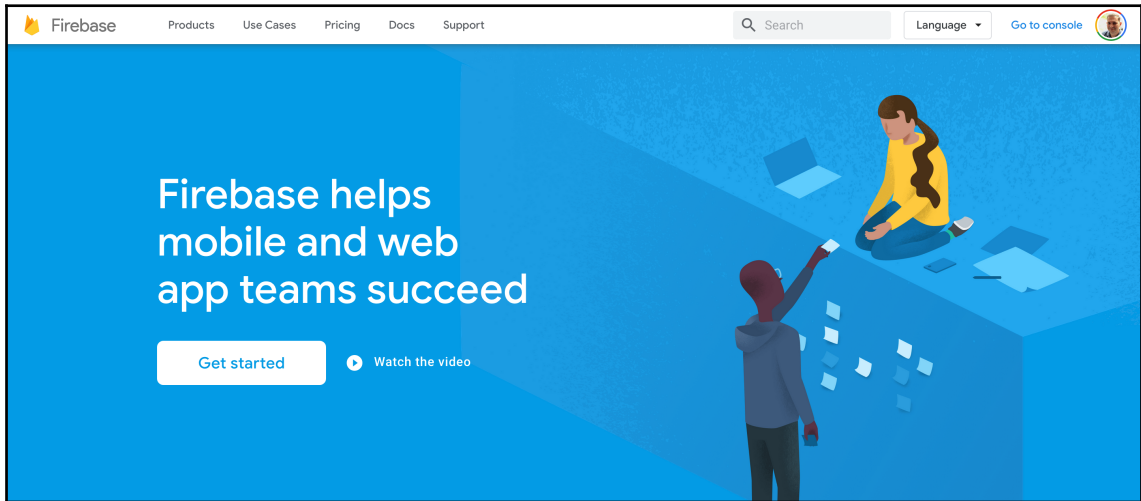


The bare application scaffold is ready. Now, it's time to create a new Firebase account.

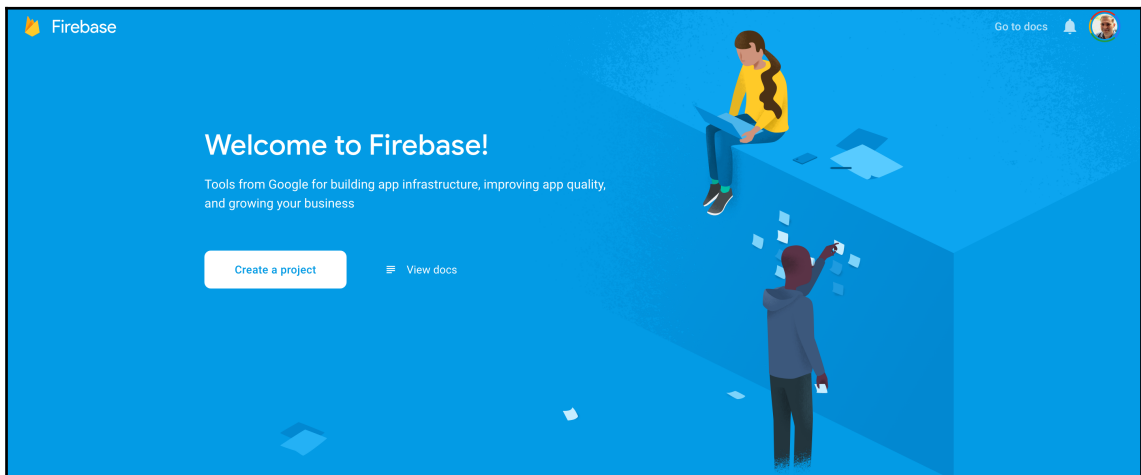
Creating a Firebase account

In this section, we are going to create a new Firebase account and gain access to the Firebase console. The good news is that all you need is an active Google account. Let's get started:

1. Navigate to <https://firebase.google.com/> and click the **Get started** button:



2. Click the **Create a project** button:



3. Fill in the form; call the project `electron-chat-app`.
4. Take note of the unique project ID value that's generated.



Your project's globally unique identifier is used in your real-time database URL, Firebase Hosting subdomains, and more. You cannot change your project ID after the project has been created.

It has a value of `electron-chat-app-df7eb`, but this value varies for every project. Leave it as it is and click **Continue**:

A screenshot of the Firebase console's 'Create a project' screen. The title bar says 'X Create a project (Step 1 of 3)'. The main heading is 'Let's start with a name for your project'. Below this, there's a 'Project name' label and a text input field containing 'electron-chat-app'. Underneath the input field is a small button with a pencil icon and the text 'electron-chat-app-df7eb'. At the bottom left is a blue 'Continue' button. On the right side of the screen is a blue illustration of a man in a suit holding a smartphone and a woman in a yellow shirt sitting at a desk with a laptop.

5. Now, we need to choose whether we want to enable Google Analytics for our project or not. Select **Not right now** since we aren't going to look at analytics in this chapter:

× Create a project (Step 2 of 2)

Google Analytics for your Firebase project

Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, Predictions, and Cloud Functions.

Google Analytics enables:

- × A/B-testing ?
- × User-segmentation & targeting across Firebase products ?
- × Predicting user behavior ?
- × Crash-free users ?
- × Free unlimited reporting ?

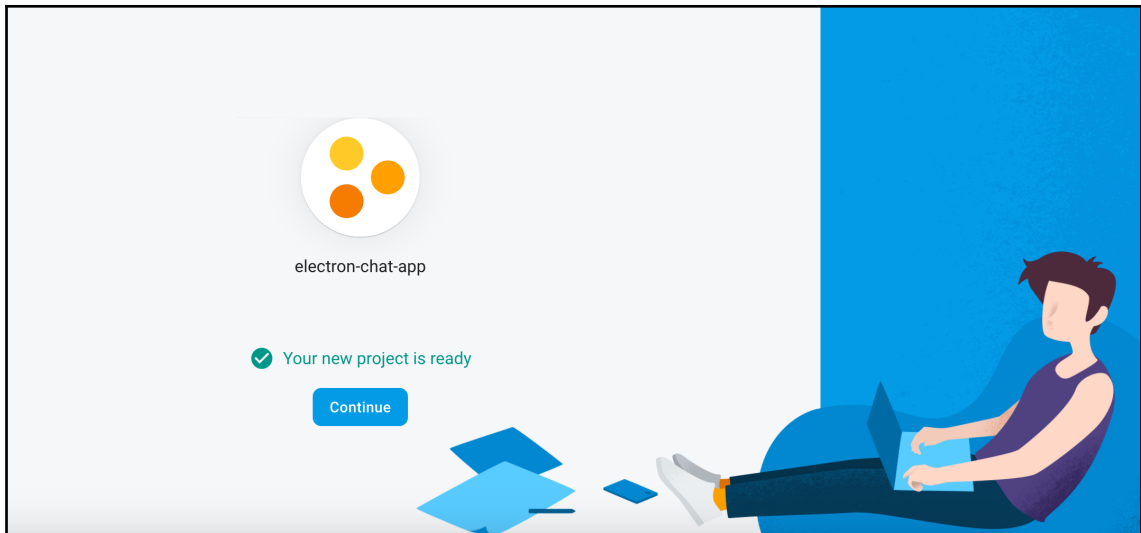
☐ **Set up Google Analytics for my project**
Configure in the next step

☒ **Not right now**
You can change this later

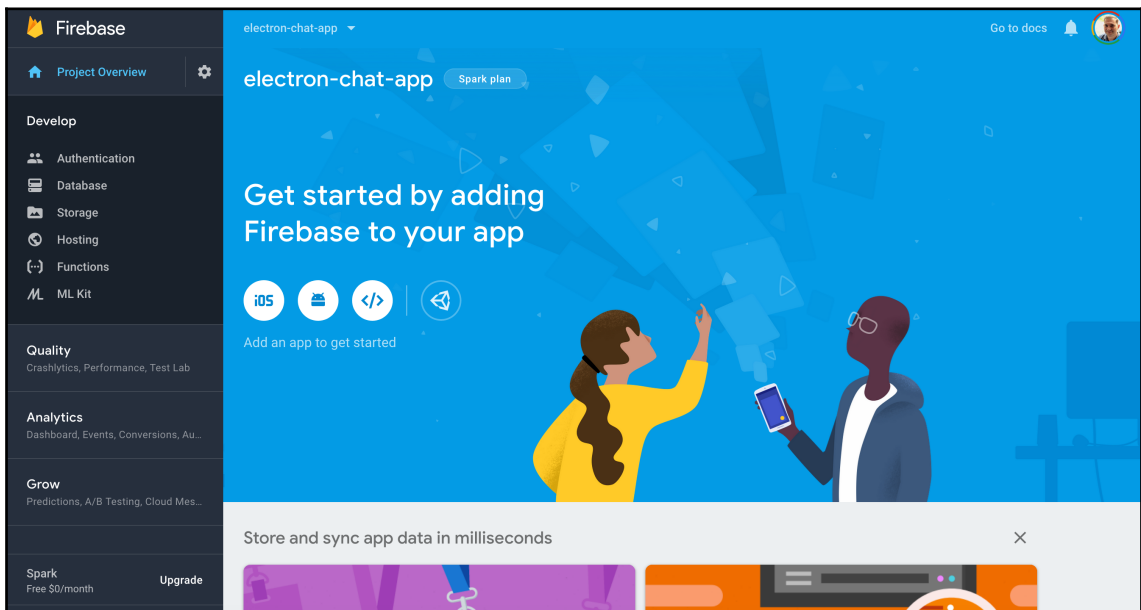


You can always enable the **Setup Google Analytics for my project** feature later when you get more familiar with Firebase.

6. Click the **Create project** button.
7. It may take a few seconds to create your new project. Watch the animated progress bar; it should show the **Your project is ready** label as soon as your project has been generated:



8. After clicking **Continue**, you should be able to see the console dashboard for your application:





Firebase provides multiple pricing plans; find out more at <https://firebase.google.com/pricing>. What you need to know is that, by default, every Firebase application gets the free "Spark" plan, which has generous limits for hobbyists. This plan is more than enough for you to get started with application development and testing.

At this point, you have an empty `electron-chat-app` project. At the time of writing, you have the option to create and register four different types of applications:

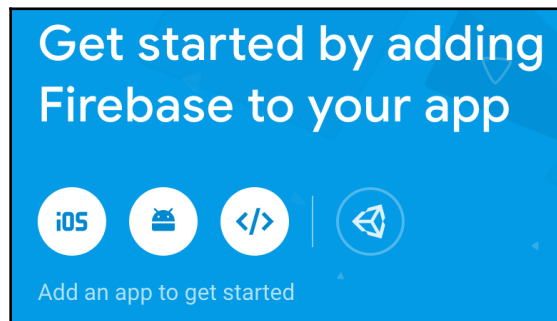
- An iOS application
- An Android application
- A web application
- A Unity game for either iOS or Android

Now that we have created a Firebase account, we are going to create a web application entry.

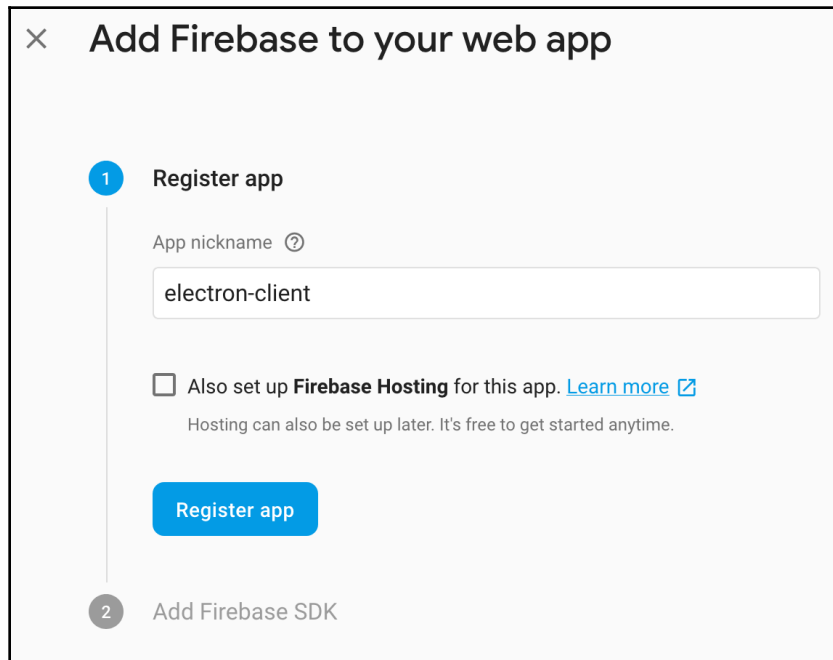
Creating a Firebase application

In this section, we are going to register a web application. Follow these steps to do so:

1. Click the corresponding button on the screen to get to the **Create an application** dialog:



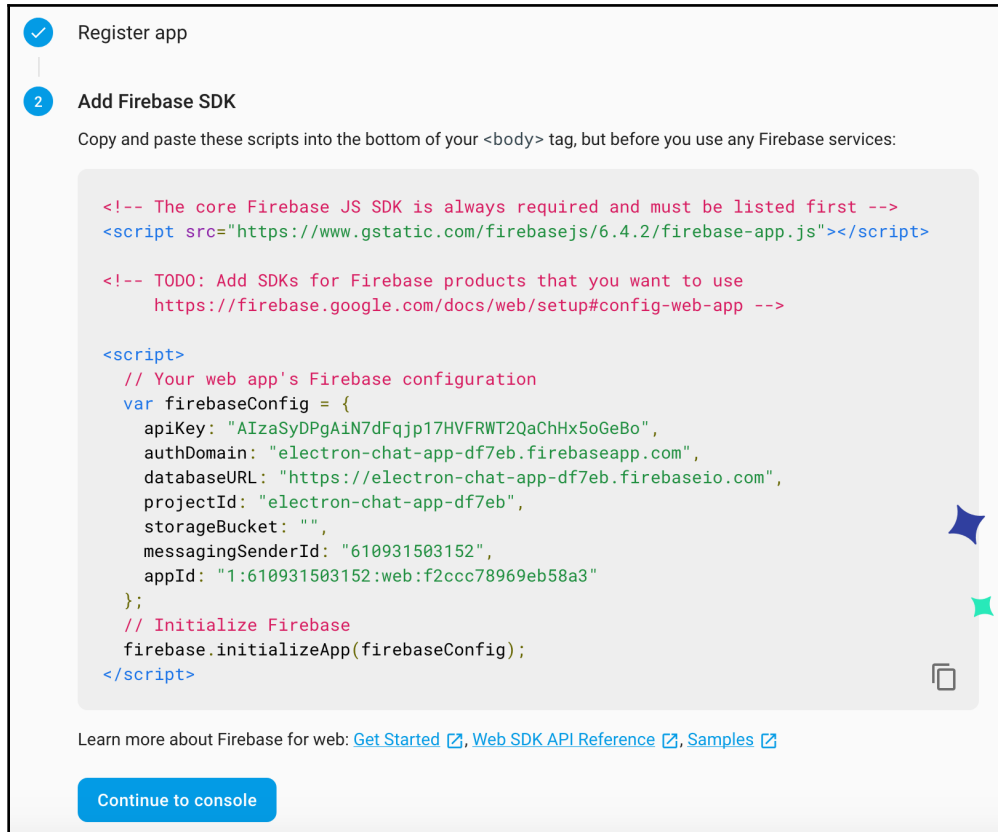
2. Next, you need to provide an application nickname. Enter `electron-client` as the nickname:



The app nickname will be used throughout the Firebase console to represent this app. Nicknames aren't visible to users.

3. It is also possible to set up Firebase Hosting for your application. Leave the value unchecked for now. Click the **Register app** button.

- Now, you will be presented with an HTML snippet that you will need in order to set up a brand new web application with your Firebase project. It should look similar to the one shown in the following screenshot:



This code contains all the values that are relevant to your current project. Please note that all the keys and identifiers may be different for you.



TIP

You can copy this code and save it somewhere in case you need it later. Alternatively, there's a section in the project settings where you can retrieve this block again.

- Click the **Continue to console** button to finish setting up the application.

Now, let's quickly set up the Angular Material libraries.

Configuring Angular Material components

In this section, we are going to install the dependencies for the Angular Material components. We are also going to integrate the necessary bits and pieces into the project. Run the following command:

```
npm install --save @angular/material @angular/cdk @angular/animations
```

Usually, developers need to perform a series of steps to set up Angular Material components with a new Angular project. You can find out more at <https://material.angular.io/guide/getting-started>.

This time, we need to walk through the following steps:

- Adding an animations module
- Configuring the default theme
- Adding the Material Icons library

We are also going to test the overall setup by adding a material toolbar component that serves as the navigation bar.

Adding a Browser Animations module

You need to integrate the Browser Animations module for the material components to work correctly. Follow these steps to do so:

1. Switch to the `src/app/app.module.ts` file.
2. Import `BrowserAnimationsModule`:

```
import { BrowserAnimationsModule } from '@angular/platform-browser
/animations';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, BrowserAnimationsModule,
    AppRoutingModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

The next step is configuring the default theme settings.

Configuring the default theme

This step is pretty simple: you need to set one of the themes as the application's default theme. Let's use the same one that comes with the documentation examples.

Update the `src/styles.scss` file by using the following command:

```
@import "~@angular/material/prebuilt-themes/indigo-pink.css";
```

Now, we need the Material Icons library.

Adding the Material Icons library

Later in this chapter, we are going to use some icons for the group chat lists. Let's integrate the Google Material Icons library into the project.

Update the `src/index.html` file and place the following code in the head section:

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">
```

Let's test the overall setup and provide a navigation bar for our application.

Adding a navigation bar

In this section, we are going to use the material toolbar component as an application header bar. This bar allows our application to navigate to the **Login** screen and possibly other areas as well.

To get the material toolbar into the application, we need the corresponding module. Let's get started:

1. Import `MatToolbarModule` into the main application module, that is, `src/app/app.module.ts`:

```
import { MatToolbarModule } from '@angular/material/toolbar';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    AppRoutingModule,
  ],
})
```

```

    MatToolbarModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}

```

2. Replace the contents of the `src/app/app.component.html` file with the following code:

```

<mat-toolbar color="primary">
  <span>Electron Chat</span>

  <!-- This fills the remaining space of the current row -->
  <span class="fill-space"></span>

  <span>Login</span>
</mat-toolbar>

<router-outlet></router-outlet>

```

3. Update the `src/app/app.component.scss` file so that it contains the following code:

```

.fill-space {
  /* This fills the remaining space, by using flexbox.
     Every toolbar row uses a flexbox row layout. */
  flex: 1 1 auto;
}

```

4. Finally, add some UI polishing to the `src/styles.scss` file. Now, we are ready to go:

```

@import '~@angular/material/prebuilt-themes/indigo-pink.css';

body,
html {
  height: 100%;
}

body {
  margin: 0;
}

```

Now, it's time to test our application's structure and see it in action.

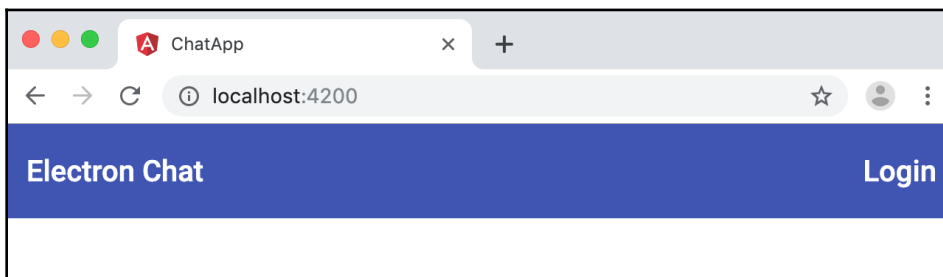
Testing the application with the material toolbar

Let's switch to Command Prompt or a Terminal window and test the application to ensure that everything goes as expected.

Run the following command:

```
ng serve --open
```

The preceding command starts the web server and opens your default browser with the application running inside it. You should see the following page:



Everything looks good so far. Now, we will build the login dialog.

Building a login dialog

At this point, you may be wondering why we need a Login dialog in our application. The answer is for application data security. In real life, all data access must be protected, and you are going to enable all kinds of security restrictions when you release your application into production. This is why we need to implement a login dialog and authenticate the session with the remote server.

Follow these steps to build a Login dialog:

1. Generate a `login` component scaffold by running the following command:

```
ng g component login
```

2. In the `src/app/app-routing.module.ts` file, create a new Route with a `/login` URL that points to the Login Dialog component:

```
// ...
import { LoginComponent } from '../login/login.component';

const routes: Routes = [
  {
    path: 'login',
    component: LoginComponent
  }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

We need to make a title area so that we can navigate to the default home page and so that the Login link navigates to our `/login` route.

3. Update the `src/app/app.component.html` file and replace the span elements with hyperlinks:

```
<mat-toolbar color="primary">
  <a [routerLink]='/'>Electron Chat</a>

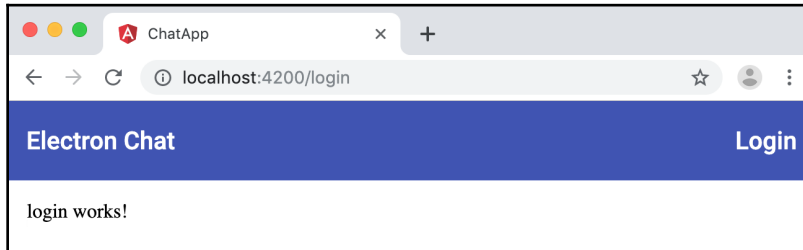
  <span class="fill-space"></span>
  <a [routerLink]='login'>Login</a>
</mat-toolbar>

<router-outlet></router-outlet>
```

4. Update the `src/app/app.component.scss` file by making some styling changes to make the hyperlinks look better on the blue background:

```
.mat-toolbar {
  & > a {
    text-decoration: none;
    color: white;
  }
}
```

5. Run the web application in the local web server once more with the `ng serve -o` command. Click the **Login** link to navigate to the Login Dialog component's implementation. By default, it should contain the text **login works!**:



6. Click on the title area to make sure you can navigate back to the home page. The home page should be blank for the time being. This is fine; we will add some content to it later.

Now that we have a placeholder for the Login dialog, let's build the traditional user interface. This interface will contain the username and password input fields, as well as a submit button.

Implementing the Material interface

We need at least two additional Material modules in order to implement a basic Login form, that is, the `Input` and `Button` component modules. Let's look at how we can add them:

1. Update the `src/app/app.module.ts` file so that it contains the following code:

```
import { MatButtonModule } from '@angular/material/button';
import { MatInputModule } from '@angular/material/input';

@NgModule({
  declarations: [AppComponent, LoginComponent],
  imports: [
    // ...
    MatInputModule,
    MatButtonModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

2. Replace the contents of the `src/app/login/login.component.html` file with the following code:

```
<div class="login-form-container">
  <div class="login-form">
    <h1>Login</h1>

    <mat-form-field class="login-field">
      <input #loginField matInput placeholder="Username"
        autocomplete="off" />
    </mat-form-field>

    <mat-form-field class="login-field">
      <input #passwordField type="password" matInput
        placeholder="Password" />
    </mat-form-field>
    <div class="login-actions">
      <button mat-raised-button>Login</button>
    </div>
  </div>
</div>
```

In the preceding code, we are declaring a heading with the `Login` text. We have also provided two input fields and a button to perform authentication.

Next, we need to provide styling for the login form. Let's have the form centered horizontally. The `Login` button should be on the right-hand side of the screen. Let's get started:

1. Update the `src/app/login/login.component.scss` file so that it contains the following code:

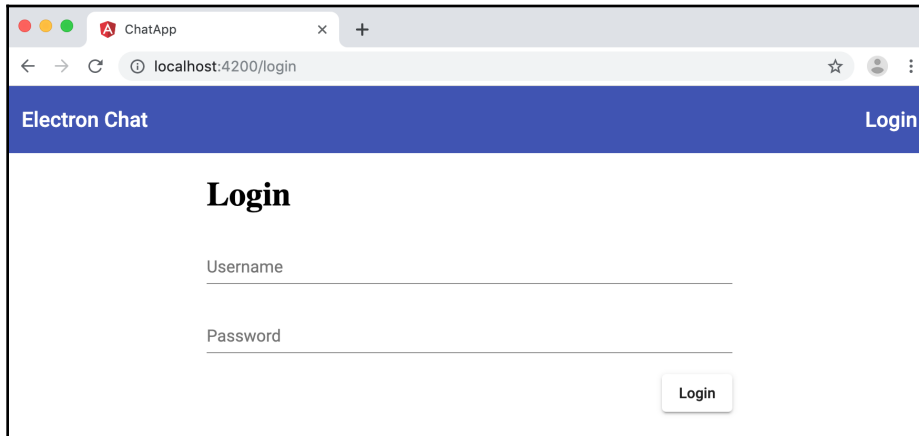
```
.login-form-container {
  display: flex;
  .login-form {
    margin: auto;
    min-width: 150px;
    max-width: 500px;
    width: 100%;

    .login-field {
      width: 100%;
    }

    .login-actions {
      text-align: right;
    }
  }
}
```

```
}  
}
```

2. Run or restart the web server and check out the `/login` route.
3. The application should now look as follows:



We need to provide at least basic validation and error handling for the dialog.

Supporting error handling

Given that the authentication may fail, let's provide some minimalistic error handling:

1. Update the `login.component.html` file with an extra `h2` element under the dialog title:

```
<h1>Login</h1>  
<h2 class="error" *ngIf="error">Error: {{ error }}</h2>
```

As you can see, we only display the `h2` element if the `error` property value is provided for us.

2. We need the label to be red, so update the `login.component.scss` stylesheet with the corresponding color for the `error` class:

```
.error {  
  color: red;  
}
```

3. Finally, update the code of the `login.component.ts` file and provide an empty `error` property:

```
// ...
export class LoginComponent implements OnInit {
  error = '';
  // ...
}
```

In the next section, we are going to address the result of successfully navigating and the Chat view that our users are going to be using. Let's build the placeholder component first and then add more content to it.

Preparing the chat component placeholder

In this section, we are going to create a placeholder component for the chat groups list. Follow these steps to do so:

1. Generate a new Chat component by running the following command:

```
ng g component chat
```

2. Update the routes in the `app-routes.module.ts` file:

```
import { ChatComponent } from './chat/chat.component';

const routes: Routes = [
  {
    path: 'login',
    component: LoginComponent
  },
  {
    path: 'chat',
    component: ChatComponent
  }
];
```

3. Update the login form so that it invokes the `login` method:

```
<div class="login-actions">
  <button
    mat-raised-button
    (click)="login(loginField.value, passwordField.value)"
  >Login</button>
</div>
```

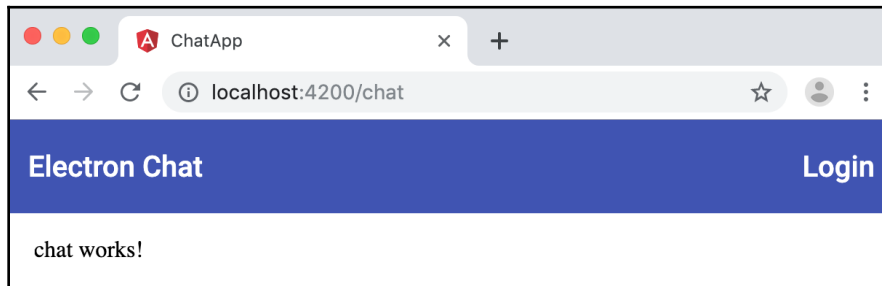
4. We need to update the Login component's code. We need to import the `Router` object and implement the `login` function. For now, we can navigate to the successful chat route without authentication.
5. Update the `chat.component.ts` file:

```
import { Router } from '@angular/router';

@Component({...})
export class LoginComponent {
  error = '';
  constructor(private router: Router) {}

  login(username: string, password: string) {
    // perform login here
    this.router.navigate(['chat']);
  }
}
```

6. Restart the web server, navigate to the `Login` page, and fill in the username and password input boxes.
7. Click the **Login** button; you should end up on the **Chat** page, which has the `/chat` route, as shown in the following screenshot:



At this point, we are ready to wire the Login dialog to Firebase Authentication. Let's learn how to integrate Firebase with the Login dialog.

Connecting the login dialog to Firebase Authentication

In the previous section, you created a login dialog component that receives `Username` and `Password` inputs and which should redirect us to the `Chats` page upon successful authentication. Now, we need to configure the Firebase project and provide an authentication mechanism that our application can use.

In this section, we are going to perform the following actions:

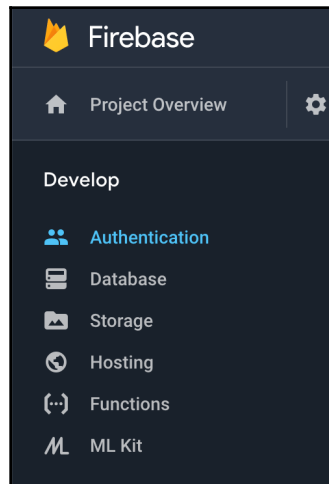
- Enable the sign-in provider so that we can use email/password authentication.
- Create some demo accounts to test the login dialog.
- Integrate the login dialog component with the remote authentication.

Let's start by enabling the sign-in provider.

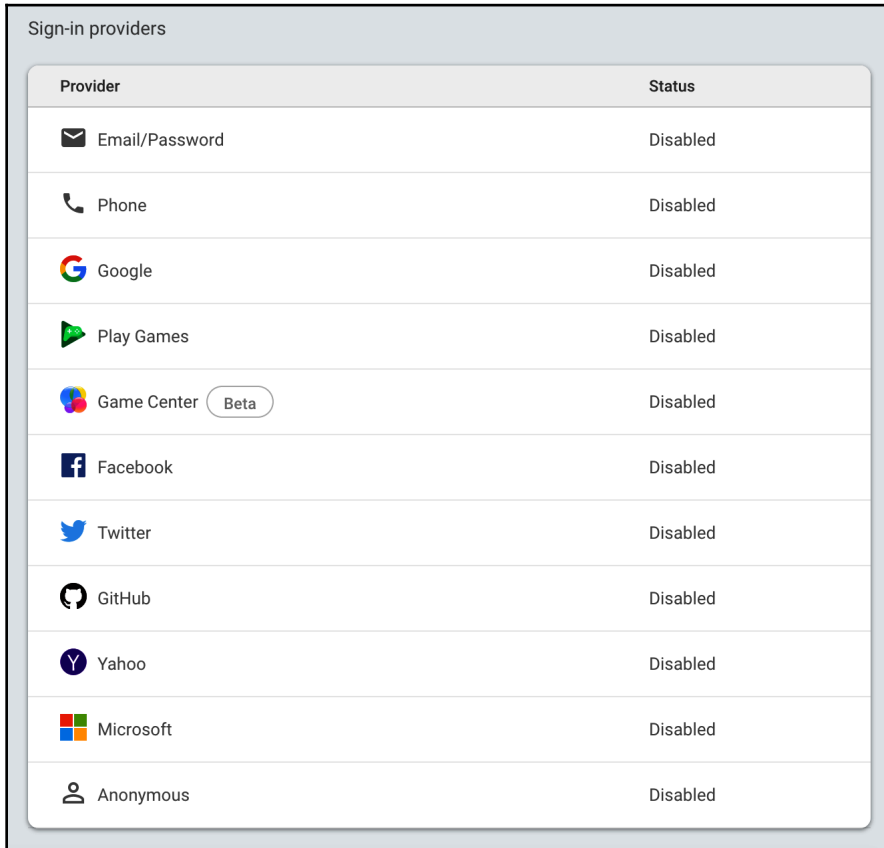
Enabling the sign-in provider












Let's enable Firebase's authentication features and pick the sign-in provider by following these steps:

1. Switch back to the Firebase console and click the **Authentication** link in the **Develop** section of the navigation sidebar:



2. You should see a list of supported sign-in providers in the main content area. As shown in the following screenshot, Firebase provides support for a wide variety of authentication mechanisms. For the sake of simplicity, let's use the traditional **Email/Password** provider:



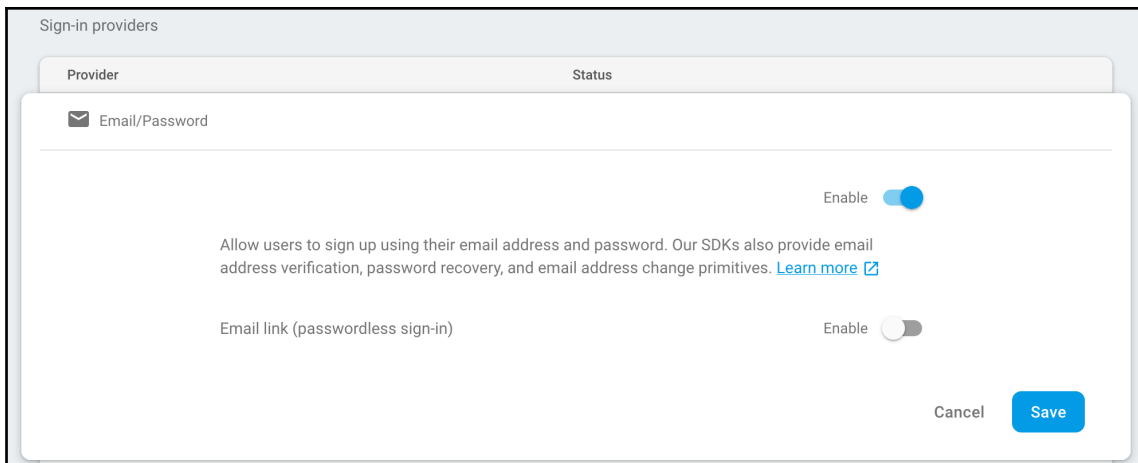
Provider	Status
 Email/Password	Disabled
 Phone	Disabled
 Google	Disabled
 Play Games	Disabled
 Game Center Beta	Disabled
 Facebook	Disabled
 Twitter	Disabled
 GitHub	Disabled
 Yahoo	Disabled
 Microsoft	Disabled
 Anonymous	Disabled

3. Click the **Email/Password** list item to get the corresponding dialog.

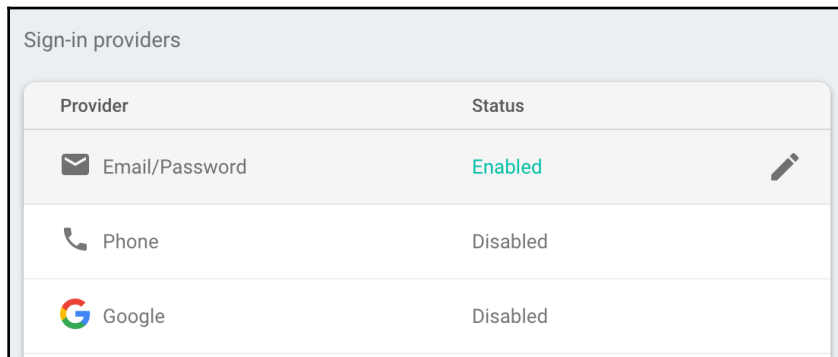


Later on, you can provide more than one sign-in provider in the Login dialog if you like, for example, Twitter, Facebook, Google, and more.

4. In the dialog, click the **Enable** toggle option and then press **Save**:



5. Now, the provider should be enabled in the list of sign-in providers, as shown in the following screenshot:



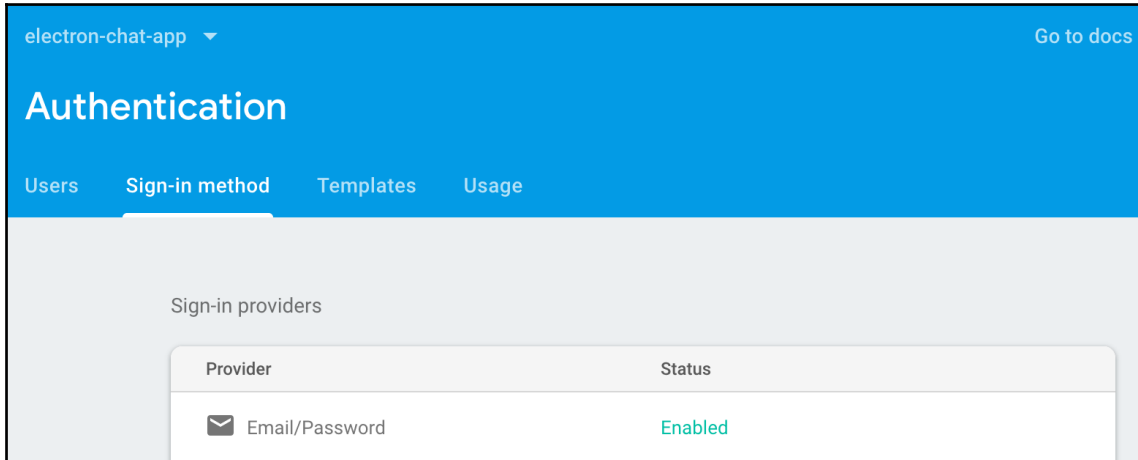
TIP

If you don't have the provider enabled, you can always perform the same steps again. We can also do this to disable it in case we ever want to switch to another sign-in provider.

We have just **Enabled** the **Email/Password** authentication provider. Let's learn how to create accounts directly within the Firebase console.

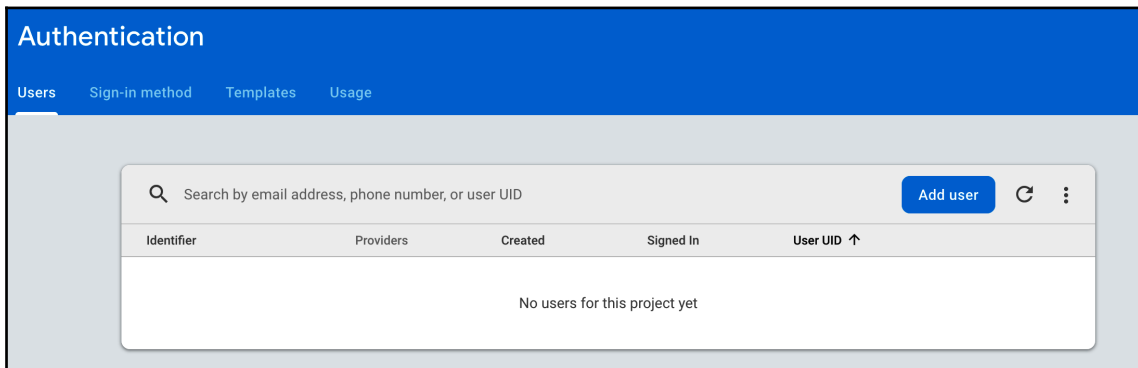
Creating demo accounts

By default, when you click the **Authentication** link in the sidebar, you will see the **Sign-in method** tab in the main content area. Scroll up and check out the other tabs we have here:



The Firebase console allows us to access all of our registered accounts. We can view them and even manually create new or edit existing accounts within the **Users** tab. Follow these steps to do so:

1. Click **Users** to view the content of that page:



2. We don't have any users so far. Let's create at least two so that we can test our chat features with two different accounts. Click the **Add user** button to invoke the user creation dialog:

Search by email address, phone number, or user UID

Add user

Add an Email/Password user

Email: denys.vuika@gmail.com

Password: password

Cancel **Add user**

No users for this project yet

Identifier	Providers	Created	Signed In	User UID ↑
------------	-----------	---------	-----------	------------

3. Fill in the input boxes and click the **Add user** button. Repeat this process at least one more time.
4. Now, you should have a list of users that can sign in from within our web application:

Authentication

Users | Sign-in method | Templates | Usage

Search by email address, phone number, or user UID

Add user

Identifier	Providers	Created	Signed In	User UID ↑
denys.vuika@outlook.com	✉	Aug 26, 2019		kTdLcJYWa9UPwe8K9s2QCIXr9Pc2
denys.vuika@gmail.com	✉	Aug 26, 2019		3OemljJzZnWz5gAfjhXbal9u3sP2

Rows per page: 50 | 1-2 of 2

Now, let's switch to our Angular application and integrate the Login dialog with Firebase.

Integrating the Login dialog with Firebase

First of all, we need to store the Firebase configuration settings inside the project environment variables. Follow these steps to do so:

1. Update the `src/environments/environment.ts` file with the Firebase configuration that you received earlier in this chapter when setting up the Firebase project:

```
export const environment = {
  production: false,
  firebaseConfig: {
    apiKey: 'AIzaSyDPgAiN7dFqjp17HVFRWT2QaChHx5oGeBo',
    authDomain: 'electron-chat-app-df7eb.firebaseio.com',
    databaseURL: 'https://electron-chat-app-df7eb.firebaseio.com',
    projectId: 'electron-chat-app-df7eb',
    storageBucket: '',
    messagingSenderId: '610931503152',
    appId: '1:610931503152:web:f2ccc78969eb58a3'
  }
};
```



Note that the actual values are going to be different for you.

2. Install the AngularFire2 library with the following command:

```
npm install @angular/fire firebase
```



AngularFire 2 is the official library for Firebase and Angular. It saves a lot of your time and effort when it comes to using Firebase APIs in applications. You can find out more about this library at <https://github.com/angular/angularfire2>.

3. Next, import and set up the Angular Fire modules in the `src/app/app.module.ts` file according to the following code:

```
import { AngularFireModule } from '@angular/fire';
import { AngularFireAuthModule } from '@angular/fire/auth';
import { environment } from '../environments/environment';

@NgModule({
  // ...
  imports: [
```

```
// ...
AngularFireModule.initializeApp(
  environment.firebaseConfig
),
AngularFireAuthModule
],
// ...
})
export class AppModule {}
```

As you can see, we've imported `AngularFireModule` and initialized it with `firebaseConfig` from the `environment.ts` file. We've also imported the `AngularFireAuthModule` module, which carries all the structures that our application may require to get our authentication up and running.

4. Switch back to the `login.component.ts` file and import the `AngularFireAuth` service. You also need to inject it via the constructor, as shown in the following code:

```
import { AngularFireAuth } from '@angular/fire/auth';

// ...
export class LoginComponent {
  // ...

  constructor(
    private router: Router,
    private firebaseAuth: AngularFireAuth) {}
  // ...
}
```

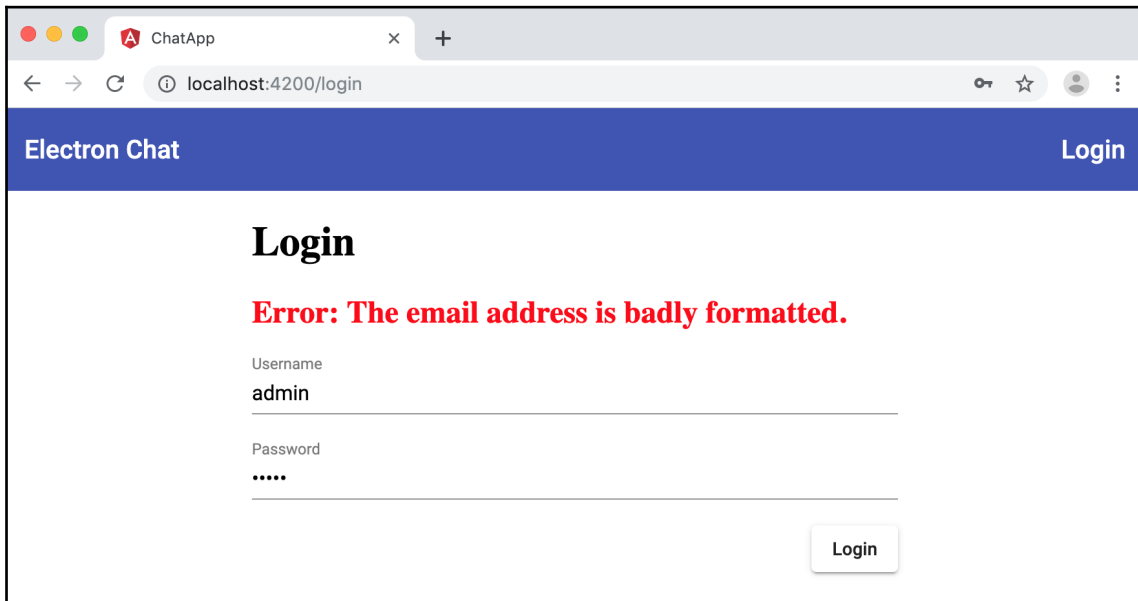
5. Next, update the implementation of the `login` function using the following code:

```
login(username: string, password: string) {
  this.firebaseAuth.auth.signInWithEmailAndPassword(username,
    password).then(
    credential => {
      console.log(credential);
      this.router.navigate(['chat']);
    },
    err => {
      this.error = err.message || 'Unknown error';
    }
  );
}
```

The preceding code is self-explanatory. Here, we call the `signInWithEmailAndPassword` function that AngularFire provides and pass our username and password to it. Upon a successful call, we log the resulting credential object to the console (for testing purposes) and then navigate to the `/chat` page. If an error occurs, we update the `error` property and display the error to the user.

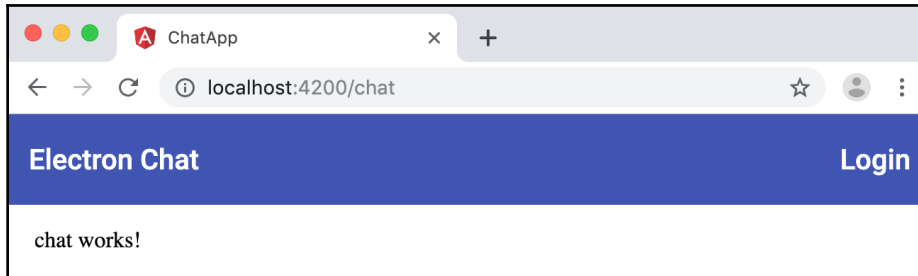
Let's try the failure scenario first:

1. Enter some incorrect credentials and press the **Login** button. You should see the following error on your screen:



2. Now, use the credentials that you created earlier in the Firebase console.

3. This time, you should see no errors. The application will display the `/chat` page after we click on the `Login` button:



4. We also dump the server response into the console log. If you switch to the developer tools, you should see the following data:

```
{...}
  additionalUserInfo: {...}
    isNewUser: false
    providerId: "password"
    > <prototype>: Object { ... }
  credential: null
  operationType: "signIn"
  > user: Object { l: "AIzaSyDPgAiN7dFqjp17HVFRWT2QaChHx5oGeBo", o: "[DEFAULT]", u: "electron-chat-app-df7eb.firebaseio.com", ... }
  > <prototype>: Object { ... }
```

As you can see, the server provides you with a set of additional information that your application may require. Please refer to the Firebase documentation to see what those fields are and how you can use them.

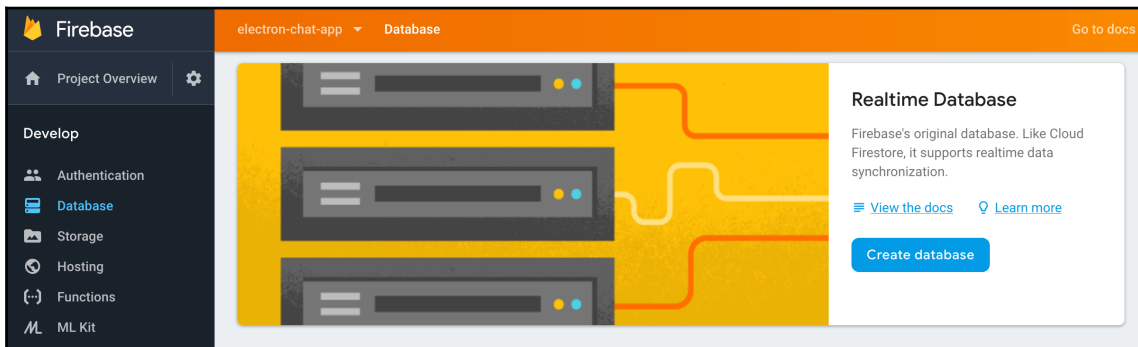
You have made significant progress so far. We have set up a Firebase project that's running authentication checks for our application. We also have a fully working, though minimalistic, Login dialog. So far, it isn't possible to log into the application and be redirected to the **Chat** page.

In the next section, we are going to start working on the chat functionality, starting with the database configuration.

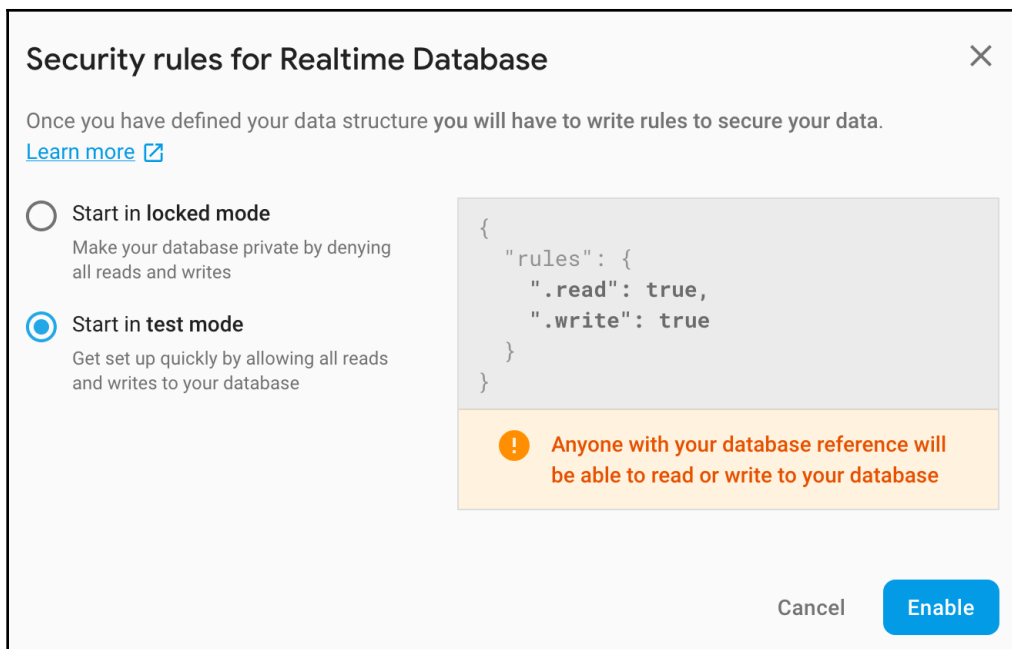
Configuring the Realtime Database

In the previous sections, we successfully configured authentication. Now, it's time to get the database ready. We need some storage for the group chat information and messages. Let's get started:

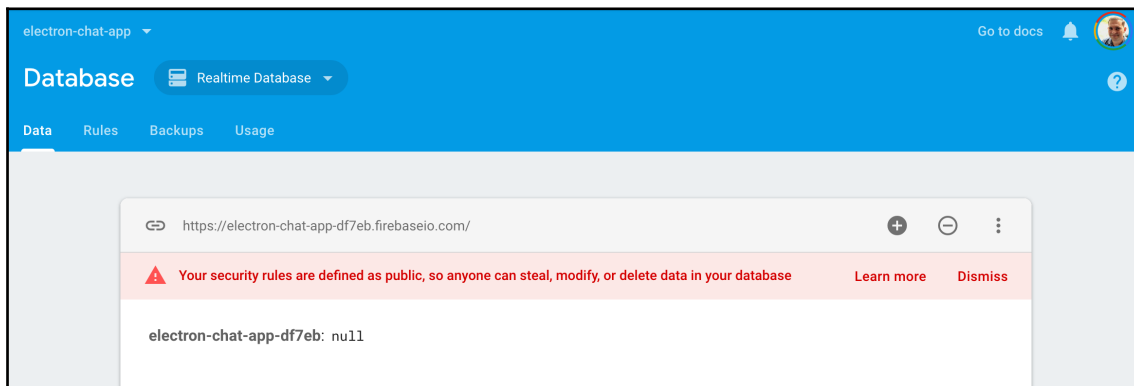
1. Navigate to the Firebase console and click on the **Database** link in the project sidebar. By default, Firebase offers two different tiers of databases: Cloud Firestore and Realtime Database. For this project, you need the Realtime Database.
2. Scroll down until you reach the **Realtime Database** selector:



3. Click the **Create database** button to launch the corresponding dialog.
4. The first thing that Firebase asks you for is the security rule preset. You can choose from the following options:
 - **Locked mode:** This makes your database private by denying all reads and writes.
 - **Test mode:** This allows all reads and writes access to your database.
5. We are going to use Test Mode right now as it is the simplest and quickest way to get your application up and running. However, note that you must provide proper security configuration before launching your application into production.
6. Select the **Start in test mode** option and click the **Enable** button, as shown in the following screenshot:



7. As soon as you click this button, you should see the **Database** page, along with the newly created root object:



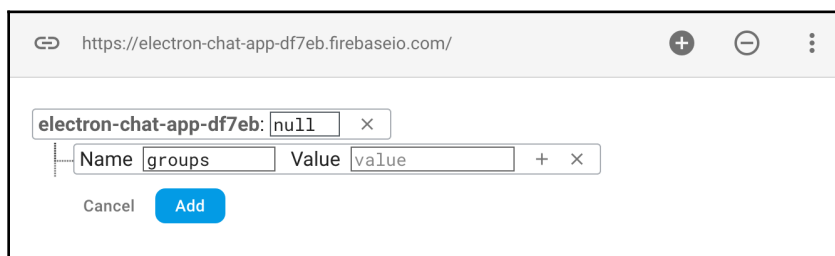
Note how Google Firebase shows you a warning label about security. For now, keep the label on and don't worry about it. This is expected and is an excellent reminder that you need to address the database security settings later on.

Let's create a few chat groups for testing and development purposes.

Creating demo groups

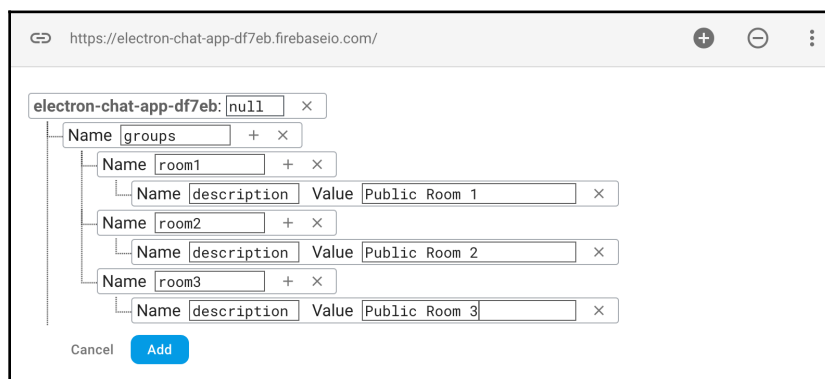
Let's create a few groups by following these steps:

1. Hover over the root object in the central database area until you see the **plus (+)** button. This button allows us to add child properties and complex objects in JSON format.
2. Click the **plus** button. You should see an inline editor for the property:



With the inline editor, we can build a basic tree hierarchy of chat groups or rooms.

3. Create three groups, as shown in the following screenshot:

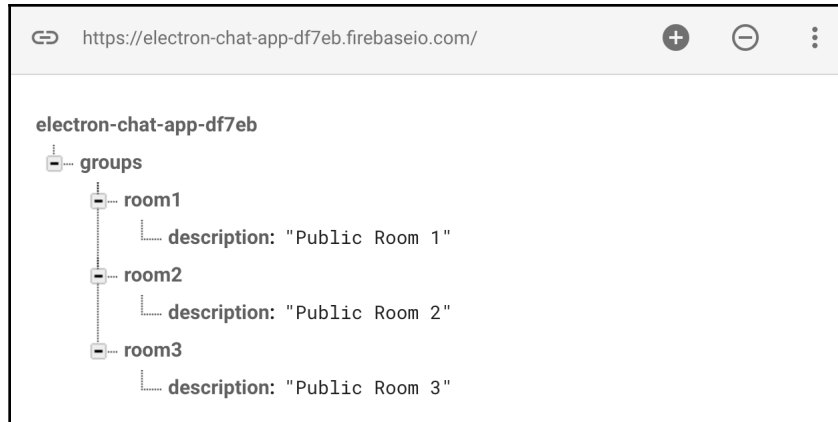


As we can see, we have a groups tree with three branches: room1, room2, and room3. We can also define the description properties so that we can display some user-friendly details on the Angular interface.

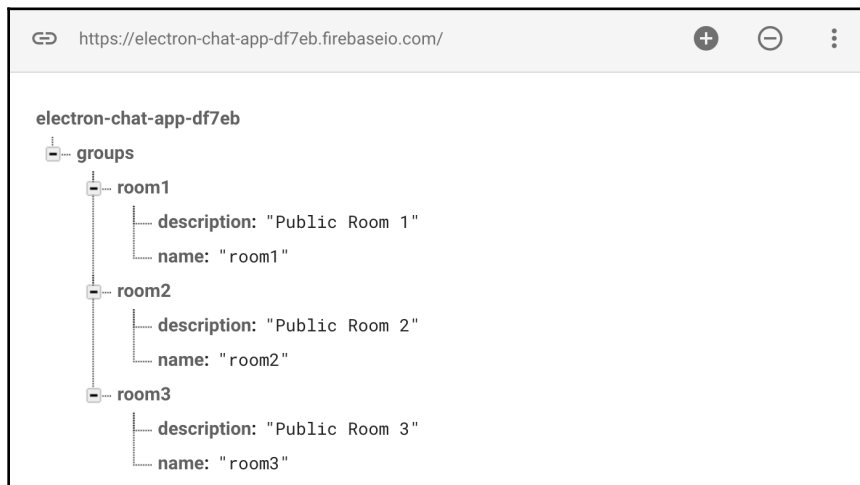


You can introduce more metadata for groups later, such as creation date, logo image, and many more.

4. Click the **Add** button as soon as you are ready. Firebase will apply your changes and render the following tree hierarchy:



5. As a practical experiment, update the data one more time. Add a **name** property to each room, as shown in the following screenshot:



This data is more than enough for us to implement the group list picker in our web interface. Now, let's move on and learn how to this with Angular and Material components.

Rendering the group list

From a web interface perspective, we have a fully functional Login component that navigates users to the `/chat` page, which contains the **chat works!** text label. Let's replace this content with a list of chat groups that users can join:

1. Import the `AngularFireDatabaseModule` module into the main application module using the following code:

```
import { AngularFireDatabaseModule } from '@angular/fire/database';

@NgModule({
  // ...
  imports: [
    // ...
    AngularFireModule.initializeApp(environment.firebaseConfig),
    AngularFireAuthModule,
    AngularFireDatabaseModule
  ],
  // ...
})
export class AppModule {}
```

The preceding code is going to enable additional APIs so that we can communicate with Firebase's databases.

2. Next, import the following classes into the `chat.component.ts` file:

```
import { AngularFireDatabase } from '@angular/fire/database';
import { Observable } from 'rxjs';
```

3. Now, let's introduce the `groups` property, which will hold a list of our group instances. Update the `chat.component.ts` code so that it looks as follows:

```
@Component({...})
export class ChatComponent implements OnInit {
  groups: Observable<any>;

  constructor(private firebase: AngularFireDatabase) {}

  ngOnInit() {
    this.groups = this.firebase.list('groups').valueChanges();
  }
}
```

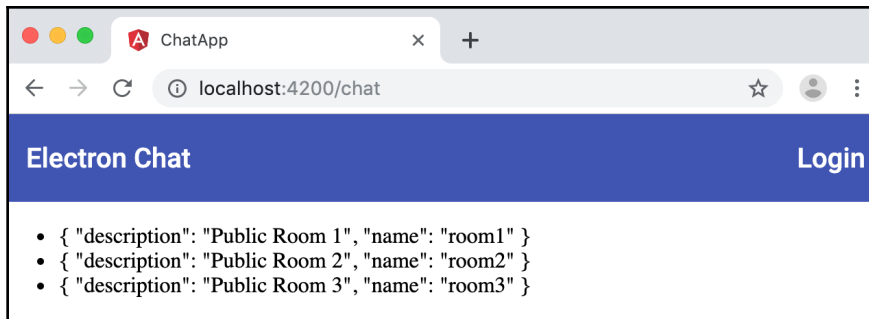
```
    }  
  }  
}
```

The AngularFire library is going to watch these changes and automatically update the collection.

4. Now, we need some HTML templates to render the list items. For testing purposes, let's output the raw JSON content. Update the `chat.component.html` file so that it looks as follows:

```
<ul>  
  <li *ngFor="let group of groups | async">  
    {{ group | json }}  
  </li>  
</ul>
```

5. Save your changes and run or restart the web server. Then, log in to see the `/chat` route. You should see a list of objects that contain name and description properties:



As you can see, we can successfully connect to the Firebase realtime database and display the data.

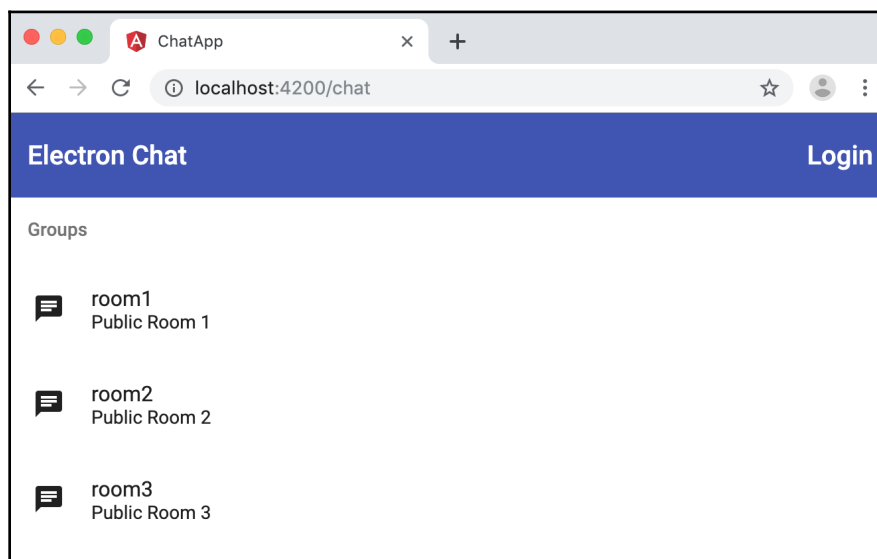
6. Let's make the user interface better by utilizing the Material List component. Import the `MatListModule` and `MatIconModule` modules into the main application module:

```
import { MatListModule } from '@angular/material/list';  
import { MatIconModule } from '@angular/material/icon';
```

7. Replace the content of the `chat.component.html` file with the Material List implementation, as shown in the following code:

```
<mat-list>
  <h3 mat-subheader>Groups</h3>
  <mat-list-item *ngFor="let group of groups | async">
    <mat-icon mat-list-icon>chat</mat-icon>
    <h4 mat-line>{{ group.name }}</h4>
    <p mat-line>{{ group.description }}</p>
  </mat-list-item>
</mat-list>
```

8. Switch to the running web application and check out what the user interface looks like now:



The list of groups now looks perfect. Now, let's learn how the AngularFire library handles real-time updates.

Testing real-time updates

When we defined the `groups` property earlier in this chapter, we used the `valueChanges` API to get an Observable instance. This Observable helps us build a list that reacts to changes. Each time the underlying data changes, the Material List component is going to redraw everything.

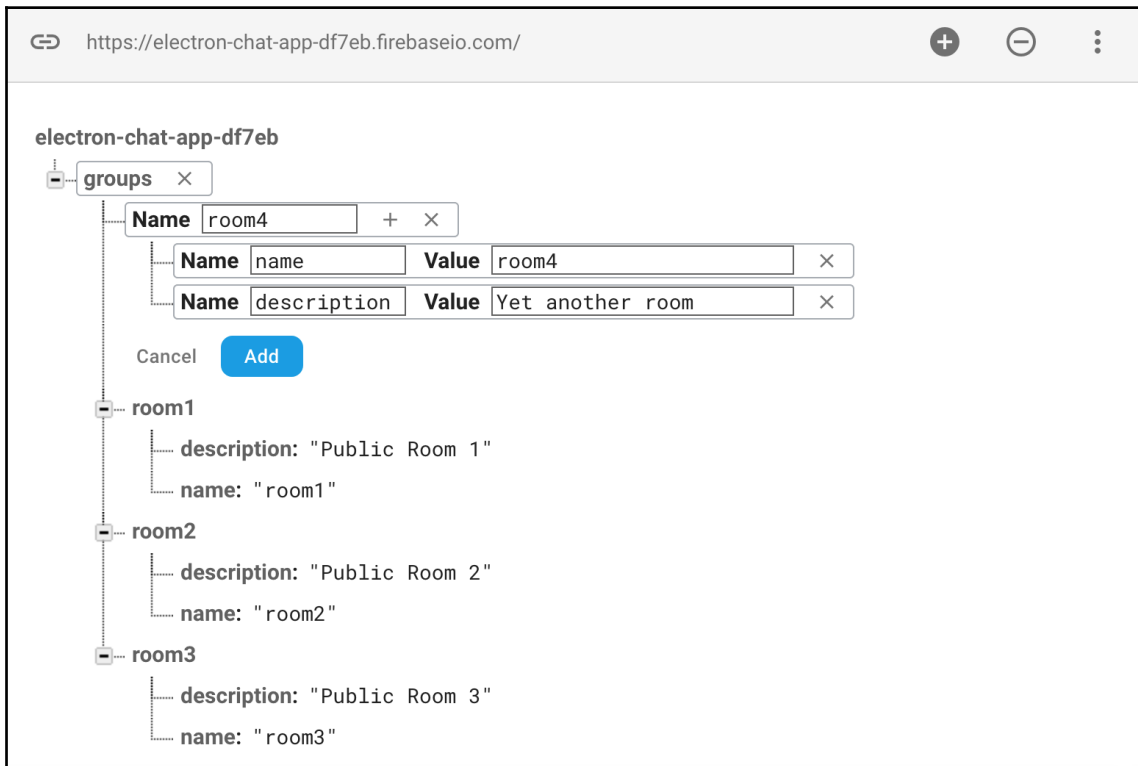
Let's see if that behavior works with our chat application:

1. Run the application and ensure that you are facing the group list.
2. Switch to the Firebase console and navigate to the **Database** section.

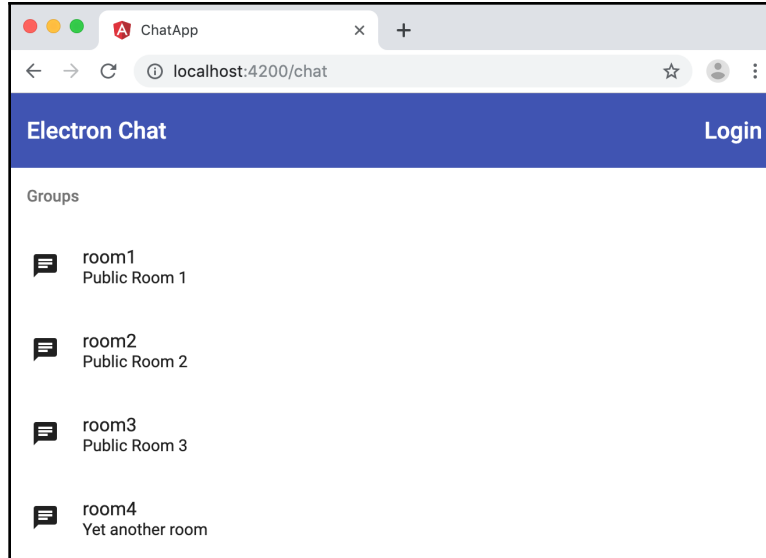


For better visibility, I recommend having both tabs open side by side so that you see the changes we make on both pages simultaneously.

3. Add a new group entry to the list of groups, as shown in the following screenshot:



4. When you are ready, click the **Add** button to confirm these changes. Notice how the **Chats** page updates in real time and that you can see the newly added entry in the list:



As you can see, Firebase provides compelling features for automatic data synchronization across all clients.

In the next section, we are going to make the group list items clickable and redirect users to the corresponding chat room details.

Implementing the group messages page

In this section, we are going to redirect users to a dedicated **Messages** page when they click the group entry. Let's get started:

1. Run the following command to generate a **Messages** component:

```
ng g component messages
```

2. The output of the preceding command should be similar to the following:

```
CREATE src/app/messages/messages.component.scss (0 bytes)
CREATE src/app/messages/messages.component.html (23 bytes)
CREATE src/app/messages/messages.component.spec.ts (642 bytes)
```

```
CREATE src/app/messages/messages.component.ts (278 bytes)
UPDATE src/app/app.module.ts (1477 bytes)
```

3. Next, you need to update the `app-routing.module.ts` file. We are doing this because we need to register a new route that we're going to map to the `MessagesComponent` we have just generated.
4. Let's use a URL path such as `chat/:group/messages`, where `:group` is going to be the name of the chat group that Angular substitutes at runtime.
5. Update the routes collection using the following code:

```
import { MessagesComponent } from './messages/messages.component';

const routes: Routes = [
  {
    path: 'login',
    component: LoginComponent
  },
  {
    path: 'chat',
    component: ChatComponent
  },
  {
    path: 'chat/:group/messages',
    component: MessagesComponent
  }
];
```

6. Now, it's time to update the HTML templates to perform a redirect. Add the `routerLink` directive to the `chat.component.html` template, as shown in the following code:

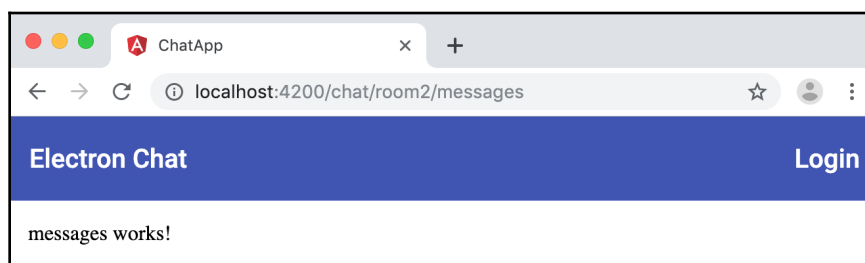
```
<mat-list>
  <h3 mat-subheader>Groups</h3>
  <mat-list-item
    *ngFor="let group of groups | async"
    [routerLink]="[group.name, 'messages']"
  >
    <mat-icon mat-list-icon>chat</mat-icon>
    <h4 mat-line>{{ group.name }}</h4>
    <p mat-line>{{ group.description }}</p>
  </mat-list-item>
</mat-list>
```

7. Let's improve the styling a bit. It would be nice to at least change the cursor so that users understand that the element is clickable and can be used for navigation. Use the following code for the content of `chat.component.scss`:

```
.mat-list-item {  
  cursor: pointer;  
  
  &:hover h4 {  
    text-decoration: underline;  
  }  
}
```

Here, we've changed the cursor to a pointer and also underlined the group name. Feel free to improve the look and feel of the list elements further as you see fit.

8. Restart the application and click on any of the groups. You should end up on the **Messages** page:



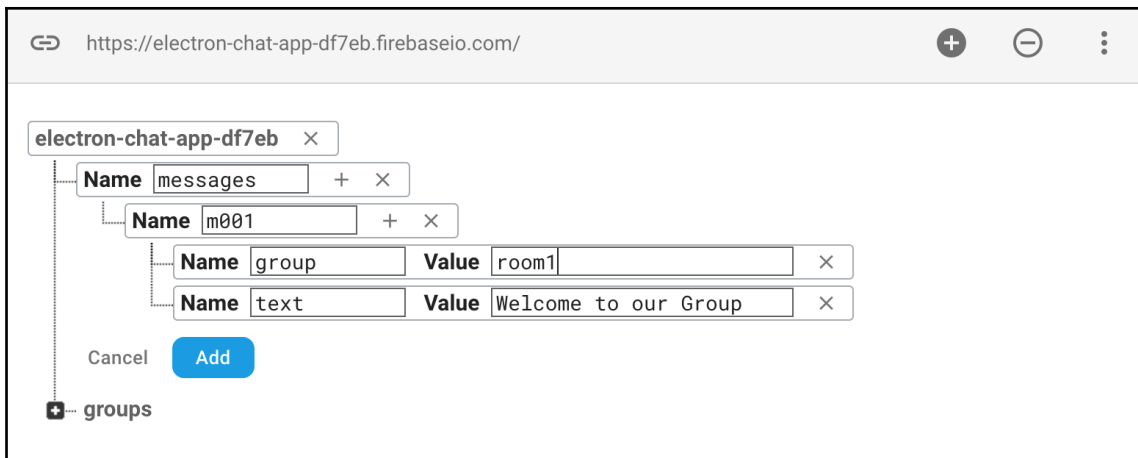
Notice how the browser URL changes to reflect the group name's value. Try doing the same with other groups to ensure that the result is what you expected.

In the next section, we are going to allow users to post and view messages.

Displaying group messages

At this point, we have an initial structure for our groups. Now, we need each of the group entries to contain a list of messages. We don't have support for posting messages to the server, so let's update the database directly and provide some dummy data for one of the groups. We will replace this dummy data with real messages later in this chapter. Let's get started:

1. Switch to the Firebase console and provide the `messages` object for the root entry, as shown in the following screenshot:



As you can see, we are keeping the data in separate branches to simplify real-time access. The `groups` branch contains information about the chat groups, while the `messages` branch stores the actual user messages. Each message object has a reference to the group. This is a very minimalistic implementation and is purely for demonstration purposes.

- Now, it's time to display this message in our Angular application. As you may recall, we have a route template called `chat/:group/messages` where `:group` is dynamic.
- Our next task is to access the `:group` portion of the URL because we need to know the name of the group before asking for the corresponding messages. We can get the value of the `group` parameter using `ActivatedRoute`.
- Update the `messages.component.ts` file so that it contains the following code:

```
import { ActivatedRoute } from '@angular/router';

@Component({...})
export class MessagesComponent implements OnInit {
  group = '';
  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.params.subscribe(params => {
      this.group = params.group;
    });
  }
}
```

5. Next, we need to import `AngularFireDatabase` and `Observable`, similar to what we did with the groups:

```
import { AngularFireDatabase } from '@angular/fire/database';
import { Observable } from 'rxjs';

@Component({...})
export class MessagesComponent implements OnInit {
  messages: Observable<any>;

  constructor(
    private route: ActivatedRoute,
    private firebase: AngularFireDatabase
  ) {}

  ngOnInit() {
    // ...
  }
}
```

This time, however, we should use the `AngularFire` API to filter out the messages that belong to our current group.

6. Update `ngOnInit`, as shown in the following code:

```
ngOnInit() {
  this.route.params.subscribe(params => {
    this.group = params.group;

    if (this.group) {
      this.messages = this.firebase
        .list('messages', ref => ref.orderByChild('group'))
        .equalTo(this.group)
        .valueChanges();
    }
  });
}
```

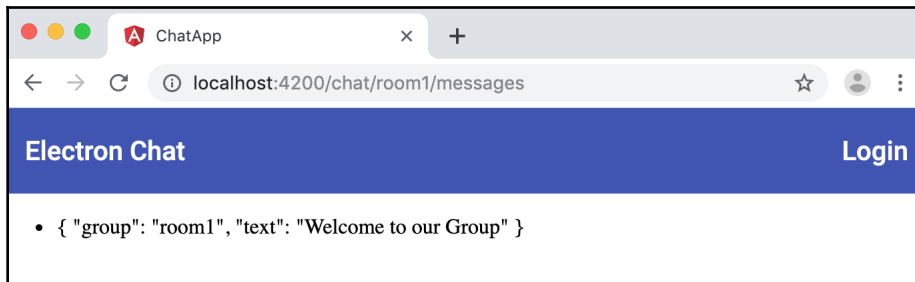


You can find out more about the Querying Lists API at <https://github.com/angular/angularfire2/blob/master/docs/rtdb/querying-lists.md>.

7. Finally, update the HTML template with a simple list element:

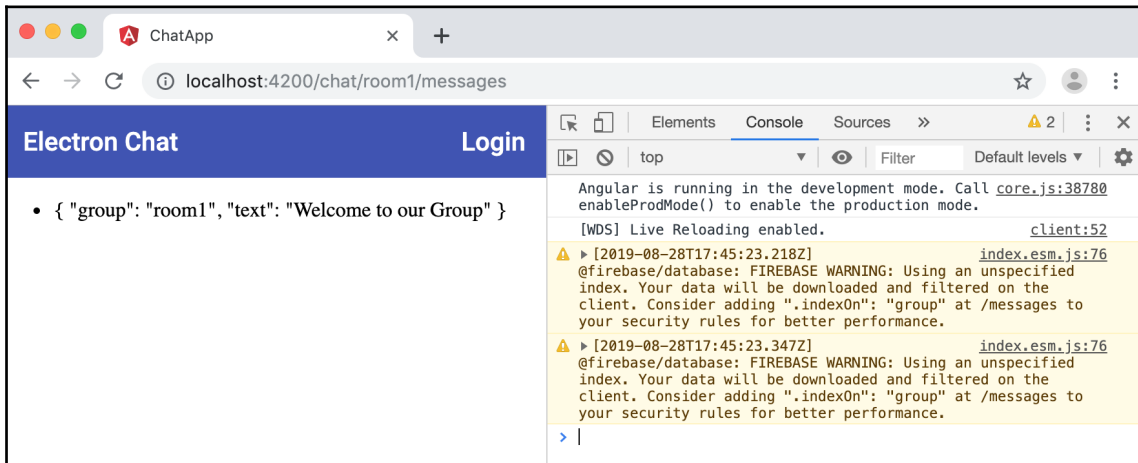
```
<ul>
  <li *ngFor="let message of messages | async">
    {{ message | json }}
  </li>
</ul>
```

8. Reload the page or restart the web server and navigate to the first chat group:



As you can see, the messages are associated with the **room1** group. Congratulations on achieving this significant milestone!

If you open the developer tools, you may see a few warnings about performance:

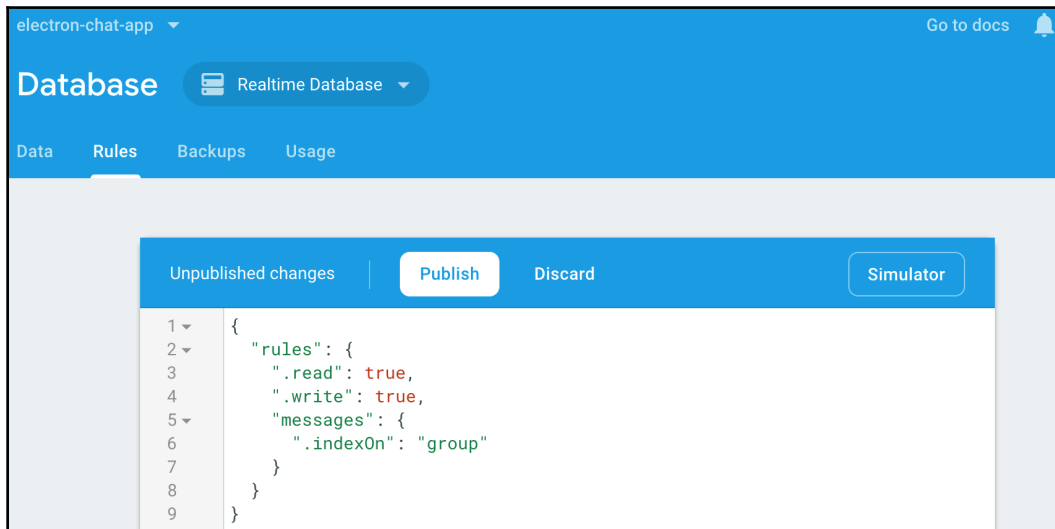


In the next section, we are going to address these performance warnings.

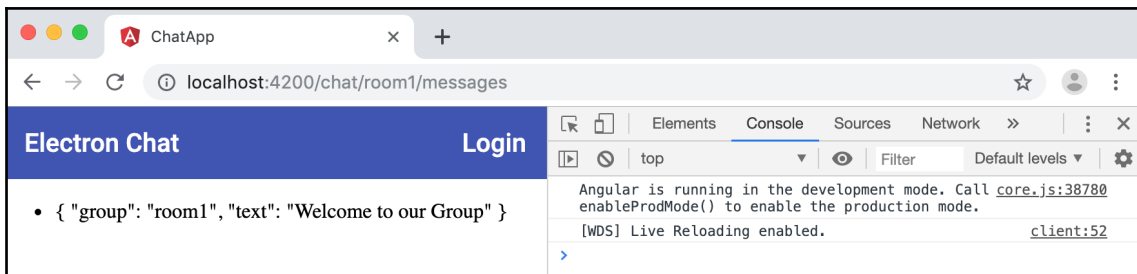
Improving query performance

Let's spend some time fixing our application's potential performance problems. According to the warning, we need to add an index for the `group` field in the `/messages` list. Let's get started:

1. Switch to the Firebase console and click the **Rules** tab of the **Database** section.
2. Update the rules, as shown in the following screenshot:



3. Once you are ready, press the **Publish** button. At this point, Firebase builds an index and updates the data. You don't need to perform any extra steps at the client-side.
4. Reload the page and check out the console's output.
5. This time, you should see no performance warnings:



That's all when it comes to performance tuning our messages list. Now, let's learn how to implement the message editor.

Sending group messages

Now, let's implement some support so that we can send messages to particular groups:

1. Import `FormsModule` into the main application module:

```
import { FormsModule } from '@angular/forms';
```

You are going to need it later to establish two-way binding to the message's text.

2. Let's introduce the message editor. Append the following code to the `messages.component.html` template:

```
<div class="message-editor">
  <mat-form-field class="message-editor-field">
    <input
      [(ngModel)]="newMessage"
      matInput
      placeholder="Message the group"
      autocomplete="off"
      (keyup.enter)="send()"
    />
  </mat-form-field>
</div>
```

Here, we are binding the input element to the `newMessage` property in a two-way fashion, and we're also handling the `keyup.enter` event by calling the `send()` function. We'll come back to these later in this section.

In terms of styling, I suggest at least stretching the input element horizontally so that it takes up all the available space.

3. Update the `messages.component.scss` file so that it matches the following code:

```
.message-editor {
  padding: 0 10px;

  .message-editor-field {
    width: 100%;
  }
}
```


4. Finally, we need to handle the *Enter* key being pressed and send the message to the server. The AngularFire library makes this whole process trivial. Switch to `messages.component.ts` and update the code according to the following listing:

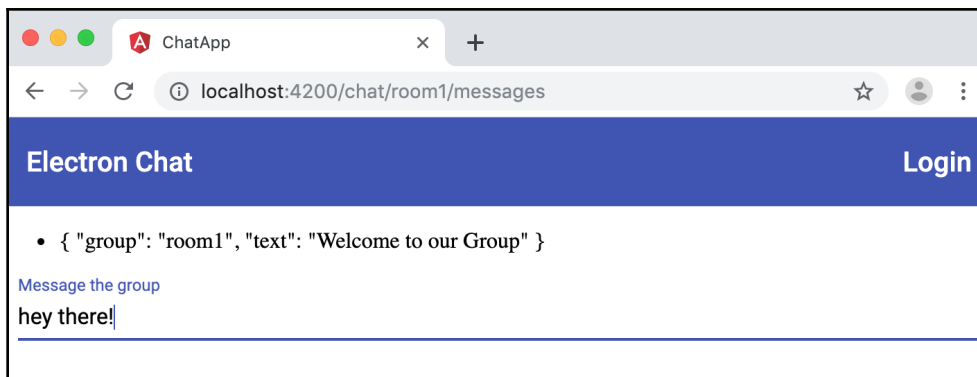
```
@Component({...})
export class MessagesComponent implements OnInit {
  newMessage = '';
  // ...

  send() {
    if (this.newMessage) {
      const messages = this.firebase.list('messages');

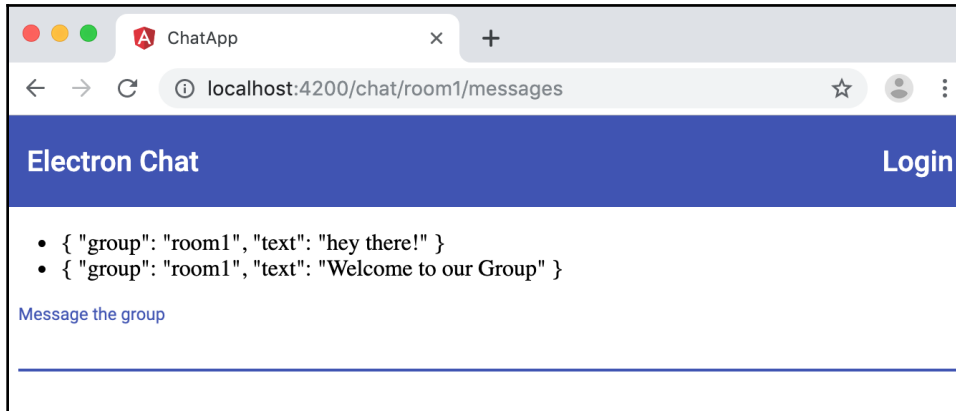
      messages.push({
        group: this.group,
        text: this.newMessage
      });

      this.newMessage = '';
    }
  }
}
```

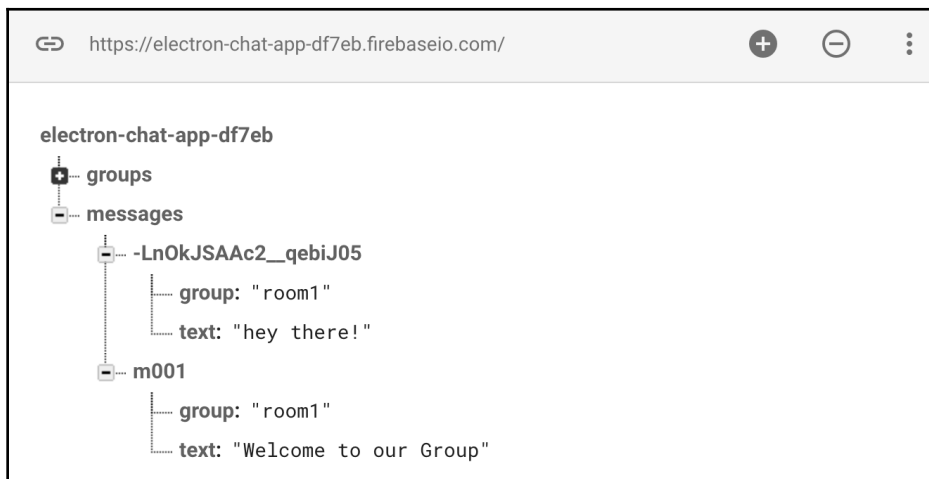
5. We take the `newMessage` value and use it with `group` to compose and send the JSON object to the `/messages` list. Let's see how this works in practice. Switch to the application and write a message, like so:



- Now, press the *Enter* key and ensure that the message is erased from the input element. At the same time, because of the Firebase Realtime Database, our message list is updated from the server:



- If you check the Firebase console, it should contain a new entry:



Try moving to another group and leaving the message there, too. Note that you only see the messages that correspond to the selected group.

I suggest that we move on and update the user interface so that it's more friendly for our users.

Updating the message list interface

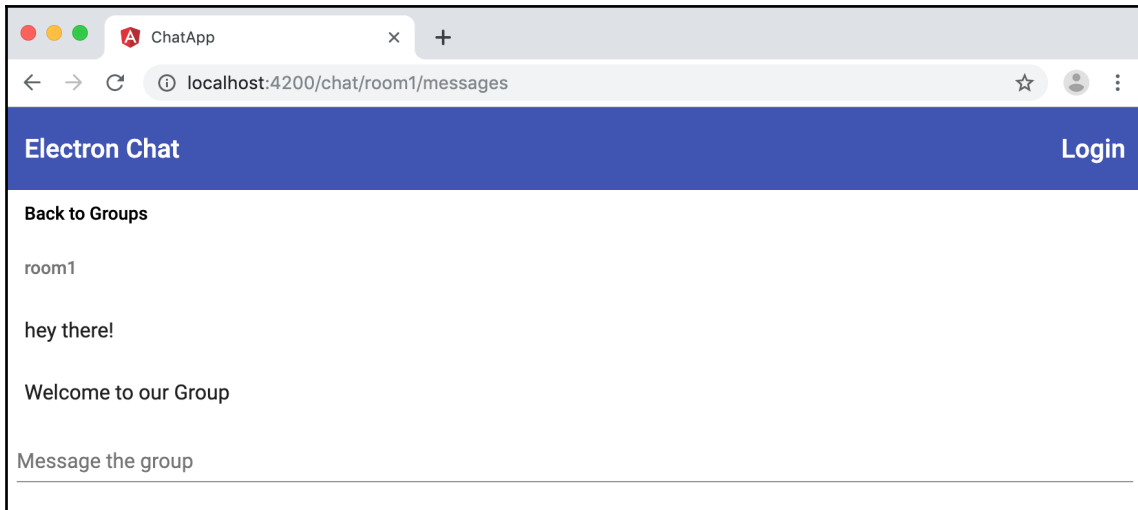
At this point, we are displaying the raw JSON content of the messages. Let's replace this with the Material List. We also need a link so that we can get back to the list of chat groups. Let's get started:

1. Replace the `ul` element in the `messages.component.html` file with the following code:

```
<button mat-button [routerLink]="['/chat']">Back to Groups</button>

<mat-list>
  <h3 mat-subheader>{{ group }}</h3>
  <mat-list-item *ngFor="let message of messages | async">
    {{ message.text }}
  </mat-list-item>
</mat-list>
```

2. The output of the preceding code is as follows:



Congratulations! You can now navigate between groups with ease. The messages list also looks cleaner.

Now, let's take a look at the possible improvements we can make in our spare time.

Ideas for further enhancements

The very first thing you may want to do in your application is sort messages by their creation date. You already know how to filter messages by group name. Feel free to update this code in order to assign the created date property to *now* each time you create and send a new message to the server.

Next, you can update the list query to order messages by date. You can find out more about this at <https://github.com/angular/angularfire2/blob/master/docs/rtdb/querying-lists.md>.

Another essential feature is to preserve the sender's information. You should retrieve the current username from the authentication layer and save it as part of the message object. By doing this, you can display author names in the user interface. Don't hesitate when it comes to experimenting; there are plenty of examples in the Google Firebase and AngularFire documentation.

In the next section, we are going to verify that the application works correctly with the Electron Shell and that it's ready to be packaged.

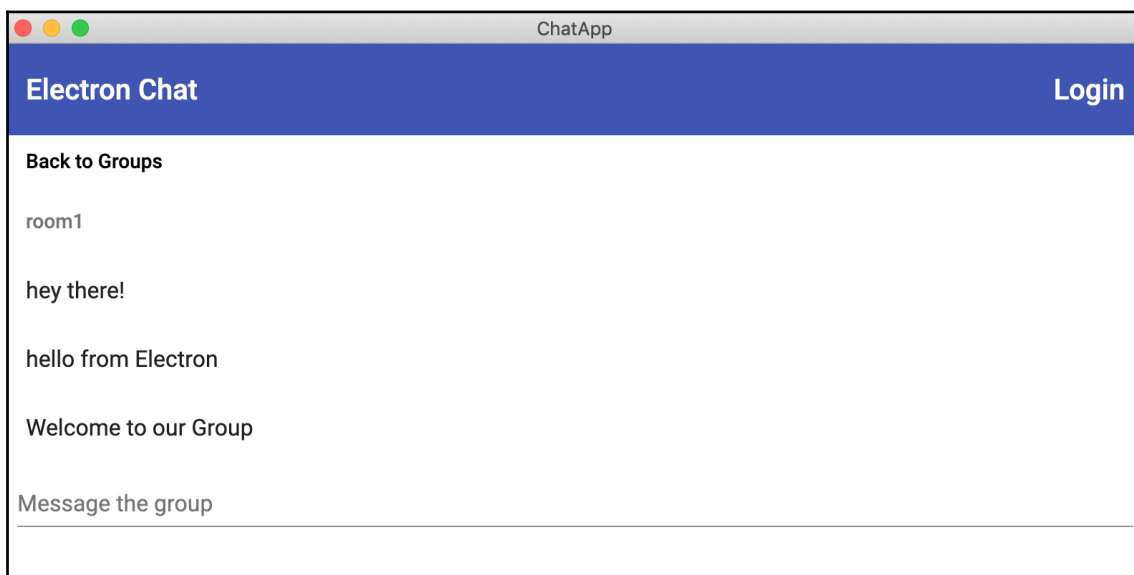
Verifying the Electron Shell

Now, we're going to verify that the application is ready for packaging. Let's get started:

1. Switch to `index.html` and ensure that the base path has a proper value, as shown in the following code:

```
<base href="." />
```
2. Run `npm run serve`. Wait until the web server starts and then run `npm start` in a separate Terminal or Command Prompt window.

3. Test the **Login** dialog with your credentials. Make sure that you can still see the list of groups and that you can post messages:



Congratulations! You have created a basic chat application that you can now expand and improve.

Summary

In this chapter, you became familiar with the Google Firebase service and built a chat application that stores data in a real-time database.

You also learned how to configure databases and build user interfaces that self-update as soon as the database is updated. We created a list of groups/chat rooms that are rendered with the server-side data. Moreover, we have provided support so that messages can be rendered for a particular chat group.

Now, you can set up an authentication layer with a customizable sign-in provider based on email and password credentials.

In the next chapter, we are going to build an eBook editor that we can use to generate books in PDF format.

9

Building an eBook Editor and Generator

In this chapter, we are going to build an Electron application that allows users to author markdown documents and generate electronic books as a result. You are going to learn how to use the Monaco Editor by Microsoft in your Electron applications and how to process text with Pandoc (<https://pandoc.org/>) by utilizing Docker containers.

Why do we need Pandoc as a Docker container rather than as a standalone installation? The typical installation of Pandoc utils is pretty huge, and the setup instructions depend on the type of platform you are using. Moreover, you may need to install various supplementary tools, as well as keep them up to date.

With Docker, you get an image of Pandoc that is universal across all platforms. Besides, the containers are isolated from your operating system. When you don't need Pandoc on your machine, you can delete its containers and images so that you have no unused files in your filesystem.

If you need a new version of the tools, you can fetch a new image from the Docker registry and get some new tools up and running. There's no need for any additional installation steps. The project build time is about two hours.

In this chapter, we will cover the following topics:

- Creating the project structure
- Updating the code to use React Hooks
- Controlling keyboard shortcuts
- Integrating with the application menu
- Setting up the book generator
- Invoking Docker commands from Electron
- Generating PDF books
- Generating ePub books

Let's start our journey by creating the project structure, which is based on the React web application.

Technical requirements

To get started with this chapter, you will need a standard laptop or desktop running macOS, Windows, or Linux.

The software you need to have installed to complete this chapter is as follows:

- Git, a version control system
- Node.js with NPM
- Visual Studio Code, a free and open-source code editor

You can find the code files for this chapter in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter09>.

Creating the project structure

In this section, we are going to create a new project structure and use the React view library to set up the frontend. You are also going to set up the Monaco Editor and its React wrapper component with your Electron application.

We are going to cover the following important aspects when setting up the project:

- Generating a new React application with the official `create-react-app` tool
- Installing the Monaco Editor, which has been developed by Microsoft
- Configuring and testing the web application
- Integrating the Electron shell wrapper and verifying that it works

First, let's learn how to generate a new application scaffold.

Generating a new React application

When creating a new Electron project, the most important part is to decide on the stack and tooling to use. For this project, we are going to use the React view library, as well as the popular `create-react-app` tool, which allows you to generate a full-fledged application structure in a matter of seconds.

Let's generate a new application called `ebook-generator`:

1. Run the following code to generate a new React application:

```
npx create-react-app ebook-generator
```

The output should be similar to the following:

```
Compiled successfully!
```

```
You can now view ebook-generator in the browser.
```

```
Local: http://localhost:3000/
```

```
On Your Network: http://192.168.0.10:3000/
```

```
Note that the development build is not optimized.
```

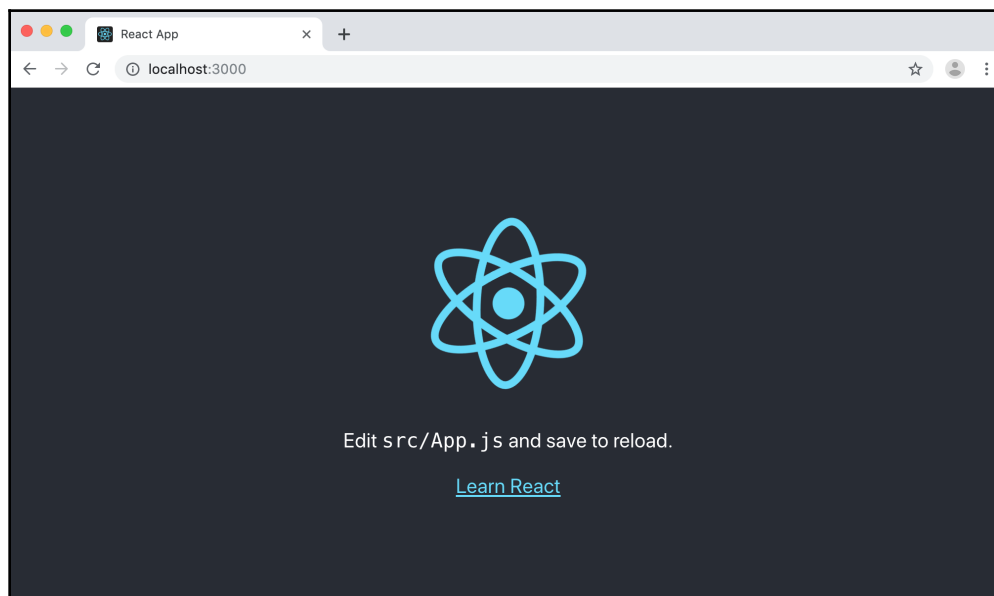
```
To create a production build, use yarn build.
```

2. Switch to the `project` folder and run the project using the following command:

```
cd ebook-generator
```

```
npm start
```

3. If you navigate to `http://localhost:3000`, you should see the following as a result:



Now, let's install the Monaco Editor component and integrate it with our React application for Electron.

Installing the editor component

First of all, to have full integration with the Monaco Editor, we need to `eject` the React application. You can read about the *ejection* process in the React documentation by following this link: <https://create-react-app.dev/docs/available-scripts#npm-run-eject>. Follow these steps to install the editor:

1. Run the following command:

```
npm run eject
```

The tool is going to ask for your confirmation. Press `y` to confirm:

```
$ react-scripts eject
NOTE: Create React App 2+ supports TypeScript, Sass, CSS Modules
and more without ejecting:
https://reactjs.org/blog/2018/10/01/create-react-app-v2.html

? Are you sure you want to eject? This action is permanent. (y/N)
```

2. Now, you need to install the `monaco-editor` library:

```
npm i monaco-editor
```

3. Next, we need the `react-monaco-editor` dependency. This is the component library that provides React bindings for the Monaco Editor component. Run the following command to install the dependency:

```
npm i react-monaco-editor
```

The `monaco-editor` library has a slightly complex configuration. There is a project called `monaco-editor-webpack-plugin` that will save you the time and effort of doing this manually.

4. Install the `webpack` plugin library by executing the following command:

```
npm i monaco-editor-webpack-plugin
```

5. To integrate the `monaco-editor-webpack-plugin` library, you need to update the `config/webpack.config.js` file. Import the `MonacoWebpackPlugin` type at the bottom of the file:

```
const MonacoWebpackPlugin = require('monaco-editor-webpack-plugin');
```

6. Find the `plugins` section of the configuration file. The file is pretty big, so you may want to use the text search functionality. `HtmlWebpackPlugin` should look as follows:

```
495     ],
496   },
497   plugins: [
498     // Generates an `index.html` file with the <script> injected.
499     new HtmlWebpackPlugin(
500       Object.assign(
501         {},
502         {
503           inject: true,
504           template: paths.appHtml,
505         },
506         isEnvProduction
507       ) ? {
508         minify: {
509           removeComments: true,
510           collapseWhitespace: true,
511           removeRedundantAttributes: true,
512           useShortDoctype: true,
513           removeEmptyAttributes: true,
514           removeStyleLinkTypeAttributes: true,
```

7. Insert the new `MonacoWebpackPlugin()` line at the beginning of the `plugins` section, at the top of `HtmlWebpackPlugin`. Please refer to the following screenshot:



```
496 },
497 plugins: [
498   new MonacoWebpackPlugin(),
499   // Generates an `index.html` file with the <script> injected.
500   new HtmlWebpackPlugin(
501     Object.assign(
502       {},
503       {
504         inject: true,
505         template: paths.appHtml,
506       },
507       isEnvProduction
508       ? {
509         minify: {
510           removeComments: true,
511           collapseWhitespace: true,
512           removeRedundantAttributes: true,
513           useShortDoctype: true,
514           removeEmptyAttributes: true,
515           removeStyleLinkTypeAttributes: true,
516           keepClosingSlash: true,
517           minifyJS: true,
```

8. Update the `index.css` file by appending the following code to the bottom of the file:

```
html, body, #root {
  height: 100%;
  width: 100%;
}
```

9. Replace the contents of the `App.css` file with the following code:

```
.App {
  width: 100%;
  height: 100%;
}
```

10. Create a new `Editor.js` file so that we can store our editor component and its configuration settings.
11. Now, import the `MonacoEditor` object from the `react-monaco-editor` namespace:

```
import MonacoEditor from 'react-monaco-editor';
```

In this chapter, we are going to be using a functional React component. Follow these steps to get started:

1. First, put the initial component scaffold inside the `Editor.js` file:

```
import React from 'react';
import MonacoEditor from 'react-monaco-editor';

const Editor = () => {
  return (
    <div></div>
  );
};

export default Editor;
```

We need to provide at least two properties so that we can run the `MonacoEditor` component: the `code` property, which will display the default text inside the editor, and the `options` property, which will provide our configuration details.

2. Create the following constants inside the component function:

```
const Editor = () => {
  const code = '# hello';
  const options = {
    selectOnLineNumbers: true,
    minimap: {
      enabled: false
    }
  };
  return (
    <div></div>
  );
};
```

3. For demonstration and development purposes, let's also add the `editorDidMount` and `onChange` handlers. Update the code inside the component function and include the following functions:

```
const editorDidMount = (editor, monaco) => {
  console.log('editorDidMount', editor, monaco);
  editor.focus();
};

const onChange = (newValue, e) => {
  console.log('onChange', newValue, e);
};
```

As you can see, when `editorDidMount` is fired, we send a message to the console log and focus the editor. Each time the text value is changed, `onChange` executes, and we send event details to the console's output.

4. Let's finish the component implementation by rendering the `MonacoEditor` component with all the necessary properties:

```
return (
  <MonacoEditor
    language="markdown"
    theme="vs-dark"
    value={code}
    options={options}
    onChange={onChange}
    editorDidMount={editorDidMount}
  />
);
```

5. The final code for our `Editor` component looks as follows:

```
1  import React from 'react';
2  import MonacoEditor from 'react-monaco-editor';
3
4  const Editor = () => {
5    const code = '# hello';
6    const options = {
7      selectOnLineNumbers: true,
8      minimap: {
9        enabled: false
10     }
11   };
12
13   const editorDidMount = (editor, monaco) => {
14     console.log('editorDidMount', editor, monaco);
15     editor.focus();
16   };
17
18   const onChange = (newValue, e) => {
19     console.log('onChange', newValue, e);
20   };
21
22   return (
23     <MonacoEditor
24       language="markdown"
25       theme="vs-dark"
26       value={code}
27       options={options}
28       onChange={onChange}
29       editorDidMount={editorDidMount}
30     />
31   );
32 };
33
34 export default Editor;
```

6. As a final step, replace the content of the `App.js` file with the following code:

```
import React from 'react';
import './App.css';
import Editor from './Editor';

function App() {
  return (
    <div className="App">
      <Editor></Editor>
    </div>
  );
}

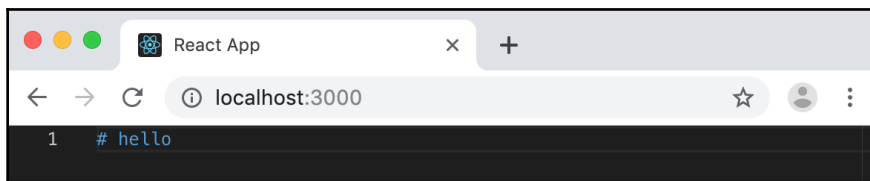
export default App;
```

Now, let's test the application and see the component in action.

Testing the web application

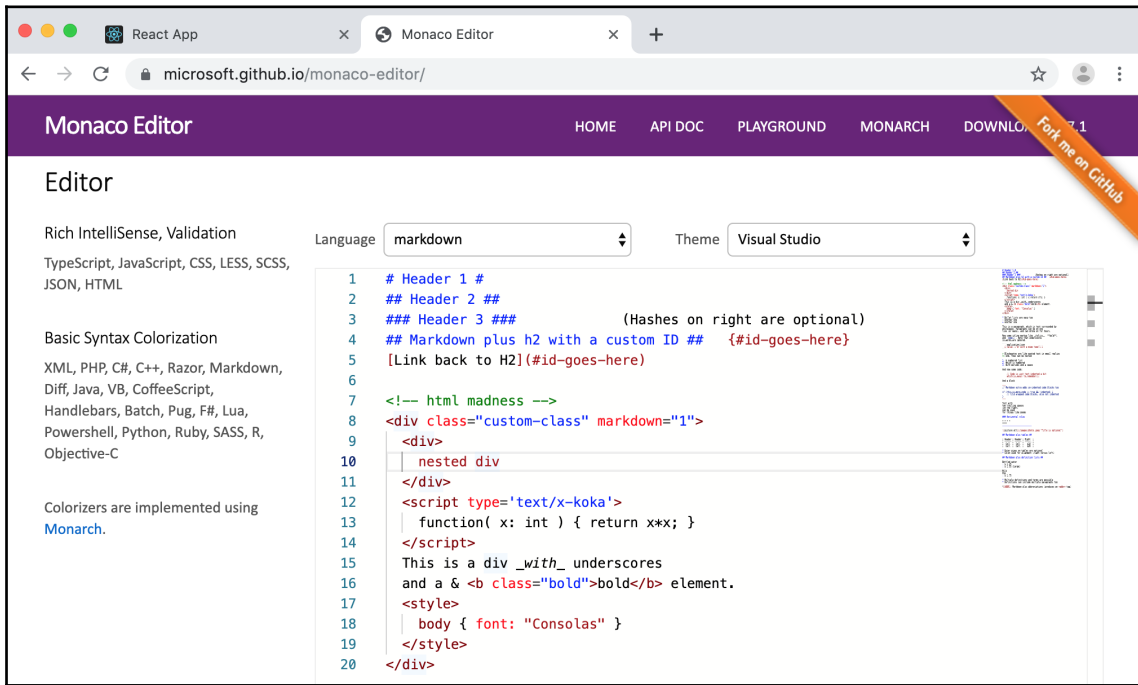
Now, it's time to test the web portion of our application and see how the component behaves. Let's get started:

1. Run the application with the `npm start` command. You should see the following in the window:



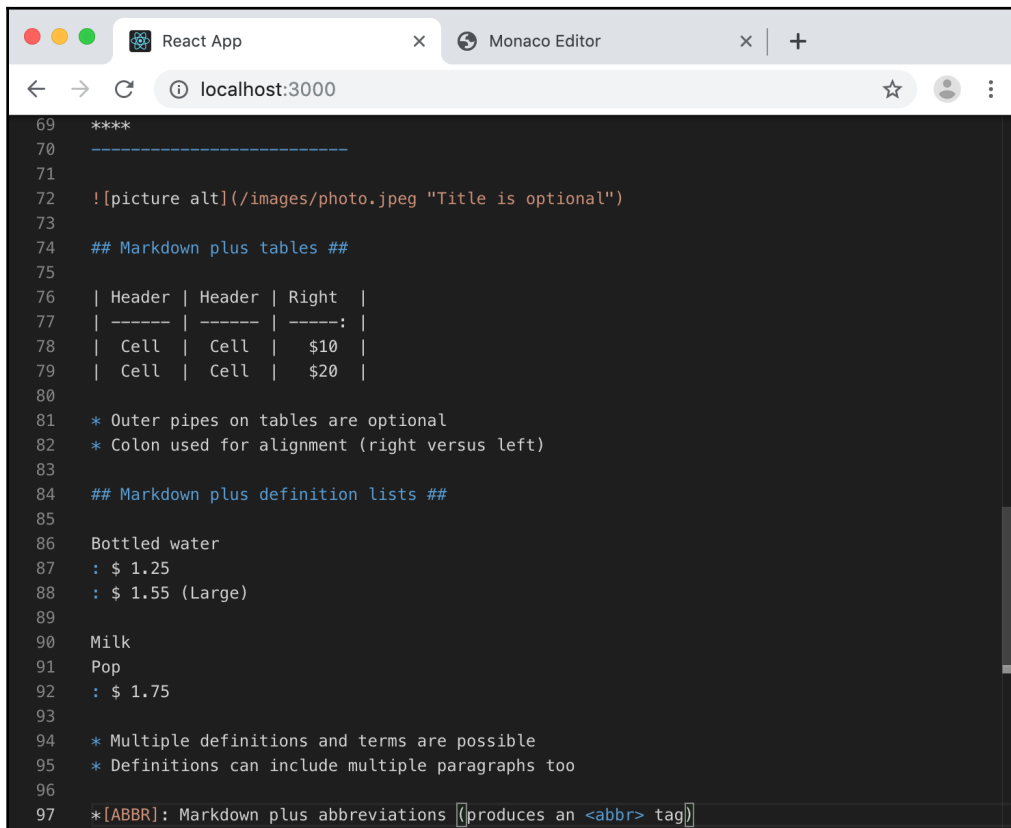
Note that the `# hello` value that we provide from within our `Editor` component is default text.

2. Navigate to <https://microsoft.github.io/monaco-editor/>, as shown in the following screenshot:



3. Select the **markdown** example from the dropdown and copy the content of the example.
4. Switch back to your application and paste the content into the text editing area.

5. Your editor should now look as follows:



```
69 ****
70 -----
71
72 ![picture alt](/images/photo.jpeg "Title is optional")
73
74 ## Markdown plus tables ##
75
76 | Header | Header | Right |
77 | ----- | ----- | -----: |
78 | Cell | Cell | $10 |
79 | Cell | Cell | $20 |
80
81 * Outer pipes on tables are optional
82 * Colon used for alignment (right versus left)
83
84 ## Markdown plus definition lists ##
85
86 Bottled water
87 : $ 1.25
88 : $ 1.55 (Large)
89
90 Milk
91 Pop
92 : $ 1.75
93
94 * Multiple definitions and terms are possible
95 * Definitions can include multiple paragraphs too
96
97 *[ABBR]: Markdown plus abbreviations [produces an <abbr> tag]
```

Notice how the editor formats the text and provides syntax highlighting. This means you have got your component up and running successfully.

Integrating with the Electron shell

Now that you have got a working application, we need to install the Electron dependency and wire the Electron shell window with our code. Let's start by installing the library and updating the package file:

1. Run the following command to install the `Electron` dependency:

```
npm i electron
```


2. Update the `package.json` file and include the `main` property. Ensure that it points to the `main.js` file:

```
{
  "name": "ebook-generator",
  "version": "0.1.0",
  "private": true,
  "main": "main.js",
  // ...
}
```

3. While you have the `package.json` file open, update the `scripts` section and add the `electron` script to invoke the shell:

```
"scripts": {
  "electron": "electron .",
  "start": "node scripts/start.js",
  "build": "node scripts/build.js",
  "test": "node scripts/test.js"
},
```

Now, you should be able to run the web server with the `npm start` command and launch the desktop shell with the `npm run electron` script.

4. Create the `main.js` file and include the following content:

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

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    },
    resizable: false
  });

  win.loadURL(`http://localhost:3000`);
}

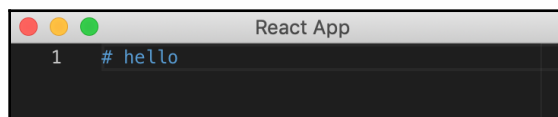
app.on('ready', createWindow);
```

Now our project development environment is ready.

5. Try out the application by running the following commands in parallel console windows:

```
npm start  
npm run electron
```

6. The application window should look as follows:



Given that we have configured the Electron app so that we can connect to `localhost:3000` directly, you also have the live reloading feature up and running as well. Try to update the code and see how the application's content changes inside the Electron window.

In this section, you have learned how to generate a new React project and install the famous Monaco Editor, which also backs Visual Studio Code. You have also configured the project for testing, both in the browser and in the Electron shell.

Now, we need to upgrade our code so that we can use the new *React Hooks* feature and load and save our files.

Updating the code to use React Hooks

Before we move on to keyboard handling, let's refactor our Editor implementation a bit so that we can use React Hooks. We need to do so this so that we can simplify how the code is handled significantly during load and save operations.

React Hooks is a relatively new feature, and if you have a background in React development, then you may have already heard of it or even used it.



Check out the official documentation on React Hooks to find out more:
<https://reactjs.org/docs/hooks-intro.html>.

The most essential hook is the `useState` one. You are going to use it a lot in your projects. Let's import and use the `useState` hook so that we can provide a pair of getters and setters for the code text:

1. Import the `useState` hook from the `react` namespace:

```
import React, { useState } from 'react';
```

2. Replace the code variable initializer with the `useState` hook:

```
// let code = '# hello world';  
const [code, setCode] = useState('# hello world');
```

3. Finally, update the `onChange` handler in order to set the `code` value according to the Monaco Editor's state:

```
const onChange = newValue => {  
  console.log('onChange', newValue);  
  setCode(newValue);  
};
```

Now that we know how to set up and use React Hooks, we are ready to implement keyboard support.

Controlling keyboard shortcuts

With Monaco Editor, you can provide custom commands that handle keyboard combinations. For example, the format of the `Open` command may look as follows:

```
editor.addCommand(monaco.KeyMod.CtrlCmd | monaco.KeyCode.KEY_O, () => {  
  // do something  
});
```



Note that the same command works as *Cmd + O* on macOS and is *Ctrl + O* on other systems, such as Windows and Linux. Apply this logic throughout.

Let's create two stubs for the `Open` and `Save` functions, which will be handled by the `Ctrl + O/Cmd + O` and `Ctrl + S/Cmd + S` commands, respectively:

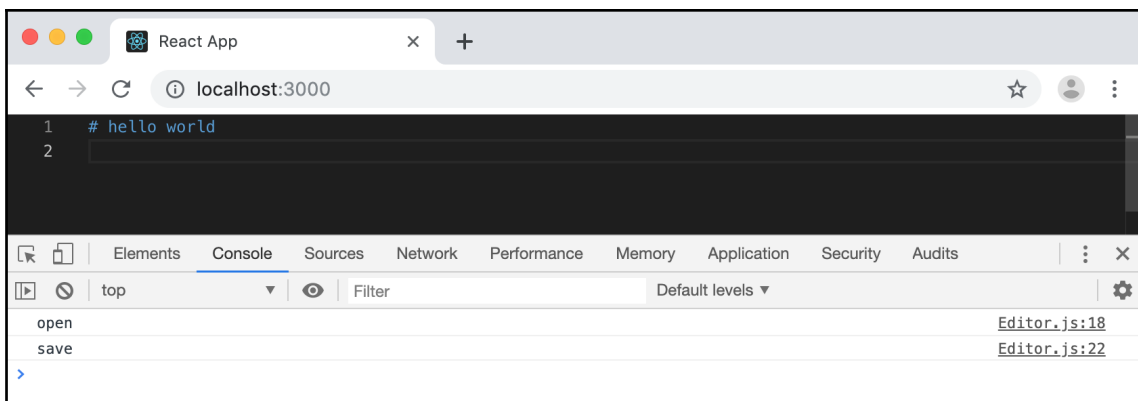
1. In the `Editor.js` file, update the `editorDidMount` handler so that it looks as follows:

```
const editorDidMount = (editor, monaco) => {
  console.log('editorDidMount', editor, monaco);
  editor.focus();

  editor.addCommand(monaco.KeyMod.CtrlCmd | monaco.KeyCode.KEY_O,
    () => {
      console.log('open');
    });

  editor.addCommand(monaco.KeyMod.CtrlCmd | monaco.KeyCode.KEY_S,
    () => {
      console.log('save');
    });
};
```

2. Run the web application and check out the developer tools. You can also do this from within the Electron shell.
3. Try the `Cmd + O` and `Cmd + S` combinations in the editor. Notice how our commands handle the keyboard events and log the messages to the console's output:



Next, let's implement the handler for `Cmd + O` and add support so that we can load files.

Loading files

In this section, we are going to provide support for loading files.



We learned about keyboard handling and Open Dialog in more detail in Chapter 2, *Building a Markdown Editor*.

For the sake of simplicity, instead of redirecting events to Node.js, let's handle loading files via the client side. Follow these steps to do so:

1. Wrap the `MonacoEditor` element with an extra `div` element and the container class name:

```
return (
  <div className="container">
    <MonacoEditor
      language="markdown"
      theme="vs-dark"
      value={code}
      options={options}
      onChange={onChange}
      editorDidMount={editorDidMount}
    />
  </div>
);
```

2. Update the `App.css` file so that it includes the container styling; we need it to take up the entirety of the window:

```
.App,
.container {
  width: 100%;
  height: 100%;
}
```

Traditionally, the programmatic way of triggering the file dialog is to have a hidden `input type=file` element that includes all the necessary settings and then invoke its `click` event.

3. In React, we need to have a reference to the `input` element, so let's define the `fileInputRef` constant. Update the `Editor.js` file and insert the `fileInputRef` constant at the bottom of the body:

```
import React, { useState, useRef } from 'react';

const Editor = () => {
  const [code, setCode] = useState('# hello world');
  const fileInputRef = useRef();

  // ...
}
```

4. Next, we need a hidden `input` element of the `file` type that accepts only text/markdown types. The `input` element should also be wired to the `fileInputRef` constant so that we can trigger its methods from the code. In the return block, place the `input` element as follows:

```
<div className="container">
  <input
    ref={fileInputRef}
    type="file"
    style={{ display: 'none' }}
    accept="text/markdown"
    onChange={onFileOpened}
  ></input>

  <MonacoEditor ... />
</div>
```

As you can see, the `input` element also requires the `onFileOpened` handler.

5. Add the following implementation for the `onFileOpened` function:

```
const onFileOpened = event => {
  if (event.target.files && event.target.files.length > 0) {
    const firstFile = event.target.files[0];

    const fileReader = new FileReader();
    fileReader.onload = e => setCode(e.target.result);
    fileReader.readAsText(firstFile);

    event.target.value = null;
  }
};
```

The code is pretty self-explanatory. We take the first file and use the `FileReader` API to get its content as text. As soon as the file's content has been loaded, we call the `setCode` hook to update the state of the code.

We also have the `code` hook, which is bound to the `MonacoEditor.value` property. This means that as soon as we set the value via the `setCode` hook, the Monaco Editor gets these changes and updates itself.

6. The last piece of the puzzle is to wire `Cmd + O` (or `Ctrl + O`) with the dialog. Update the `editorDidMount` implementation to invoke `click`, as shown in the following code:

```
editor.addCommand(monaco.KeyMod.CtrlCmd |  
monaco.KeyCode.KEY_O, () => {  
  fileInputRef.current.click();  
});
```

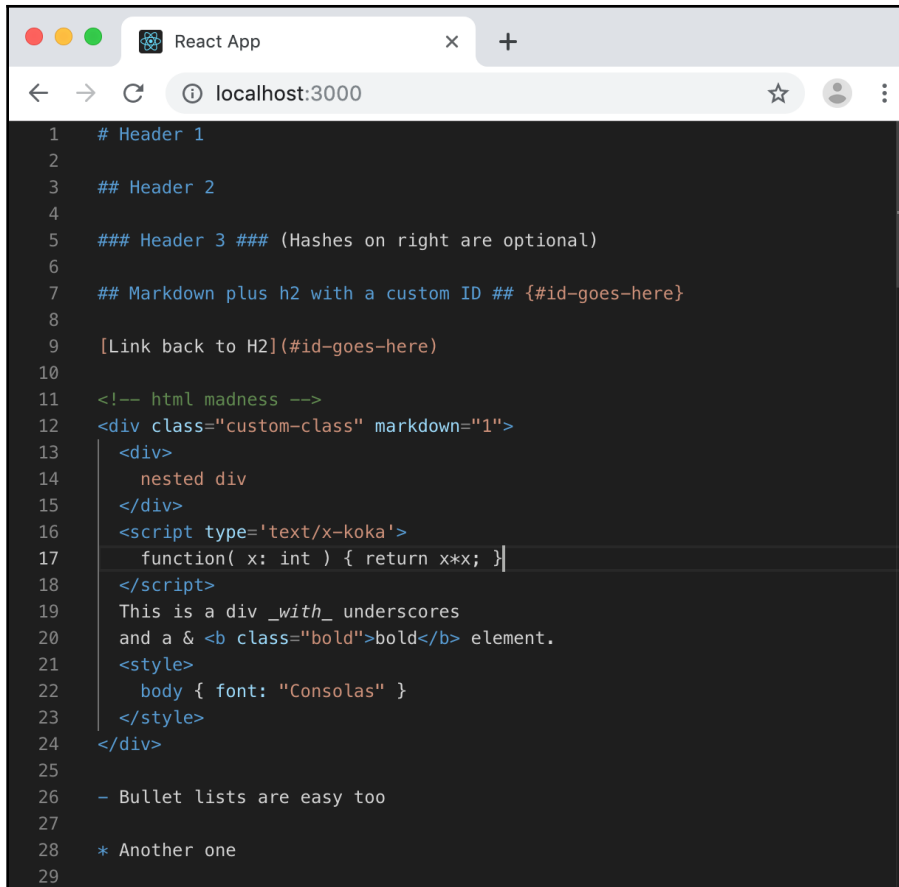
That's all you need to do to load a file and run it. Now, let's test the implementation:

1. Start the web application with the `npm start` command.
2. Press `Cmd + O` if you're using macOS; otherwise, press `Ctrl + O`.
3. In the resulting dialog, select a markdown file.



I have saved an example from the Monaco Editor demo (<https://microsoft.github.io/monaco-editor/>). You can find any markdown file on GitHub, such as a `README.md` file, and save it locally.

4. Ensure that the editor loads and displays the file correctly:



```
1  # Header 1
2
3  ## Header 2
4
5  ### Header 3 ### (Hashes on right are optional)
6
7  ## Markdown plus h2 with a custom ID ## {#id-goes-here}
8
9  [Link back to H2]({#id-goes-here})
10
11 <!-- html madness -->
12 <div class="custom-class" markdown="1">
13   <div>
14     nested div
15   </div>
16   <script type='text/x-koka'>
17     function( x: int ) { return x*x; }
18   </script>
19   This is a div _with_ underscores
20   and a & <b class="bold">bold</b> element.
21   <style>
22     body { font: "Consolas" }
23   </style>
24 </div>
25
26 - Bullet lists are easy too
27
28 * Another one
29
```

Next, let's implement the handler for `Cmd + S` and add support so that we can save files.

Saving files

In this section, we are going to provide support for saving files. Similar to the `Open` file functionality, web developers need to perform some easy workarounds to invoke the file download feature from the code.

The most popular way to do this is to dynamically create an invisible hyperlink element with the `download` attribute set to the filename and then invoke a click. This is what we are going to implement for our project. Follow these steps to get started:

1. Update the code in the `Editor.js` file and add the `saveFile` function, as shown in the following code:

```
const saveFile = contents => {
  const blob = new Blob([contents], { type: 'octet/stream' });
  const url = window.URL.createObjectURL(blob);

  const a = document.createElement('a');
  document.body.appendChild(a);
  a.style.display = 'none';
  a.href = url;
  a.download = 'markdown.md';
  a.click();

  window.URL.revokeObjectURL(url);
  document.body.removeChild(a);
};
```

We have just created a helper function, `saveFile(contents)`, that takes a string value as input and invokes a file download called `markdown.md`.

To make the `saveFile` function work, we are going to use the `URL.createObjectURL` API, which is supported by all web browsers. This static method allows us to create a URL object with the content of the file embedded into it. Then, you pass that URL to a dynamic hyperlink element and invoke the click event.

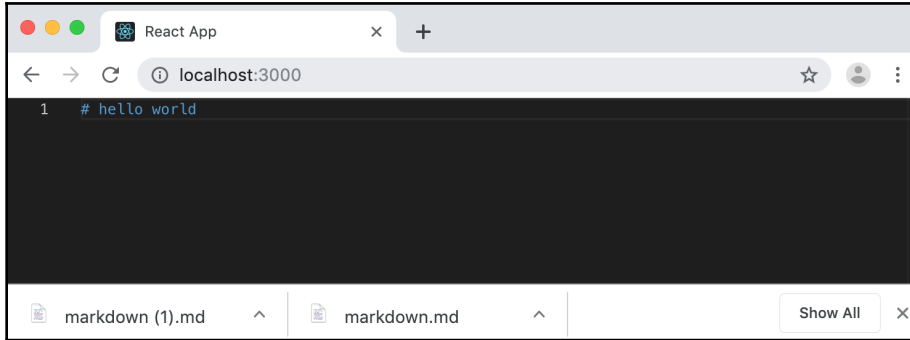


You can find out more about the `URL.createObjectURL` static method at <https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL>.

2. Now, it's time to reuse the function we have just created. Update the `editorDidMount` function in order to invoke the `saveFile` upon clicking `Cmd + S` (or `Ctrl + S`):

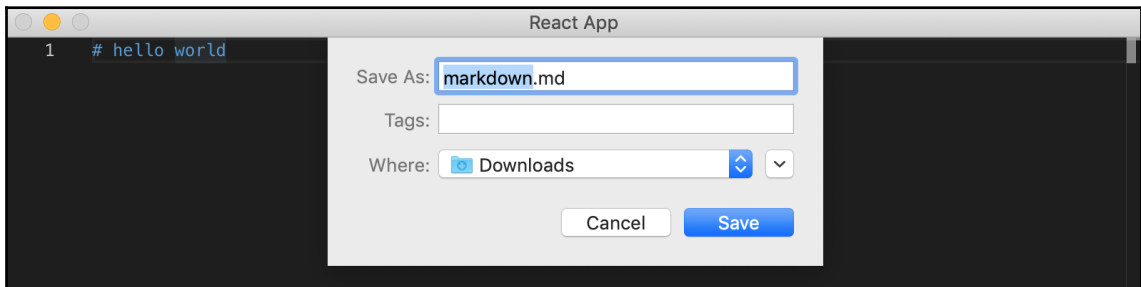
```
editor.addCommand(monaco.KeyMod.CtrlCmd | monaco.KeyCode.KEY_S, ()
=> {
  const code = editor.getModel().getValue();
  saveFile(code);
});
```

3. Run the application and try saving the file with the keyboard combinations we have just provided. Notice that the browser automatically changes its name if you download the file more than once:



This is the traditional behavior of browsers.

There is a difference, however, if you run the same code in the Electron shell. By default, Electron asks you where you want to put the file upon clicking *Cmd + S* (or *Ctrl + S*):



At this point, you've got the *Save* and *Open* functionality working. This should allow you to work on multiple documents and make backup copies of the files.

Next, we will introduce application menu integration.

Integrating with the application menu

In this section, we are going to provide support for the application menu. For the sake of simplicity, let's integrate the `Open` and `Save` features for now and extend the menu as we introduce new features to the application:



You can find out more about how to work with menus in [Chapter 2, Building a Markdown Editor](#). Examples are also provided.

1. In the project root folder, create a `menu.js` file with the following content:

```
const { Menu, BrowserWindow, dialog } = require('electron');
const fs = require('fs');

module.exports = Menu.buildFromTemplate([
  {
    label: 'File',
    submenu: [
      {
        label: 'Open',
        accelerator: 'CommandOrControl+O',
        click() {
          loadFile();
        }
      },
      {
        label: 'Save',
        accelerator: 'CommandOrControl+S',
        click() {
          BrowserWindow.getFocusedWindow().webContents.send('commands', {
            command: 'file.save'
          });
        }
      }
    ]
  }
]);
```

As you can see, we're declaring a `File` menu (the application menu on macOS) with `Open` and `Save` entries. Each menu item does nothing except send a JSON payload to the client-side code.

To make processing easy and universal, we've introduced a convention—each payload has the `command` parameter, along with the enclosed key of the action the client needs to execute. To open files, we're using the `file.open` key, while to save files, we're using the `file.save` key.



Later, we can introduce even more commands. The command handlers at the client side won't require rewrites or significant refactoring.

2. Add the `loadFile` function implementation, as shown in the following code:

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

We invoke the native Open Dialog, read the file, and send it back to the client-side code with the `file.open` command and a `value` property.



We need a native dialog here because modern browsers don't allow us to invoke file-related operations when the code isn't related to user interaction. Unfortunately, sending a message from the Node.js process to the Chrome-based process isn't going to work for the `input type=file` elements due to security reasons. This is why we use native code in the main process and provide the renderer process with the result.

3. To enable the application menu that we have just declared in the `menu.js` file, we need to update the `main.js` file. Import `Menu` from the `electron` namespace and use it to build the custom menu instance, as shown in the following code:

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

Menu.setApplicationMenu(menu);

function createWindow() {
  // ...
}

app.on('ready', createWindow);
```

4. At the client side, we need to implement a universal command handler function. Update the `Editor.js` file and add the `handleCommand` function, as shown in the following code:

```
const handleCommand = payload => {
  if (payload) {
    switch (payload.command) {
      case 'file.open':
        setCode(payload.value || '');
        break;
      case 'file.save':
        saveFile(code);
        break;
      default:
        break;
    }
  }
};
```

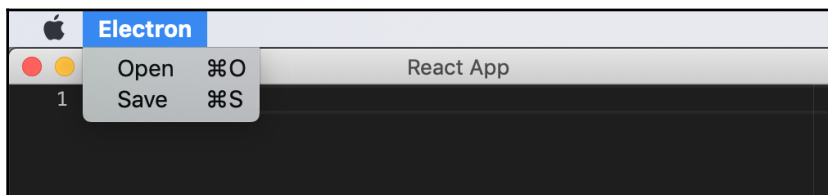
The preceding code should be easy to understand. Upon using the `file.open` command, we get `payload.value` and pass it to the `setCode` hook. Also, once the `file.save` command arrives, we invoke the `saveFile` function.

5. Now, we need to update the Editor component function so that we can handle the commands. I suggest adding some safety checks to make the code compatible with both browsers and Electron applications. Append the following function after the `handleCommand` one:

```
if (window.require) {  
  const electron = window.require('electron');  
  const ipcRenderer = electron.ipcRenderer;  
  
  ipcRenderer.on('commands', (_, args) => handleCommand(args));  
}
```

The preceding code is an excellent example of cross-application compatibility. Upon running the code in the Electron shell, the code finds the `window.require` object and performs additional configuration. If you ever decide to run your application with regular browsers, where `window.require` is missing, the code isn't going to break.

6. Run the Electron shell with the `npm run electron` command and check out the application menu:



7. Click the **Open** menu item and check that it loads markdown files into the editor correctly.

Congratulations on reaching the end of this section. Now, you are able to send commands from the application menu back to the client-side code.

Now, let's learn how to generate a digital book out of the markdown's content.

Setting up the book generator

In this section, we are going to prepare the environment for PDF and eBook generation. We are going to install Docker, a famous containerization software that allows us to run applications as containers across all platforms.

Once you have Docker up and running, we will walk through the process of downloading and running Pandoc, a universal document converter tool, as a container on our local machine.

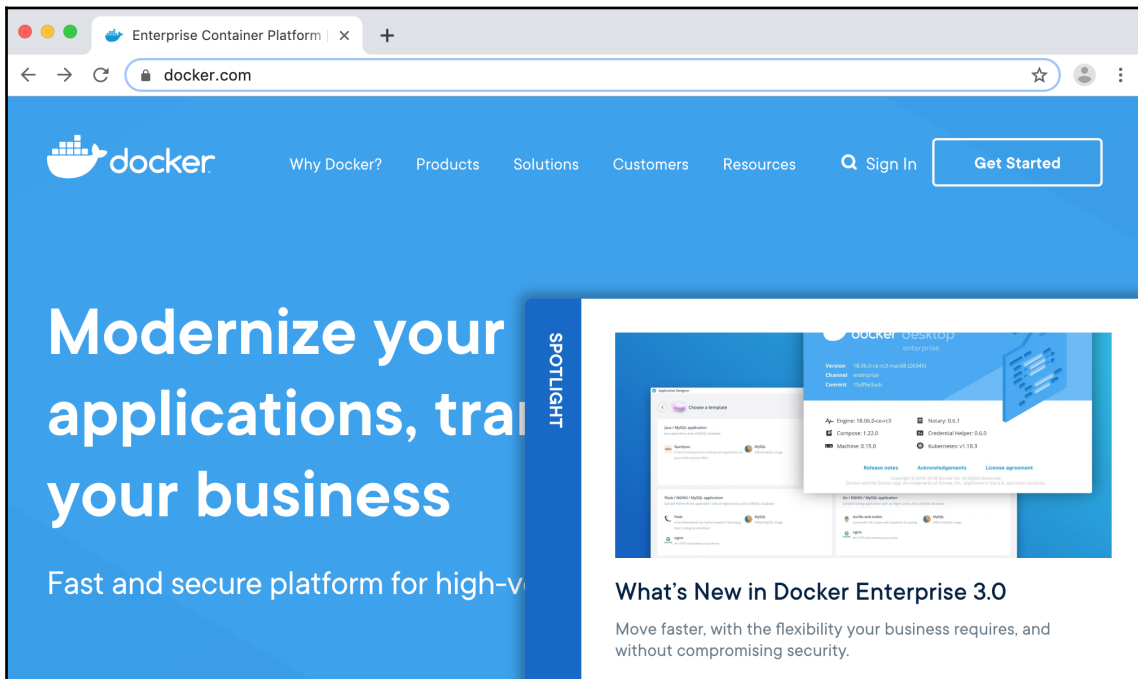
The Pandoc tool allows us to convert a massive number of different text formats into another format. Among the supported formats is markdown, which is what we are using for our Electron editor application.

Let's start with the Docker installation process.

Installing Docker

In this section, you are going to install Docker for desktop. There are two versions of Docker: Enterprise and Community. The Community version is free and is more than enough for development purposes as it handles all our scenarios perfectly well. Let's get started:

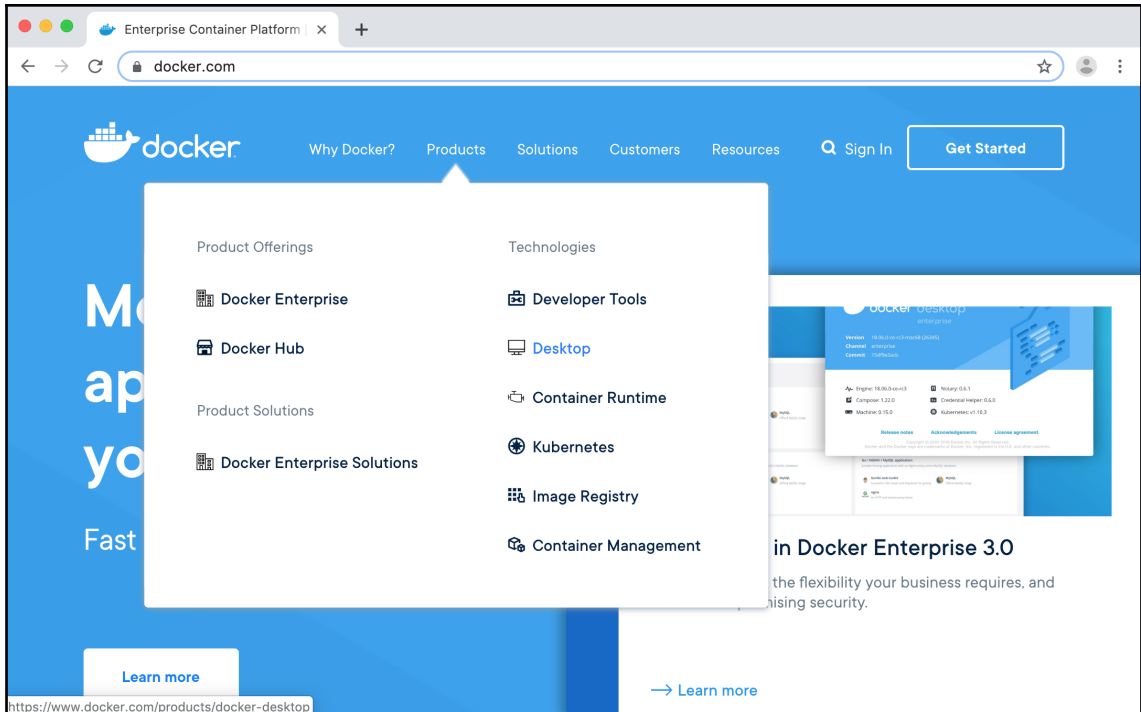
1. Navigate to the official website (<https://www.docker.com/>). You should see a landing page that looks similar to the one shown in the following screenshot:





Feel free to explore the website and read the documentation and guides on Docker if you've never used it before.

2. Scroll to the top of the page and click on **Products**. Here, **Desktop** should be one of the available options:



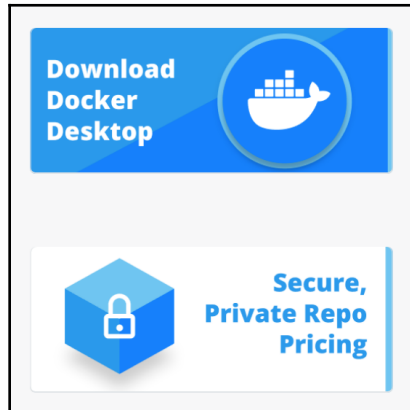
3. Click **Desktop**.



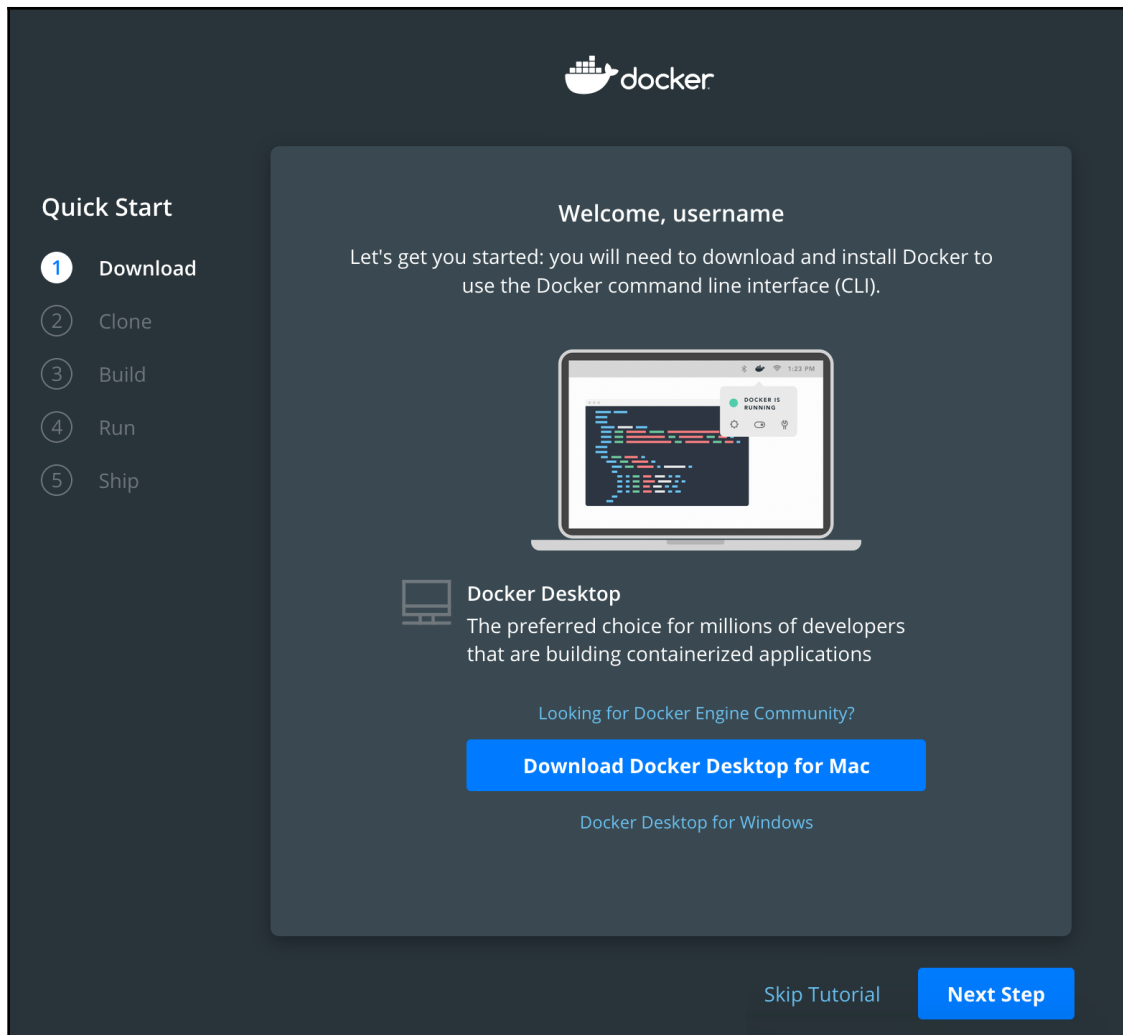
Alternatively, you can navigate to the following address: <https://www.docker.com/products/docker-desktop>.

On the **Docker Desktop** page, you can either watch an overview video or download the installation package.

4. Click the **Download Desktop for Mac and Windows** button. Note that the button name is slightly misleading. Typically, you should be redirected to the Docker Hub portal. If you don't have an account yet, please create one—it's free.
5. As soon as you log into the Docker Hub, you will find the real download button:

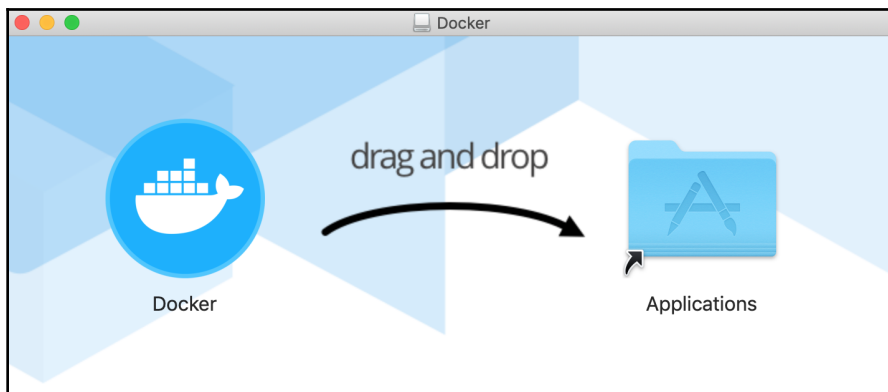


6. In the **Quick Start** dialog, you will find the download links for macOS and Windows:



You can also use the direct links for macOS (<https://download.docker.com/mac/stable/Docker.dmg>) and Windows (<https://download.docker.com/win/stable/Docker%20for%20Windows%20Installer.exe>).

In my case, I have downloaded and launched the installer for macOS:



Please follow the instructions for your operating system and refer to the online documentation if you have any questions regarding the setup and configuration of Docker Community Edition.

Next, we are going to test the Pandoc container and see it in action.

Running the Pandoc container

Given that anyone can create a Docker image with most applications and tools, you will find more than one Pandoc implementation on the internet. I have prepared a stable example in this book's GitHub repository: <https://github.com/DenysVuika/pandoc-docker>.



The image for this section was initially forked from <https://github.com/jagregory/pandoc-docker>, so credit goes to the author, James Gregory. In my version, I am keeping the base image and Pandoc libraries up to date.

Let's learn how the conversion process works:

1. Start Docker.
2. Navigate to the Monaco Editor demo page (<https://microsoft.github.io/monaco-editor/>) and copy the contents of the **markdown** example.
3. Save the markdown example somewhere on your local drive with the name `test.md`.
4. In the Terminal window, navigate to the location of the `test.md` file and run the following command:

```
// for Linux and macOS
docker run -v `pwd`:/source denysvuika/pandoc -f markdown -t html5
test.md -o test.html

// for Windows
docker run -v %cd%:/source denysvuika/pandoc -f markdown -t html5
test.md -o test.html
```

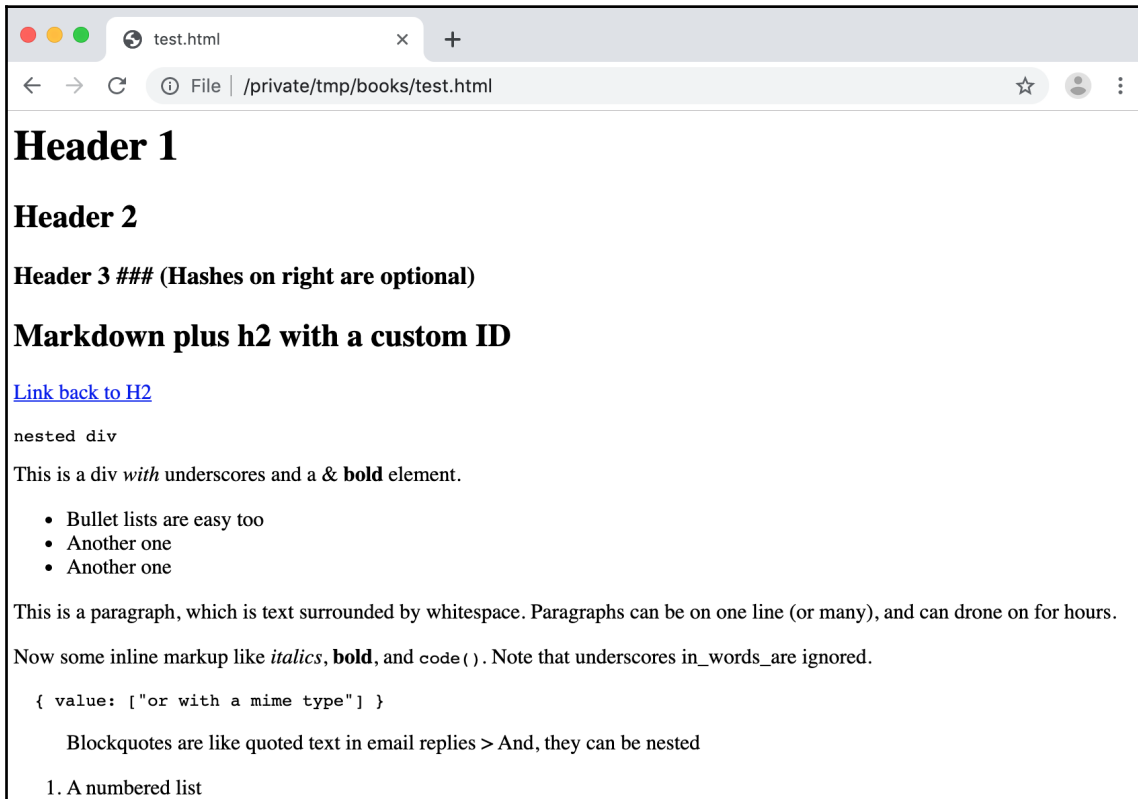
Here, we're taking `test.md` as input and using the `denysvuika/pandoc` image to convert it into an HTML5 page called `test.html`. After a while, you should see the following output:

```
Unable to find image 'denysvuika/pandoc:latest' locally
latest: Pulling from denysvuika/pandoc
bc9ab73e5b14: Pull complete
d553ba08f210: Pull complete
a5e51e378eb4: Pull complete
858ca3975bae: Pull complete
c3ecb06ceeb4: Pull complete
Digest:
sha256:010d68dcc6a3de0a8ca2a6b812ccd5be16b515524270fb4996413990a6e5
0776
Status: Downloaded newer image for denysvuika/pandoc:latest
```



The first run may take a while because of Docker downloading and caching the images. All subsequent runs are usually instant.

5. Now, if you check the output folder, you should see the `test.html` file. Open the file in your preferred browser and check its contents:



You have successfully generated HTML output from the markdown file. Feel free to experiment with changing the content of the markdown file and converting it to HTML5. Check out all the possible conversion scenarios at <https://pandoc.org/>.

Next, we need to find out how we can run the same command from within our Electron application. However, before doing that, we need to update the save process so that we can use the Node.js process.

Sending documents to the main (Node.js) process

In this section, we're going to send the document to the main process. Follow these steps to do so:

1. Import `ipcMain` into the `menu.js` file, as shown in the following code:

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

Now, we need to implement the `saveFile` function. You may have done this already in Chapter 2, *Building a Markdown Editor*.

2. Add the following code either at the beginning or at the bottom of the `menu.js` file:

```
function saveFile(contents) {  
  const window = BrowserWindow.getFocusedWindow();  
  const options = {  
    title: 'Save markdown file',  
    filters: [  
      { name: 'MyFile', extensions: ['.md'] }  
    ]  
  };  
  dialog.showSaveDialog(window, options, filename => {  
    if (filename) {  
      fs.writeFileSync(filename, contents);  
    }  
  });  
}
```

The `saveFile` function expects a `contents` parameter, which is where we pass the contents of the markdown file. After that, it displays the system `Save Dialog`, where you can pick the destination and save the file to our local drive.

We are going to listen to the `save` channel so that we can invoke the `saveFile` feature. The renderer (Chrome) part should now send the markdown's contents to the `save` channel to initiate the save dialog.

3. Update the `menu.js` file with the listener code, as follows:

```
ipcMain.on('save', (_, contents) => {  
  saveFile(contents);  
});
```

4. Now, you need to update the client-side part. At this point, you have the option to either remove the previous code or have a dual behavior. With the dual behavior, you can save the file using a regular browser. Alternatively, you can send the file's content to Node.js when you're running the Electron shell.

The code for the dual behavior can be created with a simple `if... else` statement, as follows:

```
const saveFile = contents => {  
  if (window.require) {  
    // send to the node.js  
  } else {  
    // invoke download of the file  
  }  
};
```

5. Update the `saveFile` implementation of the `Editor.js` file according to the following code:

```
const saveFile = contents => {  
  // save via node.js process  
  if (window.require) {  
    const electron = window.require('electron');  
    const ipcRenderer = electron.ipcRenderer;  
  
    ipcRenderer.send('save', contents);  
  }  
  // save via the browser  
  else {  
    // ...  
  }  
};
```

6. Run the web server and use your preferred browser to test the `Cmd + S` or `Ctrl + S` feature.
7. Run the Electron shell with the `npm run electron` command and perform the same test once again. Everything should be working as expected.

In the next section, we are going to get the HTML conversion into the application. Similar to the `Save` channel, we can introduce a separate messaging channel for content conversion. Follow these steps to do so:

1. Let's put a new keyboard combination into the application. This can be anything you like. For the sake of simplicity, let's use `Cmd + Shift + H` to generate the HTML. Later in this chapter, we will use `Cmd + Shift + P` to generate a PDF.
2. Update the `editorDidMount` function's implementation with the new keyboard handler, as shown in the following code:

```
editor.addCommand(  
  monaco.KeyMod.CtrlCmd | monaco.KeyMod.Alt |  
  monaco.KeyCode.KEY_H,  
  () => {  
    const code = editor.getModel().getValue();  
    generateHTML(code);  
  }  
);
```

3. Create the `generateHTML` function so that you can send the markdown content of the code editor to the `generate` channel:

```
const generateHTML = contents => {  
  if (window.require) {  
    const electron = window.require('electron');  
    const ipcRenderer = electron.ipcRenderer;  
  
    ipcRenderer.send('generate', {  
      format: 'html',  
      text: contents  
    });  
  }  
};
```

Note how we pass the format as a payload option. This allows us to have a single channel for all kinds of formats in case we decide to provide support for more than one. This also simplifies the Node.js process as you can have a single function that parses the payload and invokes different features.

4. Update the code in `menu.js` with the following stub:

```
ipcMain.on('generate', (_, payload) => {  
  if (payload && payload.format) {  
    switch (payload.format) {  
      case 'html':  
        generateHTML(payload.text);  
    }  
  }  
});
```



```
        break;
      default:
        break;
    }
  }
});

function generateHTML(contents) {
  // todo: implementation
}
```

As you can see, the preceding code is pretty universal, and you can extend the support for different formats without rewriting significant portions of the code.

Now, it's time to use Docker to generate the HTML output from the markdown and invoke the browser to see the results.

Invoking Docker commands from Electron

We have already generated an example HTML file from the markdown source earlier in this chapter. You used the following command to do so:

```
docker run -v `pwd`::/source denysvuika/pandoc -f markdown -t html5 test.md
-o test.html
```

Our editor needs to perform the following steps to run the same command programmatically:

- Send the markdown text to the Node.js process.
- Save the markdown text to the local drive.
- Invoke the Docker command to generate the HTML output.
- Open the browser with the end result (this is optional).

Let's go through this now, starting with sending the markdown text to the Node.js process.

Sending the markdown text to the Node.js process

You already have the first bullet point implemented. The `generate` channel in the main process watches the messages and invokes the `generateHTML` function in case the `format` parameter has an `html` value.

All we need to do now is save the markdown context in a temporary location.

Saving the markdown text to the local drive

In this section, we are going to save the markdown into a temporary file. Follow these steps to do so:

1. Import the `os` and `path` objects from Node.js:

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

2. Now, import the file generation function. For the sake of simplicity, we're going to build a very minimalistic `happy path` function without validation or safety checks. Append the following code to the `menu.js` file:

```
function writeTempFile(contents, callback) {
  const tempPath = path.join(os.tmpdir(), 'markdown');

  fs.mkdtemp(tempPath, (err, folderName) => {
    const filePath = path.join(folderName, 'markdown.md');

    fs.writeFile(filePath, contents, 'utf8', () => {
      callback(filePath);
    });
  });
}
```



The preceding code is for demonstration purposes only. You may want to improve this function later on with some error handling and error checks.

3. Update the `generateHTML` code so that it looks as follows:

```
function generateHTML(contents) {
  writeTempFile(contents, fileName => {
    console.log('converting', fileName);
  });
}
```

4. Run the application and press `Cmd + Alt + H` for macOS or `Ctrl + Alt + H` for other platforms. We are just testing that the function works correctly. The console's output should look similar to the following:

```
converting
/var/folders/6r/_zpz77x67x5kg4h9dq8fb2w0000gp/T/markdown19bsMt/markdown.md
```

You can even navigate to that file and see its contents. It should be the same text that you entered in the editor.

Now is an excellent time to learn how to invoke a child process from within Node.js. Running child processes allows you to execute external commands, including console scripts and other applications.

5. Import the `exec` function from the `child_process` namespace, as well as the `shell` object from `electron`, as shown in the following code:

```
const { exec } = require('child_process');
const { shell } = require('electron');
```

We need the `shell` object to invoke the file in the default browser. For more details, please refer to the official documentation: <https://electronjs.org/docs/api/shell#shell>.

6. Next, update the implementation of the `generateHTML` function according to the following code:

```
function generateHTML(contents) {
  writeTempFile(contents, fileName => {
    const name = 'markdown';
    const filePath = path.dirname(fileName);
    const command = `docker run -v ${filePath}:/source
denysvuika/pandoc -f markdown -t html5 ${name}.md -o
${name}.html`;
    exec(command, () => {
      const outputPath = path.join(filePath, `${name}.html`);
      shell.openItem(outputPath);
    }).stderr.pipe(process.stderr);
  });
}
```

You may be wondering what the code in the `generateHTML` function does. Let's go over this now:

1. First, it generates a `markdown.md` file, along with its contents, in a temporary folder.
2. Then, it generates a shell command to execute the `docker` command. This takes the `markdown.md` file and produces the `markdown.html` file in the same temporary folder.
3. After that, the code runs the `exec` function to execute the command. As soon as the command finishes running, the code executes a second command to open the resulting `markdown.html` file with a default program that handles `.html` files. Typically, this action triggers your default web browser.

In some cases, you may see the following error in the console's output:

```
The path
/var/folders/6r/_zpzk77x67x5kg4h9dq8fb2w0000gp/T/markdownAQbH2t
is not shared from OS X and is not known to Docker.
You can configure shared paths from Docker -> Preferences... ->
File Sharing.
```

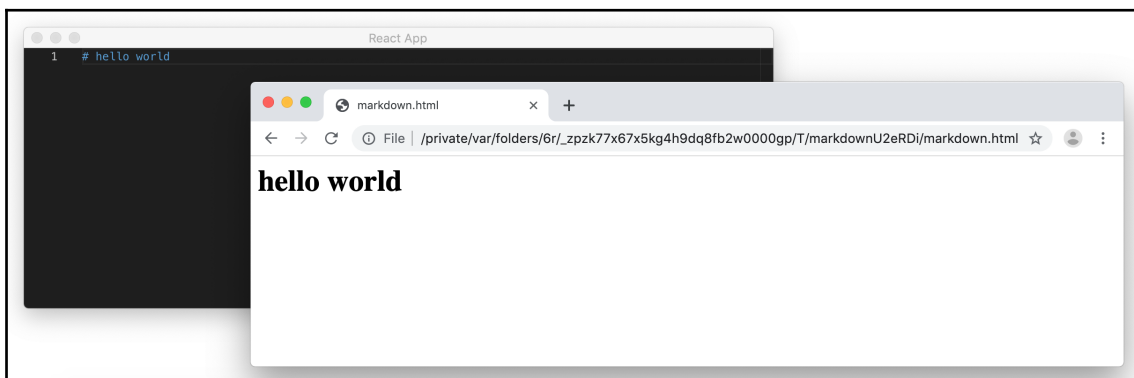
This means that your default temporary folder is not present in Docker's settings.

4. You can add any extra folders in the **File Sharing** configuration, as shown in the following screenshot:



Now it's time to test the whole workflow:

1. Run the Electron version of the application with the `npm run electron` command.
2. Press `Cmd + Alt + H` for macOS or `Ctrl + Alt + H` for other platforms. You should see the browser open with the HTML version of your markdown:



Congratulations on achieving another milestone. Now you can save and convert markdown files. You now know how to invoke external applications and execute shell commands from the Node.js process.

Let's see what it takes to generate a PDF version of our markdown content.

Generating PDF books

In this section, we are going to introduce support for PDF generation based on the markdown source. Once we've finished, the users of our application should be able to generate PDF output with `Cmd + Alt + P` on macOS and `Ctrl + Alt + P` on other systems.

Let's update the code in order to provide PDF output support:

1. Edit the `Editor.js` component so that it supports another keyboard combination:

```
editor.addCommand(  
  monaco.KeyMod.CtrlCmd | monaco.KeyMod.Alt |  
  monaco.KeyCode.KEY_P,  
  () => {  
    const code = editor.getModel().getValue();
```

```
        generatePDF(code);
    }
};
```

The `generatePDF` function implementation should be reasonably easy for you. It resembles the `generateHTML` one, but in our case, we're passing `pdf` as a format attribute.

2. Add the following function to the `Editor.js` code:

```
const generatePDF = contents => {
  if (window.require) {
    const electron = window.require('electron');
    const ipcRenderer = electron.ipcRenderer;

    ipcRenderer.send('generate', {
      format: 'pdf',
      text: contents
    });
  }
};
```

3. Update the `menu.js` code so that it matches the `pdf` format and call the `generatePDF` function from within the `generate` channel listener:

```
ipcMain.on('generate', (_, payload) => {
  if (payload && payload.format) {
    switch (payload.format) {
      case 'html':
        generateHTML(payload.text);
        break;
      case 'pdf':
        generatePDF(payload.text);
        break;
      default:
        break;
    }
  }
});
```

4. Next, add the `generatePDF` function:

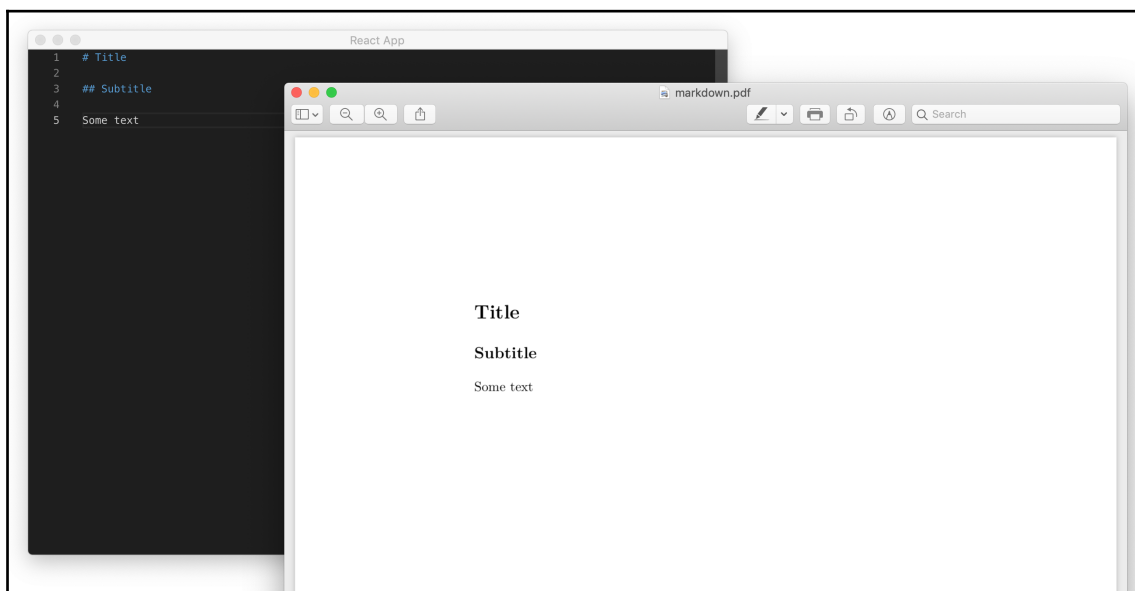
```
function generatePDF(contents) {
  writeTempFile(contents, fileName => {
    const name = 'markdown';
    const filePath = path.dirname(fileName);
    const command = `docker run -v ${filePath}:/source
```

```
denysvuika/pandoc -f markdown -t latex ${name}.md -o ${name}.pdf`;
  exec(command, () => {
    const outputPath = path.join(filePath, `${name}.pdf`);
    shell.openItem(outputPath);
  }).stderr.pipe(process.stderr);
});
}
```

The implementation that we use to generate and open PDFs is similar to the implementation for HTML. Note, however, that we pass different parameters to the Docker container.

Let's test this generation now:

1. Run the application and enter some markdown text; for example, a heading, subheading, and a bit of text.
2. Press *Cmd + Alt + P* if you are using macOS or *Ctrl + Alt + P* for other platforms.
3. Check out the default PDF viewer that appears as soon as the generation process ends:



You have successfully generated a simple PDF document out of the markdown content.



As you can imagine, Pandoc provides much more than this for PDF generation, such as templates, paper and font settings, and many other options. You can find out more at <https://pandoc.org/>.

Finally, let's learn how to produce ePub books as output from the markdown.

Generating ePub books

In this section, we are going to create a book in ePub format from the markdown content. The implementation steps should already be familiar to you now that you've gone through the HTML and PDF conversions.

On the browser side, let's slightly optimize the function so that we can run the conversion. Previously, we created separate functions, that is, `generateHTML` and `generatePDF`, to send a message to the `generate` channel.

If you take a look at the implementations of these functions carefully, you should notice that they only differ in terms of the `format` field. Instead of creating a third function called `generateEPUB`, I would suggest making the code more reusable.

Let's refactor our code and introduce the `generateOutput` function:

1. Create the `generateOutput` function, which covers every generation scenario:

```
const generateOutput = (format, text) => {
  if (window.require) {
    const electron = window.require('electron');
    const ipcRenderer = electron.ipcRenderer;

    ipcRenderer.send('generate', {
      format,
      text
    });
  }
};
```


2. Update the keyboard handlers so that we can reuse the universal function we have just introduced:

```
editor.addCommand(  
  monaco.KeyMod.CtrlCmd | monaco.KeyMod.Alt | monaco.KeyCode.KEY_H,  
  () => generateOutput(editor, 'html')  
);  
  
editor.addCommand(  
  monaco.KeyMod.CtrlCmd | monaco.KeyMod.Alt | monaco.KeyCode.KEY_P,  
  () => generateOutput(editor, 'pdf')  
);
```

3. At this point, we can remove the `generateHTML` and `generatePDF` functions as we no longer need them. Now, adding support for a new format conversion process is very trivial.
4. Add the `Cmd + Alt + E` combination (`Ctrl + Alt + E`) so that we can invoke ePub generation:

```
editor.addCommand(  
  monaco.KeyMod.CtrlCmd | monaco.KeyMod.Alt | monaco.KeyCode.KEY_E,  
  () => generateOutput(editor, 'epub')  
);
```

5. Now, switch to the `main.js` file and append the `generateEPUB` function to the code:

```
function generateEPUB(contents) {  
  writeTempFile(contents, fileName => {  
    const name = 'markdown';  
    const filePath = path.dirname(fileName);  
    const command = `docker run -v ${filePath}:/source  
denysvuika/pandoc -f markdown ${name}.md -o ${name}.epub`;  
    exec(command, () => {  
      const outputPath = path.join(filePath, `${name}.epub`);  
      shell.openItem(outputPath);  
    }).stderr.pipe(process.stderr);  
  });  
}
```

6. Next, update the channel listener code so that it takes the epub format into account and triggers the correct function:

```
ipcMain.on('generate', (_, payload) => {  
  if (payload && payload.format) {  
    switch (payload.format) {  
      case 'html':  
        generateHTML(payload.text);  
        break;  
      case 'pdf':  
        generatePDF(payload.text);  
        break;  
      case 'epub':  
        generateEPUB(payload.text);  
        break;  
      default:  
        break;  
    }  
  }  
});
```



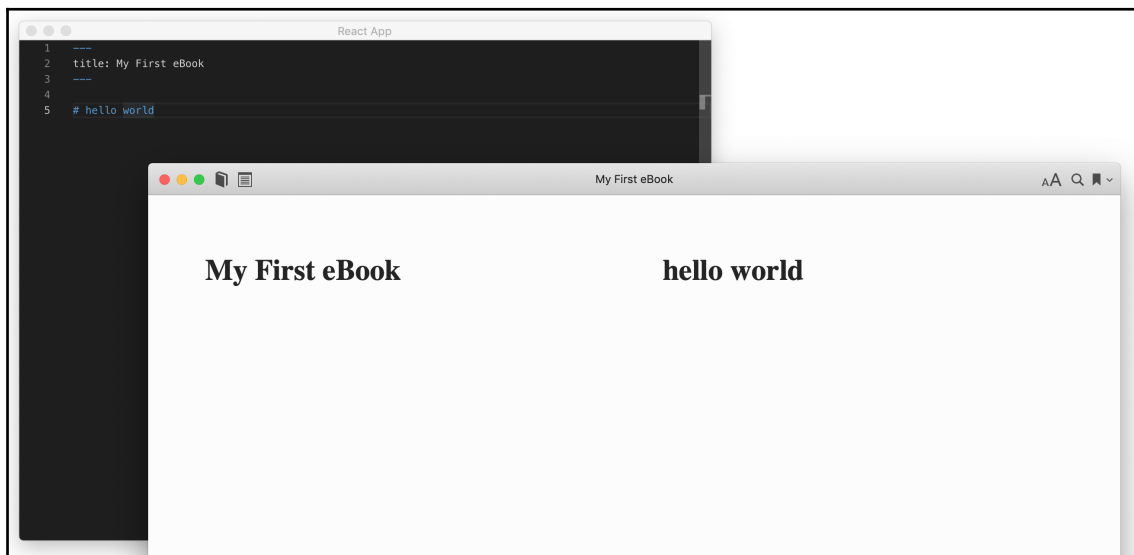
You can find out more about the eBook generation process at <https://pandoc.org/epub.html>.

7. Start your Electron application to test the resulting code.
8. Pandoc requires a document to include some specific metadata so that it can generate the eBook correctly. The bare minimum is the title value. Update the text in the editor and provide some `title` metadata, as shown in the following code:

```
---  
title: My First eBook  
---  
  
# hello world
```

9. Press `Cmd + Alt + E` if you are on macOS or `Ctrl + Alt + E` for other platforms.

10. This should make your operating system invoke the default application so that we can view `epub` files. In my case, I am using macOS and have the `Books` application running and displaying the resulting book:



Well done, and congratulations on finishing this minimalistic book generator implementation!

There are still many features you can introduce yourself if you wish to take on a practical exercise. Try adding export functionality so that you can export your files to a particular place while utilizing `Save Dialog`. Optionally, you can display generation errors on the screen.

Summary

In this chapter, you have learned how to build an Electron application so that you can generate various types of files. You have also learned how to use the Microsoft Monaco Editor component to build a markdown editing experience. In addition to that, we have walked through the process of setting up Pandoc with Docker, and you got familiar with invoking shell commands and applications from Node.js and Electron applications.

In the next chapter, we are going to walk through another exciting exercise—building a digital wallet application for the desktop.

10

Building a Digital Wallet for Desktops

In this chapter, we are going to build a simple digital wallet application based on Ethereum blockchain.

This chapter has been organized in a way that allows you to create your first Ethereum application without in-depth knowledge of blockchain and related technologies.

You are going to use the React library with the fantastic Ant Design components. You will also learn how to set up a personal blockchain for development purposes.

By the end of this chapter, you should have a good foundation project to build financial applications that can work with the Ethereum blockchain.

In this chapter, we will cover the following topics:

- Generating the project scaffold with React
- Integrating the Ant Design library
- Setting up a personal Ethereum blockchain
- Configuring the Ethereum JavaScript API
- Displaying Ethereum Node information
- Integrating with the application menu
- Rendering a list of accounts
- Showing our account balance
- Transferring ether from one account to another
- Packaging the application for distribution

Let's start our journey by generating a new React-based project that suits our digital wallet application's needs. But before that, let's quickly take a look at the technical requirements for this project.

Technical requirements

To get started with this chapter, you will need a standard laptop or desktop running macOS, Windows, or Linux.

The software you will need to complete this chapter is as follows:

- Git, a version control system
- Node.js with NPM
- Visual Studio Code, a free and open source code editor

You can find the code files for this chapter in this book's GitHub repository: <https://github.com/PacktPublishing/Electron-Projects/tree/master/Chapter10>.

Generating the project scaffold with React

The fastest way to start the project is to use the React library and the Create React App tool. Follow these steps to get started:

1. Run the following commands to generate a new React application called `crypto-wallet`:

```
npx create-react-app crypto-wallet
cd crypto-wallet
```

2. Install the latest `electron` library with the following command:

```
npm install -D electron
```

As you already know, a typical Electron application requires a `main` entry in the `package.json` file. We are going to use the `public/electron.js` file so that we can create a distributable package without any effort.

3. Update the `package.json` file and add the `main` entry:

```
{
  "name": "crypto-wallet",
  "version": "0.1.0",
  "private": true,
  "main": "public/electron.js",
  // ...
}
```

The scripts for React applications usually reserve the `start` script in order to run the local development web server. You can use the `electron` script to run the desktop version.

4. Append the `electron` command to the `scripts` section:

```
"scripts": {  
  "electron": "electron .",  
  "start": "node scripts/start.js",  
  "build": "node scripts/build.js",  
  "test": "node scripts/test.js"  
},
```

The final thing we need to do is create an `electron.js` file with a minimal amount of code so that we can run the application window.

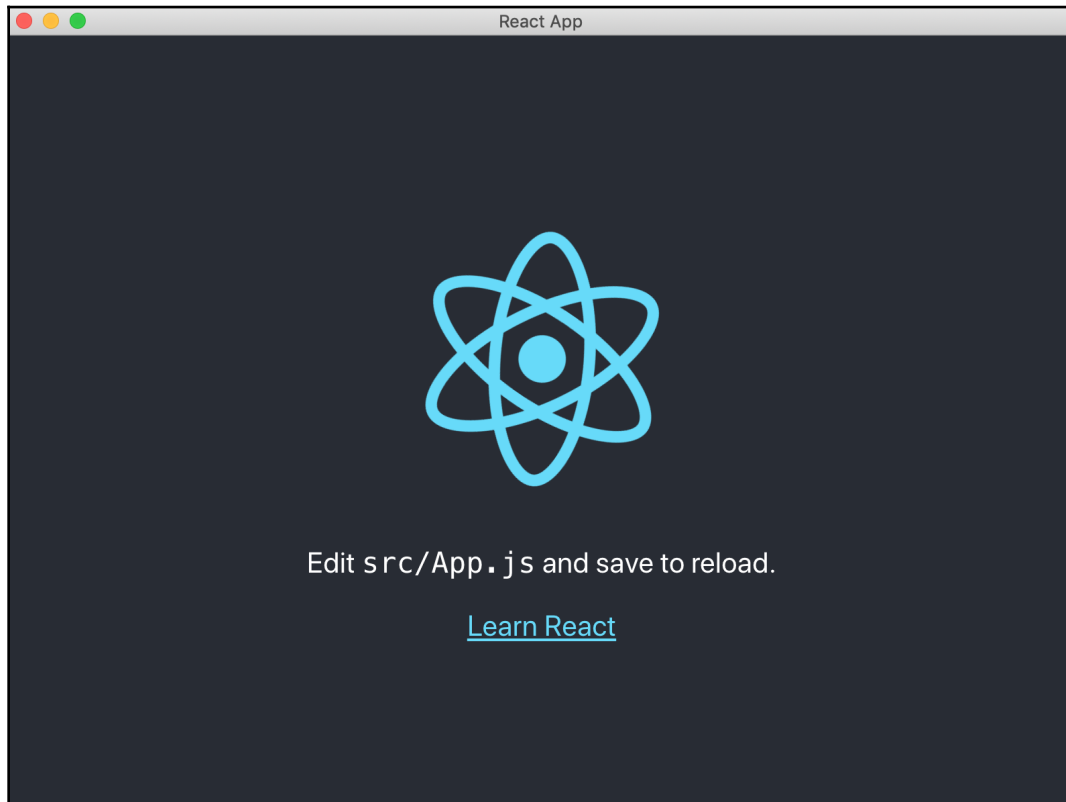
5. Create the `electron.js` file with the following content in the `public` folder:

```
const { app, BrowserWindow } = require('electron');  
  
function createWindow() {  
  const win = new BrowserWindow({  
    width: 800,  
    height: 600,  
    webPreferences: {  
      nodeIntegration: true  
    },  
    resizable: false  
  });  
  
  win.loadURL(`http://localhost:3000`);  
}  
  
app.on('ready', createWindow);
```

That is pretty much all you need to do to have a minimal project configuration. From now on, you can test the web version by running the following commands in parallel Terminal or Command Prompts windows:

```
npm start  
npm run electron
```

Once the application is up and running, you should see an Electron window that looks as follows:



Now that we've got the project scaffold up and running, let's integrate the Ant Design component library so that we can build our application quickly and make it look good.

Integrating the Ant Design library

For the digital wallet application, we are going to use Ant Design components for React.

Ant Design is a design system that uses the values of nature and determinacy to enhance the user experience of enterprise applications. Using Ant Design, you can gain access to an extensive collection of fantastic components so that you can build your applications quickly. Make sure you visit <https://ant.design/> to find out more, as well as examples and documentation.

The first thing we are going to do is install the Ant Design library. We are also going to implement a traditional application layout that contains a header, footer, sidebar, and main content area. Let's take a look at the steps for installation:

1. Run the following command to install the `antd` library into your project:

```
npm install antd
```



There are multiple ways to configure `antd` with your project. Please refer to the following link if you need more information: <https://ant.design/docs/react/introduce>.

2. Open `index.css` and append the following style to it:

```
html,  
body,  
#root {  
  height: 100%;  
  width: 100%;  
}
```

In the preceding code, we are making the application take up the full page size. You should also update `App.css` with a couple of new style rules to make the layout look neat and tidy.

3. Replace the content of the `App.css` file with the following content:

```
.App {  
  height: 100%;  
}  
  
.App > .ant-layout {  
  height: 100%;  
}
```

Besides this, we are going to take some styling from the Ant examples for the `Layout` component.

4. Append a few additional rules to make the layout look good:

```
.ant-layout-header,  
.ant-layout-footer {  
  background: #7dbcea;  
  color: #fff;  
}  
  
.ant-layout-footer {
```



```
    line-height: 1.5;
  }
  .ant-layout-sider {
    background: #3ba0e9;
    color: #fff;
    line-height: 120px;
  }
  .ant-layout-content {
    background: rgba(16, 142, 233, 1);
    color: #fff;
    min-height: 120px;
    line-height: 120px;
  }
}
```

All of the styles we need for the initial setup are now in place. Switch to the `App.js` component code to build the component tree.

5. Update the imports section by adding the `antd` styles and components:

```
import React from 'react';

import 'antd/dist/antd.css';
import './App.css';

import { Layout } from 'antd';
const { Header, Footer, Sider, Content } = Layout;
```



Note how we import `App.css` after `antd.css`. The order of imports is essential as it allows you to customize the styles of the components on top of what Ant Design already provides.

6. Replace the component function with the updated template according to the following code:

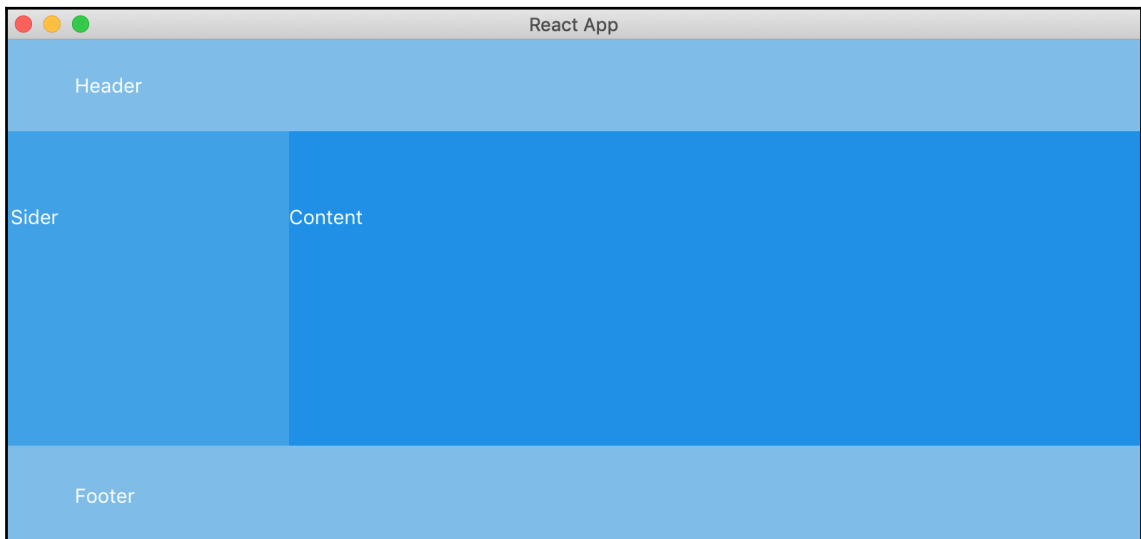
```
function App() {
  return (
    <div className="App">
      <Layout>
        <Header>Header</Header>
        <Layout>
          <Sider>Sider</Sider>
          <Content>Content</Content>
        </Layout>
        <Footer>Footer</Footer>
      </Layout>
    </div>
  );
}
```

```
    );  
  }
```



You can find multiple layout examples at <https://ant.design/components/layout/>.

7. Run the application to see what the main page looks like:



This is the initial layout for our digital wallet application. It has header and footer blocks, along with a sidebar and main content area.

Now, it's time to set up a personal Ethereum blockchain so that we can test the application on demo data and accounts without spending real money.

Setting up a personal Ethereum blockchain

You don't need to create a real Ethereum wallet or register any accounts to test the application. In this section, we are going to set up a personal Ethereum blockchain that you can use for application development purposes. You also need an amount of virtual money to test how the application works without putting any real money at risk.

The easiest way to get started with Ethereum development is to use the *Ganache* tool. Ganache positions itself as a *One-Click Blockchain* for developers and allows you to do the following:

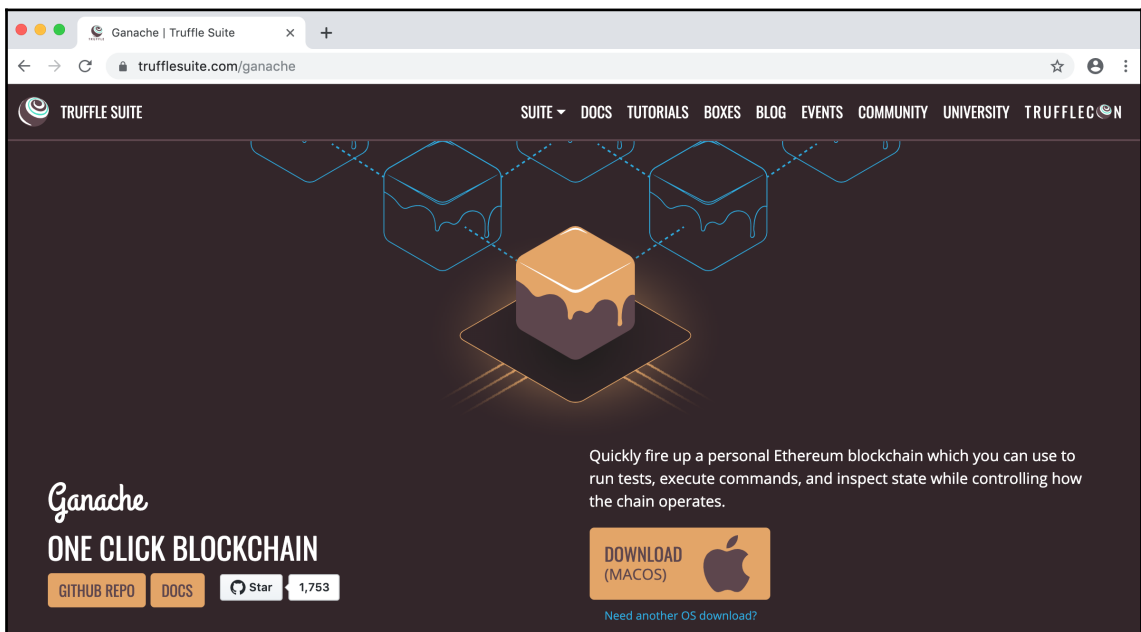
Quickly fire up a personal Ethereum blockchain that you can use to run tests, execute commands, and inspect state while controlling how the chain operates

(<https://www.trufflesuite.com/ganache>)

You can find out more about this tool at <https://www.trufflesuite.com/ganache>.

Setting up the tool is straightforward and doesn't take much time. Follow these steps to install it on your local machine:

1. Navigate to the <https://www.trufflesuite.com/ganache> website and find the download button on the main page, as shown in the following screenshot:



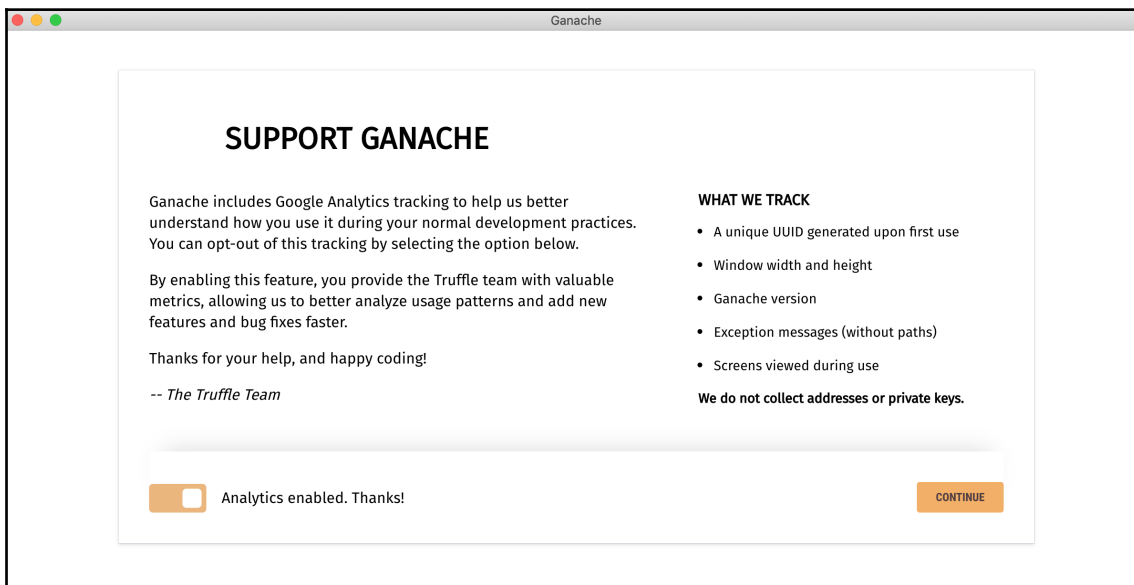


You should see different buttons, depending on the operating system you're using. In my case, I received macOS option, but you can always click the **Need another OS download?** link to see of all the available downloads.

2. Run the application installer. For macOS, you should see a standard installer experience where you can drag and drop the executable into the **Applications** folder:



3. Run the application. On the first run, you should see an `Analytics` consent dialog, asking you to enable or disable application analytics:



It is up to you to choose whether to leave with analytics enabled or go fully anonymous.

4. Click the **Continue** button. At this point, you should see the standard landing page for the application. It should look as follows:



You are going to see **Quickstart** and **New Workspace** each time the application starts so that you can decide on what type of actions you want to perform. For now, I strongly recommend going to **Quickstart** so that you have a one-click blockchain ready for development and testing.

- Click the **Quickstart** button. You should see the main application interface, along with a list of accounts on the main page.



Feel free to explore the options you have and all of the pages and dialog. You can also refer to the online documentation (<https://www.trufflesuite.com/docs/ganache/overview>) and examples if you want to find out more about the tool.

Note that you also get a few out-of-the-box accounts that have 100 Ether each:

ACCOUNTS

BLOCKS

TRANSACTIONS

CONTRACTSEVENTSLOGS

SEARCH FOR BLOCK NUMBERS OR TX HASHES

CURRENT BLOCK
0

GAS PRICE
20000000000

GAS LIMIT
6721975

HARDFORK
PETERSBURG

NETWORK ID
5777

RPC SERVER
HTTP://127.0.0.1:7545

MINING STATUS
AUTOMINING

WORKSPACE
QUICKSTART

SAVE

SWITCH

MNEMONIC

pencil small liberty shell wide hire engage genuine license old impulse improve

HD PATH
m/44'/60'/0'/0/account_index

ADDRESS
0x528D7a95fCc59C5CCb3e7f76Ad36b512b223776

BALANCE
100.00 ETH

TX COUNT
0

INDEX
0

ADDRESS
0xAb86bc0FF4083b155289E6Aa1d697478298ec343

BALANCE
100.00 ETH

TX COUNT
0

INDEX
1

ADDRESS
0x977a69924363348026e818F45d74734E01F27515

BALANCE
100.00 ETH

TX COUNT
0

INDEX
2

ADDRESS
0x192bb6Ec41fAc2315f95b0afe6065e0134761c60

BALANCE
100.00 ETH

TX COUNT
0

INDEX
3

ADDRESS
0xcc188bbcEA06832AB2ce4F5677D07b922a96bF0a

BALANCE
100.00 ETH

TX COUNT
0

INDEX
4

ADDRESS
0xC1ebF7aD5418e4c58fD76ebC8Aa8eA577bB0598A

BALANCE
100.00 ETH

TX COUNT
0

INDEX
5

ADDRESS
0xfa2dc7d5Da65824de6B9a29147486D434d26C4Da

BALANCE
100.00 ETH

TX COUNT
0

INDEX
6

This should be more than enough for you to build a digital wallet application that works with multiple accounts. Having some Ether on each account also helps us to develop and test the transfer functionality without having to spend any money.

Leave the Ganache application running for now. It is going to be our backend service for the rest of this chapter.

Next, let's learn how to enable a typical Electron application with Ethereum support. I am going to explain how to install and configure the JavaScript library so that it can work with Ethereum protocols.

Configuring the Ethereum JavaScript API

In this section, we are going to set up a `web3.js` library with our Electron application. `web3.js` is a collection of libraries that allow you to interact with a local or remote Ethereum node using an HTTP or IPC connection. You can find out more on the official documentation website: <https://web3js.readthedocs.io>.



Make sure you also visit the GitHub repository if you are interested in the library: <https://github.com/ethereum/web3.js/>.

Let's learn how to set up and integrate the `web3` library with our Electron application and see how it works:

1. Install the `web3` library with the following command:

```
npm i web3
```

2. Update the `App.js` file and add the following code to import the `Web3` client, which works with port 7545, onto your local machine:

```
import Web3 from 'web3';
const web3 = new Web3('ws://localhost:7545');

function App() {
  console.log(web3);

  return (
    <div className="App">
      <Layout>
        <Header>Header</Header>
        <Layout>
          <Sider>Sider</Sider>
          <Content>Content</Content>
        </Layout>
        <Footer>Footer</Footer>
      </div>
    )
  )
}
```

```

    </Layout>
  </div>
);
}

```

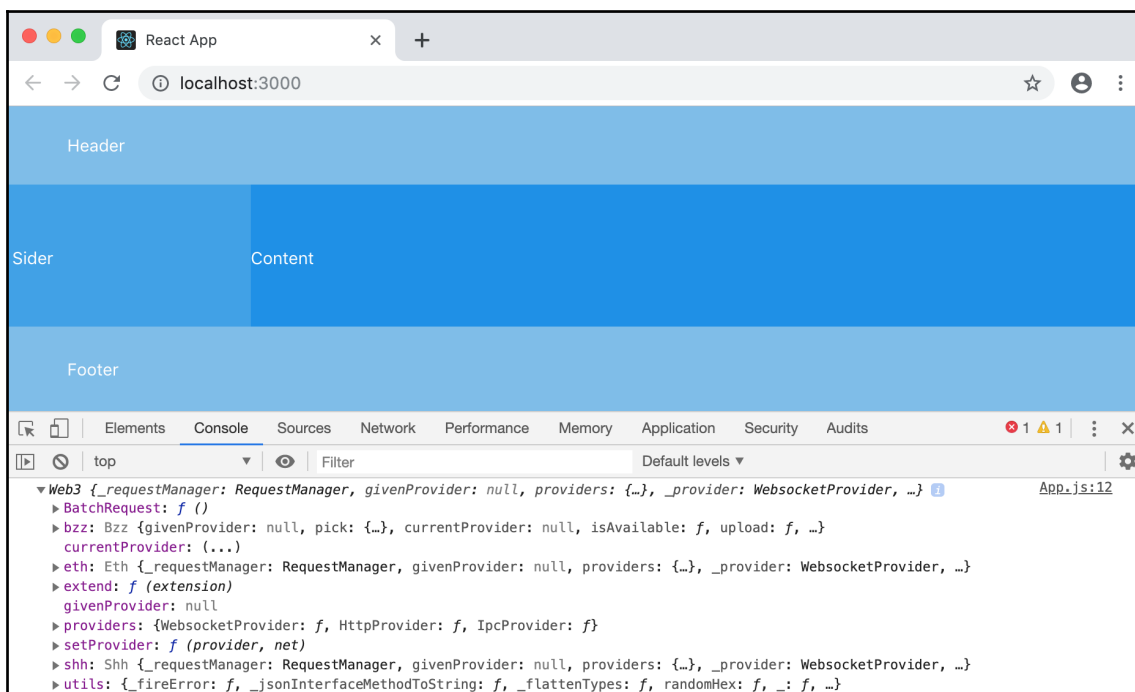


By default, Ganache runs on port 7545. We are going to use this port in all the following examples, but you can change it to serve another port in the application settings.

Make sure you import the Web3 object at the top of the file; otherwise, you may get runtime errors.

The preceding code doesn't do much. Here, we created a new client and sent the instance to the console log output to see its contents. Right now, we need to ensure that the library works as expected.

3. Run the application and enable the developer tools. You should see the following output in the **Console** window:



As you can see, the console output contains a JavaScript object with multiple properties and methods. This means that our React application has the `Web3.js` library embedded inside it and is running on startup.

At this point, you are ready to make a cross-platform Electron application that utilizes the Ethereum protocols. The `web3` library is up and running, and you are ready to make API calls against the locally running node.

In the next section, we are going to connect to the Ganache instance on our local machine and display information about the current Ethereum node.

Displaying Ethereum Node information

Retrieving node information is the first thing we can do to test that the application can connect to our locally running blockchain via the Ganache server.

In this section, we are going to retrieve some basic information and display it in the Header section of the main application page.

We shall touch on the following important aspects in this section:

- Getting information about the currently running Ethereum node
- Presenting node information to users on the screen

First, let's learn how to fetch node information.

Getting node information

We already have a `web3` client instance. We can use the following API to gather information about the current node:

```
web3.eth.getNodeInfo(callback)
```

Let's see how this works in the React application that we are building:

1. Insert the following code right after creating the `Web3` instance:

```
const web3 = new Web3('ws://localhost:7545');
console.log(web3);

web3.eth.getNodeInfo(function(error, result) {
  if (error) {
```

```
        console.error(error);  
      } else {  
        console.log('result', result);  
      }  
    });
```

2. As soon as you run the application, the console log should contain the following information:

```
result EthereumJS TestRPC/v2.8.0/ethereum-js
```

You are making good progress so far. Let's display the node information in the user interface on startup. We are going to use the React Hooks feature for this.



To find out more about React Hooks, please refer to the official documentation: <https://reactjs.org/docs/hooks-intro.html>.

Rendering node information in the header

You already know how to get information about your local Ethereum node, so now is a good time to render this information in the user interface. For the sake of simplicity, I recommend putting it into the Header area so that our users can always see what node they are working with.

Let's introduce the `node` and `setNode` React Hooks:

1. Update the `App.js` implementation according to the following code:

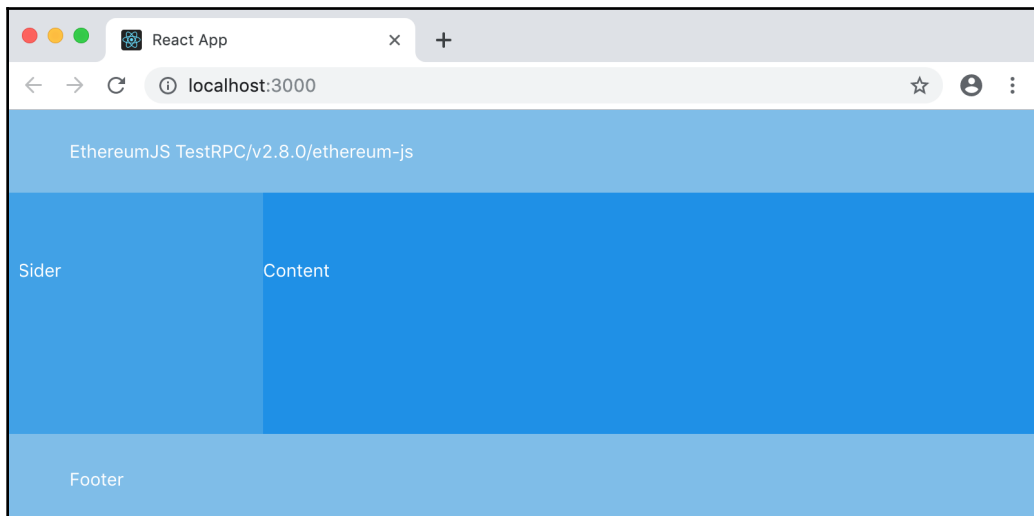
```
import React, { useState } from 'react';  
  
function App() {  
  const [node, setNode] = useState('Unknown Node');  
  
  // ...  
}
```

Now, you can use the `node` hook to display the node's information or `Unknown Node` in case errors occur. You can also use `setNode` to update the node's information. This is what we are now going to do.

2. Update the `getNodeInfo` call, as shown in the following code:

```
web3.eth.getNodeInfo(function(error, result) {  
  if (error) {  
    console.error(error);  
  } else {  
    setNode(result);  
  }  
});
```

3. Switch to the web application and check the Header area. The main page should now look as follows:



Congratulations on scoring another goal. Now, your application has a header component that displays information about the current Ethereum node that you created with the Ganache tool.

Now that you're familiar with how the `web3.js` library works, we are going to display a list of accounts that you have in your Ethereum node.

It is a good idea to integrate the application menu as well. We're going to allow our users to see node information from the **Help** menu.

Integrating with the application menu

In this section, we are going to perform a basic integration with the application menu. You are going to create a **Help/About Node** menu that sends the `show-node-info` command to the renderer process. Our React application is going to handle the command and display a simple alert box with the Ethereum node information. Later on, you can provide a more sophisticated dialog with detailed information.

Let's start with the main process and the `Menu` template:

1. Switch to the `public/electron.js` file and import the `Menu` object from the Electron framework:

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

2. Append the `Menu` code from the following listing to the bottom of file:

```
Menu.setApplicationMenu(  
  Menu.buildFromTemplate([  
    {  
      label: 'Help',  
      submenu: [  
        {  
          label: 'About Node',  
          click() {  
            const window = BrowserWindow.getFocusedWindow();  
            window.webContents.send('commands', 'show-node-info');  
          }  
        }  
      ]  
    }  
  ])  
);
```

As you can see, we find the browser window and send the `show-node-info` payload via the `commands` channel. Now, it's time to update the renderer process and the `src/App.js` file.

3. Update the `useEffect` block in the `App.js` file according to the following code:

```
useEffect(() => {  
  // ...  
  if (window.require) {  
    const electron = window.require('electron');  
    const ipcRenderer = electron.ipcRenderer;  
    const showNodeInfo = (_, command) => {  
      if (command === 'show-node-info') {  
        window.alert(`Node: ${node}`);  
      }  
    }  
    ipcRenderer.on('commands', showNodeInfo);  
    return () => {  
      ipcRenderer.off('commands', showNodeInfo);  
    }  
  }  
}, [node]);
```

The preceding code is pretty straightforward. We gain access to `ipcRenderer` and start listening to the `commands` channel. In the `show-node-info` payload, we show the alert, along with the node information.

4. Start the web server and then the Electron application. Now, we can check out the **Help/About Node** menu.

You should be able to see the same node information that the application displays in the header. The next thing we need to do is render the list of the accounts we have inside our node.

Rendering a list of accounts

You have already managed to configure the Ethereum JavaScript client with the Electron application and got the node information in the Header area. In this section, we are going to display a list of accounts in the tree component and place it into the sidebar area.

As you may have noticed, the account names are quite long and may not fit in their dedicated area. We are going to make the sidebar scrollable vertically and stop the content from overlapping so that it's hidden under the main content area.

Let's start by changing the main application stylesheet and disabling the layout overflow for our sider component:

1. Update the `App.css` file and extend the `ant-layout-sider` style:

```
.ant-layout-sider {  
  background: #3ba0e9;  
  color: #fff;  
  line-height: 120px;  
  
  overflow: hidden;  
  overflow-y: scroll;  
}
```

Now, we need to have a dedicated pair of hooks to store the account list.

2. Update the `App.js` file and add the new hooks, as shown in the following code:

```
function App() {  
  const [node, setNode] = useState('Unknown Node');  
  const [accounts, setAccounts] = useState([]);  
  
  // ...  
}
```

3. Import `useEffect` from the React namespace and create the following effect, which loads the account list from Ganache:

```
import React, { useState, useEffect } from 'react';  
  
function App() {  
  //...  
  
  useEffect(() => {  
    web3.eth.getAccounts(function(error, accounts) {  
      if (error) {  
        console.error(error);  
      } else {  
        setAccounts(accounts);  
      }  
    });  
  });  
  // ...  
};
```

As you can see, we're using the `web3.eth.getAccounts` API that the `web3` library provides to retrieve a list of accounts. If there are no errors during the API call, we can use the `setAccounts` hook with the newly received value. This value is going to be an array of strings.

4. Import the `Tree` and `TreeNode` components from the Ant Design library:

```
import { Tree } from 'antd';  
const { TreeNode } = Tree;
```



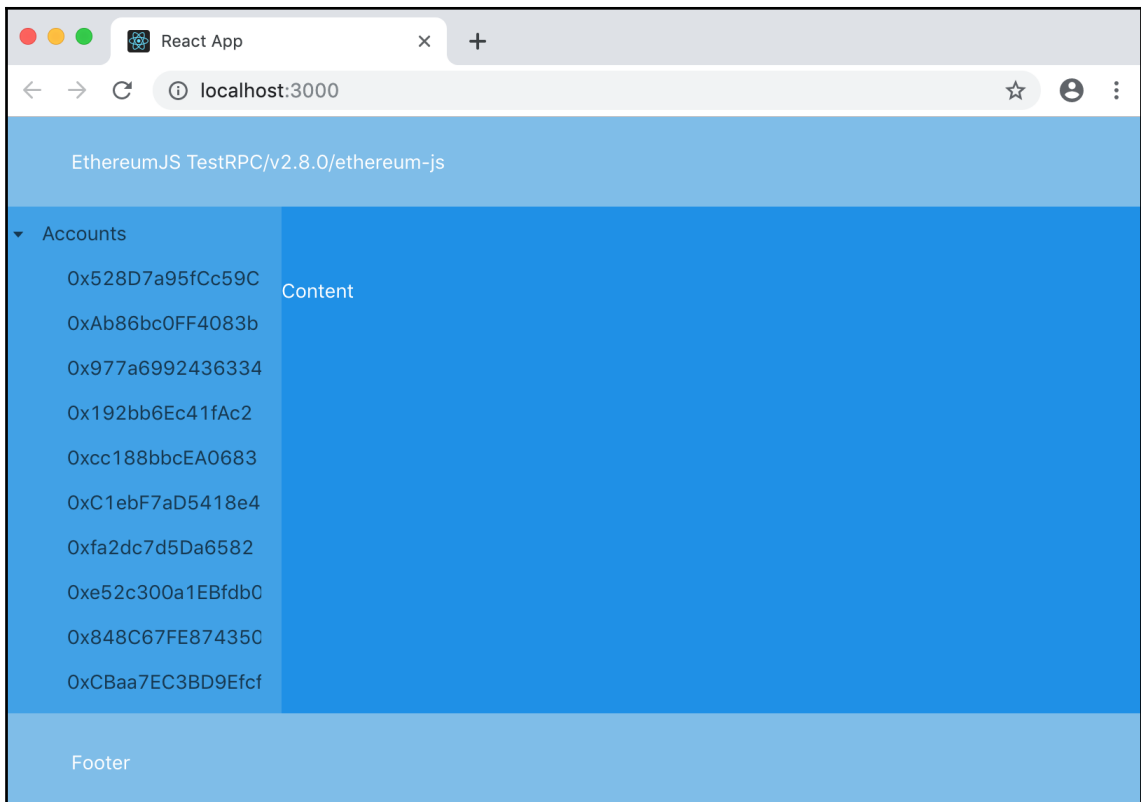
You can find examples of the `Tree` component in the official documentation: <https://ant.design/components/tree/>. Make sure you check the API and the documentation on the component if you decide to have more complicated scenarios that contain `Tree` nodes.

5. Replace the `Slider` placeholder block with the following code:

```
<Slider>  
  <Tree>  
    <TreeNode title="Accounts" key="accounts">  
      {accounts.map(account => (  
        <TreeNode key={account} title={account}></TreeNode>  
      ))}  
    </TreeNode>  
  </Tree>  
</Slider>
```

Here, we've built a root `Accounts` node and dynamically created child nodes according to the state of the `accounts` hook.

6. Run the web application and check out the sidebar. The left sidebar area should now look as follows:



Now, your application displays a list of Ethereum accounts.

Before we move on to the next section, let's improve the application's presentation. We are going to trim the account names to 10 characters and append the ellipsis symbol to the end of each name. Follow these steps to do so:

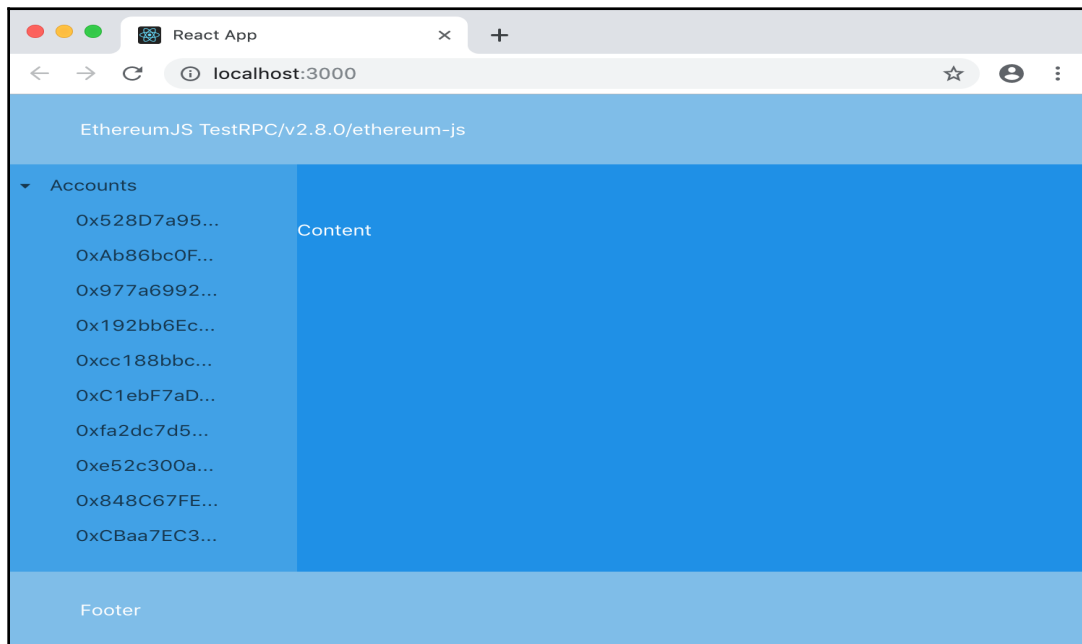
1. Add the `formatAccountName` function inside the component:

```
const formatAccountName = name => {  
  if (name && name.length > 10) {  
    return `${name.substring(0, 10)}...`;  
  }  
  return 'Noname';  
};
```


2. Update the `Tree` component according to the following code:

```
<TreeNode title="Accounts" key="accounts">
  {accounts.map(account => (
    <TreeNode
      key={account}
      title={formatAccountName(account)}
    ></TreeNode>
  ))}
</TreeNode>
```

3. This time, the list of accounts should look slightly better:



You can keep improving the look and feel of your application even more if you want to. For example, you can add a tooltip with the full account name, show icons next to titles, change the colors of the element backgrounds, and so on.

Another essential part of our wallet application is a display of the account balance. This is what we are going to address in the next section.

Showing our account balance

At this point, you have a few accounts in the local Ethereum node. We have also managed to render a tree of account names so that our users can see all of them in the user interface.

Given that each account may contain various amounts of money, it is very important to allow users to see the balance of each account from the application.

Using the following steps, we are going to wire the account tree component to the main content area and display the balance of the selected account:

1. Introduce a pair of hooks for the current account balance:

```
function App() {  
  const [node, setNode] = useState('Unknown Node');  
  const [accounts, setAccounts] = useState([]);  
  const [balance, setBalance] = useState(0);  
  
  // ....  
}
```

To get the balance of a particular account, you need to use the `web3.eth.getBalance` API from the Web3 library. This means we need a selection handler for the `Tree` component, and the handler needs to call the API and use the `setBalance` hook to save the value.

2. Add the `onSelectAccount` function inside the component function:

```
const onSelectAccount = keys => {  
  const [account] = keys;  
  
  if (account && account !== 'accounts') {  
    web3.eth.getBalance(account).then(function(result) {  
      setBalance(web3.utils.fromWei(result, 'ether'));  
    });  
  } else {  
    setBalance(0);  
  }  
};
```

Now, we need to assign `onSelectAccount` to the `Tree` selection event.

3. Update the `Tree` component declaration with the `onSelect` attribute, as shown in the following code:

```
<Tree onSelect={onSelectAccount}>
  <TreeNode title="Accounts" key="accounts">
    {accounts.map(account => (
      <TreeNode
        key={account}
        title={formatAccountName(account)}
      ></TreeNode>
    ))}
  </TreeNode>
</Tree>
```

Now, all we need to do is display the value to the users. To do this, we can use the `Statistic` component from the Ant Design library.

4. Import the `Statistic` component using the following code:

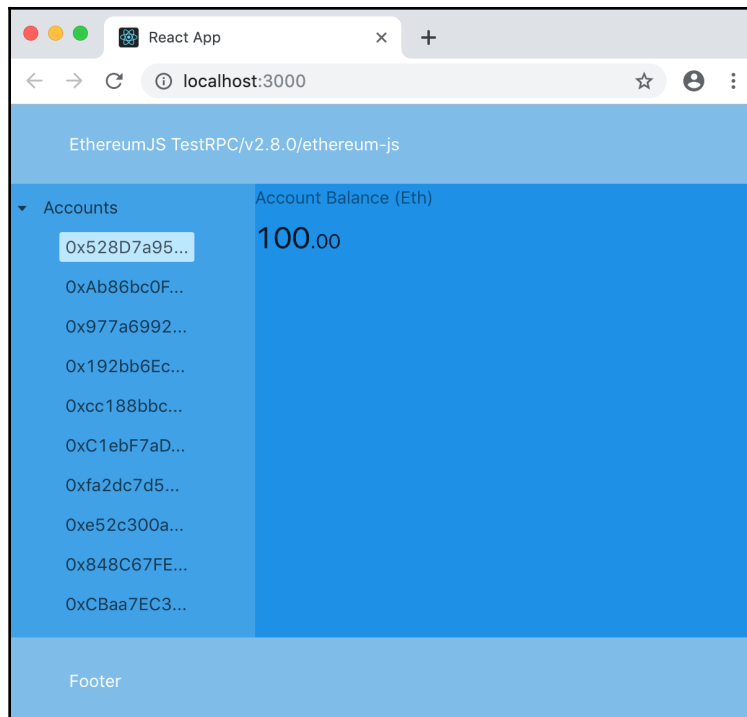
```
import { Layout, Tree, Statistic } from 'antd';
```

5. Replace the `Content` element placeholder with the following code:

```
<Content>
  <Statistic
    title="Account Balance (Eth)"
    value={balance}
    precision={2}
  />
</Content>
```

This is the minimal amount of configuration we need in order to get the balance feature working in our application.

6. Reload the page and try clicking on the account entries. The main content area should show the different account balances, as shown in the following screenshot:



Notice that you can click on the same account a second time to deselect it. The account balance label should revert to zero, as per our selection handler implementation.

Now that we know how to display the account balance, let's learn how the transfer process works.

Transferring Ether to another account

So far, you have a list of accounts in the sidebar and can see the balance of each account by selecting them. Now, it's time to implement transfers between accounts.

First of all, we need at least three parameters to perform a successful transaction:

- Source account
- Target account
- Amount to transfer

We are going to use React Hooks since this is the most efficient and fastest way to get started. You need at least three pairs of hooks. Let's get started:

1. Introduce a pair of React hooks for each attribute:

```
const [account, setAccount] = useState(null);
const [targetAccount, setTargetAccount] = useState(null);
const [transferAmount, setTransferAmount] = useState(0);
```

We already handle account selection via the `onSelectAccount` function. This function can also be updated to keep track of the selected account so that we can use that value to perform transactions.

2. Update the `onSelectAccount` function so that you can set and reset the account state:

```
const onSelectAccount = keys => {
  const [account] = keys;

  if (account && account !== 'accounts') {
    web3.eth.getBalance(account).then(function(result) {
      setBalance(web3.utils.fromWei(result, 'ether'));
      setAccount(account);
    });
  } else {
    setBalance(0);
    setAccount(null);
  }
};
```

Now, you need a form to collect all of the user input and a button to perform the transaction. The Ant Design library can provide you with everything you need.

3. Import the following extra components to build the form:

```
import { Layout, Tree, Statistic, Select, Form, Input, Button,
message } from 'antd';
```

At this point, you can start building the form layout. You already have the account state maintained, so let's display it at the top of the form.

4. Insert the following code inside the `Form` component:

```
<Content>
  <Statistic
    title="Account Balance (Eth)"
    value={balance}
```

```

    precision={2}
  />
  <Form style={{ width: 450 }}>
    <Form.Item>
      <Input value={account} disabled={true}></Input>
    </Form.Item>
  </Form>
</Content>

```

As you can see, we're showing the input element but are preventing our users from editing it. The only way to change the field is to pick another account from the tree component in the sidebar.

The next field should allow users to pick the target account. We should already have a list of accounts to back the tree component. We can use the account list to build the picker component.

5. Append the following input to the `Form` component:

```

<Form.Item>
  <Select
    placeholder="Select target account"
    onChange={value => setTargetAccount(value)}
  >
    {accounts
      .filter(acc => acc !== account)
      .map(account => (
        <Select.Option key={account} value={account}>
          {account}
        </Select.Option>
      ))}
  </Select>
</Form.Item>

```



Note that we don't allow the target account to be used as the source account. Each time `Select` is rendered, we filter out the selected account from the list.

The third field in `Form` is the number input so that we can provide the amount we wish to transfer.

6. Append the `Input` component to `Form`, as shown in the following code:

```

<Form.Item>
  <Input
    type="number"

```

```
      min="0"
      placeholder="Amount"
      value={transferAmount}
      onChange={e => setTransferAmount(e.target.value)}
    ></Input>
  </Form.Item>
```

7. Add the Transfer button to the bottom of the Form:

```
<Button disabled={!canTransfer()} onClick={onTransferClick}>
  Transfer
</Button>
```

As you can see, the button needs two additional functions: `canTransfer`, which controls the state of the button, and the `onTransferClick` event handler for the click event.

8. Add the button-related functions to the component function:

```
const canTransfer = () => {
  return account && targetAccount && transferAmount &&
  transferAmount > 0;
};

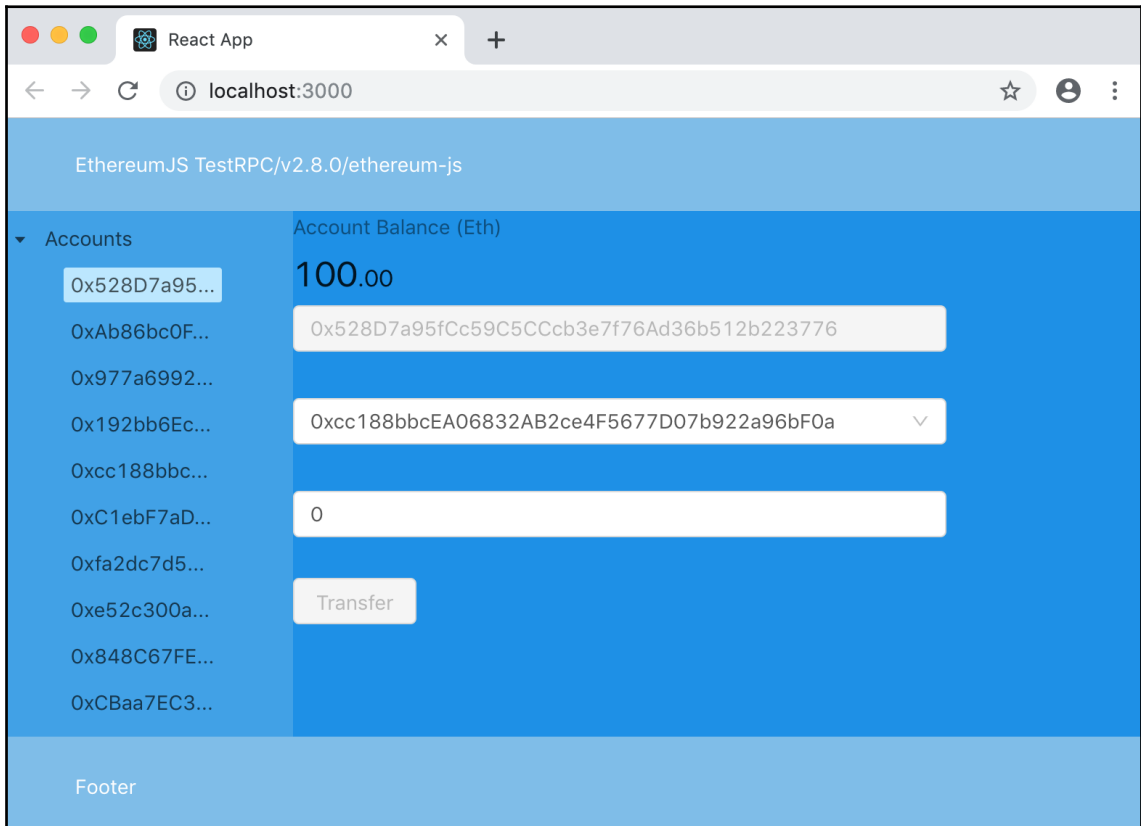
const onTransferClick = () => {
  console.log('from', account);
  console.log('to', targetAccount);
  console.log('amount', transferAmount);
};
```

`canTransfer` allows you to disable button unless the following criteria are met:

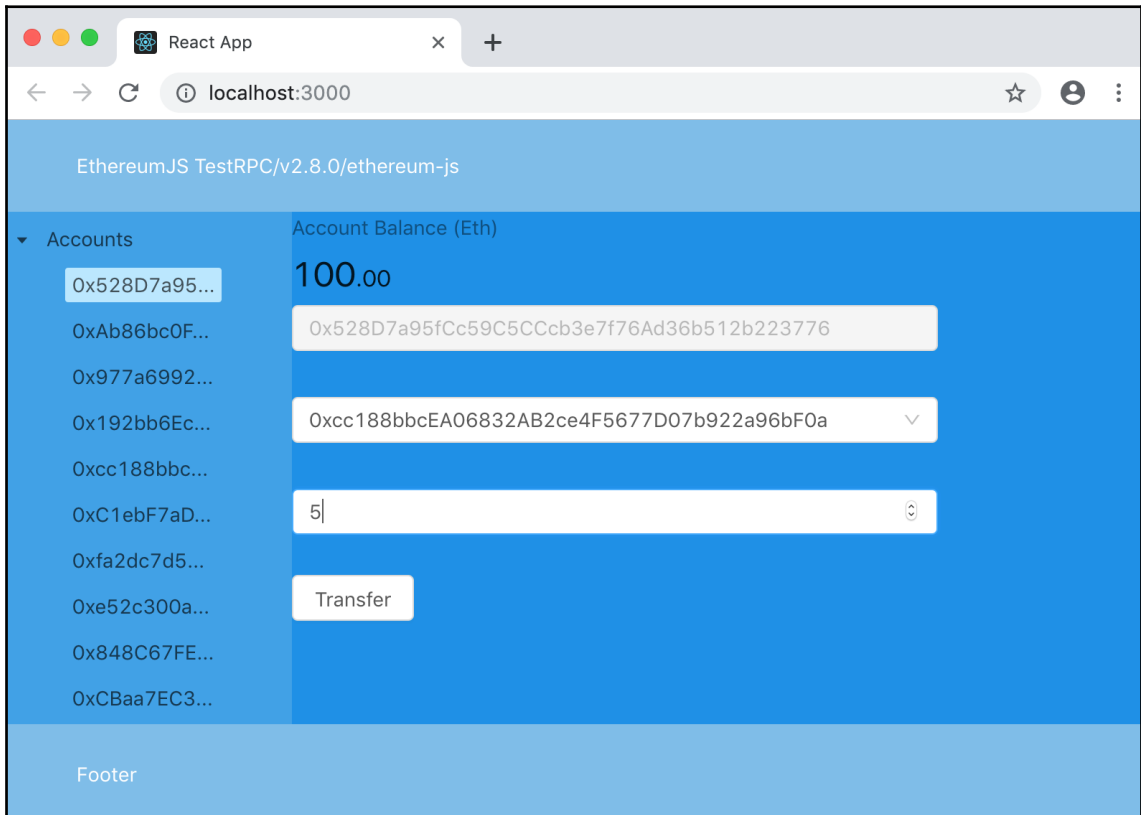
- The account is selected in the sidebar.
- The target account is selected.
- The transfer amount is provided and it's a non-zero number.

As for the `onTransferClick` code, we send the form values to the console log for now.

9. Run the web application and try entering some data. However, leave the amount input with the default zero value. Note that the **Transfer** button is disabled:



10. Set the transfer amount field to a non-zero value such as 5. This time, the **Transfer** button will be enabled:



You can try changing the value to see how the button reacts to the values you enter in real time.

11. Click the **Transfer** button and check the browser's console log. You should see the following output:

```
from 0x528D7a95fCc59C5CCcb3e7f76Ad36b512b223776
to 0xcc188bbcEA06832AB2ce4F5677D07b922a96bF0a
amount 5
```

As you can see, all three parameters are collected upon the **Transfer** button being clicked. Now is an excellent time to implement the transaction functionality.

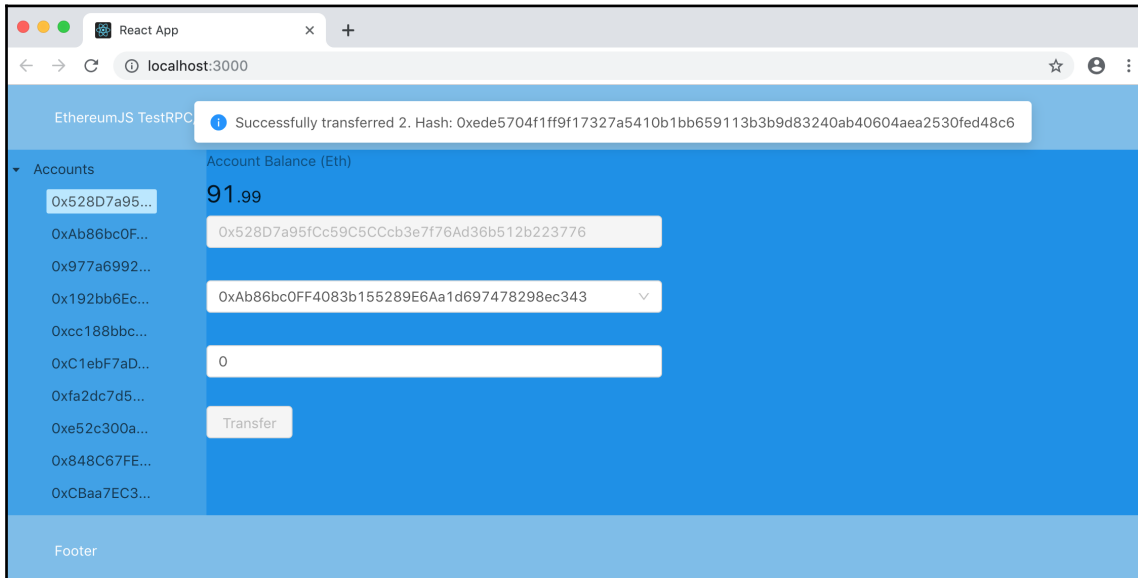
12. Replace `onTransferClick` with the following code:

```
const onTransferClick = () => {
  const transaction = {
    from: account,
    to: targetAccount,
    value: web3.utils.toWei(transferAmount, 'ether')
  };
  web3.eth.sendTransaction(transaction, function(error, hash) {
    if (error) {
      console.error('Transaction error', error);
    } else {
      message.info(`Successfully transferred ${transferAmount}.
Hash: ${hash}`);
      onSelectAccount([account]);
      setTransferAmount(0);
    }
  });
};
```

In the preceding code, we built the transaction payload and converted the number value into the expected format. Then, we used the `web3.eth.sendTransaction` API to perform the actual transaction and raised a message popup as soon as the function call succeeded.

Finally, we reloaded the current account information by invoking the `onSelectAccount` handler. We also reset the current transfer amount input so that our users can't perform another transaction by mistake.

13. Fill in all of the form parameters and click on the **Transfer** button. You should see the following output:



We have successfully transferred some Ether from one account to another.



For more details about the `message` component, as well as documentation and examples, please refer to <https://ant.design/components/message/>.

Congratulations on completing this section! Now, you have a working digital wallet application. Finally, let's learn how to prepare the application for packaging.

Packaging the application for distribution

In this section, we are going to learn how to package the Electron application for distribution.

Let's start by installing the `electron-builder` library and configuring the package scripts:

1. Run the following command to install the library:

```
npm install -D electron-builder
```



You can find more information, examples, and documentation about the `electron-builder` library by visiting the official repository: <https://github.com/electron-userland/electron-builder>.

2. Update the `package.json` file and include the `pack-app` and `dist-app` scripts. Also, provide the homepage address, as shown in the following code:

```
"homepage": "./",
"scripts": {
  "electron": "electron .",
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "pack-app": "electron-builder --dir -c.mac.identity=null",
  "dist-app": "electron-builder"
},
```

We are going to use the `pack-app` script for testing purposes. With the `--dir` switch, the Electron Builder tool generates the output without actually packaging it for production use. Developers typically use this mode to test the packaging and structure. For production use, you should use the `dist-app` script.

Don't run the scripts just yet—we still need to configure the project so that we can use the compiled resources (instead of the `http://localhost:3000` address) with live reloading for development purposes.

3. Update the `package.json` file and append the Electron builder configuration, as shown in the following code:

```
"build": {
  "files": [
    "build/**/*"
  ]
}
```

4. Install the `electron-is-dev` library with the following command:

```
npm install electron-is-dev
```

This library allows you to detect whether your Electron project is running in development mode or whether the code is being executed by the packaged application.



You can find the source code for this library in the following GitHub repository: <https://github.com/sindresorhus/electron-is-dev>.

5. Update the `public/electron.js` file, as follows:

```
const { app, BrowserWindow } = require('electron');
const isDev = require('electron-is-dev');

function createWindow() {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    },
    resizable: false
  });

  win.loadURL(
    isDev
      ? 'http://localhost:3000'
      : 'index.html'
  );
}

app.on('ready', createWindow);
```

6. Update the `public/index.html` file and add the following meta element:

```
<meta
  http-equiv="Content-Security-Policy"
  content="script-src 'self' 'unsafe-inline';"
/>
```

7. Now, you can create and test the application package by running the following commands:

```
npm run build
npm run pack-app
```

For production use, you may need to run the following commands:

```
npm run build
npm run dist-app
```

We have successfully prepared the installation packages for the distribution of our Electron application. If you want to find out more about the *Electron Builder* application, please refer to the official repository: <https://github.com/electron-userland/electron-builder>.

Summary

In this chapter, you successfully created a basic digital wallet application that talks to the Ethereum blockchain and provides Ether transfer capabilities between your accounts. Now, you can build Electron applications that are powered by the stunning Ant Design language and its vast component library.

We also walked through the process of setting up a local Ethereum blockchain that we can use across multiple projects without putting real money at risk. You can find the final source code for this project in the `crypto-wallet` folder. Good luck with extending the application and providing even more great features!

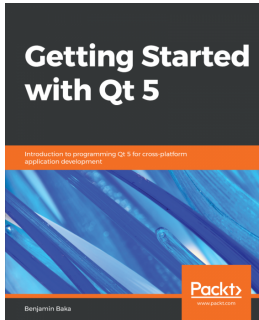
Now, you have reached the end of this book. I hope you have liked building Electron projects with various features backed by modern and popular frameworks.

Throughout this book, you have how to create and package Electron applications. We also addressed window management, keyboard handling, and native application menus. You built multiple projects that demonstrate the ease of Electron development.

I wish you luck in using the knowledge you have attained from this book to build feature-rich and cross-platform applications that are powered by web technologies and the Electron framework.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

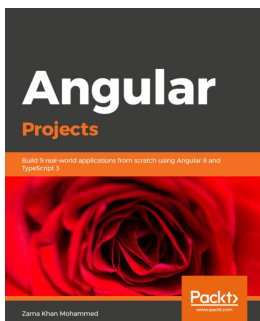


Getting Started with Qt 5

Benjamin Baka

ISBN: 978-1-78995-603-0

- Set up and configure your machine to begin developing Qt applications
- Discover different widgets and layouts for constructing UIs
- Understand the key concept of signals and slots
- Understand how signals and slots help animate a GUI
- Explore how to create customized widgets along with signals and slots
- Understand how to subclass and create a custom windows application
- Understand how to write applications that can talk to databases.



Angular Projects

Zama Khan Mohammed

ISBN: 978-1-83855-935-9

- Set up Angular applications using Angular CLI and Angular Console
- Understand lazy loading using dynamic imports for routing
- Perform server-side rendering by building an SEO application
- Build a Multi-Language NativeScript Application with Angular
- Explore the components library for frontend web using Angular CDK
- Scale your Angular applications using Nx, NgRx, and Redux

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

- account balance
 - displaying 395, 396, 397
- analytics 242, 243
- analytics services
 - creating 244
- analytics-tracking
 - setting up 250
- Angular Material components
 - application, testing with material toolbar 290
 - Browser Animations module, adding 287
 - creating 287
 - default theme, configuring 288
 - Material Icons library, adding 288
 - navigation bar, adding 288, 289
 - reference link 106
 - using 106, 107
- Angular Material
 - installing, for modifications 107, 108
- Angular project scaffold
 - generating 92, 93, 94
- Angular project
 - creating 275, 276
 - Electron shell, configuring 277, 279
 - integrating, with Electron 95, 97, 98, 99
- Angular Quickstart
 - URL 91
- Angular
 - routing 110, 111, 113, 114, 115, 116, 117
- Ant Design library
 - integrating 376, 377, 378, 379
 - URL 376
- application menu
 - integrating 38, 39, 40
 - Open and Save features, integrating with 348, 349, 350, 351

- application name
 - configuring, in menu 49, 50, 51
- application programming interfaces (APIs) 7
- application title
 - modifying 88
- application updates
 - handling 262, 263, 264
- application windows
 - dragging 165
 - hiding, on startup 178, 179
- application
 - behavior, testing 175, 176
 - integrating, with system tray 176, 177, 178
- automatic update configuration
 - reference link 81
- automatic updates
 - supporting 77, 78, 79, 81, 82, 84
 - testing 85, 86, 87

B

- background images
 - rendering 190, 191, 192
- blueprint UI toolkit
 - URL 128
- book generator
 - Docker, installing 352, 354, 356
 - documents, sending to Node.js process 359, 360, 361, 362
 - Pandoc container, executing 356, 358
 - setting up 351, 352

C

- chat component placeholder
 - preparing 295, 296
- chromes 155
- code
 - updating, with React Hooks 339, 340

- conditional loading
 - setting up 104, 105, 106
- confirmation messages
 - sending, to main process 54, 55
- custom menu item
 - creating 40, 41, 42, 43

D

- desktopCapturer API
 - reference link 168
 - using 167, 168
- development environment
 - preparing 8
- dialogs
 - reference link 64
- digital wallet application
 - account balance, displaying 395, 396, 397
 - Ant Design library, integrating 376, 377, 378, 379
 - Ether, transferring to another account 397, 399, 400, 402, 403, 404
 - Ethereum JavaScript API, configuring 384, 385, 386
 - Ethereum node information, displaying 386
 - Ethereum node information, rendering in header 387, 388
 - list of accounts, rendering 390, 391, 392, 394
 - menu, integrating 389, 390
 - packaging, for distribution 404, 406, 407
 - personal Ethereum blockchain, setting up 379, 380, 381, 382, 383, 384
 - project scaffold, generating with React 374, 375, 376
- Docker commands, invoking from Electron
 - about 362
 - markdown text, saving to local drive 363, 364, 365, 366
 - markdown text, sending to Node.js process 362
- Docker Desktop
 - reference link 354
- Docker
 - installing 352, 354, 356
 - URL 352
- documents
 - sending, to Node.js process 359, 360, 361, 362

- drag and drop support
 - adding 74, 75, 76
- drag and drop, HTML5
 - reference link 75

E

- eBook generation process
 - reference link 371
- editor component
 - installing 330, 332, 333, 334, 335
 - integrating 34
- editor-event 54
- ejection process
 - reference link 330
- Electron application, building with Angular
 - about 91, 92
 - Angular Material components, using 106, 107
 - Angular project scaffold, generating 92, 93, 94
 - Angular project scaffold, integrating with Electron 95, 97, 98, 99
 - Angular, routing 110, 111, 113, 114, 115, 116, 117
 - conditional loading, setting up 104, 105, 106
 - live reloading, configuring 99, 100, 101
 - Material Toolbar component, adding 108, 109, 110
 - modifications, creating by Angular Material installation 107, 108
 - production builds, setting up 102, 103, 104
 - testing, in browser 102
- Electron application, building with React
 - about 117
 - application menu, adding 129, 130, 131
 - blueprint UI toolkit, using 128
 - conditional loading, setting up 127, 128
 - final touches 133, 134
 - live reloading 122, 123, 124, 125
 - production builds, setting up 125, 126, 127
 - React project, generating 118, 119, 120, 121, 122
 - routing, adding 131, 132, 133
- Electron application, building with Vue.js
 - about 135, 136, 137, 138, 139
 - application toolbar, creating 149, 150, 152
 - conditional loading, setting up 144, 145, 146

- live reloading 141, 142, 143
- production builds 143, 144
- reference link 135
- routing, adding 146, 147, 148
- Vue configuration file, creating 139, 140, 141
- Vue Material, configuring 148, 149

Electron applications

- licenses, checking 271
- Nucleus, using for 245, 246

Electron Builder

- reference link 407

Electron shell

- configuring 277, 279
- integrating 337, 338, 339
- verifying 325, 326

Electron-based applications

- reference link 7

electron-builder

- about 20
- reference link 20

Electron

- about 7
- Angular project, integrating with 95, 97, 98, 99

environment

- setting up, for macOS 9
- setting up, for Ubuntu Linux 11
- setting up, for Windows 12

ePub books

- generating 369, 370, 371, 372

error handling 294

Ethereum JavaScript API

- configuring 384, 385, 386

Ethereum node information

- displaying 386
- obtaining 386
- rendering, in header 387, 388

F

file menu

- creating 72, 73, 74

files

- loading 342, 343, 344, 345
- loading, from local system 68, 69, 70, 71
- saving 346, 347
- saving, to local system 59, 61, 62, 63

final structure

- reviewing 237, 238, 239

Firebase account

- creating 280, 281, 282, 283, 284

Firebase application

- creating 284, 285, 286

Firebase authentication

- login dialog, connecting to 297

frameless windows

- configuring 155, 156, 157
- reference link 155

Free Music Archive website

- reference link 209

G

game project

- configuring 183, 184, 185, 186, 187

Git

- installing, on macOS 9
- installing, on Ubuntu 11
- installing, on Windows 12, 13

global keyboard shortcuts

- registering 179, 180, 181

global server settings

- loading 264, 266

Google Firebase, pricing plans

- reference link 284

Google Firebase

- URL 280

Google Material Icons

- download link 225

group list

- real-time updates, testing 312, 314
- rendering 310, 311, 312

group messages page

- implementing 314, 316

group messages

- displaying 316, 318, 319
- ideas, for enhancements 325
- message list interface, updating 324
- query performance, improving 320, 321
- sending 321, 322, 323

H

hello world application

creating 15, 16, 17, 18, 19, 20

Hello World example

running 187, 188, 189

K

keyboard accelerators 46, 47

keyboard input

handling 195, 196, 197

keyboard shortcuts

controlling 340, 341

files, loading 342, 343, 344, 345

files, saving 345, 347

Keygen

URL 267

L

license policies

about 266, 267

creating 267, 268, 269, 270

licenses

checking 266, 267

checking, in Electron application 271

list of accounts

rendering 390, 391, 392, 394

list query

reference link 325

live reloading

about 122

configuring 99, 100, 101

local system

files, loading from 68, 69, 70, 71

files, saving to 59, 61, 62, 63

locked mode 306

login dialog

building 290, 291, 292

chat component placeholder, preparing 295, 296

connecting, to Firebase authentication 297

demo accounts, creating 300, 301

error handling 294

integrating, with Firebase 302, 303, 305

Material interface, implementing 292, 293, 294

sign-in provider, enabling 297, 298, 299

Long-Term Support (LTS) version 10, 14

M

macOS, additional options

about 158

customButtonsOnHover titleBarStyle, using 159, 160

hidden titleBarStyle, using 158, 159

hiddenInset titleBarStyle, using 159

macOS

environment, setting up for 9

Git, installing on 9

Node.js, installing on 10

reference link 356

main process

confirmation messages, sending to 54, 55

Markdown Editor application

component, integrating 34, 35, 36, 37

project, configuring 31, 32, 33

Material Design Icons

download link 220, 222

Material interface

implementing 292, 293, 294

Material Toolbar component

adding 108, 109, 110

URL 108

menu item roles

defining 43, 44

reference link 44

menu items

hiding 51, 52, 53

menu separators

providing 45, 46

menu

application name, configuring in 49, 50, 51

messages

sending, between processes 53, 54

sending, to renderer process 56, 57

Monaco Editor

reference link 357

music metadata

displaying 229, 230, 231, 232, 233

music player application

user interface, improving 234, 235, 236

music player component

- AmplitudeJS elements, using 213
- buttons, styling 216, 217, 218, 219
- exploring 207, 209
- global pause button, implementing 215, 216
- global play button, implementing 214, 215
- global play/pause button, implementing 216
- music files, downloading 209, 210, 211
- setup, providing 212, 213

N

- Node Package Manager (NPM) 10
- Node.js process
 - documents, sending 359, 360, 361, 362
- Node.js
 - installation, verifying 15
 - installing, on macOS 10
 - installing, on Ubuntu 11, 12
 - installing, on Windows 14
 - URL 14
- Nucleus account
 - creating 246, 247, 248, 249, 250
 - sign up link 246
- Nucleus Electron library
 - installing 252, 253, 254
- Nucleus
 - reference link 243
 - using, for Electron applications 245, 246

O

- offline mode
 - supporting 261, 262

P

- packaging
 - for macOS 21, 22, 23
 - for multiple platforms 20, 21
 - for Ubuntu 24, 25, 26
 - for Windows 26, 27, 28
- Pandoc container
 - executing 356, 358
- pandoc-docker
 - reference link 356
- Pandoc
 - URL 327, 358
- PDF books

- generating 366, 368
- personal Ethereum blockchain
 - setting up 379, 380, 381, 382, 383, 384
- platform-specific menus 48, 49
- playback control buttons
 - exploring 219
 - mute buttons 222, 223, 224
 - stop button 220, 221, 222
 - unmute buttons 222, 223, 224
 - volume buttons 224, 226, 227
- primary display size
 - calculating 169, 170
- process.platform
 - reference link 49
- processes
 - messages, sending between 53, 54
- production builds
 - setting up 102, 103, 104
- project scaffold
 - creating 205, 206, 207
- project structure
 - creating 328
 - editor component, installing 330, 332, 333, 334, 335
 - Electron shell, integrating 337, 338, 339
 - React application, generating 328, 329
 - web application, testing 335, 336, 337
- project
 - creating, with tracking support 250, 252

Q

- Querying Lists API
 - reference link 318

R

- React application
 - generating 328, 329
- React Hooks
 - reference link 339
 - used, for updating code 339, 340
- real-time analytics data
 - inspecting 254, 255, 256, 257, 258
 - users, identifying 259
- real-time user statistics
 - verifying 260, 261

Realtime Database

- configuring 306, 307

- demo groups 309

- demo groups, creating 308

renderer process

- messages, sending to 56, 57

S

save dialog

- using 63, 64, 65, 66, 67

screen size

- fitting 37, 38

Screenshot Snipping Tool

- project, preparing 154, 155

services

- using 243

shell object

- reference link 364

Simple Markdown Editor

- URL 34, 36

snip toolbar button

- adding 166, 167

solution

- creating 243

song progress bar

- implementing 227, 228

sprite coordinates

- controlling 198, 199, 200, 201

sprite speed

- controlling 201, 202

sprites

- flipping, based on their direction 197, 198

- rendering 192, 193

- scaling 193, 194

system tray

- application, integrating with 176, 177, 178

T

- test mode 306

third-party analytics services

- creating 245

- using 244

thumbnail image

- cropping 172, 173, 174

- generating 170, 171

- resizing 172, 173, 174

- saving 170, 171

toggle bold menu

- wiring 58, 59

toolbar icons, Markdown editor

- reference link 59

tracking 242, 243

tracking per user request

- disabling 259, 260

transparent windows

- about 160, 161, 162, 164

- limitation, reference link 161

U

Ubuntu Linux

- environment, setting up for 11

Ubuntu

- Git, installing on 11

- Node.js, installing on 11, 12

- Visual Studio Code, installing for 9

URL.createObjectURL static method

- reference link 346

V

Visual Studio Code

- installing 8, 9

- installing, for Ubuntu 9

- URL 8

Vue configuration file

- reference link 140

W

web application

- testing 335, 336, 337

window resizing

- preventing 192

Windows

- environment, setting up for 12

- Git, installing on 12, 13

- Node.js, installing on 14