IONIC 4

BY **ED FREITAS**

Ionic 4 Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

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

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

Information is plentiful but harder to digest

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

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

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

The *Succinctly* series

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

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

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

The best authors, the best content

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

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ed Freitas is a consultant on software development applied to customer success, mostly related to financial process automation, accounts payable processing, and data extraction.

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

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

Acknowledgments

Many thanks to all the people from the amazing Syncfusion team who helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Jacqueline Bieringer from Syncfusion and Dr. James McCaffrey from Microsoft Research. Thank you.

This book is dedicated to my father, who passed away when I was writing it. Not only did my father give me the gift of life, but he also taught me the work ethic I have, to be entrepreneurial, to always be learning, to not take things for granted, to not be afraid of being an immigrant, to explore the world by living in different countries, to be responsible, to treat everyone with utmost respect and to be respected, to help everyone that crosses your path in life, to seek the common good, to seek the best in people, and to never take advantage of anyone for your benefit and their detriment. He taught me that we all fail many times, but we can always get back up and be better.

This year that you turned 81, you still wanted to start another adventure—if your body would have been strong enough, I'm sure you would have done it and immigrated again to greener fields. Your life was a living testament of all these values. You were amazing, and an inspiration.

All your love and dedication are something I will always cherish and never forget. Thank you, Papa, for everything you did—in the name of your loving wife "Cuchi," two sons, daughter, two daughters-in-law, son-in-law, and four grandchildren. You will always be present in our memories, our hearts, and in the memories of all the people that you positively influenced, helped, and touched during your life with your kindness and wisdom, forged by the university of life.

Descansa en paz mi viejito lindo, estaremos siempre contigo y tu con nosotros.

Ionic for Everyone

Ionic is one of the most exciting frameworks that exists for building cross-platform mobile apps. It is an open-source UI toolkit that allows developers to build high-quality and high-performing mobile apps using web technologies such as HTML, CSS, and JavaScript.

Ionic is primarily focused on the user interface—it is used for building and deploying apps that work across multiple platforms, such as native iOS, Android, desktop, and the web, as progressive web apps.

Ionic Succinctly covers the basics of the Ionic Framework using Angular. Ionic 4 is an evolution of the original Ionic Framework and represents the culmination of more than two years of research. It takes Ionic from a mobile-centric framework based on Angular into a powerful UI design system and app development toolset that is JavaScript-framework agnostic.

The main idea behind the development of Ionic 4 was to make Ionic available for every web developer. This was accomplished by rebuilding the framework as web components that use custom elements and shadow DOM APIs, which are available in all modern mobile and desktop browsers, targeting standard web APIs rather than third-party ones.

This means Ionic 4 departs from being a UI framework for building cross-platform mobile apps with Angular using web technologies, to become a web-based UI design system and application framework for any web developer, regardless of which JavaScript library or framework they choose to work with.

One of the most interesting aspects of Ionic 4 is its excellent performance. Each component in Ionic 4 is a web component that has been optimized for load and render performance. The increased performance of Ionic 4 components makes the framework ideal for developing progressive web apps, which are in high demand and popularity these days.

Ionic 4 can reach the performance standards set by Google for progressive web apps due to asynchronous component loading and delivery—consisting of smart collections of tightly packed components that are lazy-loaded and optimized for frequent use.

With *Ionic 4 Succinctly*, the objective is to focus on progressive web apps and see how we can use Ionic 4 to build one, using Vue as our JavaScript framework—and departing from the Angular-centric approach of *Ionic Succinctly*.

Throughout this book, we'll see how progressive web apps and Vue have first-class support in Ionic 4, and how they can be used to go beyond cross-platform mobile development.

Chapter 1 App Fundamentals

Project overview

The application we'll be building throughout this book is a progressive web app (PWA) that we can use to track flight details. We'll do this by using a package called **@ionic/vue** that will allow us to use Ionic 4 components within our Vue app. Essentially, we'll be building a PWA using Vue based on Ionic 4 components with a mobile "look and feel" that can also work offline.

Throughout this book, we will build the application, deploy it via the Firebase development platform, and implement PWA features by creating a service worker script. The service worker will cache all application assets and requests that the application will retrieve from a flight tracking API (that I built) for offline viewing later.

Installation

Before we can create our Ionic 4 PWA, we'll need the Vue CLI (command line interface) installed. This requires NPM (Node Package Manager), which requires installation of Node.js.

To install Node.js, simply go to the Node.js website and download the **LTS** (long-term support) or the latest current version.
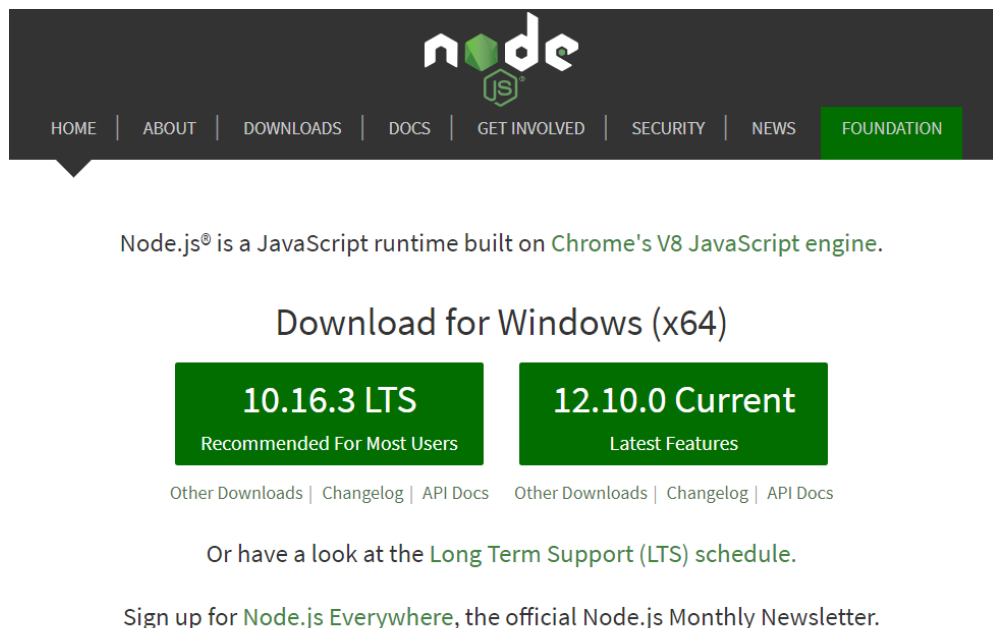
Figure 1-a: Node.js Website

The installation of Node.js is very simple and consists of a few steps that can be easily executed using the intuitive step-by-step wizard. Figure 1-b shows an example of the installation process.
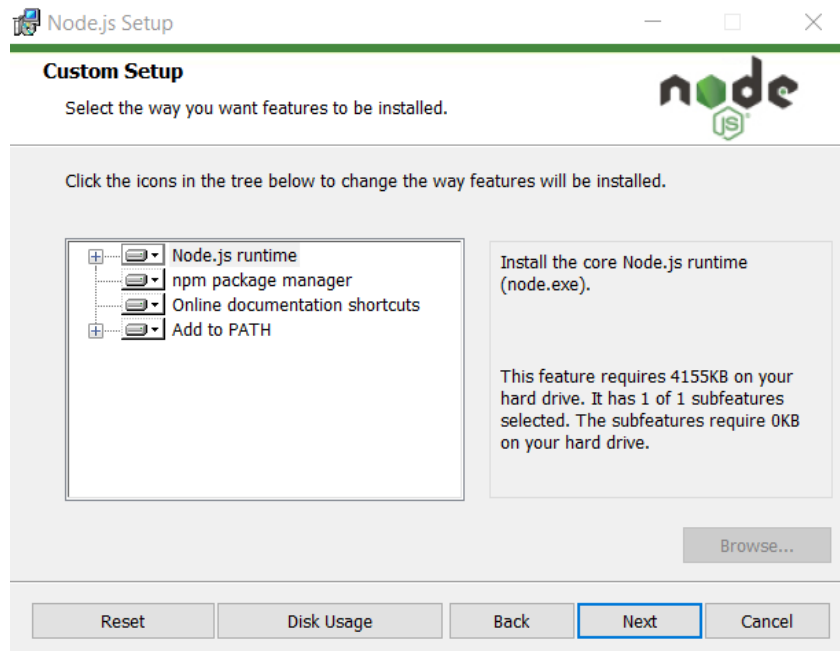
Figure 1-b: Node.js Installation Wizard

Once we've installed Node.js, we need to install the Vue CLI globally on our system. We can do this by opening the command line or terminal and typing the following command.

Code Listing 1-a: Install Vue CLI Command

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

You can run the following command to check which version of the Vue CLI was installed on your machine.

Code Listing 1-b: Check Vue CLI Version

```
vue --version
```

With the Vue CLI installed, we are ready to create an application. In my case, I already had the Vue CLI installed from a previous project, so your version might differ slightly from mine.

It is not mandatory to have the latest version of the Vue CLI installed to create the application we'll be building throughout this book, but it's still good to be up to date.

Creating the app

Creating the application with the Vue CLI is very easy. All we need to do is run the following command.

Code Listing 1-c: Creating the App with the Vue CLI

```
vue create flight-info-pwa
```

The name of our application will be **flight-info-pwa**. To execute the command, we'll be required to choose a preset. In my case, I've chosen the **default** preset (by pressing Enter), which includes **babel** and **eslint**.
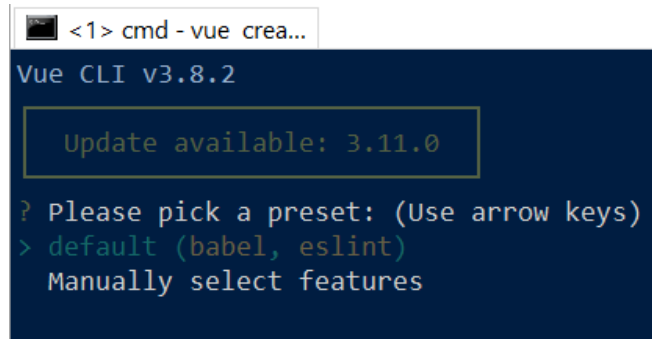
Figure 1-c: Choosing a Preset

Once selected, the Vue CLI installs the required modules. The creation of the application is then finalized.

I'll be using [Visual Studio Code](#) (VS Code) as my editor of choice, but feel free to use any other that you feel comfortable with. VS Code is easy to install, and just as easy to use. Regardless of the editor you choose to use, you'll be able to follow along with all subsequent steps.

With the application created, go into the project root folder using the following command in the command line.

Code Listing 1-d: Going into the Application Folder

```
cd flight-info-pwa
```

Once there, type **code .** from the command line to open VS Code on that directory. This is how it looks on my machine.
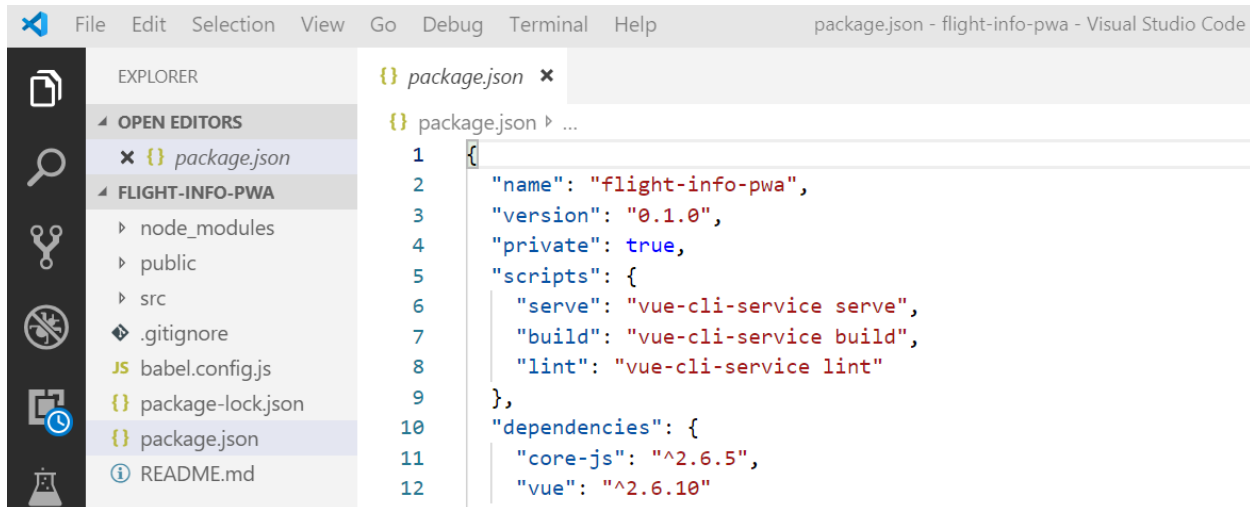
Figure 1-d: The Application Opened with VS Code

We can close the command prompt we've opened and switch to the built-in terminal that ships with VS Code by clicking on the **Terminal** menu item, and then on the **New Terminal** option. We can see this as follows.
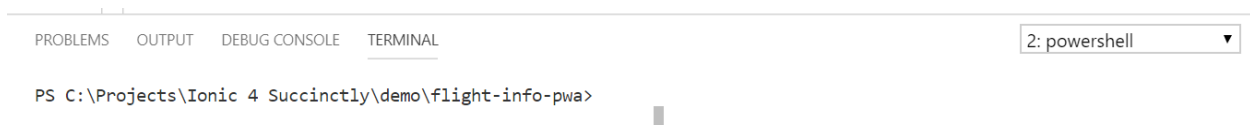
Figure 1-e: The Built-In Terminal in VS Code

From the built-in terminal, we can install any remaining components our application might require.

Additional dependencies

One of those components is Vue Router—let's get it installed. We can do this by running the following command.

Code Listing 1-e: Installing Vue Router

```
vue add router
```

When running this command, you will be prompted to use the router's history mode—when prompted, enter **Y** (Yes).

The default mode for Vue Router is hash mode, which uses the URL hash to simulate a full URL, so that the page won't be reloaded when the URL changes.

To avoid the hash, we can use the router's history mode, which leverages the **history.pushState** API to achieve URL navigation without a page reload.

Once the command has finished executing, you should see the following output on the **Terminal** tab in VS Code.


```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

  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

  You should review these changes with git diff and commit them.

PS C:\Projects\Ionic 4 Succinctly\demo\flight-info-pwa> []
```

Figure 1-f: Terminal Output—After Installing Vue Router

With Vue Router installed, let's next install the Ionic Vue (**@ionic/vue**) package, which is essential if we want to develop an Ionic application using Vue. To do this, we need to execute the following installation command.

Code Listing 1-f: Installing Ionic Vue

```
npm i @ionic/vue
```

Once it's installed, you should see the following output on the Terminal tab within VS Code.


```
+ @ionic/vue@0.0.4
added 3 packages from 6 contributors and audited 24360 packages in 11.794s
found 0 vulnerabilities

PS C:\Projects\Ionic 4 Succinctly\demo\flight-info-pwa> []
```

Figure 1-g: Terminal Output—After Installing Ionic Vue

Now, let's run the application to see how it looks. We can do this by executing the following command.

Code Listing 1-g: Running the App

```
npm run serve
```

After executing the command, the application will be shown in the browser; we can corroborate this by checking the **Terminal** output.

Figure 1-h: Terminal Output—App Running

If we open the browser and go to the URL indicated by the console output, we should be able to see the application running.
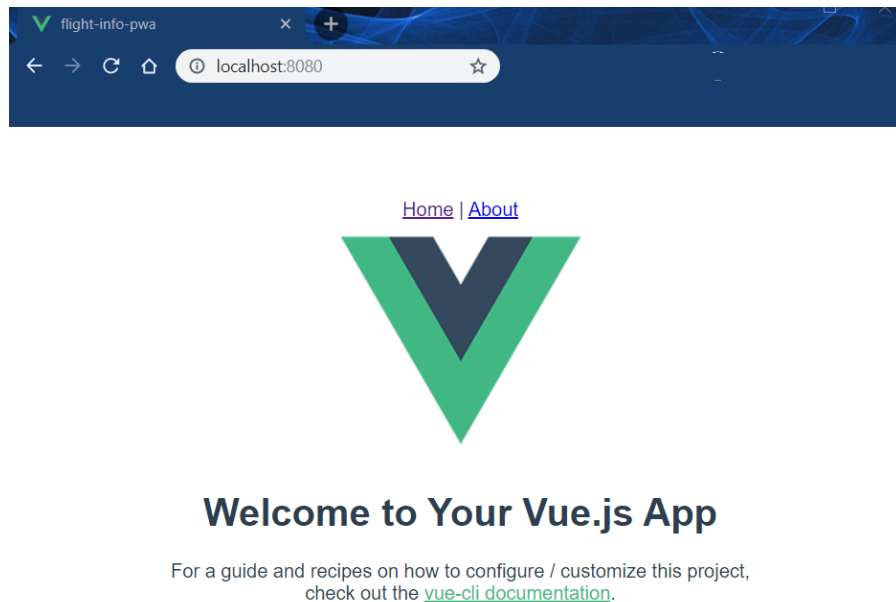
Figure 1-i: App Opened in the Browser

As you can see, it's just the default Vue app. There's nothing in the app yet that relates to Ionic, so we will have to set that up, which we will do next.

Adding Ionic: main.js and router.js

We are now ready to add Ionic 4 components to our application. To do this, let's open the **main.js** file, which sits under the **src** folder of our project, using VS Code.
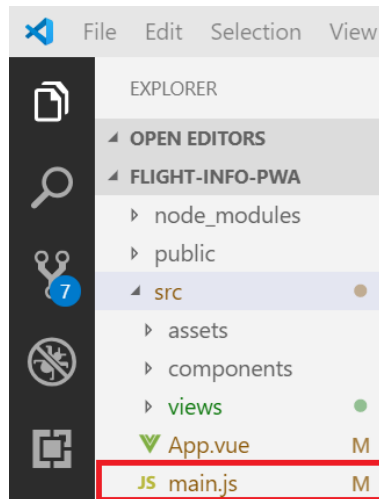
Figure 1-j: The main.js File within the src Folder

The out-of-the-box code contained within **main.js** is shown in the following.

Code Listing 1-h: Default main.js Code

```javascript
import Vue from 'vue'
import App from './App.vue'
import router from './router'

Vue.config.productionTip = false

new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

To use Ionic, we need to add a reference to Ionic Vue, add the built-in Ionic styling, and reference Ionic. These changes are highlighted in bold in Code Listing 1-i.

Code Listing 1-i: Modified main.js Code (Ionic Included)

```javascript
import Vue from 'vue'
import App from './App.vue'
import router from './router'

import Ionic from '@ionic/vue'
import '@ionic/core/css/ionic.bundle.css'

Vue.use(Ionic)

Vue.config.productionTip = false
```

```
new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

These are all the changes required for the **main.js** file. Because we've added the **ionic.bundle.css** reference to the **main.js** file, we need to install the Ionic Icons package by running the following command.

Code Listing 1-j: Command to Install the Ionic Icons Package

```
npm install ionicons@4.5.9-1 --save-dev
```

Installing this package is necessary to avoid running into the *Ionic/vue ionicons error #18640* issue, which is described here.

Next, we need to modify **router.js**, change the default Vue router, and replace it with the Ionic router. Before we make any more changes, let's have a look at how the out-of-the-box **router.js** code looks.

Code Listing 1-k: Default router.js Code

```
import Vue from 'vue'
import Router from 'vue-router'
import Home from './views/Home.vue'

Vue.use(Router)

export default new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/about',
      name: 'about',
      // route level code-splitting
      // this generates a separate chunk (about.[hash].js) for this route
      // which is lazy-loaded when the route is visited.
      component: () => import('./views/About.vue')
```

```
    }
  ]
})
```

All we need to do with this code is replace the Vue Router references with the Ionic Router ones. The code changes are highlighted in bold in Code Listing 1-l.

Code Listing 1-l: Modified router.js Code

```
import Vue from 'vue'
import { IonicVueRouter } from '@ionic/vue'
import Home from './views/Home.vue'

Vue.use(IonicVueRouter)

export default new IonicVueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    }
  ]
})
```

Note the following changes:

- **import Router from 'vue-router'** was replaced **with import { IonicVueRouter } from '@ionic/vue'**.
- **Vue.use(Router)** was replaced with **Vue.use(IonicVueRouter)**.
- The **'about'** route was removed from the code, since we won't be using it.

By making these changes to **main.js** and **router.js**, we have added Ionic to our application. However, if we view the application in the browser, it will appear as if nothing has changed. To view the changes, we need to modify **App.vue**—which is what we will do next.

Modifying App.vue

The **App.vue** file is the main Vue component file of our application. It defines the main HTML template and markup that our application will display when running.

Let's have a look at how the out-of-the-box code of **App.vue** looks.

```
<template>
  <div id="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;
  margin-top: 60px;
}
</style>
```

Now, let's modify the **App.vue** code, so we can later see Ionic in action. We'll remove the styling and the content of **<div id="app">**. Below is the modified **App.vue** code.

Code Listing 1-n: Modified App.vue Code

```
<template>
  <div id="app">
    <ion-app>
      <ion-vue-router/>
    </ion-app>
  </div>
</template>
```

If we now view the app in the browser, we will see the same content, but displayed differently.
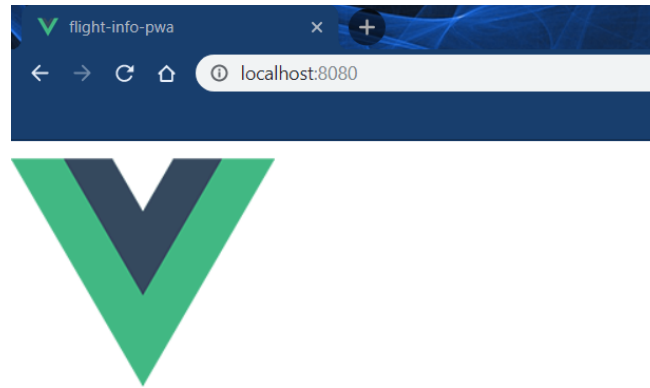
Figure 1-k: The App Running (Different Layout)

Even though we added the `ion-app` and `ion-vue-router` components to the markup, we still haven't added any Ionic UI components. This is what we'll do next.

Essential Ionic UI components

One of the most exciting parts of developing a new application is seeing how it takes shape—for that, creating the UI is an essential aspect of an app's development life cycle. To create the UI, we need to modify the **Home.vue** file located within the **views** subfolder of our project.

First, let's have a look at the out-of-the-box code contained within **Home.vue**.

Code Listing 1-o: Default Home.vue Code

```
<template>
  <div class="home">
    <img alt="Vue logo" src="../assets/logo.png">
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>
```

```
<script>
// @ is an alias to /src
import HelloWorld from '@/components/HelloWorld.vue'

export default {
  name: 'home',
  components: {
    HelloWorld
  }
}
</script>
```

Let's modify this code and add some fundamental UI components that will make the app look like an Ionic application.

Code Listing 1-p: Modified Home.vue Code

```
<template>
  <div class="ion-page">
    <ion-header>
      <ion-toolbar>
        <ion-title>
          Flight Info
        </ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content class="ion-padding">My App</ion-content>
  </div>
</template>

<script>

export default {
  name: 'home',
  components: {
  }
}
</script>
```

As we can see, all the content of the main **div** was replaced with Ionic UI components, and its **class** changed to **ion-page**—which is a placeholder for Ionic UI components.

The **ion-page** component is a placeholder for all Ionic UI components that are part of the same UI view (also known as a page). Therefore, the main **ion-header** and **ion-content** components are nested under **ion-page**.

Note how **ion-toolbar** is nested under **ion-header**, and how **ion-title** is nested under **ion-toolbar**. The ability to nest Ionic UI components is one of the great features of building UIs with Ionic, as it allows anyone to quickly scaffold a user interface by adding components to others.

With the essential Ionic UI components added to **Home.vue**, it's now time to understand how the application will be structured into functional components.

Structuring the application

The application will have three main functional components, each of which will be an independent .vue file. The first component will be called **Search.vue** and will be used to perform flight information searches. The second component will be called **Info.vue** and will be used to display flight information details resulting from a search. The third component will be called **Clear.vue** and will be used to clear the flight information details retrieved and displayed from a previous search.

We need to create these three Vue files within the **components** subfolder of our project. So, with VS Code, go ahead and add these files.
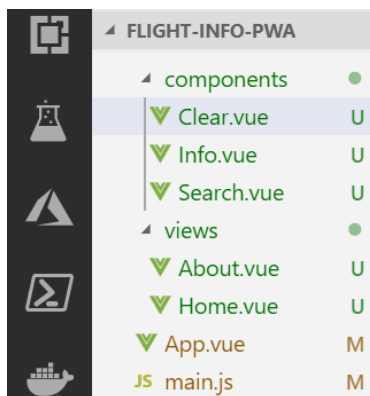
Figure 1-l: Vue Components (VS Code)

As this is going to be a PWA, we'll later create the service worker so the app can work offline. We'll later explore in depth all the technical details of PWAs. First, we'll focus and finish all the basic functional aspects of the app's UI.

Search.vue UI

Let's start off by creating the foundations of the first functional component of our application: **Search.vue**.

Code Listing 1-q: Search.vue Code

```
<template>
```

```
    <ion-grid>
        <form>
            <ion-col>
                <ion-item>
                    <ion-label>Flight number: </ion-label>
                    <ion-input name="flight"></ion-input>
                </ion-item>
            </ion-col>
            <ion-col>
                <ion-button type="submit" color="primary" expand="block">
                    Search
                </ion-button>
            </ion-col>
        </form>
    </ion-grid>
</template>

<script>
export default {
    name: 'Search'
}
</script>
```

What we have done is added **ion-grid** component and nested a **form** component within it, which will render this functional component's UI layout.

The form is made up of two columns: the first column contains an **ion-item** with an **ion-label** and **ion-input**, while the second column contains a search button, **ion-button**. The **ion-input** component will be used to enter the light number the application will retrieve and display information about. The functional component's name has also been added within the **script** section of the code.

With this done, we need to reference this functional component within **Home.vue**, so let's have a look.

Code Listing 1-r: Home.vue Referencing Search.vue

```
<template>
  <div class="ion-page">
    <ion-header>
      <ion-toolbar>
        <ion-title>
          Flight Info
        </ion-title>
      </ion-toolbar>
```

```
    </ion-header>
    <ion-content class="ion-padding">
      <Search />
    </ion-content>
  </div>
</template>

<script>
import Search from '../components/Search'

export default {
  name: 'home',
  components: {
    Search
  }
}
</script>
```

I've highlighted in bold the code changes to **Home.vue**, referencing **Search.vue**. First, the **Search** component has been embedded within **ion-content**.

Within the **script** section, the **Search** component is referenced using an **import** statement. Then, it was added to the **components** object.

If we now save the changes and check the application in the browser, we'll see that Vue's hot-reloading mechanism has rebuilt the app with the saved changes, which we can see as follows.
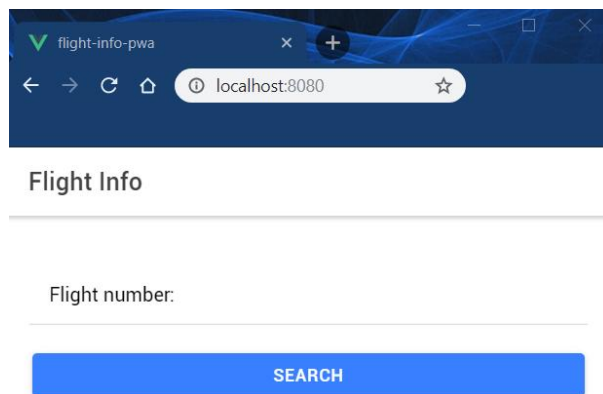
Figure 1-m: The App Running with an Ionic "Look and Feel"

With these changes, the application has a modern, fresh, and Ionic look and feel to it—which is what we want.

Something I particularly like to do when developing PWAs is to visualize them in the browser like they would look on a mobile device. I'm using Google Chrome as my predefined browser, but you may use a different one if you prefer (except Edge or Internet Explorer—you'll see why, later); the steps should be similar.

With the Chrome **Developer tools** opened, click the **Toggle device toolbar** (highlighted in Figure 1-n), which by default will display the app in Responsive mode.
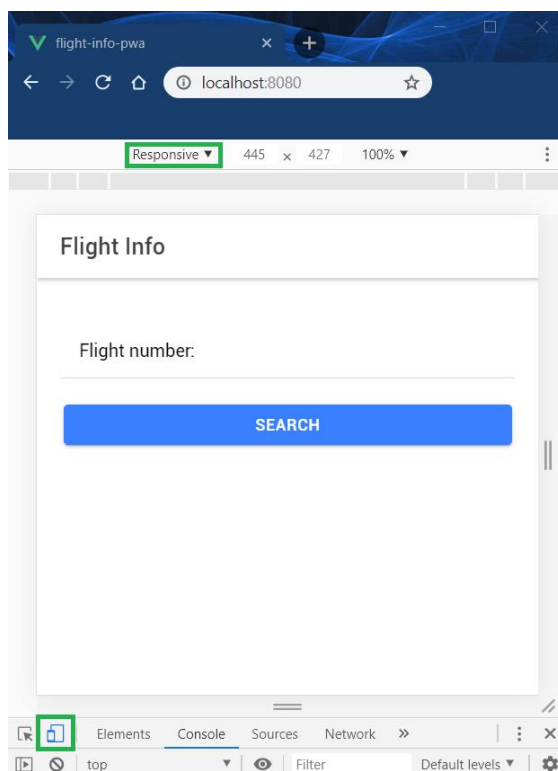
Figure 1-n: The App Running in Mobile Responsive Mode (Google Chrome)

We now have the basic structure of the application ready. Even though we haven't added any markup or code to the functional components, **Info.vue** and **Clear.vue**, we have scaffolded the foundation of our app with the markup we've just added to **Home.vue** and **Search.vue**.

Summary

Throughout this chapter, we have explored how to create the foundations of an Ionic application from scratch using Vue.

We started by installing the development tools and libraries required, then the packages and dependencies needed. We scaffolded the foundation of the application by modifying the app's main .vue and .js files. Finally, we created the files that will host the app's functional components and created the UI of one the app's main functional components, **Search.vue**.

In the next chapters, we will add the markup and the underlying logic to the rest of the app's functional components, **Info.vue** and **Clear.vue**.

At that point, we'll have a working application (without it being a PWA, which we will explore later in Chapters 4, 5, and 6.

Chapter 2 Basic App and API Logic

Quick intro

With the development environment set up, and application fundamentals covered, it's now time to create the rest of the app's functional components and all the corresponding logic for each one of them—this is what we'll do throughout this chapter.

Search.vue validation

Now that we have the UI aspect of **Search.vue** covered, let's add some logic to it. Note that if you click the **Search** button, nothing happens.

We need to add logic that validates that the flight number being submitted is not an empty string, and that it is also a valid flight number. This needs to be done before the form is submitted, so that the flight number can be validated before calling the API that will return the flight data.

Let's start by adding some logic that validates the format of the flight number before it is submitted to the flight tracking API. We can do this by using a regular expression.

Essentially, the **Search.vue** code remains the same; however, there is extra logic that has been added to perform the flight number validation. These changes have been highlighted in bold in Code Listing 2-a.

Code Listing 2-a: Search.vue—Flight Number Validation

```
<template>
    <ion-grid>
        <form @submit="onSubmit">
            <ion-col>
                <ion-item>
                    <ion-label>Flight number: </ion-label>
                    <ion-input :value="flight"
                        @input="flight = $event.target.value"
                        placeholder="such as: BA197"
                        name="flight"></ion-input>
                </ion-item>
            </ion-col>
            <ion-col>
                <ion-button type="submit" color="primary" expand="block">
                    Get Details
                </ion-button>
```

```
                </ion-col>
            </form>
        </ion-grid>
</template>

<script>
export default {
    name: 'Search',
    data() {
        return {
            flight: ''
        }
    },
    methods: {
        onSubmit(e) {
            e.preventDefault()
            const isvf =
    /^(([A-Za-z]{2,3})|([A-Za-z]\d)|(\d[A-Za-z]))(\d{1,})([A-Za-z]?)$/
                .test(this.flight)
            if (isvf) {
                console.log('Valid flight number...')
            }
            else {
                this.displayAlert()
            }
        },
        displayAlert() {
            return this.$ionic.alertController.create(
                {
                    header: 'Flight',
                    message: 'Enter a valid flight number.',
                    buttons: ['OK']
                }
            ).then(r => r.present())
        }
    }
}
</script>
```

The first change that we've made is to add the **submit** event to **form**, which is going to execute the **onSubmit** method when the form gets submitted. This occurs when **ion-button** is clicked (**type="submit"** triggers the submission of the form).

The next change to note is with the **ion-input** component. If this were a regular Vue application, we would use the **v-model** directive to bind the variable **flight** to the value of **ion-input**. By using the **v-model** directive, we can achieve two-way data binding in Vue.

Since this is not a regular Vue application, but instead we are using Ionic, we cannot use the **v-model** directive, so we assign the value of the **flight** variable to the **value** of **ion-input**. So far, we have only achieved one-way data binding.

To achieve the other part of two-way data binding, on the **input** event of the **ion-input** component, the entered value through **ion-input** (**$event.target.value**) is assigned to the **flight** variable.

The next change to note is the **data** function, which returns an object with the **flight** variable as an empty string. This is how the **flight** variable is initialized.

Then, we have the **methods** object, which is used to specify the methods that run the component's logic and validation. Within the **methods** object, we have the **onSubmit** method, which contains the main validation logic.

The first thing that is done within the **onSubmit** method is to invoke **e.preventDefault()**, which tells the browser that if the event does not get explicitly handled, its default action should not be taken as it normally would be—thus the event continues to propagate with further actions.

Next, within the **onSubmit** method, the regular expression that checks if the entered flight number is valid is invoked by calling the **test** method. If the result of that test—which is the value of the **isvf** variable—is **true**, then it is a valid flight number. Otherwise an alert is displayed; this is done my calling the **displayAlert** method.

The logic within the **displayAlert** method is very easy to understand. The **create** method from <u>**alertController**</u> is invoked by specifying the **header**, **message**, and **buttons** properties.

The **create** method returns a promise that triggers the display of the alert message by invoking the **present** method from the promise's response (**r**).

Now that we know how to validate the flight number, we need to be able to emit it upwards, so the app can process it in **Home.vue**, and from there, run a function to make a request to a flight-tracking API. Let's see how we can do this.

Emitting the flight number

We can emit the flight number upwards by calling the **$emit** method. Let's modify the **onSubmit** method within **Search.vue** to do this.

Code Listing 2-b: Search.vue—Updated onSubmit Method

```
onSubmit(e) {
  e.preventDefault()
```

```
  const isvf =
    /^(([A-Za-z]{2,3})|([A-Za-z]\d)|(\d[A-Za-z]))(\d{1,})([A-Za-z]?)$/
    .test(this.flight)
  if (isvf) {
    this.$emit('flight', this.flight)
    this.flight = ''
  }
  else {
    this.displayAlert()
    this.flight = ''
  }
}
```

The changes to the code are highlighted in bold in Code Listing 2-b. The emit gets done by invoking **$emit('flight', this.flight)**.

Once that is done, the value of the **flight** variable is cleared: **this.flight = ''**. To be on the safe side, let's also clear the value of this variable after invoking the **displayAlert** method. This way, we can prevent emitting a previous or incorrect flight number.

Following is the updated **Search.vue** code, with the latest changes.

Code Listing 2-c: Search.vue—Updated Code

```
<template>
    <ion-grid>
        <form @submit="onSubmit">
            <ion-col>
                <ion-item>
                    <ion-label>Flight number: </ion-label>
                    <ion-input :value="flight"
                        @input="flight = $event.target.value"
                        placeholder="such as: BA197"
                        name="flight"></ion-input>
                </ion-item>
            </ion-col>
            <ion-col>
                <ion-button type="submit" color="primary" expand="block">
                    Get Details
                </ion-button>
            </ion-col>
        </form>
    </ion-grid>
</template>
```

```
<script>
export default {
    name: 'Search',
    data() {
        return {
            flight: ''
        }
    },
    methods: {
        onSubmit(e) {
            e.preventDefault()
            const isvf =
        /^(([A-Za-z]{2,3})|([A-Za-z]\d)|(\d[A-Za-z]))(\d{1,})([A-Za-z]?)$/
                .test(this.flight)
            if (isvf) {
                this.$emit('flight', this.flight)
                this.flight = ''
            }
            else {
                this.displayAlert()
                this.flight = ''
            }
        },
        displayAlert() {
            return this.$ionic.alertController.create(
                {
                    header: 'Flight',
                    message: 'Enter a valid flight number.',
                    buttons: ['OK']
                }
            ).then(r => r.present())
        }
    }
}
</script>
```

Receiving the flight number

Now that we've emitted the flight number from **Search.vue**, it's time to receive and process it within **Home.vue**. This is what we'll do next.

```
<template>
  <div class="ion-page">
    <ion-header>
      <ion-toolbar>
        <ion-title>
          Flight Info
        </ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content class="ion-padding">
      <Search v-on:flight="flightDetails"/>
    </ion-content>
  </div>
</template>

<script>
import Search from '../components/Search'

export default {
  name: 'home',
  components: {
    Search
  },
  methods: {
    flightDetails(flight) {
      console.log('Flight details...')
    }
  }
}
</script>
```

I've highlighted the code changes in bold in Code Listing 2-d. All we do within the markup is include the **Search** component with the **v-on** directive, so that when the **flight** event is triggered, the **flightDetails** method can be executed.

Then, within the **methods** object, in the header of the **flightDetails** method, **flight** is passed as a variable. This is because in **Search.vue**, **this.flight** was passed when the **flight** event was emitted: **this.$emit('flight', this.flight)**.

For now, within the implementation of the **flightDetails** method, all we do is output to the Developer Tools console when the method executes.
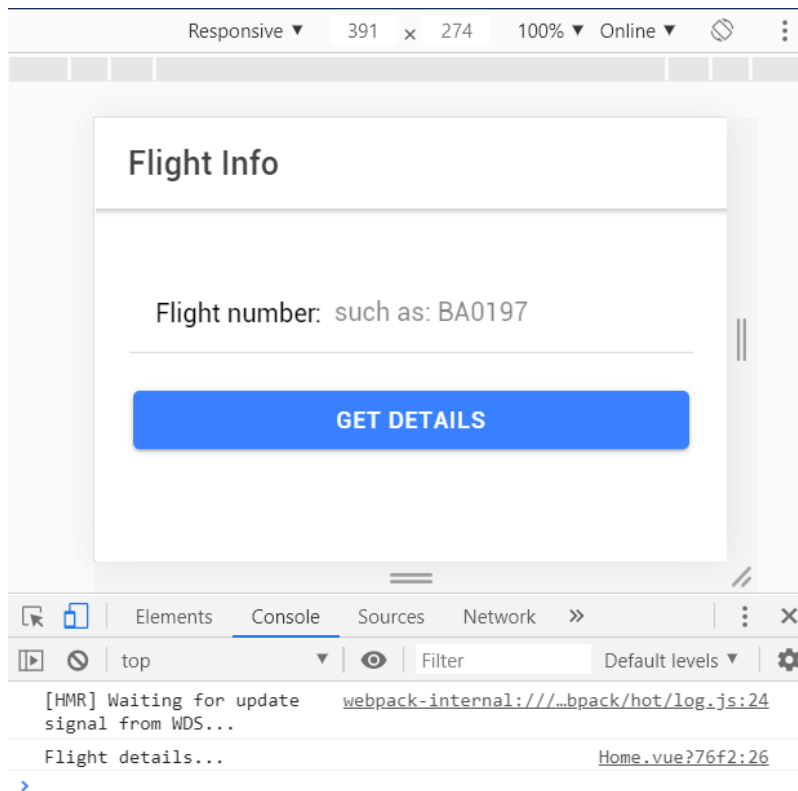
Figure 2-a: The App Running (Console Output)

Now that we know how to emit the flight number from **Search.vue** and receive it within **Home.vue**, we can focus on calling the flight tracking API and retrieving flight information.

Flight information APIs

There are many websites that provide real-time flight tracking information, such as: Flightradar24, FlightAware, and FlightStats.

In general (unlike most other APIs), flight tracking APIs are quite expensive to start with and require an upfront investment, which usually means signing up for a business account—thus being out of the reach of most independent software developers or enthusiasts.

Sites like Flightradar24 show flights all over the world in real time So, with the flight number, it's possible to know important information such as the flight date, origin, destination, departure time, estimated arrival time, the type of aircraft, flight status, and the flight route followed by the aircraft.

Essentially, it's very useful information to have about any given flight. These sites gather the information by using a combination of sources, such as data obtained from the airlines; data from transmitting stations (radar sources); and data supplied by air traffic controllers, regulators, and government organizations responsible for aviation and safety, like the Federal Aviation Administration.

If you have a look at the Flightradar24 website, you'll see a world map with an impressive number of airplanes being tracked on the map—each corresponding to a plane flying in real time.
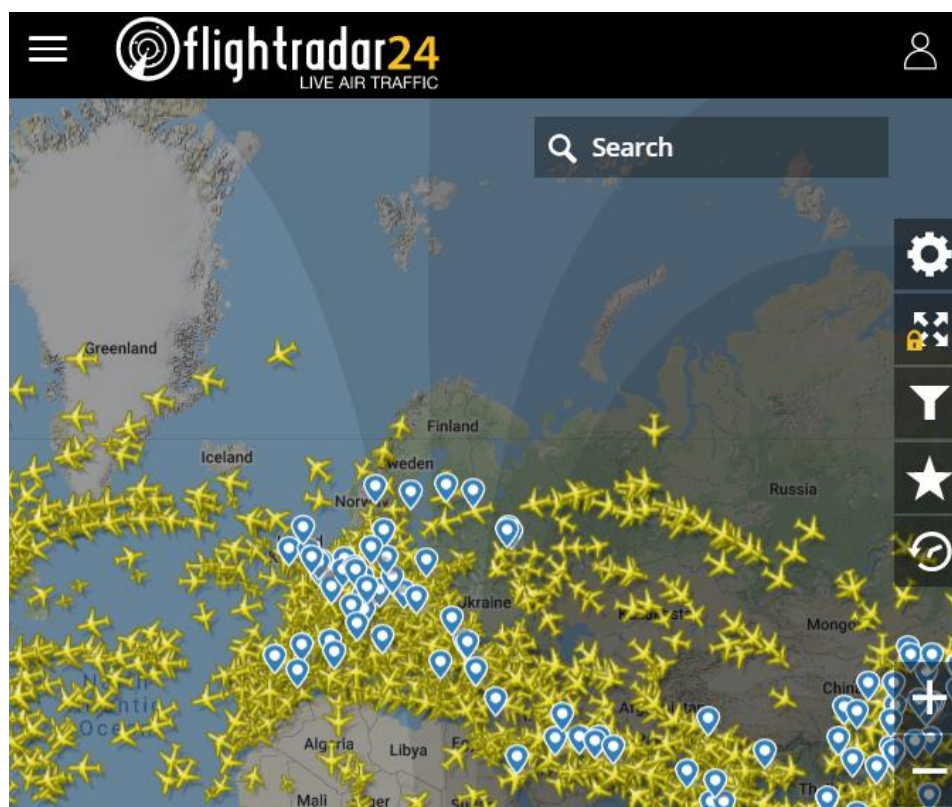
Figure 2-b: Flightradar24 Website

Given that working with sites like these is technically challenging, and that accessing this data usually requires signing up for a business account and paying high API usage fees, we won't be using any of these sites. Instead, we'll use a small API that I created, which sources the information from other highly reliable and free alternative sources, such as OpenFlights.

For creating our PWA and testing it, the small Flight API I've written uses Firebase, which contains a very small data set of flights that should be enough for testing the PWA. Although the data set is quite small, the information is accurate and fully up to date, in real time.

The way my flight API works is that the data is retrieved, curated, and verified from various sources as it is being requested—therefore, it is as accurate as the data provided by top sites like Flightradar24 and others.

Creating this API was quite a technical challenge, and something I really enjoyed as a side project for this book. However, I won't cover the steps involved to create it, but will provide it as an alternative to signing up for a business account on any of the paid sites. This way we can focus on the logic of the PWA itself, which is within the scope of this book.

It's also possible to add further data to the API by running an HTTPS request, which enables the API service to retrieve the most recent flight data for a given flight.

The API I created contains two parts. The first part consists of a service that retrieves the most recent flight data for a specific flight number, through an HTTPS call—this runs on Firebase Hosting and stores the information within Cloud Firestore.

The second part is a Firebase function invokable through an HTTPS request that queries the Cloud Firestore database and returns a JSON response containing the flight data for a specific flight number.
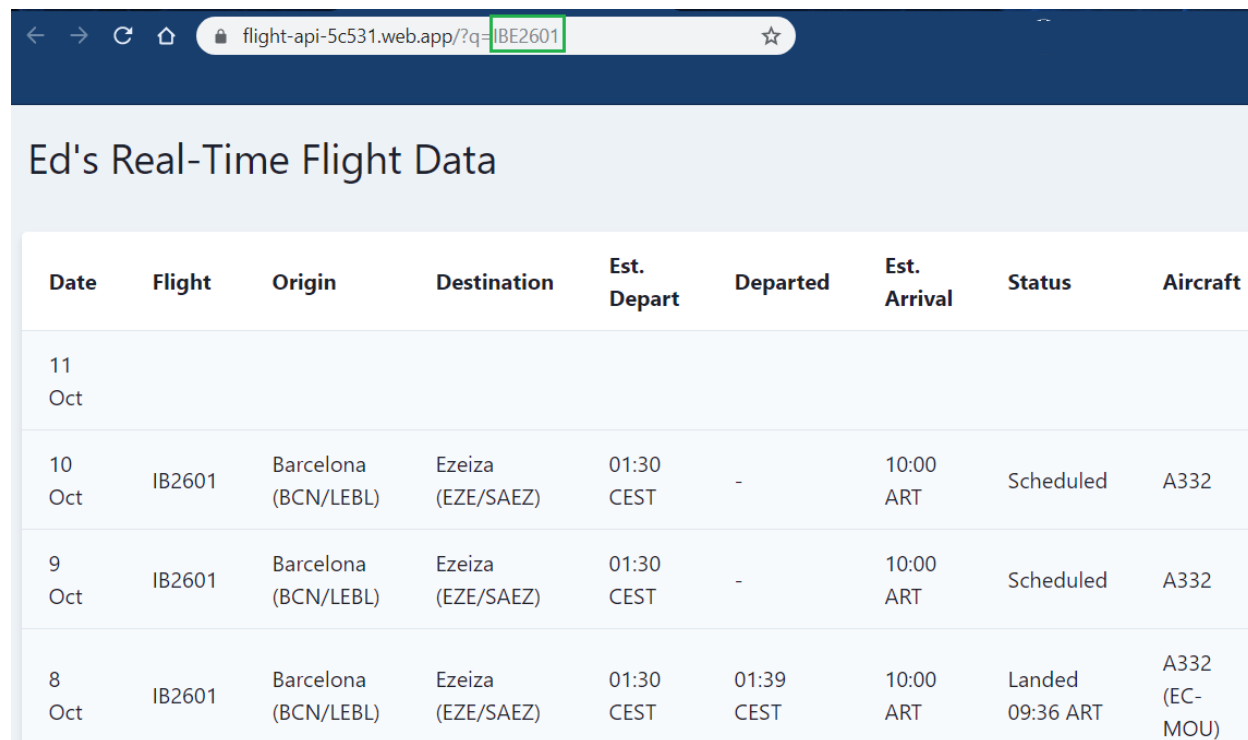
I won't cover how the first part of the API was developed, as it goes beyond the scope of this book, but I do provide the source code of the Firebase function that returns the JSON response from the API, for educational purposes.

You might be asking yourself why I created this API instead of using another API for a much simpler use case than searching for flight data.

The answer is that PWAs are about consuming APIs, and what makes a PWA interesting and attractive is what data it can provide to its users—near real-time flight information is something cool and very useful for this connected world we live in.

Ed's real-time flight data

The real-time flight information retrieval API I created fetches flight data from nonproprietary sources, and then it curates and validates the accuracy of the data. The data is then stored within Cloud Firestore. This is how the site looks.

Ed's Real-Time Flight Data

Date	Flight	Origin	Destination	Est. Depart	Departed	Est. Arrival	Status	Aircraft
11 Oct								
10 Oct	IB2601	Barcelona (BCN/LEBL)	Ezeiza (EZE/SAEZ)	01:30 CEST	-	10:00 ART	Scheduled	A332
9 Oct	IB2601	Barcelona (BCN/LEBL)	Ezeiza (EZE/SAEZ)	01:30 CEST	-	10:00 ART	Scheduled	A332
8 Oct	IB2601	Barcelona (BCN/LEBL)	Ezeiza (EZE/SAEZ)	01:30 CEST	01:39 CEST	10:00 ART	Landed 09:36 ART	A332 (EC-MOU)

Figure 2-c: Ed's Real-Time Flight Data (First Part of the API—Data Retrieval and Validation)

The flight information is displayed as a table—there are no little airplanes on a map. However, the data is up to date and accurate. It is retrieved, curated, and verified when the query is executed.

The data might take a few seconds before it is displayed, but generally the process is quite fast, and the data is as up to date and accurate as it would be from any of the paid sites.

This data is retrieved from the API by passing a flight number, highlighted in green in Figure 2-c. This is what I call the first part of the API, which is responsible for data retrieval and validation.

Once the data is retrieved and verified, it is stored as JSON objects within Cloud Firestore so that it can be queried by the PWA. This is what I call the second part of the API, which is what the PWA will invoke and consume. The flight number is also passed as a query, as can be seen in Figure 2-d.

{"data":{"flights":[{"arrival":"","atd":"","status":"","flight":"","sta":"","departure":"","aircraft":"","date":"11 Oct","std":""},
{"atd":"-","status":"Scheduled","flight":"IB2601","sta":"10:00 ART","departure":"Barcelona (BCN/LEBL)","aircraft":"A332","date":"10
Oct","std":"01:30 CEST","arrival":"Ezeiza (EZE/SAEZ)"},{"status":"Scheduled","flight":"IB2601","sta":"10:00
ART","departure":"Barcelona (BCN/LEBL)","aircraft":"A332","date":"9 Oct","std":"01:30 CEST","arrival":"Ezeiza (EZE/SAEZ)","atd":"-
"},{"status":"Landed 09:36 ART","flight":"IB2601","sta":"10:00 ART","departure":"Barcelona (BCN/LEBL)","aircraft":"A332 (EC-
MOU)","date":"8 Oct","std":"01:30 CEST","arrival":"Ezeiza (EZE/SAEZ)","atd":"01:39 CEST"},{"flight":"IB2601","sta":"10:00
ART","departure":"Barcelona (BCN/LEBL)","aircraft":"A332 (EC-NEN)","date":"7 Oct","std":"01:30 CEST","arrival":"Ezeiza
(EZE/SAEZ)","atd":"01:57 CEST","status":"Landed"},{"flight":"IB2601","sta":"10:00 ART","departure":"Barcelona
(BCN/LEBL)","aircraft":"A332 (EC-MOU)","date":"6 Oct","std":"01:30 CEST","arrival":"Ezeiza (EZE/SAEZ)","atd":"01:40
CEST","status":"Landed 09:22 ART"},{"arrival":"Ezeiza (EZE/SAEZ)","atd":"01:54
CEST","status":"Landed","flight":"IB2601","sta":"10:00 ART","departure":"Barcelona (BCN/LEBL)","aircraft":"A332 (EC-MYA)","date":"6
Oct","std":"01:30 CEST"},{"status":"Landed 09:26 ART","flight":"IB2601","sta":"10:00 ART","departure":"Barcelona
(BCN/LEBL)","aircraft":"A332 (EC-MYA)","date":"5 Oct","std":"01:30 CEST","arrival":"Ezeiza (EZE/SAEZ)","atd":"01:30 CEST"},
{"departure":"Barcelona (BCN/LEBL)","aircraft":"A332 (EC-NEN)","date":"4 Oct","std":"01:30 CEST","arrival":"Ezeiza
(EZE/SAEZ)","atd":"02:20 CEST","status":"Landed 10:06 ART","flight":"IB2601","sta":"10:00 ART"},{"arrival":"Ezeiza
(EZE/SAEZ)","atd":"01:50 CEST","status":"Landed","flight":"IB2601","sta":"10:00 ART","departure":"Barcelona
(BCN/LEBL)","aircraft":"A332 (EC-MYA)","date":"4 Oct","std":"01:30 CEST"},{"flight":"IB2601","sta":"10:00
ART","departure":"Barcelona (BCN/LEBL)","aircraft":"A332 (EC-MYA)","date":"3 Oct","std":"01:30 CEST","arrival":"Ezeiza
(EZE/SAEZ)","atd":"01:33 CEST","status":"Landed 09:51 ART"},{"flight":"IB2601","sta":"10:00 ART","departure":"Barcelona
(BCN/LEBL)","aircraft":"A332 (EC-MOY)","date":"3 Oct","std":"01:30 CEST","arrival":"Ezeiza (EZE/SAEZ)","atd":"01:39
CEST","status":"Landed"}]}}

Figure 2-d: Ed's Real-Time Flight Data (Second Part of the API—Data Consumption)

The data consumption part of the API is actually very simple, and Code Listing 2-e shows all the code required for it (this is not the case for the data retrieval and validation part, which we won't cover).

The code for the second part is basically a Node Express application that is executed by a Firebase function. The steps necessary to build it aren't covered as part of the scope of this book, but Code Listing 2-e shows the code for educational purposes.

Code Listing 2-e: Data Consumption API Logic (Node / Express Firebase Function – index.js)

```javascript
var functions = require('firebase-functions')
const admin = require('firebase-admin')
const express = require('express')
const cors = require('cors')
const app = express()
```

```javascript
var serviceAccount = require("./serviceAccountKey.json");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: "https://flight-api-5c531.firebaseio.com"
});

const db = admin.firestore()

app.use(cors({ origin: true }))

app.get('/api/:fn', (req, res) => {
    (async () => {
        try {
            if (req.params.fn !== '') {
              let fn = req.params.fn.toLowerCase()
              const document = db.collection('flights').doc(fn)
              let item = await document.get()
              let response = item.data()
              return res.status(200).send(response)
            }
            else {
              return res.status(200).send('N/A')
            }
        } catch (error) {
            return res.status(500).send(error)
        }
    })()
  })

exports.app = functions.https.onRequest(app)
```

If you've done Node development before, notice how small this code is—that's all it takes to get the JSON data stored within Cloud Firestore that was retrieved, curated, and validated by the initial part of the API.

API execution workflow

The PWAs we are building will only invoke the second part of the API, which returns a JSON response that the app can consume.

Essentially, the PWA will execute a query like the following one, which retrieves the information for a given flight number from Cloud Firestore using a Firebase function:

https://us-central1-flight-json.cloudfunctions.net/app/api/IBE2601

The last part of the query (highlighted in green) corresponds to the flight number that the PWA will retrieve details about.

The flight information (which the PWA will query using the second part of the API) stored within Cloud Firestore is not much, and it is limited to only a few flight numbers, such as: ibe2601, ar1140, ba197, bel245, glo7730, hc404, hv6148, hv6150, kqa564, sas4424, ux193, vy1374, and vy1375.

Flight numbers are treated as case-insensitive by both parts of the API, and internally they are stored in lowercase within Cloud Firestore. So, flight numbers can be written in uppercase or lowercase.

However, if you would like to get details for other flights and expand the flight data that the PWA can query, for your own testing, you'll need to follow these next steps.

First, check out the Flightradar24 website, choose one of the planes, and then copy the flight number, which in the following example is RAM505.
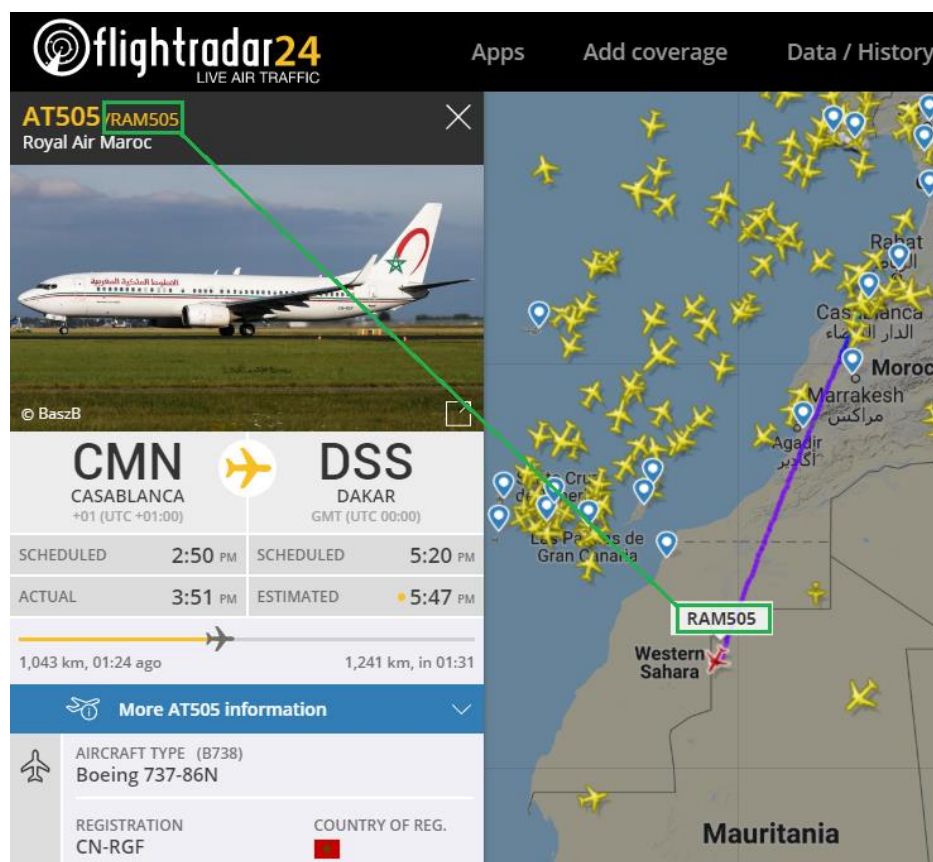

Figure 2-e: Manually Getting a Flight Number from Flight Radar 24

Then, once copied, open the first part of the API that gathers, curates, and validates the flight data, and enter the flight number (highlighted in green) as part of URL query parameter. In this case it would be:

https://flight-api-5c531.web.app/?q=RAM505

The API will gather all the flight details for that flight number, validate and cross-check its accuracy, and then store the information within Cloud Firestore.

By taking this step, you'll be able to add more flight details to the Cloud Firestore database that the API uses, which means that the PWA will have some more flight records to retrieve and display.

Note: If you call the first part of the API (the URL listed previously) and pass it an existing (stored) flight number (also shown previously), then the existing flight data will be refreshed with the most recent and up-to-date information for that particular flight number, and updated within Cloud Firestore.

The flight details for flight number RAM505 can be later retrieved (either manually or by the PWA) using the following URL: https://us-central1-flight-json.cloudfunctions.net/app/api/RAM505.

Note: The API is running on Firebase's Spark (free) plan, which means that it is limited on resources, for both usage and storage. So, please only add one or two extra flight numbers at max while you are building your own version of the PWA. It is also possible that I will (from time to time) delete the flight data stored within Cloud Firestore to keep the API operational and not incur unexpected costs.

Note: It is also possible that eventually, I might shut down the API completely, several months after the book has been published. If the API is no longer active by then, you may contact me directly, and I can provide insights or consultancy on how you may set up your own.

Summary

Now that we have explored how the API works and how it can be used to retrieve flight information, we need to expand the application's logic to be able to make calls to the second part of the API and display the results—which means that we'll also have to add some extra UI logic and functionality.

Before we do that, let's explore in more detail the requirements for building a PWA, and the essential characteristics that make an app a PWA. This is what we'll do in the next chapter.

Chapter 3 PWA Essentials

Quick intro

With the foundation of our application covered, and after reviewing how the API works, it's now time to dig deeper into PWAs and the essential aspects that make them what they are.

Characteristics of a PWA

A progressive web app, commonly referred to as a PWA, is a great way for developers to make their web application load faster and be higher performing. In a nutshell, PWAs are webpages that are intended to be applications. They use recent web standards that allow for installation on the user's computer or mobile device.

A PWA delivers a native app-like experience to users. A great example is the Twitter mobile app, which recently launched on https://mobile.twitter.com as a PWA built with React and Node. Other well-known PWAs are: Forbes, The Weather Channel, and Alibaba.

Basically, a PWA is nothing more than a web application that can be installed on your system. It works also offline when there is no internet connection, leveraging data cached during your last interactions with the app. If you are on a desktop using Chrome and have the appropriate flags turned on, you will be prompted to install the app when you visit the website.

PWAs are a trending and hot topic in web and mobile development nowadays, and are considered a next step in user-friendly app experiences that dedicated app developers should carefully explore and consider.

The great thing about developing PWAs is that, as a developer, you are still creating a web application using the web technologies you are already familiar with, such as HTML, CSS, and JavaScript. And beyond that, it also gives you the possibility to use your favorite framework, such as Ionic, Vue, or any other.

PWAs are a fusion between the look and feel of a traditional mobile app, combined with the challenges of programming a responsive modern-day website. PWAs provide a cutting-edge experience for your users to access your content by driving higher-quality engagement.

PWAs, which are responsive websites with offline capabilities, structured as apps, rely on the user's browser capabilities. They can progressively enhance their built-in features automatically to look and feel like a native app.

Essential components of a PWA

For an application to be a PWA, there are two fundamental components it needs to have:

- **Manifest file**: Used by the app to indicate features that a native app would have, such as an app icon and home screen.
- **Service worker(s)**: Used for processing background tasks and enabling offline support by caching data fetched through HTTPS requests.

In what other ways do PWAs differ from native apps? A native app is a self-contained program that resides within a mobile device, which works in a similar way as a program installed and running on a desktop computer.

A PWA, on the other hand, has no native features, other than it displays like a mobile app; it is a web app that executes via a browser. A PWA can have access to native features through its host process, the web browser.

You might be asking yourself: Why are they called progressive, and what's so special about them beyond displaying like a native app and having offline support?

Progressive by design

PWAs are progressive because they do not have the restrictions of traditional apps, which can only work on a specific platform. "Progressive" means that they should be able to work on as many platforms as possible, performing the exact same way on each. PWAs should be able to work the same way with every browser on every operating system. This is perhaps the most essential and distinctive characteristic of a PWA.

The main aspect that should stand out for a PWA is its ability to have progressive enhancements across most modern-day browsers and operating systems available on the market.

Responsive by design

Another essential and distinctive characteristic of a PWA is that it needs to be responsive. A PWA needs to be able to adjust and meet the requirements of the device being used—this is known as responsive design. This makes PWAs accessible across multiple devices, such as desktops, laptops, and phones and tablets, with different resolutions.

Connectivity independent

When you cannot visit Amazon.com and place an order, you know that your internet connectivity is down. Something great about native apps is that they can mostly still function if there is no internet connection.

A PWA must be able to allow users to interact with it, despite the connection to the internet being down—thus, the app must be able to work offline.

This is achieved by the PWA, by caching the app data ahead of time, using a service worker—this offers a programmatic way for caching the app's data and resources, such as HTML, CSS, JavaScript files, and the fonts and images used.

App-like behavior

Even though a PWA is built using web technologies, it should still give users the feeling that they are using a native app—this is ultimately the biggest difference between a PWA and a website or traditional web app.

There are many websites and web apps out there that are a collection of pages, even though they might have sophisticated authentication, routing, and database features. For a web app to be considered a PWA, it needs to include interactive features that keep the user engaged.

The PWA's main page should be able to be added to the device's home screen, allowing the user to open it in the same way as a native app.

Why are PWAs needed?

Although technology has brought improvements to people's life, one sometimes has the feeling that there's always something new to keep up with, which makes it more complex to understand why we need some of these latest technologies in the first place.

PWAs help solve some of the problems that have come as a byproduct of the internet, such as internet connection speed, slow website load times, and, to some extent, user engagement. PWAs focus on solving these problems by:

- **Providing a consistently fast experience** for users, from the moment they download the app until they start interacting with it. With a PWA, everything must be as fast as possible, which is a key element to drive user engagement.
- **Providing a reliable experience**, which means that if the internet connection fails, the app is still able to perform its job properly, if it has the data to do it.
- **Providing a seamless experience**, which means that users can also expect to have some of the features that they have come to expect from native apps, such as push notifications and access to some device functionalities.
- **Driving engagement**: By being available offline and providing notifications, PWAs can keep users engaged.

Requirements for building a PWA

There are four minimum requirements for building a PWA, two of which we already briefly covered:

- **Manifest**: A JSON file that provides meta information about the app. It has information like the app's icon, background color, name, and short name.

- **Service workers**: Event-driven workers that run in the background. A service worker acts as a proxy between the network and the application by intercepting network requests, caching information in the background, and loading data for offline use.
- **Icon**: It provides an **Add to Home Screen** app icon that a user can use to install the PWA on their device's home screen.
- **Served Over HTTPS**: PWAs must be served over a secure network connection. With services like Cloudflare and Lets Encrypt, it is quite easy to get an SSL certificate for any site. By being secure, not only does a PWA follow best practices, but also helps establish long-term trust with users and helps avoid middle-man attacks from unknown sources.

PWA advantages

PWAs need to be safe and secure. PWAs should provide a familiar app experience for today's organizational and user demands. Safety is a hot and huge topic, particularly because users and organizations alike are very sensitive to having their data compromised, lost, or stolen—which, with the advent of GDPR, also means that everyone must comply or face huge fines.

PWAs are a great way to overcome safety and security concerns, as they are offered through HTTPS, which provides major benefits for users, organizations, and developers alike.

Another advantage of PWAs is that they can be easily updated by developers, and these updates can be deployed to users without requiring any app reinstall, as the app itself resides on a web server and not the user's device. These updates can be added directly by a remote team of developers.

Users will notice new and improved features and will not have to go through the hassle of approving the installation of patches or hot fixes in a traditional way. As new features come out, they are automatically available.

Another great thing about PWAs is that they eliminate the fear of not having enough space for the app. PWAs still require some space for offline data, resources, and content, but this is relatively small compared to the space required by traditional native apps, which require a lot of free space—not only for the data they use, but also for their binaries.

Because of these reasons, PWAs are arguably the next step in web application development, interaction, and functionality, which makes the process of accessing the app convenient for users.

This technology is quickly gaining more traction and adoption, becoming a powerful movement and force in the world of software development.

Quick peek into the finished PWA

With all the theory behind what PWAs are, it's now time to have a quick look at how the finished application will look when running on Android. Notice how when running for the first time, the app displays a message to the user to have the app added to the home screen.
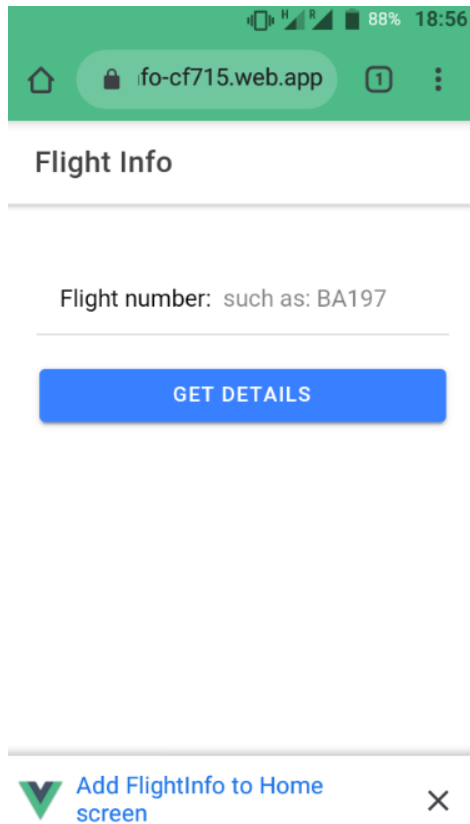
Figure 3-a: The Finished Flight Info PWA Running on an Android Device (Google Chrome)

Figure 3-b shows what the finished PWA looks like when running on Safari, using an iPhone, in offline mode. It displays information from the app's local cache, through a service worker.
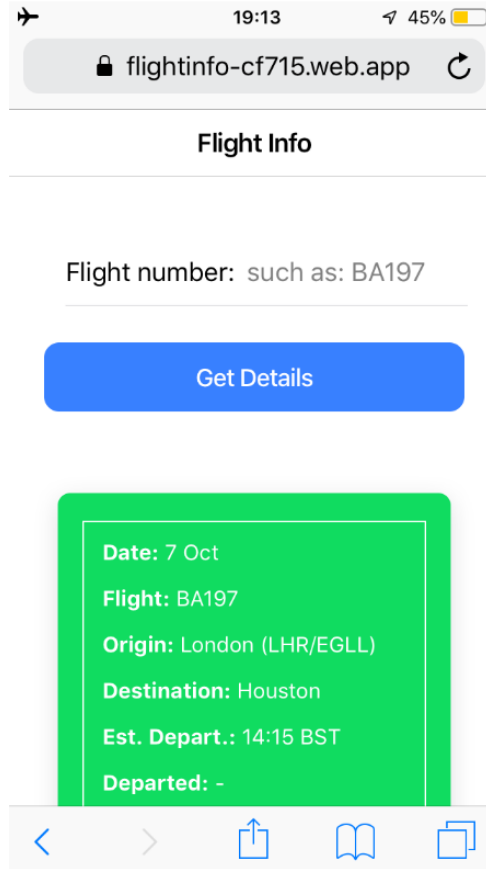
Figure 3-b: The Finished Flight Info PWA Running on an iPhone (Safari) in Offline Mode

Notice how the app has a similar look and feel on both platforms (Android and iOS), and how certain UI features adjust automatically to the host system, such as the app's title, which is aligned to the left side of the screen on Android, and centered on iOS.

Driven by fast-paced innovation

PWAs are being driven by the fast-paced innovation happening at big companies such as Google, and this is clearly highlighted throughout their web developer community and reflected in their documentation.

The most recent trend in PWA development, at the time of writing of this book, is developing and running PWAs on the desktop and on Chrome OS.

Microsoft is also betting huge on the future of PWAs and bringing them to the Windows desktop, so it's a growing trend. Even a whole section of the Windows Dev Center is dedicated to PWAs, which is welcoming to see—and a sign that PWAs are here to stay.

PWAs are checked for high quality

PWAs don't need to be deployed through Google Play or the Apple App Store. Application stores not only serve as app supermarkets, but also ensure that apps go through rigorous quality checks before they are published. This way, users know that the apps they install on their devices are tested, safe, and can be trusted.

As PWAs are not available through application stores, how can they be checked for good quality, ensuring that they not only live up to users' expectations, but also to the standards that the whole PWA movement has set forth?

Enter Lighthouse

The answer is Lighthouse, an open-source, automated tool that checks the quality of webpages that intend to become a PWA. Lighthouse has many built-in audits for verifying performance, accessibility, best-practices, SEO, and checking if a website meets the requirements needed to be a fully functional PWA.

Lighthouse is also available as a Chrome extension that any developer can use to check if their PWA is fully compliant, performing, and accessible.
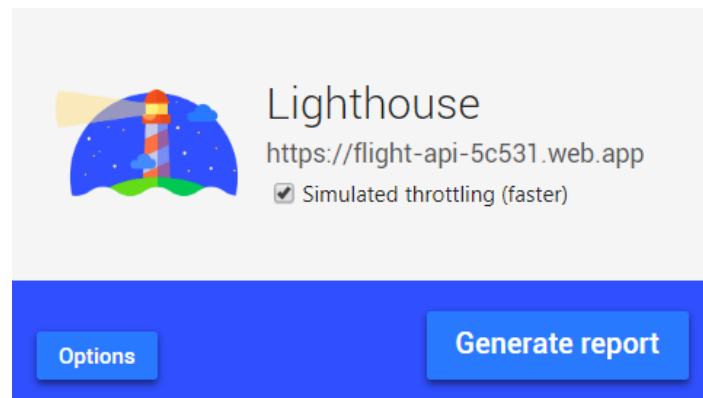

Figure 3-c: The Google Lighthouse Chrome Extension

Checking if your app is compliant enough to be a PWA is as simple as clicking the **Generate report** button. A PWA-compliant header report (specifically, the one for the finished PWA of this book) looks as follows.
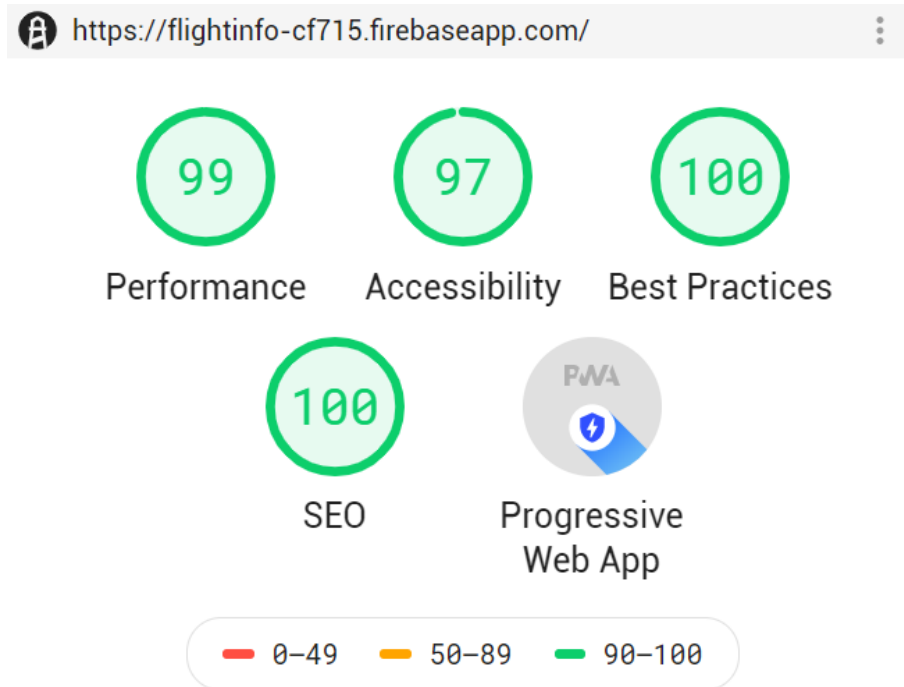
Figure 3-d: Lighthouse Report (Header) for the Finished Flight Info PWA

Notice how the header of the generated Lighthouse report displays the different categories checked and assigns an overall score for each. Each section can be further inspected to understand what details and requirements might be missing, and to see what can be further optimized.
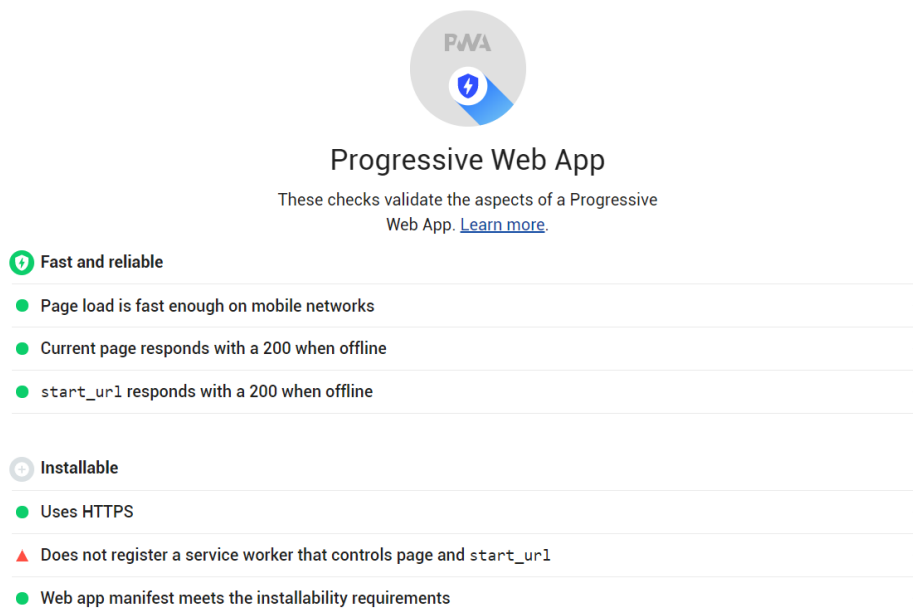

Figure 3-e: Details of a Lighthouse Report (PWA Section)

Summary

We now know what defines a PWA and what goals it should accomplish, not only from a functional point of view, but also from a technical, safety, security, and user-engagement point of view. The PWA checklist that specifies how exactly a fully compatible PWA should work is not only a guideline, but a source of inspiration.

We are now in a position to continue and modify the application we have started by adding the remaining functionality to make it a full-fledged PWA. That's what the next chapter is all about.

Chapter 4 Scaffolding the PWA

Quick intro

We've come a long way in a short time, and we are now ready to start scaffolding our PWA, as well as registering the service worker and the rest of the settings that will allow the app to be built as a PWA. That's what this chapter is all about. Let's explore the steps required to achieve that.

Vue/PWA

To start enabling our application to become PWA-compliant, a fundamental step is installing the **@vue/pwa** package. You can do this by invoking the following command from the command line or the built-in terminal within VS Code.

Code Listing 4-a: Vue/PWA Package Install Command

```
vue add @vue/pwa
```

Once this package is installed, the necessary files and configuration settings will be available to enable PWA features that the code will require, so the app can become a full-fledged PWA.

Some of the files installed after running the **@vue/pwa** package are a set of image icons for different platforms, as you can see in Figure 4-a.
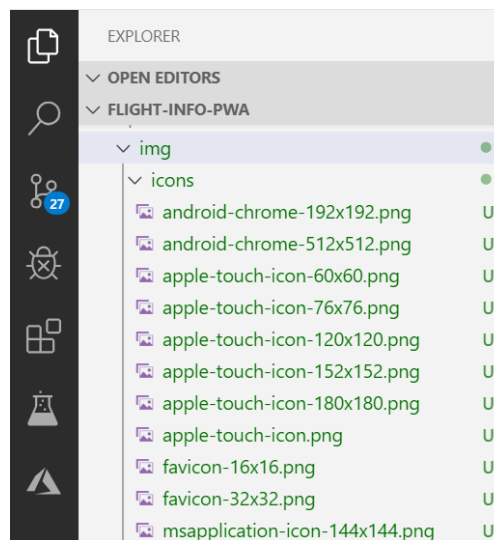

Figure 4-a: Icon Images Installed with the Vue/PWA Package

Another important change is the addition of the **manifest.json** file, located under the **public** folder of the VS Code project. Let's make some adjustments to it.

The manifest.json file

Open the **manifest.json** file, and let's make some adjustments to the default settings. Let's start off by changing the **short_name** property to **FlightInfo**, which we can see as follows.

Figure 4-b: Changing the short_name Property (manifest.json)

Let's also double-check that the **start_url** is set to **./index.html**, which we can see as follows.

Figure 4-c: The start_url Property (manifest.json)

Next, let's adjust the **background_color** and **theme_color** properties to the following values, shown in Figure 4-d.

Figure 4-d: Changing the background_color and theme_color Properties (manifest.json)

Awesome—we now have the **manifest.json** file ready. Code Listing 4-b shows what it looks like, with the changes highlighted in bold.

Code Listing 4-b: Modified / Final manifest.json File

```
{
  "name": "flight-info-pwa",
  "short_name": "FlightInfo",
  "icons": [
    {
      "src": "./img/icons/android-chrome-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "./img/icons/android-chrome-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ],
  "start_url": "./index.html",
  "scope": ".",
  "display": "standalone",
  "background_color": "#fff",
  "theme_color": "#3880ff"
}
```

Creating the service worker

One of the great things about the **@vue/pwa** package is that by using a configuration file called **vue.config.js**, we can easily define several parameters, so that when we run or build the application, it will automatically generate the app's service worker—without us having to write the code for it.

To do that, let's define the content of **vue.config.js**. If this file was not created during the installation of the **@vue/pwa** package, then in your project's root folder, please manually create it.

The **vue.config.js** file for our PWA must contain the following content.

Code Listing 4-c: The Content of the vue.config.js File

```
module.exports = {
    pwa: {
        appleMobileWebAppCapable: 'yes',
        appleMobileWebAppStatusBarStyle: 'blue',
        workboxPluginMode: 'GenerateSW',
        workboxOptions: {
            exclude: [
                /\.map$/,
                /manifest\.json$/
            ],
            navigateFallback: '/index.html',
            runtimeCaching: [
                {
                    urlPattern: new RegExp('/offline'),
                    handler: 'staleWhileRevalidate',
                },
                {
                    urlPattern: new RegExp('/'),
                    handler: 'staleWhileRevalidate',
                },
                {
                    urlPattern: new RegExp('^https://cors-
                    anywhere.herokuapp.com/https://us-central1-flight-
                    json.cloudfunctions.net/app/api/'),
                    handler: 'networkFirst',
                    options: {
                        networkTimeoutSeconds: 500,
                        cacheName: 'flight-info-cache',
                        cacheableResponse: {
                            statuses: [0, 200, 404]
                        }
```

```
                    }
                }
            ]
        }
    }
}
```

Let's explore the most crucial settings within this configuration file.

Possibly the most important setting within the file is **workboxPluginMode**, which is set to **GenerateSW**. This option literally means that the service worker (SW) will be automatically generated during the process of building the application, which is done through the Workbox webpack plugin.

Another value that can be used for the **workboxPluginMode** property is **InjectManifest**. The **GenerateSW** value will create a service worker file for you and add it to the webpack asset pipeline when the app is built.

On the other hand, using the **InjectManifest** value will create a list of URLs to precache and add, but we'll have to create the service worker code ourselves.

Given that this PWA has quite simple runtime configuration requirements, it's better to use the **GenerateSW** value for the **workboxPluginMode** property. More information about these differences can be found on the official Workbox documentation site.

Next, the **navigateFallback** setting indicates that the PWA will default to the **index.html** page if something goes wrong during runtime execution; this is the only HTML page that the application will have.

The **runtimeCaching** array is also very important, especially the third and larger item of the array that contains the definitions for the **urlPattern** that the app will be querying, which refers to the API, but routed through a Cross-Origin Resource Sharing (CORS) proxy server hosted on Heroku, to avoid running into same-origin request errors.

Basically, the API is invoked by routing it through the CORS proxy server as follows:

https://cors-anywhere.herokuapp.com/<< The API URL goes here>>

API URL: https://us-central1-flight-json.cloudfunctions.net/app/api/

If the app were to invoke the API URL directly, a same-origin request error would be returned—this is because the PWA is hosted on a different URL than the API. The PWA runs on Firebase Hosting, which has a different server address than the API, which runs as a Firebase function on another server address.

If the PWA and the API have the same base URL, then there won't be any need to use a proxy server. The proxy server is used to avoid running into same-origin request errors, as both services have different base URLs.

The other reason using a CORS proxy server is useful is because during development and testing, the PWA might be running on a local host address with a specific port number, and the API might be running on the same local host, but on another port—which makes the base URL of both different. To be able to invoke the API with different base URLs, and to avoid same-origin request errors, the proxy server would have to be used.

Furthermore, although during development and testing the PWA will likely be executed from a local host, the API might have been already deployed to Firebase functions, thus running in the cloud. Therefore, the PWA would have to use the proxy server to invoke it and avoid same-origin request errors.

In either case, using a CORS proxy server is good practice and prevents a lot of wasted time troubleshooting connection errors during development, testing, and after the go-live.

The **handler** property with the value of **networkFirst** is used to indicate that the service worker implements a network-first request strategy, which simply means that online users will get the most up-to-date content by default, and that offline users will get a cached version of the content if it has been previously downloaded and saved.

The information cached is then stored in a browser local database called **flight-info-cache** by storing responses from the API that have HTTP(S) status codes corresponding to 0, 200, and 404. More information about HTTP(S) response codes can be found here.

For offline compatibility, the stale-while-revalidate strategy is used—which means that any cached version available will be used, and an updated version will be fetched next time. This is implemented on the two other URL patterns by setting the value of the **handler** property to **staleWhileRevalidate**.

Finally, the **appleMobileWebAppCapable** and **appleMobileWebAppStatusBarStyle** properties are only specific to the WebKit web browser engine used by the Safari web browser on iOS. The former indicates that the web application runs in full-screen mode, and the latter sets the style of the bar for the app. More details about both settings can be found on the Apple Developer documentation site.

Registering the service worker

Now that we know how we can use the **vue.config.js** file to generate the service worker using the Workbox webpack plugin, let's understand how we can register it when the PWA runs. Creating it is no good if the service worker cannot be used.

To register the service worker, we need to use a file called **registerServiceWorker.js**, which should have been created under the project's **src** folder when we installed the **@vue/pwa** package. If this file doesn't exist, please create it using VS Code.
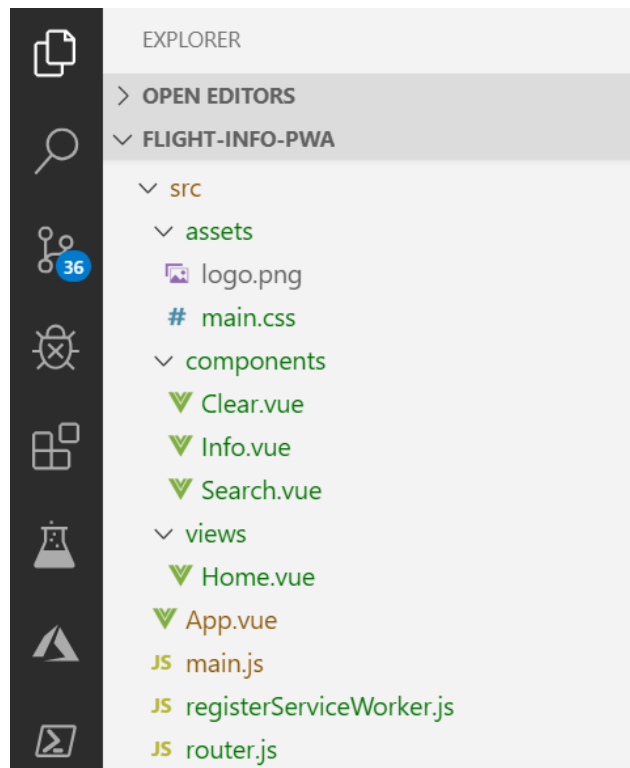
Figure 4-e: The registerServiceWorker.js File (VS Code)

Code Listing 4-d shows the content of the **registerServiceWorker.js** file.

Code Listing 4-d: The Content of the registerServiceWorker.js File

```javascript
import { register } from 'register-service-worker'

if ('serviceWorker' in navigator) {
  navigator.serviceWorker.
    register(`${process.env.BASE_URL}service-worker.js`)
}

if (process.env.NODE_ENV === 'production') {
  register(`${process.env.BASE_URL}service-worker.js`, {
    ready () {
      console.log(
        'App is being served from cache by a Service Worker.\n' +
        'For more details, visit https://goo.gl/AFskqB'
      )
    },
    registered () {
      console.log('Service Worker has been registered.')
    },
```

```
    cached () {
      console.log('Content has been cached for offline use.')
    },
    updatefound () {
      console.log('New content is downloading.')
    },
    updated () {
      console.log('New content is available; please refresh.')
    },
    offline () {
      console.log('No internet connection found. App in offline mode.')
    },
    error (error) {
      console.error('Error during Service Worker registration:', error)
    }
  })
}
```

As we can see, the code is very simple. There are two **if** statements that are in charge of executing the registration of the service worker when the app is in **production** mode, and if the browser supports service workers—which is what **'serviceWorker' in navigator** checks.

Then, there are specific events that get evoked on, during, and after the service worker registration process. All that is done is to log messages to the console, to make sure every step is executed correctly when the app is compiled and executed—as you can see in Figure 4-f.
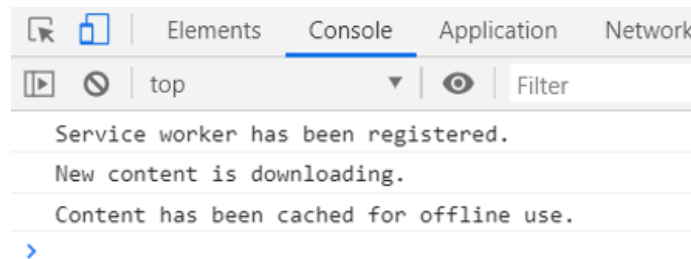

Figure 4-f: Service Worker Registration Console Output (Chrome Developer Tools)

The generated service worker

Once the app has been built for production and distribution (which is done by running the **npm run build** command), a **dist** folder will be created under the project root folder, which is what will get deployed to Firebase Hosting.

Within the **dist** folder, there is a **service-worker.js** file that was automatically generated based on the settings described in **vue.config.js**. This file should not be modified, as it gets recreated every time that the app is rebuilt.

```
/**
 * Welcome to your Workbox-powered service worker!
 *
 * You'll need to register this file in your web app and you should
 * disable HTTP caching for this file, too.
 * See https://goo.gl/nhQhGp
 *
 * The rest of the code is auto-generated. Please don't update this file
 * directly; instead, make changes to your Workbox build configuration
 * and re-run your build process.
 * See https://goo.gl/2aRDsh
 */

importScripts(
"https://storage.googleapis.com/workbox-cdn/releases/3.6.3/workbox-sw.js");

importScripts(
  "/precache-manifest.c4355da19e97306fb3d51dadf5baf4ff.js"
);

workbox.core.setCacheNameDetails({prefix: "flight-info-pwa"});

/**
 * The workboxSW.precacheAndRoute() method efficiently caches and responds
to
 * requests for URLs in the manifest.
 * See https://goo.gl/S9QRab
 */
self.__precacheManifest = [].concat(self.__precacheManifest || []);
workbox.precaching.suppressWarnings();
workbox.precaching.precacheAndRoute(self.__precacheManifest, {});

workbox.routing.registerNavigationRoute("/index.html");

workbox.routing.registerRoute(
  /\/offline/, workbox.strategies.staleWhileRevalidate(), 'GET');
workbox.routing.registerRoute(
  /\//, workbox.strategies.staleWhileRevalidate(), 'GET');
workbox.routing.registerRoute(
/^https:\/\/cors-anywhere.herokuapp.com\/https:\/\/us-central1-flight-
json.cloudfunctions.net\/app\/api\//,
workbox.strategies.networkFirst(
  { "cacheName":"flight-info-cache","networkTimeoutSeconds":500,
```

```
  plugins:
    [new workbox.cacheableResponse.Plugin(
    {"statuses":[0,200,404]})] }), 'GET'
);
```

As you can see, it's just a code equivalent of the configuration settings described in
vue.config.js.

There are only two lines of code that stand out. One is the line that imports the Workbox library
from Google's content delivery network:

`importScripts("https://.../workbox-sw.js");`

The other is the line that imports the precached application resources that was also
automatically generated during the build process:

`importScripts(`

`  "/precache-manifest.c4355da19e97306fb3d51dadf5baf4ff.js"`

`);`

If you open the **precache-manifest** file, you will notice that it is a very long file and contains
mostly references to CSS, JavaScript, and image files that the app will use. These are
downloaded and cached by the service worker when the app runs.

The JavaScript files referenced within the **precache-manifest** file are the app's compiled code,
which is split into chunks and optimized for fast browser loading and performance. This is what
webpack does when the app is built using the `npm run build` command.

Here is what the **precache-manifest** file looks like.

Figure 4-g: The Generated precache-manifest File

Polyfills and browser compatibility

The **@vua/pwa** package is compatible with most modern browsers such as Chrome, Firefox, and Safari. However, there are some known [issues](#) with Edge and Internet Explorer, which can be seen as follows.
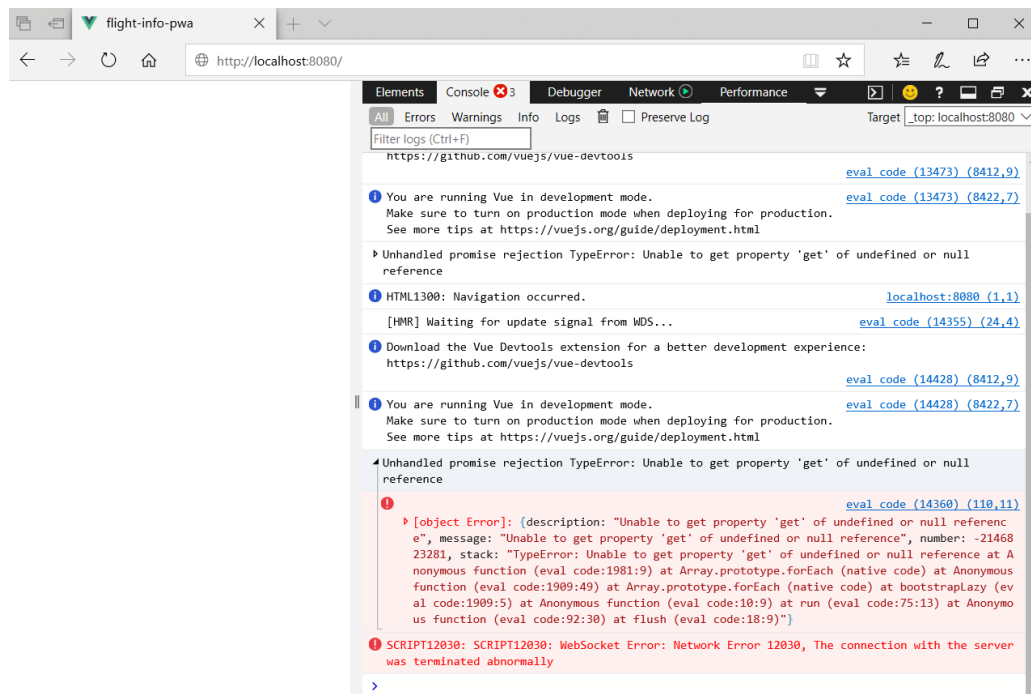

Figure 4-g: Known Issues *with Edge (Using the* @vue/pwa *Package)*

Therefore, it's always good idea to modify the settings of the **babel.config.js** file (found under the root folder of the VS Code project) by adding some extra polyfills, which provide extra support for ES6 on older browsers.

You can add these polyfills by installing the **@babel/polyfill** package with the following command.

Code Listing 4-f: Command to Install the @babel/polyfill Package

```
npm install --save @babel/polyfill
```

Once this package has been installed, it is automatically added to the **package.json** file found under the project's root folder. Code Listing 4-g shows the app's **package.json** file.

Code Listing 4-g: The package.json File

```
{
  "name": "flight-info-pwa",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
  "dependencies": {
    "@babel/polyfill": "^7.6.0",
    "@ionic/vue": "0.0.4",
    "core-js": "^2.6.5",
    "i": "^0.3.6",
    "register-service-worker": "^1.6.2",
    "vue": "^2.6.10",
    "vue-router": "^3.0.3"
  },
  "devDependencies": {
    "@vue/cli-plugin-babel": "^3.11.0",
    "@vue/cli-plugin-eslint": "^3.11.0",
    "@vue/cli-plugin-pwa": "^3.11.0",
    "@vue/cli-service": "^3.11.0",
    "babel-eslint": "^10.0.1",
    "eslint": "^5.16.0",
    "eslint-plugin-vue": "^5.0.0",
    "ionicons": "^4.5.9-1",
    "vue-template-compiler": "^2.6.10"
  },
  "eslintConfig": {
```

```
    "root": true,
    "env": {
      "node": true
    },
    "extends": [
      "plugin:vue/essential",
      "eslint:recommended"
    ],
    "rules": {},
    "parserOptions": {
      "parser": "babel-eslint"
    }
  },
  "postcss": {
    "plugins": {
      "autoprefixer": {}
    }
  },
  "browserslist": [
    "> 1%",
    "last 2 versions"
  ]
}
```

If for some reason you have some of these packages missing, you can easily add them to your project by running the following command from the project's root folder. This will install all the packages described within the **package.json** file shown in Code Listing 4-g.

Code Listing 4-h: Command to Install Missing Packages

```
npm install
```

Next, let's update the **babel.config.js** file with the polyfill settings—it should look as follows.

Code Listing 4-i: Updated babel.config.js File with Polyfills

```
module.exports = {
  presets: [
    ['@vue/app', {
      polyfills: [
        'es6.promise',
        'es6.symbol'
      ]
    }]
```

```
    ]
}
```

These changes to the **babel.config.js** file might still not solve the known <u>issues</u> with Edge when using the **@vue/pwa** package—these were still open with the core development team at the time of writing of this book—however, it is always recommended to have enabled support for older browsers.

Summary

With these settings changes in place, we are ready to update the logic of the app. We now have the PWA infrastructure all set up, which needs to be completed with code that will allow the application to have the desired functionality and the look and feel of a PWA. This is what we'll cover in the next chapter.

Chapter 5 Building the PWA

Quick intro

We've now reached (in my opinion) the most fun part of the book—which is to add the remaining logic that will make our application a true PWA. Behind us are all the tedious and required configuration settings that helped us get to this stage. Now, let's dive into the code.

Final main.js file

As its name implies, **main.js** is the app's main code entry point, which gets injected into **index.html** at build time by webpack. Let's have a look at this file's final code.

Code Listing 5-a: The Final main.js File

```
import Vue from 'vue'
import App from './App.vue'
import router from './router'

import Ionic from '@ionic/vue'
import '@ionic/core/css/ionic.bundle.css'
import './registerServiceWorker'

Vue.use(Ionic)
Vue.config.productionTip = true

new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

It is essentially the same **main.js** that we previously wrote, except for one additional, but very important, line that we were not able to previously add, which is:

```
import './registerServiceWorker'
```

This line references the **registerServiceWorker.js** file, which is responsible for registering the service worker when the application runs.

It's a small, but significant change, as it's a requirement for any PWA to register the service worker when it runs.

Final App.vue file

The **App.vue** file is the app's main HTML markup file, and the only thing it does is contain a reference to the **ion-vue-router** component—which we can see as follows.

Code Listing 5-b: The App.vue File

```
<template>
  <div id="app">
    <ion-app>
      <ion-vue-router/>
    </ion-app>
  </div>
</template>
```

Given the **App.vue** file references the **ion-vue-router** component, the next thing we need to do is look at the **router.js** file.

Final router.js file

As its name implies, the **router.js** file handles the application's routing, which in the case of this PWA, is limited to **Home.vue**. Let's have a look at the final code for **router.js**.

Code Listing 5-c: The router.js File

```
import Vue from 'vue'
import { IonicVueRouter } from '@ionic/vue'
import Home from './views/Home.vue'

Vue.use(IonicVueRouter)

export default new IonicVueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    }
  ]
})
```

No changes here—all this file does is reference **Home.vue**, which is where the most interesting code of the application resides.

Final Home.vue file

The **Home.vue** functional component is not only responsible for rendering most of the app's UI, but also calling the API and parsing and displaying the results. Essentially, it is the heart and soul of our PWA. Let's have a look at the code.

Code Listing 5-d: The Home.vue File

```
<template>
  <div class="ion-page">
    <ion-header translucent>
      <ion-toolbar>
        <ion-title>
          Flight Info
        </ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content fullscreen class="ion-text-center ion-padding">
        <Search v-on:flight="flightDetails"/>
        <ion-spinner
          v-if="!info && fn != null"
          name="dots" color="tertiary">
        </ion-spinner>
        <Info v-bind:fn="fn" v-bind:info="info" />
        <Clear v-bind:info="info" v-on:clear="clear" />
    </ion-content>
  </div>
</template>

<script>
import Search from '../components/Search'
import Info from '../components/Info'
import Clear from '../components/Clear'

export default {
  name: 'home',
  components: {
    Search,
    Info,
    Clear
  },
```

```
  data() {
    return {
      info: null,
      fn: null
    }
  },
  methods: {
    getJson(flight) {
      const proxy = 'https://cors-anywhere.herokuapp.com/'
      const site =
      'https://us-central1-flight-json.cloudfunctions.net/app/api/'

      fetch(`${proxy}${site}${flight}`)
      .then(r => r.json())
      .then(d => {
        this.info = d
      })
      .catch(err => console.log('HTTP-Error: ' + err))
    },
    async flightDetails(flight) {
      this.fn = flight
      await this.getJson(flight)

      if (this.info != null && this.info.length == 0) {
        this.info = null
        return this.$ionic.alertController.create({
          header: 'Flight',
          message: 'Flight ' + this.fn + ' not found.',
          buttons: ['OK']
        }).then(r => r.present())
      }
    },
    clear() {
      this.info = null
      this.fn = null
    }
  }
}
</script>
```

To understand what is going on, let's first focus on the markup and then the code. So, let's revise each part separately and divided into smaller chunks—this way it will be easier to understand.

```
<template>
  <div class="ion-page">
    <ion-header translucent>
      <ion-toolbar>
        <ion-title>
          Flight Info
        </ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content fullscreen class="ion-text-center ion-padding">
        <Search v-on:flight="flightDetails"/>
        <ion-spinner
          v-if="!info && fn != null"
          name="dots" color="tertiary">
        </ion-spinner>
        <Info v-bind:fn="fn" v-bind:info="info" />
        <Clear v-bind:info="info" v-on:clear="clear" />
    </ion-content>
  </div>
</template>
```

The markup is made up of two main sections—a header (**ion-header**) and content. The header is made of an **ion-title** component embedded within an **ion-toolbar** component. It simply displays the application title.

The content part is more interesting—this is wrapped within an **ion-content** component. Within it, there is a **Search**, an **ion-spinner** (which by default is not visible), an **Info**, and a **Clear** component.

To get a better sense of how this markup relates to the finished UI, let's have a look at the following diagram.
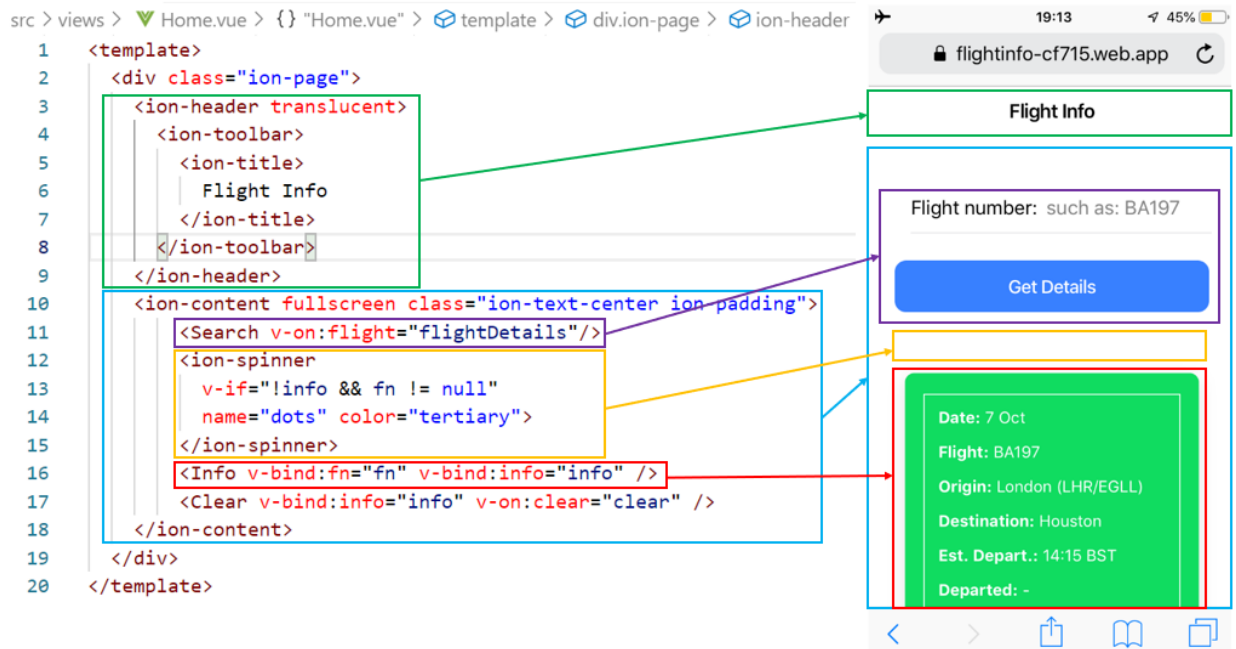
Figure 5-a: Relationship Between the Home.vue HTML and the App's UI

When the **Search** component emits the **flight** event, the **flightDetails** method is executed. This is what will make the call to the API and retrieve the flight details that are passed to the **Info** component using the **info** object (which contains the flight details retrieved from the API). The flight number (**fn**) is also passed to the **Info** component.

The **ion-spinner** component is only displayed when the flight number (**fn**) has been entered by the user, and the **info** object doesn't contain any data—which means that the app still needs to retrieve the flight data from the API. This what the **ion-spinner** object looks like.

Figure 5-b: The ion-spinner Component (During the Flight Details Search)

The **Clear** component, which is essentially a button that is shown below the search results, is only displayed when the **info** object contains data and is rendered as follows. The **Clear** component emits a clear **event** that triggers the execution of the **clear** method, which clears the search results retrieved from the API from the screen.
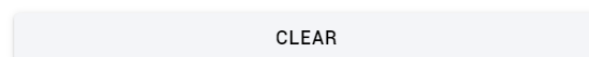

Figure 5-c: The Clear Component

Now that we understand how the UI works, let's explore the code of **Home.vue** to understand how the data is retrieved from the API and passed to the **Search** component.

```
<script>
import Search from '../components/Search'
import Info from '../components/Info'
import Clear from '../components/Clear'

export default {
  name: 'home',
  components: {
    Search,
    Info,
    Clear
  },
  data() {
    return {
      info: null,
      fn: null
    }
  },
  methods: {
    getJson(flight) {
      const proxy = 'https://cors-anywhere.herokuapp.com/'
      const site =
      'https://us-central1-flight-json.cloudfunctions.net/app/api/'

      fetch(`${proxy}${site}${flight}`)
      .then(r => r.json())
      .then(d => {
        this.info = d
      })
      .catch(err => console.log('HTTP-Error: ' + err))
    },
    async flightDetails(flight) {
      this.fn = flight
      await this.getJson(flight)

      if (this.info != null && this.info.length == 0) {
        this.info = null
        return this.$ionic.alertController.create({
          header: 'Flight',
          message: 'Flight ' + this.fn + ' not found.',
          buttons: ['OK']
        }).then(r => r.present())
      }
```

```
    },
    clear() {
      this.info = null
      this.fn = null
    }
  }
}
</script>
```

Let's revise this code from top to bottom to understand what each part does. The first three lines import the **Search**, **Info**, and **Clear** components so they can be referenced and used both in the HTML markup and within the code.

```
import Search from '../components/Search'

import Info from '../components/Info'

import Clear from '../components/Clear'
```

Then, these components are referenced within the **components** object—this way, they become available within the code.

Next, we have the **data** function, which returns an object that contains the **info** and **fn** properties. The **info** property will be used to store the flight data retrieved from the API, and the **fn** property is used to keep the flight number for which the information is retrieved.

Within the **methods** object, there are three methods defined that make the core logic of the application. The main method is **flightDetails**, which is asynchronous. The **flightDetails** method calls the **getJson** method, which is the one that executes the call to the API.

The **Clear** method is invoked when the **Clear** button is clicked—all it does is initialize the values of the **info** and **fn** properties, so a new search can take place.

The implementation of the **flightDetails** method is very straightforward. As you can see, it makes a call to the **getJson** method by passing the **flight** and awaits its response—the result that the API returns.

If no data is returned by the API (when the **this.info != null && this.info.length == 0** condition evaluates to true), then a dialog message is displayed, which is done by calling the following code.

```
this.$ionic.alertController.create({ header: 'Flight',
message: 'Flight ' + this.fn + ' not found.',
buttons: ['OK']}).then(r => r.present())
```

The **getJson** method is where the magic happens. By using the CORS proxy server, a call to the API is made through the browser's Fetch API, which in the code is done by calling **fetch**.

When the Fetch API returns a response, the code contained within the first promise (first **then** statement) is executed (**r => r.json()**), which returns the API's response as a JSON object.

When that occurs, the second promise is executed (the second **then** statement). That JSON response (represented by the variable **d**, which stands for data) is assigned to the **info** property.

If there's an error during the execution of any of the code contained within **fetch**, then an exception is raised and caught by the following code: **err => console.log('HTTP-Error: ' + err)**, which simply outputs the error to the console.

That's it—this is the main logic of our application. As you have seen, it wasn't difficult at all. Now let's have a look at the **Info** and **Clear** components.

Final Search.vue file

The **Search** component is another fundamental part of the application, and it is quite straightforward, as well. Let's have a look.

Code Listing 5-g: The Search.vue File

```
<template>
    <ion-grid>
        <form @submit="onSubmit">
            <ion-col>
                <ion-item>
                    <ion-label>Flight number: </ion-label>
                    <ion-input :value="flight"
                        @input="flight = $event.target.value"
                        placeholder="such as: BA197"
                        name="flight"></ion-input>
                </ion-item>
            </ion-col>
            <ion-col>
                <ion-button id="btn" type="submit"
                    color="primary" expand="block">
                    Get Details
                </ion-button>
            </ion-col>
        </form>
    </ion-grid>
</template>

<script>
export default {
```

```
        name: 'Search',
        data() {
            return {
                flight: ''
            }
        },
        methods: {
            onSubmit(e) {
                e.preventDefault()
                if (this.flight != '') {
                    this.$emit('flight', this.flight)
                    this.flight = ''
                }
                else {
                    this.displayAlert()
                    this.flight = ''
                }
            },
            displayAlert() {
                return this.$ionic.alertController.create(
                    {
                        header: 'Flight',
                        message: 'Enter a flight number.',
                        buttons: ['OK']
                    }
                ).then(r => r.present())
            }
        }
    }
</script>
```

Just like we did with **Home.vue**, let's split the logic into two parts to understand it better: the HTML markup that defines the UI and the code. Let's check out the markup first.

Code Listing 5-h: The Search.vue File (Markup Only)

```
<template>
    <ion-grid>
        <form @submit="onSubmit">
            <ion-col>
                <ion-item>
                    <ion-label>Flight number: </ion-label>
                    <ion-input :value="flight"
                        @input="flight = $event.target.value"
```

```
                    placeholder="such as: BA197"
                    name="flight"></ion-input>
            </ion-item>
        </ion-col>
        <ion-col>
            <ion-button id="btn" type="submit"
                color="primary" expand="block">
                Get Details
            </ion-button>
        </ion-col>
        </form>
    </ion-grid>
</template>
```

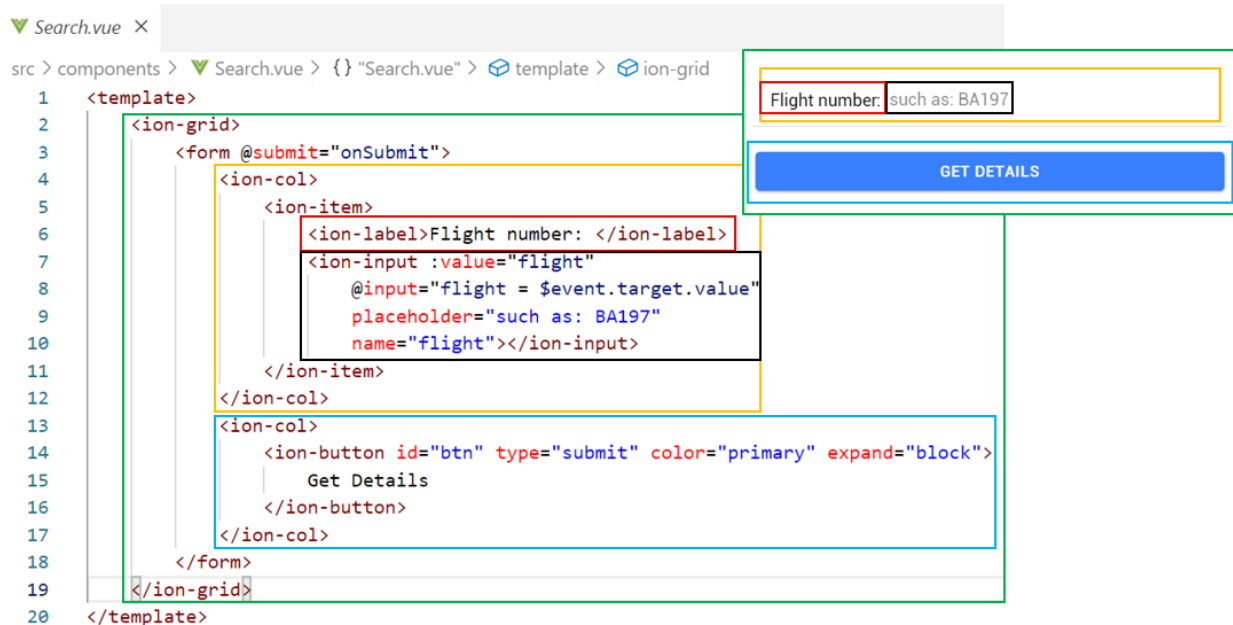The following diagram illustrates how this markup relates to the UI elements that make the **Search** component.

Figure 5-d: Relationship Between the Search.vue HTML and the Search Component UI

The core of the **Search** UI is composed by a **form**, which is embedded within an **ion-grid** component. When it is submitted, this **form** (which occurs when the **Get details** button is clicked) triggers the execution of the **submit** event, which invokes the **onSubmit** method.

The **ion-input** component, which is contained within an **ion-item** and **ion-col** component, captures the flight number entered by the user, and this value is assigned to the **flight** variable.

Finally, the **ion-button** component, contained within an **ion-col**, can trigger the **submit** event of the parent **form**, because its **type** is set to **submit**. Let's have a look at the code.

Code Listing 5-i: The Search.vue File (Code Only)

```
<script>
export default {
    name: 'Search',
    data() {
        return {
            flight: ''
        }
    },
    methods: {
        onSubmit(e) {
            e.preventDefault()
            if (this.flight != '') {
                this.$emit('flight', this.flight)
                this.flight = ''
            }
            else {
                this.displayAlert()
                this.flight = ''
            }
        },
        displayAlert() {
            return this.$ionic.alertController.create(
                {
                    header: 'Flight',
                    message: 'Enter a flight number.',
                    buttons: ['OK']
                }
            ).then(r => r.present())
        }
    }
}
</script>
```

The **data** function returns an object that has a single property called **flight**, which represents the flight number.

Within the **methods** object, we have the **onSubmit** method, which emits the **flight** event that is intercepted within **Home.vue** to make the call to the API when the flight number has been entered by the user. If not, then a message dialog is displayed via the **displayAlert** method.

Final Info.vue file

The **Info** component is responsible for displaying the flight details retrieved through the API. Following is the full code of this component.

Code Listing 5-j: The Info.vue File

```
<template>
    <div>
    <ion-card color="success" padding="true" v-if="info != null">
        <table>
            <thead>
                <tr>
                    <th>Date</th>
                    <th>Flight</th>
                    <th>Origin</th>
                    <th>Destination</th>
                    <th>Est. Depart.</th>
                    <th>Departed</th>
                    <th>Est. Arrival</th>
                    <th>Status</th>
                    <th>Aircraft</th>
                </tr>
            </thead>
            <tbody>
                <tr v-for="(itm, idx) in info.data.flights"
                    v-bind:key="itm.date + '-' + itm.flight + '-' + idx">
                    <td>{{itm.date}}</td>
                    <td>{{itm.flight}}</td>
                    <td>{{itm.departure}}</td>
                    <td>{{itm.arrival}}</td>
                    <td>{{itm.std}}</td>
                    <td>{{itm.atd}}</td>
                    <td>{{itm.sta}}</td>
                    <td>{{itm.status}}</td>
                    <td>{{itm.aircraft}}</td>
                    <td></td>
                </tr>
            </tbody>
        </table>
    </ion-card>
    </div>
</template>

<script>
```

```
export default {
    name: 'Info',
    props: ['info', 'fn']
}
</script>

<style scoped>
    /*
    Generic styling, for desktops/laptops
    */
    table {
        width: 100%;
        border-collapse: collapse;
        text-align: center;
    }
    th {
        color: white;
        font-weight: bold;
        text-align: center;
    }
    td, th {
        padding: 6px;
        text-align: center;
    }

    /*
    Max width before this PARTICULAR table gets nasty.
    This query will take effect for any screen smaller than 760px
    and also iPads specifically.
    */
    @media
    only screen and (max-width: 760px),
    (min-device-width: 768px) and (max-device-width: 1024px)  {

        /* Force table to not be like tables anymore */
        table, thead, tbody, th, td, tr {
            display: block;
            text-align: left;
        }

        /* Hide table headers (but not display: none; for accessibility) */
        thead tr {
            position: absolute;
            top: -9999px;
```

```css
        left: -9999px;
        text-align: left;
    }

    tr {
        border: 1px solid #fff;
        margin-bottom: 1%;
        margin-top: 1%;
        padding-top: 2%;
        padding-bottom: 2%;
        text-align: left;
    }

    td {
        /* Behave like a "row" */
        border: none;
        border-bottom: 0px solid #fff;
        position: relative;
        /* padding-left: 50%; */
        text-align: left;
    }

    td:before {
        /* Now like a table header */
        position: relative;
        /* Top/left values mimic padding */
        top: 0px;
        left: 6px;
        /* width: 45%; */
        padding-right: 10px;
        white-space: nowrap;
        font-weight: bold;
        text-align: left;
    }

    /*
    Label the data
    */
    td:nth-of-type(1):before { content: "Date:"; }
    td:nth-of-type(2):before { content: "Flight:"; }
    td:nth-of-type(3):before { content: "Origin:"; }
    td:nth-of-type(4):before { content: "Destination:"; }
    td:nth-of-type(5):before { content: "Est. Depart.:"; }
    td:nth-of-type(6):before { content: "Departed:"; }
```

```
        td:nth-of-type(7):before { content: "Est. Arrival:"; }
        td:nth-of-type(8):before { content: "Status:"; }
        td:nth-of-type(9):before { content: "Aircraft:"; }
    }

    /* Smartphones (portrait and landscape) ----------- */
    @media only screen
    and (min-device-width : 320px)
    and (max-device-width : 480px) {
        body {
            padding: 0;
            margin: 0;
            width: 320px; }
        }

    /* iPads (portrait and landscape) ----------- */
    @media only screen and (min-device-width: 768px) and
    (max-device-width: 1024px) {
        body {
            width: 495px;
        }
    }
</style>
```

As you have seen, most of the file content is HTML markup and CSS with very little code, except for the definition of the **info** and **fn** properties, which contain the flight details and flight number, respectively.

Let's now explore how the data is displayed, which can be better understood with the following diagram.
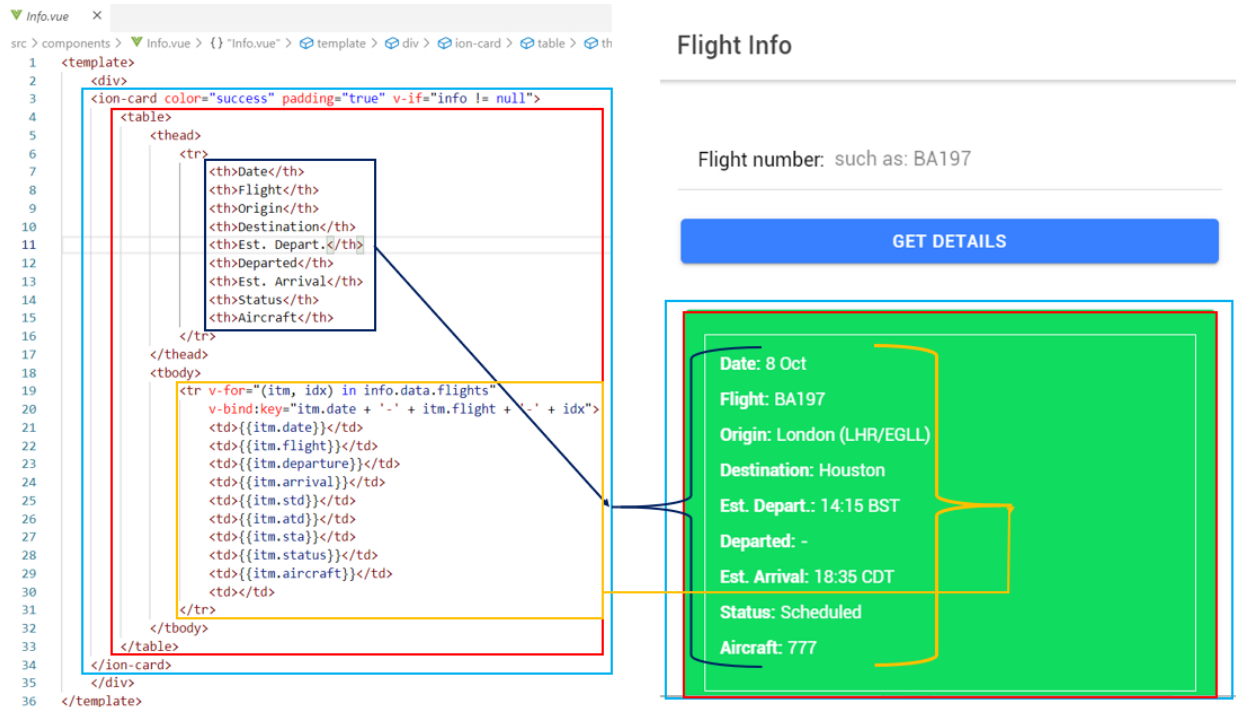
Figure 5-e: Relationship Between the Info.vue HTML and the Info Component UI

The **table** that displays the flight data retrieved from the API through the **info** property is contained within an **ion-card** component. Let's explore the HTML markup in more detail.

Code Listing 5-k: The Info.vue File (Markup Only)

```html
<template>
    <div>
    <ion-card color="success" padding="true" v-if="info != null">
        <table>
            <thead>
                <tr>
                    <th>Date</th>
                    <th>Flight</th>
                    <th>Origin</th>
                    <th>Destination</th>
                    <th>Est. Depart.</th>
                    <th>Departed</th>
                    <th>Est. Arrival</th>
                    <th>Status</th>
                    <th>Aircraft</th>
                </tr>
            </thead>
            <tbody>
                <tr v-for="(itm, idx) in info.data.flights"
```

```
                    v-bind:key="itm.date + '-' + itm.flight + '-' + idx">
                <td>{{itm.date}}</td>
                <td>{{itm.flight}}</td>
                <td>{{itm.departure}}</td>
                <td>{{itm.arrival}}</td>
                <td>{{itm.std}}</td>
                <td>{{itm.atd}}</td>
                <td>{{itm.sta}}</td>
                <td>{{itm.status}}</td>
                <td>{{itm.aircraft}}</td>
                <td></td>
            </tr>
        </tbody>
    </table>
  </ion-card>
  </div>
</template>
```

We can see that the first part of the table (**thead**) defines the table's header, which essentially is the name of the fields to display.

Then within **tbody**, we loop through each **itm** (which corresponds to a data row) for all the flight details contained within **info.data.flights**, which represents the object structure of the API's JSON response.

Notice how for every data row (**itm**), we are also getting an index (**idx**), which we combine with **itm.date** and **itm.flight** to create a unique **key** for every row within the table. Each field value is then displayed, such as **{{itm.departure}}**.

The CSS styling is designed to work and be totally responsive in both desktop and mobile modes. Code Listing 5-l shows the CSS classes and properties that apply for desktop mode. I've added some comments to the code that make it easier to understand. Let's have a look.

Code Listing 5-l: The Info.vue File (Desktop Mode CSS Only)

```
/*
Generic styling, for desktops/laptops
*/
table {
  width: 100%;
  border-collapse: collapse;
  text-align: center;
}
th {
  color: white;
```

```css
    font-weight: bold;
    text-align: center;
}
td, th {
    padding: 6px;
    text-align: center;
}
```

This styling is quite simple, but the idea is to align the table content centered, and the table can adjust to various desktop resolutions. We can see an example of a desktop resolution in Figure 5-f.

Flight Info

Flight number: such as: BA197

GET DETAILS

Date	Flight	Origin	Destination	Est. Depart.	Departed	Est. Arrival	Status	Aircraft
8 Oct	BA197	London (LHR/EGLL)	Houston	14:15 BST	-	18:35 CDT	Scheduled	777
7 Oct	BA197	London (LHR/EGLL)	Houston	14:15 BST	-	18:35 CDT	Scheduled	777
6 Oct	BA197	London (LHR/EGLL)	Houston	14:05 BST	-	18:25 CDT	Scheduled	-
5 Oct	BA197	London (LHR/EGLL)	Houston	14:15 BST	14:51 BST	18:35 CDT	Landed 18:58 CDT	B772 (G-YMML)
4 Oct	BA197	London (LHR/EGLL)	Houston	14:15 BST	15:53 BST	18:35 CDT	Landed 19:09 CDT	B772 (G-YMML)
3 Oct	BA197	London (LHR/EGLL)	Houston	14:15 BST	15:28 BST	18:35 CDT	Landed 18:53 CDT	B772 (G-YMML)
2 Oct	BA197	London (LHR/EGLL)	Houston	14:15 BST	14:49 BST	18:35 CDT	Landed 18:41 CDT	B772 (G-YMMP)
1 Oct	BA197	London (LHR/EGLL)	Houston	14:15 BST	14:52 BST	18:35 CDT	Landed 18:49 CDT	B772 (G-YMMP)
30 Sep	BA197	London (LHR/EGLL)	Houston	14:15 BST	14:31 BST	18:35 CDT	Landed 18:00 CDT	B772 (G-YMMP)

CLEAR

Figure 5-f: App with Desktop Display Mode

Notice how the table's content can adjust responsively to the screen's resolution, without any issues.

The nondesktop CSS is slightly more complex, so that the table can be displayed for any mobile device resolution.

Code Listing 5-m: The Info.vue File (Mobile Mode CSS Only)

```css
/*
Max width before this PARTICULAR table gets nasty.
This query will take effect for any screen smaller than 760px
```

```css
and also iPads specifically.
*/
@media
    only screen and (max-width: 760px),
    (min-device-width: 768px) and (max-device-width: 1024px)  {

    /* Force table to not be like tables anymore */
    table, thead, tbody, th, td, tr {
      display: block;
      text-align: left;
    }

    /* Hide table headers (but not display: none; for accessibility) */
    thead tr {
      position: absolute;
      top: -9999px;
      left: -9999px;
      text-align: left;
    }

    tr {
      border: 1px solid #fff;
      margin-bottom: 1%;
      margin-top: 1%;
      padding-top: 2%;
      padding-bottom: 2%;
      text-align: left;
    }

    td {
      /* Behave like a "row" */
      border: none;
      border-bottom: 0px solid #fff;
      position: relative;
      /* padding-left: 50%; */
      text-align: left;
    }

    td:before {
      /* Now like a table header */
      position: relative;
      /* Top/left values mimic padding */
      top: 0px;
      left: 6px;
```

```css
        /* width: 45%; */
        padding-right: 10px;
        white-space: nowrap;
        font-weight: bold;
        text-align: left;
    }

    /*
    Label the data
    */
    td:nth-of-type(1):before { content: "Date:"; }
    td:nth-of-type(2):before { content: "Flight:"; }
    td:nth-of-type(3):before { content: "Origin:"; }
    td:nth-of-type(4):before { content: "Destination:"; }
    td:nth-of-type(5):before { content: "Est. Depart.:"; }
    td:nth-of-type(6):before { content: "Departed:"; }
    td:nth-of-type(7):before { content: "Est. Arrival:"; }
    td:nth-of-type(8):before { content: "Status:"; }
    td:nth-of-type(9):before { content: "Aircraft:"; }
}

/* Smartphones (portrait and landscape) ----------- */
@media only screen
    and (min-device-width : 320px)
    and (max-device-width : 480px) {
        body {
            padding: 0;
            margin: 0;
            width: 320px; }
}

/* iPads (portrait and landscape) ----------- */
@media only screen and (min-device-width: 768px) and
    (max-device-width: 1024px) {
        body {
            width: 495px;
        }
}
```

Figure 5-g shows an example of how the app displays the data using a mobile responsive resolution. Notice how the table data changes from a tabular to a columnar layout, where each record becomes a card.

Figure 5-g: App with Mobile Mode Display

Let's see how each card corresponds to a table row when we switch from one device resolution to another.

Figure 5-h: Match Between the Data in Both Views (Mobile and Desktop)

This CSS styling gives the app an edge—it can display the flight data in any resolution, on any device. The styling is totally responsive and flexible.

Final Clear.vue file

We're now on the last component of our application: **Clear.vue**, which simply contains a button that removes the flight data displayed. Let's have a quick look at it.

Code Listing 5-n: The Clear.vue File

```
<template>
    <ion-grid>
        <ion-col>
            <ion-button
                color="light"
                expand="block"
                v-if="info != null"
                @click="$emit('clear')"
            >Clear</ion-button>
        </ion-col>
    </ion-grid>
</template>

<script>
export default {
    name: 'Clear',
    props: ['info']
}
</script>
```

The code is very simple. The **ion-button** component is embedded within an **ion-col** and **ion-grid** component, so that the button aligns perfectly with the **Search** and **info** components.

The button is only displayed when the **info** property that contains the flight data is not empty (**null**). This button emits a **clear** event when it is clicked—this event is intercepted within **Home.vue**, and the data is cleared there.

Summary

That's it—our PWA is ready from a code perspective, and we can run it locally by running the **npm run serve** command. However, we still have one final step remaining before we can deploy it to Firebase: get the Firebase tools installed. This is what we'll do in the next chapter.

Chapter 6 Deploying the PWA

Quick intro

We're on the last part of our journey—now, our PWA needs a place to live. The platform I've chosen is [Firebase](#), but you may choose and experiment with any other, such as [Azure](#), [AWS](#), or [Heroku](#). I highly encourage you to do that.

Let's give Firebase a whirl—we'll install the tools required to perform the deployment and deploy our PWA to the cloud with Firebase Hosting.

Setting Up Firebase Hosting

Before you can install Firebase, you need to sign up for it. This is easy to do, and all you need is your Google or Gmail account. If you are already signed in with either, the process is a breeze.

Once signed up, you can get set up with Firebase Hosting in three easy steps, as shown in the following figures.
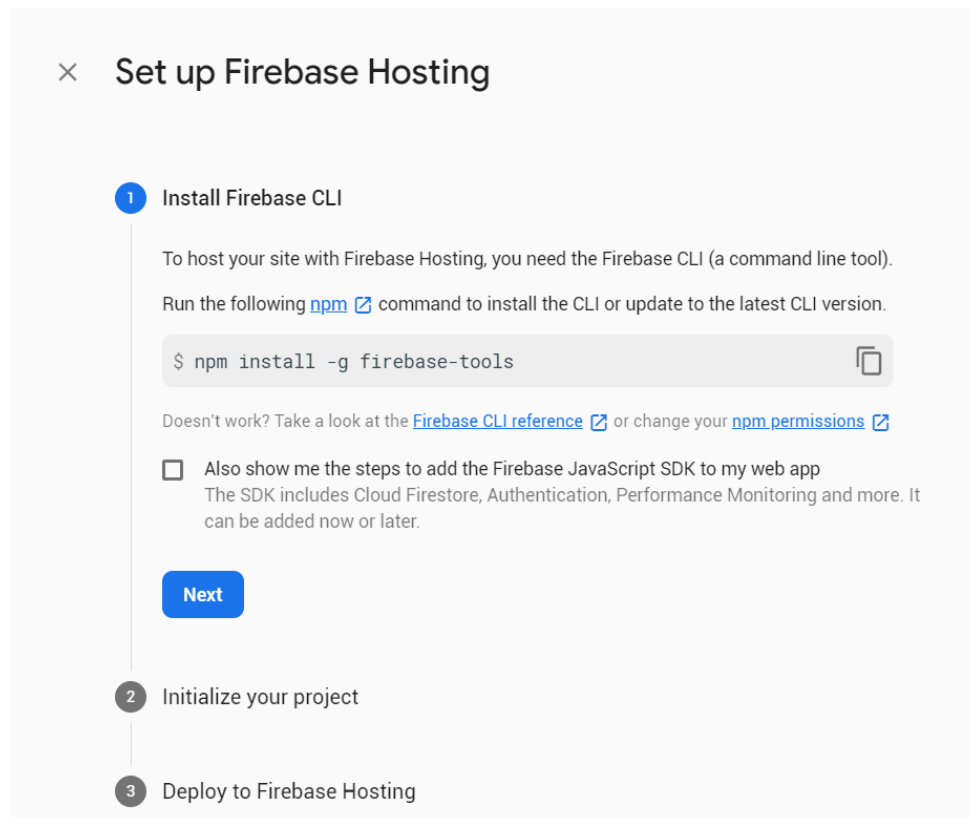

Figure 6-a: Setting Up Firebase (Step 1)

The setup steps are self-explanatory. First, we need to install Firebase Tools globally on our machine. We can do this by running the following command.

Code Listing 6-a: Command to Install Firebase Tools

```
npm install -g firebase-tools
```

Next, we need to sign in and initialize Firebase, which we can do as follows.
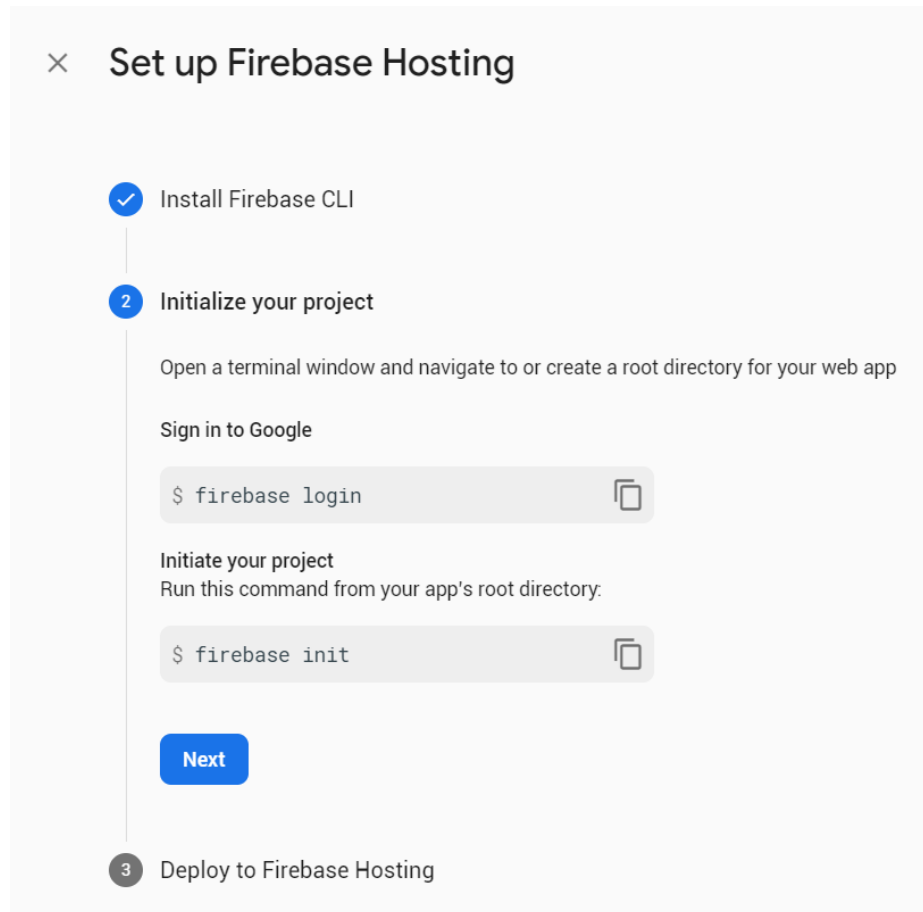

Figure 6-b: Setting Up Firebase (Step 2)

When you execute the **firebase login** command, you'll be prompted to sign in, which we can see as follows.

```
PROBLEMS   TERMINAL   ...            2: node              ▼   +  ⊡  🗑  ∧  ✕

 (current: {"os":"win32","arch":"x64"})

 + firebase-tools@7.4.0
 added 410 packages from 250 contributors in 22.916s
 PS C:\Projects\Ionic 4 Succinctly\demo\flight-info-pwa> firebase login
 i  Firebase optionally collects CLI usage and error reporting information to
 help improve our products. Data is collected in accordance with Google's priv
 acy policy (https://policies.google.com/privacy) and is not used to identify
 you.

 ? Allow Firebase to collect CLI usage and error reporting information? No

 Visit this URL on this device to log in:
 https://accounts.google.com/o/oauth2/auth?client_id=563584335869-fgrhgmd47bqn
 ekij5i8b5pr03ho849e6.apps.googleusercontent.com&scope=email%20openid%20https%
 3A%2F%2Fwww.googleapis.com%2Fauth%2Fcloudplatformprojects.readonly%20https%3A
 %2F%2Fwww.googleapis.com%2Fauth%2Ffirebase%20https%3A%2F%2Fwww.googleapis.com
 %2Fauth%2Fcloud-platform&response_type=code&state=935899105&redirect_uri=http
 %3A%2F%2Flocalhost%3A9005

 Waiting for authentication...

 +  Success! Logged in as ██████████████
```

Figure 6-c: Logging to Firebase—Command Line (Step 2)

Once signed in, we can run the **firebase init** command, which will guide us to initialize our project with Firebase.

```
PS C:\Projects\Ionic 4 Succinctly\demo\flight-info-pwa> firebase init


     ######## #### ########  ######## ########    ###      ######  ########
     ##        ## ##      ## ##       ##       ##   ##  ##      ## ##
     ######    ## ######## ######   ######## ######### ######  ######
     ##        ## ##      ## ##       ##     ## ##        ##      ## ##
     ##      #### ##       ## ######## ######## ##        ## ######  ########


You're about to initialize a Firebase project in this directory:

  C:\Projects\Ionic 4 Succinctly\demo\flight-info-pwa

Before we get started, keep in mind:

  * You are currently outside your home directory

? Are you ready to proceed? (Y/n) y
```

Figure 6-d: Initializing the Project with Firebase—Command Line (Step 3)

We need to follow the steps indicated and answer each of the questions that the initialization process asks.

Make sure that you choose the **Hosting** option from the following list when prompted. Use the arrow key to go to the option, and then use the Spacebar key to select this option.

```
? Are you ready to proceed? Yes
? Which Firebase CLI features do you want to set up for this folder? Press Sp
ace to select features, then Enter to confirm your choices.
 ( ) Database: Deploy Firebase Realtime Database Rules
 ( ) Firestore: Deploy rules and create indexes for Firestore
 ( ) Functions: Configure and deploy Cloud Functions
>(*) Hosting: Configure and deploy Firebase Hosting sites
 ( ) Storage: Deploy Cloud Storage security rules
```

Figure 6-e: Selecting the Firebase Hosting Option—Command Line (Step 3)

After selecting the Hosting option, you will be asked to either create or use an existing project. I chose to use an existing project, but in your case (unless you have already created the project in the Firebase web console), choose **Create a new project**.

```
? Please select an option:
> Use an existing project
  Create a new project
  Add Firebase to an existing Google Cloud Platform project
  Don't set up a default project
```

Figure 6-f: Creating a New Project—Command Line (Step 3)

Finally, when asked which directory to use as **public**, press Enter to finish the initialization process.

```
=== Hosting Setup

Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be uploaded with firebase deploy. If you
have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? (public) []
```

Figure 6-g: Setting the Public Folder—Command Line (Step 3)

After doing this, the project will be initialized, as we can see in the following.

```
? What do you want to use as your public directory?
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
+  Wrote /index.html

i  Writing configuration info to firebase.json...
i  Writing project information to .firebaserc...

+  Firebase initialization complete!
PS C:\Projects\Ionic 4 Succinctly\demo\flight-info-pwa> []
```

Figure 6-h: Firebase Project Initialization Finished—Command Line (Step 3)

Firebase setting files

Once Firebase has been installed and initialized, the **index.html** file within the project's root folder will be updated—here's how mine looks after these steps.

Code Listing 6-b: The index.html (root folder) After Installing Firebase

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Welcome to Firebase Hosting</title>

    <!-- update the version number as needed -->
    <script defer src="/__/firebase/7.1.0/firebase-app.js"></script>
    <!-- include only the Firebase features as you need -->
    <script defer src="/__/firebase/7.1.0/firebase-auth.js"></script>
    <script defer src="/__/firebase/7.1.0/firebase-database.js"></script>
    <script defer src="/__/firebase/7.1.0/firebase-messaging.js"></script>
    <script defer src="/__/firebase/7.1.0/firebase-storage.js"></script>
    <!-- initialize the SDK after all desired features are loaded -->
    <script defer src="/__/firebase/init.js"></script>

    <style media="screen">
      body { background: #ECEFF1; color: rgba(0,0,0,0.87);
      font-family: Roboto, Helvetica, Arial, sans-serif;
      margin: 0; padding: 0; }
      #message { background: white; max-
      width: 360px; margin: 100px auto 16px;
      padding: 32px 24px; border-radius: 3px; }
      #message h2 { color: #ffa100; font-weight: bold; font-
      size: 16px; margin: 0 0 8px; }
```

```css
      #message h1 { font-size: 22px; font-
      weight: 300; color: rgba(0,0,0,0.6); margin: 0 0 16px;}
      #message p { line-height: 140%; margin: 16px 0 24px; font-
      size: 14px; }
      #message a { display: block; text-
      align: center; background: #039be5; text-transform: uppercase; text-
      decoration: none; color: white; padding: 16px; border-radius: 4px; }
      #message, #message a {
      box-shadow: 0 1px 3px rgba(0,0,0,0.12), 0 1px 2px rgba(0,0,0,0.24); }
      #load { color: rgba(0,0,0,0.4);
      text-align: center; font-size: 13px; }
      @media (max-width: 600px) {
        body, #message { margin-top: 0; background: white;
        box-shadow: none; }
        body { border-top: 16px solid #ffa100; }
      }
    </style>
  </head>
  <body>
    <div id="message">
      <h2>Welcome</h2>
      <h1>Firebase Hosting Setup Complete</h1>
      <p>You've successfully setup Firebase Hosting.</p>
      <a target="_blank"
       href="https://firebase.google.com/docs/hosting/">
       Open Hosting Documentation</a>
    </div>
    <p id="load">Firebase SDK Loading&hellip;</p>

    <script>
      document.addEventListener('DOMContentLoaded', function() {
// // 🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎
// // The Firebase SDK is initialized and available here!
//
// firebase.auth().onAuthStateChanged(user => { });
// firebase.database().ref('/path/to/ref').on('value', snapshot => { });
// firebase.messaging().requestPermission().then(() => { });
// firebase.storage().ref('/path/to/ref').getDownloadURL().then(() => { });
//
// // 🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎🎎

        try {
          let app = firebase.app();
          let features =
```

```
              ['auth', 'database', 'messaging', 'storage'].
                filter(feature => typeof app[feature] === 'function');
            document.getElementById('load').innerHTML =
                `Firebase SDK loaded with ${features.join(', ')}`;
        } catch (e) {
            console.error(e);
            document.getElementById('load').innerHTML =
                'Error loading the Firebase SDK, check the console.';
        }
      });
    </script>
  </body>
</html>
```

You will notice that some additional files to the project have been added, such as **.firebaserc** (a list of project aliases) and **firebase.json**.

Let's make a small modification to the **firebase.json** file so that when we deploy the app, we do it with the files contained within the **dist** folder. These are the files that the webpack build process will create.

Figure 6-i: Modifying the Public Parameter—firebase.json

Set the **public** parameter within the **firebase.json** file to **./dist**—this means that when we build and deploy the application, the files that are deployed to the server will be the ones contained within the **dist** folder.

Building and deploying

The final part of the process is to build the application and deploy it to Firebase Hosting. We can do this by running the following commands, in this order.

Code Listing 6-c: Building the App
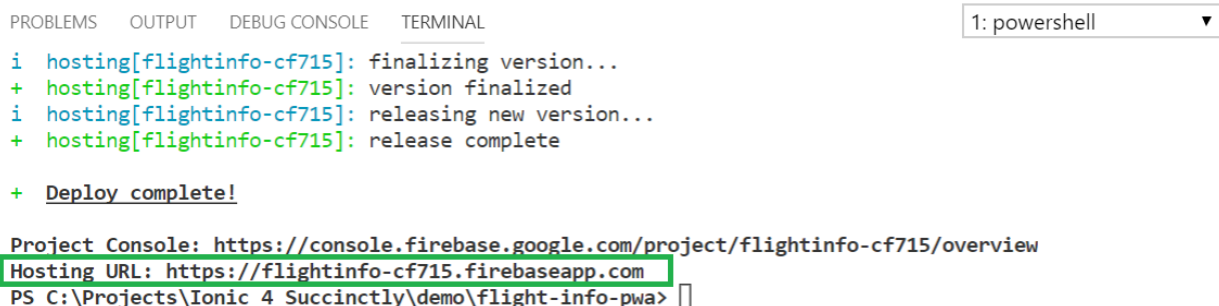
```
npm run build
```

This will create the production-ready code for the application within the **dist** folder. We can then deploy the app by executing the following.

Code Listing 6-d: Deploying the App

```
firebase deploy
```

Once the command has executed, you should see, within the VS Code built-in terminal or command prompt, the URL where the app was deployed to. In my case, this is: https://flightinfo-cf715.firebaseapp.com/.

Figure 6-j: Command Line Output Post App Deployment

Firebase also provides an alternative URL for your app, which in my case is: https://flightinfo-cf715.web.app/.

The subdomain is the same for both URLs (in my case, the name of the app's subdomain is: **flightinfo-cf715**), and the domain is either **firebaseapp.com** or **web.app**; both will resolve to the same location where the PWA is hosted.

Awesome—we now have the application hosted on Firebase. Now, the moment of truth has arrived.

Testing with Lighthouse

If you installed the Lighthouse Chrome extension, open your Chrome browser, navigate to the app's public URL, and click **Generate report**, as shown in Figure 6-k.
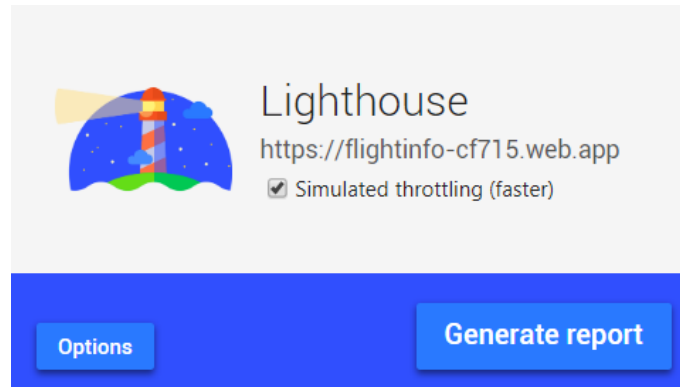
Figure 6-k: Lighthouse—Generate report Option

Lighthouse will do an exhaustive auditing of your app and determine how well-performing and accessible it is, as well as whether it follows best practices, is SEO-optimized, and is eligible to be a PWA. My initial results were as follows.
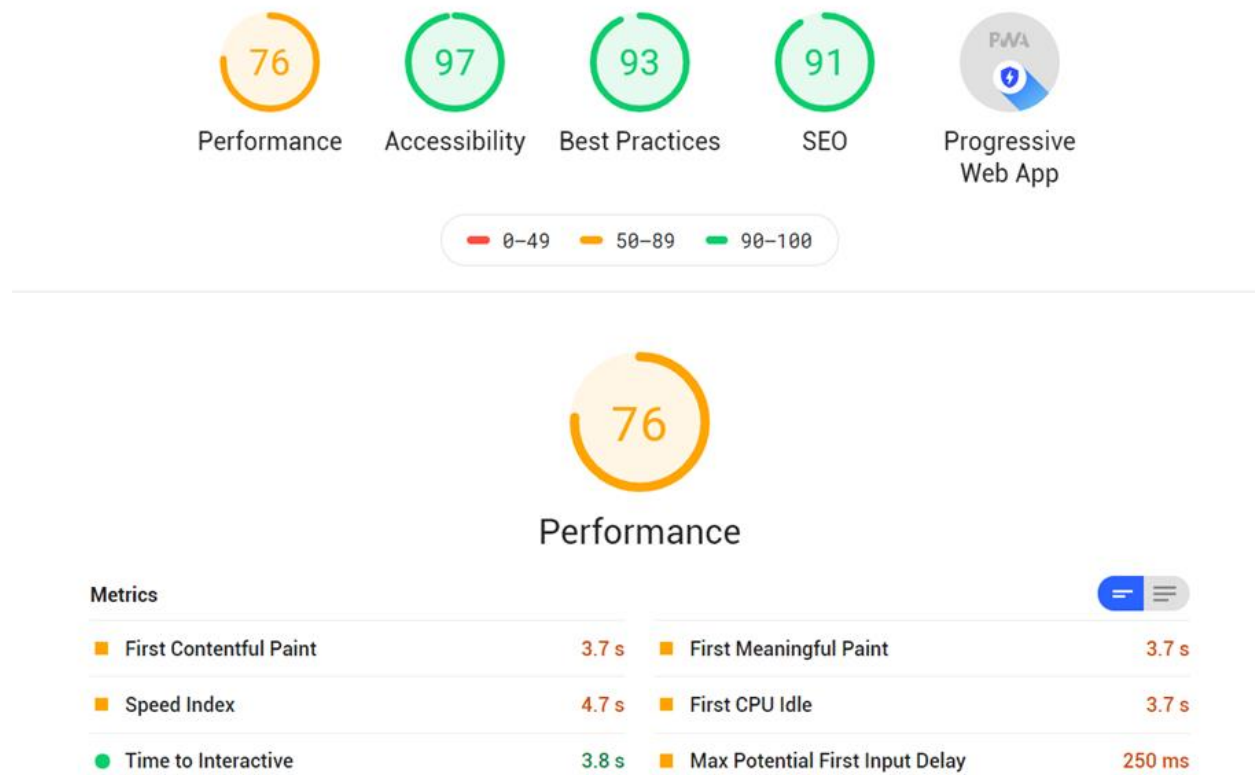

Metrics			
First Contentful Paint	3.7 s	First Meaningful Paint	3.7 s
Speed Index	4.7 s	First CPU Idle	3.7 s
Time to Interactive	3.8 s	Max Potential First Input Delay	250 ms

Figure 6-l: Initial Lighthouse Results

Performance improvement

It seems that the app's performance can be improved. Let's inspect this metric in detail to understand how we can improve it.

Figure 6-m: Initial Lighthouse Results (Performance Details)

We can see that the issues for the performance metric are all related to static content that is not served through an efficient server cache policy. We can easily resolve this by making a small change to the **firebase.json** file, as follows.

Figure 6-n: Updates to the firebase.json File—Improved Server Caching Policy

As a result of adding the settings highlighted above, any static resources will be cached the first time the app runs—and these will only be refreshed when one or more static files change, or after they have been cached for longer than the value of the **max-age** parameter.

Code Listing 6-e shows the full content of the **firebase.json** settings file after these modifications.

Code Listing 6-e: Updated firebase.json File

```
{
  "hosting": {
    "public": "./dist",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ],
    "rewrites": [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ],
    "headers": [
      {
        "source" : "**/*.@(jpg|jpeg|gif|png|css|js)",
```

```
        "headers" : [ {
            "key" : "Cache-Control",
            "value" : "max-age=31557600"
        } ]
    }
  ]
 }
}
```

If we run the Lighthouse report again, we should see a significant improvement. Let's have a look.
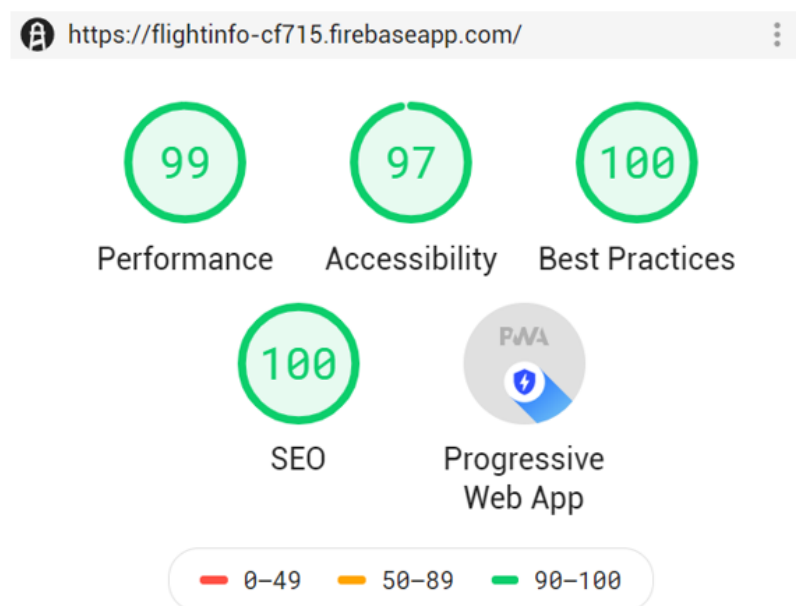

Figure 6-o: Improved Lighthouse Report Results

How cool is that? By making a simple change to the **firebase.json** settings file, we've drastically improved the app's performance. We now have a PWA!

Redeploying the app

If we execute the `npm run build` command, followed by `firebase deploy`, and browse to the app's public URL, we can start to use our PWA.
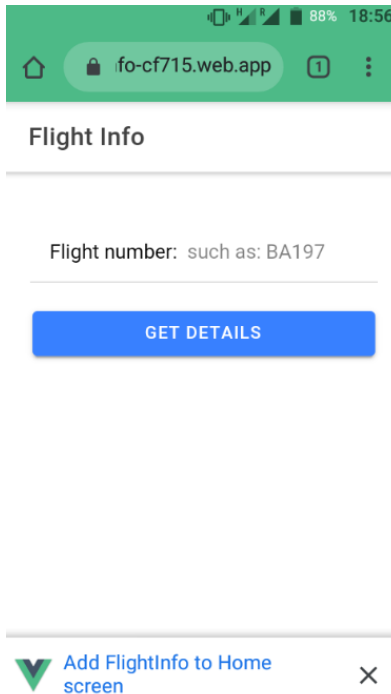
Figure 6-p: The PWA on Android

Go ahead and enter some of those flight numbers that exist within the Cloud Firestore database that the API uses, such as: ibe2601, ar1140, ba197, bel245, glo7730, hc404, hv6148, hv6150, kqa564, sas4424, ux193, vy1374, and vy1375.

If you enter some of those flight numbers while you are connected to the internet, the app will download and cache those flight details. Then, set your device to airplane mode and try again by entering a flight number you had previously entered. The app should display the flight details, but this time from the cache.

Full project source code

You can download the full source code for the PWA from this location. To optimize file and download size, this source does not contain the **node_modules** folder; you will need to run the `npm install` command from the project's root folder to install all the dependencies required to be able to run, build, and deploy the project.

Final thoughts

We now have a cool and working PWA from just a few lines of code—if you really think about it, the code required to build this application was not much.

This was possible in part because we combined different frameworks and libraries, such as Ionic with Vue, and used tools like Workbox and the **@vue/pwa** package, which gave us development superpowers throughout this process.

After undergoing this journey, I became even more fascinated with PWAs. The great thing about this technology is not only its potential—although they are quite new, the toolset for creating PWAs is quite mature and stable, with no signs of slowing down.

PWAs are certainly on a growth path, and it's worthwhile investing the time to dig deeper and expand your knowledge even further.

I hope you have enjoyed and been inspired by this book, as much as I loved writing it. Go build an amazing PWA, and if you do so, please let me know about it—I'd love to see those seeds grow.

Until next time—thank you reading this book, and all the best. Fly high Papa—I love you.