

VUE 3

SUCCINCTLY[®]

BY ED FREITAS

Vue 3 Succinctly

Ed Freitas

Foreword by Daniel Jebaraj



Copyright © 2025 by Syncfusion®, Inc.

2501 Aerial Center Parkway

Suite 111

Morrisville, NC 27560

USA

All rights reserved.

ISBN: 978-1-64200-241-6

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, ESSENTIAL, ESSENTIAL STUDIO, BOLDDesk, BOLDSIGN, BOLD BI, and BOLD REPORTS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.

Proofreader: Graham High, content team lead, Syncfusion, Inc.

Table of Contents

The <i>Succinctly</i>® Series of Books	7
About the Author	8
Acknowledgments	9
Introduction	10
GitHub code repo	11
Chapter 1 Setup	12
Overview	12
Setup.....	12
Root component.....	17
Recap.....	18
Chapter 2 App Basics	19
Prerequisites	19
Additional dependencies	19
Project cleanup.....	22
Adding fonts and styles	23
Creating the header component	25
Adding routes to the header component.....	28
Implementing the router.....	30
Adjusting main.js	31
Initial DocsPage	32
Initial AboutPage	33
Recap.....	34
Chapter 3 Enhancing the App	35
Finalizing the header	35

DocsPage architecture	38
Styling DocsPage	39
Installing Vue Datepicker	40
Creating DocInput	41
Using VueDatePicker	44
Add button.....	47
DocInput JavaScript logic	48
Validating docDescription	51
Wiring up the DocInput component.....	54
Creating the Document component	57
Wiring up the Document component.....	58
Continuing the Document component.....	60
Document actions.....	66
Running the app	69
Recap.....	71
Chapter 4 Vue 2 vs. Vue 3.....	72
Composition API.....	72
Props declaration	73
Emits declaration.....	74
Reactive data	75
Lifecycle hooks.....	75
Directives	76
Custom event handling.....	77
Scoped slots.....	78
Provide and inject.....	79
Teleport.....	79

Recap.....	80
Chapter 5 Firebase Setup.....	81
Installing Firebase	81
Firebase console (optional)	88
Creating a datastore	88
Recap.....	92
Chapter 6 Adding Firebase.....	93
Updating firebase.js.....	93
Updating Document.vue	94
Updating DocsPage.vue.....	101
Querying the database	105
Adding a document	106
Updating a document	108
Deleting a document	109
Update bindings	109
Testing the app.....	110
Closing thoughts.....	112

The *Succinctly*[®] Series of Books

Daniel Jebaraj
CEO of Syncfusion[®], Inc.

When we published our first *Succinctly* series book in 2012, *jQuery Succinctly*, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 2.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, 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.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

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 succinctlyseries@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 social media and help us spread the word about the *Succinctly* series!



About the Author

Ed Freitas is a business process automation consultant, technologist, and solutions architect focused on customer success. He likes technology and enjoys learning, playing soccer, running, traveling, and being around his family.

Ed is available at <https://edfreitas.me>.

Acknowledgments

Thank you to the fantastic [Syncfusion](#) team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript manager, Graham High from Synfusion, and technical editor, James McCaffrey from Microsoft Research, thoroughly reviewed the book's organization, code quality, and overall accuracy. Thank you all.

I dedicate this book to John Diego—may your journey be blessed.

Introduction

[Vue.js](#) or Vue (pronounced *view*) is a progressive (incrementally adoptable) front-end JavaScript framework that you can use to create engaging user interfaces and primarily [single-page applications](#).

Vue was created by [Evan You](#) after using [Angular](#) for various projects while working for Google. His idea was to use the parts of Angular that he liked and build something straightforward and lightweight, while keeping many of the fantastic features and the power of Angular.

Vue uses an HTML-based template syntax that allows reactive data binding to the DOM and can extend essential HTML elements as reusable components. It also supports front-end hashed routing through the open-source [Vue Router](#) package.

Given its simplicity and powerful features, developers have flocked to the framework, making Vue one of the top currently [trending](#) projects on GitHub, and one of the world's leading JavaScript frameworks (at the time of writing this book) with no signs of slowing down.

Besides its excellent productivity features, Vue's success is closely tied to JavaScript's ascension as the go-to language for developing modern web applications.

Vue can also be used when starting an application from scratch—when you want to add more business logic to your front end rather than the back end. It has a rich package ecosystem that provides great core features and others that manage state and handle routing.

Vue allows applications to quickly scale by supporting reusable components out of the box—each has its own HTML, CSS, and JavaScript, meaning that an application can be split into smaller functional parts, which can be reused.

All these features make Vue an excellent choice for developing front-end applications and a great framework to learn, making you a more productive and valuable developer in the market.

Vue 3 introduces several significant changes and improvements over Vue 2, such as the Composition API (the most critical difference between both versions), custom directives, smaller bundle size, improved TypeScript support, Vue Router, and performance improvements. We'll explore some of these differences later in the book.

Throughout this book, we'll build a web app that we can use to keep track of important personal documents that have expiration dates, such as passports, driver's licenses, or credit cards—the same app described in [Flutter Succinctly](#) and [Vue.js Succinctly](#) (which covers Vue 2).

The following figure shows what the Flutter app concept looks like:



Figure 0-a: The Mobile Version of the Demo App

We'll use Vue 3 to re-create this concept, but instead of giving it the same mobile app look and feel, we'll create a web app with similar functionality; however, the UI will look different.

The approach behind this book is to create the Vue 3 app step-by-step while simultaneously exploring the essential Vue 3 features required to build a minimal functional application by using the Composition API and focusing on the implementation aspect of it.

Having some prior experience working with Vue 2 is recommended. If you don't, the *Succinctly* series has you covered with either [Vue.js Succinctly](#) or [Nuxt.js Succinctly](#), which I recommend you check out first.

So, without further ado, let's explore and dive right in.

GitHub code repo

You can download the code for this book from this GitHub [repo](#).

Chapter 1 Setup

Overview

Welcome to the world of Vue 3 and its Composition API, which didn't exist in the previous version of Vue. In this chapter, we will explore the process of setting up a Vue 3 project by using [Vite](#) (pronounced *veet*, and also created by Evan You, the creator of Vue) and understand the basic structure of the application.

Setup

To work with Vue 3, we need to have [Node.js](#) installed, which is a cross-platform and open-source JavaScript runtime environment.

To install Node.js, go to the Node.js website and download the **LTS** (long-term support) or the latest **Current** version. I suggest using the LTS version, which is recommended for most users, even though the latest Current version is OK for following along with this book.

Node.js® is an open-source, cross-platform JavaScript runtime environment.

Security releases now available

Download for Windows (x64)

18.18.2 LTS

Recommended For Most Users

20.8.1 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#) [Other Downloads](#) | [Changelog](#) | [API Docs](#)

For information about supported releases, see the [release schedule](#).

Figure 1-a: Node.js Website

The installation of Node.js is straightforward and consists of a few steps that can easily be performed by using a step-by-step installation wizard shown in the following figure:

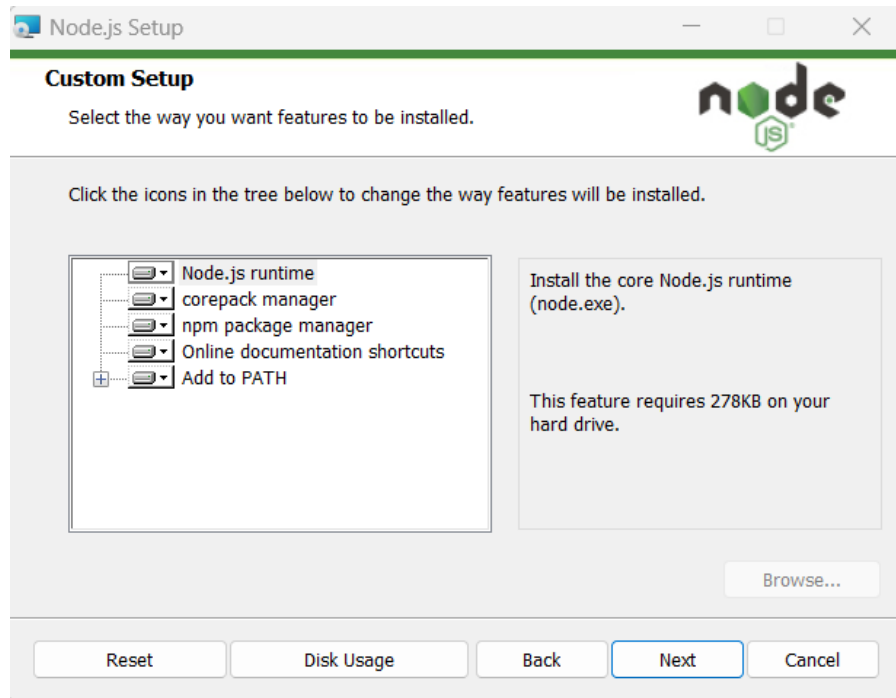


Figure 1-b: Node.js Installation Wizard

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

Code Listing 1-a: Vue CLI Installation Command

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

After installation, 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 install Vite. We can install Vite globally by using the following command:

Code Listing 1-c: Vite Installation Command

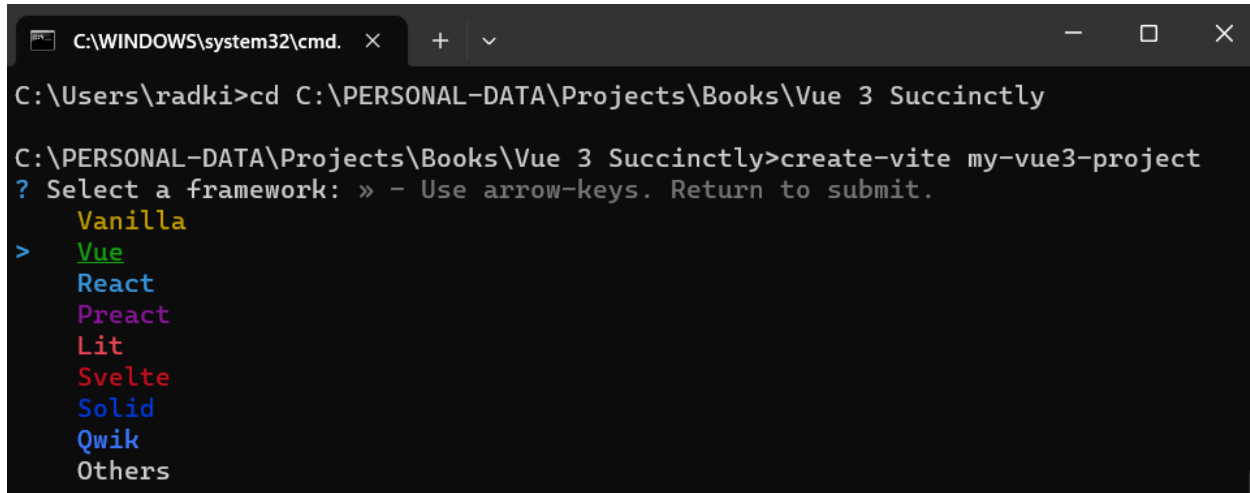
```
npm install -g create-vite
```

Once Vite is installed, you can navigate to your project directory and create a new Vue 3 project by running the following command:

Code Listing 1-d: Creating a Vue Project with Vite

```
create-vite my-vue3-project
```

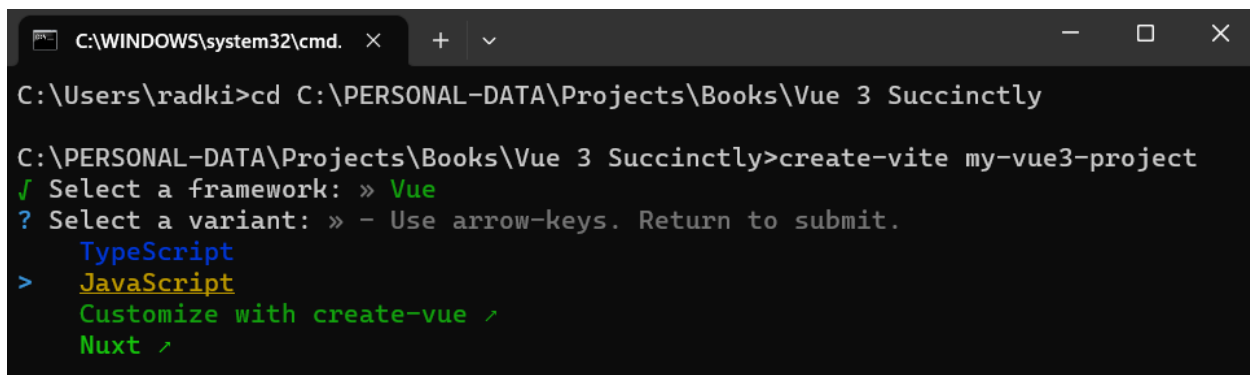
Once you have run this command, you will be requested to select a framework, as shown in the following figure:



```
C:\WINDOWS\system32\cmd. x + v - □ x
C:\Users\radki>cd C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly
C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly>create-vite my-vue3-project
? Select a framework: » - Use arrow-keys. Return to submit.
  Vanilla
>  Vue
  React
  Preact
  Lit
  Svelte
  Solid
  Qwik
  Others
```

Figure 1-c: Selecting a Framework (Creating a Vue Project with Vite)

Let's select **Vue** as the option and choose a variant; I will choose **JavaScript** rather than **TypeScript**.



```
C:\WINDOWS\system32\cmd. x + v - □ x
C:\Users\radki>cd C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly
C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly>create-vite my-vue3-project
✓ Select a framework: » Vue
? Select a variant: » - Use arrow-keys. Return to submit.
  TypeScript
>  JavaScript
  Customize with create-vue ✓
  Nuxt ✓
```

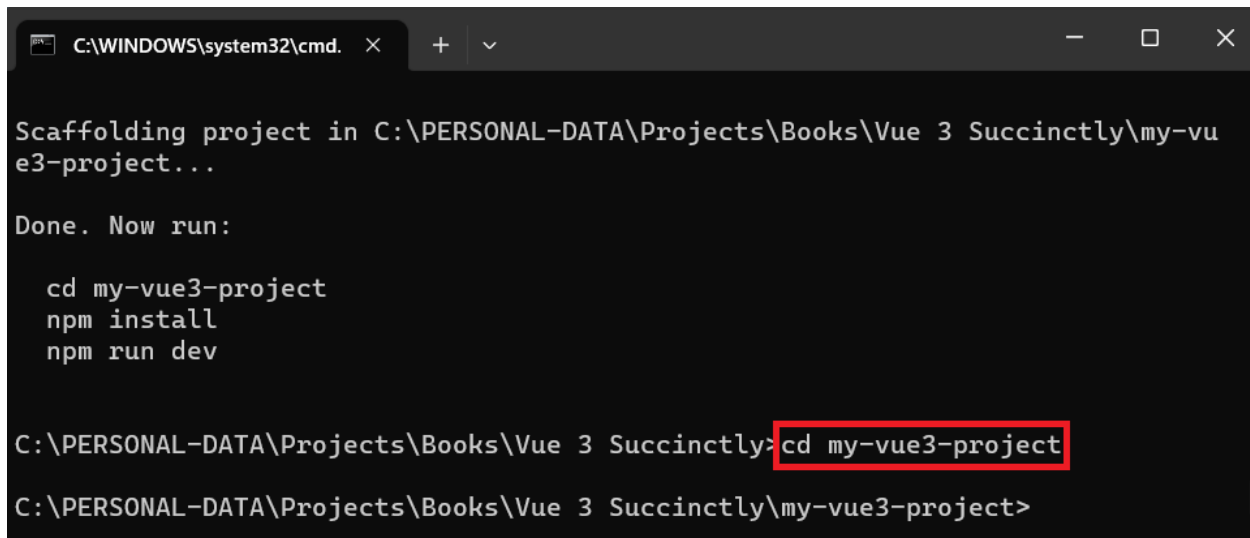
Figure 1-d: Selecting a Variant (Creating a Vue Project with Vite)

Once you're done, navigate to the project directory and run the following commands in the order specified:

Code Listing 1-e: Commands to Install Packages and Run the Project in Development Mode

```
cd my-vue3-project
npm install
npm run dev
```

Let's visualize these steps: first, navigate to the created Vue application folder.



```
C:\WINDOWS\system32\cmd. x + v - □ x

Scaffolding project in C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly\my-vue3-project...

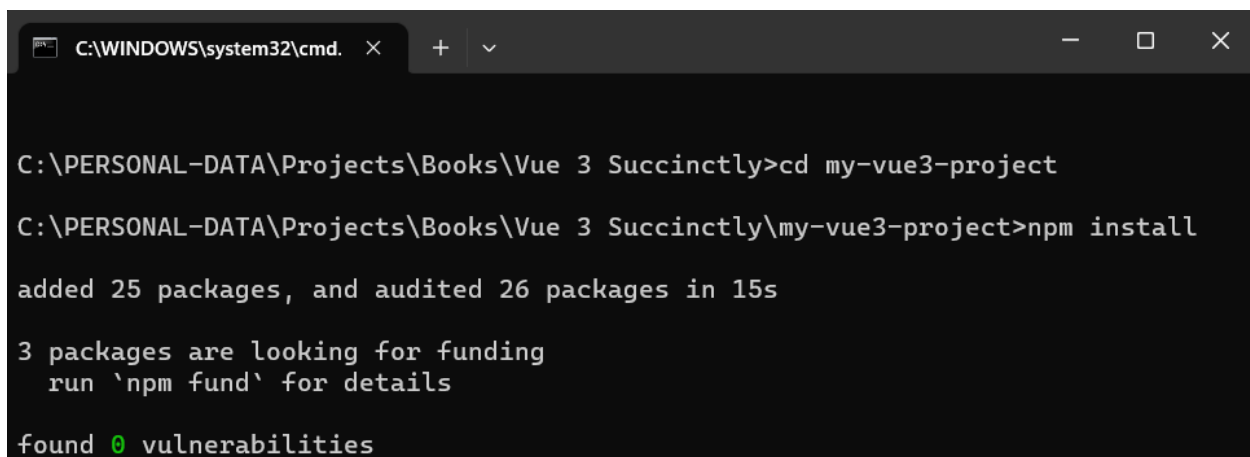
Done. Now run:

  cd my-vue3-project
  npm install
  npm run dev

C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly>cd my-vue3-project
C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly\my-vue3-project>
```

Figure 1-e: Navigating to the Vue Application Folder

Next, install the required NPM packages by running the `npm install` command.



```
C:\WINDOWS\system32\cmd. x + v - □ x

C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly>cd my-vue3-project
C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly\my-vue3-project>npm install

added 25 packages, and audited 26 packages in 15s

3 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figure 1-f: Installing NPM Packages

Once we have installed the required NPM packages, we can run the application created in development mode by using the `npm run dev` command.

```
C:\WINDOWS\system32\cmd. x + v - □ ×
found 0 vulnerabilities

C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly\my-vue3-project>npm run dev

> my-vue3-project@0.0.0 dev
> vite

VITE v4.4.11 ready in 3486 ms

→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h to show help
```

Figure 1-g: Running the App in Dev Mode

Once the application is running, we can navigate to the highlighted **localhost** URL and see what the application looks like.

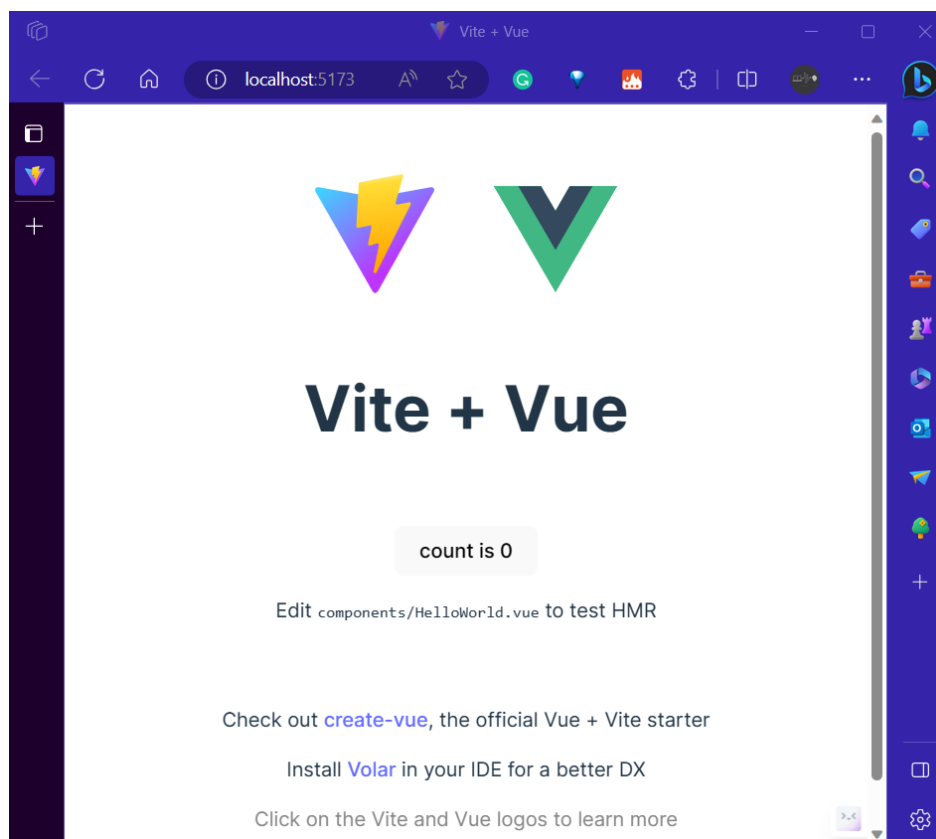


Figure 1-h: The App in Dev Mode (Edge Browser)

Excellent—that's the scaffolded application running in the browser.

Root component

Let's look at the default project structure of the application just created by using Vite.

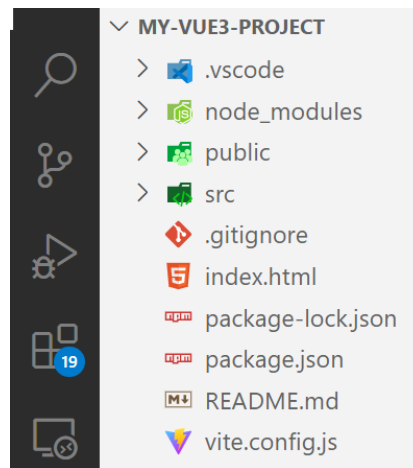


Figure 1-i: The Default App Structure (VS Code)

Understanding the structure of a Vue 3 application is essential for effective development. In a Vite project, your application's entry point is usually the **src/main.js** file. We import and mount the root (**App**) Vue component within this file.

Code Listing 1-f: The src/main.js File

```
import { createApp } from 'vue'
import './style.css'
import App from './App.vue'

createApp(App).mount('#app')
```

The root (**App**) Vue component is defined in the **src/App.vue** file. This component serves as the main structure for the application and is where we'll start building the UI later.

Code Listing 1-g: The src/App.vue File

```
<script setup>
import HelloWorld from './components/HelloWorld.vue'
</script>

<template>
  <div>
    <a href="https://vitejs.dev" target="_blank">
      
    </a>
    <a href="https://vuejs.org/" target="_blank">
```

```

    
  </a>
</div>
<HelloWorld msg="Vite + Vue" />
</template>

<style scoped>
.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.vue:hover {
  filter: drop-shadow(0 0 2em #42b883aa);
}
</style>

```

The **src/App.vue** file has three sections. First, we have the **script** section, which includes the JavaScript or TypeScript code.

Then, we have the **template** section, where the HTML and markup are defined, and then the **style** section, where the CSS is defined. The order in which these sections appear is not relevant.

As you can see, within one Vue file, we can have code, HTML markup, and CSS, making it convenient and easy to create modular applications.

Recap

In this short chapter, we've looked at setting up the environment and creating a Vue 3 app by using Vite, which, as you saw, was straightforward.

In the chapters that follow, we'll dive deeper into Vue 3 and its concepts and features, and implement the application.

Chapter 2 App Basics

Prerequisites

Before we jump into the specifics of Vue 3, I'd like to cover some prerequisites that will help you follow along and make the most of what this book covers.

First, you must understand HTML and JavaScript well, and have at least some notion of CSS. Furthermore, some knowledge of Node.js and its ecosystem is excellent to have.

Second, you need to use a good text editor and development environment. In my case, I'll be using [Visual Studio Code](#) (VS Code), which I highly recommend.

If you use VS Code, some great extensions will come in handy, such as [Volar](#) and [Vue VSCode Snippets](#), which I suggest you install, although they are not mandatory to follow along.

Additional dependencies

For the application that we are going to build, we will be using some additional dependencies.

We will use an NPM package called [uid](#), which creates random strings as unique IDs by using the [UUID](#) version 4 format.

Beyond that, we will use the [sass](#) NPM package and install [Oh, Vue Icons!](#), which includes modern, nice-looking icons.

So, let's get these additional dependencies installed. We'll begin by installing the uid package. We can do this by running the following command from the built-in terminal within VS Code:

Code Listing 2-a: Command to Install the uid Package

```
npm install --save uid
```

If you open the project's **package.json** file, you'll see that uid has been added (highlighted in bold).

Code Listing 2-b: The Project's Updated package.json File

```
{
  "name": "my-vue3-project",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
```

```

    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "uid": "^2.0.2",
    "vue": "^3.3.4"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^4.2.3",
    "vite": "^4.4.5"
  }
}

```

Now, let's install the sass package. We can do this by running the following command from the built-in terminal within VS Code:

Code Listing 2-c: Command to Install the sass Package

```
npm install --save-dev sass
```

In this case, the sass package will be added to the project's development dependencies. After running the command, we can verify this by looking at the **package.json** file (the package is highlighted in bold).

Code Listing 2-d: The Project's Updated package.json File

```

{
  "name": "my-vue3-project",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "uid": "^2.0.2",
    "vue": "^3.3.4"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^4.2.3",
    "sass": "^1.69.4",

```

```
    "vite": "^4.4.5"
  }
}
```

Finally, let's install the icon library, which we can do by running the following command from the built-in terminal within VS Code:

Code Listing 2-e: Command to Install Oh, Vue Icons!

```
npm install oh-vue-icons
```

The Oh, Vue Icons! package will be added to the project's dependencies. After running the command, we can verify this by looking at the **package.json** file (the package is highlighted in bold).

Code Listing 2-f: The Project's Updated package.json File

```
{
  "name": "my-vue3-project",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "oh-vue-icons": "^1.0.0-rc3",
    "uid": "^2.0.2",
    "vue": "^3.3.4"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^4.2.3",
    "sass": "^1.69.4",
    "vite": "^4.4.5"
  }
}
```

Awesome. Now that all the additional dependencies have been installed, we can do some project cleanup before we add any extra code.

Project cleanup

We first want to remove the **HelloWorld.vue** file since we won't use this Vue component as we move on with the application.

We can do this within the VS Code **Explorer** pane by clicking on the **HelloWorld.vue** file and pressing **Shift+Delete**.

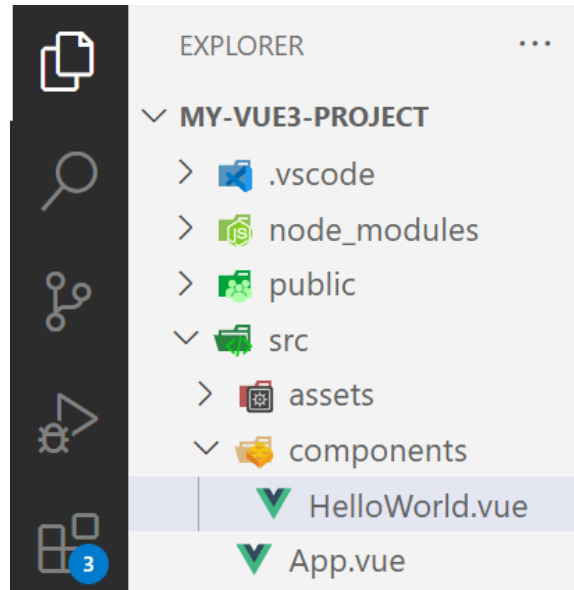


Figure 2-a: The HelloWorld.vue File within VS Code

We can remove the **vue.svg** file from the **src/assets** folder by clicking on the file name in the Explorer and then pressing **Shift+Delete**.

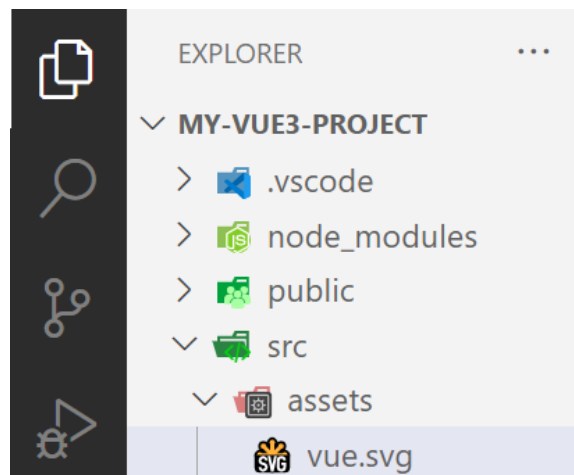


Figure 2-b: The vue.svg File within VS Code

Following that, let's make some changes to the **App.vue** file by removing some of the original boilerplate code and making the following changes (highlighted in bold):

Code Listing 2-g: The Updated App.vue File

```
<script setup>
import { RouterView } from "vue-router";
</script>

<template>
  <RouterView />
</template>

<style lang="scss">
</style>
```

What we have done here is removed the original code and reference to the **HelloWorld** component, imported the Vue Router library, added the **RouterView** component, and also specified that we'll be using [Sass](#) (**scss**) rather than pure CSS.

Adding fonts and styles

The next thing we want to do is import Google Fonts right into the **style** section. But first, let's navigate to the Google Fonts [website](#) and search for the **Roboto** font.

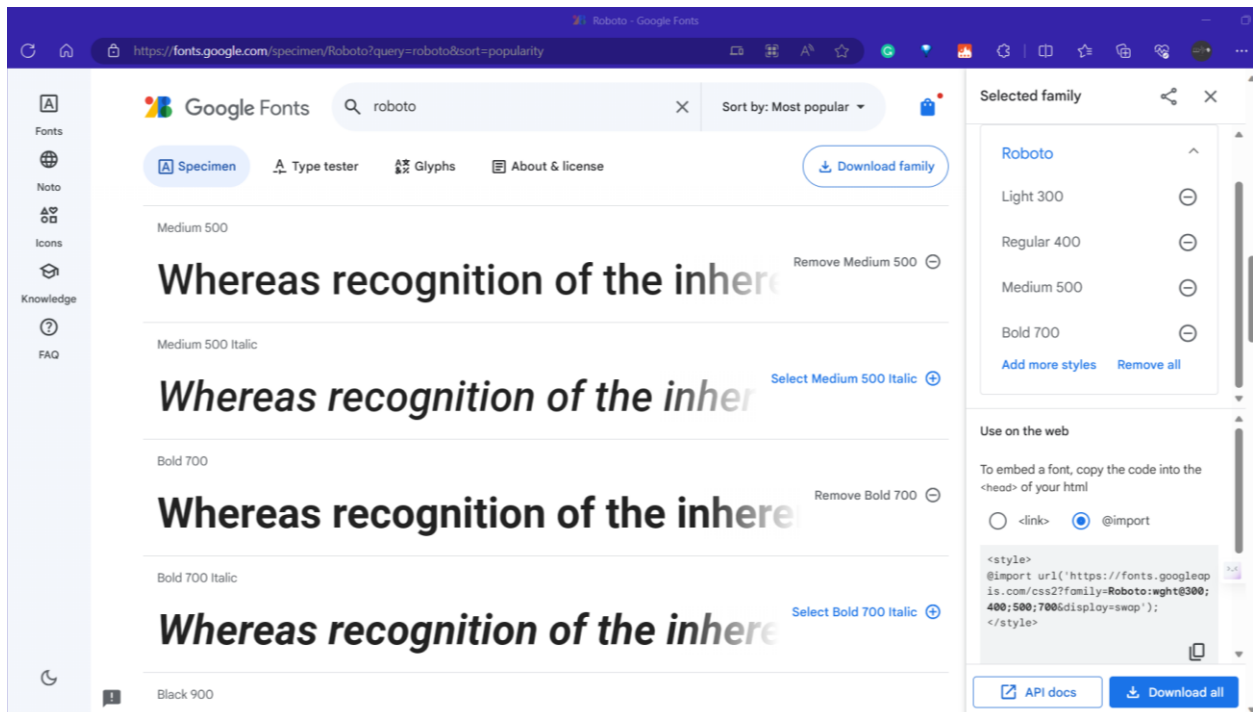


Figure 2-c: Roboto on the Google Fonts Website

Then, within the **Roboto** font, select **Light 300**, **Regular 400**, **Medium 500**, and **Bold 700**. Select the **@import** option, copy the generated embed code, and paste it within the **style** section of the **App.vue** file (change highlighted in bold).

Code Listing 2-h: The Updated App.vue File

```
<script setup>
import { RouterView } from "vue-router";
</script>

<template>
  <RouterView />
</template>

<style lang="scss">
@import
url('https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;500;700&display=swap');
</style>
```

Now that we have imported our application's fonts, let's add some basic styles. The changes are highlighted in bold, following the font's **@import** statement.

Code Listing 2-i: The Updated App.vue File

```
<script setup>
import { RouterView } from "vue-router";
</script>

<template>
  <RouterView />
</template>

<style lang="scss">
@import
url('https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;500;700&display=swap');

* {
  font-family: "Roboto", sans-serif;
  box-sizing: border-box;
  padding: 0;
  margin: 0;
}
.container {
```



```
margin: 0 auto;
max-width: 1100px;
}
</style>
```

Since we are building a relatively small application, adding the styles to each Vue file rather than the project's **src/assets** folder is acceptable.

With this done, we can begin coding the application. Let's start by creating the app's header component, which we'll use for navigation.

Creating the header component

Within the project's **src/assets** folder, I've removed the default **vue.svg** file and added the **logo-design.png** file, which I downloaded from the [Flaticon](#) website.

Then, within the **src/components** folder, create a new file called **Header.vue**.

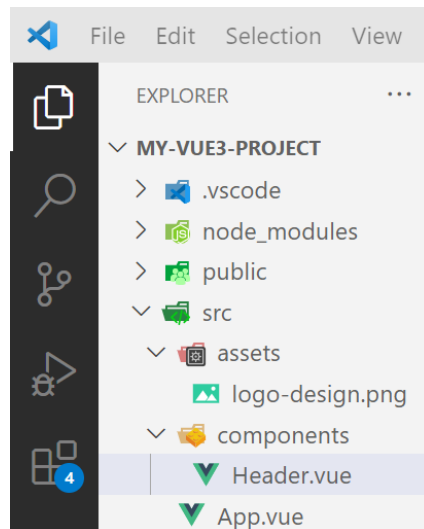


Figure 2-d: The Header.vue File (VS Code Explorer)

For now, let's leave the **Header.vue** file empty; we'll return to it shortly. Let's return to the **App.vue** file and import and reference the **Header** component (changes highlighted in bold).

Code Listing 2-j: The Updated App.vue File

```
<script setup>
import { RouterView } from "vue-router";
import Header from "../components/Header.vue";
</script>
```

```

<template>
  <Header />
  <RouterView />
</template>

<style lang="scss">
@import
url('https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;500;700&dis
isplay=swap');

* {
  font-family: "Roboto", sans-serif;
  box-sizing: border-box;
  padding: 0;
  margin: 0;
}
.container {
  margin: 0 auto;
  max-width: 1100px;
}
</style>

```

Let's add the following code to the **Header.vue** file:

Code Listing 2-k: The Header.vue File

```

<script setup>
</script>

<template>
  <header>
    <nav class="container">
      <div class="b">
        
        <h1>Doc Expire</h1>
      </div>
    </nav>
  </header>
</template>

<style lang="scss" scoped>
</style>

```

For now, the only thing we have done is add the logo to the **Header** component.

Before we run the application to see the changes, let's install the Vue Router package by running the following command from the built-in terminal in VS Code:

Code Listing 2-l: Command to Install Vue Router

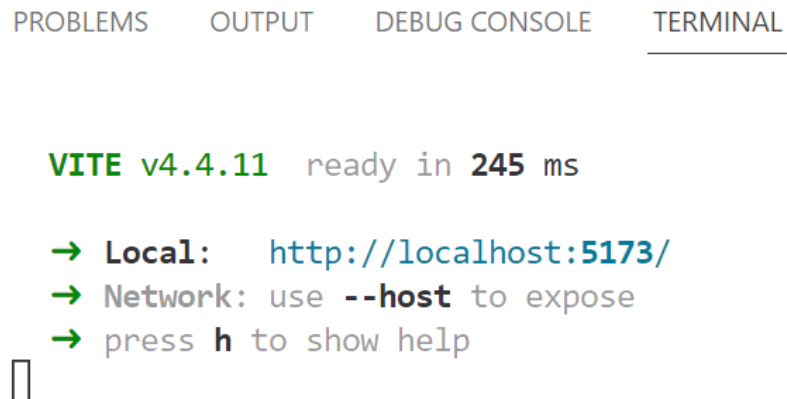
```
npm install vue-router
```

With Vue Router installed, let's run the application in development mode by running the following command within the built-in terminal in VS Code:

Code Listing 2-m: Command to Run the App in Dev Mode

```
npm run dev
```

Once we have run the command, the application will be running in development mode, and we can view the changes by opening the **localhost** URL.



The screenshot shows the VS Code interface with the 'TERMINAL' tab selected. The terminal output displays the Vite development server status: 'VITE v4.4.11 ready in 245 ms'. Below this, it provides the local URL 'http://localhost:5173/' and instructions to use '--host' for network access and to press 'h' for help. A cursor is visible at the bottom of the terminal.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
```

```
VITE v4.4.11 ready in 245 ms
```

```
→ Local:   http://localhost:5173/
```

```
→ Network: use --host to expose
```

```
→ press h to show help
```

```
█
```

Figure 2-e: The App Running (as Seen in the VS Code Terminal)

We can see the application's appearance by navigating to the **localhost** URL.



Doc Expire

Figure 2-f: The App Running in Dev Mode

Adding routes to the header component

Now that we have taken the initial steps with the **Header** component, let's add some routes to it (changes highlighted in bold).

Code Listing 2-n: The Modified Header.vue File

```
<script setup>
import { RouterLink } from "vue-router";
</script>

<template>
  <header>
    <nav class="container">
      <div class="b">
        
        <h1>Doc Expire</h1>
      </div>
      <ul class="nav-routes">
        <RouterLink to="/">Home</RouterLink>
        <RouterLink to="/about">About</RouterLink>
      </ul>
    </nav>
  </header>
</template>
```

```
</template>

<style lang="scss" scoped>
</style>
```

We first imported the reference to the **RouterLink** component from Vue Router and then created a **ul** element with the **nav** element. Within the **ul** element, we've added two **RouterLink** components.

By doing this, we have indicated that we'll use the **RouterLink** component to navigate to the **Home** page, and the other **RouterLink** component to navigate to the **About** page.

If we save the changes and run the application by running the **npm run dev** command from the built-in terminal within VS Code, we'll see minor changes to the app's UI (highlighted in **red** in the following figure).

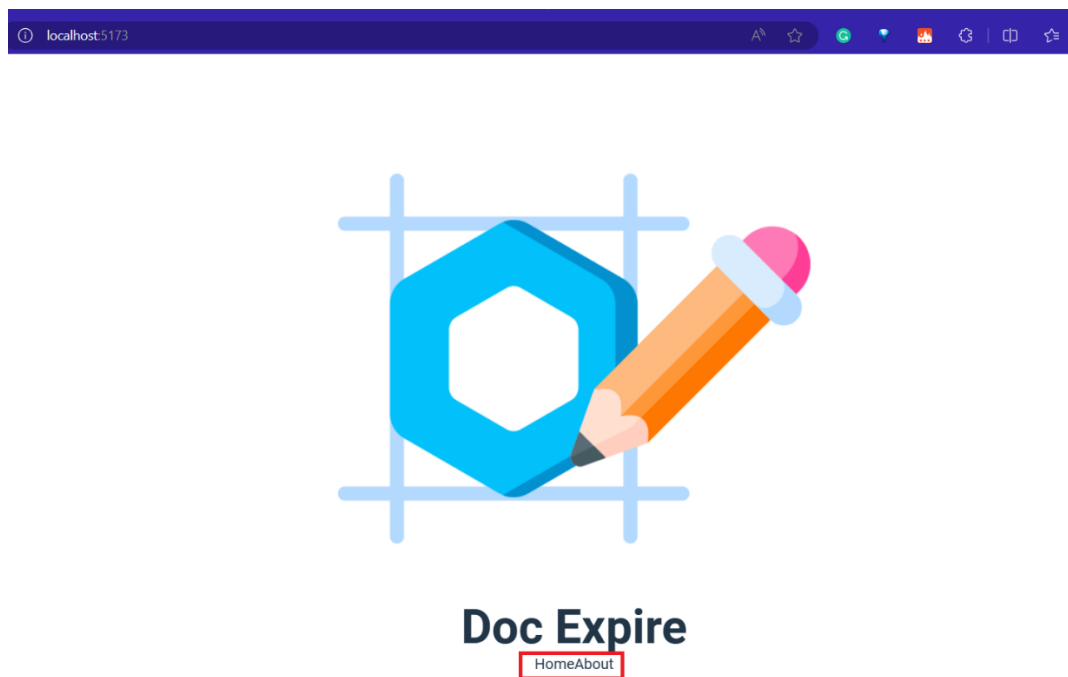


Figure 2-g: The App Running in Dev Mode with Recent Changes Highlighted

So, although we have declared within the **Header** component our intent to use routing, we haven't implemented it. Let's do that next.

Implementing the router

We have taken the initial steps with the **Header** component and added the two **RouterLink** components.

However, we need to implement the router. Let's create a new **router** folder under the project's **src** folder to do that. Within the **router** folder, let's create an **index.js** file.

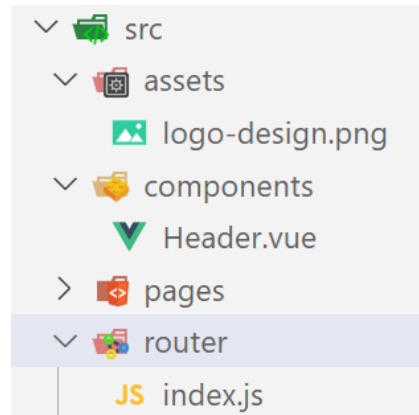


Figure 2-h: The index.js File (Within the router Folder)

Now, let's open the **index.js** file and add the following code:

Code Listing 2-o: The index.js File

```
import { createRouter, createWebHistory } from "vue-router";
import DocsPage from "../pages/DocsPage.vue";

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: "/",
      name: "home",
      component: DocsPage
    },
    {
      path: "/about",
      name: "about",
      component: () => import("../pages/AboutPage.vue")
    }
  ]
});

export default router;
```

The first thing we do is **import** the **createRouter** and **createWebHistory** components from the Vue Router.

Then, we import the **DocsPage** component (which we will create shortly) from the **DocsPage.vue** file that we'll create later within the **pages** folder of our project.

The **DocsPage** component will be where we will list all the documents that our application will track (with their expiration dates). Following good Vue development practices, we'll place the **DocsPage** component within the file **DocsPage.vue**.

The next thing we do is create a **createRouter** instance and assign that to the **router** constant. To the **createRouter** instance, we pass as a parameter an object that includes the **history** and **routes** properties.

We assign the value returned after invoking **createWebHistory** to the **history** property, to which the base URL (**BASE_URL**) is passed as a parameter. We assign an array containing two objects to the **routes** property, each representing a route our application will use.

The first object or element within the **routes** array is the **home** route, which points to the **DocsPage** component. The second object or element within the **routes** array is the **about** route, which points to the **AboutPage** component, included within the **AboutPage.vue** file (which we have yet to create).

Something to notice is that contrary to the **DocsPage** component, the **AboutPage** component is [lazy loaded](#), which defers the element's loading until it is needed. This helps optimize performance and resources.

Adjusting main.js

With the routing part of the application implemented, we need to adjust the code within the **main.js** file to use the routing mechanism we have just implemented. We can do that as follows (the changes are highlighted in bold):

Code Listing 2-p: The Updated main.js File

```
import { createApp } from "vue";
import App from "./App.vue";
import router from "./router";

const app = createApp(App);

app.use(router);

app.mount("#app");
```

So, we have imported the **router** component from the **index.js** file contained within the **router** folder.

Given that the **router** component resides within the **index.js** file, it is enough to indicate the folder name (**router**) within the file path, and the file name (**index.js**) can be omitted.

Then, **createApp** is invoked, and its value is assigned to **app**. Next, the **router** component is passed as a parameter to the **app.use** method—in this way, we are letting Vue know we'll be using this router.

Finally, the Vue app is mounted onto the **div** with an **id** value of **app**. This **div** is in the **index.html** file (located within the root of the project's **src** folder), and highlighted in bold in the following code listing:

Code Listing 2-q: The index.html File

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.1" />
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

So now we have implemented our app's routing mechanism. Next, let's create the **DocsPage** and **AboutPage** components to see our app's routing in action.

Initial DocsPage

Before creating the **DocsPage.vue** file, we must create a **pages** folder within the project's **src** folder, so let's do that.

Once we have created the **pages** folder, let's create the **DocsPage.vue** and **AboutPage.vue** files within the **pages** folder.

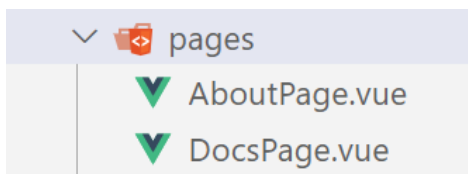


Figure 2-i: The pages Folder with AboutPage.vue and DocsPage.vue

With that done, let's add the following code to the **DocsPage.vue** file:

Code Listing 2-r: The DocsPage.vue File

```
<script setup>
</script>

<template>
  <main>
  </main>
</template>

<style lang="scss" scoped>
</style>
```

We now have the boilerplate we'll use later to build the **DocsPage** component.

Initial AboutPage

Let's add the boilerplate code we'll use for the **AboutPage** component, which we can add to the **AboutPage.vue** file.

Code Listing 2-s: The AboutPage.vue File

```
<script setup>
</script>

<template>
  <div>
    <h1>About Page...</h1>
  </div>
</template>

<style lang="scss" scoped>
</style>
```

After saving all the changes, we can run the application by using the **npm run dev** command; we'll be able to navigate between both pages using the routing mechanism we implemented. If we click the **About** link, we'll see the following:

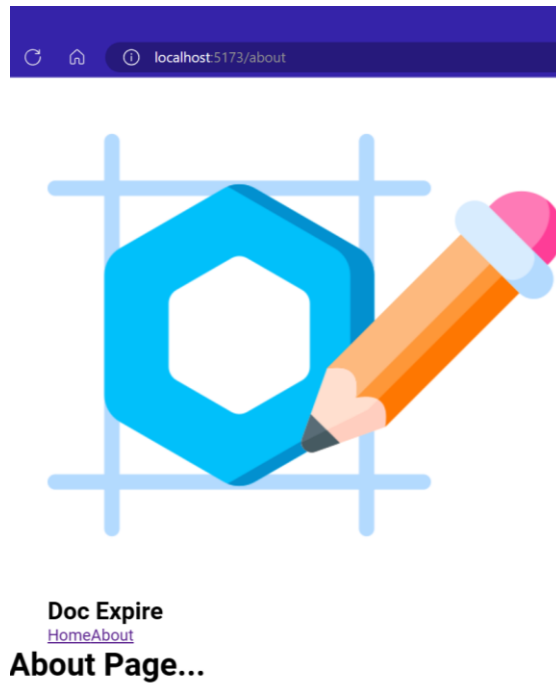


Figure 2-j: The About Page/Route

Great, so routing works, even though the UI doesn't look that appealing yet.

Recap

Throughout this chapter, we've managed to get our project off the ground and lay the foundations for the app we'll be building, such as the app's header, pages, and routing connections.

In the next chapter, we'll dive deeper into the application's functionality and look at critical aspects of Vue 3, such as the Composition API.

Chapter 3 Enhancing the App

With the application's foundation created, we can add additional functionality to the app to make it work as intended, and that's what this chapter is all about. Let's dive right in.

Finalizing the header

We need to change the **Header.vue** file to make our application look better. Let's add some styling and minor modifications to each **RouterLink** component. The changes are highlighted in bold in the following code listing:

Code Listing 3-a: The Updated Header.vue File

```
<script setup>
import { RouterLink } from "vue-router";
</script>

<template>
  <header>
    <nav class="container">
      <div class="b">
        
        <h1>Doc Expire</h1>
      </div>
      <ul class="nav-routes">
        <RouterLink :to="{ name: 'home' }">Docs</RouterLink>
        <RouterLink :to="{ name: 'about' }">About</RouterLink>
      </ul>
    </nav>
  </header>
</template>

<style lang="scss" scoped>
header {
  background-color: #bfffff;

  nav {
    .nav-routes {
      display: flex;
      list-style: none;
      flex: 1;
      justify-content: flex-end;
    }
  }
}
```

```

    gap: 15px;

    a {
      color: inherit;
      text-decoration: none;
    }
  }

  h1 {
    font-size: 20px;
  }

  padding: 9px 20px;
  display: flex;
  align-items: center;
}

.b {
  img {
    max-width: 40px;
  }
  display: flex;
  align-items: center;
  gap: 8px;
}
}
}
</style>

```

If you save the changes and run the application by using the `npm run dev` command, you should see that the application looks much better.

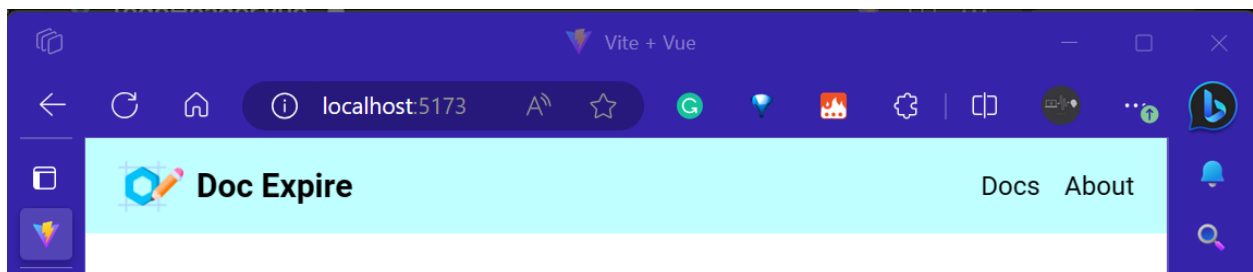


Figure 3-a: The Updated App UI (Header)

Essentially, we've only done two things. The first and most significant change was changing from static routing paths to dynamic ones. If you recall, the original routing paths (before these changes) were as follows:

```
<RouterLink to="/">Home</RouterLink>
<RouterLink to="/about">About</RouterLink>
```

Now, notice how the **to** attribute for each **RouterLink** component was set to a hardcoded route. Given that both pages exist, this works; however, by doing this, we aren't using the routing mechanism we created within the **index.js** file of the **router** folder.

To use the routing mechanism we created within the **index.js** file of the **router** folder, we have to bind the **to** attribute to the actual name of the route, i.e., `{ name: 'home' }`.

```
<RouterLink :to="{ name: 'home' }">Docs</RouterLink>
<RouterLink :to="{ name: 'about' }">About</RouterLink>
```

To understand this better, let's look at the following diagram:

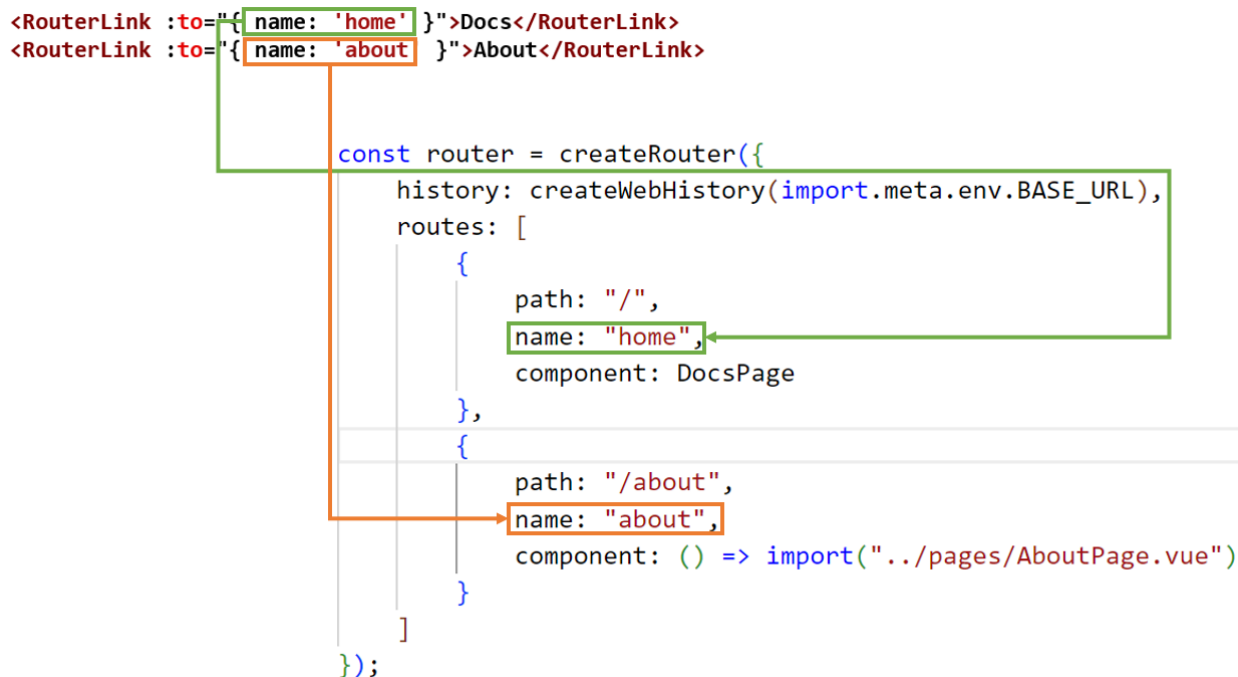


Figure 3-b: RouterLink Binding to the App's Router

The preceding diagram shows that the **to** attribute binds (is connected) to the **name** property within the router.

By doing it this way, if needed, we can change the route at any point without changing the **RouterLink** component.

Another advantage is that if we use the **RouterLink** component in another part of the application, we don't have to worry about updating it because if the route changes, the only place we'll have to update is the router.

The second change was adding styling to improve the header. This book focuses on Vue 3 functionality and not styling, so we won't go into the styling details. As you can see, we added some padding, colors, and margins, and used a flexbox to position the header correctly. That styling code will be presented shortly.

DocsPage architecture

Now that the look and feel of the navigation header are better and the router is correctly used, let's see how we can add functionality to the **DocsPage.vue** file. But before we do anything, let's understand how the **DocsPage** component will work and its functionality.

The **DocsPage** component needs to accomplish two things: The first is to allow the user to enter a document to track, and the second is to display the list of documents. To understand this better, let's look at the following diagram:

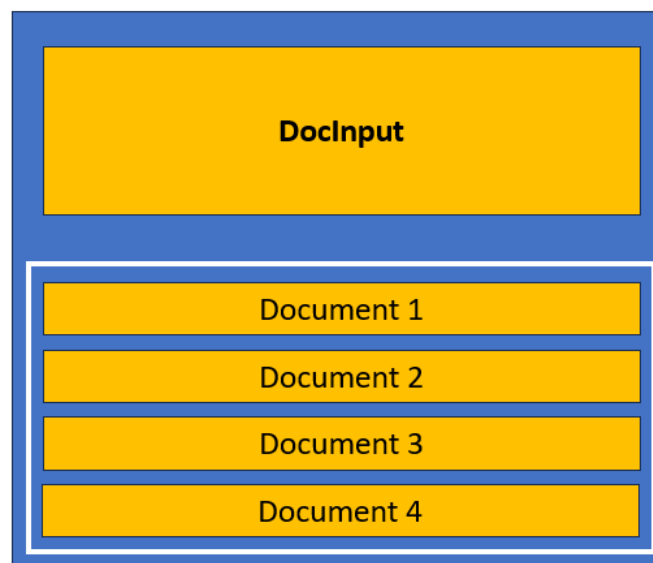


Figure 3-c: The DocsPage Component Architecture

The preceding diagram shows that the **DocsPage** component will comprise two other elements: the **DocInput** and **Document** components.

The **DocInput** component will be used for entering a new document, and the **Document** component will display a document previously entered by the user. The **DocsPage** component will display a list of **Document** components.

Later, we'll create the **DocInput** and the **Document** components within the project's **components** folder.

Styling DocsPage

Now that we have seen a high-level overview of the **DocsPage** component architecture, let's add some proper functionality. We can first add some styling for the list of documents we'll want to display. The changes are highlighted in bold in the following code listing:

Code Listing 3-b: The Updated DocsPage.vue File with Styling

```
<script setup>
</script>

<template>
  <main>
  </main>
</template>

<style lang="scss" scoped>
main {
  h1 {
    text-align: center;
    margin-bottom: 18px;
  }

  margin: 0 auto;
  width: 100%;
  max-width: 500px;
  display: flex;
  flex-direction: column;
  padding: 30px 15px;

  .docs-list {
    display: flex;
    flex-direction: column;
    gap: 18px;
    list-style: none;
    margin-top: 25px;
  }

  .docs-msg {
    align-items: center;
    justify-content: center;
    display: flex;
    gap: 10px;
    margin-top: 25px;
  }
}
```

```
}  
</style>
```

This styling will come to life when we finish creating the **DocsPage** component. We'll come back to the **DocsPage.vue** file shortly.

Installing Vue Datepicker

When adding a new document, we need to be able to indicate the document's expiration date; for that, we'll need to use a date picker.

There is an excellent date picker component for Vue 3 called [Vue Datepicker](#), which we can use for this purpose. So, let's install this component by running the following command:

Code Listing 3-c: Command to Install Vue Datepicker

```
npm install @vuepic/vue-datepicker
```

Once the component has been installed, if we open the project's **package.json** file, we can see it has been added to the dependencies section (highlighted in bold).

Code Listing 3-d: Updated package.json File

```
{  
  "name": "my-vue3-project",  
  "private": true,  
  "version": "0.0.0",  
  "type": "module",  
  "scripts": {  
    "dev": "vite",  
    "build": "vite build",  
    "preview": "vite preview"  
  },  
  "dependencies": {  
    "@vuepic/vue-datepicker": "^7.2.0",  
    "oh-vue-icons": "^1.0.0-rc3",  
    "uid": "^2.0.2",  
    "vue": "^3.3.4",  
    "vue-router": "^4.2.5"  
  },  
  "devDependencies": {  
    "@vitejs/plugin-vue": "^4.2.3",  
    "sass": "^1.69.4",  
    "vite": "^4.4.5"  
  }  
}
```



```
}  
}
```

Creating DocInput

As we have previously seen, the **DocsPage** component will use two parts (subcomponents); one is called **DocInput**, which will be used to enter a new document. Let's create the **DocInput** component; within the project's **components** folder, we'll create a new file called **DocInput.vue**.

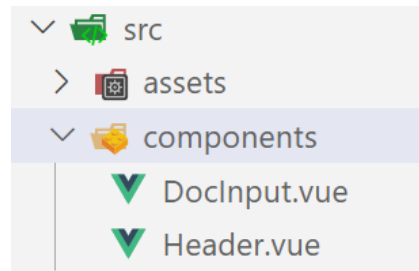


Figure 3-d: The components Folder (Including DocInput.vue)

Once the **DocInput.vue** file has been created, open it and add the following styling (highlighted in bold):

Code Listing 3-e: The DocInput.vue File (Styling Included)

```
<script setup>  
</script>  
  
<template>  
</template>  
  
<style lang="scss" scoped>  
  .inputdoc {  
    display: flex;  
    border: 1px solid #d4f4fa;  
  
    &.input-err {  
      border-color: red;  
    }  
  
    input {  
      width: 100%;  
      padding: 5px 4px;  
      border: none;  
    }  
  }  
</style>
```

```

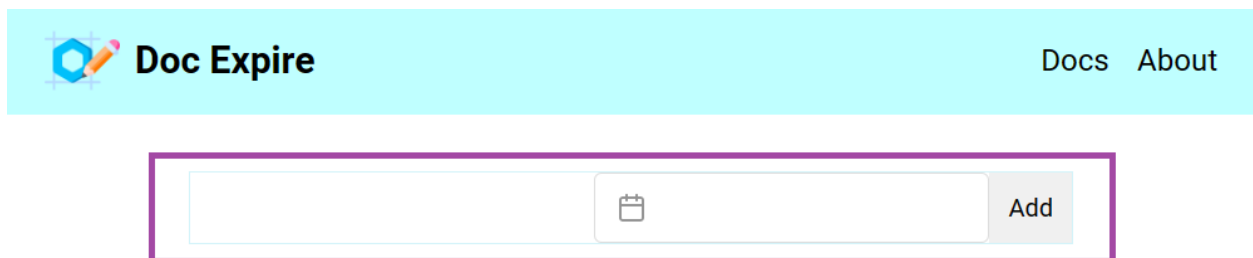
    &:focus {
      outline: none;
    }
  }

  button {
    padding: 5px 10px;
    border: none;
  }
}

.err-msg {
  margin-top: 6px;
  font-size: 12px;
  text-align: center;
  color: red;
}
</style>

```

What we've done here is add some simple styles that we'll use for entering a document. Let me show you what the finished **DocInput** component will look like (highlighted in **purple**, which is not part of the styling).



The screenshot shows a web application header with a logo and the text "Doc Expire" on the left, and links "Docs" and "About" on the right. Below the header is a form component highlighted with a thick purple border. The form consists of two input fields: the first is for a document description, and the second is for an expiration date, indicated by a calendar icon. To the right of the second input field is a button labeled "Add".

Figure 3-e: The Finished DocInput Component (Highlighted in Purple)

In essence, the **DocInput** component will have two fields and a button. The first field will be for entering the document's description, and the second field will be for entering the document's expiration date. The document will be added to the document list when the **Add** button is clicked.

To make this happen, we need to add the markup next. So, let's do that. The changes are highlighted in bold.

Code Listing 3-f: The DocInput.vue File (Markup Included)

```
<script setup>
</script>

<template>
  <div class="inputdoc">
    <input v-model="docDescription" type="text" />
    <VueDatePicker
      :enable-time-picker="false"
      :min-date="new Date()"
      :max-date=tenYears
      :flow="flow"
      now-button-label="Today"
      v-model="docDate" />
    <button @click="createDoc">Add</button>
  </div>
</template>

<style lang="scss" scoped>
.inputdoc {
  display: flex;
  border: 1px solid #d4f4fa;

  &.input-err {
    border-color: red;
  }

  input {
    width: 100%;
    padding: 5px 4px;
    border: none;

    &:focus {
      outline: none;
    }
  }

  button {
    padding: 5px 10px;
    border: none;
  }
}

.err-msg {
```

```
margin-top: 6px;
font-size: 12px;
text-align: center;
color: red;
}
</style>
```

Let's take the time to dissect and understand the markup code we've just included. As you can see, everything has been wrapped around a **div**, which uses the **inputdoc** CSS class—this way, all the elements contained within the **div** have the same styling.

Then, within the **div**, the first element is an **input** text field, which we will use for entering a document's description.

```
<input v-model="docDescription" type="text" />
```

Using Vue's two-way data binding mechanism (using the **v-model** directive), we bind this **input** text field to a **docDescription** variable, which we'll declare later.

So, two-way binding works this way. The value entered through the **input** field is assigned to the **docDescription** variable. If the value of the **docDescription** variable changes, the value of the **input** text field also changes.

Using VueDatePicker

Our next element is the **VueDatePicker** component, which we previously installed. We will use this element to indicate the document's expiration date.

```
<VueDatePicker
  :enable-time-picker="false"
  :min-date="new Date()"
  :max-date=tenYears
  :flow="flow"
  now-button-label="Today"
  v-model="docDate" />
```

As you can see, this element has a few properties that are vital to our application. These properties are documented adequately on the Vue Datepicker [website](#).

The first property is **enable-time-picker**. This property, by default, is set to **true**, which means that when a date is chosen, it is also possible to select the time. We don't want that since we are just interested in selecting an expiration date, as official documents like passports, driver's licenses, and even credit cards don't expire at a specific time.

This is why we expressly indicate that we don't need to use the time picker, and why we set the **enable-time-picker** property to **false**.

Another feature we want is the ability to select an expiration date in the future, not a date in the past. We need to indicate a minimum date (**min-date**) value, using today's date (**new Date()**) as a starting point.

However, we also don't want the future date that we select as the document's expiration date to be too far into the future. Most official documents, such as passports, ID cards, or driver's licenses, usually have a maximum validity of ten years (with some rare exceptions). Debit or credit cards, on the other hand, are valid for five years, most of the time.

So, to simplify our lives, the maximum date we can select for a document's expiration date will be 10 years from today's date, which should be enough for most cases.

To achieve that, we assign to the **max-date** property the value returned by the **tenYears** function (which we will create shortly).

Another nice-to-have feature when entering a document's expiration date would be to quickly drill down to the specific date.

Usually, when using a calendar widget in most web applications, you are presented with a view in which you see all the days of the current month for the current year.

Imagine having to click a forward arrow (highlighted in **red** in Figure 3-f) on the calendar widget to move to the following month, one month at a time, to eventually get to the month corresponding to the document's expiration date five or ten years in the future. Let that sink in for a moment.

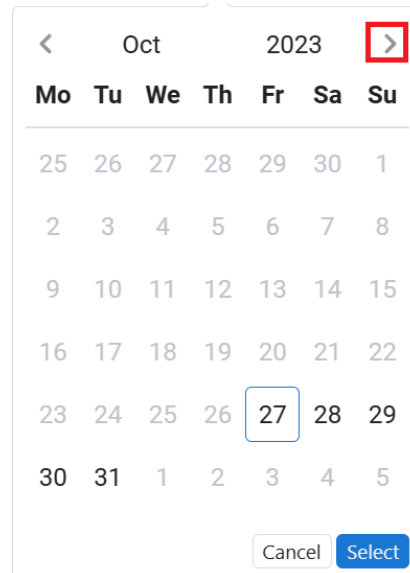


Figure 3-f: The Typical Calendar Widget

That's a lot of clicks—and a guaranteed way to discourage users from using the application. This is why the **flow** property is crucial for the app's ease of use (and the main reason I chose to use the **VueDatePicker** component for this app).

So, what this **flow** property does is quite impressive. By assigning the **flow** value to the **flow** property, we can instruct the **VueDatePicker** component to display the calendar differently. In our case, we want to be able to select the document's expiration year first, then the month, and finally the day—all in no more than four clicks. Let me show you what I mean.

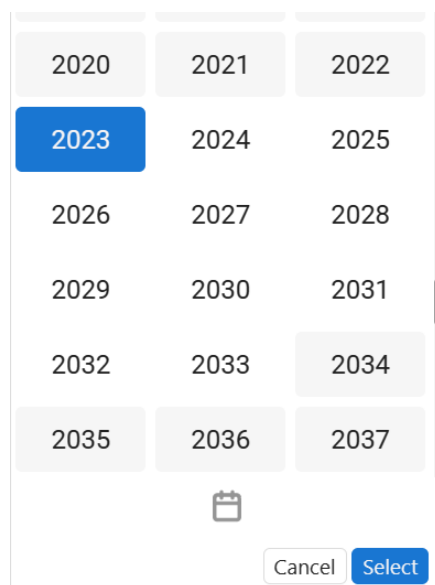


Figure 3-g: Year Selection in the Calendar Widget

First, we select the year. Notice how only the current year (2023 at the time of writing) and the following ten years can be chosen. The other years are unavailable (based on the values assigned to the **min-date** and **max-date** properties).

Once the year has been selected (in my case, I've chosen 2033), we can determine the month as shown in the next figure.

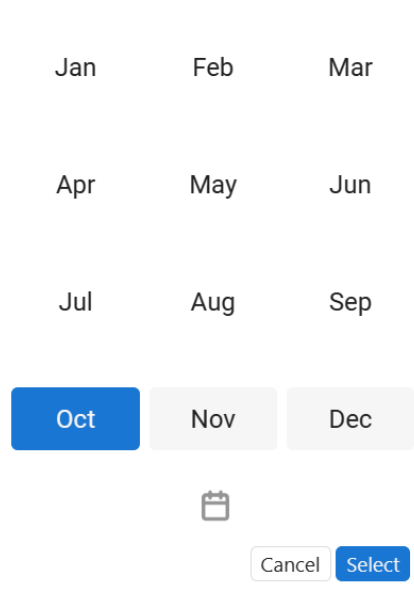


Figure 3-h: Month Selection in the Calendar Widget

Notice that because of the current month and year (October 2023 at the time of writing), we are not allowed to select the months of November and December of 2033, which makes sense because those are more than ten years after the initial date.

So, let's choose the month. In my case, I've selected October. After that, we can see the days of the month and choose one.



Figure 3-i: Day Selection in the Calendar Widget

Once we have chosen a day, we can click **Select**. The beauty of using the **flow** property is that the calendar can be adjusted to select the document's expiration date with as few clicks as possible.

The **now-button-label** property is nothing but a button (not visible) with the caption **Today**, which you can click to select today's date. (It is not visible because we are using the calendar with **flow** mode enabled.)

You might have noticed that all the previous properties described, except the **now-button-label** property, use one-way data-binding, which is what the **:** character means before the name of each property.

The last property corresponds to the **v-model** directive, which is used for two-way binding the date selected to the **docDate** variable, which we will create shortly.

As you have seen, there's quite a lot of functionality within the **VueDatePicker** component, and we've just scratched the surface of what this component can do.

Add button

The last part of the markup of the **DocInput** component is the **Add** button. As its name implies, this button adds a new document to the list; we will create the list of documents later.

This button works because when the user clicks it, the `createDoc` event is triggered; this is why the `@click` directive is used. We'll explore the code for the `createDoc` event next.

DocInput JavaScript logic

Now that we have covered the markup functionality of the `DocInput` component, let's add the necessary logic to bring it to life.

Let's add the following code to the `script setup` section of the `DocInput.vue` file; the changes are highlighted in bold:

Code Listing 3-g: The DocInput.vue File with JavaScript Logic Included

```
<script setup>
import { ref } from "vue";
import VueDatePicker from '@vuepic/vue-datepicker';
import '@vuepic/vue-datepicker/dist/main.css';

const docDescription = ref("");

const today = new Date();
const docDate = ref();
const flow = ref(['year', 'month', 'calendar']);

const tenYears = new Date(today.setFullYear(today.getFullYear() + 10));

const emit = defineEmits(["create-doc"]);

const createDoc = () => {
  emit("create-doc", docDescription.value, docDate.value);
}
</script>

<template>
  <div class="inputdoc">
    <input v-model="docDescription" type="text" />
    <VueDatePicker
      :enable-time-picker="false"
      :min-date="new Date()"
      :max-date=tenYears
      :flow="flow"
      now-button-label="Today"
      v-model="docDate" />
    <button @click="createDoc">Add</button>
  </div>
```



```

</template>

<style lang="scss" scoped>
.inputdoc {
  display: flex;
  border: 1px solid #d4f4fa;

  &.input-err {
    border-color: red;
  }

  input {
    width: 100%;
    padding: 5px 4px;
    border: none;

    &:focus {
      outline: none;
    }
  }

  button {
    padding: 5px 10px;
    border: none;
  }
}

.err-msg {
  margin-top: 6px;
  font-size: 12px;
  text-align: center;
  color: red;
}
</style>

```

Let's review each of the parts of the code highlighted in bold in the previous code. The first thing we do is import the modules and dependencies required.

```

import { ref } from "vue";
import VueDatePicker from '@vuepic/vue-datepicker';
import '@vuepic/vue-datepicker/dist/main.css';

```

We begin by importing **ref** from the Vue core module, which allows us to add reactivity to any variable and perform the data binding. Unlike **reactive**, which applies only to objects, **ref** can apply to variables.

Then, we import the **VueDatePicker** component from **@vuepic/vue-datepicker** and this component's style from **@vuepic/vue-datepicker/dist/main.css**.

Next, we declare **docDescription** with an initial empty string value by using **ref**, making it reactive, which we bind to the **input** field by using the **v-model** directive. We declare it by using the **const** keyword because we won't change its value by code.

```
const docDescription = ref("");
```

Moving on, we declare **today** using **const**, as this will store today's date (**new Date()**), which we will use to calculate the date ten years in the future.

Next, we declare **docDate** with no initial value by using **ref**, making it reactive, and bind it to the **VueDatePicker** component by using the **v-model** directive. We also declare it by using the **const** keyword because we won't change its value by code.

```
const docDate = ref();
```

We also declare **flow** with an array as an initial value and use **ref** to make it reactive. The **flow** property of the **VueDatePicker** component binds to **flow**.

As previously explained, the **flow** array values are listed in that specific order (**'year'**, **'month'**, **'calendar'**) so the user doesn't have to click too many times to select the document's expiration date.

```
const flow = ref(['year', 'month', 'calendar']);
```

Then, we declare **tenYears** as a **const**, creating a new date 10 years in the future based on today's date, as follows:

```
const tenYears = new Date(today.setFullYear(today.getFullYear() + 10));
```

Next, we declare **emit** as a **const** and assign **defineEmits** to it by indicating which event (in this case, **create-doc**) the component can emit to the parent component (**DocsPage**).

```
const emit = defineEmits(["create-doc"]);
```

In essence, **defineEmits** is a Vue compiler macro that is only usable within a **script setup** section, and we use it to specify that we will emit the **create-doc** event to the parent component of **DocInput**—in this case, **DocsPage**.

In other words, when the user clicks the **Add** button, adding the document to the list won't be done by the **DocInput** component; instead, it will be run by the **DocsPage**.

The **DocInput** component notifies the **DocsPage** component by emitting the **create-doc** event.

The **create-doc** event is emitted when **createDoc** runs, which happens when the **Add** button is clicked.

```
<button @click="createDoc">Add</button>
```

This is possible because the `createDoc` function binds to the button's `click` event. When the `create-doc` event is emitted, the values of `docDescription` and `docDate` are passed to the parent component.

```
emit("create-doc", docDescription.value, docDate.value);
```

The other important aspect of the code is that we are using `script setup`. [This](#) is syntactic sugar for the Composition API as it reduces the amount of code required and provides better runtime performance.

So, that's all the essential logic required for the `DocInput` component. However, there's one more thing I'd like to do, which is to validate that the document's description (`docDescription`) is always entered.

Validating docDescription

To ensure that the document's description is entered and not left blank, there's a bit of extra code that we need to add. The following is the finished code for the `DocInput` component with these changes highlighted in bold:

Code Listing 3-h: The DocInput.vue File (Description Validation Included)

```
<script setup>
import { ref } from "vue";
import VueDatePicker from '@vuepic/vue-datepicker';
import '@vuepic/vue-datepicker/dist/main.css';

const docDescription = ref("");
const invalid = ref(false);
const errMsg = ref("");

const today = new Date();
const docDate = ref();
const flow = ref(['year', 'month', 'calendar']);

const tenYears = new Date(today.setFullYear(today.getFullYear() + 10));

const emit = defineEmits(["create-doc"]);

const createDoc = () => {
  invalid.value = false;
  if (docDescription.value !== "") {
    emit("create-doc", docDescription.value, docDate.value);
    docDescription.value = "";
    return;
  }
}
```

```

    }
    invalid.value = true;
    errMsg.value = "Document description not provided";
  }
</script>

<template>
  <div class="inputdoc" :class="{ 'input-err': invalid }">
    <input
      v-model="docDescription" type="text" />
    <VueDatePicker
      :enable-time-picker="false"
      :min-date="new Date()"
      :max-date=tenYears
      :flow="flow"
      now-button-label="Today"
      v-model="docDate"
    />
    <button
      @click="createDoc">Add
    </button>
  </div>
  <p v-show="invalid" class="err-msg"> {{ errMsg }}</p>
</template>

<style lang="scss" scoped>
.inputdoc {
  display: flex;
  border: 1px solid #d4f4fa;

  &.input-err {
    border-color: red;
  }

  input {
    width: 100%;
    padding: 5px 4px;
    border: none;

    &:focus {
      outline: none;
    }
  }
}

```

```

    button {
      padding: 5px 10px;
      border: none;
    }
  }

  .err-msg {
    margin-top: 6px;
    font-size: 12px;
    text-align: center;
    color: red;
  }
</style>

```

The first thing to notice is that I've declared `invalid` and `errMsg` as reactive by using `ref`. We use `ref` to check whether the `docDescription` is invalid (an empty string) and use `errMsg` to display the error on the screen if `docDescription` is empty.

Then, within the `createDoc` function, we set the value of `invalid` to `false`. We do this to ensure that, by default, `errMsg` is not displayed.

Then, we check if `docDescription` is not an empty string. If so, we emit the `create-doc` event, clear the value of the description `input` field, and then skip the rest of the code within the function by using the `return` statement.

```

if (docDescription.value !== "") {
  emit("create-doc", docDescription.value, docDate.value);
  docDescription.value = "";
  return;
}

```

If `docDescription` is empty, we set the value of `invalid` to `true` and specify the error message (`errMsg`).

```

invalid.value = true;
errMsg.value = "Document description not provided";

```

Then, for this to work within the template, we can change the border of the `input` area to `red` if there's an error. We can do this by adding the `input-err` class to the `div` as follows:

```

:class="{ 'input-err': invalid }"

```

Essentially, we are using class binding (`:class`) to indicate that if the value of `invalid` is `true`, then we can also add the `input-err` CSS class to the `div`.

On the other hand, if the value of `invalid` is `false`, then the user has provided the document description; therefore, the `input-err` CSS class is not added to the `div`.

Finally, if the user has not entered the document description—when `invalid` is `true`—the error message (`errMsg`) is displayed as follows:

```
<p v-show="invalid" class="err-msg"> {{ errMsg }}</p>
```

This works because if the value of `invalid` is `true`, then the paragraph (`p`) tag is shown, which is achieved by using the `v-show` directive. In Vue, double braces are the syntax for text interpolation.

On the other hand, if the value of `invalid` is `false`, then the paragraph (`p`) tag is not shown. In other words, using the `v-show` directive, Vue can toggle whether the paragraph (`p`) tag is visible (or not) without re-creating it every time within the DOM.

Wiring up the DocInput component

So, with the `DocInput` component ready, let's add it and use it within the `DocsPage` component. We can do this as follows (changes are highlighted in bold):

Code Listing 3-i: Wiring Up The Document Component (DocsPage.vue)

```
<script setup>
import { ref } from "vue";
import { uid } from "uid";
import DocInput from '../components/DocInput.vue';

const docsList = ref([]);

const createDoc = (description, expiration) => {
  docsList.value.push({
    id: uid(),
    description,
    expiration,
    isExpired: null,
    isEditing: null
  })
}
</script>

<template>
  <main>
    <DocInput @create-doc="createDoc" />
  </main>
</template>

<style lang="scss" scoped>
```

```

main {
  h1 {
    text-align: center;
    margin-bottom: 18px;
  }

  margin: 0 auto;
  width: 100%;
  max-width: 500px;
  display: flex;
  flex-direction: column;
  padding: 30px 15px;

  .docs-list {
    display: flex;
    flex-direction: column;
    gap: 18px;
    list-style: none;
    margin-top: 25px;
  }

  .docs-msg {
    align-items: center;
    justify-content: center;
    display: flex;
    gap: 10px;
    margin-top: 25px;
  }
}
</style>

```

Let's go over these changes. The first thing we did was import the required modules and components. We first imported `ref` from `Vue` and then `uid` from the `uid` NPM package we previously installed.

```

import { ref } from "vue";
import { uid } from "uid";

```

We then imported the `DocInput` component from the `components` folder.

```

import DocInput from '../components/DocInput.vue';

```

After that, we declared `docsList` as a reactive empty array by using `ref`, which we will use to keep the list of documents entered by users.

```
const docsList = ref([]);
```

Then, we created the `createDoc` function, with the document's **description** and **expiration** as parameters.

The function adds the newly created document to the list of existing documents (**docsList**) by creating an object with the document's details.

The object that represents the new document has an **id**, **description**, **expiration** date, and two additional properties: **isExpired** and **isEditing**.

```
const createDoc = (description, expiration) => {  
  docsList.value.push({  
    id: uid(),  
    description,  
    expiration,  
    isExpired: null,  
    isEditing: null  
  })  
}
```

We use the `uid()` function from the `uid` package we previously installed to get the document's unique **id**, whereas the **description** and **expiration** value are emitted from the **DocInput** component.

The **isExpired** and **isEditing** properties are helpful to know whether the document has expired or is being edited.

Finally, we add and use the **DocInput** component within the markup as follows:

```
<DocInput @create-doc="createDoc" />
```

Because this component emits the **create-doc** event to its parent component (**DocsPage**), we bind the `createDoc` function to it.

If we save the changes and run the app using the `npm run dev` command from the built-in terminal within VS Code, and then click **Add** without entering a description, we should see the following behavior:

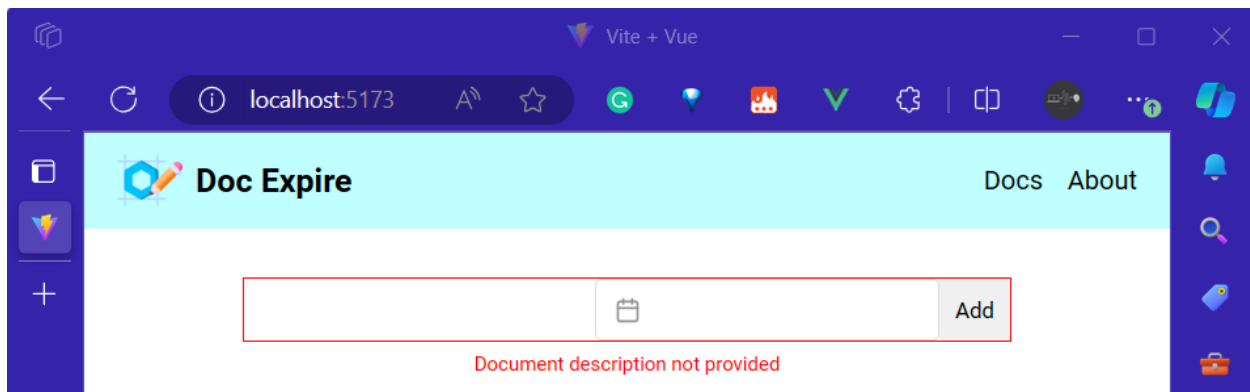


Figure 3-j: The Description Validation

By adding this code, we can ensure that a description is entered before emitting the event to the parent component.

Creating the Document component

Now that we have finalized the **DocInput** component, it's time to focus on displaying the document the user entered. To do that, we need to create a new Vue component within our application called **Document**, which we can place in the **Document.vue** file within the **components** folder.

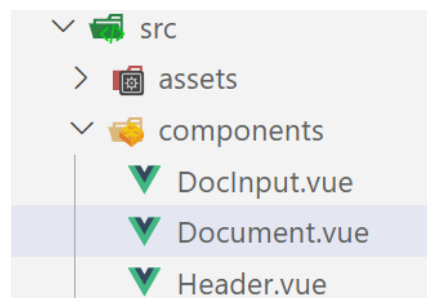


Figure 3-k: The New Document.vue File

With the file created, let's open it and add the following boilerplate code:

Code Listing 3-j: The Initial Document.vue File Boilerplate Code

```
<script setup>
</script>

<template>
  <div>
  </div>
</template>
```

```
<style lang="scss" scoped>
</style>
```

With this boilerplate created, and before adding any logic to the **Document** component, let's go back to the **DocsPage** component and wire this up.

Wiring up the Document component

To integrate and use the recently created **Document** component, we first need to import it, and then we need to create a list and use the **v-for** directive to iterate through any documents that users entered.

So, let's see how we can do that (the changes are highlighted in bold).

Code Listing 3-k: Wiring Up The Document Component (DocsPage.vue)

```
<script setup>
import { ref } from "vue";
import { uid } from "uid";
import DocInput from '../components/DocInput.vue';
import Document from '../components/Document.vue';

const docsList = ref([]);

const createDoc = (description, expiration) => {
  docsList.value.push({
    id: uid(),
    description,
    expiration,
    isExpired: null,
    isEditing: null
  })
}
</script>

<template>
  <main>
    <DocInput @create-doc="createDoc" />
    <ul>
      <Document :doc="doc" v-for="doc in docsList" :key="doc.id" />
    </ul>
  </main>
</template>
```

```

</template>

<style lang="scss" scoped>
main {
  h1 {
    text-align: center;
    margin-bottom: 18px;
  }

  margin: 0 auto;
  width: 100%;
  max-width: 500px;
  display: flex;
  flex-direction: column;
  padding: 30px 15px;

  .docs-list {
    display: flex;
    flex-direction: column;
    gap: 18px;
    list-style: none;
    margin-top: 25px;
  }

  .docs-msg {
    align-items: center;
    justify-content: center;
    display: flex;
    gap: 10px;
    margin-top: 25px;
  }
}
</style>

```

As you can see, the wiring process is straightforward. First, we imported the recently created **Document** component from the **components** folder.

```
import Document from '../components/Document.vue';
```

Then, we create a list of elements (**ul**) and add the **Document** component as follows:

```
<Document :doc="doc" v-for="doc in docsList" :key="doc.id" />
```

We use the **v-for** directive to loop through the list of existing documents (**docsList**) and render each document (**doc**).

Each document will be rendered with the values of the properties of the **doc** object, which binds to the **doc** property of the **Document** component. Notice that each document generated must have a unique **key**—the value of **doc.id**.

We'll create the **doc** property when we continue to work on the **Document** component, which is what we'll do next.

Continuing the Document component

Now that we have things wired up, let's add the logic, markup, and styling for the **Document** component.

Code Listing 3-1: The Document Component (Document.vue)

```
<script setup>

const props = defineProps({
  doc: {
    type: Object,
    default: () => {},
  },
  index: {
    type: Number,
    default: 0,
  },
});

defineEmits(["edit-doc", "update-doc", "delete-doc"]);
</script>

<template>
  <li>
    <div class="doc">
      <input
        v-if="doc.isEditing"
        type="text"
        :value="doc.description"
        @input="$emit('update-doc', $event.target.value, index)"
      />
      <span
        v-else
        :class="{
          'expired-doc': doc.isExpired,
        }"
      >
```

```

    >
    {{ doc.description +
      (doc.expiration !== undefined ?
        (": " + doc.expiration)
        :
        "")
      }}
  </span>
</div>
<div class="doc-actions">
  <button
    v-if="doc.isEditing"
    @click="$emit('edit-doc', index)"
  >
    Save
  </button>
  <button
    v-else
    @click="$emit('edit-doc', index)"
  >
    Edit
  </button>
  <button
    @click="$emit('delete-doc', doc.id)"
  >
    Delete
  </button>
</div>
</li>
</template>

<style lang="scss" scoped>
li {
  background-color: #bfffffff;
  display: flex;
  align-items: center;
  gap: 15px;
  padding: 20px 15px;

  &:hover {
    .doc-actions {
      opacity: 1;
    }
  }
}

```

```

}

.doc {
  flex: 1;

  .expired-doc {
    text-decoration: line-through;
  }

  input[type="text"] {
    border: 2px solid #edfaff;
    width: 100%;
    padding: 3px 8px;
  }
}

.doc-actions {
  transition: 105ms ease-in-out;
  display: flex;
  gap: 3px;
  opacity: 0;

  .icon {
    cursor: pointer;
  }
}
}
</style>

```

Let's go over this component to understand what it does. We define the component's properties (**props**) by using the **defineProps** Vue macro.

```

const props = defineProps({
  doc: {
    type: Object,
    default: () => {},
  },
  index: {
    type: Number,
    default: 0,
  },
});

```

By doing this, we indicate that the **Document** component will have two properties: one is the **doc** property, and the other is the **index** property. The **doc** property is an **Object** containing all the data a document can include, such as the **description** and **expiration** date. By default, it is an empty object. On the other hand, the **index** property is a **Number**, indicating the current document being updated or deleted from the list of documents, with a default value of zero.

Following that, we use the **defineEmits** Vue macro to indicate which events the **Document** component will emit to its parent component, the **DocsPage** component.

```
defineEmits(["edit-doc", "update-doc", "delete-doc"]);
```

Within the markup, we define a **li** element rendered within an **ul** element (included within the **DocsPage** parent component).

```
<li>
  <div class="doc">
    ...
  </div>
  <div class="doc-actions">
    ...
  </div>
</li>
```

Within the **li** element, we have one **div**, which we style using the **doc** CSS class. This contains the markup that renders a document item with the document's **description** and **expiration** date.

Figure 3-1 shows what this **div** would look like (within the **orange** rectangle—note that the **orange** rectangle is not part of the component's styling).

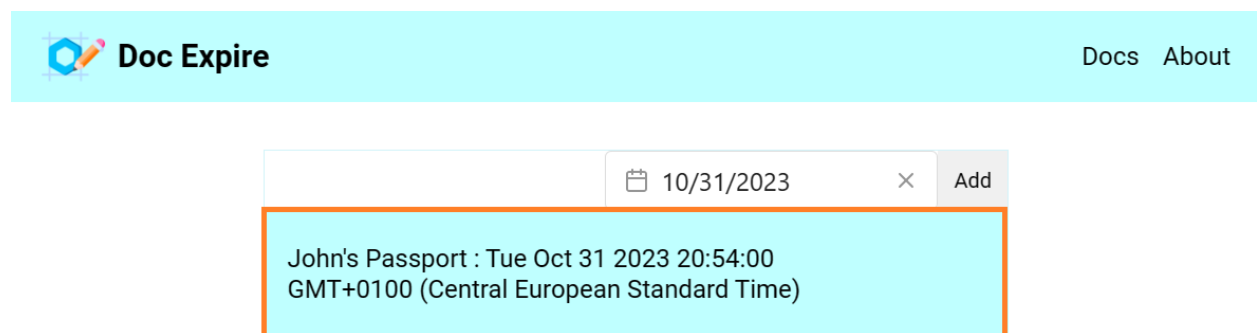


Figure 3-1: The **div** with the **doc** CSS Class (*Document.vue*)

Furthermore, we also have another **div**, which we style by using the **doc-actions** CSS class. Due to the styling that we are using, this **div** is only visible once we hover over the **div** with the **doc** CSS class.

The following is an example of what that looks like (highlighted in **red**—note that the **red** rectangle is not part of the component's styling):

John's Passport : Tue Oct 31 2023 20:54:00
GMT+0100 (Central European Standard Time)

Figure 3-m: The div with the doc-actions CSS Class (Document.vue)

Now, let's explore each **div**. Within the **div** with the **doc** CSS class, we have an **input** field that can be used to modify the document's **description**, and we also have a **span** that is used for displaying the document's **description** and **expiration** date.

The **input** field only appears when the document's **isEditing** property is **true**, which is possible by using the **v-if** directive.

```
<input
  v-if="doc.isEditing"
  type="text"
  :value="doc.description"
  @input="$emit('update-doc', $event.target.value, index)"
/>
```

Furthermore, the input field's **value** binds to the document's **description**. After the user has entered the input (**@input**), the **update-doc** event is emitted to the parent component (**DocsPage**) with the input value (**\$event.target.value**) as a parameter and the document's **index** being updated.

This means updating the document won't occur within the **Document** component, but within the parent component (**DocsPage**) instead, as we'll see later.

On the other hand, the **span** element is only shown when the document is not being edited (when the value of **doc.isEditing** is **false**), which is why the **v-else** directive is used.

```
<span
  v-else
  :class="{ 'expired-doc': doc.isExpired, }">
  {{ doc.description +
    (doc.expiration != undefined ?
      (": " + doc.expiration) : "")
  }}
</span>
```

If the document's **isExpired** property is **true**, we can bind the **span** element to the **expired-doc** CSS class.

The expression within the double braces `{{ }}` is displayed on the UI. The document's description (`doc.description`) is always shown, whereas the expiration date is only displayed if its value is not `undefined`.

Notice also that to avoid overcomplicating the app's functionality when editing a document, I've explicitly excluded the possibility of updating the document's expiration date. So, when editing a document, we can only edit the description.

If you want to change the value of the expiration date for any document, you will have to delete the document and add it again. The reason is that allowing the user to edit the expiration date adds an extra layer of complexity to the application, which doesn't add any intrinsic value to the core aspects of Vue 3 that we are learning. However, once you have finished this book and want to take on that challenge, it's a great way to enhance the app, which I would encourage you to do.

Next, within the `div` with the `doc-actions` CSS class, we have three buttons, but only two are visible simultaneously: **Edit** and **Delete**, or **Save** and **Delete**.

```
<button
  v-if="doc.isEditing"
  @click="$emit('edit-doc', index)"
>
  Save
</button>

<button
  v-else
  @click="$emit('edit-doc', index)"
>
  Edit
</button>

<button
  @click="$emit('delete-doc', doc.id)"
>
  Delete
</button>
```

Only two buttons are visible simultaneously because, for the **Save** and **Edit** buttons, we are using `v-if` and `v-else` directives, respectively. So, when the current document's `isEditing` property is `true`, we edit that document so that the **Save** button will be shown.

On the other hand, when the current document's `isEditing` property is `false`, the document is not being edited, so the **Edit** button is shown. In either case, the **Delete** button is always displayed.

As you can see, each button emits an event to the parent component (`DocsPage`) when a click occurs. Both the **Save** and **Edit** buttons emit the same event, `edit-doc`, passing the document's current `index` (position) within the list of documents. The **Delete** button emits the `delete-doc` event, passing the current document's ID (`doc.id`) as a parameter.

As we'll see next, the **DocsPage** component is responsible for implementing and running those edit or delete actions.

Document actions

To edit, update, or delete a document, we must add some additional logic to the **DocsPage.vue** file. The changes are highlighted in bold in the following code:

Code Listing 3-m: The Updated DocsPage.vue File

```
<script setup>
import { ref } from "vue";
import { uid } from "uid";
import DocInput from '../components/DocInput.vue';
import Document from '../components/Document.vue';

const docsList = ref([]);

const createDoc = (description, expiration) => {
  docsList.value.push({
    id: uid(),
    description,
    expiration,
    isExpired: null,
    isEditing: null
  })
}

const toggleEdit = (docIdx) => {
  docsList.value[docIdx].isEditing = !docsList.value[docIdx].isEditing;
}

const updateDoc = (d, docIdx) => {
  docsList.value[docIdx].description = d;
}

const deleteDoc = (id) => {
  docsList.value = docsList.value.filter(
    (doc) => doc.id !== id
  );
}
</script>

<template>
```

```

<main>
  <DocInput @create-doc="createDoc" />
  <ul>
    <Document
      :doc="doc"
      v-for="(doc, index) in docsList"
      :key="doc.id"
      :index="index"
      @edit-doc="toggleEdit"
      @update-doc="updateDoc"
      @delete-doc="deleteDoc"
    />
  </ul>
</main>
</template>

<style lang="scss" scoped>
main {
  h1 {
    text-align: center;
    margin-bottom: 18px;
  }

  margin: 0 auto;
  width: 100%;
  max-width: 500px;
  display: flex;
  flex-direction: column;
  padding: 30px 15px;

  .docs-list {
    display: flex;
    flex-direction: column;
    gap: 18px;
    list-style: none;
    margin-top: 25px;
  }

  .docs-msg {
    align-items: center;
    justify-content: center;
    display: flex;
    gap: 10px;
    margin-top: 25px;
  }
}

```

```
    }  
  }  
</style>
```

The first thing to notice is that three functions were added that we didn't have previously: **toggleEdit**, **updateDoc**, and **deleteDoc**.

These functions bind to the **edit-doc**, **update-doc**, and **delete-doc** events (emitted by the **Document** component), respectively.

```
@edit-doc="toggleEdit"  
@update-doc="updateDoc"  
@delete-doc="deleteDoc"
```

The **toggleEdit** function, as its name implies, is responsible for allowing a document from the list (the current document) to be in edit mode or not.

```
const toggleEdit = (docIdx) => {  
  docsList.value[docIdx].isEditing = !docsList.value[docIdx].isEditing;  
}
```

The current document is obtained by using its position (**docIdx**) within the list of documents: **docsList.value[docIdx]**. The **edit-doc** event emits the position value as **index**, which binds to the **index** as follows:

```
:index="index"
```

So, all the **toggleEdit** function does is change the value of the current document's **isEditing** property to **true** when **false**, and vice versa.

The **updateDoc** function updates a document's **description** once the user clicks the **Save** button. This is possible because the **Document** component emits the **update-doc** event, passing the updated document's **description** (**d**) and **index** (**docIdx**) with the list of documents as parameters.

```
const updateDoc = (d, docIdx) => {  
  docsList.value[docIdx].description = d;  
}
```

The document's current description (**docsList.value[docIdx].description**) gets updated by simply assigning it the value of the edited description (**d**) emitted by the **Document** component.

Finally, the **deleteDoc** function runs when the **Document** component emits the **delete-doc** event, to which the document's **id** is passed as a parameter.

To delete the current document from the list of documents, we use JavaScript's array **filter** method. To the **filter** method, we pass as a parameter a lambda function that checks whether the document's current ID (**doc.id**) has the same value as the **id** of the document about to be deleted from the list.

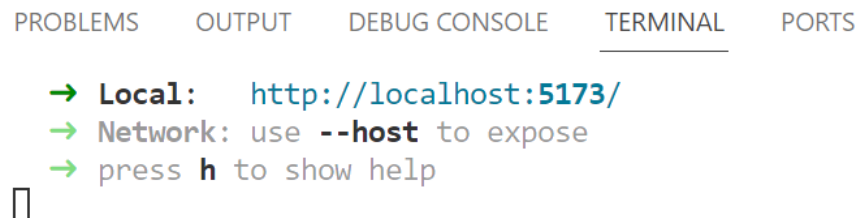
```
const deleteDoc = (id) => {  
  docsList.value = docsList.value.filter(  
    (doc) => doc.id !== id  
  );  
}
```

So, in essence, the **filter** method excludes (removes) the document from the list and leaves the other documents as they are.

Running the app

Well done for following along! If you have saved all the changes to the various project files we have been working with, it's time to run the app and give it a whirl. If not, please save any changes to any open files within VS Code.

Once you have saved the changes, run the **npm run dev** command from the built-in terminal within VS Code. Then, open the browser and navigate to the **localhost** URL highlighted in the built-in terminal.



The image shows a screenshot of the VS Code interface with the 'TERMINAL' tab selected. The terminal displays the following text:
→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h to show help
Below the text is a small icon of a document with a checkmark.

Figure 3-n: The App's localhost URL (VS Code Built-in Terminal)

With the application running, enter three test documents like the following:



Add

Ed's Passport: Thu Nov 28 2030 23:48:00
GMT+0100 (Central European Standard Time)

John's Drivers License: Thu Feb 05 2026 23:48:00
GMT+0100 (Central European Standard Time)

Dummy document

Figure 3-o: The App Running with Recent Changes

Notice that the first two documents I entered have an expiration date; however, the third document doesn't. So, let's hover over the second document and we'll see the **Edit** button appear.

Ed's Passport: Thu Nov 28 2030 23:48:00
GMT+0100 (Central European Standard Time)

John's Drivers License: Thu Feb 05 2026 23:48:00
GMT+0100 (Central European Standard Time)

Edit Delete

Dummy document

Figure 3-p: The Edit and Delete Buttons Next to the Second Document)

After you click **Edit**, you should see the following:

Ed's Passport: Thu Nov 28 2030 23:48:00
GMT+0100 (Central European Standard Time)

John's Drivers License

Save Delete

Dummy document

Figure 3-q: Editing the Second Document

At this stage, you can change the document's description.

Let's change the document's description to something else, and then click **Save**. Then, we should see the following:



Figure 3-r: The Second Document Edited

As you can see, the second document's description has been correctly changed, and the expiration date remains the same, which is the expected functionality based on the code we wrote.

You can test this further by adding, editing, or deleting other documents; everything should work as expected. Remember, there's no data persistence, so any changes will be lost if you refresh the page.

Recap

We have explored some of the essential features of Vue 3 and now have a basic but functional working application. The only major limitation is that the app only keeps the data in memory.

For what remains of the book, we will mostly explore how to persist data, make the application work with a database, and touch upon other Vue 3 aspects and how they compare to Vue 2.

Next, before we dive into how to make the app work with a database, we will look at some of the critical differences between Vue 2 and 3.

Chapter 4 Vue 2 vs. Vue 3

Now that we know how to create the basics of a Vue 3 application, it's time to review (succinctly) some of the critical differences between Vue 2 and 3 before adding more complexity by using a database. So, let's explore these crucial differences between versions.

Composition API

Vue 2 relies on the Options API for defining components. On the other hand, Vue 3 introduced the Composition API, allowing for more flexibility and reusability in component logic. Let's have a look at a code example of each.

Code Listing 4-a: Using the Options API in Vue 2

```
<template>
  <button @click="increment">{{ count }}</button>
</template>

<script>
  export default {
    data() {
      return {
        count: 0,
      };
    },
    methods: {
      increment() {
        this.count++;
      },
    },
  };
</script>
```

In the previous example, we can see that Vue 2 primarily uses the Options API, which separates a component's logic into various options (e.g., **data**, **methods**, **computed**).

Contrary to that, in Vue 3, the Composition API was introduced, which allows you to organize logic based on functions and reuse code more efficiently.

The introduction of the **script setup** syntax is one of the most notable aspects of Vue 3. Here is an example:

Code Listing 4-b: Using the Composition API in Vue 3

```
<template>
  <div>{{ message }}</div>
</template>

<script setup>
  let message = 'Hello, Vue 3!';
</script>
```

Vue 3's Composition API with **script setup** simplifies the structure of your component, making it more concise and easier to understand.

Furthermore, the Composition API offers improved code organization and encourages a more modular approach to building components. It's especially beneficial for complex components where logic can be grouped logically, making code easier to understand and maintain.

Props declaration

Vue 2 uses the **props** option to declare props and their types, whereas Vue 3, within the **script setup**, uses the **defineProps** macro, as we saw in the previous chapter. So, let's look at a code example to understand this better.

Code Listing 4-c: Using Props in Vue 2

```
<script>
export default {
  props: {
    propA: String,
    propB: Number
  }
};
</script>
```

On the other hand, in Vue 3, prop declarations are more explicit, and you can use ES6 features like destructuring.

Code Listing 4-d: Using Props in Vue 3

```
<script setup>
import { defineProps } from 'vue';
```

```
const props = defineProps({
  propA: String,
  propB: Number
});
</script>
```

Using **defineProps** in Vue 3 offers better type checking and makes it easier to see which props a component accepts. It also allows for destructuring, making it more concise and clear.

Emits declaration

Vue 2 uses the **emits** option to declare custom events, whereas with Vue 3, we can use the **defineEmits** macro within the **script setup**. Let's look at the following Vue 2 code example:

Code Listing 4-e: Using Emits in Vue 2

```
<script>
export default {
  methods: {
    handleClick() {
      this.$emit('custom-event', data);
    }
  }
};
</script>
```

However, using the Composition API in Vue 3 allows us to separate concerns within the app, making it easier to see which props and events a component uses.

Code Listing 4-f: Using Emits in Vue 3

```
<script setup>
import { defineEmits } from 'vue';

const emit = defineEmits();
</script>
```

Defining custom events using **defineEmits** in Vue 3 provides improved visibility of which events a component can emit. It also simplifies event handling, making it cleaner and more organized.

Reactive data

Vue 2 uses **data** and **computed** properties for reactive data, whereas Vue 3, within the **script setup**, uses **ref**, **reactive**, and **computed**. Let's have a look.

Code Listing 4-g: Reactive Data in Vue 2

```
<template>
  <div>{{ message }}</div>
</template>

<script>
  export default {
    data() {
      return {
        message: 'Hello, Vue 2!'
      };
    }
  };
</script>
```

Vue 3's **ref**, **reactive**, and **computed** provide more explicit reactivity, making it easier to work with reactive data.

Code Listing 4-h: Reactive Data in Vue 3

```
<template>
  <div>{{ message }}</div>
</template>

<script setup>
  import { ref } from 'vue';

  const message = ref('Hello, Vue 3!');
</script>
```

Lifecycle hooks

Vue 2 uses options like **created** and **mounted** for lifecycle hooks. On the other hand, Vue 3, in **script setup** mode, uses the **onMounted** and **onUnmounted** functions. Let's have a look.

Code Listing 4-i: Lifecycle Hooks in Vue 2

```
<script>
```

```
export default {
  created() {
    // Logic on component creation
  }
};
</script>
```

In Vue 3, the lifecycle hooks are more explicit and can be organized within the component, as they are defined within the **script setup**, improving code readability and maintainability.

Code Listing 4-j: Lifecycle Hooks in Vue 3

```
<script setup>
import { onMounted } from 'vue';

onMounted(() => {
  // Logic on component creation
});
</script>
```

Directives

Although the directives, such as **v-for** and **v-model**, are the same between both versions, Vue 3 allows you to define reactive properties more explicitly, making your code more readable and maintainable. Let's have a look.

Code Listing 4-k: Directives in Vue 2

```
<template>
  <div v-for="item in items">{{ item }}</div>
  <input v-model="message">
</template>

<script>
export default {
  data() {
    return {
      items: [1, 2, 3],
      message: ''
    };
  }
};
</script>
```

Code Listing 4-l: Directives in Vue 3

```
<template>
  <div v-for="item in items">{{ item }}</div>
  <input v-model="message">
</template>

<script setup>
  import { ref } from 'vue';

  const items = ref([1, 2, 3]);
  const message = ref('');
</script>
```

Custom event handling

Vue 2 uses `$emit` for custom event handling, whereas Vue 3 uses the `emit` function within the `script setup`. Let's have a look.

Code Listing 4-m: Custom Event Handling in Vue 2

```
<template>
  <button @click="handleClick">Click me</button>
</template>

<script>
  export default {
    methods: {
      handleClick() {
        this.$emit('custom-event', data);
      }
    }
  };
</script>
```

Using `emit` in Vue 3 is more straightforward and clarifies concerns between emitting custom events and handling them, improving code organization.

Code Listing 4-n: Custom Event Handling in Vue 3

```
<template>
  <button @click="handleClick">Click me</button>
```

```

</template>

<script setup>
  import { defineEmits } from 'vue';

  const emit = defineEmits();

  function handleClick() {
    emit('custom-event', data);
  }
</script>

```

Scoped slots

Vue 2 uses **slot-scope** for scoped slots; however, Vue 3 uses the **slot** property within the **script setup**. Let's have a look.

Code Listing 4-o: Scoped Slots in Vue 2

```

<template>
  <slot-scope="props">
    <div>{{ props.data }}</div>
  </slot>
</template>

```

Vue 3's approach simplifies scoped slots and integrates them more naturally with the template.

Code Listing 4-p: Scoped Slots in Vue 3

```

<template>
  <slot :data="slotProps.data"></slot>
</template>

<script setup>
  const slotProps = {
    data: 'Some data'
  };
</script>

```

Vue 3 simplifies scoped slots, making them more intuitive and natural to use within the template. The **slot** property provides a cleaner and more concise way to work with slot content.

Provide and inject

Although both Vue 2 and 3 use **provide** and **inject** for component communication, their implementations differ. Let's have a look.

Code Listing 4-q: Provide and Inject in Vue 2

```
<script>
export default {
  provide: {
    user: 'John Diego'
  }
};
</script>
```

Vue 3's **provide** and **inject** functions offer a cleaner and more expressive way to share data between components. They improve the transparency of component dependencies and promote better code structure.

Code Listing 4-r: Provide and Inject in Vue 3

```
<script setup>
  provide('user', 'John Diego');
</script>
```

Teleport

When developing an app, it might be possible that a part of a component's template belongs to it logically; however, from a UI (DOM) point of view, it should be displayed elsewhere.

For that purpose, Vue 3 introduced the [teleport component](#) that allows us to "teleport" (as in [Star Trek](#)) a part of a component's template into a DOM node outside that component's DOM hierarchy.

Code Listing 4-s: Simple Teleport Example in Vue 3

```
<template>
  <teleport to="body">
    <div>Teleported content</div>
  </teleport>
</template>
```

In this example, the **div** contained within the **teleport** component is "teleported" into the **body** section of the DOM.

Introducing the **teleport** component in Vue 3 simplifies creating portal-like functionality, which was previously quite difficult to achieve in Vue 2. It allows you to render content outside of the component's parent element, offering greater flexibility in UI design.

Recap

Vue 3 with **script setup** offers several advantages over Vue 2, including improved code organization, reactivity, and component structure.

It encourages cleaner and more maintainable code, making it an excellent choice for both small and large-scale Vue applications.

The Composition API and **script setup** provide a modern and efficient development experience.

Using the Composition API with **script setup** is particularly beneficial when building large and complex applications, making your code more readable, maintainable, and efficient.

Chapter 5 Firebase Setup

If you've read some of my previous books, you've probably noticed that I use Firebase quite a bit. The reason is that [Firebase](#) is a great way to give a front-end application back-end functionality without writing back-end code. With Firebase, getting a back end up and running for any web app is easy, independently of the front-end framework used.

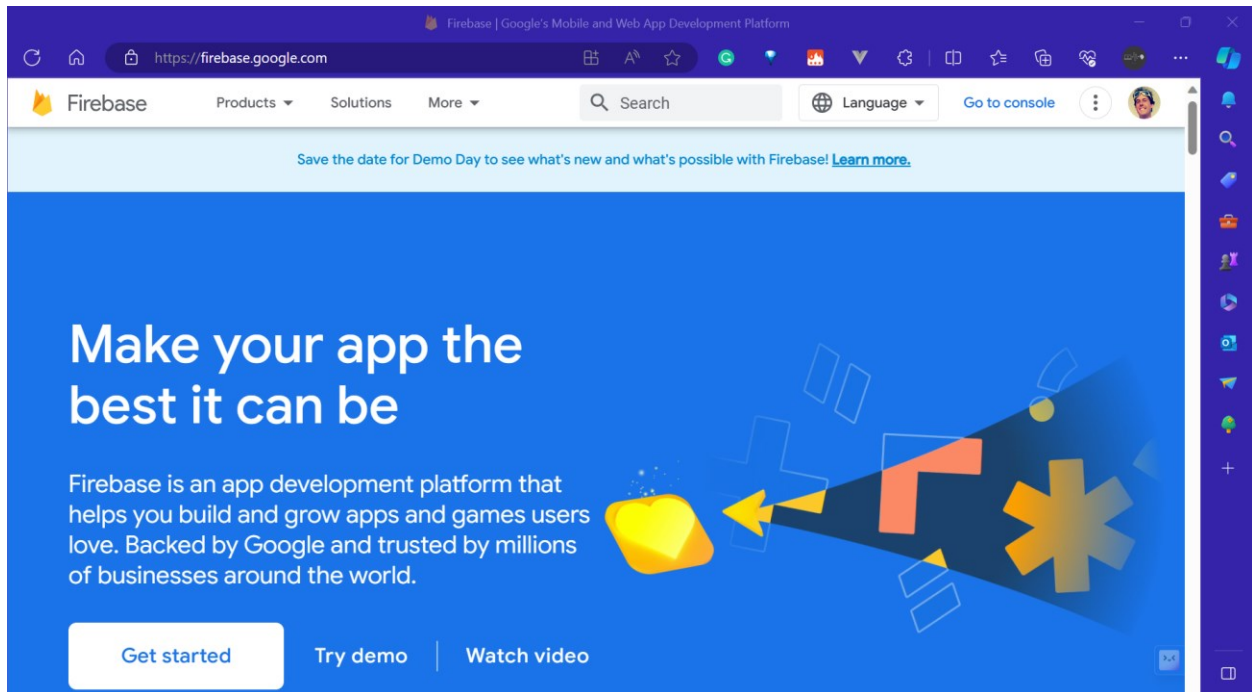


Figure 5-a: Firebase Home Page



Note: These webpages might change over time, and might not entirely match the screenshots shown; however, you should be able to continue easily with the steps provided.

Installing Firebase

Getting started with Firebase is straightforward, and you need to be signed up with a Google Workspace or Gmail account. On the Firebase home page, click **Get started** to go to the Firebase console.

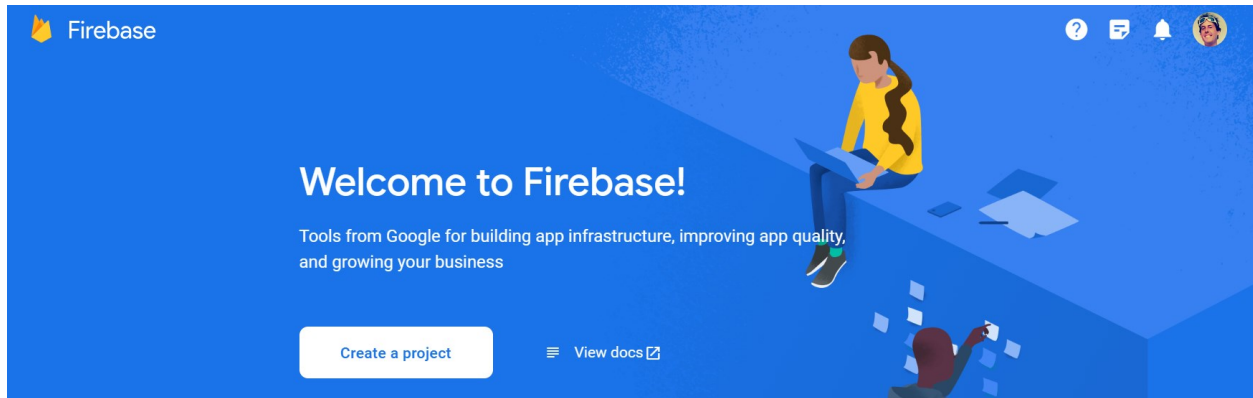


Figure 5-b: Firebase Console Home Page

Next, click **Create a project** and then provide a Firebase project name. I'll call the app **Vue3SuccinctlyApp**, but you can call it something else. Just make sure your app name is unique.

The image shows a dialog box titled "Create a project (Step 1 of 3)". The main text says "Let's start with a name for your project®". Below this, there is a label "Project name" and a text input field containing "Vue3SuccinctlyApp". Underneath the input field, there is a small preview of the name "vue3succinctlyapp" with a pencil icon. At the bottom, there is a blue button labeled "Continue".

Figure 5-c: Creating a Firebase Project (Step 1 of 3)

Once you have entered the project name, click **Continue**. Then, you will be asked about using Google Analytics for the app.

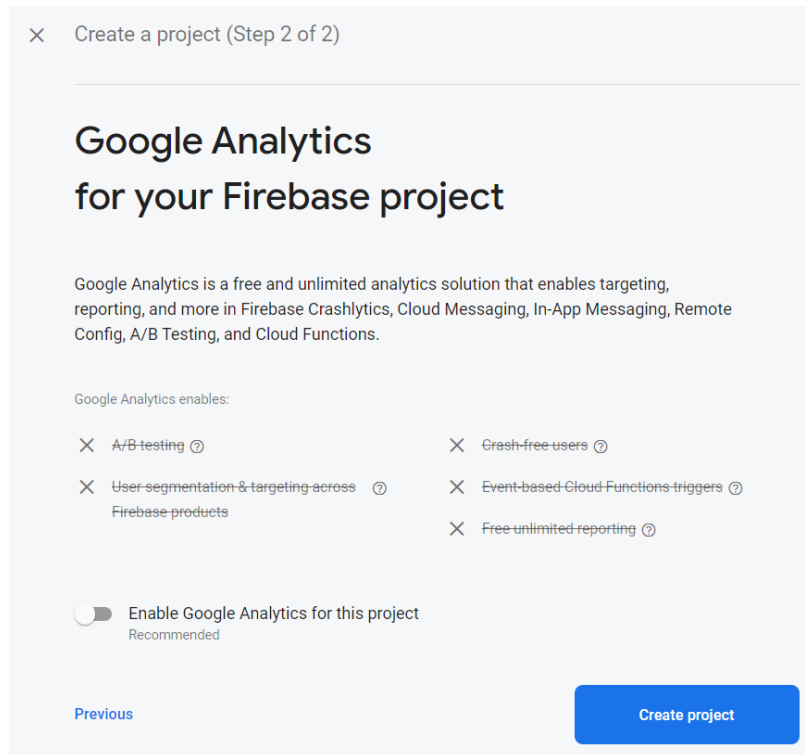


Figure 5-d: Creating a Firebase Project (Step 2 of 3)

Firebase enables **Google Analytics for your Firebase project** by default. However, we won't use it, so clear the **Enable Google Analytics for this project** option to turn it off.

Once you're done, click **Create project**. This will create the Firebase project. After the Firebase project has been provisioned, we can click **Continue**.

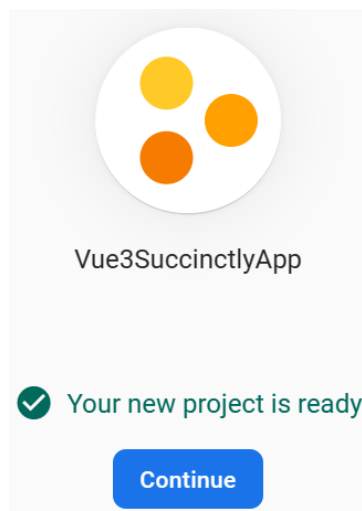


Figure 5-e: Creating a Firebase Project (Step 3 of 3)

Next, we'll be directed to the project's main page within the Firebase console.

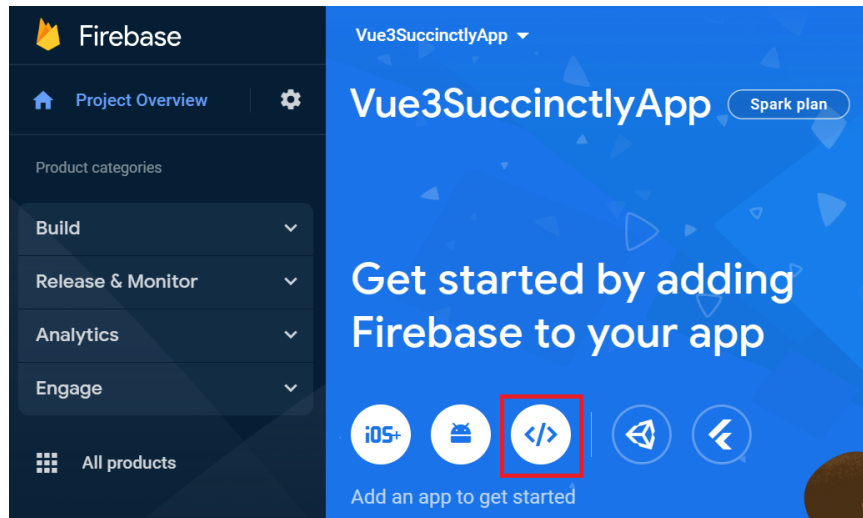


Figure 5-f: Firebase Main Project Page

Click the web icon (highlighted in **red** in the preceding image) to add a web app to the Firebase project. After doing that, we'll see the **Add Firebase to your web app** screen, where we can register our app by clicking **Register app**.

Figure 5-g: Add Firebase to Your Web App

Next are the steps to include the **Firebase SDK** in our Vue 3 project. Once we've done those steps, we can click **Continue to console**.

2 Add Firebase SDK

☒ Use npm ☐ Use a <script> tag

If you're already using [npm](#) and a module bundler such as [webpack](#) or [Rollup](#), you can run the following command to install the latest SDK ([Learn more](#)):

```
$ npm install firebase
```

Then, initialize Firebase and begin using the SDKs for the products you'd like to use.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
const firebaseConfig = {
  apiKey: "AIza[REDACTED]",
  authDomain: "vue3succinctlyapp.firebaseio.com",
  projectId: "vue3succinctlyapp",
  storageBucket: "vue3succinctlyapp.appspot.com",
  messagingSenderId: "[REDACTED]",
  appId: "[REDACTED]"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
```

Note: This option uses the [modular JavaScript SDK](#), which provides reduced SDK size.

Learn more about Firebase for web: [Get Started](#), [Web SDK API Reference](#), [Samples](#)

[Continue to console](#)

Figure 5-h: Add Firebase SDK



Note: I've hidden the `apiKey`, `messagingSenderId`, and `appId`, as these are specific to my environment and cannot be shared. For your environment, you'll have different values than mine.

Adding the Firebase SDK is a two-step process. The first step is to install Firebase in the project by running the `npm install firebase` command.

So, let's switch to VS Code and run the following command by using the built-in terminal:

Code Listing 5-a: Command to Install Firebase in Our Project

```
npm install firebase
```

After running this command, you'll see output similar to the following within the built-in terminal:

```
C:\PERSONAL-DATA\Projects\Books\Vue 3 Succinctly\my-vue3-project>npm install firebase
added 89 packages, and audited 142 packages in 55s

11 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figure 5-i: Firebase Installed (VS Code Built-in Terminal)

Next, let's initialize Firebase. Within our Vue 3 project's **src** folder, let's create a new file called **firebase.js**.

Then, copy the code shown in Figure 5-h and paste it into the **firebase.js** file. This is how it appears in VS Code in my environment:

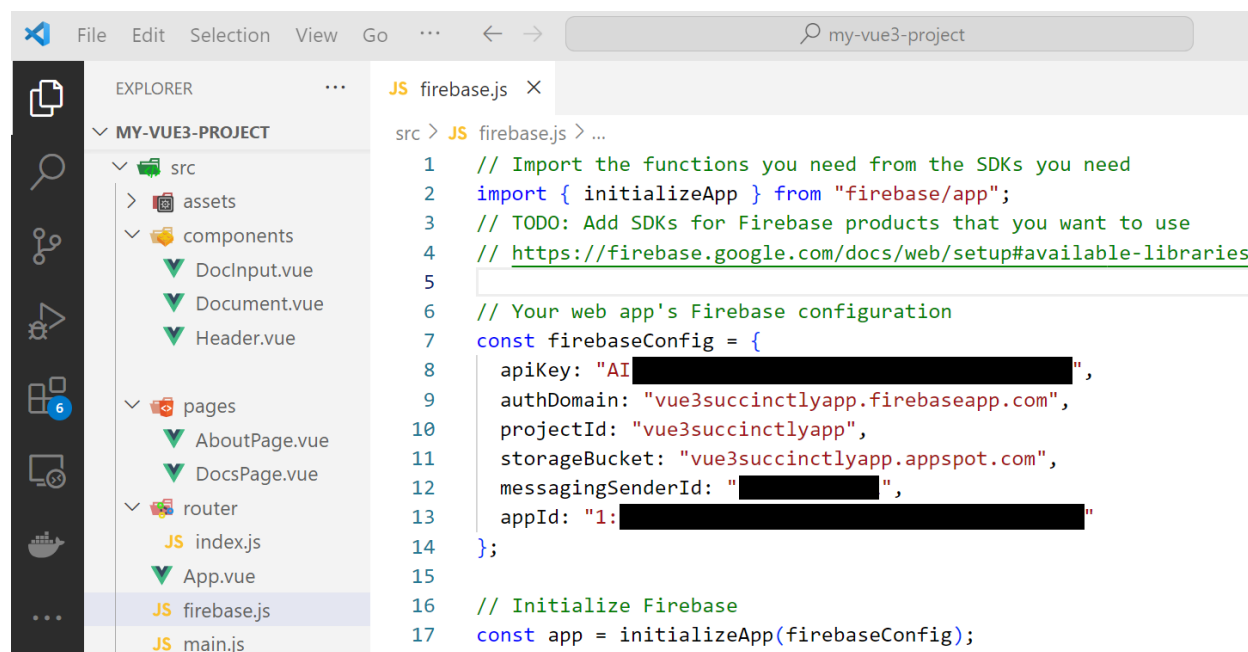


Figure 5-j: The firebase.js File (VS Code)

 **Note:** I've hidden the `apiKey`, `messagingSenderId`, and `appId` because they are specific to my environment and cannot be shared. For your environment, you'll have different values than mine.


Here is the **firebase.js** code.

Code Listing 5-b: The `firebase.js` Code

```
import { initializeApp } from "firebase/app";

const firebaseConfig = {
  apiKey: "A...",
  authDomain: "vue3succinctlyapp.firebaseio.com",
  projectId: "vue3succinctlyapp",
  storageBucket: "vue3succinctlyapp.appspot.com",
  messagingSenderId: "...",
  appId: "1:..."
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
```

 **Note:** I've deleted the `apiKey`, `messagingSenderId`, and `appId` values (highlighted in yellow), as these are specific to my environment and cannot be shared. For your environment, you'll have different values than mine.

With the Firebase SDK set up and the **firebase.js** file ready within VS Code, we can click **Continue to console** (see Figure 5-h). We should now see the following:

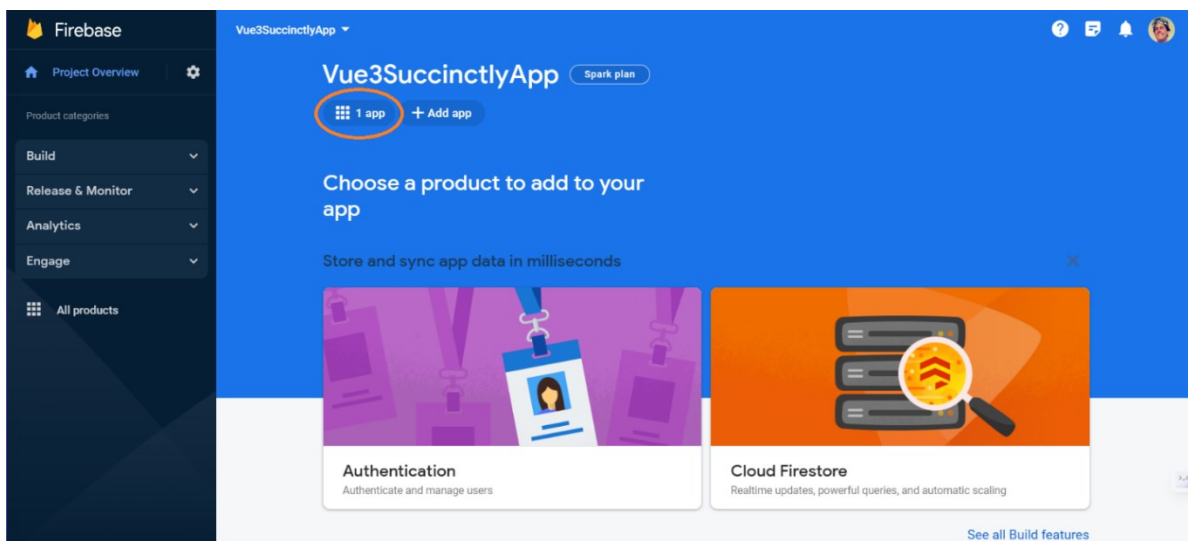


Figure 5-k: The App's Main Page in the Firebase Console

Firestore console (optional)

Beyond the Cloud Firestore database that Firebase provides (which is what we'll be using), Firebase also supports hosting, so therefore, for production, we can also choose to host the app itself with Firebase.

We must install the Firebase console by using the following command to use Firebase Hosting:

Code Listing 5-c: Command to Install the Firebase Console

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

Once the Firebase console has been installed, we can use the command **firebase login** to log on to Firebase. We can run the **firebase init** command from the project's root directory to initiate Firebase Hosting. We can run the **firebase deploy** command when ready to host the app.



Note: Run these commands individually, one after the other, to deploy the app to Firebase Hosting.

Code Listing 5-d: Commands to Deploy to Firebase Hosting

```
firebase login
firebase init
firebase deploy
```

Creating a datastore

From the app's main page within the Firebase console, click **Cloud Firestore** to create the database (or click **Firestore Database** under the **Build** menu option).

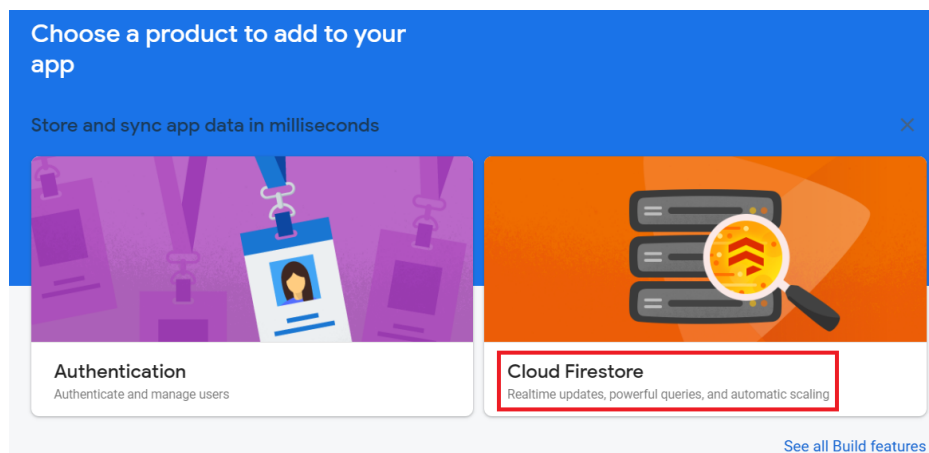


Figure 5-I: Products to Add to the App (Cloud Firestore Highlighted)

Once you're done, click **Create database**, as shown in the following figure:

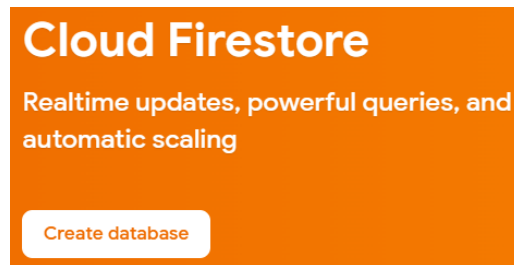


Figure 5-m: Cloud Firestore Create Database Option

After clicking **Create database**, we can choose the database's location, which I'll leave as is, and then click **Next**.

A blue dialog box titled "Create database" with a close button (X) in the top right corner. Below the title bar, there are two steps: "1 Set name and location" (active) and "2 Secure rules". The "Database ID" field contains the text "(default)". The "Location" dropdown menu is set to "nam5 (United States)". Below the dropdown, a small information icon and text state: "Your location setting is where your Cloud Firestore data will be stored". A yellow warning box contains an exclamation mark icon and the text: "After you set this location, you cannot change it later. Also, this location setting will be the location for your default Cloud Storage bucket." with a "Learn more" link and an external link icon. At the bottom, there is a note: "Enabling Cloud Firestore will prevent you from using Cloud Datastore with this project". On the right side of the bottom, there are "Cancel" and "Next" buttons.

Figure 5-n: Setting the Database Name and Location

After clicking **Next**, we can set the security rules for the database. By default, the **Start in production mode** option is selected. So, let's change that to **Start in test mode** instead.

Create database

✓ Set name and location

2 Secure rules

After you define your data structure, you will need to write rules to secure your data.
[Learn more](#)

☐ Start in **production mode**

Your data is private by default. Client read/write access will only be granted as specified by your security rules.

☒ Start in **test mode**

Your data is open by default to enable quick setup. However, you must update your security rules within 30 days to enable long-term client read/write access.

```

rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
        request.time < timestamp.date(2023, 12, 2);
    }
  }
}

```

The default security rules for test mode allow anyone with your database reference to view, edit and delete all data in your database for the next 30 days

Enabling Cloud Firestore will prevent you from using Cloud Datastore with this project

Cancel

Enable

Figure 5-o: Selecting a Mode for Starting the Database

Let's click **Enable** to create the database with the selected options. Once the database has been created, you should see the following:

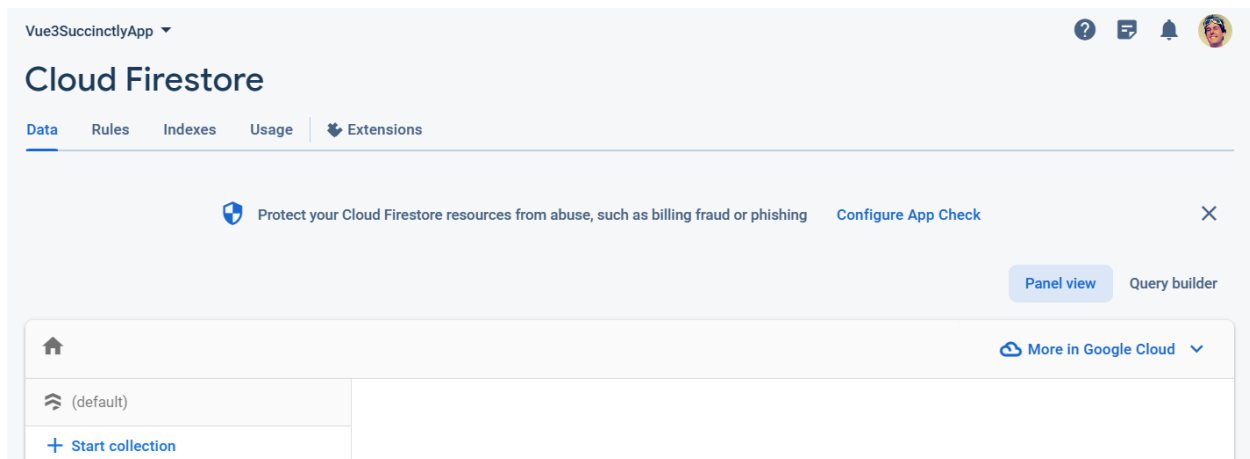
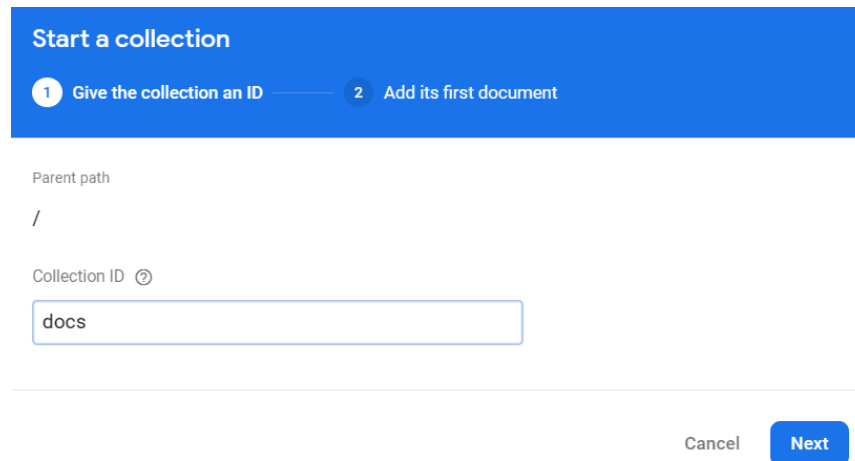


Figure 5-p: Cloud Firestore (Database Created)

Great! We now have the database ready. But before we add the remaining logic to our application to work with this database, let's create a collection with some test data.

To do that, click the **Start collection** option and then enter the **Collection ID**—let's call it **docs**.



Start a collection

1 Give the collection an ID — 2 Add its first document

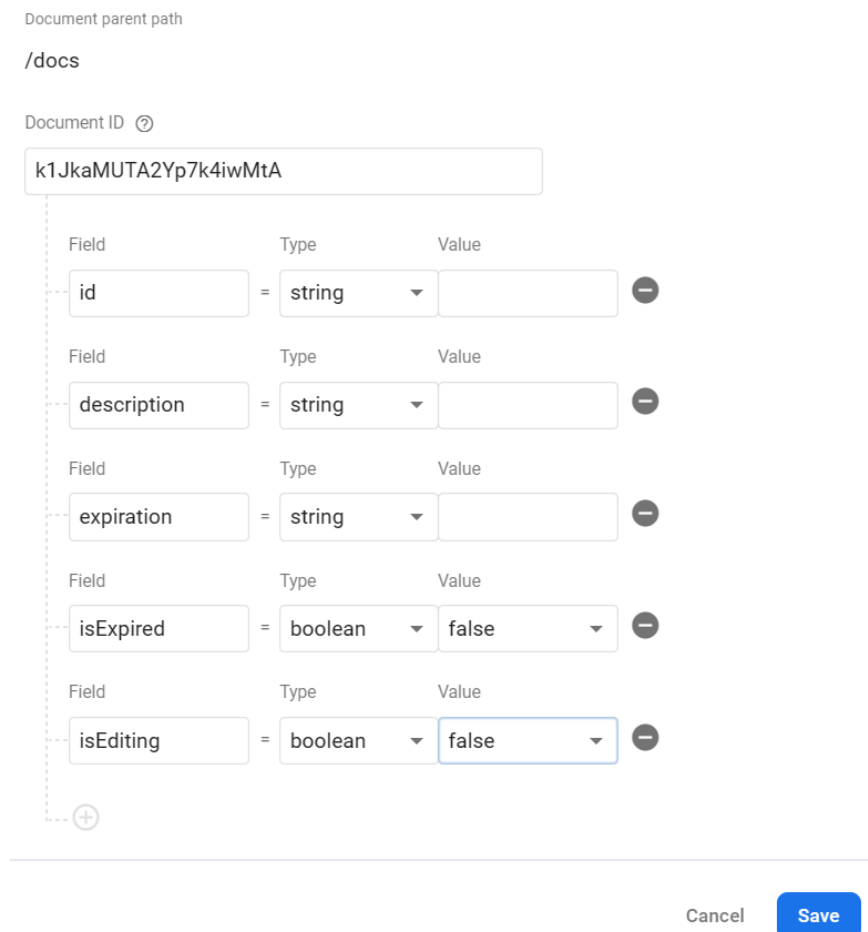
Parent path
/

Collection ID ?
docs

Cancel Next

Figure 5-q: Starting a Collection and Setting a Collection ID

Then, click **Next**. We can indicate the document fields as follows:



Document parent path
/docs

Document ID ?
k1JkaMUTA2Yp7k4iwMtA

Field	Type	Value
id	string	
description	string	
expiration	string	
isExpired	boolean	false
isEditing	boolean	false

Cancel Save

Figure 5-r: Document Fields (Cloud Firestore Database)

These are the same ones we previously specified in the `createDoc` function of the `DocsPage.vue` file.

To add the test document with these fields to the Cloud Firestore database, click **Save**. After doing that, we should see the following in the Firebase console:

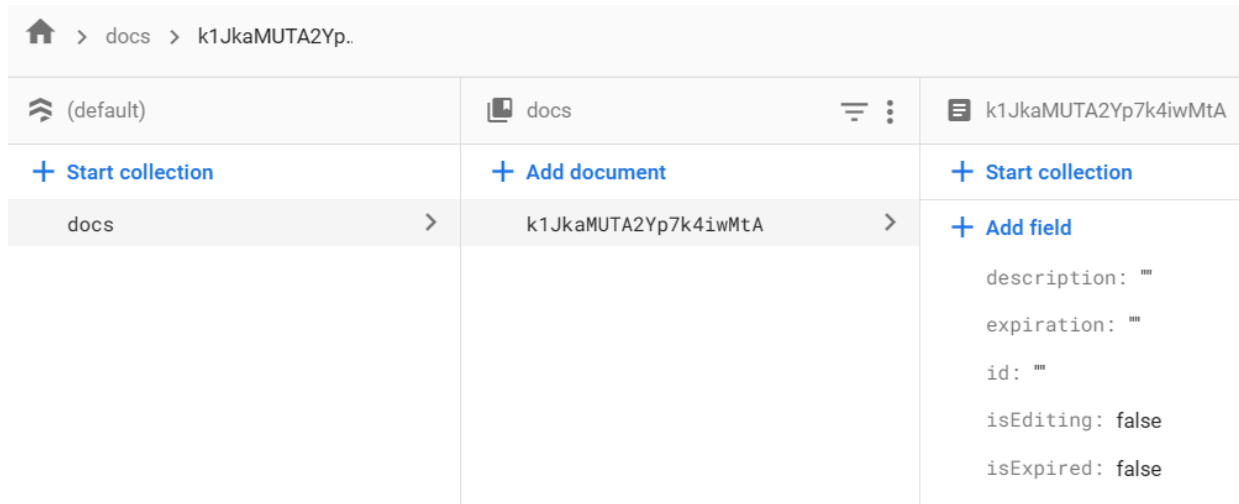


Figure 5-s: Test Document Added to Cloud Firestore

Recap

We now have a Cloud Firestore database ready. Throughout this chapter, we've seen how straightforward it is to set up Firebase.

We still need to add logic to our application to retrieve and save data to Firebase. That's what we'll do next.

Chapter 6 Adding Firebase

Our simple demo application works great in the browser, but cannot store information. Now that we have set up Firebase, we are ready to add the logic for our application to work with Cloud Firestore.

Updating firebase.js

Before we add any Firebase logic to our application, we need to make a small change to the **firebase.js** file and export the database object. Let's have a look—the changes are highlighted in bold.

Code Listing 6-a: Updated firebase.js File

```
import { initializeApp } from "firebase/app";

const firebaseConfig = {
  apiKey: "A...",
  authDomain: "vue3succinctlyapp.firebaseio.com",
  projectId: "vue3succinctlyapp",
  storageBucket: "vue3succinctlyapp.appspot.com",
  messagingSenderId: "...",
  appId: "1:..."
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);

const db = getFirestore(app);

export {
  db
}
```



Note: I've deleted the `apiKey`, `messagingSenderId`, and `appId` values (highlighted in yellow), as these are specific to my environment and cannot be shared. For your environment, you'll have different values than mine.

First, we get a reference to the Cloud Firestore database.

```
const db = getFirestore(app);
```

Then, we export the database object so that other files within our project can use it.

```
export {  
  db  
}
```

Updating Document.vue

If you recall, our **Document.vue** file contains the code in Listing 6-b. This code works fine if our application is not using a database, which is currently the case. However, if we want the application to work with Firebase, there are some minor tweaks we'll need to make, which I've highlighted in bold with a yellow background.

Code Listing 6-b: The Current Document Component (Document.vue)

```
<script setup>  
  
const props = defineProps({  
  doc: {  
    type: Object,  
    default: () => {},  
  },  
  index: {  
    type: Number,  
    default: 0,  
  },  
});  
  
defineEmits(["edit-doc", "update-doc", "delete-doc"]);  
</script>  
  
<template>  
  <li>  
    <div class="doc">  
      <input  
        v-if="doc.isEditing"  
        type="text"  
        :value="doc.description"  
        @input="$emit('update-doc', $event.target.value, index)"  
      />  
      <span  
        v-else  
        :class="{  
          'expired-doc': doc.isExpired,  

```

```

    }"
  >
    {{ doc.description +
      (doc.expiration !== undefined ?
        (": " + doc.expiration)
        :
        ""
      )
    }}
  </span>
</div>
<div class="doc-actions">
  <button
    v-if="doc.isEditing"
    @click="$emit('edit-doc', index)"
  >
    Save
  </button>
  <button
    v-else
    @click="$emit('edit-doc', index)"
  >
    Edit
  </button>
  <button
    @click="$emit('delete-doc', doc.id)"
  >
    Delete
  </button>
</div>
</li>
</template>

<style lang="scss" scoped>
li {
  background-color: #bfffffff;
  display: flex;
  align-items: center;
  gap: 15px;
  padding: 20px 15px;

  &:hover {
    .doc-actions {
      opacity: 1;
    }
  }
}

```

```

    }
  }

  .doc {
    flex: 1;

    .expired-doc {
      text-decoration: line-through;
    }

    input[type="text"] {
      border: 2px solid #edfaff;
      width: 100%;
      padding: 3px 8px;
    }
  }

  .doc-actions {
    transition: 105ms ease-in-out;
    display: flex;
    gap: 3px;
    opacity: 0;

    .icon {
      cursor: pointer;
    }
  }
}
</style>

```

Let's go over the tweaks we need to do individually.

:value="doc.description"

Here, we are doing one-way data binding. So, whatever value the user enters in the description **input** field gets assigned (binds) to the **doc.description** property.

This does the job because we are not using a database. However, as we will use Cloud Firestore, we must ensure that if the data changes in the database, the change is reflected in the app. So, what we need is to implement two-way data binding. We can do this by using the **v-model** directive as follows:

v-model="doc.description"

Furthermore, when using a database, we don't want to emit the **update-doc** event every time the user inputs something, which is happening now.


```
@input="$emit('update-doc', $event.target.value, index)"
```

So, we don't need this functionality at all when using a database. Therefore, we can change this:

```
<input
  v-if="doc.isEditing"
  type="text"
  :value="doc.description"
  @input="$emit('update-doc', $event.target.value, index)"
/>
```

To this:

```
<input
  v-if="doc.isEditing"
  type="text"
  v-model="doc.description"
/>
```

Also, by using `doc.expiration` within the `span`, we display the document's full expiration date on the screen, including a time stamp and the time zone beside the date. Although internally, we can keep the full date and time, we should only display the expiration date (without the time stamp and the time zone).

We can achieve this by using this instead: `doc.expiration.substring(3, 15).trim()`.

So, we can change this:

```
<span
  v-else
  :class="{
    'expired-doc': doc.isExpired,
  }"
>
  {{ doc.description +
    (doc.expiration !== undefined ?
      (": " + doc.expiration)
      :
      ""
    )
  }}
</span>
```

To this:

```
<span
  v-else
  :class="{
    'expired-doc': doc.isExpired,
```

```

    }"
  >
    {{ doc.description +
      (doc.expiration !== undefined ?
        (": " + doc.expiration.substring(3, 15).trim())
        :
        ""
      )
    }}
  </span>

```

Lastly, when clicking the **Save** button, we don't need to emit the **edit-doc** event when using a database. This works fine when the app stores the data in memory, but it won't work when using a real-time database like Cloud Firestore.

```
@click="$emit('edit-doc', index)"
```

What we must do instead is emit the **update-doc** event as follows:

```
@click="$emit('update-doc', doc.description, index)"
```

Therefore, instead of doing this:

```

<button
  v-if="doc.isEditing"
  @click="$emit('edit-doc', index)"
>
  Save
</button>

```

We should do this:

```

<button
  v-if="doc.isEditing"
  @click="$emit('update-doc', doc.description, index)"
>
  Save
</button>

```

By applying these changes to our **Document.vue** file, the updated code should be as follows—changes are highlighted in bold:

Code Listing 6-c: The Updated Document Component (Document.vue)

```

<script setup>

const props = defineProps({
  doc: {
    type: Object,
    default: () => {},
  },

```

```

    },
    index: {
      type: Number,
      default: 0,
    },
  });

defineEmits(["edit-doc", "update-doc", "delete-doc"]);
</script>

<template>
  <li>
    <div class="doc">
      <input
        v-if="doc.isEditing"
        type="text"
        v-model="doc.description"
      />
      <span
        v-else
        :class="{
          'expired-doc': doc.isExpired,
        }"
      >
        {{ doc.description +
          (doc.expiration !== undefined ?
            (": " + doc.expiration.
              substring(3, 15).trim())
            :
            "")
          }}
      </span>
      <div class="doc-actions">
        <button
          v-if="doc.isEditing"
          @click="$emit('update-doc', doc.description, index)"
        >
          Save
        </button>
        <button
          v-else
          @click="$emit('edit-doc', index)"

```

```

    >
      Edit
    </button>
    <button
      @click="$emit('delete-doc', doc.id)"
    >
      Delete
    </button>
  </div>
</li>
</template>

<style lang="scss" scoped>
li {
  background-color: #bfffff;
  display: flex;
  align-items: center;
  gap: 15px;
  padding: 20px 15px;

  &:hover {
    .doc-actions {
      opacity: 1;
    }
  }

  .doc {
    flex: 1;

    .expired-doc {
      text-decoration: line-through;
    }

    input[type="text"] {
      border: 2px solid #edfaff;
      width: 100%;
      padding: 3px 8px;
    }
  }

  .doc-actions {
    transition: 105ms ease-in-out;
    display: flex;
    gap: 3px;
  }
}

```

```

    opacity: 0;

    .icon {
      cursor: pointer;
    }
  }
}
</style>

```

This will come to life after we make the final changes to the project, particularly to the **DocsPage.vue** file, which is what we'll do next.

Updating DocsPage.vue

It's in the **DocsPage.vue** file that I've added all the logic for the application to work with Cloud Firestore. This is because **DocsPage** is the parent of the **Document** component, and the **Document** component emits the **edit-doc**, **update-doc**, and **delete-doc** events up to its parent component, implementing them.

Let's look at the updated **DocsPage.vue** file, and then we'll review each change. The changes are highlighted in bold in the following code:

Code Listing 6-d: The Updated DocsPage Component (DocsPage.vue)

```

<script setup>
import { ref, onMounted } from "vue";
import { uid } from "uid";
import DocInput from '../components/DocInput.vue';
import Document from '../components/Document.vue';

import { collection, doc, onSnapshot,
  addDoc, updateDoc, deleteDoc,
  query, orderBy, limit } from "firebase/firestore";

import { db } from '../firebase.js';

const docsList = ref([]);
const docsQuery = query(collection(db, "docs"), orderBy("due", "asc"));

onMounted(() => {
  onSnapshot(docsQuery, (querySnapshot) => {
    docsList.value = []
    querySnapshot.forEach((doc) => {
      const d = {

```

```

        id: doc.id,
        description: doc.data().description,
        expiration: doc.data().expiration,
        isExpired: doc.data().isExpired,
        isEditing: doc.data().isEditing
      }
      docsList.value.push(d);
    });
  });
});

const createDoc = async (description, expiration) => {
  const today = new Date();
  const xYears = new Date(today.setFullYear(today.getFullYear() + 10));

  const d = (expiration == undefined) ? {
    id: uid(),
    description,
    expiration: xYears.toString(),
    isExpired: false,
    isEditing: false,
    due: xYears
  }
  :
  {
    id: uid(),
    description,
    expiration: expiration.toString(),
    isExpired: false,
    isEditing: false,
    due: new Date(expiration.toString().substring(3, 15).trim())
  };

  const ndoc = await addDoc(collection(db, "docs"), d);

  updateDoc(ndoc, {
    id: ndoc.id
  });
}

const toggleEdit = (docIdx) => {
  docsList.value[docIdx].isEditing = !docsList.value[docIdx].isEditing;
}

```

```

const updateDocument = async (d, docIdx) => {
  toggleEdit(docIdx);

  docsList.value[docIdx].description = d;

  const dc = await doc(db, "docs", docsList.value[docIdx].id);

  updateDoc(dc, {
    description: d
  });
}

const deleteDocument = async (id) => {
  docsList.value = docsList.value.filter(
    (doc) => doc.id !== id
  );

  await deleteDoc(doc(db, "docs", id));
}
</script>

<template>
  <main>
    <DocInput @create-doc="createDoc" />
    <ul>
      <Document
        :doc="doc"
        v-for="(doc, index) in docsList"
        :key="doc.id"
        :index="index"
        @edit-doc="toggleEdit"
        @update-doc="updateDocument"
        @delete-doc="deleteDocument"
      />
    </ul>
  </main>
</template>

<style lang="scss" scoped>
main {
  h1 {
    text-align: center;
    margin-bottom: 18px;
  }
}

```

```

margin: 0 auto;
width: 100%;
max-width: 500px;
display: flex;
flex-direction: column;
padding: 30px 15px;

.docs-list {
  display: flex;
  flex-direction: column;
  gap: 18px;
  list-style: none;
  margin-top: 25px;
}

.docs-msg {
  align-items: center;
  justify-content: center;
  display: flex;
  gap: 10px;
  margin-top: 25px;
}
}
</style>

```

The first thing we did was import the **onMounted** lifecycle hook from Vue. This is where we load the documents from the Cloud Firestore database.

```
import { ref, onMounted } from "vue";
```

Then, we import all the Firebase functions that we need to retrieve, add, update, and delete documents from Firebase.

```
import { collection, doc, onSnapshot, addDoc, updateDoc, deleteDoc, query,
orderBy } from "firebase/firestore";
```

Next, we import the reference to the database from the **firebase.js** file.

```
import { db } from '../firebase.js';
```


Querying the database

Following that, we create a Cloud Firestore query that retrieves all the documents within the database in ascending order, meaning that the documents that expire earlier will appear first.

```
const docsQuery = query(collection(db, "docs"), orderBy("due", "asc"));
```

The query requires a **collection** instance as the first parameter and the **order** condition as the second parameter.

To the **collection** function we pass the reference to the Cloud Firestore database (**db**) and the collection's name, which is **docs**.

The order condition (**orderBy**) orders the results in ascending order (**asc**) and uses the **due** property as the order criteria. The **due** property contains the same value as the document's expiration date but in date-time format.

Then, we have the **onMounted** lifecycle hook, which uses Firebase's **onSnapshot** event to listen for any changes in the database.

To Firebase's **onSnapshot** we pass the reference to the query (**docsQuery**), which retrieves all the documents from the database in the order previously specified as **querySnapshot**.

Then, for each document (**doc**) retrieved from the database, we create a new object (**d**) and add that to the list of documents (**docsList**).

```
onMounted(() => {
  onSnapshot(docsQuery, (querySnapshot) => {
    docsList.value = [];
    querySnapshot.forEach((doc) => {
      const d = {
        id: doc.id,
        description: doc.data().description,
        expiration: doc.data().expiration,
        isExpired: doc.data().isExpired,
        isEditing: doc.data().isEditing
      }
      docsList.value.push(d);
    });
  });
});
```

This is how the list of documents is retrieved from the database and kept in sync. If a document is added to the database manually, the application will immediately show that document—this is possible because of **onSnapshot**.

Adding a document

Next, we have the `createDoc` function, which, as its name implies, is responsible for adding a document to the database when the user clicks the **Add** button in the app's UI.

```
const createDoc = async (description, expiration) => {
  const today = new Date();
  const xYears = new Date(today.setFullYear(today.getFullYear() + 10));

  const d = (expiration == undefined) ? {
    id: uid(),
    description,
    expiration: xYears.toString(),
    isExpired: false,
    isEditing: false,
    due: xYears
  }
  : {
    id: uid(),
    description,
    expiration: expiration.toString(),
    isExpired: false,
    isEditing: false,
    due: new Date(expiration.toString().substring(3, 15).trim())
  };

  const ndoc = await addDoc(collection(db, "docs"), d);

  updateDoc(ndoc, { id: ndoc.id });
}
```

You'll first notice that this function is now asynchronous (before these changes, it wasn't), which is why `async` is used. This is because we must `await` the result returned by the `addDoc` Firebase function.

The function receives as a parameter the **description** and the **expiration** date of the document that will be created.

By default, for any new document for which the user has not indicated (selected) an **expiration** date, the application will automatically set the document's **expiration** date to 10 years from when the user added the document. This is what these two instructions do:

```
const today = new Date();
const xYears = new Date(today.setFullYear(today.getFullYear() + 10));
```

Then, we create the document object that will contain the information about the document added by the user. There are two ways to create the document object (**blue** and **green**) based on one condition (highlighted in **yellow**).

```

const d = (expiration == undefined) ? {
  id: uid(),
  description,
  expiration: xYears.toString(),
  isExpired: false,
  isEditing: false,
  due: xYears
}
:
{
  id: uid(),
  description,
  expiration: expiration.toString(),
  isExpired: false,
  isEditing: false,
  due: new Date(expiration.toString().substring(3, 15).trim())
};

```

If the document's **expiration date is undefined**, most likely, the user entered the document's **description** but did not explicitly select an **expiration** date. In that case, the document is created with an **expiration** date of 10 years (**xYears**) in the future from the document's creation date (**today**). On the other hand, if the document's **expiration date is not undefined**, then the user specifies the document's **expiration** date, and that date is used instead.

The **expiration** and **due** properties are the same, representing the document's expiration date. The difference is that the **expiration** property uses the date as a string, and the **due** property contains the date in a date-time format (without a time stamp, which is why **substring** is used).

With the document (**d**) now created in memory, we can add the document to Cloud Firestore using Firebase's **addDoc** function, as follows:

```

const ndoc = await addDoc(collection(db, "docs"), d);

```

By default, the document (**d**) created gets assigned a random and unique **id** (**id: uid()**). However, this **id** is for internal use only, and differs from the one that Cloud Firestore will automatically assign to the document within the database.

Given that we need the document's **id** assigned by Firebase for deleting documents, we won't be able to find a document to delete within the database using the **id** value we supplied when creating it—in other words, **id: uid()** is no good.

Therefore, once the document has been created using the default **id**, we need to update that document, specifically the document's **id** property, with the Firebase **id** value (which is only available after the document has been added to the database). We can do this as follows:

```

updateDoc(ndoc, { id: ndoc.id });

```

Updating a document

Next, the **updateDocument** function updates a document within the Cloud Firestore database. This function was previously called **updateDoc** (before we added Firebase to our project). This function gets triggered when the user clicks the **Save** button. I renamed this function because there's already a Firebase function called **updateDoc**. So, let's have a look at it:

```
const updateDocument = async (d, docIdx) => {
  toggleEdit(docIdx);

  docsList.value[docIdx].description = d;

  const dc = await doc(db, "docs", docsList.value[docIdx].id);

  updateDoc(dc, {
    description: d
  });
}
```

The first thing you'll notice is that the function is asynchronous (**async**), and that's because it must **await** the result returned by invoking Firebase's **doc** function.

To the **updateDocument** function, we pass the description (**d**) and the position (**docIdx**) of the document to update within the **docsList** array, which represents the list of documents.

First, we call the **toggleEdit** function, which changes the **isEditing** status of the document. By doing this, once the document has been updated, the **Edit** button will be shown again, and the **Save** button will be hidden.

```
toggleEdit(docIdx);
```

Then, we change the document's **description** (for the current document being edited) within the **docsList** array.

```
docsList.value[docIdx].description = d;
```

After that, we call Firebase's **doc** function, to which we pass the document object we want to retrieve as a third parameter.

```
const dc = await doc(db, "docs", docsList.value[docIdx].id);
```

Once we have retrieved the document we want to update from the database, we can update its description as follows:

```
updateDoc(dc, { description: d });
```

By calling Firebase's **updateDoc** function, we can update specific document properties, such as the document's description in this case.

Deleting a document

We can delete a document from the database by using the **deleteDocument** function.

```
const deleteDocument = async (id) => {  
  docsList.value = docsList.value.filter(  
    (doc) => doc.id !== id  
  );  
  
  await deleteDoc(doc(db, "docs", id));  
}
```

This function is now asynchronous (**async**) because we have to invoke Firebase's **deleteDoc** function and **await** its completion.

First, we have to remove the document we want to delete from the list of documents (**docsList** array), which we can do by using JavaScript's array **filter** method, as previously explained.

```
docsList.value = docsList.value.filter(  
  (doc) => doc.id !== id  
);
```

Then, using the Firebase **id** of the document we want to remove, we can invoke Firebase's **deleteDoc** function.

```
await deleteDoc(doc(db, "docs", id));
```

This will remove the document from the Cloud Firestore database.

Update bindings

Lastly, two additional minor changes were made. Originally, we had this:

```
<Document  
  :doc="doc"  
  v-for="(doc, index) in docsList"  
  :key="doc.id"  
  :index="index"  
  @edit-doc="toggleEdit"  
  @update-doc="updateDoc"  
  @delete-doc="deleteDoc"  
>
```

This meant that:

- The **update-doc** event would trigger the original **updateDoc** function (renamed **updateDocument** due to the name clash with the **updateDoc** Firebase function).
- The **delete-doc** event would trigger the original **deleteDoc** function (renamed to **deleteDocument** due to the name clash with the **deleteDoc** Firebase function).

Considering these, we changed it to:

```
<Document
  :doc="doc"
  v-for="(doc, index) in docsList"
  :key="doc.id"
  :index="index"
  @edit-doc="toggleEdit"
  @update-doc="updateDocument"
  @delete-doc="deleteDocument"
/>
```

Testing the app

It's now time to save all the changes in VS Code and run the app by running the **npm run dev** command from the built-in terminal within VS Code.

Open the Firebase console, navigate to the **Cloud Firestore** section, click the **Data** tab, and manually remove any test documents you might have added to the database before. Let's start with a clean database.

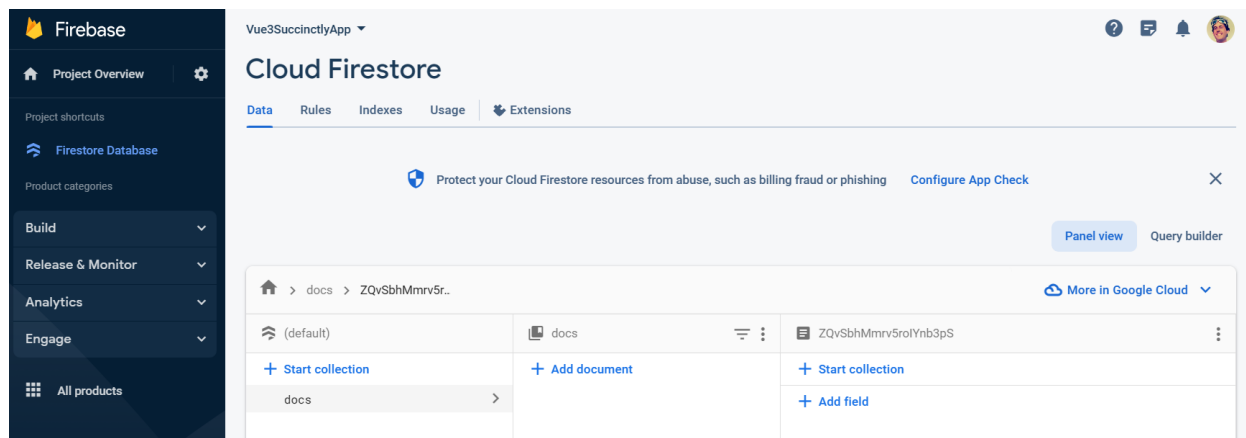


Figure 6-a: Cloud Firestore (Data Panel Without Documents)

Switch over to the application and enter some documents. In my case, I'll enter a couple of passports with their respective expiration dates, as shown in the following figure:

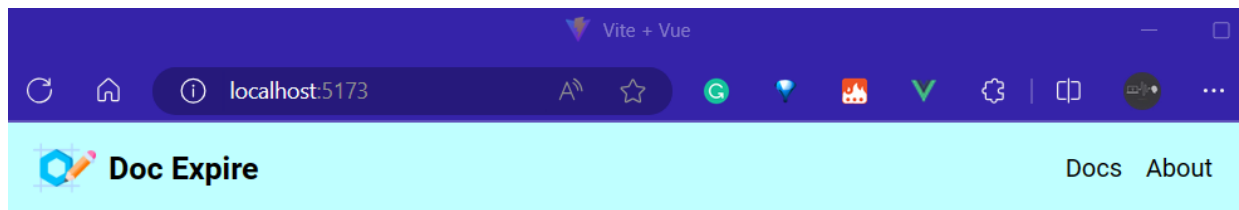


Figure 6-b: The App Running with Documents

If you switch to the Firebase console, you should see the documents within the database.

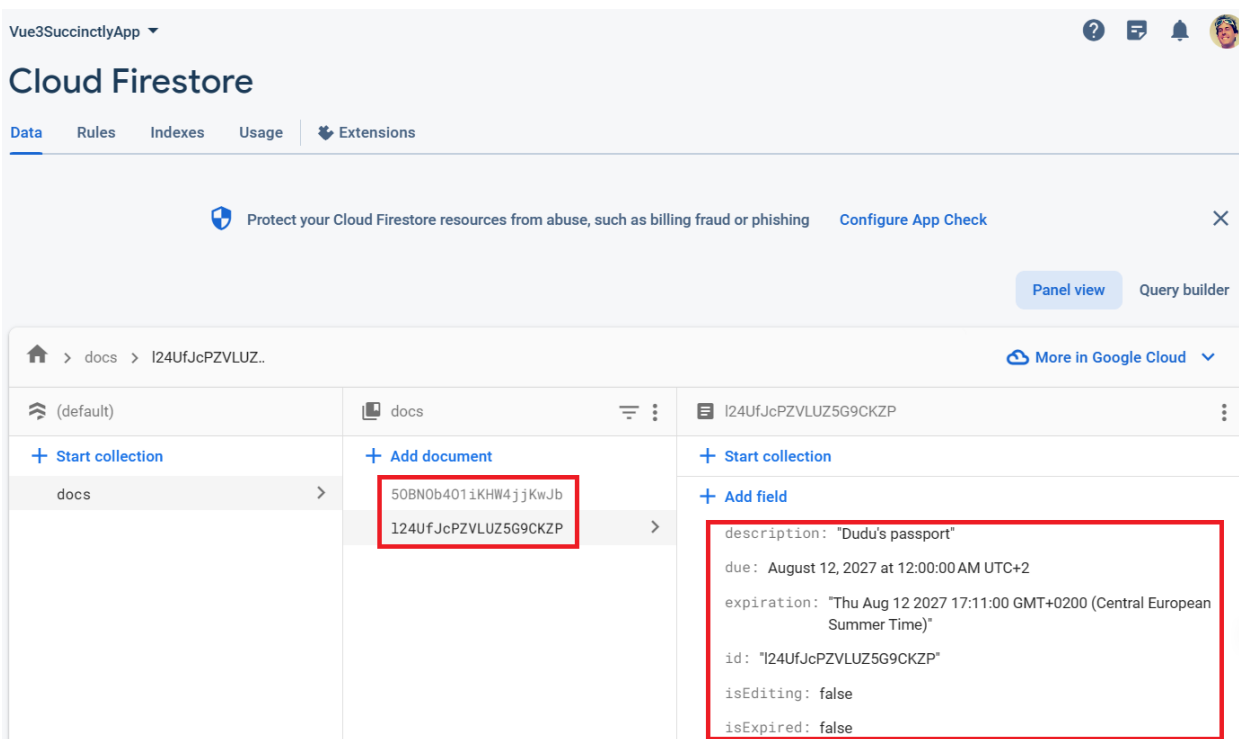


Figure 6-c: Cloud Firestore (Data Panel with Documents)

Any changes you make through the application or the Firebase console will reflect immediately in the other. This is because we use Firebase's **onSnapshot** to get a real-time copy of the records in the database.

Feel free to add, update, or delete documents from the app's user interface or the Firebase console so you can see how the real-time data syncing works (and how cool it is).

Closing thoughts

Throughout these past chapters, we've built this small demo application and highlighted some of the crucial features of Vue 3 along the way.

The book aimed to focus on understanding the fundamentals of Vue 3 rather than building a full-blown app full of UI gimmicks and features, which is why I left some of those features out. Nevertheless, you can take that challenge into your own hands and expand the app further. For example, add a feature to update the document's expiration date.

Another thing you can do is add some of those cool-looking icons from the library we previously installed, which we did not get to use. You could add functionality to trigger a notification when a document gets close to its expiration date—that would be an excellent feature. You can add some of these suggestions to the application, but I'm sure there are other potentially helpful features to add as well.

Hopefully, this book has given you some insights on how to get up and running quickly with Vue 3 and take your learning journey further with this technology. Thank you for choosing the *Succinctly* series and this book, and until next time, take care and keep learning.