VUE.JS

SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Vue.js Succinctly

By

Author Ed Freitas

Foreword by Daniel Jebaraj

Copyright © 2019 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

All rights reserved.

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, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator and Proofreader: Tres Watkins, content development manager, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books 6

About the Author 8

Acknowledgements 9

Introduction 10

Chapter 1 Setup 11

Project overview 11

Installation 12

Creating an app (CLI) 13

Creating an app (Vue UI) 15

Project dashboard: Vue Project Manager 18

Summary 24

Chapter 2 App Basics 25

Quick intro 25

Editor 25

Default project structure 25

Index.html, main.js, and App.vue 28

The HelloWorld component 33

App component structure 38

Vue’s data within components 40

Docs component 42

Vue DevTools 44

Getting the data into the Docs component 45

Summary 49

Chapter 3 Expanding the App: UI 50

Quick intro 50

Item.vue 50

Vuetify 53

Styling App.vue 54

Styling Docs.vue..57

Styling Item.vue..58

Creating Doc.vue..60

Adapting App.vue for Doc.vue 69

Adapting Docs.vue for Doc.vue 72

Adapting Item.vue for Doc.vue 73

Adjustments to Doc.vue..79

Summary..85

Chapter 4 Finalizing the App: Database 86

Quick intro 86

Sheetsu dashboard 86

Dynamic data loading..88

Deleting documents..91

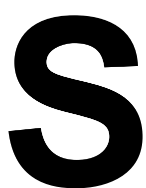
Saving new or existing documents 94

Project source code..100

Closing comments..101

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.



taying on the cutting edge

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

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

Information is plentiful but harder to digest

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

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

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

The *Succinctly* series

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

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

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

The best authors, the best content

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

Free forever

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

Free? What is the catch?

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

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

Let us know what you think

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

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

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

About the Author

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

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

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

Acknowledgements

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

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

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

Introduction

Vue.js—commonly referred to as Vue and pronounced like *view*—is an open-source JavaScript framework for building user interfaces and single-page applications using JavaScript, HTML, and CSS.

Vue was created by Evan You after he worked for Google using Angular for various projects. His idea was to use the parts of Angular that he really liked, and build something that was very easy to use and lightweight, yet would still have many of the amazing parts and power of Angular.

Vue uses an HTML-based template syntax that allows reactive data binding to the DOM and can extend basic HTML elements as reusable components. It also provides support for front-end hashed routing though the open-source Vue Router package.

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

If current trends continue, Vue is on the path to become the world's most popular JavaScript framework as developer adoption steadily increases.

Aside from its awesome productivity features, Vue's success is closely tied with JavaScript's ascension as the *go-to language* for developing modern web applications—as logic that used to run on a web server can now be executed in a browser.

Vue is also a progressive framework, which means that if you already have an existing web application, you can just plug Vue into one of its parts—which might need a dynamic and more interactive user experience. Vue has excellent performance characteristics.

Vue can also be used when starting an application from scratch—when you want to add more business logic into your frontend, rather than the backend. It has a rich ecosystem of packages that provide great core features, and others that manage state and handle routing.

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

All these features make Vue a great choice for developing frontend applications, and a great framework to learn—which as a developer, will make you more productive and valuable in the market.

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

Chapter 1 Setup

Project overview

The application that we'll be building throughout this book will be a web app that we can use to keep track of important personal documents that have an expiration date, such as passports, driver's licenses, or credit cards.

This is the same app concept that was explored within *Flutter Succinctly*—with the difference that before, we created an Android application using Flutter, and now, we'll create a web app using Vue. Below is what the Flutter app concept looks like.

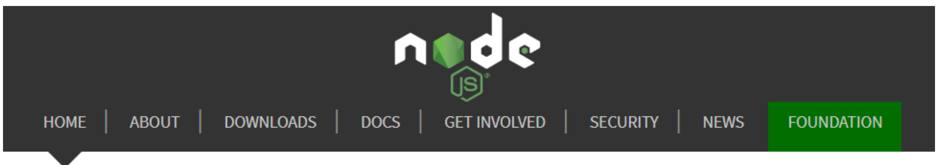
Figure 1-a: The Mobile Version of the Demo App (Flutter)

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

Sounds awesome—let's get our engines ready, so we can start setting up our development environment.

Installation

The Vue CLI (command-line interface) requires NPM (Node Package Manager), which needs Node.js to be installed. To install Node.js, simply go to the Node.js website and download the **LTS** (Long Term Support) or the latest **Current** version.



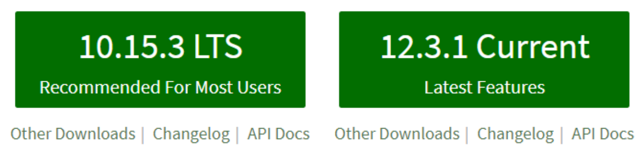


Figure 1-b: Node.js Website

The installation of Node.js is very simple and consists of a few steps that can easily be carried out using an installation step-by-step wizard.

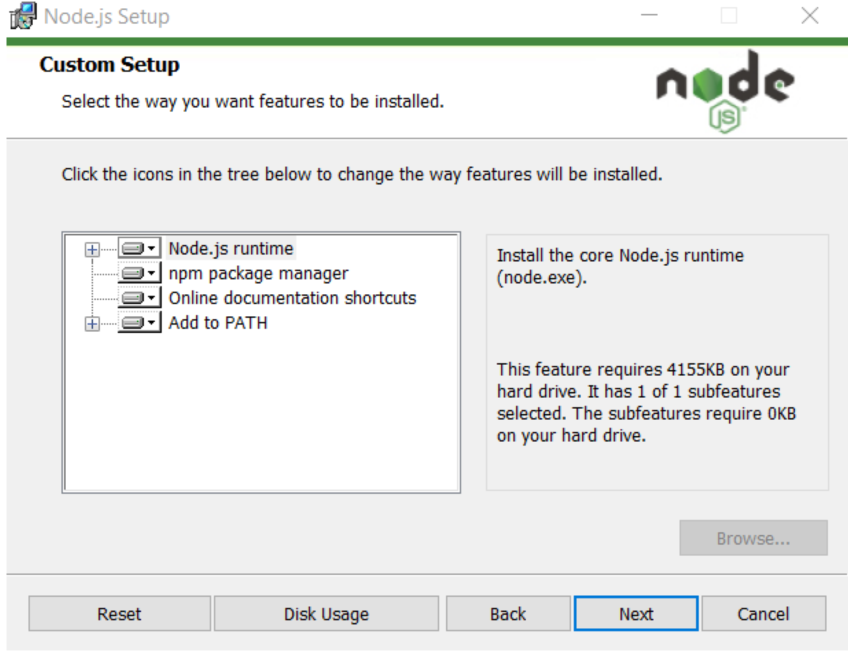


Figure 1-c: Node.js Installation Wizard

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 typing in the following command.

Listing 1-a: Install Vue CLI Command

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

Once installed, you can run the following command to check which version of the CLI was installed on your machine.

Listing 1-b: Check Vue CLI Version

```
vue --version
```

With the Vue CLI installed, we are ready to create an application.

Creating an app (CLI)

There are two ways to create an application with Vue: one is using the Vue CLI, and the other is using the Vue UI tool. Let's explore both ways.

Open the command line and type in the following command—this will require us to choose a preset.

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

```
vue create test
```

In my case, I've chosen the default preset, which includes **babel** and **eslint**.

```
Vue CLI v3.8.2
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
  Manually select features
```

Figure 1-d: Choosing a Preset

Once selected, the CLI installs the required modules—we can see this as follows.

```
Vue CLI v3.8.2
✨  Creating project in C:\Projects\VueJs Succinctly\demo\test.
🗃  Initializing git repository...
⚙  Installing CLI plugins. This might take a while...

[          ......] / extract:is-path-cwd: sill extract is-path-cwd@^2.0.0 ext
```

Figure 1-e: CLI Installing Modules

After the process has finished, you'll see the following information—which describes the commands to run the application we just created.

```
Successfully created project test.
Get started with the following commands:

$ cd test
$ npm run serve
```

Figure 1-f: App Creation Finalized

Let's go ahead and test that out. I'll enter the following commands on the command line.

Listing 1-d: Running the Created App

```
cd test
npm run serve
```

This fires up a local development server that supports hot-reloading, which can be found on the following addresses.

```
App running at:
- Local:   http://localhost:8080/
- Network: http://192.168.1.82:8080/

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

Figure 1-g: App Running CLI Message

Let's open a browser instance and point it to the local address indicated—in my case: **http://localhost:8080**.

I then see the following welcome screen on my browser window.

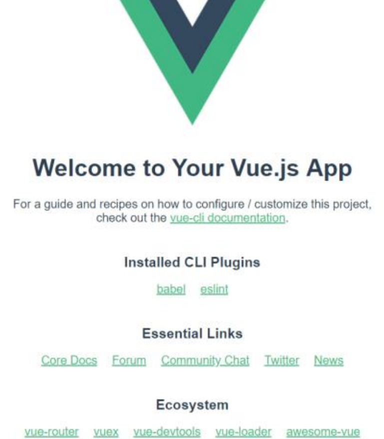


Figure 1-h: App Running on the Browser

Cool—we now have the basic Vue demo application running. Now let's explore how we can create the same application but using the Vue UI.

Creating an app (Vue UI)

Before we can create the demo app with the Vue UI, we have some cleaning up to do. So, close the browser where the app is running, then within the command line, press Ctrl+C to interrupt the dev server—this will show the following message.

```
  App running at:
  - Local:   http://localhost:8080/
  - Network: http://192.168.1.82:8080/

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

Terminate batch job (Y/N)?
```

Figure 1-i: Interrupting the Dev Server

Then, type **Y** (uppercase or lowercase) and press **Enter** to stop the dev server. Once done, remove all the files and subfolders contained within the test folder that the Vue CLI created.

We are now ready to create the app with the Vue UI tool. To do that, let's go back to the command line and enter the following command.

Listing 1-e: Starting Vue UI

```
vue ui
```

This will start and open the Vue UI interface on your browser—in my case on the following URL: **http://localhost:8000/project/select**.

Here's how the Vue UI looks.

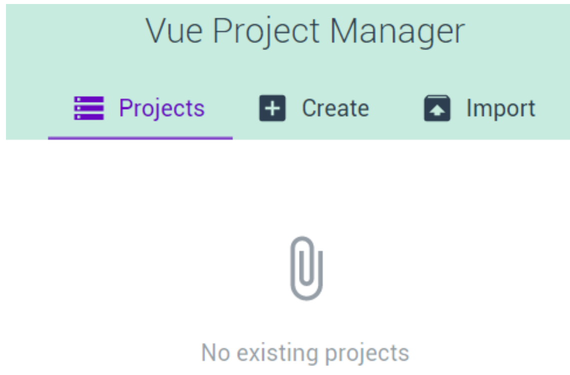


Figure 1-j: Vue UI (Vue Project Manager)

Awesome! Next, let's click **Create**, which is just next to the **Projects** button.

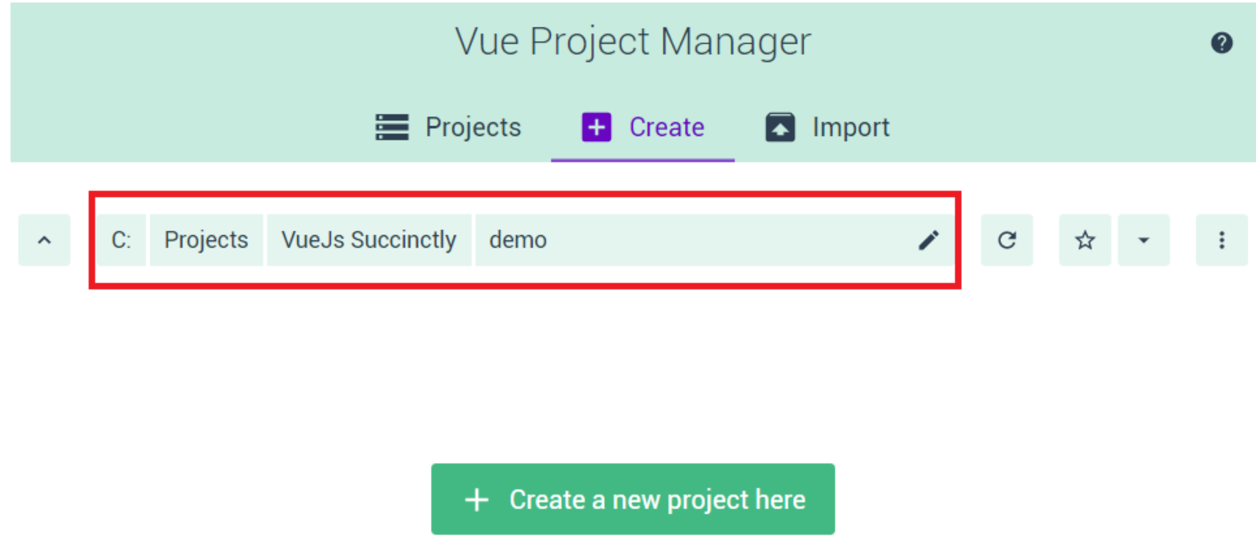


Figure 1-k: Vue UI (Vue Project Manager) after Clicking the Create Button

The Vue Project Manager shows the folder path where the application can be created. This is the same folder path where the `vue ui` command was executed from.

So, to create the application, let's click **+ Create a new project here**. Once we've done that, we'll see the following screen.

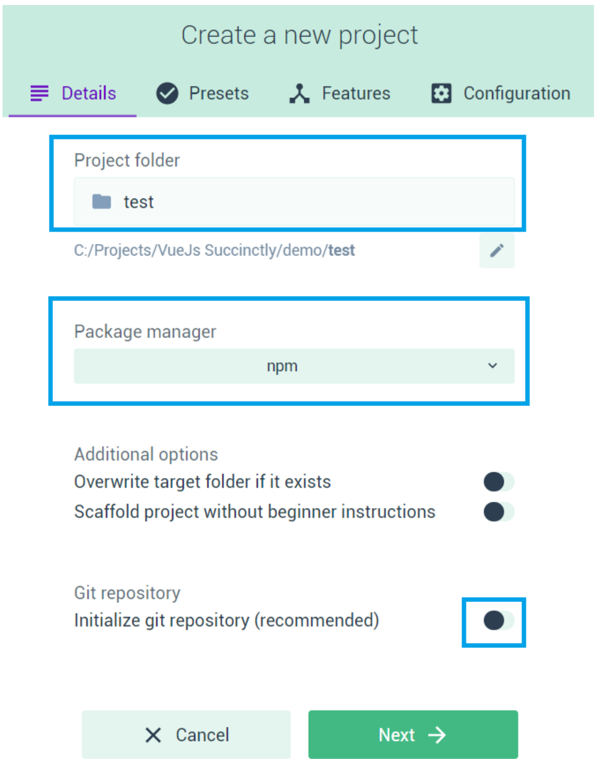


Figure 1-l: Vue Project Manager: Creating a New Project (Step 1)

I've set the **Project folder** value to **test** and the **Package Manager** to **npm**, and disabled the **Git repository** option—you can choose other options if you wish. Once you're done, click **Next**—this will take us to the final creation screen, which we can see as follows.

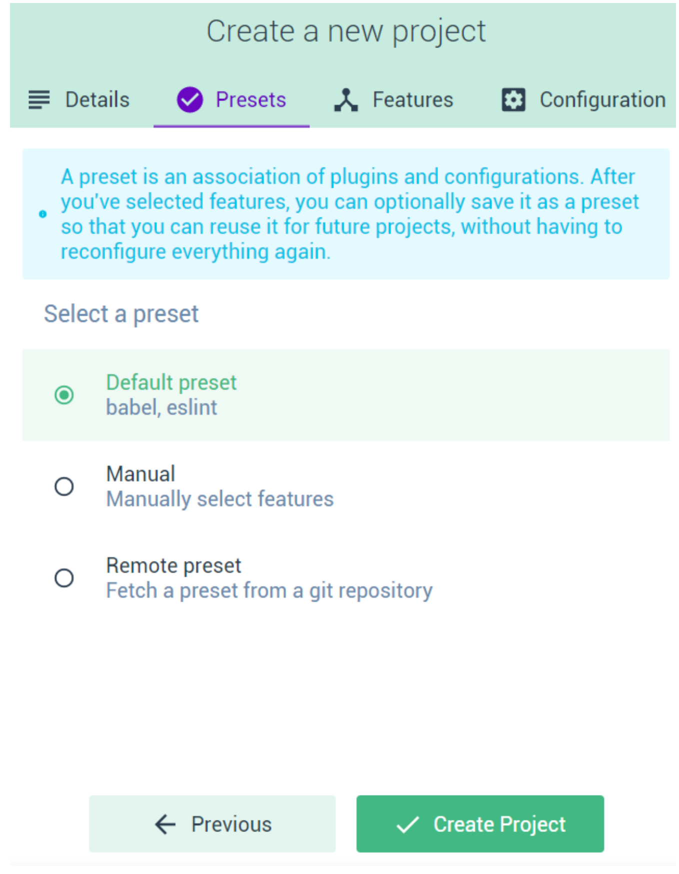


Figure 1-m: Vue Project Manager: Creating a New Project (Step 2)

In this step, we will choose the combination of plugins and configurations to use—this is known as a **preset**. In my case, I've chosen the **Default preset**, which uses **babel** and **eslint**. You may choose another option if you wish. Once you're done, click **Create Project**—this will display the following screen.

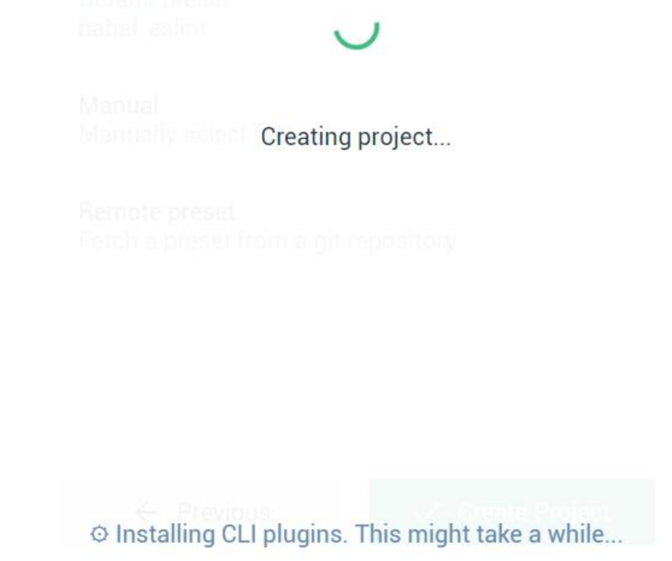


Figure 1-n: Vue Project Manager: New Project Being Created

Once the project has been created, the Vue Project Manager will present a Project dashboard for the project that was just created—which is what we'll explore next.

Project dashboard: Vue Project Manager

One of the great features about the Vue Project Manager is the ability it gives us to manage multiple Vue projects via a clear and easy-to-use interface. Let's explore the options it provides us for managing the **test** project we've just created.

The main screen of the Project dashboard is a welcome screen, which by default, contains the Welcome tips and Kill port widgets.

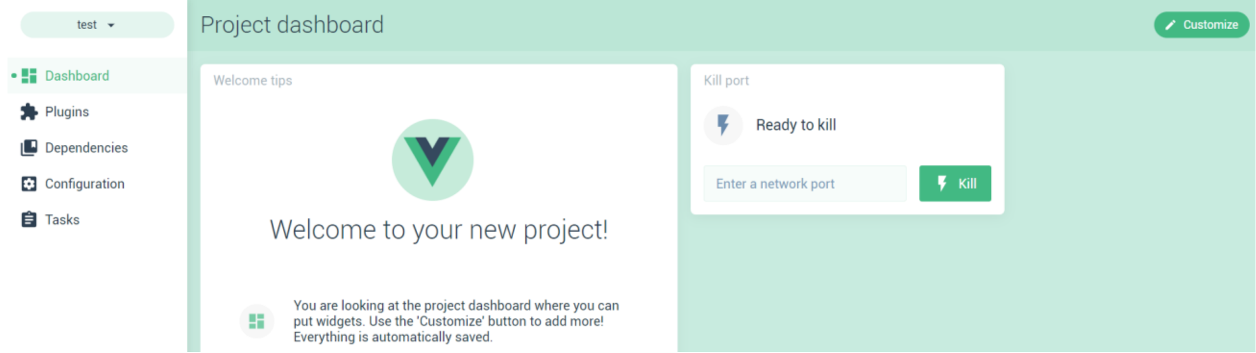


Figure 1-o: Project Dashboard Main Screen

This screen is customizable—you can use it to add or remove widgets.

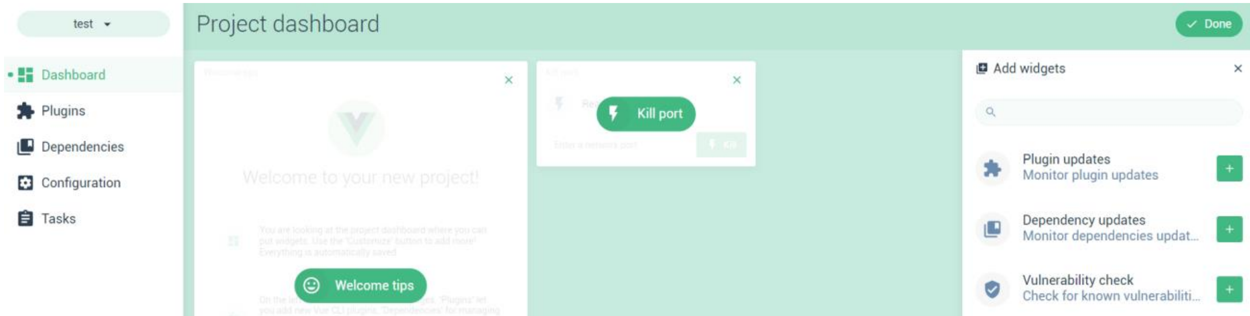


Figure 1-p: Project Dashboard: Main Screen (After clicking the Customize button)

Click **Customize**, and you will be presented the option to add widgets or remove any of the existing ones.

In my case, I'm going to add the **Run task** widget, which I would like to use to run a serve task—which will basically allow me to serve (run) my **test** application from the Vue Project Manager UI, without having to use the command line.

Once you've added the **Run task** widget, click **Configure widget** as shown in the following figure.

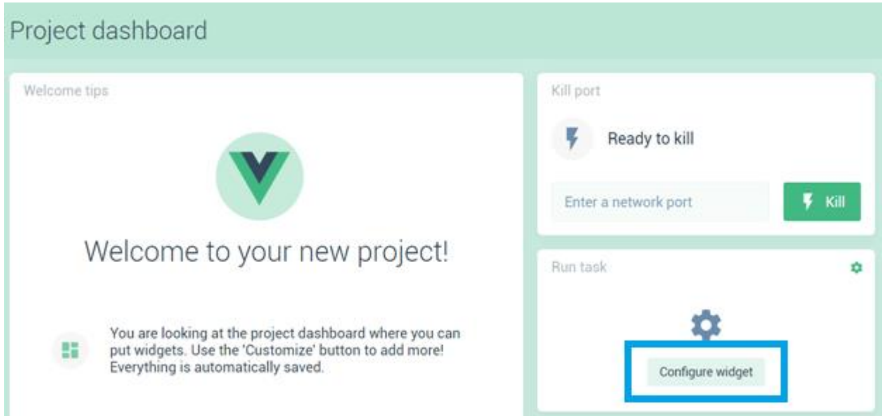


Figure 1-q: Project Dashboard: Main Screen (After adding the Run task widget)

Next, we'll see the following screen, which we can use to select the task we would like to execute.

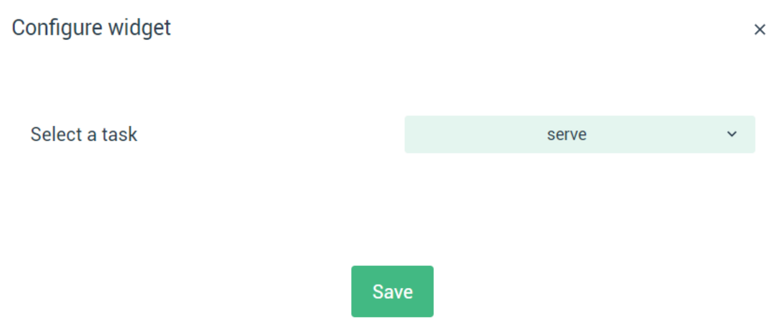


Figure 1-r: Project Dashboard: Run task widget (After adding the Configure widget)

In my case, I’ve chosen the **serve** task option. Once you have chosen the task, click **Save**. After that, the Project dashboard screen will look as follows.

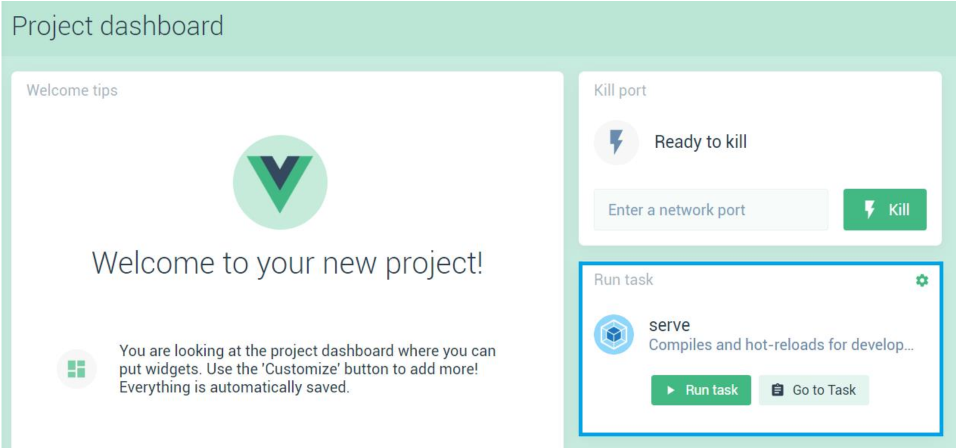


Figure 1-s: Project Dashboard: Run task widget (After the changes)

Notice how the Run task widget has changed, and the serve option has been populated and configured. This can be initiated by clicking **Run task**.

Let’s explore some of the other options available on the Project dashboard. The next option we have available is the Project plugins screen. Here we can see what plugins our project will use.

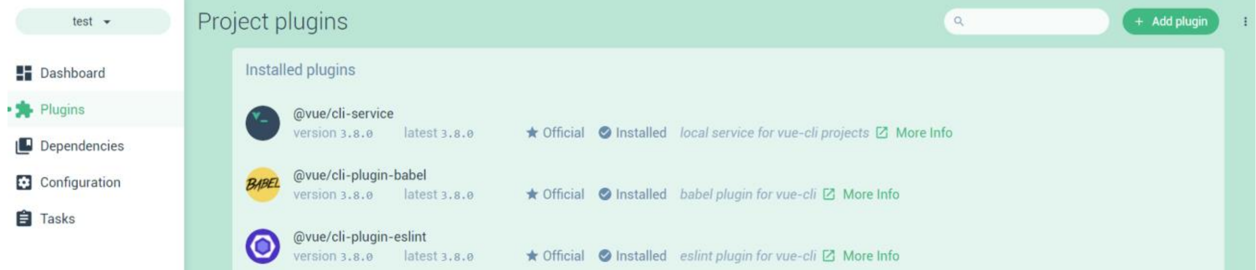


Figure 1-t: Project Dashboard: Project plugins

Notice that based on the default **preset** option that I chose when the **test** application was created, the following plugins were installed:

- @vue/cli-service (version 3.8.0)
- @vue/cli-plugin-babel (versión 3.8.0)
- @vue/cli-plugin-eslint (versión 3.8.0)

There’s also the option to add additional plugins—let’s click **+ Add plugin**, just to explore which other plugins are available.

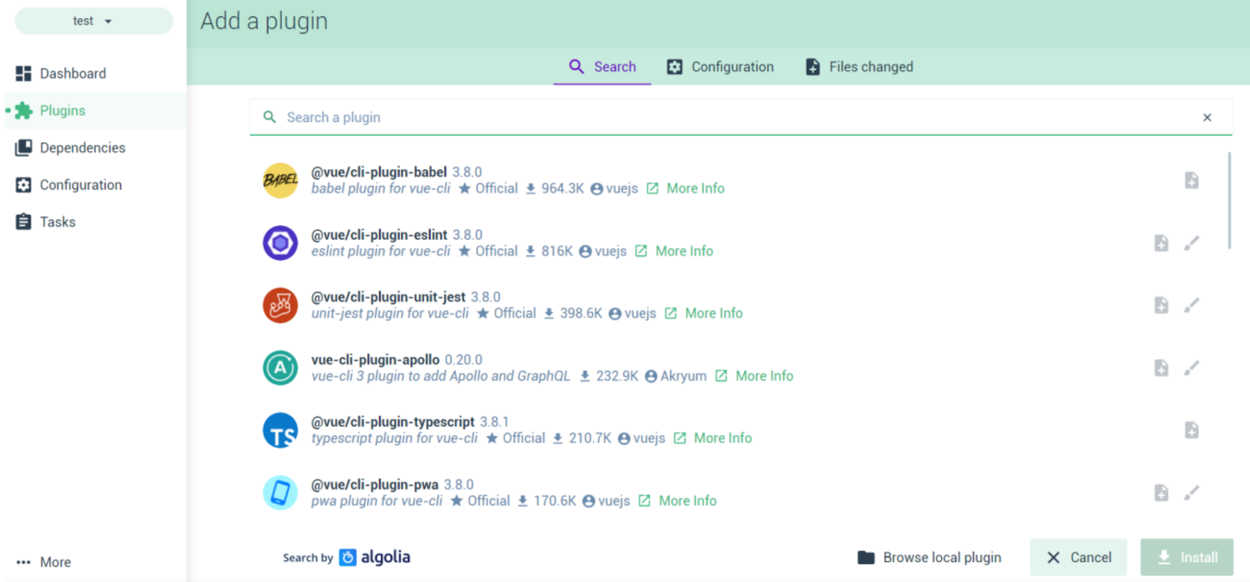


Figure 1-u: Project Dashboard: Other plugins

This is a full list of all the available public published plugins that exist for Vue—basically, Vue's ecosystem of plugins.

There's also the option to use local plugins, by clicking on the Browse local plugin button. Now, let's have a look at the Project dependencies screen.

There are two types of project dependencies: main dependencies and development dependencies.

Main dependencies are those that are required for the execution of the **test** application—they are needed for the runtime execution of the app.

Development dependencies are required during the development of the application. Once the application is linted (compiled) and served, these dependencies are not essential for the application to run—they are only needed during development.

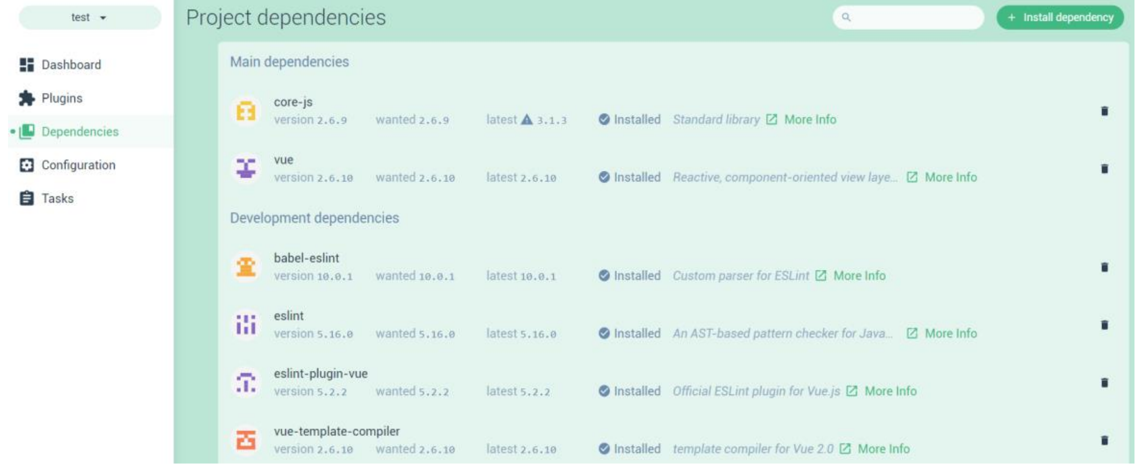


Figure 1-v: Project Dashboard: Project dependencies

There's also the option to install additional dependencies, either main dependencies or development dependencies, which are publicly available and published on the NPM registry.

The following figure shows the list of available dependencies visible after clicking **+ Install dependency**.

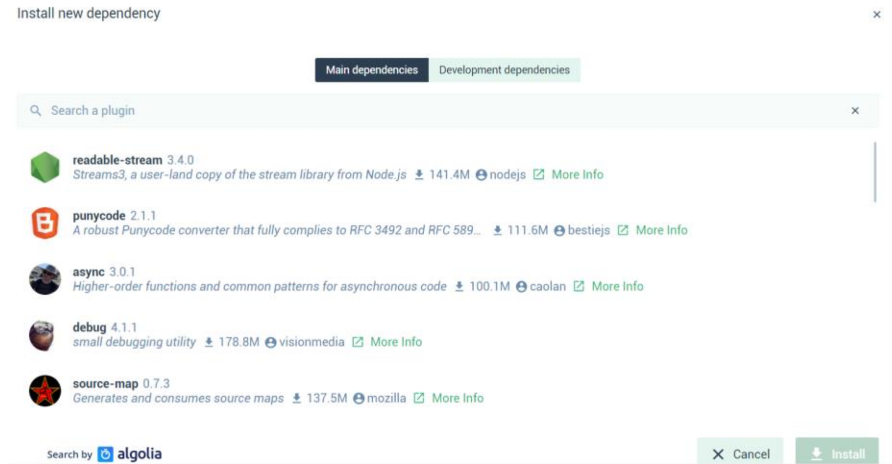


Figure 1-w: Project Dashboard: Install dependencies

The Project dashboard also gives us the possibility to easily configure the plugins installed. This can be done on the Project configuration screen, where each plugin that has configuration options is displayed.

The following figure shows the configuration options for the Vue CLI plugin. Being able to set the options of plugins using a UI such as this is much easier than having to use JSON files or the command line—this is one of the advantages of using the Vue UI.

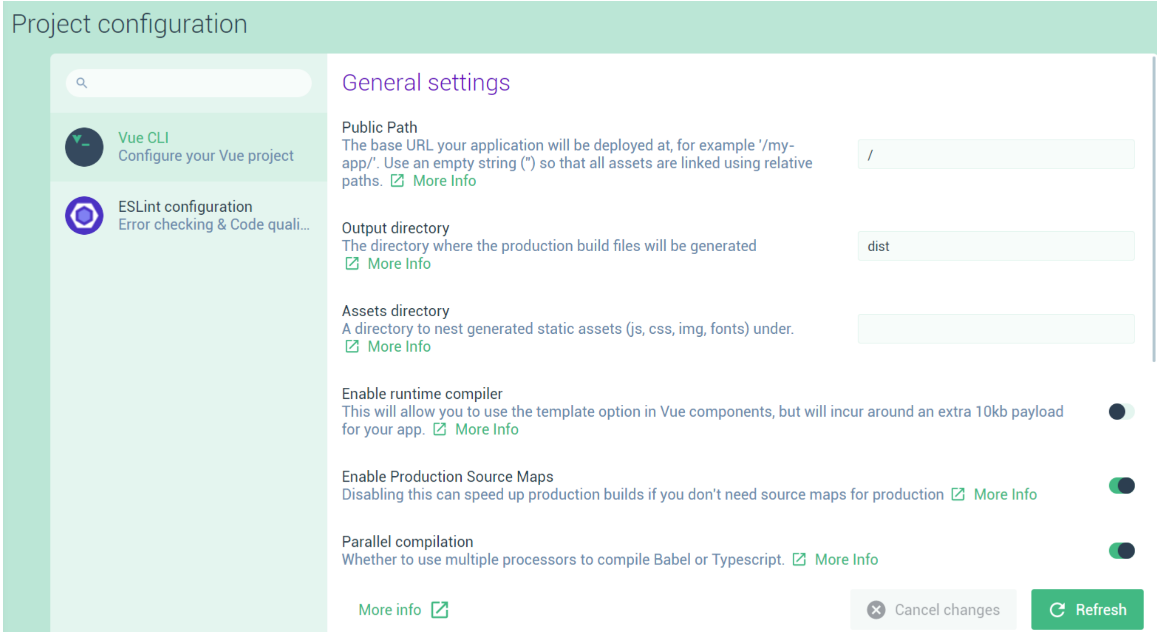


Figure 1-x: Project Dashboard: Vue CLI Plugin Options

The last screen of the Project dashboard is the Project tasks screen. Here's where we can serve, build, lint, and inspect our application when it is running.

The Project tasks screen is not only useful for executing the serve, build, lint, and inspect tasks, but it's incredibly valuable given that we can keep track of how the application is running and performing—this is visible through the Output, Dashboard, and Analyzer tabs—which we can see as follows.

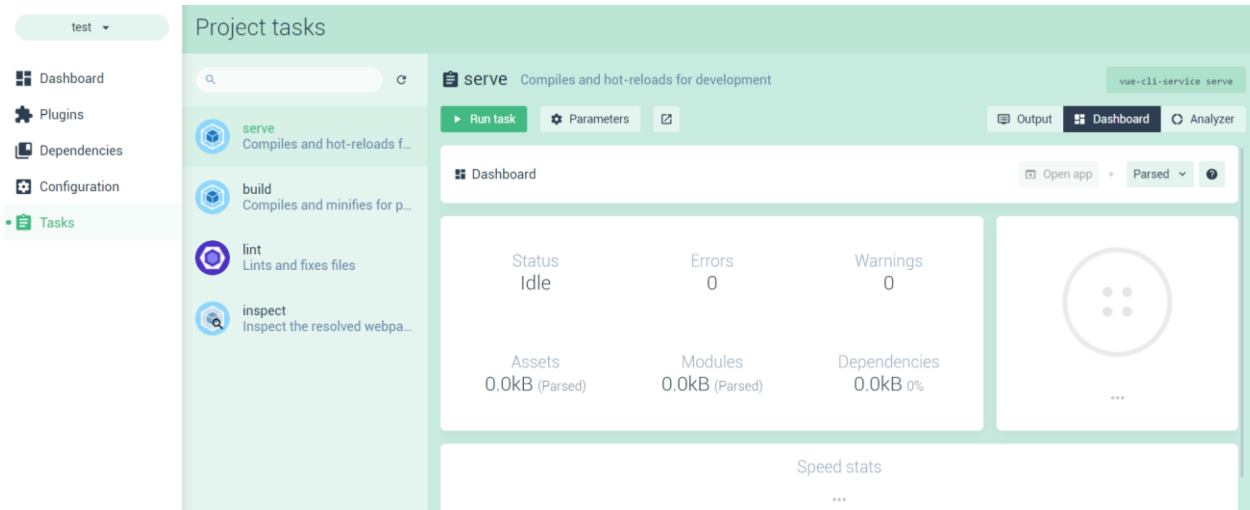


Figure 1-y: Project Dashboard: Project tasks

Notice how we have the option to execute the task by clicking the **Run task** button, and then monitoring what happens by looking at the Dashboard, which is specific to the task being executed. This feature is not only cool, but extremely powerful—it gives us real-time details of the task running.

The following figure shows how the Dashboard looks a few seconds after the **serve** task has been executed—notice the speed stats and general status of our running application.

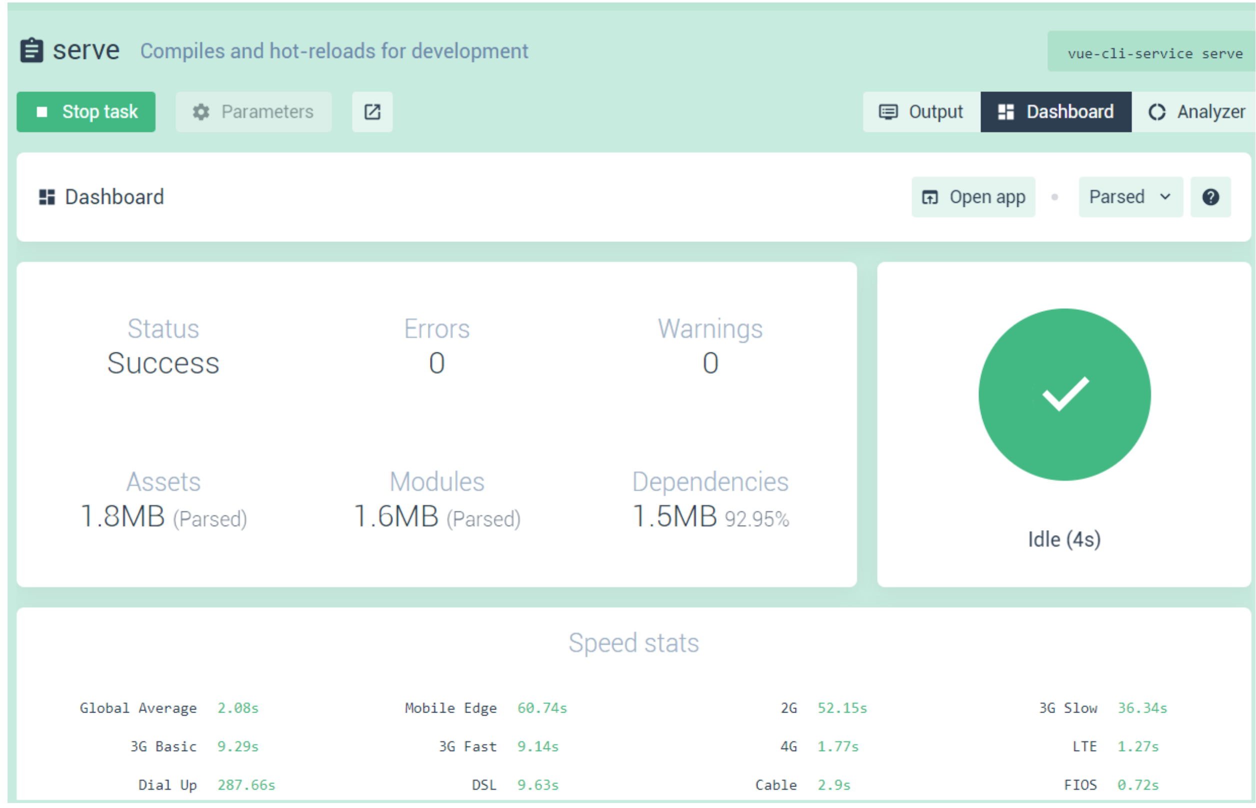


Figure 1-z: Project Dashboard: Serve task execution details

As we have seen, the Vue UI is a very powerful tool that makes the development of our application easier to manage and analyze.

I tend to prefer the Vue UI method for creating apps, but this is mostly a matter of personal taste. Feel free to choose your own method.

Summary

Throughout this chapter, we've explored how to get a Vue environment set up and ready. We've been able to do this by installing the necessary requirements and then setting up a **test** application using both the Vue CLI and the Vue UI.

Next, we'll explore the default project structure and start modifying the **test** app and lay the foundation for the application we will be building throughout this book.

Chapter 2 App Basics

Quick intro

With our Vue environment set up, we are now ready to start inspecting our test application and modifying it—which is what we will do in this chapter. Let's dive in.

Editor

I'll be using Visual Studio Code (also known as VS Code) on Windows as my editor and Integrated Development Environment (IDE) of choice throughout this book—as it is easy to use and has great features. However, feel free to use any other editor you feel comfortable with.

If you decide to go with VS Code, I suggest you install the Vetur extension, which provides Vue tooling for VS Code—as we can see in the following figure.

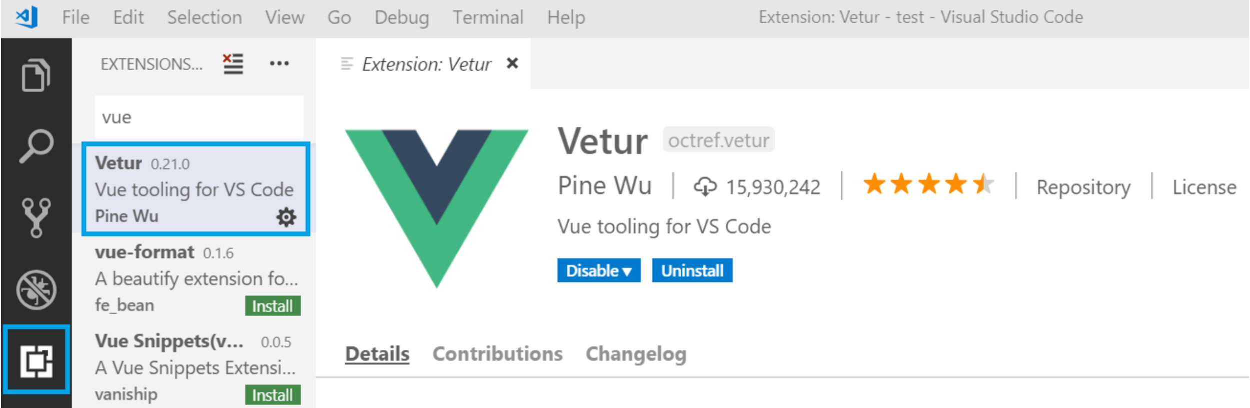


Figure 2-a: The Vetur Extension for VS Code Installed

With VS Code set up, let's explore the default project and file structure created by the Vue CLI or Vue UI.

Default project structure

By default, the application project structure created by the Vue CLI or Vue UI contains three main folders:

- **node_modules**: The repository of node modules that the application will use during development and runtime.
- **public**: Contains the resultant files that can be deployed to a web server once the Vue app has been built.
- **src**: Contains the source files that we will be working on.

Let's open VS Code and explore the folder structure of the test application we created—this is how it looks:

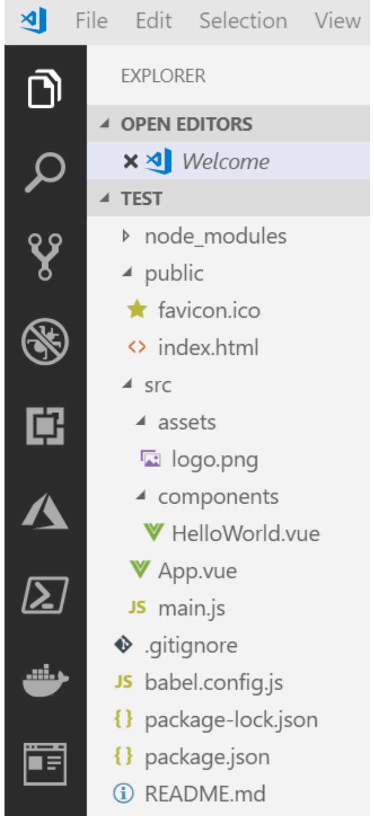


Figure 2-b: The Default Project Folder Structure

Notice that the **src** folder contains two subfolders: **assets** and **components**. There's also an **App.vue** file in the root of the **src** folder, which is the application's main component and entry point.

The **assets** subfolder, as its name implies, is where our application's static assets will reside, such as logos, images, and anything else that is static.

The **components** folder is where we will spend most of our time. Vue applications are made up of components, which can be anything from a complete page or smaller parts of a page. Components in Vue are a great way of organizing code (as we'll see later), which helps build scalable and maintainable applications.

On the application's root folder—referred within the VS Code project structure as **TEST**—there are also some important files that have been created by the Vue CLI or Vue UI, such as **package.json** and **babel.config.js**.

The **package.json** file contains specific information on what runtime and development dependencies the application uses, and other configuration details on how to serve, build, and lint the app.

Let's have a look at what the **package.json** file looks like.

Listing 2-a: The package.json File

```
{
  "name": "test",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
  "dependencies": {
    "core-js": "^2.6.5",
    "vue": "^2.6.10"
  },
  "devDependencies": {
    "@vue/cli-plugin-babel": "^3.8.0",
    "@vue/cli-plugin-eslint": "^3.8.0",
    "@vue/cli-service": "^3.8.0",
    "babel-eslint": "^10.0.1",
    "eslint": "^5.16.0",
    "eslint-plugin-vue": "^5.0.0",
    "vue-template-compiler": "^2.6.10"
  },
  "eslintConfig": {
    "root": true,
    "env": {
      "node": true
    },
    "extends": [
      "plugin:vue/essential",
      "eslint:recommended"
    ],
    "rules": {},
    "parserOptions": {
      "parser": "babel-eslint"
    }
  },
  "postcss": {
    "plugins": {
      "autoprefixer": {}
    }
  },
  "browserslist": [
    "> 1%",
```

```
    "last 2 versions"
  ]
}
```

The **babel.config.js** file contains configuration details that allow Babel to compile ECMAScript 2015+ code into a backwards-compatible version of JavaScript that can be executed by most browsers.

Listing 2-b: The babel.config.js File

```
module.exports = {
  presets: [
    '@vue/app'
  ]
}
```

Now that we've explored the most relevant parts of our project's folder structure, let's start modifying the application to what we want to build.

Index.html, main.js, and App.vue

Vue is a single-page application (SPA) framework. For our project this means that it loads one file, which is the **index.html** file found under the **public** folder.

This file contains a placeholder that will be used to automatically inject the Vue application into it during runtime—let's have a look.

Listing 2-c: The index.html File

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title>test</title>
  </head>
  <body>
    <noscript>
      <strong>...Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
```

```
    <!-- built files will be automatically injected -->
  </body>
</html>
```

The placeholder where the complied code will be injected is the **`div`** with the **`id`** of **`app`**—we can see this in the following code listing.

Listing 2-d: The div Placeholder (index.html)

```
<div id="app">
  <!-- This is where the compiled code will be injected -->
</div>
```

In the **src** folder root, there is a **main.js** file, which is basically the entry point for Vue—let's have a look at the contents of this file.

Listing 2-e: The main.js File

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

Vue.config.productionTip = false

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

The first **`import`** statement basically imports the Vue library. The second **`import`** statement imports the app's main component, which we will look at shortly.

Then we create a new **`Vue`** instance and mount that instance, which will be rendered inside the **`div`** element with the **`id`** of **`app`**.

Let's now explore the out-of-the-box content of the **App.vue** file also found within the **src** root folder.

Listing 2-f: The App.vue File

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

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

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

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

In Vue, files with the .vue extension are component files. This means that within a single file, we can find the HTML, the JavaScript and the style (CSS) for the component. These sections are clearly identified as follows.

Listing 2-g: Sections within a .vue File

```
<template>
  <!-- This is where the component's HTML resides -->
</template>

<script>
  <!-- This is where the component's JavaScript resides -->
</script>

<style>
  <!-- This is where the component's CSS resides -->
</style>
```

To get a better understanding of how this component is rendered, let's run the **serve** command from the **Project dashboard**.

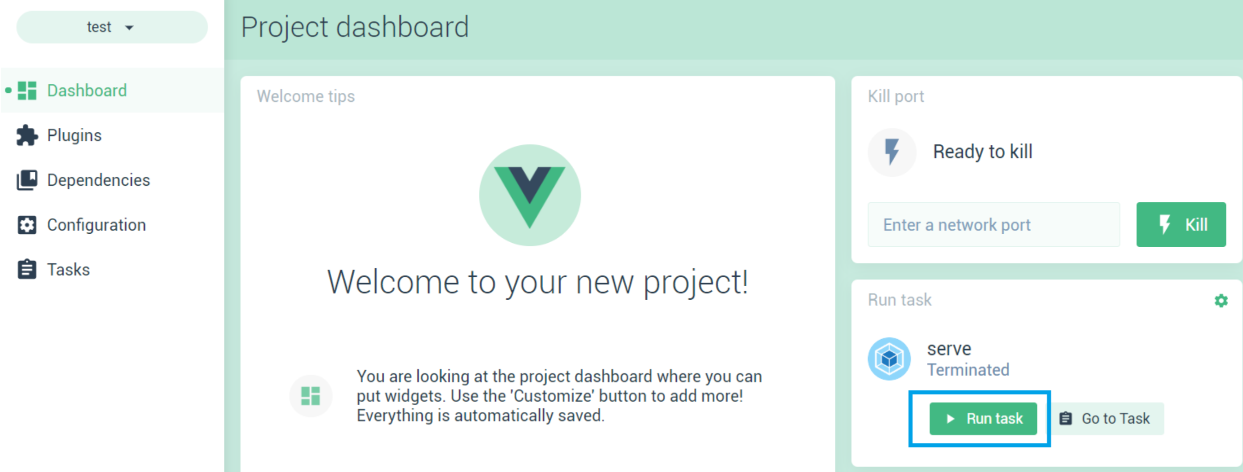


Figure 2-c: The serve Command (Project dashboard)

Click **Run task**, and then click **Go to Task** to get check the details of the application running within the Project dashboard. We can see this as follows.

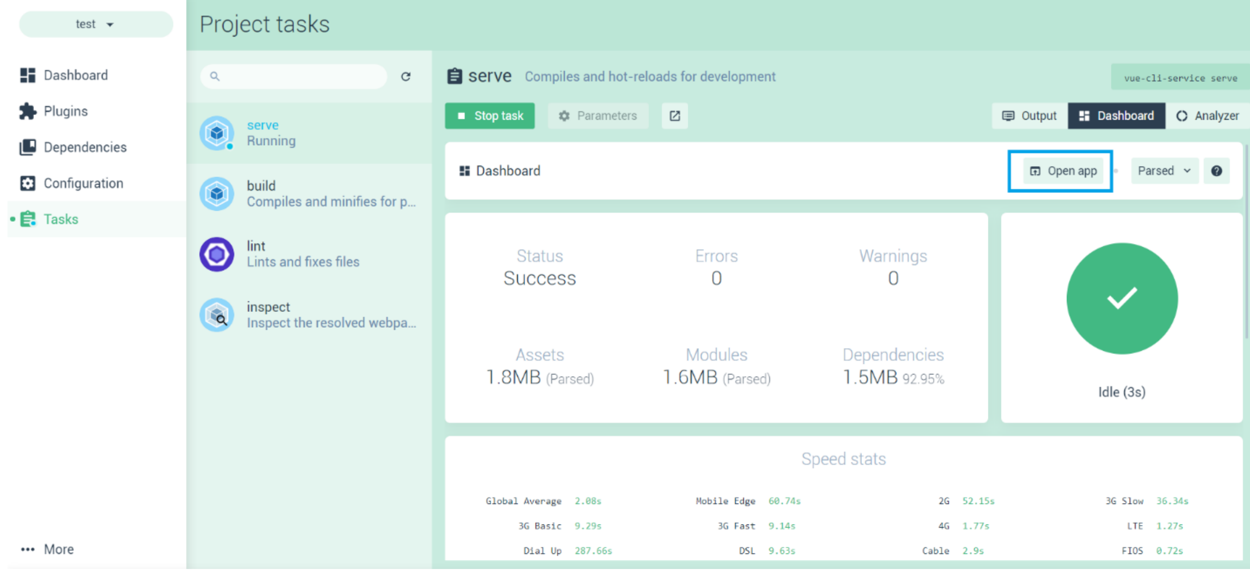


Figure 2-d: The App Runtime Details (Project dashboard)

To run the app, click **Open app**—this will open a new browser tab with the Vue application running.

In the following figure, I've put side by side VS Code with the **App.vue** source file, and the Vue app running on the browser, so we can see how Vue updates the app using hot reloading.

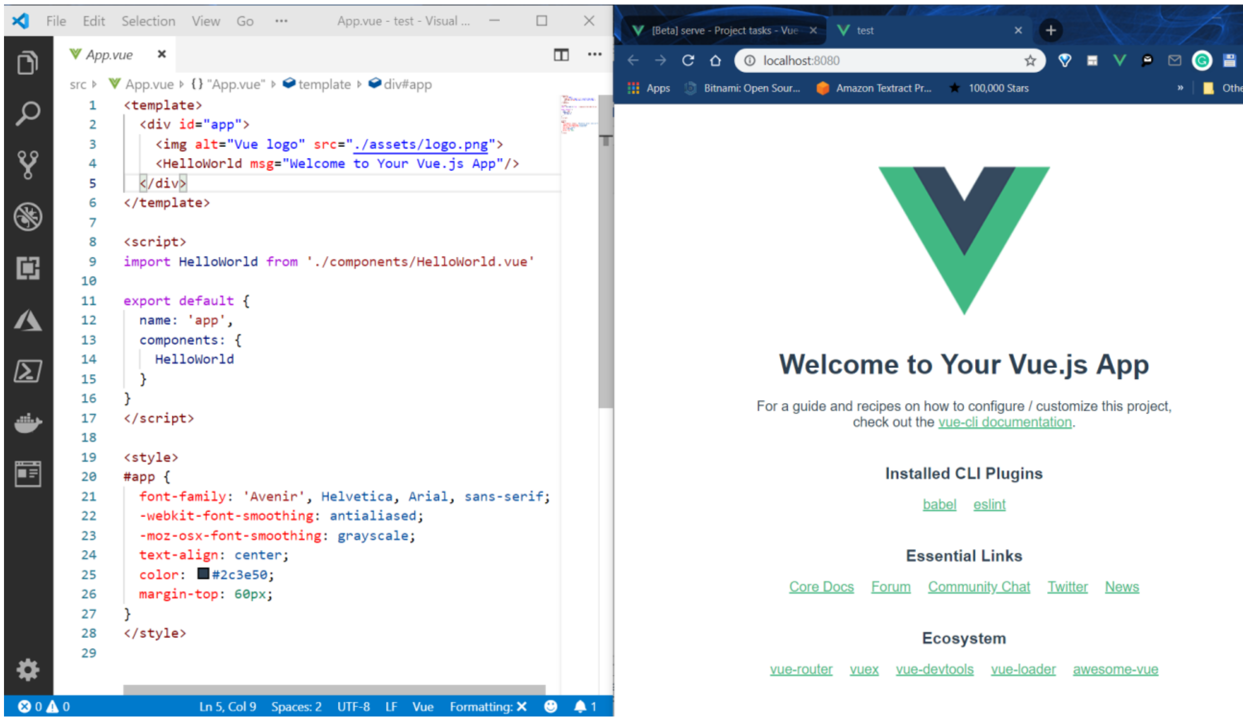


Figure 2-e: App.vue (Left) and The App Running (Right)

If I now make a change to **App.vue** and remove the Vue logo from the code, you'll notice that Vue performs a hot reload of the application, and you'll be able to see the changes getting applied immediately—which is awesome.

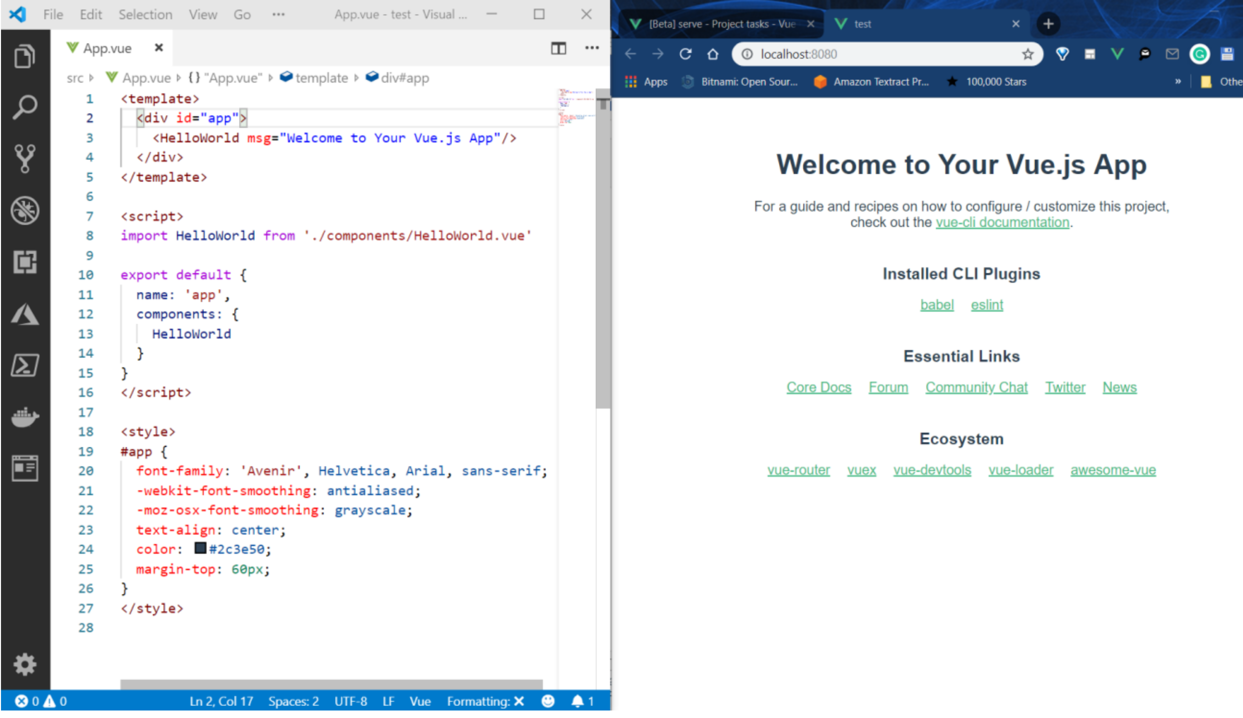


Figure 2-f: App.vue (Left) and the App Running (Right): Hot Reload

The HelloWorld component

Now that we have seen **index.html**, **main.js**, and **App.vue**, let's quickly talk about the **`HelloWorld`** component.

With the **App.vue** file, just below the line where we had the reference to the logo, is the reference to the **`HelloWorld`** component.

Listing 2-h: The template section of App.vue

```
<template>
  <div id="app">
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>
```

This **`HelloWorld`** component is imported within the script section of **App.vue** and then referenced within the **`components`** property of the **`export`** statement. We can see this as follows.

Listing 2-i: The script section of App.vue

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

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

The **`import`** statement is responsible for telling Vue that this component resides within the **components** source folder, and it is available within the **HelloWorld.vue** file—which we can see as follows.

Listing 2-j: HelloWorld.vue

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <p>
      For a guide and recipes on how to configure / customize
      this project,<br> check out the
```

```
      <a href="https://cli.vuejs.org" target="_blank" rel="noopener">vue-
    cli documentation</a>.
    </p>
    <h3>Installed CLI Plugins</h3>
    <ul>
      <li><a href="https://github.com/vuejs/vue-
        cli/tree/dev/packages/%40vue/cli-plugin-babel" target="_blank"
        rel="noopener">babel</a></li>
      <li><a href="https://github.com/vuejs/vue-
        cli/tree/dev/packages/%40vue/cli-plugin-eslint" target="_blank"
        rel="noopener">eslint</a></li>
    </ul>
    <h3>Essential Links</h3>
    <ul>
      <li>
        <a href="https://vuejs.org" target="_blank" rel="noopener">
        Core Docs</a>
      </li>
      <li><a href="https://forum.vuejs.org"
        target="_blank" rel="noopener">Forum</a></li>
      <li><a href="https://chat.vuejs.org" target="_blank"
        rel="noopener">Community Chat</a></li>
      <li><a href="https://twitter.com/vuejs" target="_blank"
        rel="noopener">Twitter</a></li>
      <li><a href="https://news.vuejs.org" target="_blank"
        rel="noopener">News</a></li>
    </ul>
    <h3>Ecosystem</h3>
    <ul>
      <li><a href="https://router.vuejs.org" target="_blank"
        rel="noopener">vue-router</a></li>
      <li><a href="https://vuex.vuejs.org" target="_blank"
        rel="noopener">vuex</a></li>
      <li><a href="https://github.com/vuejs/vue-devtools#vue-devtools"
        target="_blank" rel="noopener">vue-devtools</a></li>
      <li><a href="https://vue-loader.vuejs.org" target="_blank"
        rel="noopener">vue-loader</a></li>
      <li><a href="https://github.com/vuejs/awesome-vue" target="_blank"
        rel="noopener">awesome-vue</a></li>
    </ul>
  </div>
</template>

<script>
```

```
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
h3 {
  margin: 40px 0 0;
}
ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  margin: 0 10px;
}
a {
  color: #42b983;
}
</style>
```

As we can see, the **`HelloWorld`** component is structured the same way as **App.vue**—which means that it has a template section for the HTML markup, a script section that contains the JavaScript code, and a style section for the CSS.

There are a couple of interesting things beyond the boilerplate HTML and CSS markup code, which I would like to explain.

The first one is **`{{ msg }}`** within the **`h1`** tag—known as declarative rendering in Vue—which allows us to render data to the DOM using a template syntax.

Notice how the **`msg`** variable has been declared within the script section inside **`props`**—this is because **`msg`** is passed as a parameter to the **`HelloWorld`** component within **App.vue**. The following diagram illustrates this better.

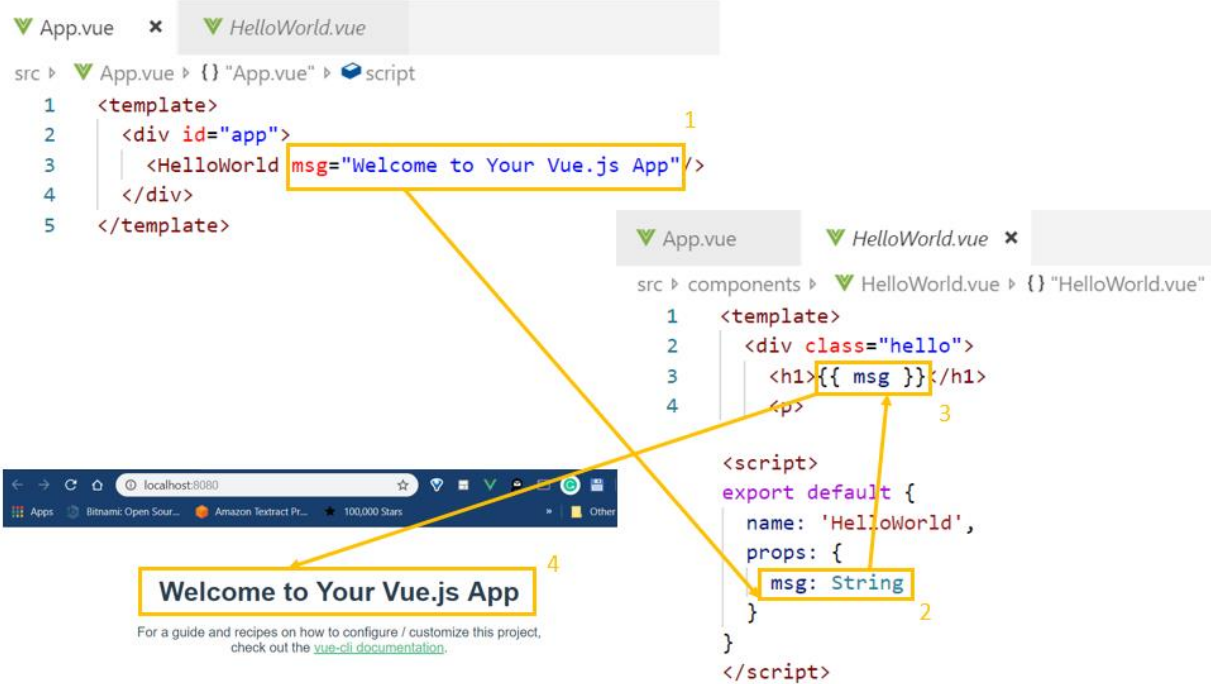


Figure 2-g: Declarative Rendering Props in Vue

As you can see, **`msg`** is passed as a parameter to the **`HelloWorld`** component—this is possible because **`msg`** is defined within the component's properties (**`props`**). Then **`msg`** is rendered on the DOM using the template syntax: **`{{ msg }}`**, which displays the text on the UI.

The other interesting feature of the **`HelloWorld`** component is the **`scoped`** attribute for the style tag, which limits the CSS to be specific for that component only, and is not available to other components that are part of the application. This is very useful as individual components can have their own style.

The rest of the HTML markup with the template tag of the **`HelloWorld`** component is just boilerplate code, which we can remove—so let's go ahead and do that. This is how the **`HelloWorld`** component code looks now:

Listing 2-k: Modified HelloWorld.vue

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
```

```
    msg: String
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
h3 {
  margin: 40px 0 0;
}
ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  margin: 0 10px;
}
a {
  color: #42b983;
}
</style>
```

Another thing that is very important is that within the template section there can only be one main element—the following listing describes the correct way of doing it.

Listing 2-l: Correct Main Element (template section)

```
<!-- This is correct -->
<template>
  <div class="hello">
  </div>
</template>
```

The following listing describes the incorrect way of doing it—there cannot be two main elements within a template section.

Listing 2-m: Incorrect Main Element (template section)

```
<!-- This is incorrect -->
<template>
  <div class="hello">
  </div>
```

```
  <div class="hello">
  </div>
</template>
```

So, within any template section there can only be one main element. If you go to the **App.vue** file, you will notice that this is also the case—there's only one main element within the template section.

App component structure

One of the fundamental aspects of any Vue application is how the app is structured and how the components that make up the application relate to each other, which is what we are going to explore next.

But before we do that, let's clean things up a bit and get rid of boilerplate code within **App.vue**, so that we are left with the following.

Listing 2-n: Modified App.vue

```
<template>
  <div id="app">
  </div>
</template>

<script>

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

<style>
</style>
```

Before we start to add any more code to **App.vue**, let's set the stage for what we are going to build.

As mentioned earlier, we are going to build a web-based Vue equivalent of the mobile-based app that was written in *Flutter Succinctly*. This is an app that will keep track of important documents that have an expiry date, such as passports and credit cards. The following figure shows what the finished Flutter app looks like.

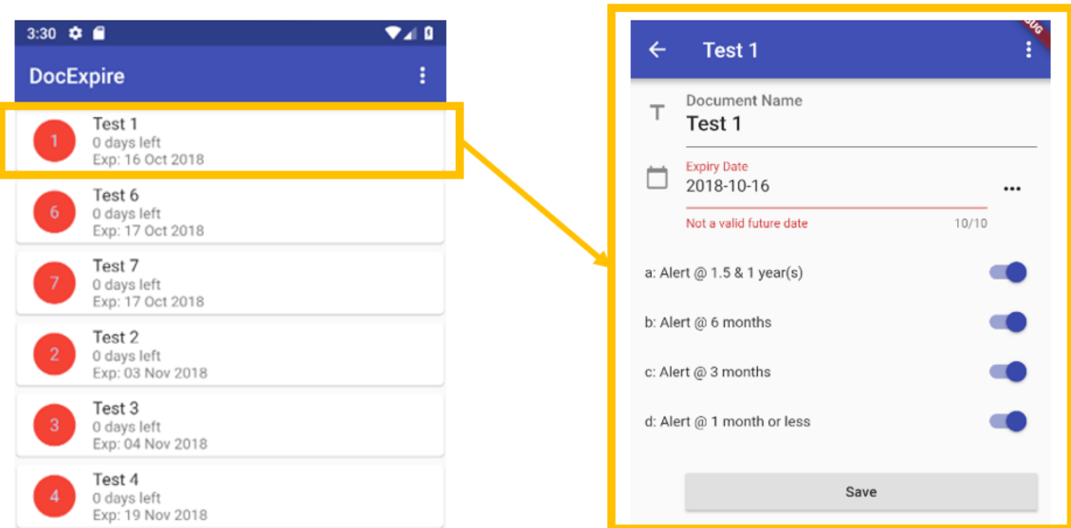


Figure 2-h: The Finished Flutter App

This app essentially has two screens: the main layout, which includes the list of all the documents that the application keeps track of, and a secondary screen, which is used to enter a new document or edit existing ones.

If we think of this application in terms of Vue components, we can come up with a component structure that looks as follows.

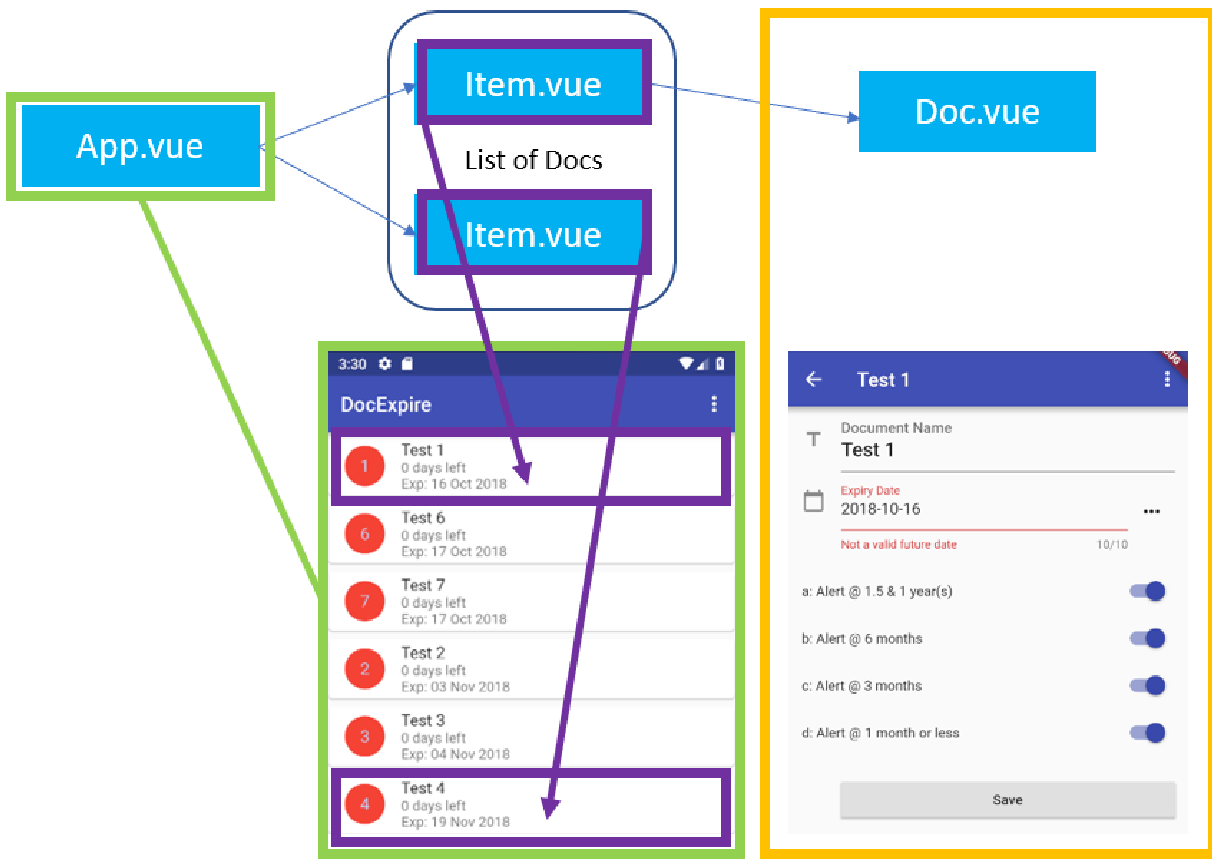


Figure 2-i: Relationships between Vue Components in the Finished Flutter App

By looking at the preceding diagram, we can see that **App.vue** is going to be the app's main component file, which will display the list of documents (**Docs.vue**)—each of the documents on that list is going to be displayed as an item, through the **Item.vue** component file.

When an item is clicked, the document details will be displayed using **Doc.vue**—which will also be used when a new document is created.

Now that we know how our application will be structured from a component's functional point of view, let's talk about how we can organize the data related to this structure within **App.vue**.

Vue's data within components

Within a Vue component, data is not provided as an object, but instead as a function—this is done so that each instance can maintain an independent copy of the returned data object. So, within a Vue component, data is not defined like this:

Listing 2-o: How Data is Not Defined (Within a Vue Component)

```
export default {
  ...
```

```
  data: {
    items: []
  }
}
```

But instead like this:

Listing 2-p: How Data is Defined (Within a Vue Component)

```
export default {
  ...
  data: () => {
    return {
      items: []
    }
  }
}
```

Now that we know how to define data in Vue components, let's add some boilerplate data so that we can start to scaffold our application as it will eventually look and end up like. To do that, let's add the following code to **App.vue**, which I've highlighted in bold.

Listing 2-q: Adding Boilerplate Data to App.vue

```
<template>
  <div id="app">
  </div>
</template>

<script>

export default {
  name: 'app',
  components: {
  },
  data: () => {
    return {
      items: [
        {
          id: 1,
          name: "Test 1",
          exp: "16 Oct 2019"
        },
```

```
            {
              id: 2,
              name: "Test 2",
              exp: "16 Nov 2019"
            }
          ]
        }
      }
    }
</script>
<style>
</style>
```

Now that we've defined some data boilerplate code, let's wrap this around a component, which we can use to display this data.

Docs component

Under the **components** folder within our application (which is the same folder that contains **HelloWorld.vue**), let's create the **Docs.vue** file, which we will use to display the list of documents and embed this within the **App.vue** markup.

Listing 2-r: Docs.vue

```
<template>
    <div>
        <h1>DocExpire</h1>
    </div>
</template>

<script>
export default {
    name: "Docs"
}
</script>

<style scoped>

</style>
```

Now we can add a reference to the `Docs` component and embed it within **App.vue** as follows.

Listing 2-s: Updated App.vue referencing Docs

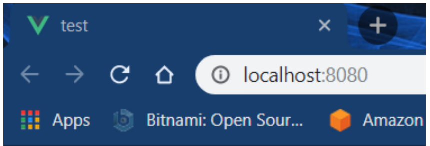
```
<template>
  <div id="app">
    <Docs />
  </div>
</template>

<script>
import Docs from './components/Docs';

export default {
  name: 'app',
  components: {
    Docs
  },
  data: () => {
    return {
      items: [
        {
          id: 1,
          name: "Test 1",
          exp: "16 Oct 2019"
        },
        {
          id: 2,
          name: "Test 2",
          exp: "16 Nov 2019"
        }
      ]
    }
  }
}
</script>
<style>
</style>
```

The parts in bold are the references added for the **`Docs`** component. Notice how it was added to the template section as a single HTML entity: **`<Docs />`**. Then, within the scripts section, it was imported using the statement **`import Docs from './components/Docs'`**. Finally, it was added to the **`components`** object as **`Docs`**.

If you have your application still running, Vue will have reloaded it after making those changes and you should be able to see the following on the screen.



DocExpire

Figure 2-j: The App Running (After Changes)

If your app is running, you can run `npm run serve` from the command line, or run the app using the Vue UI as previously indicated.

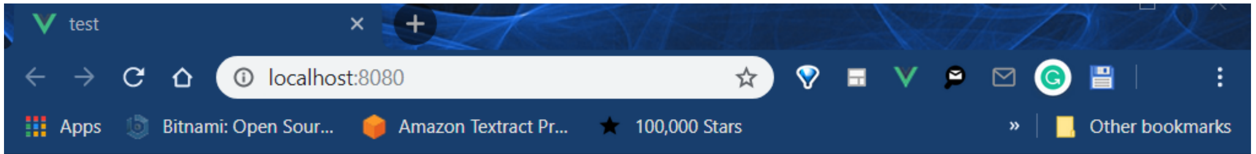
Vue DevTools

This is a good time to talk about the **[Vue DevTools]** browser extension, which might come in handy during development, and I would recommend you to install it.

Once your have installed the extension, you'll see the Vue DevTools icon on your browser. Let's check it out.

Figure 2-k: The Vue DevTools Browser Icon

On Chrome, open the **Developer tools** (Ctrl+Shift+I on Windows) on the same page where the app is running. Once the **Developer tools** are opened, click on the **Vue** tab. This is how it looks on my machine.



DocExpire

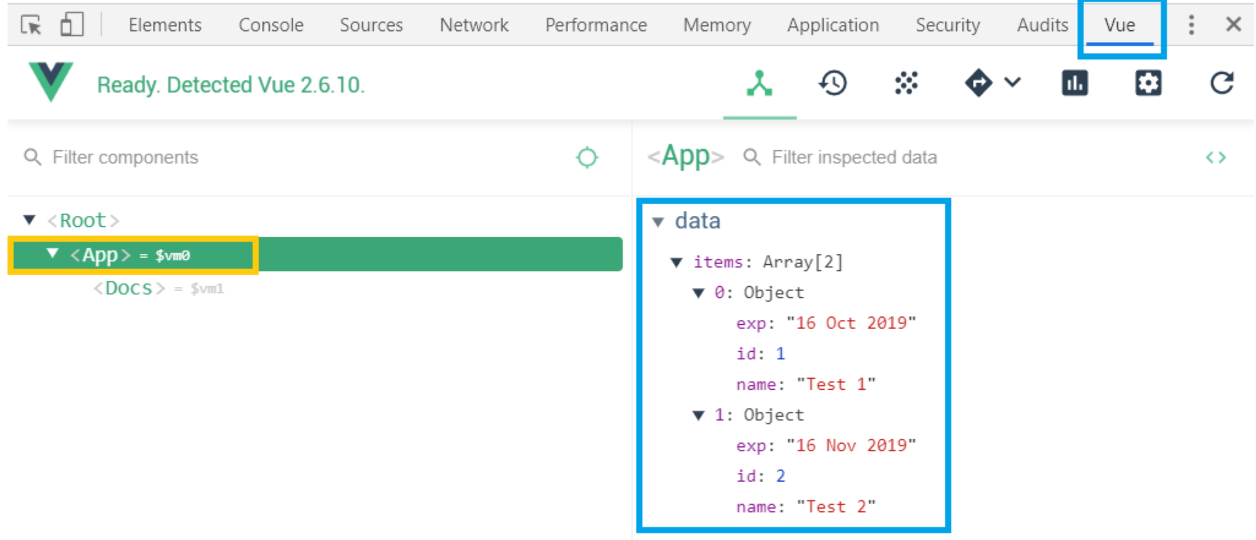


Figure 2-l: The Vue DevTools Opened

Vue DevTools is a great browser extension to use when you're developing Vue apps. Not only does it help inspecting Vue components and their data—like we can see in the previous figure—but it can also be used to filter events, check state and routing, and inspect component rendering and frames per second.

Notice how when we're using Vue DevTools, we can see the data contained within the **`App`** component: the **`items`** array and each of its elements.

Using Vue DevTools is quite straightforward, and we'll cover some parts of it as we go. If you would like to know more details about it, here's a good tutorial.

Getting the data into the Docs component

Now that we have seen briefly how to use Vue DevTools to inspect data, the goal is to get the data from those documents into the **`Docs`** component.

We can do this by using a directive called **`v-bind`**, which is used for data binding. Let's have a look at how to implement this in App.vue.

Listing 2-t: Updated App.vue (Docs with v-bind)

```
<template>
  <div id="app">
    <Docs v-bind:items="items"/>
  </div>
</template>

// The part of the code that follows remains the same
```

Notice that by passing **`v-bind:items="items"`** to **`Docs`**, what we are saying is that we are binding the **`items`** array returned by the **`data`** function to a property called **`items`**—which we still have to create—within the **`Docs`** component.

So, to do that, let's go over to **Docs.vue** and make the following modifications to the code.

Listing 2-u: Updated Docs.vue (items property added)

```
// The previous part of the code remains the same

<script>
export default {
    name: "Docs",
    props: ["items"]
}
</script>

// The part of the code that follows remains the same
```

Now that we have declared the **`items`** property, we need to be able to display each of the elements of the **`items`** array passed to the **`Docs`** component. To do that, we need to loop through them—this is done in Vue with another directive called **`v-for`**.

Listing 2-v: Updated Docs.vue (v-for added)

```
<template>
    <div>
        <h1>DocExpire</h1>
        <div v-for="item in items">
          <h3>{{item.name}}</h3>
          <p>{{item.exp}}</p>
        </div>
    </div>
</template>

<script>
```

```
export default {
    name: "Docs",
    props: ["items"]
}
</script>

<style scoped>
</style>
```

What this means is that for each **`item`** in the **`items`** array, passed as the **`items`** property to the **`Docs`** component, we are displaying the **`item.name`** and **`item.exp`** properties of each **`item`** object.

If we saved the changes in VS Code and we look at the app running on the browser, we should see the following.

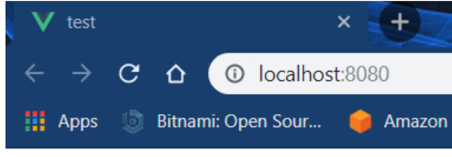


Figure 2-m: The Updated Vue App (List of Docs)

To have a better visual understanding of what we have just done with both the **`v-bind`** and **`v-for`** directives in **App.vue** and **Docs.vue** respectively, let’s have a look at the following diagram.

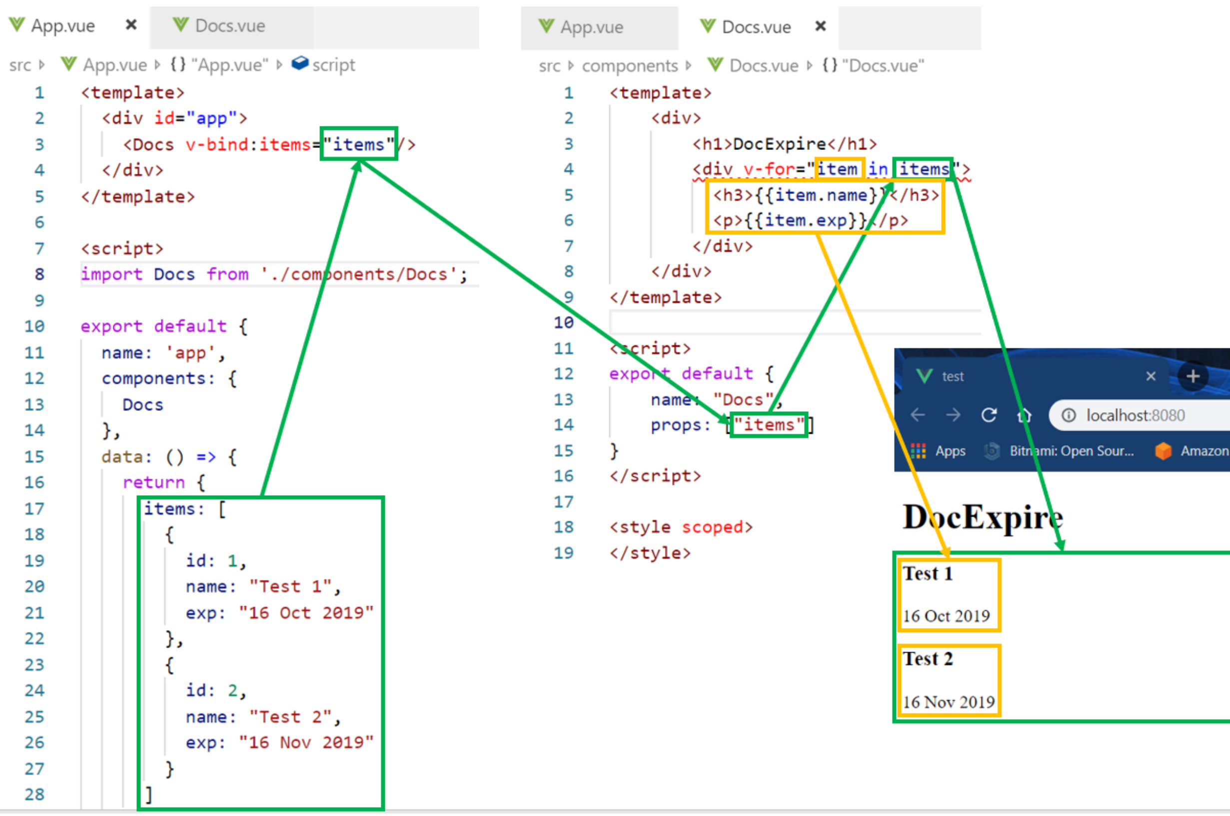


Figure 2-n: Relationship between v-bind (App.vue) and v-for (Docs.vue)

You can clearly see how the **`items`** array returned by the **`data`** function in **App.vue** binds to the **`Docs`** component and is passed as a property. Those **`items`** are looped and rendered using the **`v-for`** directive in **Docs.vue**.

If you installed the Vetur extension, you might have noticed the following syntax issue after you added the **`v-for`** directive.

```
<template>
    <div>
        <h1>DocExpire</h1>
        <div v-for="item in items">
          <h3>{{item.name}}</h3>
          <p>{{item.exp}}</p>
        </div>
    </div>
</template>
```

Figure 2-o: Syntax Issue v-for Directive

This is because we need to specify a key, which is a similar concept used in the React framework. We can achieve this by binding to a key as follows.

```
<template>
    <div>
        <h1>DocExpire</h1>
        <div v-bind:key="item.id" v-for="item in items">
          <h3>{{item.name}}</h3>
          <p>{{item.exp}}</p>
        </div>
    </div>
</template>
```

Figure 2-p: Syntax Issue v-for Directive with v-bind:key

Notice how the syntax error goes away once we specify **`item.id`** as the key, using **`v-bind:key`** directive.

Summary

We've covered quite a bit of ground and managed to lay out the foundations of our application, but we still need to add quite a lot of logic and further functionality.

In the next chapters, we'll dig deeper and add application-specific functionality, and the rest of the components that our application will need, to create our web-based document expiration tool.

Chapter 3 Expanding the App: UI

Quick intro

In the previous chapter, we laid the foundation for our application by looking at its component structure, and looked at some of the fundamental constructs of Vue and useful directives, which are commonly used throughout most Vue applications. We also created the `Docs` component and made it work with the boilerplate data within **App.vue**, which we will later make dynamic.

We are now able to take these concepts further and add the remaining components our application requires—which is what we'll do in this chapter.

Item.vue

To keep our code clean and organized, let's take a step further and add a new component file called **Item.vue**, which will be responsible for displaying the information of an individual document. The `Item` component will be invoked from the `Docs` component.

Using VS Code, let's go ahead and add the **Item.vue** file under the **Components** folder. I've added the standard boilerplate code to it, which looks as follows.

Listing 3-a: Item.vue

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

<script>
export default {
    name: "Item"
}
</script>

<style scoped>
</style>
```

So far, it's nothing out of the ordinary, as you can see. Let's go back to **Docs.vue** and import the `Item` component and bind it to the existing data.

Listing 3-b: Updated Docs.vue

```
<template>
    <div>
        <h1>DocExpire</h1>
        <div v-bind:key="item.id" v-for="item in items">
          <Item v-bind:item="item"/>
        </div>
    </div>
</template>

<script>
import Item from './Item';

export default {
    name: "Docs",
    props: ["items"],
    components: {
        Item
    }
}
</script>

<style scoped>
</style>
```

Notice the changes to the **`Docs`** component highlighted in bold. We have now added a reference to the **`Item`** component directly within the markup: **`<Item v-bind:item="item"/>`**.

Notice how we are passing the **`item`** current object from the **`items`** array and binding that to the **`item`** property of the **`Item`** component.

Also notice how we have imported the **`Item`** component (**Item.vue**) using the **`import`** statement and declared it within the **`components`** object within **Docs.vue**.

Now let's make the required modifications to **Item.vue** to accommodate for these changes.

Listing 3-c: Updated Item.vue

```
<template>
    <div>
        <h3>{{item.name}}</h3>
        <p>{{item.exp}}</p>
    </div>
</template>
```

```
<script>
export default {
    name: "Item",
    props: ["item"]
}
</script>

<style scoped>
</style>
```

As you can see, we’ve added the **`{{item.name}}`** expression to the HTML markup and declared the **`item`** property, which we pass from **Docs.vue**.

To completely understand what we’ve just done, let’s look at the following diagram.

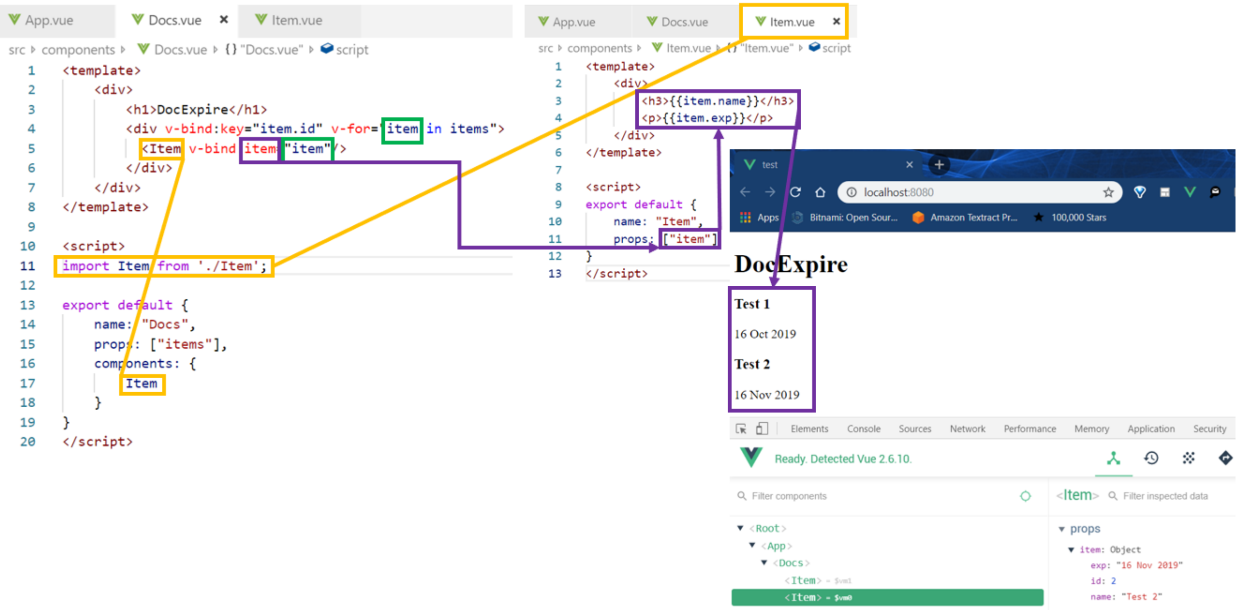


Figure 3-a: Relationship between Docs.vue and Item.vue

Now that we understand how both components relate to each other, we need find a visual way to differentiate between documents that have expired and those that are still valid (expire in the future).

To do that, we’ll have to add some CSS styling and make use of logic that shows one icon or another, depending whether the document has expired or not. But before we do that, let’s add some style to our application by using Google’s awesome Material Design library.

Vuetify

I'm a big fan of Material Design, and while writing *Flutter Succinctly*, I was pleased that Material Design was provided out of the box with the Flutter framework.

With Vue, Material Design is not part of the framework; however, there's a Material Design Component framework that works great with Vue called Vuetify—which is what we'll be using.

Before installing Vuetify, please back up your existing **App.vue** file, as Vuetify might overwrite it. To install Vuetify, all you need to do is run the following command on your project root folder.

Listing 3-d: How to Install Vuetify

```
vue add vuetify
```

Once you have executed the command, you will be requested to choose a preset, with the following options.

```
$ ? Choose a preset: (Use arrow keys)
$ > Default (recommended)
$   Prototype (rapid development)
$   Configure (advanced)
```

Figure 3-b: Vuetify Presets

In my case, I simply chose the **Default (recommended)** preset. Once you've chosen your preset option, Vuetify will be added to the project and the **package.json** file will be updated with a reference to it.

Once installed, the next thing we need to do is to add it to our existing application, so we can start to benefit from the Material Design styling.

To do that, open the **main.js** file found within the **src** folder of your project. Once you've opened **main.js**, add the following lines to the file.

```
import Vuetify from 'vuetify'
Vue.use(Vuetify)
```

This has been highlighted in bold the code listing below, which corresponds to the **main.js** file within the project **src** folder.

Listing 3-e: Updated main.js (Adding Vuetify)

```
import Vue from 'vue'
import './plugins/vuetify'
import App from './App.vue'
import Vuetify from 'vuetify'
```

```
Vue.use(Vuetify)
Vue.config.productionTip = false

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

Now, we are ready to give a Material Design look and feel to our application. Vuetify is fully enabled and ready to be used.

Styling App.vue

Besides making our application look prettier, one of the key aspects of choosing Vuetify is to try to make the application look as close as possible to the one developed in *Flutter Succinctly*.

One of the distinct features about the Flutter app is that it includes a floating button, which is used to add new documents to the application.

Based on that concept, let’s modify our application accordingly, so we can end up with something that looks like this.

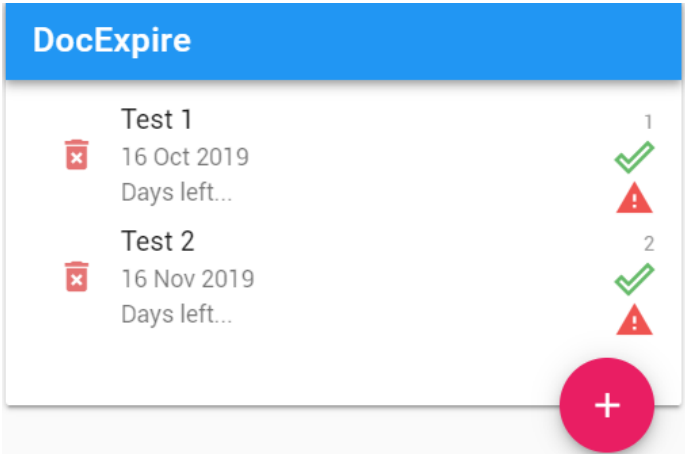


Figure 3-c: Our App’s UI (Using Vuetify)

Let’s start off by modifying **App.vue** with the following code.

Listing 3-f: Updated App.vue (Using Vuetify)

```
<template>
  <div id="app">
    <v-app>
```

```
      <v-layout row>
        <v-flex xs12 sm6 offset-sm3>
          <v-card>
            <Docs :items="items"/>
            <v-card-text tyle="height: 100px; position: relative">
              <v-btn
                big
                color="pink"
                dark
                absolute
                bottom
                right
                fab
              >
                <v-icon>add</v-icon>
              </v-btn>
            </v-card-text>
          </v-card>
        </v-flex>
      </v-layout>
    </v-app>
  </div>
</template>

<script>
import Docs from './components/Docs';

export default {
  name: 'app',
  components: {
    Docs
  },
  data: () => {
    return {
      items: [
        {
          id: 1,
          name: "Test 1",
          exp: "16 Oct 2019"
        },
        {
          id: 2,
          name: "Test 2",
          exp: "16 Nov 2019"
```

```
        }
      ]
    }
  }
}
</script>

<style>
</style>
```

I've highlighted in bold the parts of the code that have been added. As you can see, it's basically HTML markup that has changed. The rest of the **App.vue** code remains the same.

Notice the **<Docs :items="items"/>** syntax instead of **<Docs v-bind:items="items"/>**. This is because **v-bind:items** can be shortened to simply **:items**. From now on, to make the syntax simpler, we'll use the **:** shortcut syntax instead of **v-bind**.

One great thing about Vuetify is that it comes with prebuilt Material Design components, which make the development of the app's UI much faster and easier than using regular HTML markup.

To better understand the markup added to **App.vue**, and the UI that is visible on the screen, let's look at the following diagram.

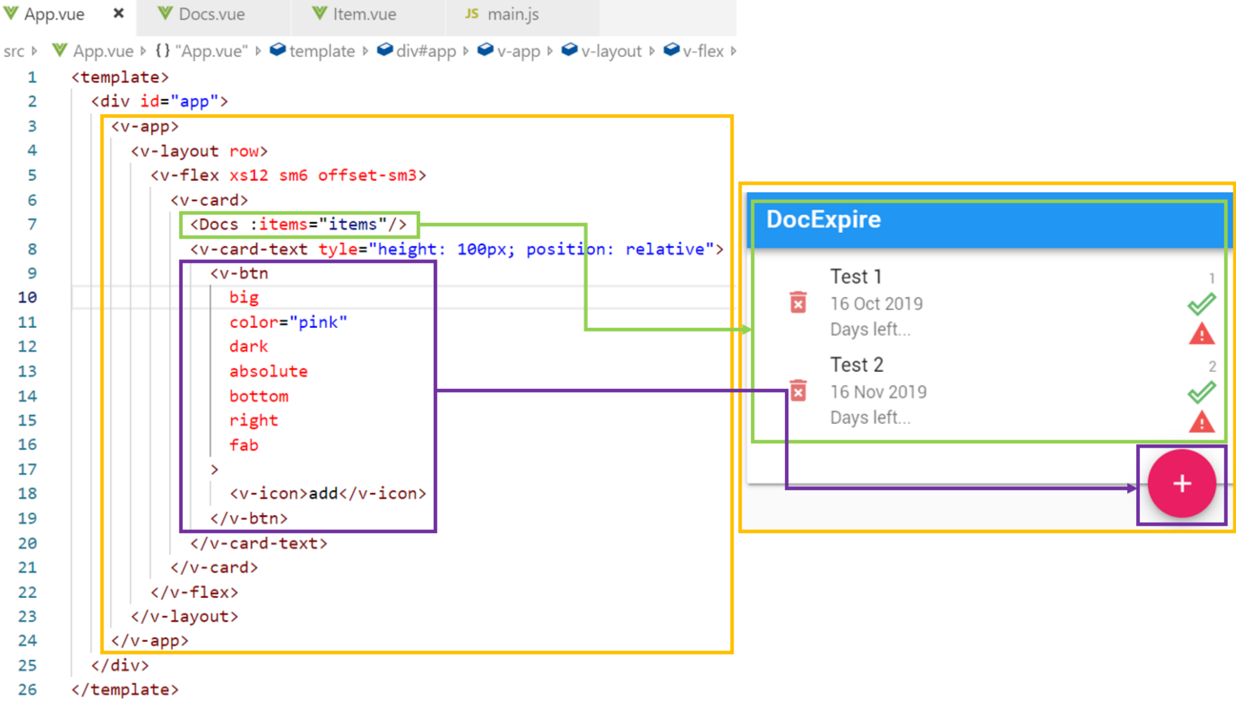


Figure 3-d: Relationship between App.vue and the UI

As you can see, there are two main distinct UI areas: one is the **Docs** component (highlighted in green), which encapsulates most of the UI, and then the floating button area (highlighted in purple).

The area highlighted in yellow in Figure 3-d defines the layout that is used to render the application; this is the reason why the **`v-layout`**, **`v-flex`**, and **`v-card`** Vuetify components are used—they define the point grid system Vuetify uses.

Styling Docs.vue

Just like we've done with **App.vue**, we need to do the same with **Docs.vue** and style it accordingly using Vuetify. Let's go ahead and do that.

Listing 3-g: Updated Docs.vue (Using Vuetify)

```
<template>
    <div>
        <v-toolbar color="blue" dark>
            <v-toolbar-title>DocExpire</v-toolbar-title>
            <v-spacer></v-spacer>
        </v-toolbar>

        <v-list two-line>
            <template v-for="item in items">
                <Item :key="item.id" :lgth="items.length" :item="item"/>
            </template>
        </v-list>
    </div>
</template>

<script>
import Item from './Item';

export default {
    name: "Docs",
    props: ["items"],
    components: {
        Item
    }
}
</script>

<style scoped>
</style>
```

As you might have noticed, we added the **`v-toolbar`** and **`v-list`** components to **Docs.vue** to give it the Material Design look and feel.

The following diagram illustrates the relationship between the **Docs.vue** markup and the UI. Let’s have a look.

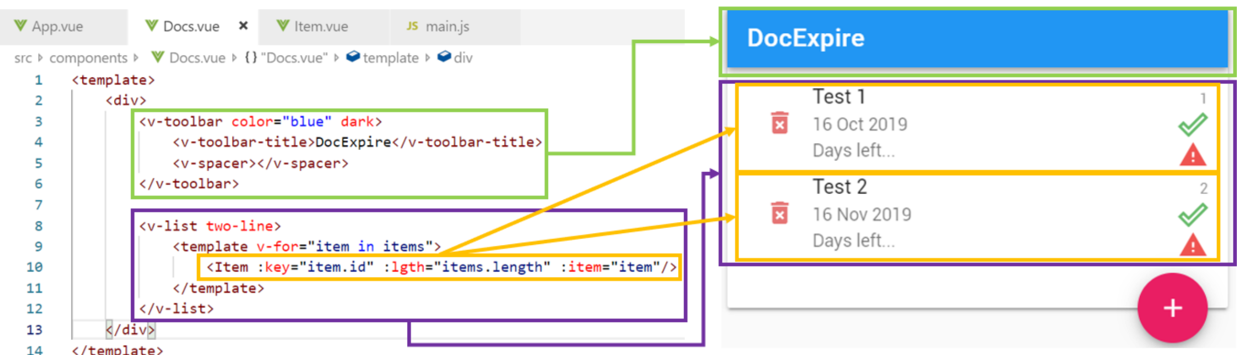


Figure 3-e: Relationship between Docs.vue and the UI

The markup is quite self-explanatory: the **`v-toolbar`** component creates the **DocExpire** toolbar, and the **`v-list`** component creates the list that will contain each of the **`Item`** components.

Styling Item.vue

Now that we’ve added Vuetify components to **App.vue** and **Docs.vue**, let’s style **Item.vue** as well. Here’s what the updated code looks like.

Listing 3-h: Updated Item.vue (Using Vuetify)

```
<template>
    <div>
        <v-list-tile avatar ripple>
            <v-btn flat icon color="red lighten-2">
                <v-icon>delete_forever</v-icon>
            </v-btn>
            <v-list-tile-content>
                <v-list-tile-title class="text--primary">
                    {{ item.name }}
                </v-list-tile-title>
                <v-list-tile-sub-title>
                    {{ item.exp }}
                </v-list-tile-sub-title>
                <v-list-tile-sub-title>
                    Days left...
                </v-list-tile-sub-title>
            </v-list-tile-content>
            <v-list-tile-action>
                <v-list-tile-action-text>
```

```
                        {{ item.id }}
                    </v-list-tile-action-text>
                    <v-icon color="green lighten-1">done_outline</v-icon>
                    <v-icon color="red lighten-1">warning</v-icon>
                </v-list-tile-action>
            </v-list-tile>
            <v-divider v-if="item.id + 1 < lgth"
                       :key="`divider-${item.id}`">
            </v-divider>
        </div>
</template>

<script>
export default {
    name: "Item",
    props: ["item", "lgth"]
}
</script>

<style scoped>
</style>
```

The **`Item`** component contains the markup logic that displays each of the documents on the list, so to understand this better, let's have a look at the following diagram that explains the relationship between the markup and the UI.

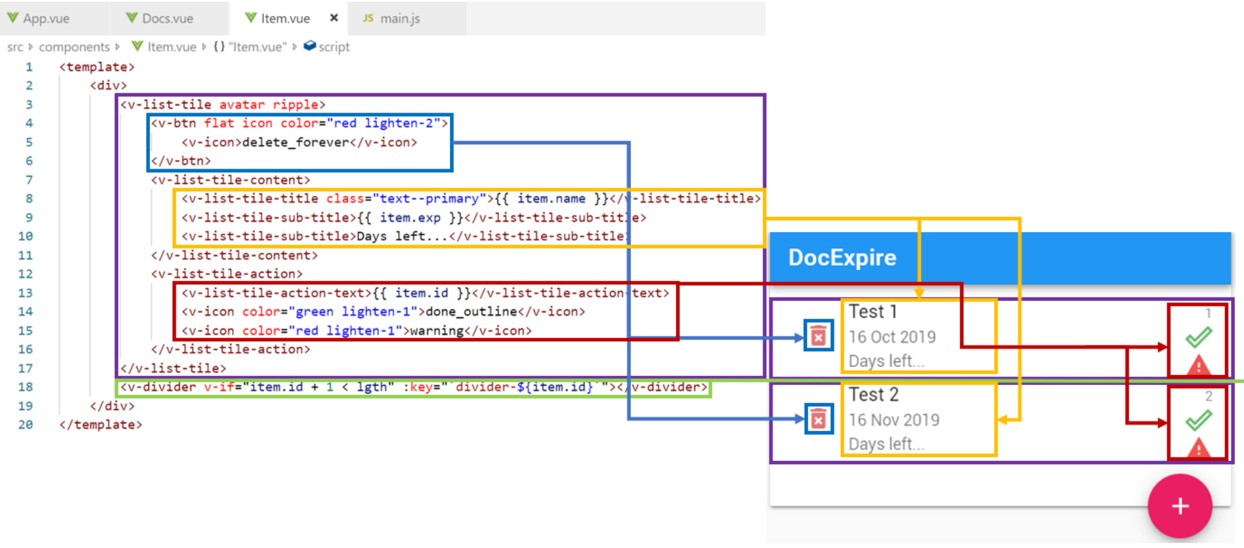


Figure 3-f: Relationship between Item.vue and the UI

As we can see, the **`v-btn`** component is responsible for creating the Delete button that appears on the left side of each document. This is highlighted in blue in Figure 3-f.

There is a **`v-list-tile-content`** component (highlighted in yellow) that encapsulates and displays the name of the document (**`item.name`**), the document's expiration (**`item.exp`**), and the number of days left, which for now is static text, but we'll modify it and make it dynamic later.

Finally, there is a **`v-list-tile-action`** component (highlighted in red) that encapsulates and displays the icons seen to the right of the text, for each document.

Notice that between each document (**`item`**), there is a **`v-divider`** component (highlighted in green). This divider is related to the **`Item`** component by the **`item.id`** property.

You've also probably noticed that there's an additional property being passed from the code in **Docs.vue** to the **`Item`** component. This is the **`lgth`** property, which is used for adding a separator between each document.

```
<template v-for="item in items">
    <Item :key="item.id" :lgth="items.length" :item="item"/>
</template>
```

Figure 3-g: The lgth Property passed to the Item component

The **`lgth`** property is nothing more than the length of the **`items`** array, which indicates the number of documents that exist and are returned by the **`data`** function with **App.vue**.

The reason that we pass the **`lgth`** property to the **`Item`** component is that we only want to display the divider between items. So, if there are two items to display, there should be only one divider. This can be achieved by using the **`v-if`** directive with the following condition.

```
<v-divider v-if="item.id + 1 < lgth" :key="`divider-${item.id}`"></v-divider>
```

Figure 3-h: The v-if Directive Used for Showing the Divider

So, the divider will only be shown when **`item.id + 1`** is less than the value of **`lgth`** (number of documents).

Now that we have styled our existing components, we still need to create a component that we can use when creating a new document or when modifying an existing one.

Creating Doc.vue

The component file that will be responsible for creating a new document or modifying an existing one is going to be called **Doc.vue**. So, in VS Code, go ahead and create this file under the **components** folder of your project.

To get a sense of what we will create, the figure that follows shows what the finished component's UI looks like.

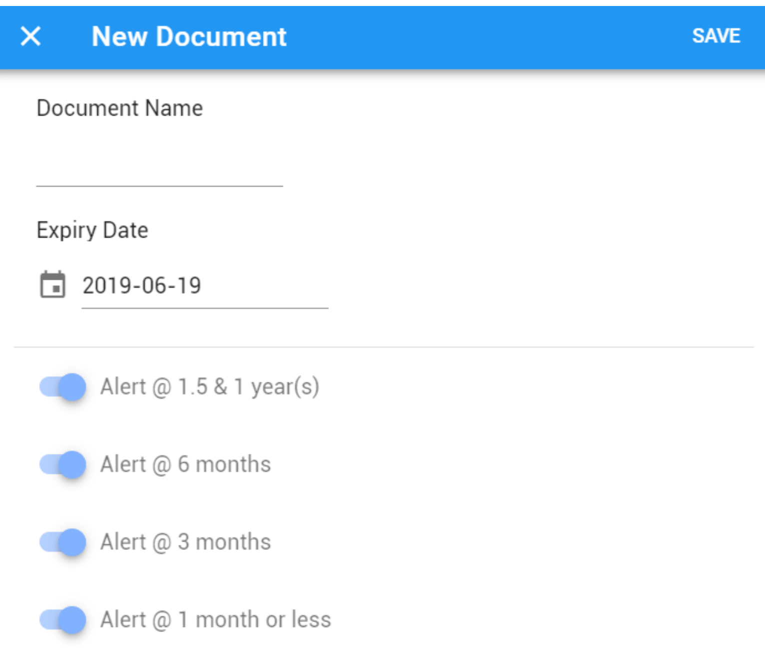


Figure 3-i: The Finished UI of the Doc Component

As you can see, this is UI is almost identical to the one from the Flutter application—from a functional point, it will be the same.

Now that we've seen the finished UI, let's have a look at the code required to build it, which is shown in the listing that follows.

Listing 3-i: Doc.vue (Using Vuetify)

```
<template>
    <div>
        <v-dialog v-model="showModal"
          fullscreen
          hide-overlay
          transition="dialog-bottom-transition"
          scrollable
        >
            <v-card tile>
                <v-toolbar color="blue" dark>
                <v-btn icon dark @click="hide">
                    <v-icon>close</v-icon>
                </v-btn>
                <v-toolbar-title>{{item.name}}</v-toolbar-title>
                <v-spacer></v-spacer>
                <v-toolbar-items>
```

```
        <v-btn dark flat @click="save">Save</v-btn>
    </v-toolbar-items>
    </v-toolbar>
    <v-card-text>
    <v-list three-line subheader>
        <v-list-tile avatar>
        <v-list-tile-content>
            <v-list-tile-title>
                Document Name
            </v-list-tile-title>
            <v-list-tile-sub-title></v-list-tile-sub-title>
            <v-text-field
                ref="name"
                v-model="name"
                :rules="[() => !!name || 'Required field']"
                required
                >
            </v-text-field>
        </v-list-tile-content>
        </v-list-tile>
        <v-list-tile avatar>
        <v-list-tile-content>
            <v-list-tile-title>
                Expiry Date
            </v-list-tile-title>
            <v-list-tile-sub-title></v-list-tile-sub-title>
            <v-menu
                ref="menu"
                v-model="menu"
                :close-on-content-click="false"
                :nudge-right="40"
                :return-value.sync="date"
                lazy
                transition="scale-transition"
                offset-y
                full-width
                min-width="290px"
                >
                <template v-slot:activator="{ on }">
                    <v-text-field
                    v-model="date"
                    prepend-icon="event"
                    readonly
                    v-on="on"
```

```
                    ></v-text-field>
                </template>
                <v-date-picker
                    v-model="date"
                    :min=date
                    max="2099-12-31"
                    no-title scrollable>
                    <v-spacer></v-spacer>
                    <v-btn flat color="primary"
                       @click="menu = false">Cancel</v-btn>
                    <v-btn flat color="primary"
                       @click="$refs.menu.save(date)">
                       OK
                    </v-btn>
                </v-date-picker>
            </v-menu>
        </v-list-tile-content>
        </v-list-tile>
    </v-list>
    <v-divider></v-divider>
    <v-list four-line subheader>
        <v-list-tile avatar>
            <v-list-tile-action>
                <v-switch
                    v-model="alert1year"
                    :label="`Alert @ 1.5 & 1 year(s)`"
                ></v-switch>
            </v-list-tile-action>
        </v-list-tile>
        <v-list-tile avatar>
            <v-list-tile-action>
                <v-switch
                    v-model="alert6months"
                    :label="`Alert @ 6 months`"
                ></v-switch>
            </v-list-tile-action>
        </v-list-tile>
        <v-list-tile avatar>
            <v-list-tile-action>
                <v-switch
                    v-model="alert3months"
                    :label="`Alert @ 3 months`"
                ></v-switch>
            </v-list-tile-action>
```

```
                    </v-list-tile>
                    <v-list-tile avatar>
                        <v-list-tile-action>
                            <v-switch
                                v-model="alert1month"
                                :label="`Alert @ 1 month or less`"
                            ></v-switch>
                        </v-list-tile-action>
                    </v-list-tile>
                </v-list>
                </v-card-text>

                <div style="flex: 1 1 auto;"></div>
            </v-card>
        </v-dialog>
    </div>
</template>

<script>
export default {
    name: "Doc",
    props: ["item"],
    data: () => {
        return {
            name: "",
            date: new Date().toISOString().substr(0, 10),
            alert1year: true,
            alert6months: true,
            alert3months: true,
            alert1month: true,
            menu: false,
            showModal: false
        }
    },
    methods: {
        show() {
            this.showModal = true;
        },
        hide(){
            this.showModal = false;
        },
        save(){
            this.showModal = false;
        }
```

```
    }
  }
</script>

<style scoped>
</style>
```

There’s quite a lot going on here—not only is the code building the UI, but it also has logic for displaying and using a [date/month picker]() Vuetify component.

In the statement **`!!name`**, the double exclamation point characters convert any false-like value (0, null, undefined, false) to a strictly Boolean value.

Let’s break this code into smaller pieces to understand what is going on. First, we’ll have a look at the main toolbar and see which Vuetify components are being used.

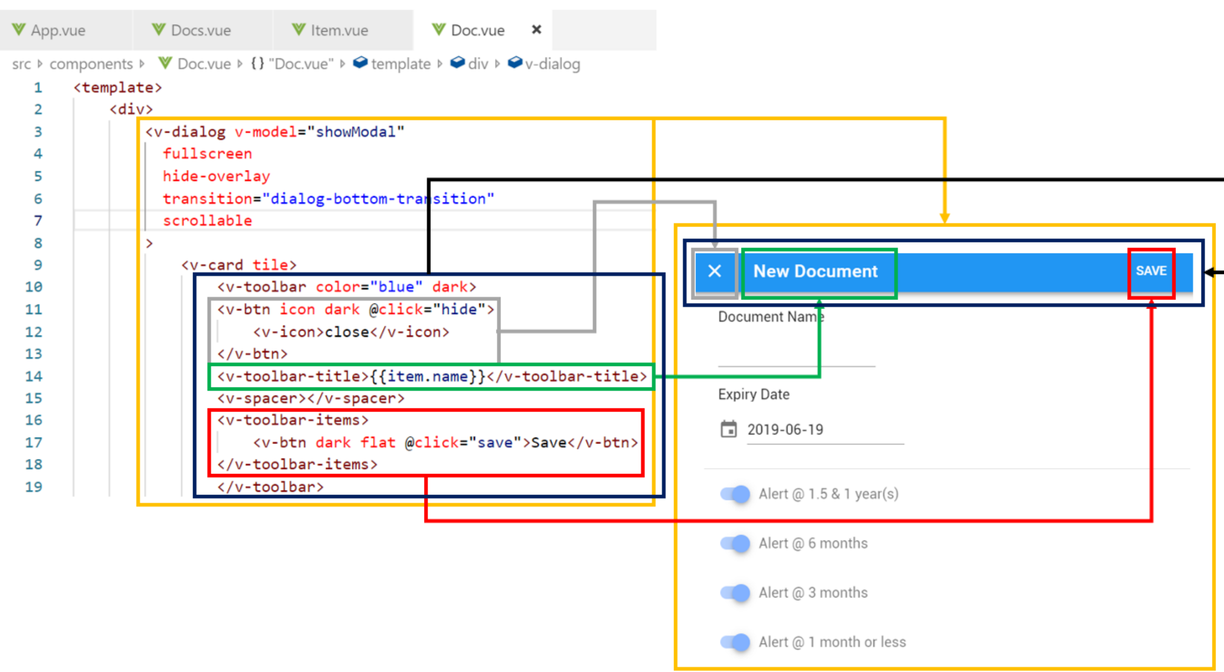


Figure 3-j: Relationship between the Toolbar Code and the UI (Doc.vue)

We can see that the **`v-dialog`** component wraps up the complete UI of the **Doc.vue**. Within **`v-dialog`** there’s a **`v-card`**, and within it we have the **`v-toolbar`** component—which is what we are going to focus on now.

The toolbar is made up of three main parts: The **X** button (**`v-btn`**), which closes the dialog and returns the control to the main screen, the toolbar title (**`v-toolbar-title`**), and the **Save** button (**`v-btn`**).

Now that we’ve covered the toolbar, let’s explore the part of the UI that renders the Document Name field.

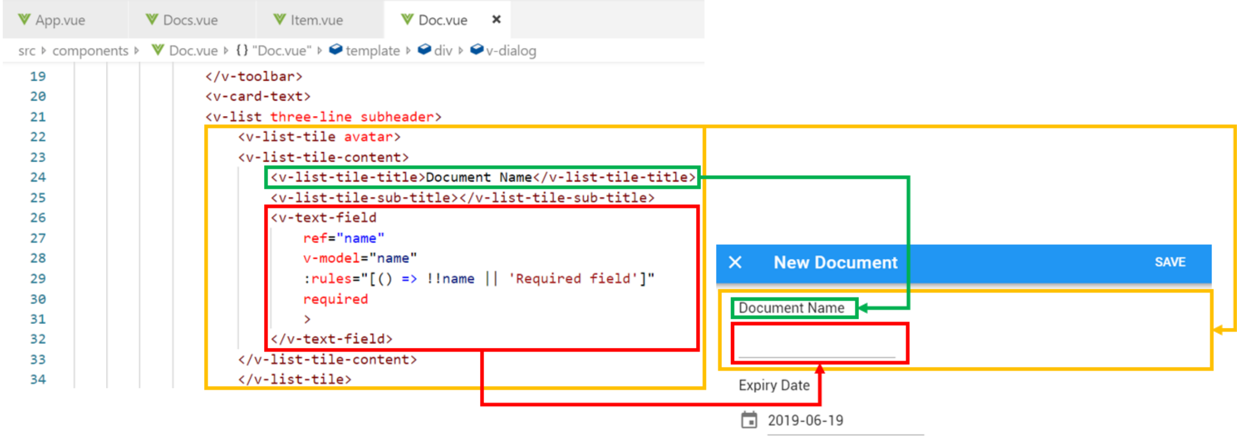


Figure 3-k: Relationship between the "Document Name" Code and UI (Doc.vue)

We can see that after the **`v-toolbar`** component, there's a **`v-list`** component that encapsulates the remaining UI.

Within **`v-list`** there's a **`v-list-tile`** component that is used for displaying the **Document Name** label and field.

Inside of **`v-list-tile`** there's a **`v-tile-tile-content`** component that encapsulates the label and field components.

The **`v-list-tile-title`** component is the one that displays the label name, which is clearly highlighted in green in Figure 3-k.

The **`v-text-field`** component is the one that renders the textbox field. This component has some interesting properties and directives, which determine its functionality.

The **`ref`** and **`v-model`** directives are responsible for binding the **`v-text-field`** component to the **`name`** field.

The **`rules`** property is used for validating the data entered through the textbox. This executes a lambda function that performs the validation, which simply checks that the value of **`name`** is not empty. This is used in combination with the **`required`** property.

This covers the Document Name field. Now, let's explore the Expiry Date field, which is quite interesting as it contains a date/month picker Vuetify component. When the focus is on the Expiry Date field, the picker is displayed—which we can see in the following figure.

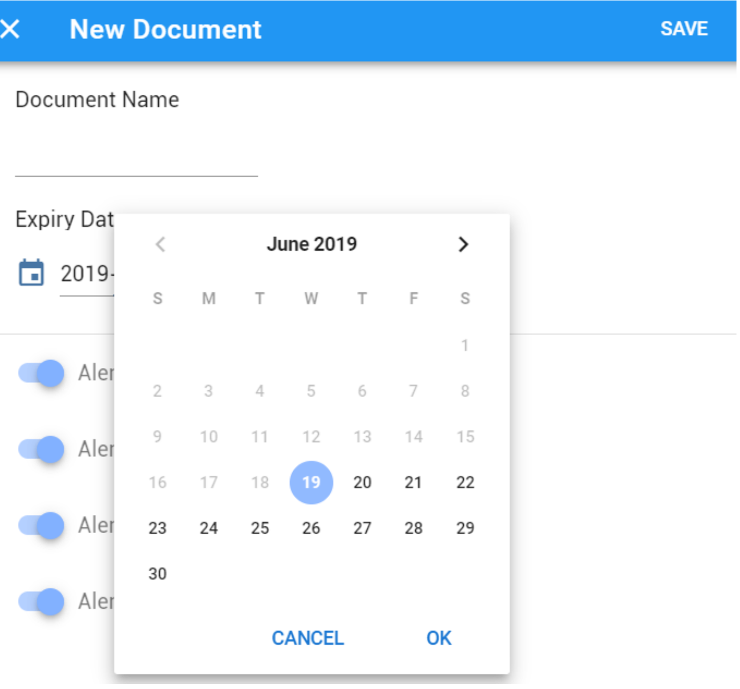


Figure 3-l: "Expiry Date" Picker (Doc.vue)

Figure 3-m illustrates how the date/month picker code relates to the actual component on the screen—let's have a look.

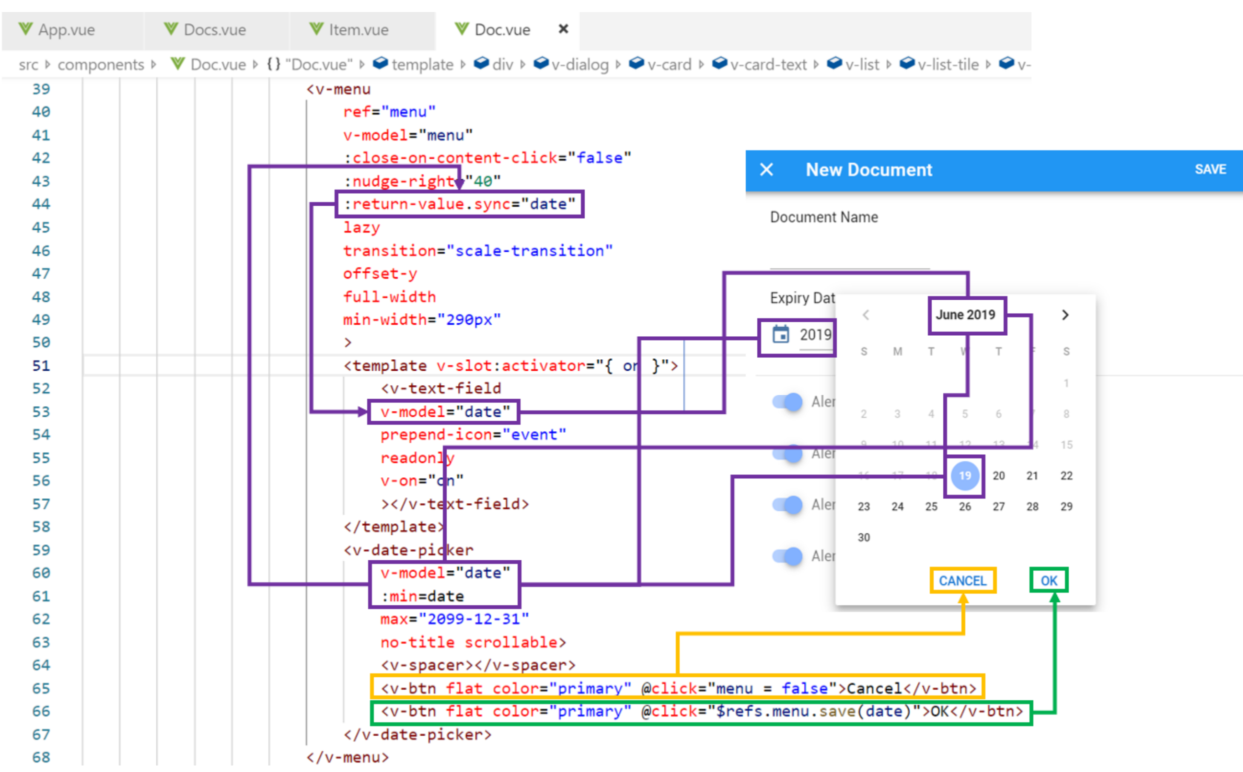


Figure 3-m: The Date/Month Picker and Date Relationship (Doc.vue)

We can see that the date/month picker component (**`v-date-picker`**) binds to **`date`**, which is returned by the **`Doc`** component's **`data`** function. The **`v-date-picker`** component also binds to **`v-text-field`** through **`date`**.

Notice that **`date`** also binds to the **`min`** property of **`v-text-field`**, which means that the minimal **`date`** that can be selected is today's date—this means that only dates that are either in the future or equal to today's date can be selected using the picker. This ensure that documents cannot have an expiry date in the past unless the document was created in the past and the date has already expired.

The last part of the **`Doc`** component's UI has to do with the four alert switches—let's explore this markup code and see how it relates to the UI through the following diagram.

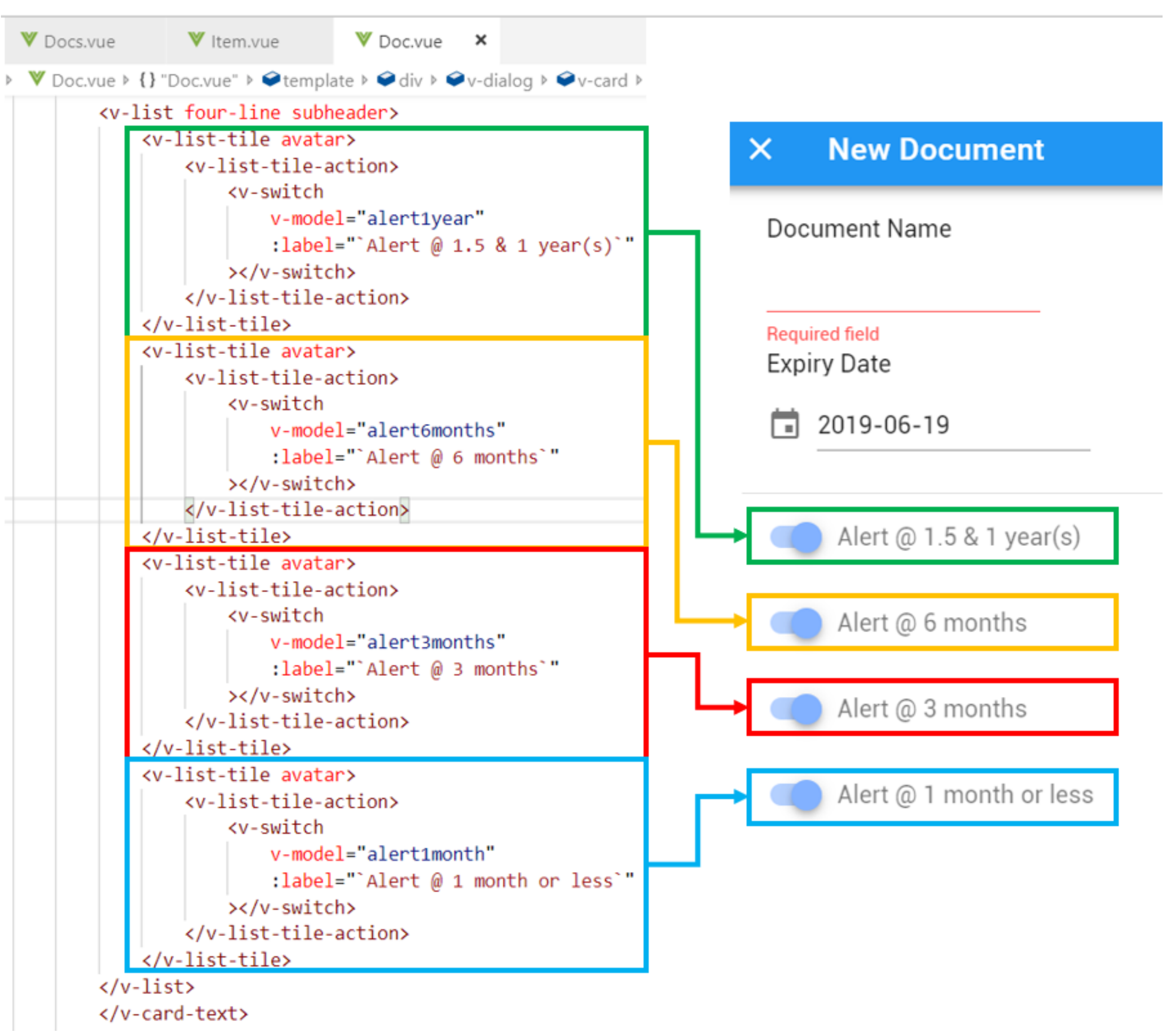


Figure 3-n: The Relationship between the Alert Switches and the UI (Doc.vue)

As we can see, each alert switch is represented by an individual **`v-switch`** component, which is contained within the **`v-list-tile-action`** and **`v-list-tile`** components—which makes them aligned and correctly positioned on the UI layout.

Each **`v-switch`** component has a binding to a variable through the **`v-model`** directive. These variables will store within the **`Doc`** component the values set through the **`v-switch`** components.

You might have noticed that the **`v-model`** directive has been used quite a lot within the **`Doc`** component. The **`v-model`** directive is used in Vue for two-way data binding—which means that if the data changes, the UI components referencing it update. If the UI component value changes, then the data also changes. This is a powerful feature that significantly speeds up development.

This concludes the UI aspects of the **`Doc`** component. However, to invoke it, we need to make some adjustments to **App.vue** (when creating a new document) and to **Item.vue** (when modifying an existing document)—which is what we'll do next.

Adapting App.vue for Doc.vue

With the UI of the **`Doc`** component (**Doc.vue**) ready, we need to be able to invoke the **`Doc`** component, when creating a new document, by clicking on the floating button within **App.vue**. To be able to do that, we need to adjust the code we have written for **App.vue**. The following code listing contains the adapted code that can invoke the **`Doc`** component—let's have a look at what those changes are (highlighted in bold).

Listing 3-j: App.vue (Able to invoke Doc.vue)

```
<template>
  <div id="app">
    <v-app>
      <Doc ref="modal" :item="newdoc" :items="items" />
      <v-layout row>
        <v-flex xs12 sm6 offset-sm3>
          <v-card>
            <Docs :items="items"/>
            <v-card-text tyle="height: 100px; position: relative">
              <v-btn @click="openModal"
                big
                color="pink"
                dark
                absolute
                bottom
                right
                fab
              >
                <v-icon>add</v-icon>
```

```
              </v-btn>
            </v-card-text>
          </v-card>
        </v-flex>
      </v-layout>
    </v-app>
  </div>
</template>

<script>
import Docs from './components/Docs';
import Doc from './components/Doc';

export default {
  name: 'app',
  components: {
    Docs, Doc
  },
  methods: {
    openModal() {
      this.$refs.modal.show();
    }
  },
  data: () => {
    return {
      newdoc: {
        id: -1,
        name: "New Document",
        exp: "",
        alert1year: true,
        alert6months: true,
        alert3months: true,
        alert1month: true
      },
      items: [
        {
          id: 1,
          name: "Test 1",
          exp: "2019-10-16",
          alert1year: false,
          alert6months: true,
          alert3months: false,
          alert1month: false
        },
```

```
        {
          id: 2,
          name: "Test 2",
          exp: "2018-11-16",
          alert1year: false,
          alert6months: false,
          alert3months: false,
          alert1month: true
        }
      ]
    }
  }
}
</script>
<style>
</style>
```

The first noticeable change is that `<Doc ref="modal" :item="newdoc" :items="items" />` has been included within the markup. This means that the **Doc** component is now included, but it won't be visible until it is invoked.

We'll need to be able to invoke the **show** method from the **Doc** component to display it. To do that, we need to add the **ref** attribute to the **Doc** HTML element inside the template tag, so we can access it using the **$refs** property, as follows.

```
openModal() {
    this.$refs.modal.show();
}
```

The **openModal** method simply invokes the **show** method from the **Doc** component, making it visible.

Notice that we are also passing two other properties to the **Doc** component: One is the **newdoc** object, which binds to the **item** property, and the other is the **items** array (which contains the list of documents), which binds to the **items** property.

The **newdoc** object is simply a JSON representation of a new document that contains default settings—it's an empty placeholder for creating a new document, which we can see as follows.

```
newdoc: {
    id: -1,
    name: "New Document",
    exp: "",
    alert1year: true,
    alert6months: true,
```

```
    alert3months: true,
    alert1month: true
}
```

By passing the **newdoc** object to the **Doc** component, we are telling **Doc.vue** that we want to create a new document and not modify an existing one.

Notice that besides the **id**, **name**, and **exp** properties, we now have also the **alert1year**, **alert6months**, **alert3months**, and **alert1month** properties—these will be used to set the values of the **v-switch** components within **Doc.vue**.

However, you might be asking yourself: Why do we need to pass the complete list of documents (the **items** array) to **Doc.vue**?

The reason is that when we add the logic that will be responsible for adding the new document to a database, we'll need to add that new element to the **items** array. In other words, the new document will have to be added to the existing list of documents—and that will be done from **Doc.vue**, and not **App.vue**. This is why the **items** array is passed to the **Doc** component.

Notice that a **click** event handler has been added to the floating button (**v-btn**). This **click** event will execute the **openModal** method, which is responsible for invoking the **show** method from the **Doc** component.

```
<v-btn @click="openModal" …>
```

Next, we can see that we are importing the **Doc** component within the script section: **import Doc from './components/Doc';** and that we have added **Doc** to the **components** object.

```
components: {
    Docs, Doc
}
```

As for the **items** array, notice how I've added the **alert1year, alert6months, alert3months**, and **alert1month** properties to each of the records. For now, these are static records, but later we'll make this dynamic.

Notice as well that I've changed the date format of the expiry date (**exp**) property to **"yyyy-mm-dd"** (for example, **"2018-11-16"**) instead of **"dd MMM yyyy"**, as this will make the functionality of the application easier to write (without the need to convert dates from one format to another).

Adapting Docs.vue for Doc.vue

Although **Docs.vue** does not directly invoke **Doc.vue**, there is a minor change that is important for the overall functionality of the adapted version of **Item.vue**, which is required for it to work properly with **Doc.vue**.

Listing 3-k: Docs.vue (For Item.vue to be able to invoke Doc.vue)

```
<template>
```

```
    <div>
        <v-toolbar color="blue" dark>
            <v-toolbar-title>DocExpire</v-toolbar-title>
            <v-spacer></v-spacer>
        </v-toolbar>

        <v-list two-line>
            <template v-for="item in items">
                <Item :key="item.id" :lgth="items.length"
                 :item="item" :items="items"/>
            </template>
        </v-list>
    </div>
</template>

<script>
import Item from './Item';

export default {
    name: "Docs",
    props: ["items"],
    components: {
        Item
    }
}
</script>

<style scoped>
</style>
```

The minor change has been highlighted in bold in the preceding code. As you can see, we are passing the list of documents (`items` array) to **Item.vue**—which previously we didn't have.

We are doing this because **Item.vue** will need to be able to update the `items` array when a document from the list is deleted, something we will see shortly with the updated code for **Item.vue**.

Adapting Item.vue for Doc.vue

There are some interesting changes that are required for **Item.vue** to be able to invoke **Doc.vue**. Remember that **Item.vue** needs to invoke **Doc.vue** when an existing document needs to be edited. Let's have a look at the updated code for **Item.vue**, which can do this.

Listing 3-l: Item.vue (Able to invoke Doc.vue)

```
<template>
    <div>
        <Doc ref="modal" :item="item" :items="items" />
        <v-list-tile avatar ripple>
            <v-btn @click="removeItem" flat icon color="red lighten-2">
                <v-icon>delete_forever</v-icon>
            </v-btn>
            <v-btn @click="openModal" flat icon color="green lighten-2">
                <v-icon>edit</v-icon>
            </v-btn>
            <v-list-tile-content>
                <v-list-tile-title class="text--primary">
                    {{ item.name }}
                </v-list-tile-title>
                <v-list-tile-sub-title>
                    {{ item.exp }}
                </v-list-tile-sub-title>
                <v-list-tile-sub-title>
                    {{daysLeft(item.exp)}}
                </v-list-tile-sub-title>
            </v-list-tile-content>
            <v-list-tile-action>
                <v-list-tile-action-text>
                    {{ item.id }}
                </v-list-tile-action-text>
                <v-icon v-if="!hasExpired(item.exp)"
                    color="green lighten-1">done_outline</v-icon>
                <v-icon v-else color="red lighten-1">warning</v-icon>
            </v-list-tile-action>
        </v-list-tile>
        <v-divider v-if="item.id + 1 < lgth"
          :key="`divider-${item.id}`"></v-divider>
    </div>
</template>

<script>
import Doc from './Doc';
import moment from 'moment';

export default {
    name: "Item",
    components: {
        Doc
```

```
    },
    props: ["item", "lgth", "items"],
    methods: {
        openModal() {
            this.$refs.modal.show();
        },
        removeItem() {
            let idx = this.items.indexOf(this.item);
            if (idx > -1) {
                this.items.splice(idx, 1);
            }
        },
        daysLeft(dt) {
            let ends = moment(dt);
            let starts = moment(Date.now());
            let years = ends.diff(starts, "year");
            starts.add(years, "years");
            let months = ends.diff(starts, "months");
            starts.add(months, "months");
            let days = ends.diff(starts, "days");
            let rs = years + " years " + months +
                " months " + days + " days";
            return this.hasExpired(dt) ?
                "Expired " + rs.replace(/-/g, '') +
                " ago": "Left: " + rs;
        },
        hasExpired(dt) {
            return (Date.parse(dt) <= Date.now()) ? true : false;
        }
    }
}
</script>

<style scoped>
</style>
```

The first thing we can see is that the **Doc** component has been added to the markup, even though it is not visible.

```
<Doc ref="modal" :item="item" :items="items" />
```

This is practically identical as to what was done within **App.vue**—where the **Doc** component was added to the markup, with the only difference that in this case, the **item** property binds to the existing **item** and not the **newdoc** object.

Notice how the **items** array is also passed to the **Doc** component—this is because when the **Save** button is clicked within **Doc.vue**, the document edited will need to be updated on the **items** array—which is something we'll do later.

Next, we've added two event handlers: one that invokes **removeItem**, and another that calls the **openModal** method.

The **removeItem** method is executed when the red trash bin (**delete_forever**) icon is clicked, whereas the **openModal** method gets triggered when the green pencil (**edit**) icon is clicked.

The next change from the previous version of the code is that we have replaced the static text for the days left with dynamic content: **{{daysLeft(item.exp)}}**.

Basically, what we do here is to invoke the **daysLeft** method by passing to it the expiry date (**exp**) of the current **item**, which returns the number of years, months, and days before the document expires—or if the document has already expired, the number of years, months and days since that occurred.

Test 1
2019-10-16
Left: 0 years 3 months 25 days
Test 2
2018-11-16
Expired 0 years 7 months 4 days ago

Figure 3-o: Years, Months, and Days (Item.vue)

The next change we see in the HTML markup is an interesting one. What we are doing is displaying either the icon that indicates that the document is active—has not expired (**done_outline**)—or we display the icon that indicates that the document has expired (**warning**).

```
<v-icon v-if="!hasExpired(item.exp)"
        color="green lighten-1">done_outline</v-icon>
<v-icon v-else color="red lighten-1">warning</v-icon>
```

The way we do this is by using the **v-if** and **v-else** directives. The **v-if** directive basically means that the first **v-icon** component will be shown only if the document (**item**) has not expired—if the **hasExpired** method returns **false**.

Figure 3-p: The v-icon if the Document (item) has not expired

The second **v-icon** component will only be shown when the **hasExpired** method returns **true**, for the current **item**—when the document has expired.

Figure 3-q: The v-icon if the Document (item) has expired

Next, notice the following statement: **`import moment from 'moment';`**

This statement imports the [Moment.js]() library—which is a great way to parse, validate, display, and manipulate dates in JavaScript. This is what we'll use within the **`hasExpired`** method to check if the expiry date (**`exp`**) of the document has expired.

Before we can use this library, we need to install it. To do that, open the command prompt on your project's root folder and type in the following command.

Listing 3-m: Command to Install Moment.js

```
npm install moment --save
```

Once Moment.js has been successfully installed, we can use it in our code—which we will look at shortly.

Moving on, we can also see that the **`items`** array has been added to the **`props`** array. As explained previously, the **`items`** array is passed on to the **`Doc`** component.

Next, it's time to look at the **`methods`** object and explore each of the methods declared there. The first one is the **`openModal`** method.

```
openModal() {
    this.$refs.modal.show();
}
```

This method is identical to the one we declared within the **App.vue** **`methods`** object. It basically invokes the **`show`** method of **Doc.vue**—this way the **`Doc`** component can be displayed on the screen.

The next method is the **`removeItem`** method, which is responsible for removing the current document (**`item`**) if the user clicks on the **`delete_forever`** icon.

```
removeItem() {
    let idx = this.items.indexOf(this.item);
    if (idx > -1) {
        this.items.splice(idx, 1);
    }
}
```

This method finds the current document (**item**) with the list of documents (**items** array) by calling the **indexOf** method. If found (**idx** > -1), then the current document (**item**) is removed from the **items** array, using the **splice** method.

The **daysLeft** method is an interesting one, as it makes extensive use of the Moment.js library to determine the number of **years**, **months**, and **days** until a document expires or after it has expired. Let's have a look.

```
daysLeft(dt) {
    let ends = moment(dt);
    let starts = moment(Date.now());
    let years = ends.diff(starts, "year");
    starts.add(years, "years");
    let months = ends.diff(starts, "months");
    starts.add(months, "months");
    let days = ends.diff(starts, "days");
    let rs = years + " years " + months +
        " months " + days + " days";
    return this.hasExpired(dt) ?
        "Expired " + rs.replace(/-/g, '') +
        " ago": "Left: " + rs;
}
```

The first two lines of code get the **ends** and **starts** dates by calling the **moment** function. The **ends** date is **item.exp**, which is passed to the **daysLeft** method as **dt**. The **starts** date is the current date, which is retrieved by calling **Date.now**.

The next five lines calculate the differences between both dates by **years**, **months**, and **days**.

Next, the current document's expiry date (**dt**) is evaluated by the **hasExpired** method to see if the date has expired. The corresponding message, which will get displayed on the screen, is returned by the **daysLeft** method.

Finally, the **hasExpired** method, as its name implies, checks if the current document's expiry date (**dt**) is before today's date—if so, it would mean that the document has expired and **true** is returned—otherwise **false** is returned as a result. We can see this as follows.

```
hasExpired(dt) {
    return (Date.parse(dt) <= Date.now()) ? true : false;
}
```

That concludes the review of the changes to **Item.vue**. Next, let's explore some minor adaptations required for **Doc.vue**. All the changes to **App.vue**, **Docs.vue**, and **Item.vue** we've made work seamlessly with **Doc.vue**.

Adjustments to Doc.vue

We need to make some minor changes to **Doc.vue** so this can all work. Let's have a look at the updated code.

Listing 3-n: Updated Doc.vue (which works with the rest of the updated code)

```
<template>
    <div>
        <v-dialog v-model="showModal"
          fullscreen
          hide-overlay
          transition="dialog-bottom-transition"
          scrollable
        >
            <v-card tile>
                <v-toolbar color="blue" dark>
                <v-btn icon dark @click="hide">
                    <v-icon>close</v-icon>
                </v-btn>
                <v-toolbar-title>{{item.name}}</v-toolbar-title>
                <v-spacer></v-spacer>
                <v-toolbar-items>
                    <v-btn dark flat @click="save">Save</v-btn>
                </v-toolbar-items>
                </v-toolbar>
                <v-card-text>
                <v-list three-line subheader>
                    <v-list-tile avatar>
                    <v-list-tile-content>
                        <v-list-tile-title>Document Name
                        </v-list-tile-title>
                        <v-list-tile-sub-title></v-list-tile-sub-title>
                        <v-text-field
                            ref="name"
                            v-model="item.name"
                            :rules="[() => !!item.name ||
                                'Required field']"
                            required
                            >
                        </v-text-field>
                    </v-list-tile-content>
                    </v-list-tile>
                    <v-list-tile avatar>
                    <v-list-tile-content>
```

```
                        <v-list-tile-title>Expiry Date
                        </v-list-tile-title>
                        <v-list-tile-sub-title></v-list-tile-sub-title>
                        <v-menu
                            ref="menu"
                            v-model="menu"
                            :close-on-content-click="false"
                            :nudge-right="40"
                            :return-value.sync="date"
                            lazy
                            transition="scale-transition"
                            offset-y
                            full-width
                            min-width="290px"
                            >
                            <template v-slot:activator="{ on }">
                                <v-text-field
                                v-model="date"
                                prepend-icon="event"
                                readonly
                                v-on="on"
                                ></v-text-field>
                            </template>
                            <v-date-picker
                                v-model="date"
                                :min=today
                                max="2099-12-31"
                                no-title scrollable>
                                <v-spacer></v-spacer>
                                <v-btn flat color="primary"
                                    @click="menu = false">Cancel</v-btn>
                                <v-btn flat color="primary"
                                    @click="$refs.menu.save(date)">
                                    OK
                                </v-btn>
                            </v-date-picker>
                        </v-menu>
                    </v-list-tile-content>
                    </v-list-tile>
                </v-list>
                <v-divider></v-divider>
                <v-list four-line subheader>
                    <v-list-tile avatar>
                        <v-list-tile-action>
```

```
                            <v-switch
                                v-model="item.alert1year"
                                :label="`Alert @ 1.5 & 1 year(s)`"
                            ></v-switch>
                        </v-list-tile-action>
                    </v-list-tile>
                    <v-list-tile avatar>
                        <v-list-tile-action>
                            <v-switch
                                v-model="item.alert6months"
                                :label="`Alert @ 6 months`"
                            ></v-switch>
                        </v-list-tile-action>
                    </v-list-tile>
                    <v-list-tile avatar>
                        <v-list-tile-action>
                            <v-switch
                                v-model="item.alert3months"
                                :label="`Alert @ 3 months`"
                            ></v-switch>
                        </v-list-tile-action>
                    </v-list-tile>
                    <v-list-tile avatar>
                        <v-list-tile-action>
                            <v-switch
                                v-model="item.alert1month"
                                :label="`Alert @ 1 month or less`"
                            ></v-switch>
                        </v-list-tile-action>
                    </v-list-tile>
                </v-list>
                </v-card-text>

                <div style="flex: 1 1 auto;"></div>
            </v-card>
        </v-dialog>
    </div>
</template>

<script>
export default {
    name: "Doc",
    props: ["item", "items"],
    data: () => {
```

```
        return {
            name: "",
            date: new Date().toISOString().substr(0, 10),
            today: new Date().toISOString().substr(0, 10),
            menu: false,
            showModal: false
        }
    },
    methods: {
        show() {
            this.showModal = true;
            if (this.item.id > -1) {
                this.date = this.item.exp;
            }
        },
        hide(){
            this.showModal = false;
        },
        save(){
            this.showModal = false;
        }
    }
  }
</script>

<style scoped>
</style>
```

The first change we can see is that the **`min`** property of the **`v-date-picker`** component binds to **`today`** instead of **`date`**.

The reason for this change is that if **`date`** (which represents the document's expiry date), when **Doc.vue** loads, has expired—then the minimum date that **`v-date-picker`** should allow needs to be today's date—so no dates in the past can be selected as a new expiry date.

Let's have a look at the following example to understand this better.

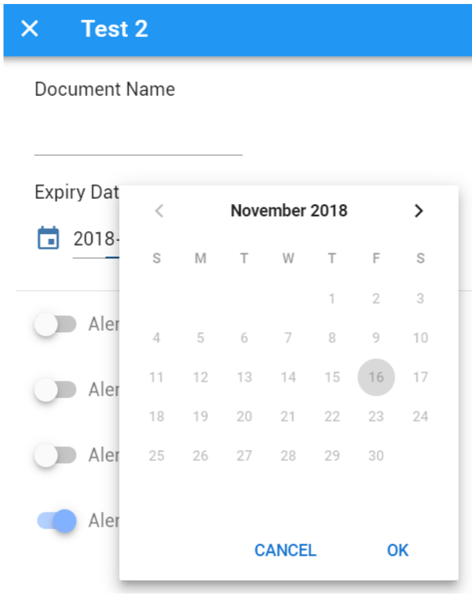


Figure 3-r: The min property of v-date-picker set to today's date

The document's expiry date that has been loaded is **16 November 2018**, which is a date in past—in other words, this date has already expired.

Because the **`min`** property of the **`v-date-picker`** component is set to the value of **`today`** rather than the value of **`date`**, the **`v-date-picker`** component doesn't allow dates in the past to be selected—which is why they appear grayed out.

This is exactly what we want—the expiration date for new or modified documents should only be equal to today's date or a date in the future.

Next, we can see that the **`v-text-field`** component now binds to **`item.name`** rather than the **`name`** property, as it did previously. This ensures that the document's name is loaded from the current document (**`item`**).

The next change we can see is that **`v-switch`** components now have a two-way data binding to their respective properties within the **`item`** object.

So, now we have **`<v-switch v-model="item.alert1year" … >`** instead of **`<v-switch v-model="alert1year" … >`**, and the same for the other **`v-switch`** components—with their respective **`item`** properties.

The reason for these changes for each **`v-switch`** component is that if any of those settings change, then those values should be automatically updated into their respective object properties, so they can be later persisted to the database.

Next, we can see that the **`items`** array has been added to the **`props`** array. We'll need this later when persisting data to the database and then refreshing the list of documents (**`items`**) from within **Doc.vue**, which we will look at in the next chapter.

Following this, notice that the **`today`** property has been added with a default value to the **`data`** function within the script section.

The final change we can appreciate is within the **`show`** method, which adds a bit of logic that checks if a document already exists (**`item.id > -1`**), then assigns to **`date`** the document's expiry date (**`item.exp`**); otherwise, the **`date`** property will have today's date (for a new document).

Let's now run our app and see the changes we have made in action. If your app is not running, execute **`npm run serve`** from the command prompt and refresh your browser page. When the app is running, you should see the following screen.

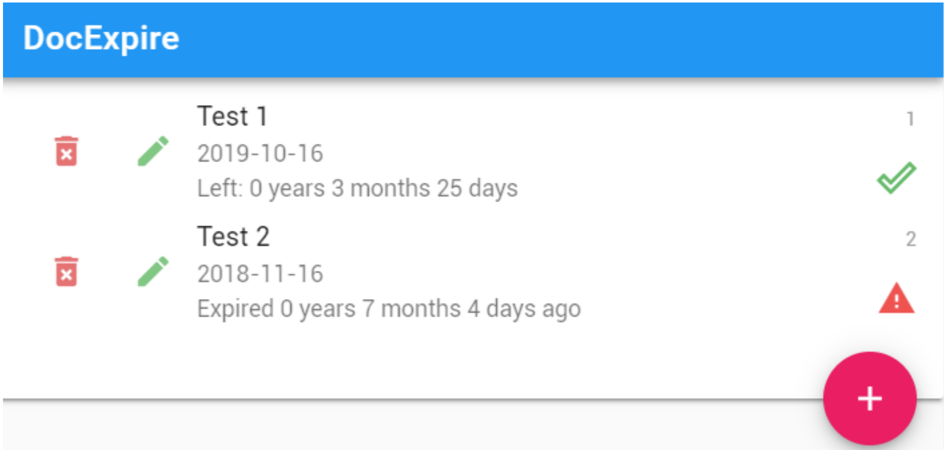


Figure 3-s: The App's Main Screen (After all the changes done)

If we click on the pencil icon on one of the documents, the **`Doc`** component opens with the correct values loaded.

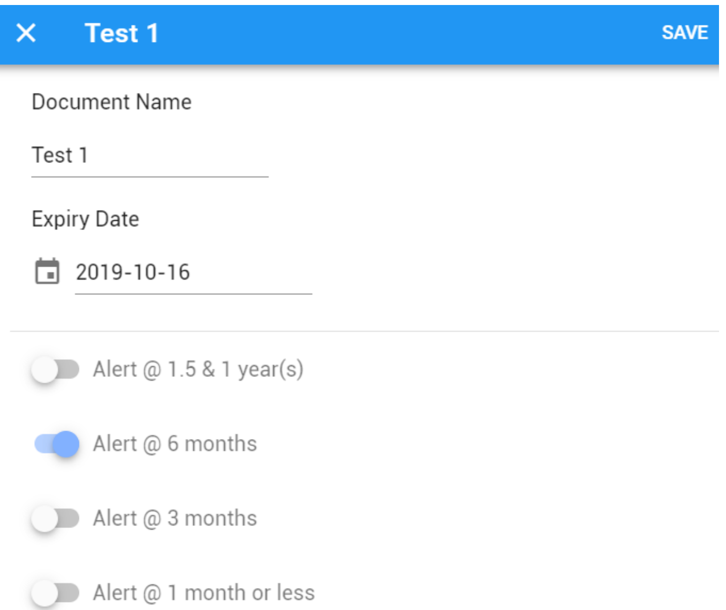


Figure 3-t: The document loaded with the correct data (After all the changes done)

If you click on the floating button, the same dialog window will open, but with the default data for a new document. If you click the `delete_forever` icon, the item will be deleted from the document list.

If you delete one of the documents and reload the browser page, then the list will be populated again with the two documents.

Summary

Now that all our changes are working, we almost have a fully working application. We're only missing four final parts, which are:

- Loading all the documents from the database.
- Saving a new document to the database.
- Editing an existing document and saving it to the database.
- Updating the database when a document is deleted.

These required changes are the final elements that our application needs to be fully functional. We'll explore them in the next chapter.

Chapter 4 Finalizing the App: Database

Quick intro

Throughout the previous chapters, we built the foundation of our application, and now have an almost functional application. The only thing we are missing is being able to persist the data the app uses, which will be the focus of this chapter.

To make things simple, we are going to use a Google spreadsheet as our app's database table, and we'll access the data using a JavaScript library and service specifically designed to use a Google Sheets spreadsheet—this service is called Sheetsu.

Sheetsu dashboard

The first thing to do is to sign up for the Sheetsu service—this can easily be done by signing up with your existing Gmail or Google account. The setup process is fast and simple, and can be completed in less than a minute.

Once you're signed up for the Sheetsu service, you'll be redirected to the dashboard, which looks as follows.

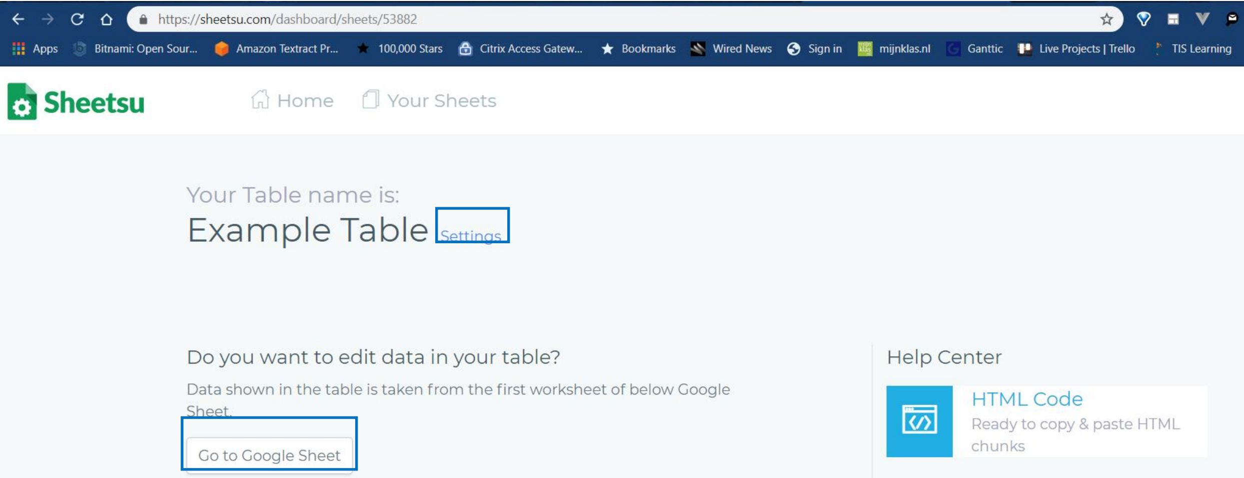
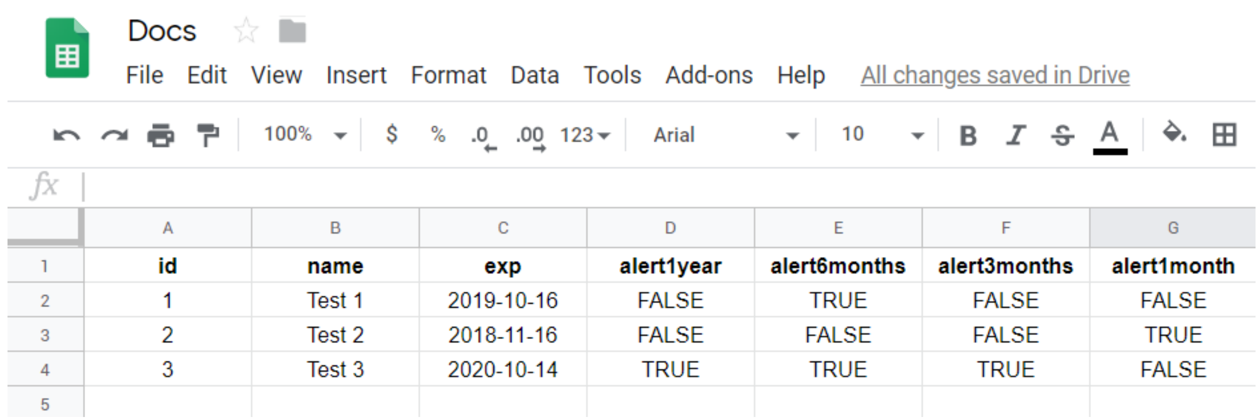


Figure 4-a: The Sheetsu Dashboard

To get started, click **Go to Google Sheet**. This will open the Google spreadsheet in a different browser tab, which we can see as follows.



	A	B	C	D	E	F	G
1	id	name	exp	alert1year	alert6months	alert3months	alert1month
2	1	Test 1	2019-10-16	FALSE	TRUE	FALSE	FALSE
3	2	Test 2	2018-11-16	FALSE	FALSE	FALSE	TRUE
4	3	Test 3	2020-10-14	TRUE	TRUE	TRUE	FALSE
5							

Figure 4-b: The Google Spreadsheet Table

I've added the column names, the two records described within the `data` function of **App.vue**, and a third, additional record.

Click on the **Settings** link next to the name of the table, in this case called **Example Table**, to give it another name. You should see the following dialog with the default properties shown.

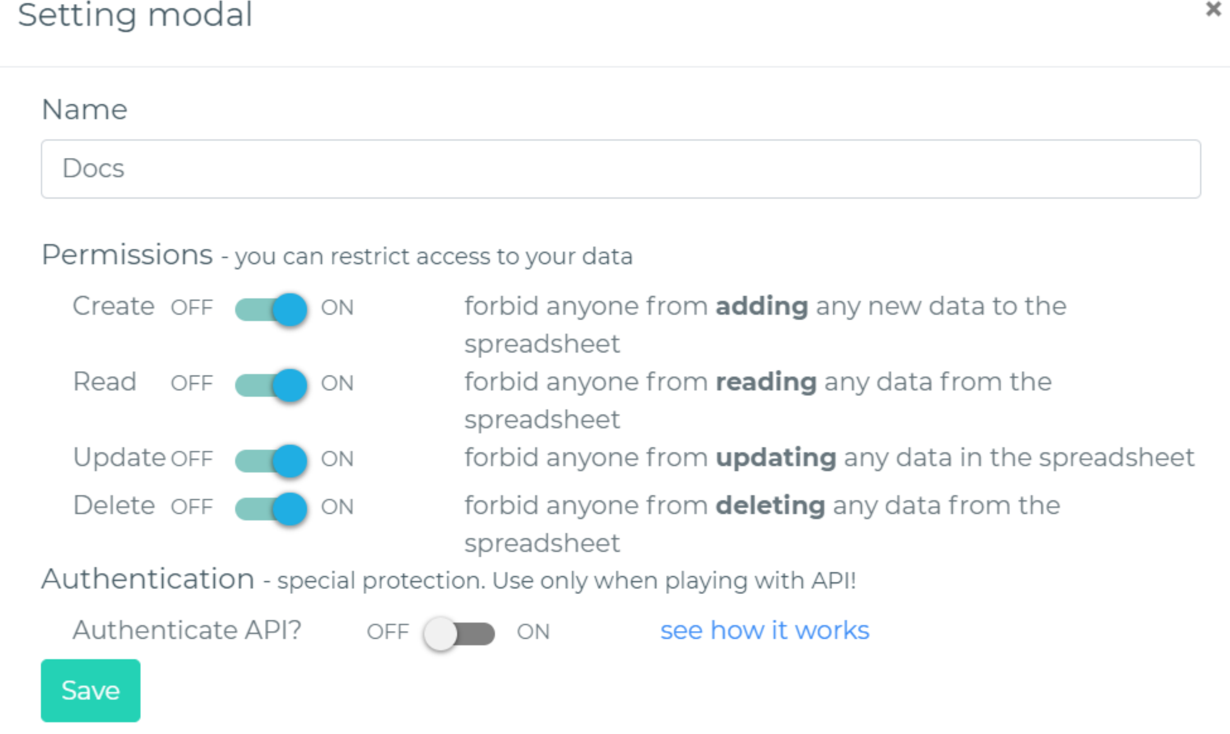


Figure 4-c: Sheetsu Default Table Settings

In my case, the only thing I have done is renamed the table from **Example Table** to **Docs**, by changing the value of the **Name** field.

Before we can start using Sheetsu within our code, we need to go to the root folder project and install Sheetsu from the command line as: **`npm install sheetsu-node --save`**.

Dynamic data loading

Now that we are set up and ready to work with Sheetsu, the next thing we need to do is to add it to **App.vue**, make some modifications to the existing code, and load the **`items`** array from the **`Docs`** table using Sheetsu.

Listing 4-a: Updated App.vue

```
<template>
  <div id="app">
    <v-app>
      <Doc ref="modal" :item="newdoc" :items="items" />
      <v-layout row>
        <v-flex xs12 sm6 offset-sm3>
          <v-card>
            <Docs :items="items"/>
            <v-card-text tyle="height: 100px; position: relative">
              <v-btn @click="openModal"
                big
                color="pink"
                dark
                absolute
                bottom
                right
                fab
              >
                <v-icon>add</v-icon>
              </v-btn>
            </v-card-text>
          </v-card>
        </v-flex>
      </v-layout>
    </v-app>
  </div>
</template>

<script>
import Docs from './components/Docs';
import Doc from './components/Doc';
import sheetsu from 'sheetsu-node';
```

```
let db = sheetsu({
  address: "https://sheetsu.com/apis/v1.0su/..."}); // put your URL here

export default {
  name: 'app',
  components: {
    Docs, Doc
  },
  created () {
      this.getdocs();
  },
  methods: {
    openModal() {
      this.$refs.modal.show();
    },
    getdocs() {
      db.read().then((data) => {
        this.items = JSON.parse(data);
      });
    }
  },
  data: () => {
    return {
      newdoc: {
        id: -1,
        name: "New Document",
        exp: "",
        alert1year: true,
        alert6months: true,
        alert3months: true,
        alert1month: true
      },
      items: []
    }
  }
}
</script>
<style>
</style>
```

I've highlighted the changes to **App.vue** in bold. The first change made to the code is to import a reference to the Sheetsu module, using an **`import sheetsu from 'sheetsu-node';`** statement.

Next, we need to instantiate the Sheetsu object by passing the API address of the Google spreadsheet we created. You can do this from the Sheetsu dashboard by scrolling down to the bottom of the page and clicking **Go to your API**—as we can see in the following figure.

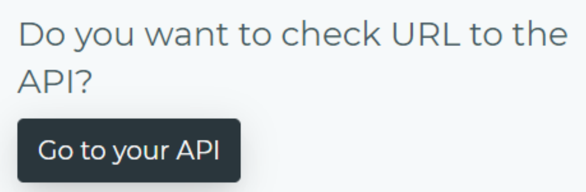


Figure 4-d: The "Go to your API" Button (Sheetsu Dashboard: bottom of the page)

When you click this button, you will be taken to a webpage that displays the JSON representation of data existing on the Google spreadsheet—which looks something like this.

https://sheetsu.com/apis/v1.0su/0575...

Apps Bitnami: Open Sour... Amazon Textract Pr... 100,000 Stars » Other

[{"id":"1","name":"Test 1","exp":"2019-10-16","alert1year":"FALSE","alert6months":"TRUE","alert3months":"FALSE","alert1month":"FALSE"},{"id":"2","name":"Test 2","exp":"2018-11-16","alert1year":"FALSE","alert6months":"FALSE","alert3months":"FALSE","alert1month":"TRUE"},{"id":"3","name":"Test 3","exp":"2020-10-14","alert1year":"TRUE","alert6months":"TRUE","alert3months":"TRUE","alert1month":"FALSE"}]

Figure 4-e: The JSON Sheetsu (Google Spreadsheet) Results

The URL of this JSON result is used when creating the Sheetsu instance using the following line of code: `let db = sheetsu({address: "https://sheetsu.com/apis/v1.0su/..."});`.

Next, we can see that the list of documents gets loaded from the Google spreadsheet when the component's `created` event is executed by calling the `getdocs` method—which happens when **App.vue** is rendered on the screen.

```
created () {
    this.getdocs();
}
```

The `getdocs` method is responsible for executing the `read` method from the Sheetsu instance, retrieving those values and assigning them to the `items` array as JSON results—as seen in Figure 4-e.

```
getdocs() {
    db.read().then((data) => {
        this.items = JSON.parse(data);
    });
}
```

The result needs to be converted from a string JSON representation to its JSON object equivalent—which is why **`JSON.parse`** is invoked.

For this to fully work, there's one final change required: The default data that was assigned to the **`items`** array needs to be cleared, which is why it is initialized as **`items: []`**.

If you now run the application, you should see that the data is loaded dynamically.

Deleting documents

Now that we can load the documents dynamically from the Google spreadsheet table, the next thing we need to do is to be able to delete a document from the database when clicking the **`delete_forever`** icon. The following listing shows the updated **Item.vue** code that does this.

Listing 4-b: Updated Item.vue

```
<template>
    <div>
        <Doc ref="modal" :item="item" :items="items" />
        <v-list-tile avatar ripple>
            <v-btn @click="removeItem" flat icon color="red lighten-2">
                <v-icon>delete_forever</v-icon>
            </v-btn>
            <v-btn @click="openModal" flat icon color="green lighten-2">
                <v-icon>edit</v-icon>
            </v-btn>
            <v-list-tile-content>
                <v-list-tile-title class="text--primary">
                    {{ item.name }}
                </v-list-tile-title>
                <v-list-tile-sub-title>
                    {{ item.exp }}
                </v-list-tile-sub-title>
                <v-list-tile-sub-title>
                    {{daysLeft(item.exp)}}
                  </v-list-tile-sub-title>
            </v-list-tile-content>
            <v-list-tile-action>
                <v-list-tile-action-text>
                    {{ item.id }}
                </v-list-tile-action-text>
                <v-icon v-if="!hasExpired(item.exp)"
                    color="green lighten-1">done_outline</v-icon>
                <v-icon v-else color="red lighten-1">warning</v-icon>
            </v-list-tile-action>
```

```
        </v-list-tile>
        <v-divider v-if="item.id + 1 < lgth"
            :key="`divider-${item.id}`"></v-divider>
    </div>
</template>

<script>
import Doc from './Doc';
import moment from 'moment';
import sheetsu from 'sheetsu-node';

let db = sheetsu({
  address: "https://sheetsu.com/apis/v1.0su/..."}); // put your URL here

export default {
    name: "Item",
    components: {
        Doc
    },
    props: ["item", "lgth", "items"],
    methods: {
        openModal() {
            this.$refs.modal.show();
        },
         removeItem() {
            db.delete(
                "id",
                this.item.id
            ).then(() => {
                let idx = this.items.indexOf(this.item);
                if (idx > -1) {
                    this.items.splice(idx, 1);
                }
            });
        },
        daysLeft(dt) {
            let ends = moment(dt);
            let starts = moment(Date.now());
            let years = ends.diff(starts, "year");
            starts.add(years, "years");
            let months = ends.diff(starts, "months");
            starts.add(months, "months");
            let days = ends.diff(starts, "days");
            let rs = years + " years " + months + " months " +
```

```
                days + " days";
            return this.hasExpired(dt) ?
                "Expired " + rs.replace(/-/g, '') + " ago"
                : "Left: " + rs;
        },
        hasExpired(dt) {
            return (Date.parse(dt) <= Date.now()) ? true : false;
        }
    }
}
</script>

<style scoped>
</style>
```

There are only three changes to the code: the first is the **`import`** statement that imports the **`sheetsu`** module, the second is the creation of the **`sheetsu`** instance, and the third is the change to the **`removeItem`** method that performs the deletion operation.

The logic within the **`removeItem`** module is quite simple—it basically calls the **`delete`** method from the **`db`** instance that points to the Google spreadsheet table.

```
removeItem() {
    db.delete(
        "id",
        this.item.id
    ).then(() => {
        let idx = this.items.indexOf(this.item);
        if (idx > -1) {
            this.items.splice(idx, 1); // remove 1 item at [idx]
        }
    });
}
```

The way the **`delete`** method works, is that if we pass it the name of the column we want to focus on—the **`id`** and the value which we want to remove, in this case **`this.item.id`**—then the rows that match that search criteria will be deleted.

After the document is deleted, then **`item`** gets removed from the **`items`** array, which is what **`then(() => {} ... );`** does.

Because we are using Sheetsu's free plan, you will notice that when you click on the **`delete_forever`** icon, the document will not be deleted on the Google spreadsheet. This is because the free version doesn't allow for items to be deleted. If you open your browser's developer tools, you should see the following message.

```
[HMR] Waiting for update signal from WDS...                                          log.js?1afd:24
▶DELETE https://sheetsu.com/apis/v1.0su/05755177b2fb/id/3 402 (Payment Required)     delete.js?d5ac:40
▶Uncaught (in promise) TypeError: Cannot read property 'items' of undefined          Item.vue?ca5b:49
    at eval (Item.vue?ca5b:49)
```

Figure 4-f: Developer tools (DELETE message): Sheetsu

Notice that message returned by the Sheetsu API is quite clear—a **402 (Payment Required)** response is returned. If you would like to use a paid Sheetsu plan, feel free to give it a try.

This validates that our code works, but the delete operation was not carried out on the Google spreadsheet due to the paywall option; however, you will notice that the document has been removed from the list of documents on the UI.

Saving new or existing documents

There's one final thing to do before we have a finished application—we need to be able to save new or existing documents. To achieve this, we need to make some changes to **Doc.vue** as follows.

Listing 4-c: Updated Doc.vue

```
<template>
    <div>
        <v-dialog v-model="showModal"
          fullscreen
          hide-overlay
          transition="dialog-bottom-transition"
          scrollable
        >
            <v-card tile>
                <v-toolbar color="blue" dark>
                <v-btn icon dark @click="hide">
                    <v-icon>close</v-icon>
                </v-btn>
                <v-toolbar-title>{{item.name}}</v-toolbar-title>
                <v-spacer></v-spacer>
                <v-toolbar-items>
                    <v-btn dark flat @click="save">Save</v-btn>
                </v-toolbar-items>
                </v-toolbar>
                <v-card-text>
                <v-list three-line subheader>
                    <v-list-tile avatar>
                    <v-list-tile-content>
                        <v-list-tile-title>
                            Document Name
```

```
                </v-list-tile-title>
                <v-list-tile-sub-title></v-list-tile-sub-title>
                <v-text-field
                    ref="name"
                    v-model="item.name"
                    :rules="[
                    () => !!item.name || 'Required field']"
                    required
                    >
                </v-text-field>
            </v-list-tile-content>
            </v-list-tile>
            <v-list-tile avatar>
            <v-list-tile-content>
                <v-list-tile-title>
                    Expiry Date
                </v-list-tile-title>
                <v-list-tile-sub-title></v-list-tile-sub-title>
                <v-menu
                    ref="menu"
                    v-model="menu"
                    :close-on-content-click="false"
                    :nudge-right="40"
                    :return-value.sync="date"
                    lazy
                    transition="scale-transition"
                    offset-y
                    full-width
                    min-width="290px"
                    >
                    <template v-slot:activator="{ on }">
                        <v-text-field
                        v-model="date"
                        prepend-icon="event"
                        readonly
                        v-on="on"
                        ></v-text-field>
                    </template>
                    <v-date-picker
                        v-model="date"
                        :min=today
                        max="2099-12-31"
                        no-title scrollable>
                        <v-spacer></v-spacer>
```

```
                            <v-btn flat color="primary"
                            @click="menu = false">Cancel</v-btn>
                            <v-btn flat color="primary"
                            @click="$refs.menu.save(date)">OK</v-btn>
                        </v-date-picker>
                    </v-menu>
                </v-list-tile-content>
                </v-list-tile>
            </v-list>
            <v-divider></v-divider>
            <v-list four-line subheader>
                <v-list-tile avatar>
                    <v-list-tile-action>
                        <v-switch
                            v-model="item.alert1year"
                            :label="`Alert @ 1.5 & 1 year(s)`"
                        ></v-switch>
                    </v-list-tile-action>
                </v-list-tile>
                <v-list-tile avatar>
                    <v-list-tile-action>
                        <v-switch
                            v-model="item.alert6months"
                            :label="`Alert @ 6 months`"
                        ></v-switch>
                    </v-list-tile-action>
                </v-list-tile>
                <v-list-tile avatar>
                    <v-list-tile-action>
                        <v-switch
                            v-model="item.alert3months"
                            :label="`Alert @ 3 months`"
                        ></v-switch>
                    </v-list-tile-action>
                </v-list-tile>
                <v-list-tile avatar>
                    <v-list-tile-action>
                        <v-switch
                            v-model="item.alert1month"
                            :label="`Alert @ 1 month or less`"
                        ></v-switch>
                    </v-list-tile-action>
                </v-list-tile>
            </v-list>
```

```
                </v-card-text>

                <div style="flex: 1 1 auto;"></div>
            </v-card>
        </v-dialog>
    </div>
</template>

<script>
import sheetsu from 'sheetsu-node';

let db = sheetsu({
  address: "https://sheetsu.com/apis/v1.0su/..."});

export default {
    name: "Doc",
    props: ["item", "items"],
    data: () => {
        return {
            name: "",
            date: new Date().toISOString().substr(0, 10),
            today: new Date().toISOString().substr(0, 10),
            menu: false,
            showModal: false
        }
    },
    methods: {
        show() {
            this.showModal = true;
            if (this.item.id > -1) {
                this.date = this.item.exp;
            }
        },
        hide() {
            this.showModal = false;
        },
        save() {
            if (this.item.id == -1) {
                this.adddoc();
            }
            else {
                this.editdoc();
            }
            this.showModal = false;
```

```
        },
        adddoc() {
            let doc = {
                id: this.items.length + 1,
                name: this.item.name,
                exp: this.date,
                alert1year: this.item.alert1year,
                alert6months: this.item.alert6months,
                alert3months: this.item.alert3months,
                alert1month: this.item.alert1month
            };
            db.create(doc).then(() => {
                this.items.push(doc);
            });
        },
        editdoc() {
            this.item.exp = this.date;
            let doc = {
                id: this.item.id,
                name: this.item.name,
                exp: this.date,
                alert1year: this.item.alert1year,
                alert6months: this.item.alert6months,
                alert3months: this.item.alert3months,
                alert1month: this.item.alert1month };
            db.update(
                "id",
                this.item.id,
                doc).then(() => {
                let idx = this.items.findIndex
                    ((itm => itm.id == this.item.id));
                this.items[idx] = doc;
            });
        }
    }
  }
</script>

<style scoped>
</style>
```

We can see that the first two changes highlighted in bold have to do with importing the Sheetsu module and instantiating it.

Next, within the **Save** method, we check if an **item** is new or if it is already existing. If it is a new **item** (**this.item.id == -1**), then the **adddoc** method is called. If it is an existing item, then the **editdoc** method is called.

Let's now explore in detail the **adddoc** method.

```
adddoc() {
    let doc = {
      id: this.items.length + 1,
      name: this.item.name,
      exp: this.date,
      alert1year: this.item.alert1year,
      alert6months: this.item.alert6months,
      alert3months: this.item.alert3months,
      alert1month: this.item.alert1month
    };
    db.create(doc).then(() => {
        this.items.push(doc);
    });

}
```

We begin the **adddoc** method by creating a **doc** object to which we assign the values of each of the new document's properties.

This **doc** object is then committed to the Google spreadsheet table (through Sheetsu) by calling the **db.create** method. The **doc** object is also added to the **items** array, which represents the list of documents.

Now, let's explore the **editdoc** function.

```
editdoc() {
    this.item.exp = this.date;
    let doc = {
      id: this.item.id,
      name: this.item.name,
      exp: this.date,
      alert1year: this.item.alert1year,
      alert6months: this.item.alert6months,
      alert3months: this.item.alert3months,
      alert1month: this.item.alert1month };
    db.update("id", this.item.id, doc).then(() => {
      let idx = this.items.findIndex
      ((itm => itm.id == this.item.id));
        this.items[idx] = doc;
      });
}
```

The first thing we do is assign the **date** value to **item.exp**. Then the **doc** object is created by assigning the values of the active document to their respective **doc** properties.

The update to the Google Spreadsheet is done by calling **`db.update`**, by looking for the **`id`** column that matches the value of the **`item.id`**.

The **`doc`** object is passed to **`db.update`**, which represents the document to be updated on the Google Spreadsheet.

Once the document has been updated, the document (**`item`**) is found within the list of documents (**`items`** array) using the **`findIndex`** method, and then the document is updated within the **`items`** array by calling **`this.items[idx] = doc;`**.

If you run the application and attempt to add, edit, or delete a document, you will notice that all these operations return a **402 (Payment Required)** response from the Sheetsu service.

This means that all code for adding, editing, and deleting a document work. However, to enable that functionality on the Google spreadsheet table and reflect those changes, you'll have to upgrade your Sheetsu plan.

Awesome—by using Google Sheets as a data source, we now have a small and fully working Vue application that works like the application we built in *Flutter Succinctly*.

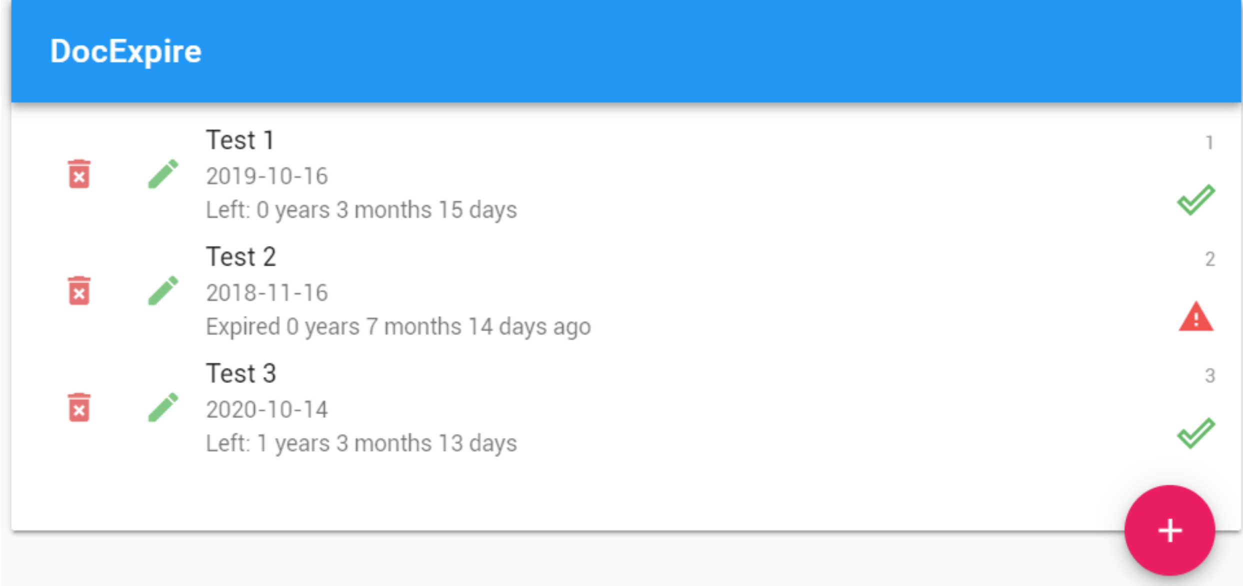


Figure 4-g: The Finished Vue App

Project source code

You can download the complete source code files for the demo project built throughout this book from here.

Node modules and dependencies are not included as part of the code package, so you'll need to run the **`npm install`** command from the project root folder (where **package.json** resides) to download and install all the modules and dependencies required.

Closing comments

We've covered quite a bit of ground. We've explored the essentials of Vue and how to quickly build an application with this amazing framework.

However, there's quite a lot to learn about Vue. The goal of this book was to get you started and acquainted with the fundamentals and foundation of the framework, and help you build something relatively easily, without having to first become an expert on the framework.

Going forward, there are a few things to keep exploring about the framework, such as routing (using Vue Router), state management (using Vuex), and server-side rendering (using Nuxt.js).

Vue is more than a framework—it's a mature and vibrant ecosystem that keeps growing and welcomes new adopters and fans every day.

I invite you to take this application a step further and add extra functionality, such as authentication and possibly transforming it into a single-page application. Go beyond the Sheetsu (Google Sheets) approach and replace it with a different backend, such as Firebase.

The possibilities with Vue are endless, and I'm excited to hear about what you build going forward. Thank you for reading, and goodbye until next time!