

NEW EDITION

LARGE SCALE APPS WITH **VUE 3** AND TYPESCRIPT



DAMIANO FUSCO

LARGE SCALE APPS WITH VUE 3 AND TYPESCRIPT

Build front-ends that can grow to a large code base that is organized and easy to expand and maintain using technique like:

- **Development of UI components in isolation** using an API client that can easily serve live data or mocked data
- **A modular Vuex store** organized into application domains
- **Internationalization and Localization** for language translation and number/dates formatting according to a specific culture
- **TypeScript type-checking** at development time to decrease run-time bugs or errors
- **Directory structure, file, and code naming conventions**
- **Unit tests** for models and components
- And more

Companion Code:

<https://github.com/damianof/large-scale-apps-my-vue3-project>

Damiano Fusco

Web: damianofusco.com

Email: me@damianofusco.com

Twitter: @damianome

GitHub: github.com/damianof

Copyright © 2020 by Damiano Fusco

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author and publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the author and publisher, addressed “Attention: Permissions Coordinator,” at the email me@damianofusco.com.

LARGE SCALE APPS WITH VUE 3 AND TYPESCRIPT

PREFACE	13
Thanks	13
Audience	13
Goal	14
Conventions	14
PREREQUISITES	16
COMPANION CODE	17
Chapter 1	18
SETTING UP THE PROJECT WITH VUE-CLI	
Create Project Wizard	18
Differences Between Operating Systems	26
Chapter 1 Recap	29
What We Learned	
Observations	
Improvements	
Chapter 2	30
YOUR FIRST COMPONENT	
ItemsList Component Requirements	30
ItemsList Component Code	32
Home View	35
App.vue Updates	40
Chapter 2 Recap	43
What We Learned	
Observations	
Improvements	
Chapter 3	44
DATA MODEL INTERFACES	

Models Directory	44
Interface ItemInterface	44
ItemsList Component	46
Home View	48
Chapter 3 Recap	52
What We Learned	
Observations	
Improvements	
Chapter 4	53
ADDING EVENTS TO THE ITEMS COMPONENT	
ItemsList Component	53
Chapter 4 Recap	56
What We Learned	
Observations	
Improvements	
Chapter 5	57
INTRO TO UNIT TESTING	
Item Component	57
Item Component Unit Tests	62
ItemsList component	70
Chapter 5 Recap	76
What We Learned	
Observations	
Improvements	
Chapter 6	77
INTRODUCING VUEX	
Vuex Overview	77
Main Vuex Store (file: store/index.ts)	78

Interface ItemsStateInterface	79
Main Vuex Store (File: store/index.ts)	80
Home View	87
Web Browser	90
ItemsList.component.vue	91
Home.vue	92
Web Browser	94
Loader Component	95
ItemsList Component	99
ItemsList Component enhancing Item click handler	101
Vuex Store: selectItem Mutation	105
Vuex Store: selectItem Action	106
Chapter 6 Recap	108
What We Learned	
Observations	
Improvements	
Chapter 7	110
API CLIENT	
API Client Overview	112
Domains	113
The Main ApiClient	114
Items Api Client	115
ItemsApiClientUrls.interface.ts	116
ItemsApiClient.interface.ts	117
ItemsApiClient.model.ts	117
index.ts	121
Mock and Live Api Clients	122
Mock Api Client	122
Live Api Client	126

Api Client Factory	130
Vuex Store Instance updates	134
Alternatives	138
Chapter 7 Recap	139
What We Learned	
Observations	
Improvements	
Chapter 8	141
ENHANCE THE API CLIENT	
HttpClient Interfaces and Models	142
HttpRequestParams.interface.ts	143
HttpClient.interface.ts	143
HttpClient.model.ts	144
HttpClient.ts	148
index.ts	148
HttpClient Unit Tests	149
Testing a successful “get” response	151
Testing an unsuccessful “get” response	154
ItemsApiClientModel Update	157
Chapter 8 Recap	160
What We Learned	
Observations	
Improvements	
Chapter 9	161
MODULARIZE THE VUEX STORE	
Current Vuex Store instance (src/store/index.ts)	162
Store and State Models	164
Root Store	166
RootState.interface.ts	167

RootStore.interface.ts	167
RootStore.model.ts	168
Items Store	170
ItemsMutationType.ts	171
ItemsState.interface.ts	171
Main MutationType Namespace	173
MutationType.ts	174
Module Names Namespace	176
StoreModuleNames.ts	177
index.ts	177
Store Instance	179
Items Store Instance	181
initialState.ts	182
index.ts	183
Root Store Instance	189
initialState.ts	189
index.ts	190
src/store/index.ts	192
src/main.ts Updates	193
Home.vue Updates	193
Summary	196
Chapter 9 Recap	197
What We Learned	
Observations	
Improvements	
Chapter 10	198
LOCALIZATION AND INTERNATIONALIZATION – LANGUAGE LOCALIZATION – part 1	
Plugin: vue-i18n	200
App.vue	205

Chapter 10 Recap	209
What We Learned	
Observations	
Improvements	
Chapter 11	210
LOCALIZATION AND INTERNATIONALIZATION – LANGUAGE LOCALIZATION – part 2	
The LocaleSelector Component	210
Vuex State Module for Locales	210
LocalesMutationType.ts	211
LocalesState.interface.ts	212
src/models/store/index.ts Updates	213
StoreModuleNames.ts Updates	213
MutationType.ts Updates	213
RootStore.interface.ts Updates	214
LocalesStateInterface implementation	216
initialState.ts	216
index.ts	219
src/root/index.ts updates	221
LocaleSelector.component.vue	223
LocaleFlagRadio.component.vue	225
File src/main.ts	228
File src/App.vue	229
Browser	233
Chapter 11 Recap	235
What We Learned	
Observations	
Improvements	
Chapter 12	236
LOCALIZATION AND INTERNATIONALIZATION – LANGUAGE LOCALIZATION – part 3	

en-US.json:	239
it-IT.json:	240
fr-FR.json:	240
es-ES.json:	240
Localize the Items Component	245
en-US.json:	245
it-IT.json:	246
fr-FR.json:	246
es-ES.json:	246
ItemsList.component.vue	247
Browser	249
Chapter 12 Recap	251
What We Learned	
Observations	
Improvements	
Chapter 13	252
LOCALIZATION AND INTERNATIONALIZATION – NUMBER AND DATETIME FORMATS	
JSON Locales Files Updates	253
en-US.json:	253
it-IT.json:	255
fr-FR.json:	257
es-ES.json:	259
vue-i18n-next-plugin/index.ts Updates	261
App.vue	263
Web Browser	265
Chapter 13 Recap	267
What We Learned	
Observations	
Improvements	

Chapter 14	268
APP CONFIGURATION	
File: src/shims-vue.d.ts	268
src/config	269
ConfigInterface and Configuration Files	269
mock.json	271
local.json	272
beta.json	273
live.json	274
tsconfig.json	275
src/config/index.ts	276
Unit Tests against Config	279
Unit Tests against configsMap	279
Unit Tests against Config instance	280
Update package.json Script section	282
HttpClient.model.ts	286
src/api-client/index.ts	286
src/api-client/live/items/index.ts	287
src/api-client/mock/items/index.ts	288
Chapter 14 Recap	290
What We Learned	
Observations	
Improvements	
Chapter 15	291
USING SCSS LIBRARIES	
SCSS Library	292
Remove all the <style> code from the components	292
Flag-Icons Plugin Code update	292
MyAppScss Plugin	293

main.ts Updates	293
public/index.html Updates	294
ThemeSelector component	295
App.vue updates	300
Chapter 15 Recap	303
What We Learned	
Observations	
Improvements	
THE VUE.CONFIG.JS FILE	
Naming Standards	307
NAMING CONVENTIONS	
Coding Standards	307
TypeScript any	307
Interfaces	307
Directory/File Naming and Structure	307
Directory Names	307
File Names	308
Interface File Names	308
Vue single-file Components File Names	308
Vue single-file Views File Names	308
Unit Tests file names	309
Directory src	309
Directory tests/unit	311
Chapter XYZ	312
MORE CHAPTERS COMING SOON	

PREFACE

Thanks

First, I would like to thank **Evan You**¹, the creator of **Vue.js**². Thank you for not giving up on this project a few years ago when it was still small and people were still skeptical about it. You gave us an amazing gift with such a lightweight and performing framework, which is progressive and not opinionated like many others out there.

I also want to thank my son for helping me proof read and validate the steps in each chapter by building the same project, and my wife for translating the Italian version (coming soon on **Leanpub**)

Audience

The audience for this book is from beginners with some experience in **MV*** applications, to advanced developers. The format is similar to a cookbook, but instead of individual recipes we'll go through creating a project and keep enhancing, refactoring, and make it better as we move forward to more advanced chapters to show different patterns, architecture, technologies.

Note: Some of the patterns illustrated here are not specific to **Vue**, but can

¹ You can reach Evan You on eTwitter here: <https://twitter.com/youyuxi>

² Official website: <https://vuejs.org/>

applied in any application written in **TypeScript** or **JavaScript**. For example, code from Chapters 3, 7, 8, 14 can also be used in **React/Angular** or other front-end apps. Similarly, code from Chapters 3 and 14 can also be used in **NodeJS** apps.

Goal

The primary goal of this book is to show you how to structure a Vue.js project, its directories and files, adopt naming conventions, follow state management patterns, enforce type checking at development time through TypeScript, write unit tests, and more, in proven ways that offer a solid foundation for building a large-scale application that is more easily to expand and maintain.

We'll start creating a simple project and grow this throughout the chapters to show how some patterns, naming conventions, and strategies, will help create a more solid foundation and keeping the code organized and avoid cluttering.

We'll create a TypeScript API client that can easily serve static mocked data or switch to communicate with a live API once that is available, thus enabling to develop our front-end early even before we (or another team) has completed their back-end coding for the API. We'll integrate internationalization and localization, learn how we can create plugins, and more.

Conventions

Going forward, I would refer to **Vue.js** simply with **Vue**. I will highlight most terms or names in bold, rather than define different fonts/styles depending on

whether a term is code, or a directory name or something else.

PREREQUISITES

This book assumes that you are familiar with the **terminal** (command prompt on Windows), have already worked with the **Node.js** and **NPM** (Node Package Manager), know how to install packages, and are familiar with the **package.json** file.

It also assumes you are familiar with **JavaScript**, **HTML**, **CSS** and in particular with **HTML DOM** elements properties and events.

It will also help if you have some preliminary knowledge of **TypeScript**³ as we won't get into details about the language itself or all of its features but mostly illustrate how to enforce type checking at development time with it.

You will need a text editor like **VS Code** or **Sublime Text**, better if you have extensions/plugins installed that can highlight Vue code syntax (like in **.vue** single files). For VS Code for example, you could use extensions like **Vetur**⁴ (just search for it within the VS code extensions tab).

³ <https://www.typescriptlang.org/>

⁴ <https://vuejs.github.io/vetur/>

COMPANION CODE

The entire companion code for the book can be found on **GitHub** at: <https://github.com/damianof/large-scale-apps-sample-project>

If you find any errors, or have difficulty completing any of the steps described in the book, please report them to me through the GitHub **issues** section here: <https://github.com/damianof/large-scale-apps-my-vue3-project/issues>

You are also free to reach out to me directly through Twitter at **@damianome**

Chapter 1

SETTING UP THE PROJECT WITH VUE-CLI

*IMPORTANT: This chapter assumes that you already have installed a recent version of **Node.js** on your computer. If you do not have it yet, you can download it here: <https://nodejs.org/en/download/>*

To set up the project you will use the terminal and the **vue-cli**⁵. If you do not have this installed on your computer yet, you can install it globally using **npm**:

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

Create Project Wizard

To create our project, do⁶:

```
vue create my-vue3-project
```

⁵ *vue-cli* version used at the time of this writing is 4.5.3

⁶ reference <https://cli.vuejs.org/guide/creating-a-project.html>

Next it will ask you to choose between these default presets:

- Default Vue 2
- Default Vue 3
- Manually Select Features

Select the **Manually Select Features** option (using arrows up/down) and hit enter:

```
Vue CLI v4.5.3
? Please pick a preset:
  Default ([Vue 2] babel, eslint)
  Default (Vue 3 Preview) ([Vue 3] babel, eslint)
> Manually select features
```

Next, it will ask you which features you want. Use the arrow keys to move up/down and press the space-bar to check/uncheck the features. For this project select only the **TypeScript, Router, Vuex, CSS Pre-processors, Unit Testing** features, then hit the Enter key:

```
? Check the features needed for your project:
```

- ☒ Choose Vue version
- ☐ Babel
- ☒ TypeScript
- ☐ Progressive Web App (PWA) Support
- ☒ Router
- ☒ Vuex
- ☒ CSS Pre-processors
- ☐ Linter / Formatter
- ☒ Unit Testing
- ☐ E2E Testing

Next, it will ask you which version of Vue you want to use. Use the arrow key to move up/down and select **3.x (preview)** by pressing the enter key:

```
? Choose a version of Vue.js that you want to start the project with
```

- 2.x
- ☒ 3.x (Preview)

It will then ask you a series of questions.

Make the following choices:

Use class-style component syntax? N (*NOTE: we will be using the **Vue Composition API style** in this book, but we will give an example on how to use the **class-style syntax** as well in one of the advanced chapters*)

```
? Use class-style component syntax? (y/N) N
```

Use Babel alongside TypeScript? No

```
? Use Babel alongside TypeScript (required for modern mode, auto-  
detected polyfills, transpiling JSX)? (y/N) N
```

Use history mode for router? No

```
? Use history mode for router? (Requires proper server setup for index  
fallback in production) (Y/n) n
```

Pick a CSS pre-processor: Sass/SCSS (with node-sass)

```
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are  
supported by default):  
  Sass/SCSS (with dart-sass)  
➤ Sass/SCSS (with node-sass)  
  Less  
  Stylus
```

Pick a unit testing solution: Mocha + Chai

```
? Pick a unit testing solution: (Use arrow keys)
> Mocha + Chai
  Jest
```

Where do you prefer placing config ... ? In dedicated config files

```
? Where do you prefer placing config for Babel, ESLint, etc.? (Use
arrow keys)
> In dedicated config files
  In package.json
```

The final choice will prompt you to save all the previous selections as a preset:

```
? Save this as a preset for future projects? (y/N) N
```


It's ok to say No here, but if later you wish to create another project with the exact same features as this one and not having to go through all the steps, then

go give it a name and save it.

*NOTE: At this point, it might also ask you if you want to use **yarn** or **npm** as your package manager. If so, please choose **npm** as that is what we'll be using in the book and thus will be easier to follow the examples. Of course, you are always free to choose **yarn** if that is your favorite tool.*

The **vue-cli** will now create the project, install all the required NPM packages, create the configuration files, and stub some preliminary code (**Home** and **About** views, a simple **HelloWorld** component, **Vuex** store, etc)

At the end it should display a message similar to this:

A terminal window with a dark background. The first line shows a success message with a party popper emoji: 'Successfully created project my-vue3-project.' The second line shows a hand pointing emoji followed by the text 'Get started with the following commands:'. Below this, two commands are listed, each preceded by a dollar sign: '\$ cd my-vue3-project' and '\$ npm run serve'.

```
🎉 Successfully created project my-vue3-project.  
👉 Get started with the following commands:  
  
$ cd my-vue3-project  
$ npm run serve
```

The first command will navigate to the current sub-directory called **my-vue3-project**, the second will serve the app with the **vue-cli-service**. You'll see a message similar to this displayed:

```
DONE Compiled successfully in 3767ms  
7:39:17 AM
```

```
App running at:
```

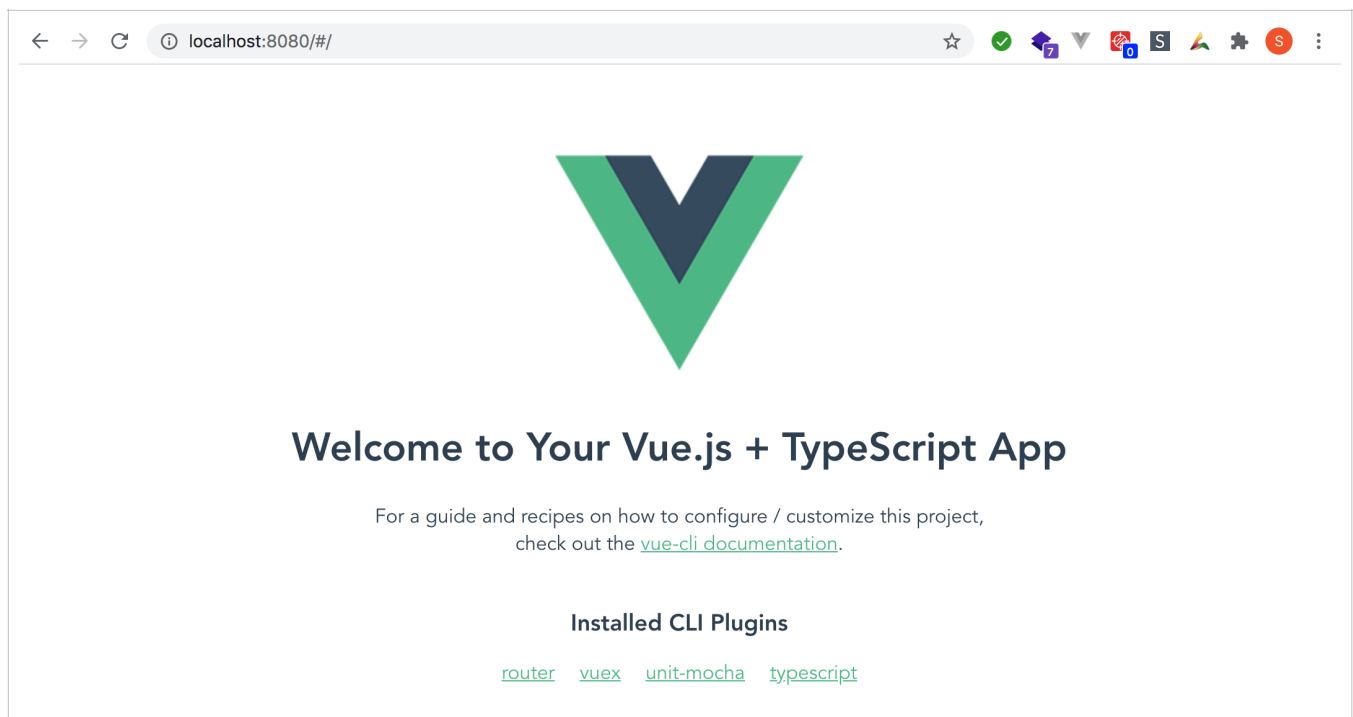
- Local: <http://localhost:8080/>
- Network: <http://192.168.1.3:8080/>

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

```
Issues checking in progress...  
No issues found.
```

NOTE: with vue-cli 4.5.3 I got a compile time error complaining about the router/index.ts Home component assignment type. If that should happen to you, just changed it like this “component: <any>Home” and should be able to pass that error. I did not experience this in existing Vue 2 projects upgraded to Vue 3 via “vue add vue-next”. Hopefully as the Vue team gets closer to the official Vue 3 release, issues like these will be better addressed.

From the web browser, navigate to the <http://localhost:8080/> address and you’ll see application home page rendered:



The **my-vue3-project** has been created with two stubbed views, **Home.vue** and **About.vue** and two routes have been added to the **Vue** router to allow navigation to those two views.

Differences Between Operating Systems

There are some difference between different operating systems in the way we set environment variables in the “scripts” commands of our **package.json** file.

You can see the use of the keyword **export**. This works on **OSX** and **Linux** environment, but **Windows** uses a different keyword and syntax.

For example this command:

```
"serve": "export VUE_APP_API_CLIENT=mock; vue-cli-service serve  
--mode development"
```

On Windows it would have to be converted to this:

```
"serve": "set \"VUE_APP_API_CLIENT=mock\" && vue-cli-service  
serve --mode development"
```

Note how the **export** keyword becomes **set** and the **semi-colon** becomes **&&**. It is also recommended to wrap the **variable=value** expression within escaped quotes, like in `\"VUE_APP_API_CLIENT=mock\"`

You could add additional shortcuts that are **Windows** specific by prefixing them with **win-** like here fore example:

```
"win-serve": "set \"VUE_APP_API_CLIENT=mock\" && vue-cli-service  
serve --mode development",
```

However, in this book and in the companion code on **GitHub** I opted to use the **cross-env**⁷ package. With this package, all I had to do is to update the commands to use the following syntax and stopped worrying about specific operating systems:

```
"serve": "cross-env VUE_APP_API_CLIENT=mock vue-cli-service serve  
--mode development",  
"build": "cross-env VUE_APP_API_CLIENT=live vue-cli-service build  
--mode production",  
"build-mock": "cross-env VUE_APP_API_CLIENT=mock vue-cli-service  
build --mode production"
```

Just remember to be aware that these differences exists if you do not use something like **cross-env**.

To install the package, first stop the app by typing **CTRL+C** and then execute this command:

⁷ <https://www.npmjs.com/package/cross-env>

```
npm install cross-env
```

Then update the commands within the **scripts** section of your **package.json** file as described above.

Chapter 1 Recap

What We Learned

- How to create the basic plumbing for a **Vue 3** app using the **vue-cli**
- How to serve the app using the **vue-cli** service through the command **npm run serve**

Observations

- The app has been created with a preliminary router, routes, and views
- The app does not do much yet, has only two very simple views with static html in them

Based on these observations, there are a few improvements that will be making into the next chapter:

Improvements

- Expand our app functionality by creating our first component

Chapter 2

YOUR FIRST COMPONENT

THE ITEMS LIST

Let's now pretend we have been giving requirements for our app to have a component that displays a list of “items”. We will keep this simple initially and as we move towards more advanced chapter expand on it to show how we can better structure our application to support:

- Quick prototyping and development using mocked data
- Component Organization
- Unit Testing
- State Management with Vuex
- Internationalization support so we can render our user interface using different languages
- Localization for number and date formatting for different cultures

ItemsList Component Requirements

Your initial version of the ItemsList component, will have to implement the following requirements (later, in more advanced chapters, we will expand on these as we get into more advanced topics):

- The component will display a list of items
- An item will have 3 properties:
 1. id
 2. name
 3. selected

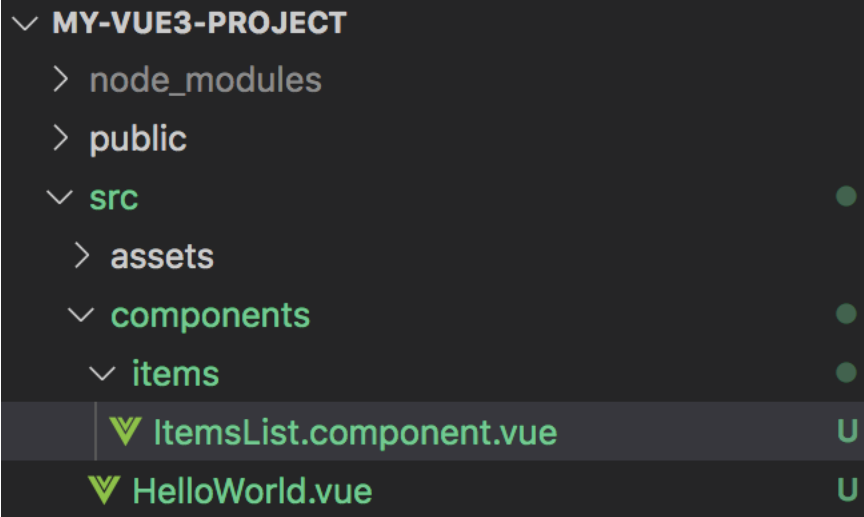
- The item name will be displayed to the user
- The user should be able to select/deselect one or more item
- An icon will be shown next to the name to indicate if the item is selected

ItemsList Component Code

Within the **src/components** directory, create a sub-directory called **items**.

Within this folder add a new file called **ItemsList.component.vue**⁸

Your directory structure will now look like this:



⁸ We are following a file naming convention where Vue components are pascal-case and follow this format `[ComponentName].component.vue` (Reference: Naming Conventions section at the end of this book)

Within the `ItemsList.component.vue` file, paste the following code:

```
<template>
  <div>
    <h3>Items:</h3>
    <ul>
      <li v-for="item in items" :key="item.id">
        {{ item.name }}
      </li>
    </ul>
  </div>
</template>

<script lang="ts">
  import { defineComponent, PropType } from 'vue'

  const ItemsListComponent = {
    props: {
      items: {
        type: Array as PropType<any[]>
      }
    }
  }

  export default defineComponent(ItemsListComponent)
</script>
```

A few things to notice here. First, we specify the `lang` attribute on the `<script>` element with the value `ts` so we can use **TypeScript**. For our html template, we added a `<h3>` element with hard-coded text just saying “**Items:**”. Then a `` with a v-for binding that will render all our `items` within `` elements.

Within the `<script>` section, we import **defineComponent** and **PropType** from **'vue'**.

We then declare the **items** property as an array of **any**⁹ for now, using Vue's **PropType** (later we'll replace **any** with an interface we'll create):

```
props: {  
  items: {  
    type: Array as PropType<any[]> // avoid using "any"  
  }  
}
```

⁹ With **'any'**, TypeScript does not enforce type-checking on a property or variable. However, this is considered a bad practice as we lose the main benefit of TypeScript. There might be exceptions to this rule when using older 3rd party packages/libraries/plugins that do not offer type definitions. However, even in those cases it would be strongly recommended to provide interfaces and types so that you can still avoid using **'any'**.

Home View

Within the **src/views** directory, open the existing **Home.vue** file. Replace the existing code with this:

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

<script lang="ts">

</script>
```

Start implementing our **Home** view by adding this inside the **<script>** section:

```
import { defineComponent } from 'vue'

const HomeView = {
  name: 'Home'
}

export default defineComponent(HomeView)
```

Then import a reference to the **ItemsList.component.vue** and register it through the **components** section (new and update code in yellow):

```
import { defineComponent } from 'vue'
import ItemsListComponent from '@components/items/
ItemsList.component.vue'

const HomeView = {
  name: 'Home',
  components: {
    ItemsListComponent
  }
}

export default defineComponent(HomeView)
```

For now, also quickly mock some data for our list of **items** that we will feed to our **ItemsListComponent**. For this we instantiate a local variable called **items** and initialize it with some hard-coded data¹⁰.

We do this in the **setup** method:

¹⁰ Note: using hard-coded data is a bad practice and here we are only doing it to first illustrate how things flow, and later in the next chapters will remove in favor of best practices and patterns (see Chapter 5)

```

const HomeView = {
  name: 'Home',
  components: {
    ItemsListComponent
  },
  setup() {

    const items: any[] = [{
      id: 1,
      name: 'Item 1'
    }, {
      id: 2,
      name: 'Item 3'
    }, {
      id: 3,
      name: 'Item 3'
    }]

    return {
      items
    }
  }
}

```

*NOTE: The items property is not **reactive** yet. In Vue 3 we'll have to explicitly make it reactive and we'll do that in Chapter 5 when we'll expand our ItemsListComponent while writing our first unit tests against a component.*

Finally, we add an `<ItemsListComponent>` element within the `<template>` markup. We can insert it within the `<div class="home">` element for now. Add an attribute called `:items` to our `<ItemsListComponent>`. This is how you do

one-way binding in Vue¹¹. The Home **items** property with the hard-coded data is fed into the component **items** property this way. The complete code within the **Home.vue** file should not look like this:

¹¹ In Vue, the colon prefix on a DOM element attribute is a shorthand for the `v-bind` directive. In this case **:items** works exactly as **v-bind:items** but we prefer the shorthand as this makes the code less cluttered. Reference: <https://vuejs.org/v2/guide/syntax.html#v-bind-Shorthand>

```
<template>
  <div class="home">
    <ItemsListComponent :items="items" />
  </div>
</template>
```

```
<script lang="ts">
  import { defineComponent } from 'vue'
  import ItemsListComponent from '@components/items/
ItemsListComponent.vue'
```

```
const HomeView = {
  name: 'Home',
  components: {
    ItemsListComponent
  },
  setup() {
```

```
    const items: any[] = [{
      id: 1,
      name: 'Item 1'
    }, {
      id: 2,
      name: 'Item 3'
    }, {
      id: 3,
      name: 'Item 3'
    }
  ]
```

```
    return {
      items
    }
  }
}
```

```
export default defineComponent(HomeView)
</script>
```

App.vue Updates

Now replace the `<template>` section within the **App.vue** file with this:

```
<template>
  <div id="app">
    <h2>My Vue 3 Project</h2>
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
    </div>
    <router-view />
  </div>
</template>
```

Replace also the `<script>` section with this:

```
<script lang="ts">
  import { defineComponent } from 'vue'

  export default defineComponent({
    name: 'App'
  })
</script>
```


And replace the `<style>` section with this:

```
<style lang="scss">
  #app {
    font-family: Avenir, Helvetica, Arial, sans-serif;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    text-align: center;
    color: #2c3e50;
    h2 {
      margin: 0;
    }
  }

  #nav {
    padding: 30px;
    a {
      font-weight: bold;
      color: #2c3e50;
      &.router-link-exact-active {
        color: #42b983;
      }
    }
  }
</style>
```

Save the App.vue file. The web browser will refresh and display our preliminary items list being rendered more or less like this:



localhost:8080/#/



[Home](#) | [About](#)

My Items:

- Item 1
- Item 3
- Item 3

Chapter 2 Recap

What We Learned

- How to create a basic component that displays a list of items
- How to consume that component in a view

Observations

- The **items** property within the **ItemsList.component.vue** is declared as an array of type **any**
- The **Home.vue** view contains hard-coded data (items) which is also declared as an array of **any**
- This means we are not leveraging strong-type checking at development time using TypeScript interfaces/models/types

Based on these observations, there are a few improvements that we will make in the next chapters:

Improvements

- Create a TypeScript interface called **ItemInterface** for enforcing type checking at development time for our **items** data
- Update our code so it uses the new **ItemInterface** interface

Chapter 3

DATA MODEL INTERFACES

In this chapter, we will keep building and improving our Vue project by starting to leverage TypeScript interfaces for strong-type checking at development time. One of the disadvantage of pure JavaScript is that is loosely typed, and this might cause issues at run-time as there are no checks on the type and or the expected properties of a value or object passed around through our code. TypeScript main advantage is the ability to enforce strong-type checking at development time through the use of interfaces, types, classes, and more.

Models Directory

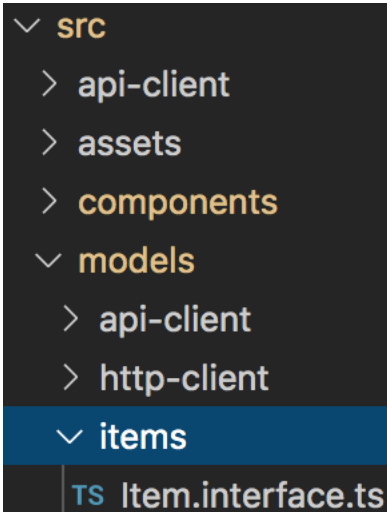
Start by creating a new sub-directory under **src** called **models**. Since the focus of this book is on building a foundation for large applications, we'll keep structuring our files and directories in a consistent way, following our own convention. You and your team are free to decide what your standards will be, but it's important to have both files/directory naming conventions and structuring in place as soon as you start building a large application. This will save you a lot of confusion and headaches later as the application grows exponentially and the number of source files and directories grows with it.

Interface ItemInterface

Create the directory **src/models/items** and here add a new TypeScript file called **ItemInterface.ts**. Note: we'll be following a naming convention for TypeScript files that represents the interface by adding the **Interface** suffix (in the first edition of this book I used the "I" prefix was like in **IItem** but this is definitely

harder especially on people coming from cultures other than the US which are less used to abbreviations and acronyms in code). In this version of the book, we'll save each interface into its own file with a file name convention like **Item.interface.ts**. (see the Naming Conventions section at the end of this book for more information)

Your directory structure should now look similar to this (ignore **api-client** and **http-client** directories for now, we'll add those later in other chapters):



```

✓ src
  > api-client
  > assets
  > components
  ✓ models
    > api-client
    > http-client
  ✓ items
    TS Item.interface.ts

```

Let's write an interface that represents one item that will be rendered in our **Item** component. Our interface will have three properties:

- **id**: this is a unique number for each item in the list
- **name**: is a string containing the name of the item
- **selected**: is a boolean value that shows if the user has selected the item

The code for your interface should look like this:

```
export interface ItemInterface {  
  id: number  
  name: string  
  selected: boolean  
}
```

For now, that is all we need. Since this will only represent a piece of data, we do not need to implement a class.

*NOTE: In this case our `ItemInterface` only holds fields, but no methods. You can think of this more like the type **struct** in language like C or C#. Unfortunately TypeScript does not have an explicit struct type¹² and their guidance is to use interfaces for this.*

ItemsList Component

Now that we have our interface, we can finally leverage TypeScript type checking ability by changing our items property on the items component from `any[]` to `ItemInterface[]`. First, import a reference for `ItemInterface`:

```
<script lang="ts">  
  import { defineComponent, PropType } from 'vue'  
  import { ItemInterface } from '@models/items/Item.interface'
```

¹² There have been suggestions presented, but I do not think they will ever add a struct type. See the TypeScript team answers here <https://github.com/microsoft/TypeScript/issues/22101>

Then modify our items property declaration from `any[]` to `ItemInterface[]`:

```
const ItemsListComponent = {  
  props: {  
    items: {  
      type: Array as PropType<ItemInterface[]>  
    }  
  }  
}
```

The complete update code should look like this:

```

<template>
  <div>
    <h3>Items:</h3>
    <ul>
      <li v-for="item in items" :key="item.id">
        {{ item.name }}
      </li>
    </ul>
  </div>
</template>
<script lang="ts">
  import { defineComponent, PropType } from 'vue'
  import { ItemInterface } from '@models/items/Item.interface'

  const ItemsListComponent = {
    props: {
      items: {
        type: Array as PropType<ItemInterface[]>
      }
    }
  }
  export default defineComponent(ItemsListComponent)
</script>

```

Make sure the terminal does not display any error, and that the web browser refreshed and no error are displayed in the browser console.

Home View

We should also update the **Home.vue** code so it uses the **ItemInterface**

interface for the locally private property also called **items**.

*Please note, that as soon as you change the items property from **any[]** to **ItemInterface[]** it will complain that each item does not correctly implement the interface. This is because we did not initially include the **selected** property required by the interface. This is one of the powerful things of using TypeScript correctly. It will help catching errors like this at development time rather than run time and increase the code quality and have it less prone to bugs.*

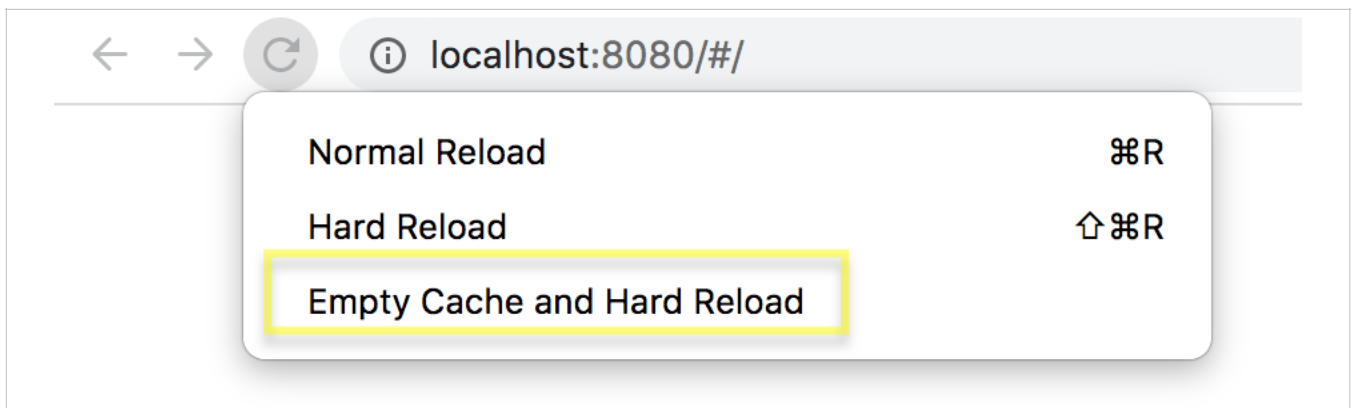
```
<script lang="ts">
  import { defineComponent } from 'vue'
  import ItemsListComponent from '@components/items/
ItemsListComponent.vue'
  import { ItemInterface } from '@models/items/Item.interface'
```

```
const HomeView = {
  name: 'Home',
  components: {
    ItemsListComponent
  },
  setup() {
    const items: ItemInterface[] = [{
      id: 1,
      name: 'Item 1',
      selected: false
    }, {
      id: 2,
      name: 'Item 3',
      selected: false
    }, {
      id: 3,
      name: 'Item 3',
      selected: false
    }]
  }
}
```

```
  return {
    items
  }
}
```

```
export default defineComponent(HomeView)
</script>
```

Again, make sure the terminal does not display any errors, and that the web browser refreshed and no error are displayed in the browser console. As you make changes is also a good idea occasionally to do an **Empty Cache and Hard Reload** by right clicking on the Chrome refresh icon and selecting the last option:



Chapter 3 Recap

What We Learned

- It's important to follow files and directories naming convention and structure convention
- How to leverage **TypeScript** interfaces and avoid using **any** so that strong-type checking is enforced at development time and avoiding potential runtime errors or hidden bugs

Observations

- The **Home.vue** contains a local property that holds hard-coded mocked data that enabled us to prototype our component quickly
- **ItemsList.component.vue** just display the list of items, but the user has still no ability to click on them to change their **selected** property

Based on these observations, there are a few improvements that we will make into the next chapter:

Improvements

- Update our component so that when a user clicks on an item displayed on the page, the item selected property will toggle from false to true (and vice versa)

Chapter 4

ADDING EVENTS TO THE ITEMS COMPONENT

In this chapter we keep building our **ItemsList.component.vue** so we can handle when the user clicks on an item in the list.

ItemsList Component

Start by updating the **<template>** section by adding a **@click**¹³ attribute to the **** element, pointing to an handler called **onItemClick** and passing the reference to the item as an argument:

```
<li v-for="item in items" :key="item.id"
@click="onItemSelect(item)">
  {{ item.name }}
</li>
```

Then within the **<script>** section inside our component class, add a function that implements our **onItemSelect** handler, toggles the **item.selected** property from true to false or vice versa and logs the item id and selected properties to the console for preliminary debugging. Note we have are doing this in the

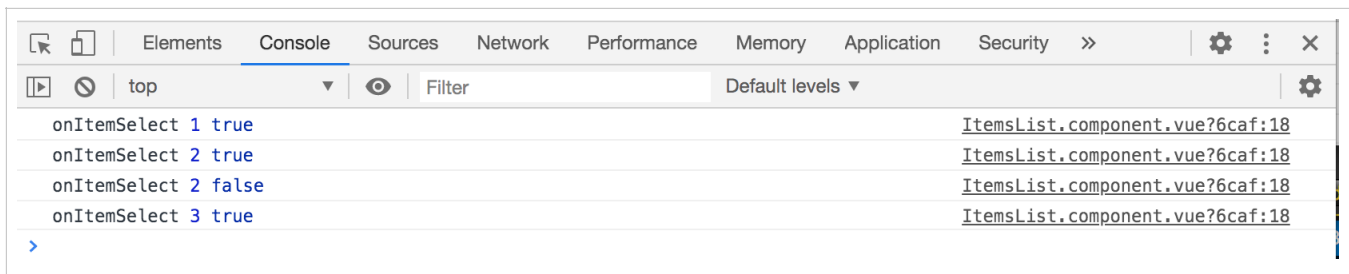
¹³ In Vue, the **@** prefix on a DOM element attribute is a shorthand for the **v-on** directive. Here, **:items** works exactly as **v-on:click**, but we prefer the shorthand as this makes the code less cluttered. Reference: <https://vuejs.org/v2/guide/syntax.html#v-on-Shorthand>

setup() method of the **Vue** component by setting the **onItemSelect** on a variable and returning an object that contains the method. There are other ways of doing this, but for now will stick to using the **setup()** method:

```
const ItemsListComponent = {
  props: {
    items: {
      type: Array as PropType<ItemInterface[]>
    }
  },
  setup() {
    const onItemSelect = (item: ItemInterface) => {
      item.selected = !item.selected
      console.log('onItemSelect', item.id, item.selected)
    }

    return {
      onItemSelect
    }
  }
}
```

Then, the web browser should have refreshed, and when clicking on the items in the list you should see the message being displayed in the browser developer console, and when clicking multiple time on the same item it should print true then false etc showing that toggling is working:



Now, we learned how to add a click handler to our component and changing the data item selected property that way. However, in an app that will grow large, have many components/views that will need to be aware of each other state, or the state of the data they manipulate, this is not the best pattern to use.

In the next chapter we'll introduce **Vuex** to manage our state in a more appropriate pattern where changes in state are done in a centralized place (State Management Pattern) and offers a way to separate state from views and actions.

Here is the official **Vuex** definition from the official website¹⁴:

*“Vuex is a **state management pattern** + **library** for **Vue.js** applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion. It also integrates with Vue's official **devtools extension** to provide advanced features such as zero-config time-travel debugging and state snapshot export / import.”*

¹⁴ Reference: <https://vuex.vuejs.org/>

Chapter 4 Recap

What We Learned

- How to add a click handler to our **ItemsList** component
- How to manipulate the **item.selected** property through our click handler

Observations

- The **items selected** property is being manipulated directly within our component
- We need a more centralized way to handle changes on the data and state of the application

Based on these observations, there are a few improvements that we will make in the next chapters:

Improvements

- Implement a **Vuex** state store and commit the changes to our items in a centralized place

Chapter 5

INTRO TO UNIT TESTING

WHILE REFACTORING A BIT

We will now see how to add some unit tests to our project. There are two main categories of unit tests that we will write in this book:

- unit tests for models/classes/structures/interfaces (i.e. Api client, Vuex mutations and actions, Api etc)
- unit tests for Vue components

Note: there is also a 3rd category called e2e (end to end) tests, but we will not be getting into those in this book.

Since we coded our first component already, we should start with the second category for now. We will add our first unit tests for a component and while doing so, we'll refactor our **ItemsList** component a little and we can use the unit tests to validate our changes.

Item Component

Remember how in our **ItemsList** component we have a **v-for** loop that creates individual **** for each item in our **items** property? Let's extract the code for the **** element and create a child component just for that. Let's start by adding a new file called **Item.component.vue** under the **src/components/items/children** directory:

```

  ▾ src
    > api-client
    > assets
    ▾ components
      ▾ items
        ▾ children
          ▽ Item.component.vue
          ▽ ItemsList.component.vue

```

Now, paste the following code at the top for the **template** section:

```

<template>
  <li :class="css" @click="onClick">
    <div class="selected-indicator">*</div>
    <div class="name">{{ model.name }}</div>
  </li>
</template>

```

For the **<script>** section paste the following code after the template section:

```
<script lang="ts">
  import { defineComponent, computed, PropType } from 'vue'
  import { ItemInterface } from '@models/items/Item.interface'

  const ItemComponent = {
    props: {
      model: {
        type: Object as PropType<ItemInterface>
      },
    },
    setup(props: any, { emit }: any) {

      const css = computed(() => {
        let css = 'item'
        if (props.model.selected) {
          css += ' selected'
        }
        return css.trim()
      })

      const onClick = () => {
        emit('select', props.model)
      }

      return {
        css,
        onClick
      }
    }
  }

  export default defineComponent(ItemComponent)
</script>
```

We just created a template for a single `` element and we enhanced this a little by including two child `<div>` elements:

- one to display the Item name
- one that will show a star icon (we are just using a char here, but you could use an icon library like material icons or similar).

Then we added a computed property called `css` that will return the string `“item”` or `“item selected”`. We then bind this to the `` `class` attribute through the `:class` binding, based on whether the `model.selected` property is true or false: `<li :class="css" @click="onClick">`

We also bind to the click event with `@click` binding and in the local `onClick` handler we just emit the custom `select` event and pass the model as the argument. We will then handle this in the parent component (`ItemsList` component).

Also add some custom CSS to make it look nicer:

```
<style lang="scss">
  li.item {
    padding: 0;
    outline: solid 1px #eee;
    display: grid;
    grid-template-columns: 1.7em auto;
    cursor: pointer;
    transition: background-color 0.3s ease;

    .name {
      padding: 5px;
      text-align: left;
    }
    .selected-indicator {
      padding: 5px;
      font-size: 3em;
      line-height: 0.5em;
      padding: 5px;
      color: lightgray;
    }
    &.selected {
      .selected-indicator {
        color: skyblue;
      }
    }
    &:hover {
      background-color: #eee;
    }
  }
</style>
```

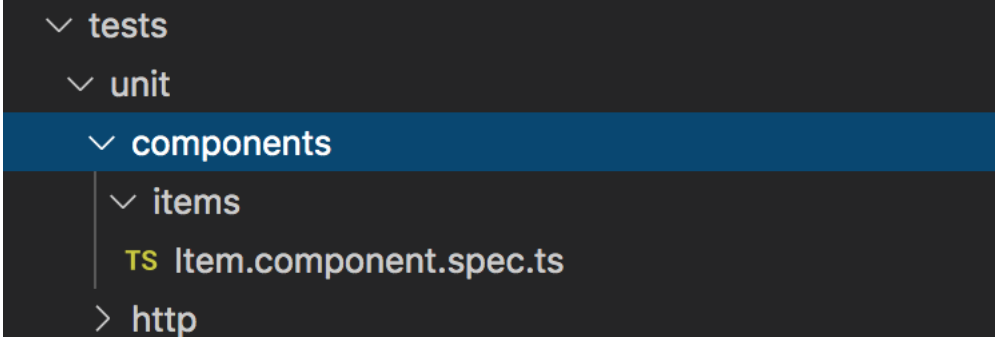
NOTE: I am not suggesting here that CSS should be part of the single .vue files, but for now just doing it this way because is the easiest way to style a component in our project without

using a css framework or similar technique. I might talk more about ways to add third-party CSS frameworks globally to our project in more advanced chapters in this book.

Item Component Unit Tests

Now, let's add a unit test for our newly created component.

Create the directory **tests/unit/components/items** and here add a file called **Item.component.spec.ts**¹⁵



```

  ∨ tests
    ∨ unit
      ∨ components
        ∨ items
          TS Item.component.spec.ts
        > http

```

Within the unit tests file, paste the following code:

¹⁵ It's customary to use the suffix `.spec` on file names that represent Mocha tests. Other testing framework also following similar convention, i.e. Jasmine

```

import { expect } from 'chai'
import { shallowMount } from '@vue/test-utils'
import ItemComponent from '@components/items/children/
Item.component.vue'
import { ItemInterface } from '@models/items/Item.interface'

describe('Item.component.vue', () => {

  it('renders an Item correctly', () => {

    const model: ItemInterface = {
      id: 1,
      name: 'Unit test item 1',
      selected: false
    }

    const wrapper = shallowMount(ItemComponent, {
      props: {
        model: model
      }
    })

    expect(wrapper.text()).to.include('Unit test item 1')

  })

})

```

Here we test that the component renders the data model properties as expected. For now, we are checking if the entire text rendered by the component contains the model.name. This is not very precise as our component later might render additional labels and our test might match these instead resulting in possible

false positives.

A better and more precise way to test what our component has rendered, is to use the `wrapper.find` utility to select specific html DOM elements and then check if those have render the expected text (or even check for other properties like if they exist, they are visible etc).

Run our unit tests from the terminal with this command:

```
npm run test:unit
```

It should run the unit tests and print the results on the terminal, similar to this:

```
DONE Compiled successfully in 2779ms
[=====] 100% (completed)

WEBPACK Compiled successfully in 2779ms
MOCHA Testing...

HelloWorld.vue
  ✓ renders props.msg when passed

Item.component.vue
  ✓ renders an Item correctly

2 passing (49ms)

MOCHA Tests completed successfully
```


Let's now improve our unit test so we can select the `<div>` with the class `"name"` and check that it renders our `model.name`:

```
import { expect } from 'chai'
import { shallowMount } from '@vue/test-utils'
import ItemComponent from '@components/items/children/
Item.component.vue'
import { ItemInterface } from '@models/items/Item.interface'
```

```
describe('Item.component.vue', () => {
```

```
  it('renders an Item correctly', () => {
```

```
    const model: ItemInterface = {
      id: 1,
      name: 'Unit test item 1',
      selected: false
    }
```

```
    const wrapper = shallowMount(ItemComponent, {
      props: {
        model: model
      }
    })
```

```
    // this just tests that the entire text rendered by the
    component somewhere rendered
```

```
    // 'Unit test item 1', but this is not very precise.
    expect(wrapper.text()).to.include('Unit test item 1')
```

```
    // this is more precise as we are selecting the div with the
    class name and check if it rendered the correct text
```

```
    let domEl = wrapper.find('div.name')
    expect(domEl.text()).to.equal('Unit test item 1')
```

```
  })
```

```
})
```

Let's add two more tests within the same **describe** section to check that the component has the expected CSS classes:

```
it('has expected css class when selected is false', () => {
```

```
  const model: ItemInterface = {
    id: 2,
    name: 'Unit test item 2',
    selected: false
  }
```

```
  const wrapper = shallowMount(ItemComponent, {
    props: {
      model: model
    }
  })
```

```
  // check component css classes list
  const classes = wrapper.classes()
  expect(classes).to.be.an('array')
    .that.includes('item')
```

```
  expect(classes).to.be.an('array')
    .that.does.not.include('selected')
})
```

```

it('has selected css class when selected is true', () => {
  const model: ItemInterface = {
    id: 3,
    name: 'Unit test item 3',
    selected: true /* setting selected = true here */
  }

  const wrapper = shallowMount(ItemComponent, {
    props: {
      model: model
    }
  })

  // check component css classes list
  const classes = wrapper.classes()
  expect(classes).to.be.an('array')
    .that.includes('item')

  expect(classes).to.be.an('array')
    .that.includes('selected')
})

```

We could also trigger a click with **wrapper.trigger('click')** but our component only emits the event to be handled by a parent component, so we might have to do that in a unit tests that will write for our ItemsList component later in the book. For now, let's stop here with unit tests, we'll get into more advanced testing later in the book.

If you execute **npm run test:unit** again, this time should print results similar to these:

```
Item.component.vue
  ✓ renders an Item correctly
  ✓ has expected css class when selected is false
  ✓ has selected css class when selected is true

3 passing (29ms)
```

ItemsList component

We now have to change our **ItemsList.component.vue** so that it uses our newly created **Item.component.vue** in place of the `` element. Here are the changes required in our **ItemsList.component.vue** code:

- Import a reference to our Item component
- Add a reference within the **components** section
- Remove the existing `` code within the `<template>` section and replace it with our imported Item component

Here is the `<template>` section with the highlighted changes:

```
<template>
  <div>
    <h3>Items:</h3>
    <ul>
      <li v-for="item in items" :key="item.id"
@click="onItemClick(item)">
        {{ item.name }}
      </li>
      <ItemComponent v-for="item in items"
        :key="item.id"
        :model="item"
        @select="onItemSelect" />
    </ul>
  </div>
</template>
```

Here is the **<script>** section with the highlighted changes:

```

<script lang="ts">
  import { defineComponent, PropType } from 'vue'
  import { ItemInterface } from '@models/items/Item.interface'
  import ItemComponent from '@components/items/children/
Item.component.vue'

  const ItemsListComponent = {
    components: {
      ItemComponent
    },
    props: {
      items: {
        type: Array as PropType<ItemInterface[]>
      }
    },
    setup() {
      const onItemSelect = (item: ItemInterface) => {
        item.selected = !item.selected
        console.log('onItemSelect', item.id, item.selected)
      }

      return {
        onItemSelect
      }
    }
  }

  export default defineComponent(ItemsListComponent)
</script>

```

And add some custom css here to make the whole `` looks a little better:


```
<style lang="scss">
  ul {
    list-style-type: none;
    margin-block-start: 0;
    margin-block-end: 0;
    margin-inline-start: 0px;
    margin-inline-end: 0px;
    padding-inline-start: 0px;
  }
</style>
```

If you refresh the web browser, the list should render with the new style applied. However, clicking on the items does not seem to change their state in the UI. This is because we did not make the **items** property in our **Home** page **reactive** yet.

Open **Home.vue** and make these changes. Add **reactive** to the import list from **'vue'**, and then wrap the mock data array with **reactive(...)**:

```
<script lang="ts">
  import { defineComponent, reactive } from 'vue'
  import ItemsListComponent from '@components/items/
ItemsListComponent.vue'
  import { ItemInterface } from '@models/items/Item.interface'
```

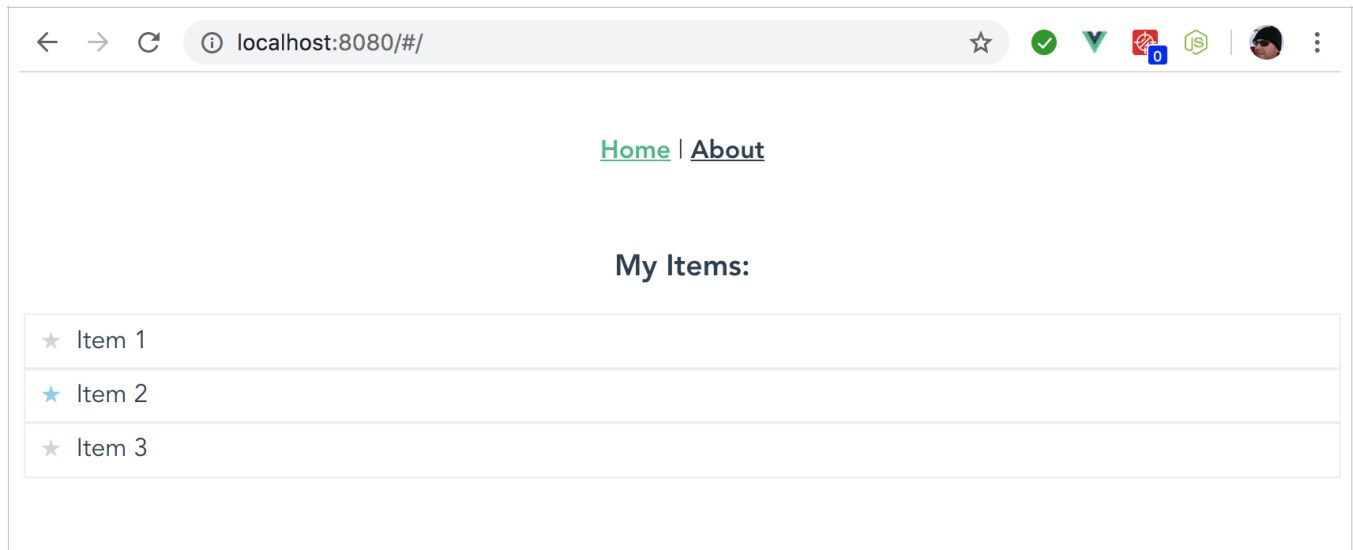
```
const HomeView = {
  name: 'Home',
  components: {
    ItemsListComponent
  },
  setup() {
```

```
    const items: ItemInterface[] = reactive([
      {
        id: 1,
        name: 'Item 1',
        selected: false
      }, {
        id: 2,
        name: 'Item 3',
        selected: false
      }, {
        id: 3,
        name: 'Item 3',
        selected: false
      }
    ])
```

```
    return {
      items
    }
  }
}
```

```
export default defineComponent(HomeView)
</script>
```

From the web browser, now the list should render similar to this (here we are showing it with the 2nd item element being clicked):



Chapter 5 Recap

What We Learned

- How to write unit tests against a component
- How to test that DOM elements render specific text, or have specific attributes like CSS classes, etc.
- How to re-factor parts of a component to create a child component and use unit tests to validate our changes

Observations

- We did not test user interaction without component

Based on these observations, there are a few improvements that will make in the next chapters when we get into more advanced unit tests:

Improvements

- Add unit tests also for events like click etc to test user interaction

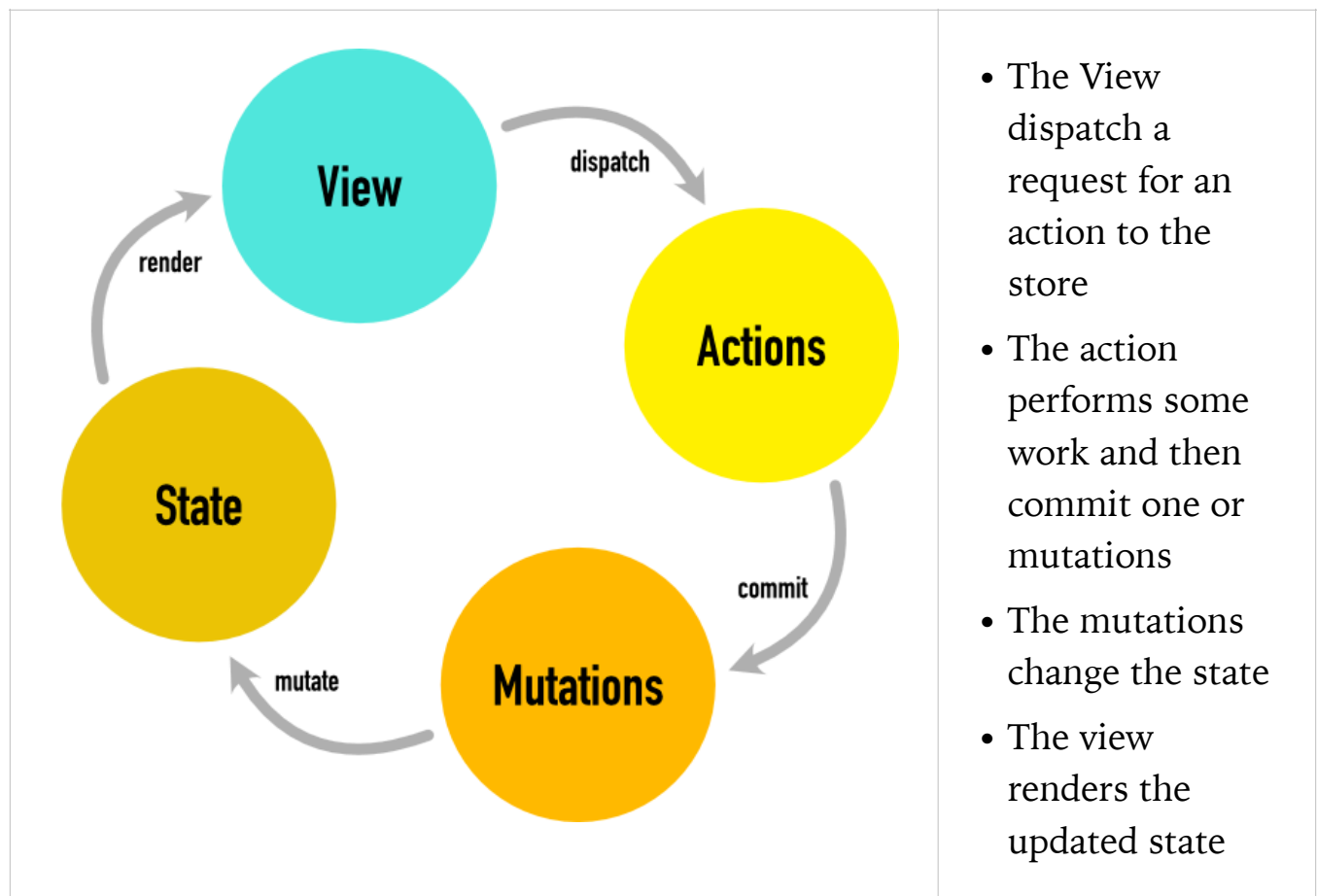
Chapter 6

INTRODUCING VUEX

As mention at the end of the previous chapter, **Vuex** offer a **State Management Pattern** to keep views/state/actions separated and allow to manage state changes from a centralized place, the store, which becomes the single source of truth for our application state.

Vuex Overview

The main concept around **Vuex** State Management Pattern is that of “one-way data flow” as illustrated in the following diagram:



Main Vuex Store (file: store/index.ts)

When we initially created our project in Chapter 1, the **vue-cli** also created a preliminary stub for the **Vuex** state under the **src/store** directory, file **index.ts**

If you open this file, the current content should look like this:

```
import { createStore } from 'vuex'

export default createStore({
  state: {
  },
  mutations: {
  },
  actions: {
  },
  modules: {
  }
})
```

Let's add some code to implement a simple store that enables to keep track of our data and state and its mutations in one place.

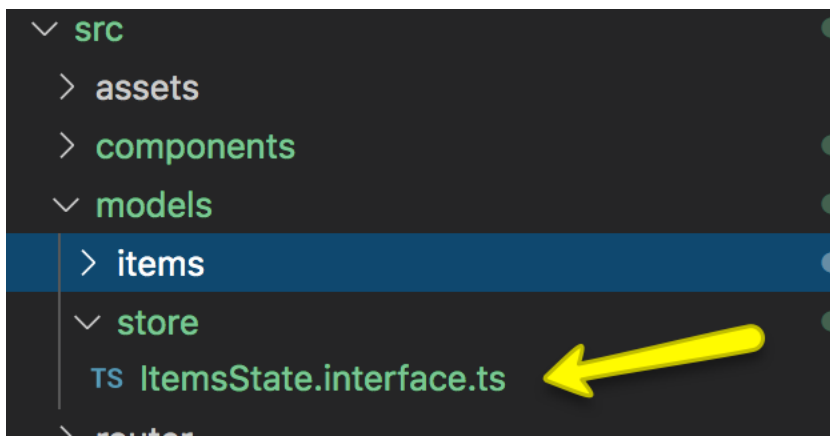
First, we need to add a property to hold our items, which will be an array of items. For this we could keep building on the stubbed code above:

```
state: {  
  items: []  
},
```

However, remember how in Chapter 4 we introduced the use of TypeScript interfaces to enforce type checking at development time and avoid possible run-time errors. In this book our focus is to build a strong foundation that can better serve building large-scale applications. So let's take a step back and add an interface that will represent our entire state for the items list data and component.

Interface ItemsStateInterface

Create directory **src/models/store** and here add a file called **ItemsState.interface.ts**. Our directory structure should look like this:



Inside the **ItemsState.interface.ts** file, paste the following code:

```
import { ItemInterface } from '@models/items/Item.interface'

/**
 * @name ItemsStateInterface
 * @description
 * Interface for the Items state
 */
export interface ItemsStateInterface {
  loading: boolean
  items: ItemInterface[]
}
```

As you can see in the code above, our state will have a property called **items** that will hold our items data, and a **loading** property which we'll use to toggle UI state and show whether or not our data is loading, or similar operation.

Main Vuex Store (File: store/index.ts)

Now we have to import a reference for the new interface into the **store/index.ts** file, and change the stubbed state with an instance of our **ItemsStateInterface** that represents the initial state:


```
import { createStore } from 'vuex'
import { ItemInterface } from '@models/items/Item.interface'
import { ItemsStateInterface } from '@models/store/ItemsState.interface'

// our initial state:
const state: ItemsStateInterface = {
  loading: false,
  items: []
}

export default createStore({
  state: state,
  mutations: {
  },
  actions: {
  },
  modules: {
  }
})
```

Now we need to create some **actions** that allow mutating our state. In **Vuex**, actions are simply functions that perform some tasks like loading data from an API end-point and then commit mutations to the state. First, we need an action that fires the loading of our data. We'll call this **loadItems**:

```
mutations: {  
  },  
  actions: {  
    loadItems({ commit, state }) {  
      commit('loadingItems')  
    }  
  },  
}
```

NOTE: in the next chapter we'll be replace hard-coded strings like 'loadingItems' with constant names in an organized way.

This action will commit a simple mutation called **loadingItems**. For this to work, we need to add a mutation function with the same name within the **mutations** section. Within the mutation, we can clear existing data from our **items** array and set the **loading** property to true. So add the following function within the mutations block:

```
mutations: {  
  loadingItems(state: ItemsStateInterface) {  
    state.loading = true  
    state.items = []  
  }  
},
```

Now, we need to add more code within our action **loadItems** so we can load our

mocked data. Let's get the hard-coded data we initially stored within our **Home.vue** code and put it within our **loadItems** action. This time lets store it into a variable called **mockItems** just after our **commit** line:

```
actions: {  
  loadItems({ commit, state }) {  
    commit('loadingItems')  
  
    // mock some data  
    const mockItems: ItemInterface[] = [{  
      id: 1,  
      name: 'Item 1',  
      selected: false  
    }, {  
      id: 2,  
      name: 'Item 3',  
      selected: false  
    }, {  
      id: 3,  
      name: 'Item 3',  
      selected: false  
    }  
  ]  
}
```

...

Then lets use a **setTimeout** to simulate a delay and commit a new mutation called **loadedItems** in which we also passed our mockItems data as the second argument:

```

...

// let's pretend we called some API end-point
// and it takes 1 second to return the data
// by using javascript setTimeout with 1000 for the
milliseconds option
setTimeout(() => {
  commit('loadedItems', mockItems)
}, 1000)

```

Also add a new function called **loadedItems** within the **mutations** block where we set the **state.items** to the loaded data, and the **loading** property back to false:

```

mutations: {
  loadingItems(state: ItemsStateInterface) {
    state.loading = true
    state.items = []
  },
  loadedItems(state: ItemsStateInterface, items:
ItemInterface[]) {
    state.items = items
    state.loading = false
  }
},

```

The complete code including the new **loadingItems** mutation will look like this:

```
import { createStore } from 'vuex'
import { ItemInterface } from '@models/items/Item.interface'
import { ItemsStateInterface } from '@models/store/
ItemsState.interface'
```

```
// our initial state:
const state: ItemsStateInterface = {
  loading: false,
  items: []
}
```

```
export default createStore({
  state: state,
  mutations: {
    loadingItems(state: ItemsStateInterface) {
      state.loading = true
      state.items = []
    },
    loadedItems(state: ItemsStateInterface, items:
ItemInterface[]) {
      state.items = items
      state.loading = false
    }
  },
  ...
```

```

...
actions: {
  loadItems({ commit, state }) {
    commit('loadingItems')

    // mock some data
    const mockItems: ItemInterface[] = [{
      id: 1,
      name: 'Item 1',
      selected: false
    }, {
      id: 2,
      name: 'Item 3',
      selected: false
    }, {
      id: 3,
      name: 'Item 3',
      selected: false
    }]

    // let's pretend we called some API end-point
    // and it takes 1 second to return the data
    // by using javascript setTimeout with 1000 for the
    milliseconds option
    setTimeout(() => {
      commit('loadedItems', mockItems)
    }, 1000)
  }
},
modules: {
}
})

```

Home View

In our `src/views/Home.vue`, we need to re-work a bit our `items` property.

Remove the hard-coded data and change this to a **computed** property that just return the data from the store state:

```

<script lang="ts">
  import { defineComponent, reactive, computed } from 'vue'
  ...

const HomeView = {
  ...
  setup() {

    const items: ItemInterface[] = reactive([
      {
        id: 1,
        name: 'Item 1',
        selected: false
      }, {
        id: 2,
        name: 'Item 3',
        selected: false
      }, {
        id: 3,
        name: 'Item 3',
        selected: false
      }
    ])

    const items = computed(() => {
      return store.state.items
    })

    return {
      items
    }
  }
}

```

Also, dispatch our **loadItems** action in the view mounted event handler so it

will let our **Vuex** store know that we want to load our items. The complete code for the `<script>` section will be like this:

```
<script lang="ts">
  import { defineComponent, computed, onMounted } from 'vue'
  import store from '@store'
  import ItemsListComponent from '@components/items/
ItemsList.component.vue'
  import { ItemInterface } from '@models/items/Item.interface'

  const HomeView = {
    name: 'Home',
    components: {
      ItemsListComponent
    },
    setup() {
      const items = computed(() => {
        return store.state.items
      })

      onMounted(() => {
        store.dispatch('loadItems')
      })

      return {
        items
      }
    }
  }

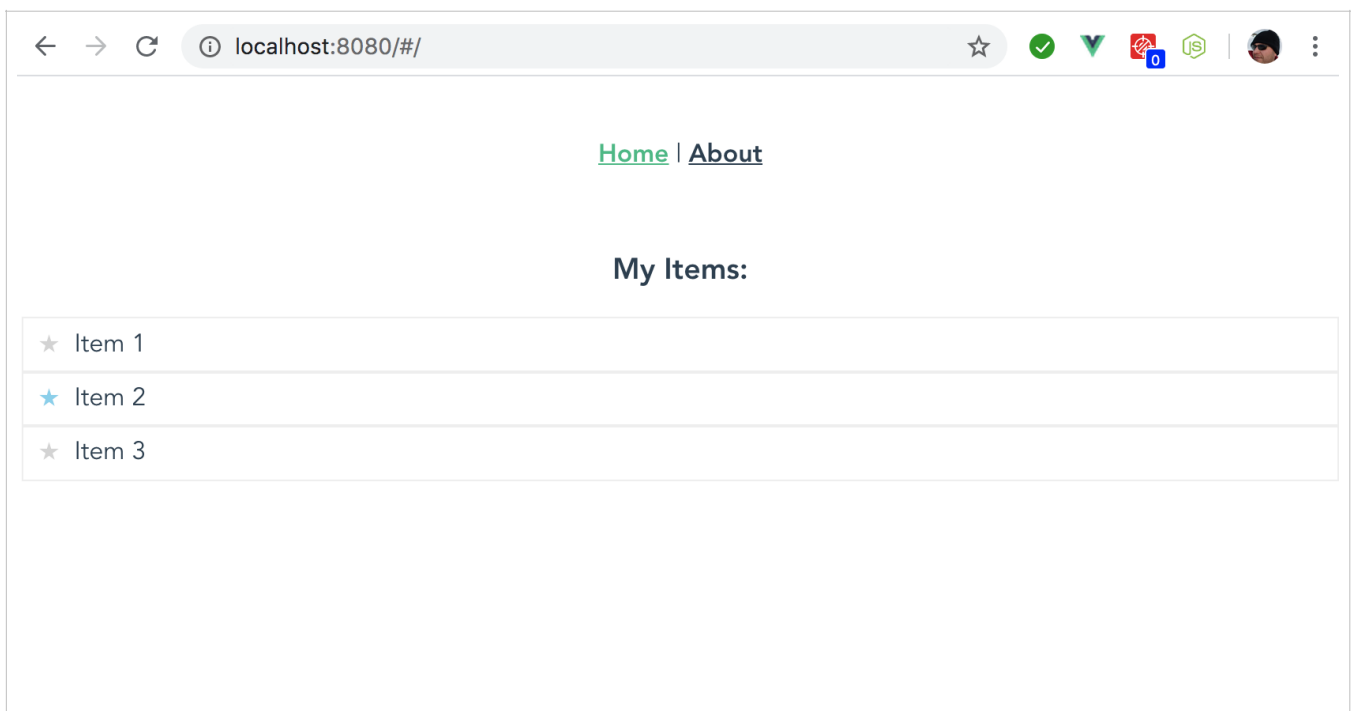
  export default defineComponent(HomeView)
</script>
```

When the user will browse to our Home view, the view will dispatch a request for action **loadItems** to our **Vuex** store. In turn, our store will:

1. commit a **loadingItems** mutation where we clear the **items** array and set the **loading** property to true
2. load the items data and commit a **loadedItems** mutation where we set our **items** property to the data loaded, and set the **loading** property to false

Web Browser

Refresh the Home page on the web browser and notice that now it will take about 1 second before the items will be rendered. However, there is no way to tell what it's happening during that second.



So let's enhance a bit more the **ItemsList.component.vue** code so we can start getting some feedback in the UI.

ItemsList.component.vue

Add a property called **loading** of type **boolean** to the **ItemsList** component:

```
const ItemsListComponent = {
  components: {
    ItemComponent
  },
  props: {
    items: {
      type: Array as PropType<ItemInterface[]>
    },
    loading: {
      type: Boolean
    }
  },
  ...
```

Now within the **<h3>** element, add a one-way binding using the double curly braces to print out the value of the **loading** property:

```
<template>
  <div>
    <h3>Items - loading: {{ loading }}:</h3>
    ...
  </div>
</template>
```

Home.vue

In the **Home** view, now we have to add a computed property called **loading** that returns the value from our **state.loading** and bind it to **:loading** attribute of the **ItemsListComponent**:

```
<template>
  <div class="home">
    <ItemsListComponent
      :items="items"
      :loading="loading" />
    </div>
  </template>
```

```
<script lang="ts">
  ...
  const HomeView = {
    ...
    setup() {
      const items = computed(() => {
        return store.state.items
      })
      const loading = computed(() => {
        return store.state.loading
      })
```

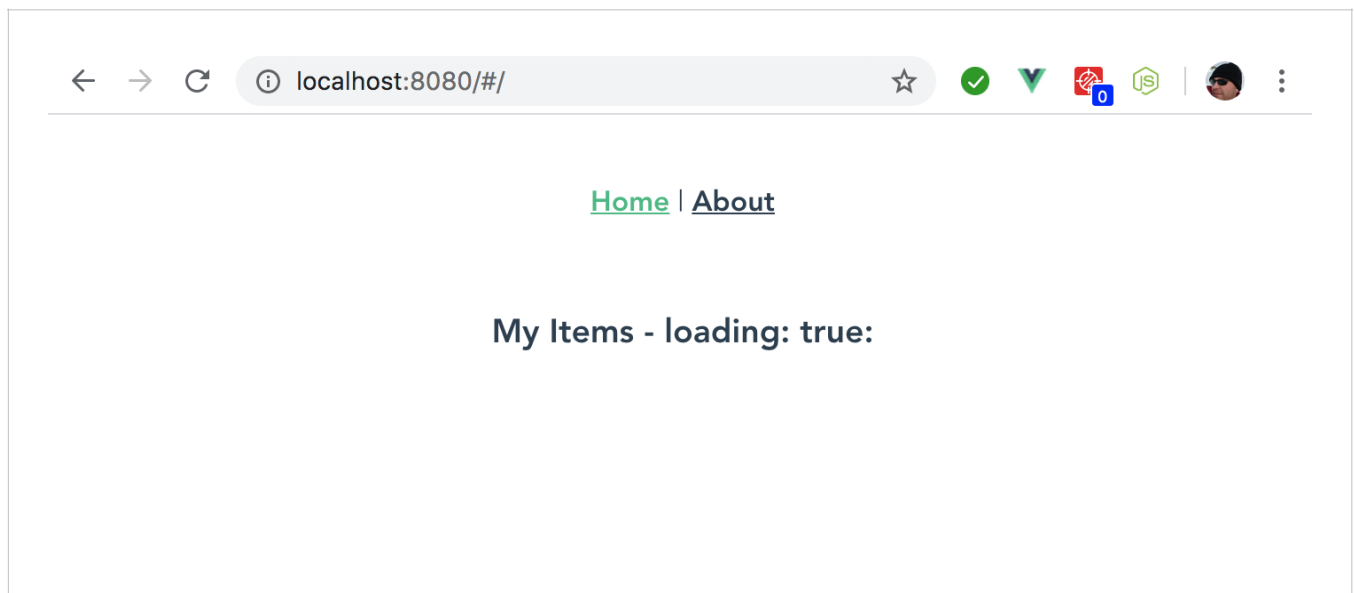
```
      onMounted(() => {
        store.dispatch('loadItems')
      })
```

```
      return {
        items,
        loading
      }
    }
  }
}
```

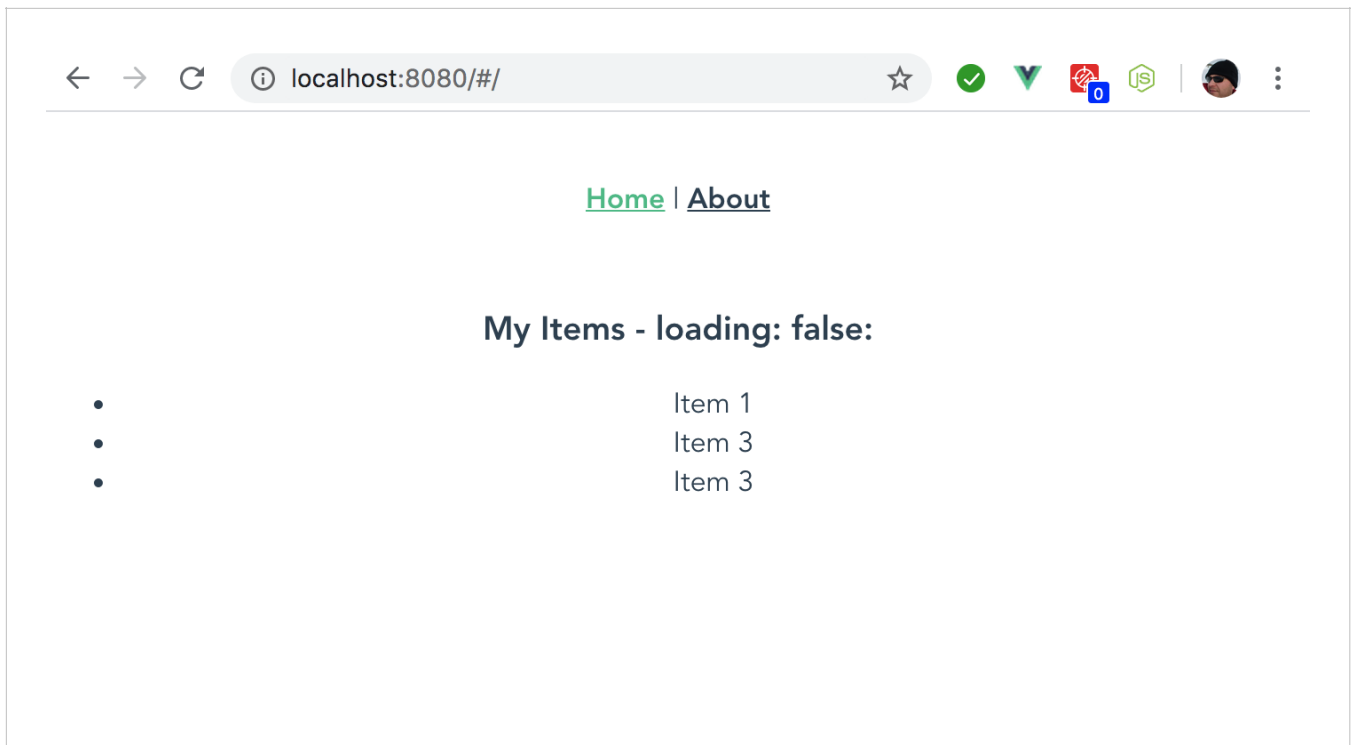
```
export default defineComponent(HomeView)
</script>
```

Web Browser

Now, when we refresh the browser, we'll first see a blank list, but in the header we'll see the text **My items - loading: true**:



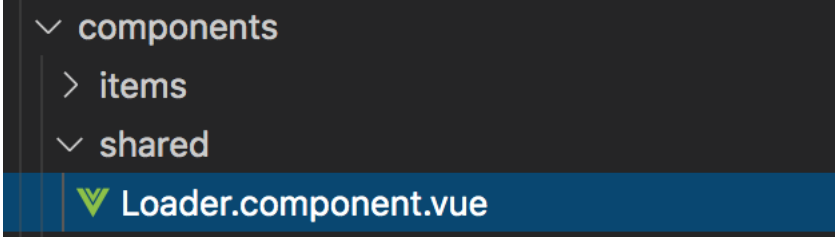
After 1 second the items will render and the h3 element will display the text **My items - loading: false**:



Let's take it a step further, let's create a **loader** component that will be consumed throughout the application and instead of just displaying the value of our loading property, we could use that with a **v-show** binding to so that the loader will be displayed in the list area while the data is loading.

Loader Component

Under the **src/components** create a sub-directory called **shared**. Within this directory create a file called **Loader.component.vue**. Your directory structure should now look like this:



```

  components
    items
    shared
  Loader.component.vue

```

Within the **Loader.component.vue** file, paste the following code:

```

<template>
  <div class="loader" v-show="show">
    <div class="bounceball"></div>
  </div>
</template>

<script lang="ts">
  import { defineComponent } from 'vue'

  const LoaderComponent = {
    props: {
      show: {
        type: Boolean
      }
    }
  }

  export default defineComponent(LoaderComponent)
</script>

```


Also add this <style> section at the bottom of the file:

```
<style lang="scss">
  .loader {
    $color: #42b983;
    $width: 30px;
    $height: 30px;
    $bounceHeight: 60px;
    display: inline-block;

    .bounceball {
      position: relative;
      width: $width;
      &:before {
        position: absolute;
        content: '';
        top: 0;
        width: $width;
        height: $height;
        border-radius: 50%;
        background-color: $color;
        transform-origin: 50%;
        animation: bounce 500ms alternate infinite ease;
      }
    }
  }
  ...

```

```

...
@keyframes bounce {
  0% {
    top: $bounceHeight;
    height: 10px;
    border-radius: 60px 60px 20px 20px;
    transform: scaleX(2);
  }
  25% {
    height: $height;
    border-radius: 50%;
    transform: scaleX(1);
  }
  100% {
    top: 0;
  }
}
}
</style>

```

This provides a basic loader that uses pure CSS for the animation. You are free to use an animated **gif**, or **svg** image, or **font-icon** etc.

All the needed CSS (we are using SASS here, as you can see from the style **lang** attribute value **scss**) is contained within the same file¹⁶.

¹⁶ You do not have to include CSS as part of the component if you do not want to, or you already use a CSS framework with centralized files

ItemsList Component

Now, lets go back into our **ItemsList.component.vue** code and import a reference to our new **Loader** component:

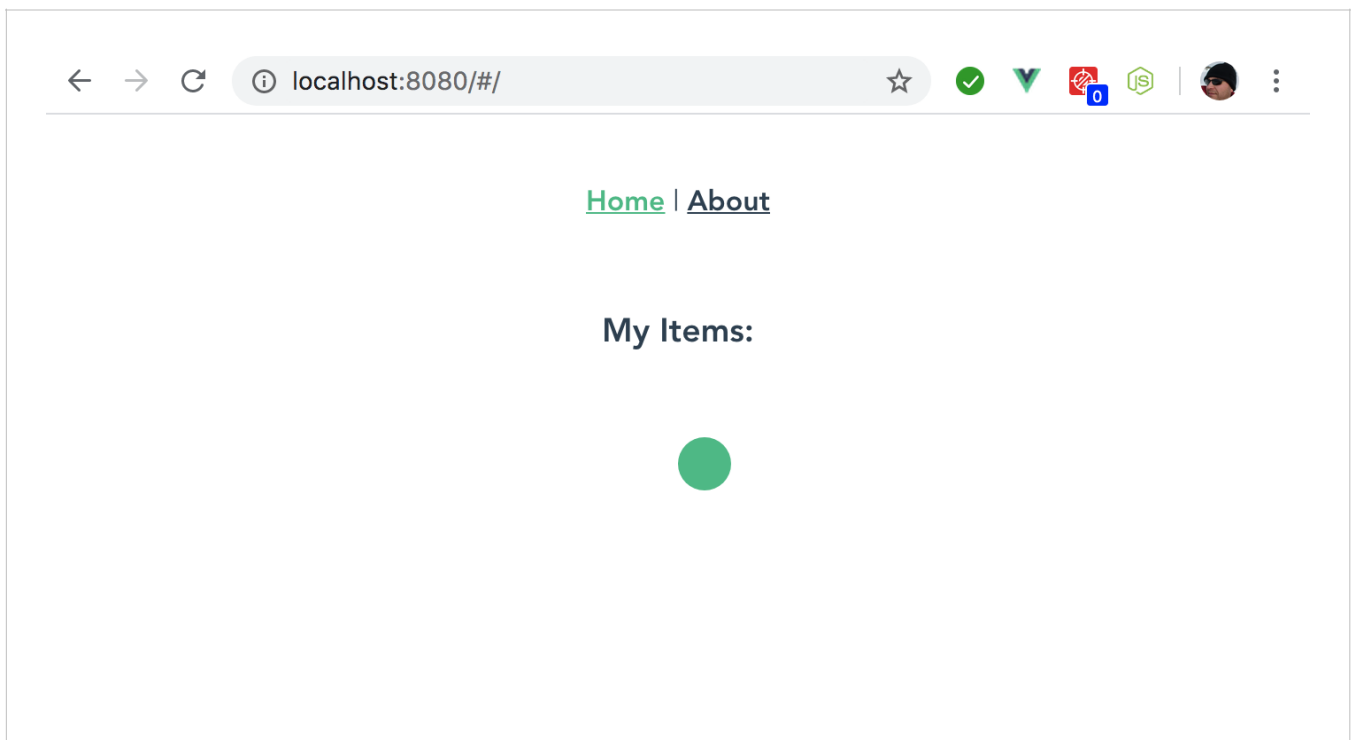
```
<script lang="ts">
  import { defineComponent, PropType } from 'vue'
  import { ItemInterface } from '@/models/items/Item.interface'
  import ItemComponent from '@/components/items/children/
Item.component.vue'
  import Loader from '@/components/shared/Loader.component.vue'

  const ItemsListComponent = {
    components: {
      ItemComponent,
      Loader
    },
    ...
```

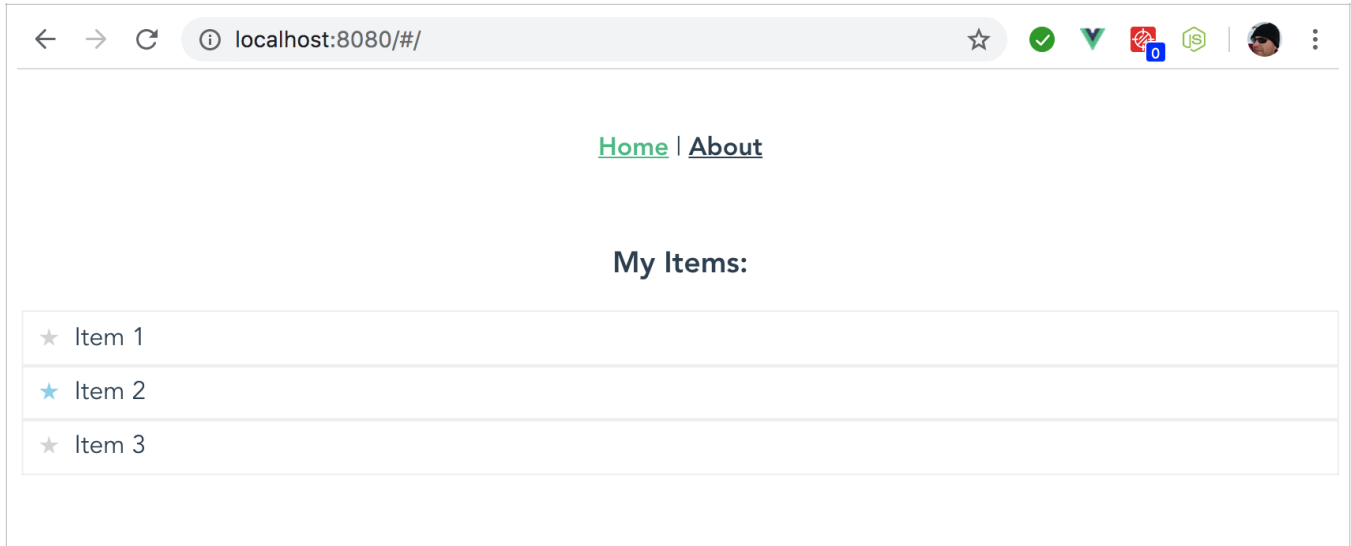
And within the **<template>** section, add an instance of **<Loader>** and set its **show** property to the **loading** property. Also add a **v-show** binding to **!loading** to our **** element so that is not visible when **loading** is false, and vice versa:

```
<template>
  <div>
    <h3>My Items —loading: {{ loading }}</h3>
    <Loader v-show="loading" />
    <ul v-show="!loading">
      <ItemComponent v-for="item in items"
        :key="item.id"
        :model="item"
        @select="onItemSelect" />
    </ul>
  </div>
</template>
```

Now refresh the web page you'll see our loader showing for about 1 second before it renders the items:



Then the loader will hide and the items list is rendered:



ItemsList Component enhancing Item click handler

Now we need to do also one more thing. Remember how in chapter 4 our Item component did not exist yet and in the ItemsList component we were just updating each item **selected** property directly in our **onItemSelect** handler there? Here is the old code:

```

...

setup() {
  const onItemSelect = (item: ItemInterface) => {
    item.selected = !item.selected
    console.log('onItemSelect', item.id, item.selected)
  }
}

...

```

As a general rule, we should never change state outside our **Vuex** store. We need to change the code above to notify our parent view (Home.vue) that an item is being selected. Then the **Home.vue** will dispatch the **selectItem** action to our **Vuex** store.

Let's change the code so we emit a custom event called **selectItem**. Remove the lines in red, and add the lines in yellow:

```

setup({ emit }: any) {
  const onItemSelect = (item: ItemInterface) => {
    item.selected = !item.selected
    console.log('onItemSelect', item.id, item.selected)
    emit('selectItem', item)
  }
}

```

Now in the **Home.vue** code, within the template section, we need to add an event handler for **selectItem** custom event to the **ItemsListComponent** element:

```
<template>
  <div class="home">
    <ItemsListComponent
      :items="items"
      :loading="loading"
      @selectItem="onSelectItem" />
    </div>
  </template>
```

And within our **setup()** we need to also add a method called **onSelectItem** where we'll dispatch the action to our **Vuex** store:

```

    setup() {
      ...

      const onSelectItem = (item: ItemInterface) => {
        store.dispatch('selectItem', {
          id: item.id,
          selected: !item.selected
        })
      }

      return {
        items,
        loading,
        onSelectItem
      }
    }

    ...

```

Note how we pass a parameter that is an anonymous object with two properties. The id of the item, and the new selected value which we create by toggling the current item.selected property by setting to either false or true depending on the current item.selected value:

```

{
  id: item.id,
  selected: !item.selected
}

```


Now when you click on an item in the browser, it should log a **JavaScript error** in the console:

```
[vuex] unknown action type: selectItem
```

This is because we did not add a **selectItem** action to our **Vuex** store yet. So let's do that.

Vuex Store: selectItem Mutation

Within in our **store/index.ts** code we need to add a new mutation called **selectItem**:

```

mutations: {
  ...

  selectItem(state: ItemsStateInterface, params: {
    id: number,
    selected: boolean
  }) {
    const { id, selected } = params
    const item = state.items.find(o => o.id === id)
    if (item) {
      item.selected = selected
    }
  }
}
  ...

```

Vuex Store: selectItem Action

Now add the **selectItem** action which will receive our anonymous type with **id** and **selected** property through the **params** argument. We extract the **id** and **selected** values from our **params**, commit the **selectingItem** mutation (which sets `state.selectingItem` to `true`), then find the item instance by `id` from our `state.items` and update its **selected** property. Finally, commit the **selectedItem** mutation (which sets the `state.selectedItem` back to `false`):

```
actions: {
```

```
  ... ,
```

```
  selectItem({ commit }: any, params: {
```

```
    id: number
```

```
    selected: boolean
```

```
  }) {
```

```
    commit('selectItem', params)
```

```
  }
```

```
  ...
```

```
  ...
```

Now if refresh the browser and click on the items, the **item.selected** property will be updated through the **Vuex** state.

Chapter 6 Recap

What We Learned

- How to create a **TypeScript** interface for our Item state to enforce type checking at development time
- How to use **Vuex** to manage our **Items** state
- How to create **Vuex actions**
- How to create **Vuex mutations**
- How to commit **state mutations** from within the **actions**
- How to **dispatch** (invoke) a state action when our view or component is mounted
- How to use a **loading** property on our state to provide feedback to the user about long-running processes through a **loader**
- How to create a simple **Loader** component

Observations

- We are using hard-coded strings for our mutations names, which is considered a bad practice in a large-scale application
- We are still using hard-coded data (**mockItems** within our **store/index.ts** file), instead of loading the data through an API client

Based on these observations, there are a few improvements we will make in the next chapters:

Improvements

- Replace the hard-coded string values for the state mutations keys and replace them with TypeScript constants

- Create an API client that can serve mocked data for quick front-end development and prototyping, and an API client that can communicate with real API end-points

Chapter 7

API CLIENT

So far we have worked with **Vuex** and managed the app state through our **Vuex** store. However, we are still “pretending” to load data by using a **mockItems** variable with hard-coded data within our store **loadItems** action, and using the **setTimeout** trick to add a bit of delay before returning the data (so we have at least 1 second to show our Loader to the user).

In the real world, we’ll be most likely writing a component that has to get the data from a server API end-point. At the same time, we do not want to lose our ability to do quick prototyping and development of our front-end, even if the server API has not been developed yet. Now there are different ways of accomplishing this. Some people like to use mock data returned by a real API (there are packages and services out there that do just this¹⁷). Others prefer to have 2 different implementations for each API client, one that returns the mocked data (either by loading from disk or invoking a mocked API service), and one that returns the live data from the real server API. We’ll be implementing the latter pattern in this chapter so we have better control on our data and also have better control on different scenarios (TODO see our mocking of a successful or failing authentication in [chapter])

Another pattern is to create a separate API client for each area of our application. This will enable for better separation of concerns, avoid code cluttering, more easily write unit tests against each client. This is the pattern we’ll be following in this book, but remember this is not the only way to

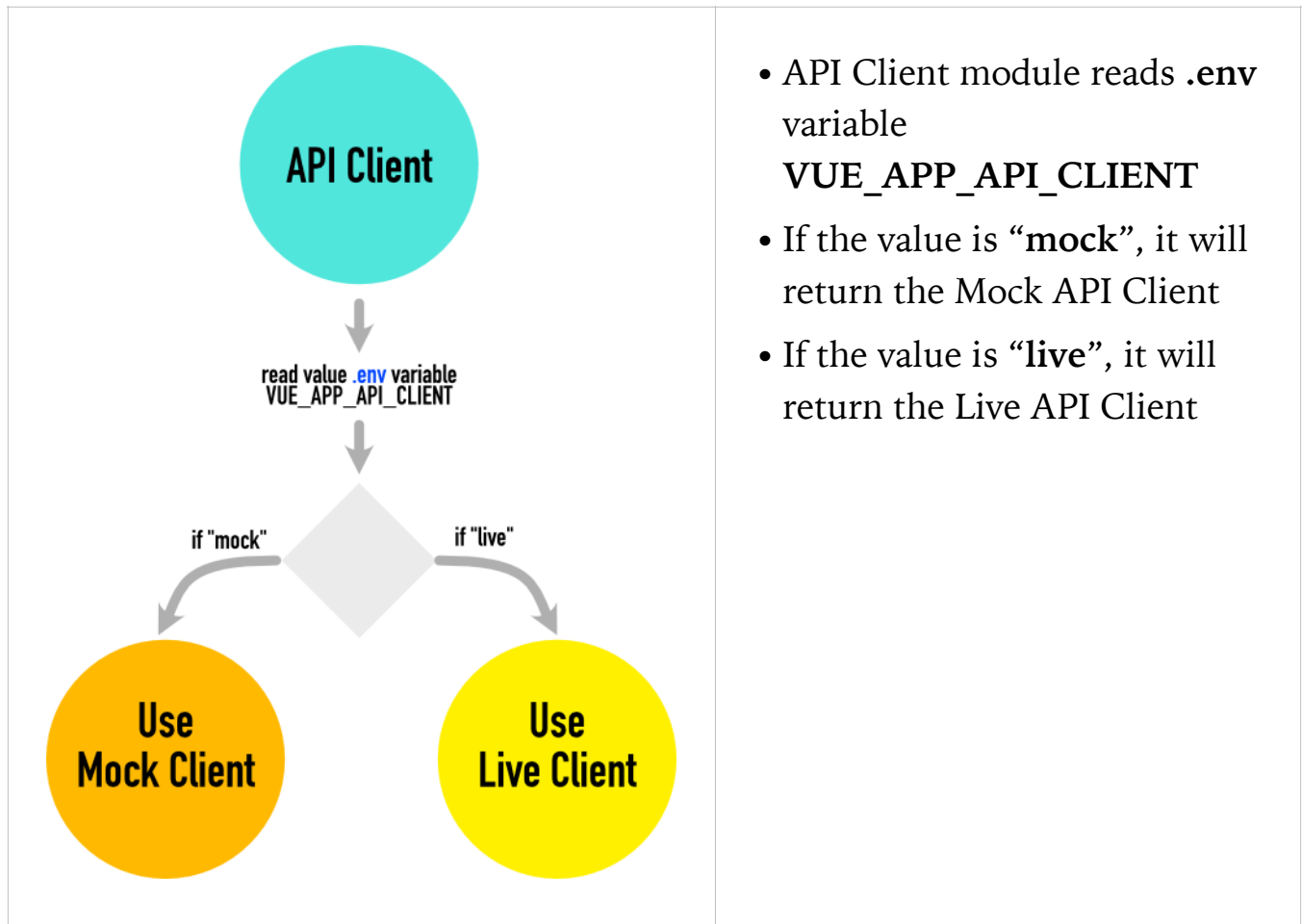
¹⁷ <https://jsonplaceholder.typicode.com/> or <https://miragejs.com/> for example

accomplish this. You should always evaluate what is the best solution for your specific requirements and evaluate that it fits your needs.

You should also read about Domain Driver Design, even though this book is not strictly following DDD principles, still the overall idea here is to try to keep code organized by application domain.

API Client Overview

Here is an overview of our API client architecture:



Domains

We'll create a global **ApiClient** that wraps additional clients organized by application domain. Our **ApiClient** will have for example a property called **items** which is the actual API client for the **Items** domain. As our application grows, we'll be adding more domains specific API clients.

Our goal is to eventually consume our API client code from our **Vuex** state in this way:

```
ApiClient
  .items
  .fetchItems()
```

Here we have an instance of our main **ApiClientInterface**. We then access its **items** property which is the domain-specific API client (of type **ItemsApiClientInterface**) and call its methods or access its properties.

Later, if for example need to add a new **people** domain, we will add a **people** property to our main **ApiClientInterface** that points to an instance of **PeopleApiClientInterface**. Then we will be able to call its methods like this:

```
ApiClient
  .people
  .fetchPeople()
```

As you can see, this makes the code much more concise and readable.

NOTE: This might seem to complicate things at first. However, remember that the scope of this book is to build a foundation for large-scale applications. Our primary goal is a solid code organization and structuring to avoid cluttering as the code might grow very large with many files.

The Main ApiClient

Create the directory **src/models/api-client**. Inside this directory, create the file **ApiClient.interface.ts** with the following code:

```
import { ItemsApiClientInterface } from './api-client/items'

/**
 * @Name ApiClientInterface
 * @description
 * Interface wraps all api client modules into one places for
 * keeping code organized.
 */
export interface ApiClientInterface {
  items: ItemsApiClientInterface
}
```

As you can see in the code above, our ApiClient will have a property called items

of type **ItemsApiClientInterface**, which will be the API client specific to the **Items** domain.

Now let's create the **Items** API client.

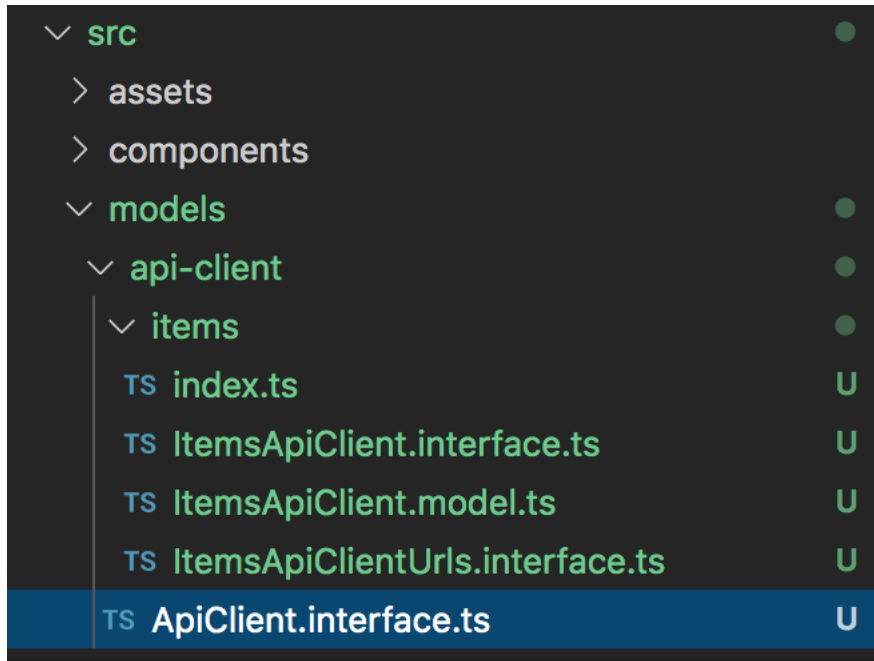
Items Api Client

Now we create the interfaces and model that defines a domain-specific API client.

Create the directory **src/models/api-client/items**. Inside the **src/models/api-client/items** directory, create the following files:

- `index.ts`
- `ItemsApiClient.interface.ts`
- `ItemsApiClient.model.ts`
- `ItemsApiClientUrls.interface.ts`

Your directory structure will look like this:



Following is the the description and code for each of the files.

ItemsApiClientUrls.interface.ts

In order to avoid using hard-coded strings, and to enforce type-checking at development time, we'll be using interface **ItemsApiClientUrlsInterface** for the Urls/paths values that indicates the API end-points consumed by the **ItemsApiClient**. Here is the code to paste into **ItemsApiClientUrls.interface.ts**:

```

/**
 * @Name ItemsApiClientUrlsInterface
 * @description
 * Interface for the Items urls used to avoid hard-coded strings
 */
export interface ItemsApiClientUrlsInterface {
  fetchItems: string
}

```

ItemsApiClient.interface.ts

This is the interface for our **ItemsApiClient**. Our interface requires implementing a method called **fetchItems** the will return a list of items. Here is the code to paste into **ItemsApiClient.interface.ts**:

```

import { ItemInterface } from '@models/items/Item.interface'

/**
 * @Name ItemsApiClientInterface
 * @description
 * Interface for the Items api client module
 */
export interface ItemsApiClientInterface {
  fetchItems: () => Promise<ItemInterface[]>
}

```

ItemsApiClient.model.ts

This is the model (class) for our **ItemsApiClient** which implements our Items

API client interface.

For the initial version of this, we will be using a third-part open-source NPM package called **axios**. This is just a library that allows to make Ajax call in a much easier way. Let's go back to the terminal, from within **my-vue3-project** directory, and install **axios** with the command:

```
npm install axios --save
```

NOTE: we will improve this even more later (see [TODO chapter]) to avoid having references to a third-party NPM package spread throughout the code.

Back to the editor, open **ItemsApiClient.model.ts** and start importing all the things we need:

```
import axios, { AxiosRequestConfig, AxiosError, AxiosResponse }  
from 'axios'
```

```
import { ItemsApiClientUrlsInterface } from './  
ItemsApiClientUrls.interface'
```

```
import { ItemsApiClientInterface } from './  
ItemsApiClient.interface'
```

```
import { ItemInterface } from '@models/items/Item.interface'
```

```
...
```

And here is the class that implement our **ItemsApiClientInterface**:

```

/**
 * @Name ItemsApiClientModel
 * @description
 * Implements the ItemsApiClientInterface interface
 */
export class ItemsApiClientModel implements
ItemsApiClientInterface {
    private readonly urls!: ItemsApiClientUrlsInterface

    constructor(urls: ItemsApiClientUrlsInterface) {
        this.urls = urls
    }

    fetchItems(): Promise<ItemInterface[]> {
        return new Promise<ItemInterface[]>((resolve) => {
            const url = this.urls.fetchItems

            // axios options
            const options: AxiosRequestConfig = {
                headers: {
                }
            }

            axios
                .get(url, options)
                .then((response: AxiosResponse) => {
                    resolve(response.data as ItemInterface[])
                })
                .catch((error: any) => {
                    console.error('ItemsApiClient: HttpClient: Get:
error', error)
                })
            })
        })
    }
}

```


index.ts

This just exports all our interfaces and models under items/ so that we can more easily import them later in other parts of the code:

```
export * from './ItemsApiClientUrls.interface'  
export * from './ItemsApiClient.interface'  
export * from './ItemsApiClient.model'
```

Mock and Live Api Clients

Now that we have defined our models for **ApiClientInterface** and **ItemsApiClientInterface**, let's implement a mechanism that will allow us to either use a **mock** api-client that returns static **json** data, or a **live** api-client that returns data from as real API.

Under the **src** directory, create a sub-directory called **api-client**. Within this directory create two new sub-directories called:

- **mock** (this will contain our mock implementations to return static **json** data)
- **live** (this will contain the implementation that call the real API end-points)

We'll be writing two implementations of our **ApiClientInterface** and its child **ItemsApiClientInterface**. One for the mock and one for the live API.

Mock Api Client

Let's start with the mock first. Within the **mock** directory, add a child directory called **items**, and within that one create a new file named **index.ts**. Your directory structure should look like this:



Inside the `src/api-client/mock/items/index.ts` file, paste the following code:

```
import {
  ItemsApiClientUrlsInterface,
  ItemsApiClientInterface,
  ItemsApiClientModel
} from '@models/api-client/items'

const urls: ItemsApiClientUrlsInterface = {
  fetchItems: '/static/data/items.json'
}

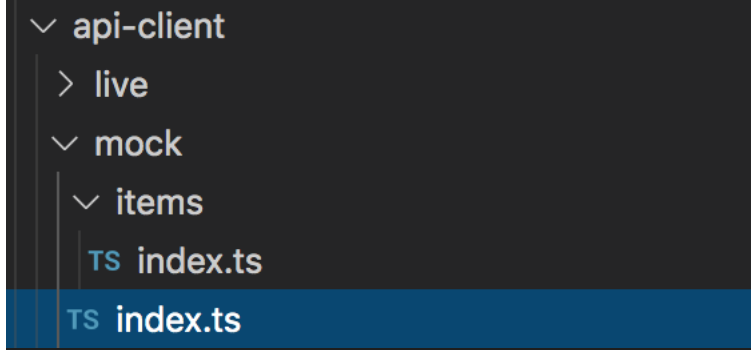
// instantiate the ItemsApiClient pointing at the url that
// returns static json mock data
const itemsApiClient: ItemsApiClientInterface = new
ItemsApiClientModel(urls)
// export our instance
export default itemsApiClient
```

Here we import all our interfaces and models, then we instantiate a variable called **urls** of type **ItemsApiClientUrlsInterface** that holds the path to the API end-point that returns the **Items** data. In this case, since this is the mock implementation, we will point to some static **json** file with the mock data. Note that we have only **fetchItems**, but we could have multiple end-points. For now we'll focus only on returning data. Later, in more advanced chapter I'll show you how to do something similar for CRUD operations.

We then create an instance of our **ItemsApiClient** class by passing our **urls** instance into the constructor (as you can see, later in our live implementation we'll pass an instance of **ItemsApiClientUrlsInterface** that contains the paths/urls to the real end-points)

Finally, we just export our instance called **itemsApiClient**.

Now let's move one directory up, under **src/api-client/mock** and create another **index.ts** file here. Your directory structure should look like this:



```

  api-client
  ├── live
  └── mock
      └── items
          └── index.ts

```

Inside the **src/api-client/mock/index.ts** file, paste the following code:

```
import { ApiClientInterface } from '@models/api-client/  
ApiClient.interface'  
import itemsApiClient from '@api-client/mock/items'  
  
// create an instance of our main ApiClient that wraps the mock  
child clients  
const apiMockClient: ApiClientInterface = {  
  items: itemsApiClient  
}  
// export our instance  
export default apiMockClient
```

This is the mock implementation of our main `ApiClient` that wraps that items client.

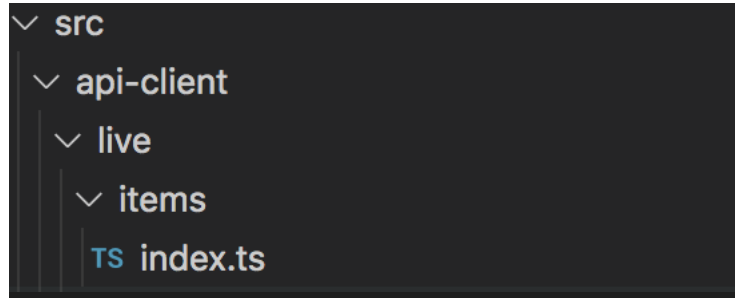
Here we import our **ApiClientInterface** interface, and our mock instance of **ItemsApiClient**. We then create an instance of our **ApiClientInterface** that is called **apiMockClient** because it will use the mock implementation of the **ItemsApiClient**.

Live Api Client

Similar to what we did with our mock api client, we'll be implementing the live api client now. Note that the **live** directory structure will be the same as the **mock** directory structure.

*NOTE: Here we'll be following similar steps as per the mock one. Alternatively, you can copy all the files under **api-client/mock** to **api-client/live** and make the necessary changes.*

Create directory **src/api-client/live/items** and here add a new file named **index.ts**. Your directory structure should look like this:



```
✓ src
  ✓ api-client
    ✓ live
      ✓ items
        TS index.ts
```

Inside the **src/api-client/live/items/index.ts** file, paste the following code:

```

import {
  ItemsApiClientUrlsInterface,
  ItemsApiClientInterface,
  ItemsApiClientModel
} from '@models/api-client/items'

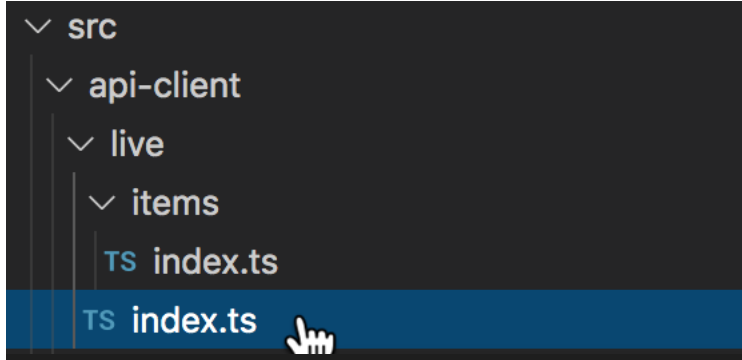
const urls: ItemsApiClientUrlsInterface = {
  fetchItems: '/static/data/items.json'
}

// instantiate the ItemsApiClient pointing at the url that
// returns static json mock data
const itemsApiClient: ItemsApiClientInterface = new
ItemsApiClientModel(urls)
// export our instance
export default itemsApiClient

```

*NOTE: this code is almost exactly the same as the mock client. The only difference is the **fetchItems** property that here says for now “/path/to/your/real/api/and-point”. You’ll replace this with the actual value of your real server API end-point url/path. If you do not have one yet, leave the current value as a place holder and updated once in the future you’ll have your server API ready.*

Now let’s move one directory up, under **src/api-client/live** and create another **index.ts** file here. Your directory structure should look like this:



Inside the `src/api-client/live/index.ts` file, paste the following code:

```
import { ApiClientInterface } from '@models/api-client/ApiClient.interface'
import itemsApiClient from '@api-client/live/items'

// create an instance of our main ApiClient that wraps the live
// child clients
const apiMockClient: ApiClientInterface = {
  items: itemsApiClient
}
// export our instance
export default apiMockClient
```

This code is also almost identical to the related mock `index.ts` file. The only exceptions are:

1. We use the live `ItemsApiClient` from `api-client/live-items`

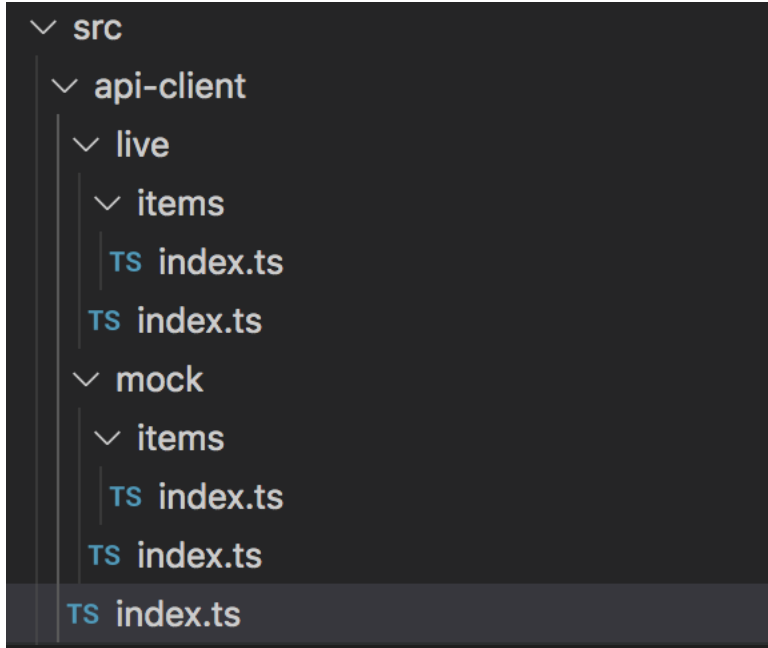
2. We name the instance **apiLiveClient** for more clarity

We then just export our **apiLiveClient** instance.

Now we need one final **index.ts** that will server our main API client factory and return either the **mock** or the **live** API client based on an environment variable (later you might find easier to drive this with different configuration files).

Api Client Factory

Let's move up one more directory. Under **src/api-client** let's add another **index.ts** file. Your directory structure should look like this:



Inside the **src/api-client/index.ts** file, start by importing a reference to our **ApiClientInterface** interface, and both the instances for the **mock** and the **live** clients:

```
import { ApiClientInterface } from '@models/api-client/
ApiClient.interface'
import apiMockClient from '@api-client/mock'
import apiLiveClient from '@api-client/live'
```

Now we will add some code that will export either the **mock** or **live** clients based on an environment variable. The complete code will look like this:

```
import { ApiClientInterface } from '@models/api-client/ApiClient.interface'
import apiMockClient from '@api-client/mock'
import apiLiveClient from '@api-client/live'

let env: string = 'mock'
if (process.env && process.env.VUE_APP_API_CLIENT) {
  env = process.env.VUE_APP_API_CLIENT.trim()
}
// return either the live or the mock client
let apiClient: ApiClientInterface
if (env === 'live') {
  apiClient = apiLiveClient
} else {
  apiClient = apiMockClient
}

export default apiClient
```

Depending on your TypeScript and **ESlint** configurations, you might have to add also a declaration for the `process.env` types within the `src/shims-vue.d.ts` file:

```
declare interface process {  
  env: {  
    VUE_APP_API_CLIENT: string  
  }  
}
```

Finally, we have to create an `.env` file that will store our default environment variables. Create this file under the root of your project directory and add the following content:

```
VUE_APP_API_CLIENT=mock
```

We also need to also update the `scripts` section within the `package.json` file so it will correctly set the expected environment variables when running locally for development with `npm run build`, or when building for production with `npm run build`. The current content of your script section should be like this:

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

Change the **serve** command to:

```
serve": "cross-env VUE_APP_API_CLIENT=mock vue-cli-service serve  
--mode development",
```

Change the **build** command to:

```
"build": "cross-env VUE_APP_API_CLIENT=live vue-cli-service build  
--mode production",
```

Optional: You could also add a **build-mock** command that uses the mock api client, if you are do not plan to have a real API in your project, or maybe to test new front-end functionality in production when the server API is not yet ready:

```
"build-mock": "cross-env VUE_APP_API_CLIENT=mock vue-cli-service  
build --mode production",
```

Similarly, you could also have a **serve-mock** or **serve-live**, with the environment variables set as you wish.

Vuex Store Instance updates

Back into our **store/index.ts** code, we can now finally remove the reference to the hard-coded data and use our new API client to retrieve these data. Start by adding an import for our **apiClient** (note how we no longer have to worry about using the **mock** or the **live** one, the system we'll handle that automatically based on the **VUE_APP_API_CLIENT** environment variable we created earlier):

```
import { createStore } from 'vuex'
import { ItemInterface } from '@models/items/Item.interface'
import { ItemsStateInterface } from '@models/store/ItemsState.interface'
import apiClient from '@api-client'
```

Then, within the **loadItems** action, remove the hard-coded **mockItems** variable and its data. Then remove the **commit('loadedItems', mockItems)** line within the **setTimeout** and replace it with a call to our **apiClient.items.fetchItems** and this time commit **loadedItems** with the **data** returned by our **fetchItems**:

```
actions: {
  loadItems({ commit, state }) {
    commit('loadingItems')
```

```
    // mock some data
```

```
    const mockItems: ItemInterface[] = [{
      id: 1,
      name: 'Item 1',
      selected: false
    }, {
      id: 2,
      name: 'Item 3',
      selected: false
    }, {
      id: 3,
      name: 'Item 3',
      selected: false
    }]
  }
```

```
    // let's pretend we called some API end-point
    // and it takes 1 second to return the data
    // by using javascript setTimeout with 1000 for the
    milliseconds option
```

```
    setTimeout(() => {
      commit('loadedItems', mockItems)
      apiClient
        .items
        .fetchItems((data) => {
          commit('loadedItems', data)
        })
    }, 1000)
```

```
  },
```

We also need to create data folder from where our **mock** api-client will load the static **json** files.

If you remember, earlier during our Mock Api Client implementation we set the `urls fetchItems` end-point path to be `/static/data/items.json`.

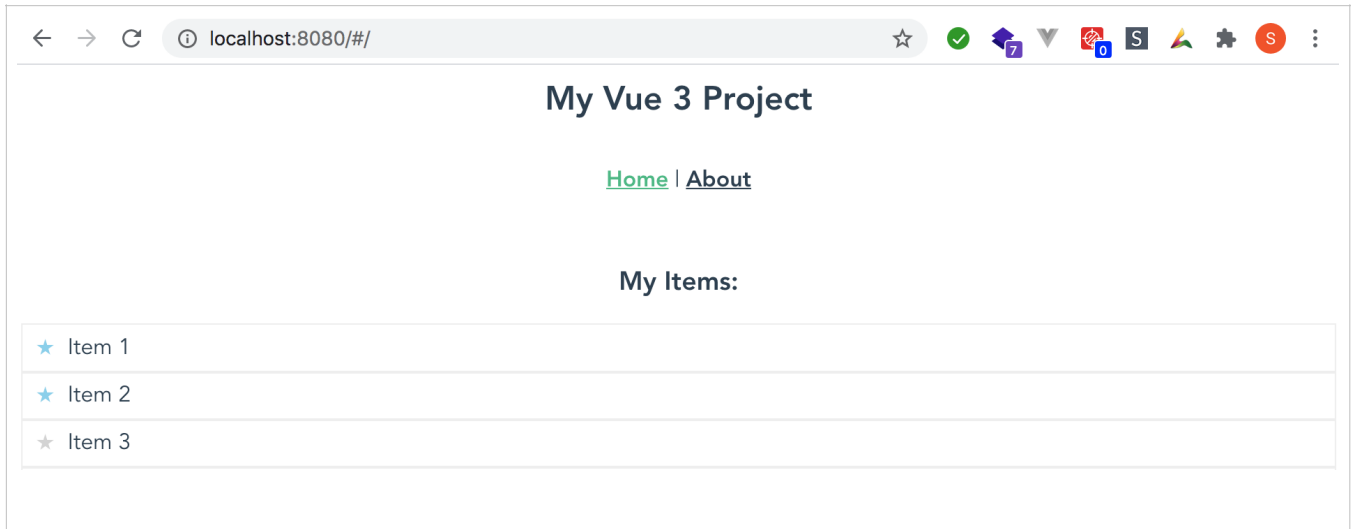
We need to create a directory called **static** under our **public** folder, because that is what **Vue** consider our root directory when running the application. And within the static directory create one called **data**. Here create a file called **items.json**.

Within the **items.json** files and paste in the following data:

```
# File: public/static/data/items.json:
```

```
[{
  "id": 1,
  "name": "Item 1",
  "selected": false
}, {
  "id": 2,
  "name": "Item 2",
  "selected": false
}, {
  "id": 3,
  "name": "Item 3",
  "selected": false
}]
```


Make sure there are no errors in the terminal. If needed stop it with **CTRL-C** and run again with **npm run serve**. The browser should display a loader, then render our items list as before:



Alternatives

There are other ways in which you could use a mocked API. There are services or libraries out there that can help you build a mocked API, and you could simplify the code here by having only one **apiClient** that uses either mock or live API end-points based on environment variables only etc.

I opted to show you how you can do this using static **.json** files that are located in the same project under **public/static/data** as this gives you a lot of flexibility to play around with different things when you are starting out. The other thing is that by having a specific implementation of the mock **apiClient** you do not have to necessarily return the .json files, but you could simulate fake responses or pretend to have saved or delete an item without actually modifying any static data (so it will be just in memory, and when you refresh the web browser the data would be reload as in its original state).

I will leave it up to you to explore the alternatives as you see fit.

Chapter 7 Recap

What We Learned

- How to implement an **apiClient** that automatically can serve either mock or real data
- How to continue enforcing type checking at development time with TypeScript interfaces and models
- How to structure directories and files in an organized way
- How to invoke our api client from the Vuex store

Observations

- We have a reference to a third NPM package (**axios**) in our **ItemsApiClient** mode and if we keep following this pattern we'll keep polluting new api client implementations for different areas with references to this NPM package in several parts of our code. This will cause a build up in technical debt that will make it harder to later replace **axios** with something else one day we'll have to. This might happen either because **axios** will no longer be supported, or maybe better NPM packages will be available that we want to use in its place. Either way, we should structure our code in a way so that we can more easily replace **axios** with something else without having to change a lot of code in too many places.
- Our **Vuex** store is still contained all in one file, **store/index.ts**, and this will quickly become cluttered as we add more properties, actions, mutations. We need a better way to structure our **Vuex** store so that can scale up in a cleaner and more maintainable way.

Based on these observations, there are a few improvements that will be making into the next two chapters:

Improvements

- Create an **HttpClient** model that implements an **HttpClientInterface** where we can encapsulate the reference to **axios** all in one place and make it easier to change later if we find the need to use a different NPM package.
- Split the **Vuex** store into modules so that code can be more easily maintained and unit tested.

Chapter 8

ENHANCE THE API CLIENT

From the previous chapter recap, we observed that the **ItemsApiClient** contains hard-coded references to the **axios** NPM package. We understand that is not a good practice to follow as, when adding more API clients, we do not want to have references to a 3rd party NPM packages spread throughout our code. What we need to do is abstract the generic client get/post/etc methods into their own implementation that we can then consume from our **ItemsApiClient** and future API clients implementations that we'll be adding later.

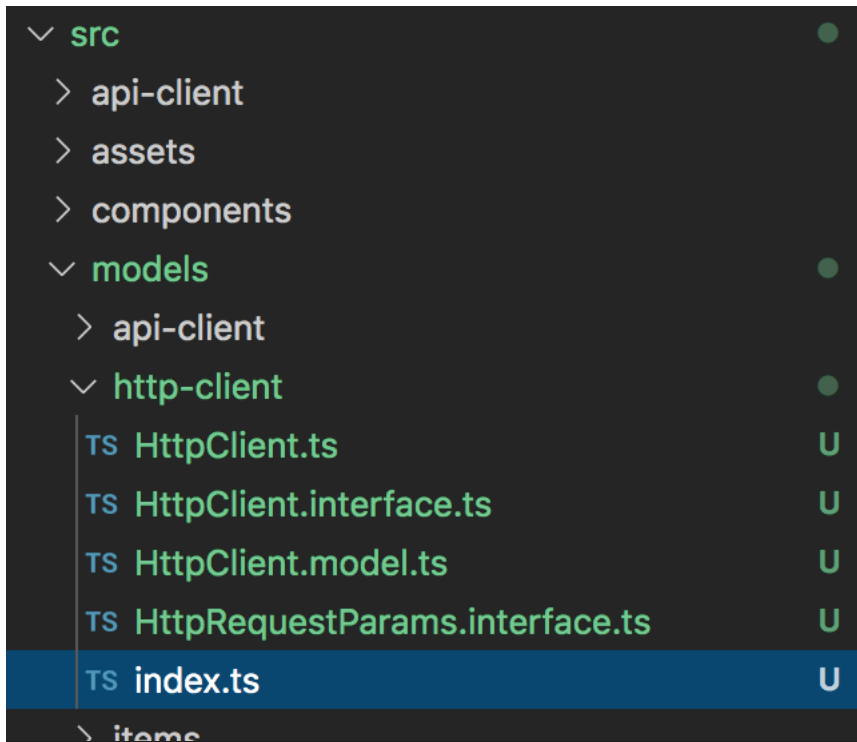
There are multiple ways we could do this. You could create an Http client factory, use dependency injection, but it should be enough to just wrap our calls done with **axios** in one place, within an **HttpClient** class. If later we have to switch to a different NPM package, all the code that needs to be updated will be inside this class and as long as we do not change the signature of our HttpClient get/post/etc methods, everything should still work as before.

HttpClient Interfaces and Models

Create the directory **src/models/http-client**. Within this directory, create the following files

- HttpClient.ts
- HttpClient.interface.ts
- HttpClient.model.ts
- HttpRequestParams.interface.ts
- index.ts

Your directory structure will look like this:



Following is the the description and code for each of the files.

HttpRequestParams.interface.ts

The `HttpRequestParamsInterface` will allow us to pass parameters to the `HttpClient` methods. These are things like the API end point **url**, an optional **payload** (if POST or PUT), and a flag that indicates if the request requires an authentication token.

```
/**
 * @name HttpRequestParamsInterface
 * @description
 * HttpClient requests parameters for get/post/put etc operations
 */
export interface HttpRequestParamsInterface {
  url: string
  requiresToken: boolean
  payload?: any
}
```

NOTE: If you want to be even more strict, you could make the payload a generic type by changing the interface declaration to `HttpRequestParamsInterface <T | any | undefined>` and the property to `payload?: T | any | undefined`

HttpClient.interface.ts

The `HttpClientInterface` is a generic interface that will define the methods out `HttpClient` will have to implement (i.e. **get/post/put/delete** etc):

```
import { HttpRequestParamsInterface } from './
HttpRequestParams.interface'

/**
 * @Name HttpClientInterface
 * @description
 * Interface for our HttpClient wrapper
 */
export interface HttpClientInterface {
  get<T>(parameters: HttpRequestParamsInterface): Promise<T>
  post<T>(parameters: HttpRequestParamsInterface): Promise<T>
}
```

HttpClient.model.ts

The **HttpClientModel** is the class that implements our **HttpClientInterface** methods (for now just **get** and **post**). Since the code here is longer, let me split in multiple parts.

First the import section:

```
import axios, {
  AxiosRequestConfig,
  AxiosError,
  AxiosResponse
} from 'axios'

import { HttpRequestParamsInterface } from './
HttpRequestParams.interface'
import { HttpClientInterface } from './HttpClient.interface'
```


First part of the class and its constructor:

```
/**
 * @name HttpClientModel
 * @description
 * Wraps http client functionality to avoid directly using a
 * third party npm package like axios
 * and simplify replacement in the future if such npm package
 * would stop being developed or other reasons
 */
export class HttpClientModel implements HttpClientInterface {
  private getToken(): string {
    const TOKEN_KEY =
      process.env && process.env.VUE_APP_TOKEN_KEY
      ? process.env.VUE_APP_TOKEN_KEY
      : 'myapp-token'
    const token = localStorage.getItem(TOKEN_KEY) || ''
    return token
  }

  constructor() {
    // OPTIONAL for now: Add request interceptor to handle errors
    // or other things for each request in one place
  }

  ...
}
```

The `get<T>` implementation:

```

...

get<T>(parameters: HttpRequestParamsInterface): Promise<T> {
    return new Promise<T>((resolve, reject) => {
        const { url, requiresToken } = parameters

        // axios options
        const options: AxiosRequestConfig = {
            headers: {}
        }

        if (requiresToken) {
            const token = this.getToken()
            options.headers.RequestVerificationToken = token
        }

        axios
            .get(url, options)
            .then((response: any) => {
                resolve(response.data as T)
            })
            .catch((response: any) => {
                console.info('----- rejecting -----')
                reject(response)
            })
    })
}
...

```

The `post<T>` implementation and end of file:

...

```
post<T>(parameters: HttpRequestParamsInterface): Promise<T> {  
  return new Promise<T>((resolve, reject) => {  
    const { url, requiresToken, payload } = parameters
```

```
    // axios options
```

```
    const options: AxiosRequestConfig = {  
      headers: {}  
    }  
  }
```

```
    if (requiresToken) {  
      const token = this.getToken()  
      options.headers.RequestVerificationToken = token  
    }  
  }
```

```
    axios  
      .post(url, payload, options)  
      .then((response: any) => {  
        resolve(response.data as T)  
      })  
      .catch((response: any) => {  
        reject(response)  
      })  
    })  
  }  
}
```

*NOTE: we added a preliminary implementation of the private **getToken** method that retrieves a token from local storage. For now this does not do anything, but when we'll get to consume a real API, we could store an authentication or JWT token into local storage during the login process with the same key as in `VUE_APP_TOKEN_KEY`. We also need to add this*

new env variables to our shims-vue.d.ts like we did for VUE_APP_API_CLIENT.

HttpClient.ts

This file contains the export of a single instance of our **HttpClientModel** as **HttpClient**. This is what we'll be consuming in our API client:

```
import { HttpClientInterface } from './HttpClient.interface'
import { HttpClientModel } from './HttpClient.model'

// export instance of HttpClientModel
export const HttpClient: HttpClientInterface = new
HttpClientModel()
```

index.ts

This just exports all our interfaces and models under **http-client/** so that we can more easily import them later in other parts of the code:

```
export * from './HttpRequestParams.interface'
export * from './HttpClient.interface'
export * from './HttpClient.model'
export * from './HttpClient'
```

HttpClient Unit Tests

Now we need to first add unit tests against our newly create **HttpClient** before we can re-factor the **ItemApiClient** code to use it in place of **axios**.

Start by installing an **NPM** package called **sinon** (and its TypeScript types) which we'll use for start using **mocks** and **stubs**¹⁸:

```
npm install --save-dev sinon @types/sinon
```

Then create the directory **src/tests/unit/http** and within this directory add a new file called **MockedPromiseFactory.ts**. Within the file, let's implement a factory that returns a mocked **promise**¹⁹ that we can use as an helper for our unit tests:

¹⁸ <https://sinonjs.org/>

¹⁹ Promises: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

```
import axios, { AxiosRequestConfig, AxiosResponse } from 'axios'

export interface MockedPromiseFactoryParamsInterface {
  url: string
  requestConfig: AxiosRequestConfig
  statusCode: number
  statusText: string
  data: any
  reject: boolean
}

export const MockedPromiseFactory = (
  params: MockedPromiseFactoryParamsInterface
): Promise<AxiosResponse<string>> => {
  return new Promise<AxiosResponse<string>>((resolve, reject) => {
    setTimeout(() => {
      const response: AxiosResponse = {
        data: params.data,
        status: params.statusCode,
        statusText: params.statusText,
        headers: [],
        config: params.requestConfig
      }

      if (params.reject) {
        reject(response)
      } else {
        resolve(response)
      }
    }, 100)
  })
}
```

This allows us to avoid code duplication in the unit tests as will mock both successful and unsuccessful responses.

Note how we resolve the promise by default, but if the **params.reject** is true we reject it. This gives us the option to simulate a **Promise** that will reject.

Testing a successful “get” response

Still within the `src/tests/unit/http` directory add a new file called `HttpClient.get.200.spec.ts`. Within the file, start by importing all the things we need:

```
import { expect } from 'chai'
import sinon from 'sinon'
import axios, { AxiosRequestConfig } from 'axios'
import { HttpClient, HttpRequestParamsInterface } from '@models/http-client'
import { MockedPromiseFactory } from '../MockedPromiseFactory'
```

Then create an instance of **HttpRequestParamsInterface**:

```
const mockParams: HttpRequestParamsInterface = {
  url: 'path/to/a/get/api/endpoint',
  requiresToken: false
}
```

Then let's start writing the core of our unit tests by adding a **before** and **after** function (these are special **mocha** hook functions that will be run before the actual unit tests²⁰). Within the **before** we will create an instance of **AxiosRequestConfig**, then use our **MockedPromiseFactory** to create a fake promise, then stub the **axios.get** call with **sinon** so that it will return our mockedPromise. Within the **after**, we just call **sinon.restore()** to clear our mocks and stubs:

²⁰ <https://mochajs.org/#hooks>


```

describe('HttpClient.get', () => {
  before(() => {
    const mockedRequestConfig = {
      headers: {}
    } as AxiosRequestConfig

    const mockedPromise = MockedPromiseFactory({
      url: mockParams.url,
      statusCode: 200,
      statusText: 'Success',
      requestConfig: mockedRequestConfig,
      data: 'get completed',
      reject: false
    })

    // since HttpClient uses axios internally, stub axios here
    sinon
      .stub(axios, 'get')
      .withArgs(mockParams.url, mockedRequestConfig)
      .returns(mockedPromise)
  })

  after(() => {
    sinon.restore()
  })

  ...

```

Now, let's write the unit test that invokes the **httpClient.get** method passing the **mockParams** and check that the response equals our expected value:

```

it('should succeed and return data', (done) => {
  HttpClient.get<string>(mockParams)
    .then((response: any) => {
      expect(response).to.equal('get completed')
      done()
    })
})

```

...

We can now add even more tests for different conditions and be able to use our **MockedPromiseFactory** to simulate the return of different **http** responses.

Testing an unsuccessful “get” response

So create a file called **HttpClient.get.400.spec.ts**. Within the file, start by importing all the things we need:

```

import { expect } from 'chai'
import sinon from 'sinon'
import axios, { AxiosRequestConfig } from 'axios'
import { HttpClient, HttpRequestParamsInterface } from '@models/http-client'
import { MockedPromiseFactory } from '../MockedPromiseFactory'

```

Then, create an instance of **HttpRequestParamsInterface**:

```
const mockParams: HttpRequestParamsInterface = {  
  url: 'path/to/a/get/api/endpoint',  
  requiresToken: false  
}
```

This time we stub a **400** response within the **before** block (note the **reject** parameter set to true):

```

describe('HttpClient.get', () => {
  before(() => {
    const mockedRequestConfig = {
      headers: {
      }
    } as AxiosRequestConfig

    const mockedPromise = MockedPromiseFactory({
      url: mockParams.url,
      statusCode: 400,
      statusText: 'Error',
      requestConfig: mockedRequestConfig,
      data: 'get completed with errors',
      reject: true
    })

    // since HttpClient uses axios internally, stub axios here
    sinon.stub(axios, 'get')
      .withArgs(mockParams.url, mockedRequestConfig)
      .returns(mockedPromise)
  })

  after(() => {
    sinon.restore()
  })

  ...

```

Again write a unit test that invokes the `httpClient.get` method passing the `mockParams` but this time also add a `catch` block as we are expecting our mocked promise to reject and return a `response.status` to be `400`:

```

it('should reject and return 400', (done) => {
  HttpClient.get<string>(mockParams)
    .then((response: any) => {
      // should not get in here
    })
    .catch((response: any) => {
      expect(response.status).toEqual(400)
      done()
    })
})

```

*NOTE: In a similar way, you could also write the code to test the post method with files **HttpClient.post.200.spec.ts** and **HttpClient.post.400.spec.ts** etc.*

We can now finally change our **ItemApiClient** so it uses our newly implemented **HttpClient** instead of **axios**:

ItemsApiClientModel Update

Open the file **src/models/api-client/items/ItemsApiClient.model.ts**.

Remove the import **axios** line and replace it with an import for our **HttpClient** instance and the **HttpRequestParamsInterface**:

```

import axios, { AxiosRequestConfig, AxiosError, AxiosResponse } from 'axios'
import { HttpClient, HttpRequestParamsInterface } from '@models/http-client'

```

Then change the **fetchItems** implementation but removing the whole blocks that returns the new **Promise**:

```
fetchItems(): Promise<ItemInterface[]> {  
    return new Promise<ItemInterface[]>(resolve => {  
        const url = this.urls.fetchItems  
  
        // axios options  
        const options: AxiosRequestConfig = {  
            headers: {}  
        }  
  
        axios  
            .get(url, options)  
            .then((response: AxiosResponse) => {  
                resolve(response.data as ItemInterface[])  
            })  
            .catch((error: any) => {  
                console.error('ItemsApiClient: HttpClient: Get: error',  
error)  
            })  
        })  
    })  
}
```

Replace it with the following code. This creates a const variable to hold our **HttpRequestParamsInterface** parameters, and then return the call to **httpClient.get** (which is already a Promise, so we do not have to do anything else here):

```
fetchItems(): Promise<ItemInterface[]> {  
    const getParameters: HttpRequestParamsInterface = {  
        url: this.urls.fetchItems,  
        requiresToken: false  
    }  
  
    return HttpClient.get<ItemInterface[]>(getParameters)  
}
```

Now, make sure there are no errors in the terminal and the browser refreshes correctly and load the data correctly.

Chapter 8 Recap

What We Learned

- How to abstract and encapsulate functionality into a **wrapper** class that ultimately uses a 3rd party package (our **HttpClient** wraps all the code consuming **axios** in one place)
- How to use **sinon** for stubs and a helper factory that returns a **Promise** to simulate different kinds of **Http** responses so we can unit tests different scenarios.

Observations

- We did not write unit tests against the **HttpClient** “post” method
- We did not write unit tests against the **ItemsApiClientModel**
-

Based on these observations, there are a few improvements that you could make, but we’ll not be doing them as part of this book:

Improvements

- Add unit tests against the **HttpClient** post method as well
- Add unit tests against the **ItemsApiClient** methods as well

Chapter 9

MODULARIZE THE VUEX STORE

Most basic examples and articles you can find on the web about **Vuex** usually show only how to create a monolithic store that contains all the app state/mutations/actions. All this code is contained in the same **store/index.ts** file. This is the same pattern we have followed so far, as we kept building on the default code stubbed initially by the **vue-cli** when we created the project.

We will now break apart our **store** into **modules** and show how we can better separate concerns and keep code that is related to one area of our application in one place. We will do this by using multiple sub-directories under the directory **src/store** and following our established naming conventions.

While doing this re-factoring, we'll also add unit testings against our store so we can validate our changes as we break the current code into modules.

NOTE: this is going to be a longer chapter and you'll be adding quite a few files and interfaces/models/namespaces. If you encounter any trouble, always refer to the GitHub code companion to this book, or feel free to reach out to me on Twitter with questions or comments.

Current Vuex Store instance (src/store/index.ts)

In **Chapter 6** we expanded on top of the default code that the **vue-cli** initially stubbed when we first created our project. We added **actions** and **mutations** to load our items and to select/unselect items.

The code looks more or less like this (I omitted the functions within actions and mutations to keep the example short here):

```
import { createStore } from 'vuex'
import { ItemInterface } from '@models/items/Item.interface'
import { ItemsStateInterface } from '@models/store/ItemsState.interface'
import apiClient from '@api-client'

// our initial state:
const state: ItemsStateInterface = {
  loading: false,
  items: []
}

export default createStore({
  state: state,
  mutations: {
    ...
  },
  actions: {
    ...
  },
  modules: {}
})
```

Currently, we have the code all in one place, and the mutations and actions here revolve around changing (or if you prefer, mutating) the **ItemsStateInterface** only.

In a large code base that will have many high-level components, this approach would lead to a gigantic **store/index.ts** file with thousands of lines of code. This will make it very hard to read, maintain, troubleshoot and unit tests against it.

The solution is to modularize our store. Thankfully, **Vuex** provides already functionality to do that. We have to only make some decisions on how to structure our files and code and follow these standard consistently as we are doing for the rest of the application code.

Store and State Models

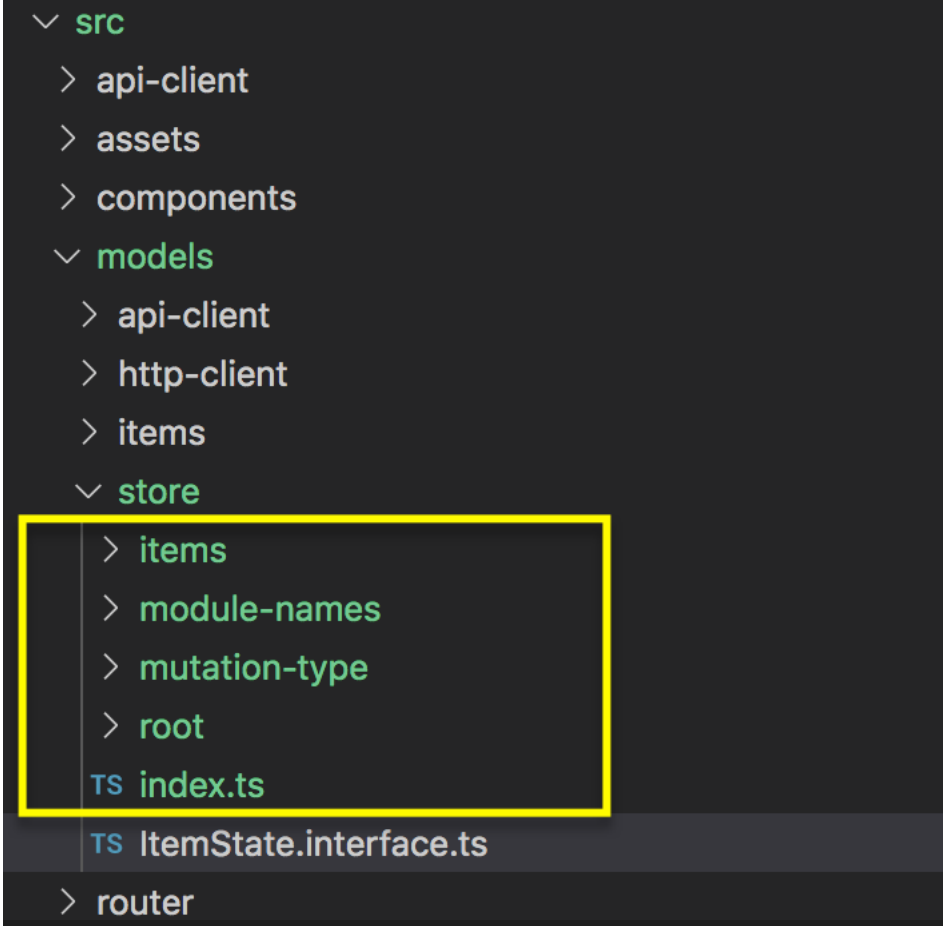
Let's start by taking care of our **models** and **interfaces** first. We'll need to create an interface for the main store (Root) and add an additional module for the Items domain. Later, in the next chapters, we'll be adding even more domain-specific modules as we grow our project larger.

Let's create sub-directories within our **src/models/store** to keep things more organized.

Add a file called **index.ts** and the following sub-directories:

- module-names
- mutation-type
- root
- items

Your **src/models/store** sub-directory structure should look like this:



```

▼ src
  > api-client
  > assets
  > components
  ▼ models
    > api-client
    > http-client
    > items
  ▼ store
    > items
    > module-names
    > mutation-type
    > root
  TS index.ts
  TS ItemState.interface.ts
  > router

```

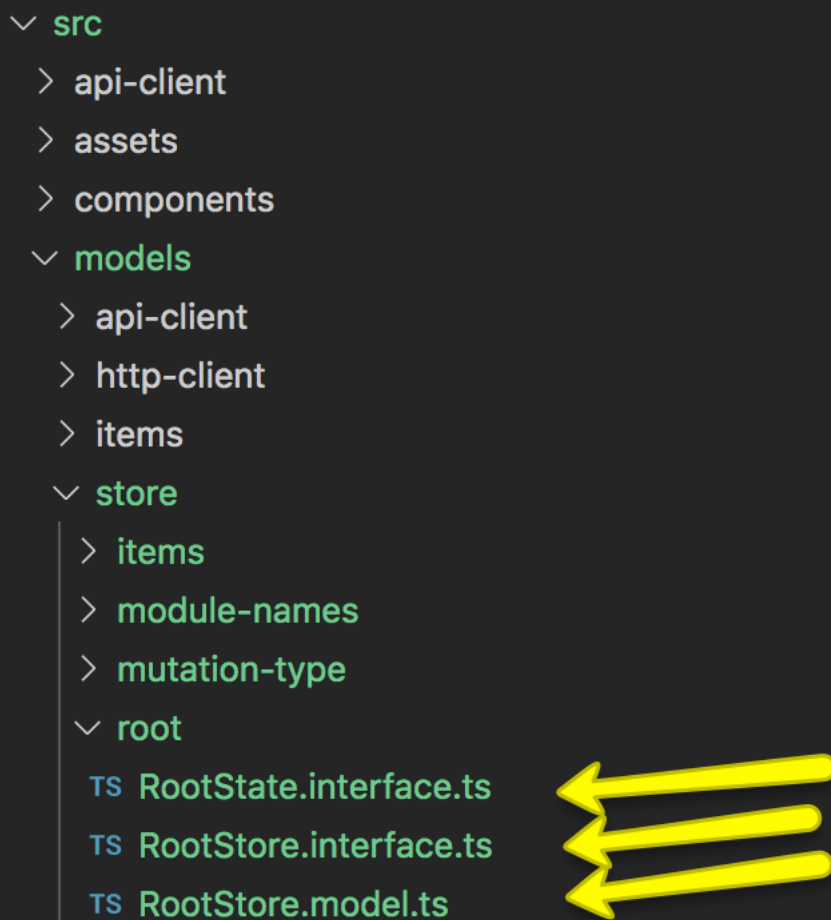
Now let's create our Root store interfaces and model.

Root Store

Inside the `src/models/store/root` directory, add these 3 files:

- `RootState.interface.ts`
- `RootStore.interface.ts`
- `RootStore.model.ts`

Your `src/models/store/root` file structure should look like this:



```

✓ src
  > api-client
  > assets
  > components
  ✓ models
    > api-client
    > http-client
    > items
  ✓ store
    > items
    > module-names
    > mutation-type
  ✓ root
    TS RootState.interface.ts
    TS RootStore.interface.ts
    TS RootStore.model.ts
```

Following, is a description and the code for each of the **root** store files.

RootState.interface.ts

The **RootStateInterface** represent the **root state** of our application. We do not have a plan to use any specific root state for now, but for the sake of providing an example complete enough, let's say that our root state will have one property called **loading** that we can set to true when the whole application is initially loading (if we want to, but we would not do it as part of this book).

```
/**
 * @name RootStateInterface
 * @description
 * Interface for the Root state
 */
export interface RootStateInterface {
  loading: boolean
}
```

RootStore.interface.ts

The **RootStoreInterface** just wraps all the other store modules in one place. This is similar to what we did with the **ApiClientInterface**. For now, we will only have one module represented by the **ItemsStateInterface**.

```
import {
  ItemsStateInterface
} from '@models/store'

/**
 * @name RootStoreInterface
 * @description
 * Wraps together each store module interface in one place
 */
export interface RootStoreInterface {
  itemsState: ItemsStateInterface
  // additional domain-specific module interfaces we'll be added
  // here in the next book chapters
}
```

RootStore.model.ts

The **RootStoreModel** extends the **Vuex.Store** and implements our **RootStoreInterface**. This model represent the root store that we will be consuming in our application.


```
import Vuex from 'vuex'
import { RootStateInterface } from './RootStore.interface'

/**
 * @name RootStateModel
 * @description
 * Extends Vuex Store declaration with our RootStateInterface
 */
export class RootStateModel<S> extends
Vuex.Store<RootStoreInterface> {
}
```

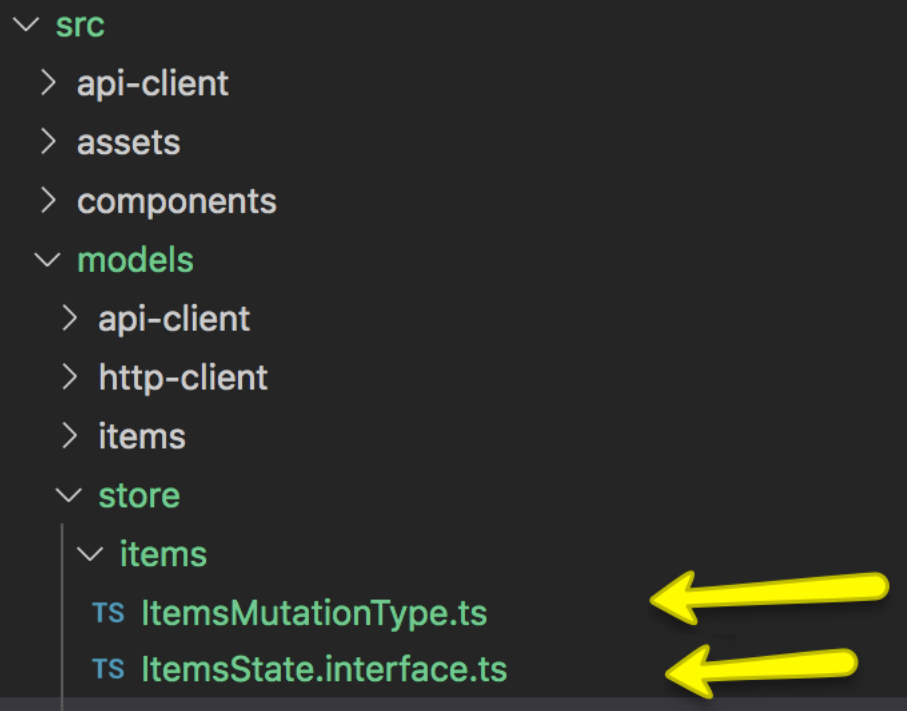
Items Store

NOTE: most of the code here is similar to the one in the first edition of this book (Vue 2), but here we reduced the number of files and code we create by leveraging TypeScript namespaces instead of using additional interfaces for constant names wrapper like `ItemsMutationType`, `MutationType`, and `StoreModuleNames`.

Inside the **src/models/store/items** directory, add these 2 files:

- `ItemsMutationType.ts`
- `ItemsState.interface.ts`

Your **src/models/store/items** file structure should look like this:



Following is a description and the code for each of the **items** store files.

ItemsMutationType.ts

The **ItemsMutationType** is a namespace that export the Items mutation type names constants. We will use these later to replace the hard-coded strings we initially used in Chapter 6. As you add more mutation type, you will be adding to this additional constants as well:

```
// group our constants in a namespace
export namespace ItemsMutationType {
  export const loadItems: string = 'loadItems'
  export const loadingItems: string = 'loadingItems'
  export const loadedItems: string = 'loadedItems'
  export const selectItem: string = 'selectItem'
  // as you add new mutations to the Items store module, keep
  adding their names here as above
}
```

ItemsState.interface.ts

We already created file in Chapter 6 when we first introduced Vuex. You can just move it from **src/models/store/ItemsState.interface.ts** to **src/models/store/items/ItemsState.interface.ts**. For completeness, here is the code for it again:

```
import { ItemInterface } from '@models/items/Item.interface'
```

```
/**
```

```
 * @name ItemsStateInterface
```

```
 * @description
```

```
 * Interface for the Items state
```

```
 */
```

```
export interface ItemsStateInterface {
```

```
  loading: boolean
```

```
  items: ItemInterface[]
```

```
}
```

Main MutationType Namespace

We now need to add the main **MutationType** namespace. This just wraps all the mutation types for each domain (i.e. **ItemsMutationType**) in one place.

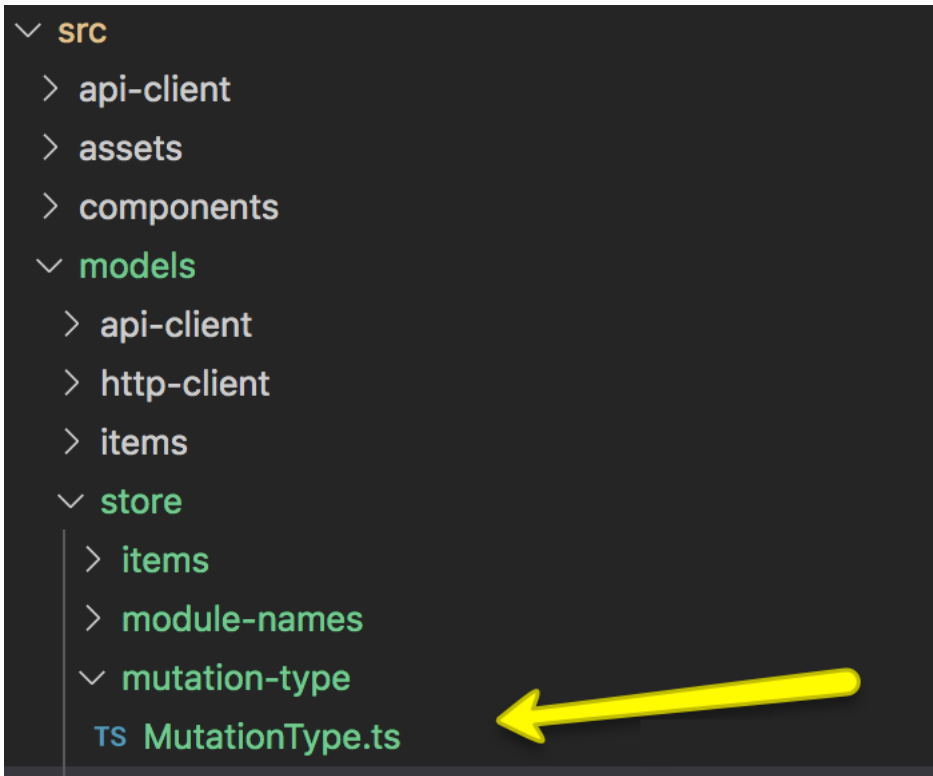
Again, for now we have only the Items domain. But this enables us access our mutation type constants in a clear and organized way. I.e.:

MutationType.items.selectItem.

Inside the **src/models/store/mutation-type** directory, add 1 file:

- MutationType.ts

Your **src/models/store/mutation-type** file structure should look like this:



Following is a description and the code for the **mutation-type/MutationType.ts** file.

MutationType.ts

The **MutationType** is a namespace the groups and exports our domain specific mutation types like **ItemsMutationType** etc in an organized way:

```
// group our constants in a namespace
import { ItemsMutationType } from '../items/ItemsMutationType'

export namespace MutationType {
  export const items = ItemsMutationType
  // as you add new state modules, add additional exports here
  following the same convention
}
```

The way we'll consume this later is to replace hard-coded strings like 'loadItems' for example:

```
// source: src/views/Home.vue
...
onMounted(() => {
  store.dispatch('loadItems')
})
...

```

With our constant value **MutationType.items.loadItems**:

```
// source: src/views/Home.vue
...
onMounted(() => {
  store.dispatch(MutationType.items.loadItems)
})
...

```

As you can see this is much more readable than using constants like **MUTATION_LOAD_ITEM** or similar as you see often in trivial example on blog posts on the web.

Module Names Namespace

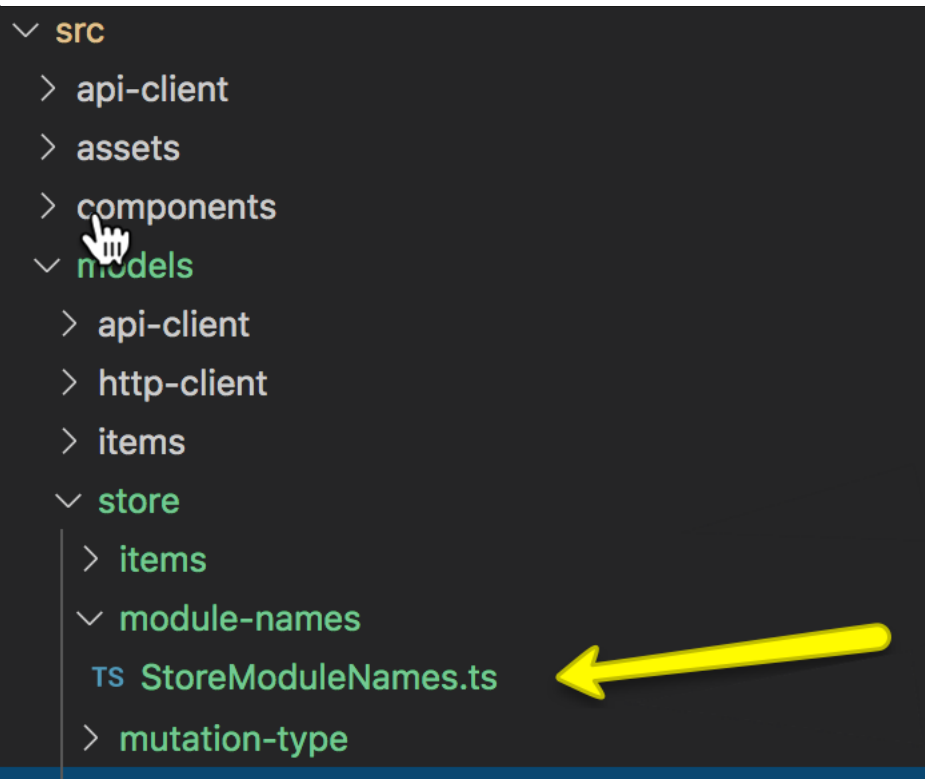
In a similar way, we'll be using constants for the **Vuex** module names. For this too we'll need a namespace that groups and exports the actual values in an organized way.

Again, the goal is to avoid hard-coded strings and as you will see in a little bit, with a modular Vuex store we'll be dispatching actions/mutations with a name-spaced syntax.

Inside the **src/models/store/module-names** directory, add 1 file:

- StoreModuleNames.ts

Your **src/models/store/module-names** file structure should look like this:



Following is a description and the code for the **module-names/StoreModuleNames.ts** file.

StoreModuleNames.ts

The **StoreModuleNames** is the namespace that groups and exports all our **Vuex** module names constants. As you add more **Vuex** store modules, you'll be adding here their names as well:

```
// group our constants in a namespace:
export namespace StoreModuleNames {
  export const itemsState: string = 'itemsState'
  // as you add new state modules, add additional properties here
  following the same convention
}
```

index.ts

Finally the **src/models/store/index.ts** file will just export all our state models and interfaces:

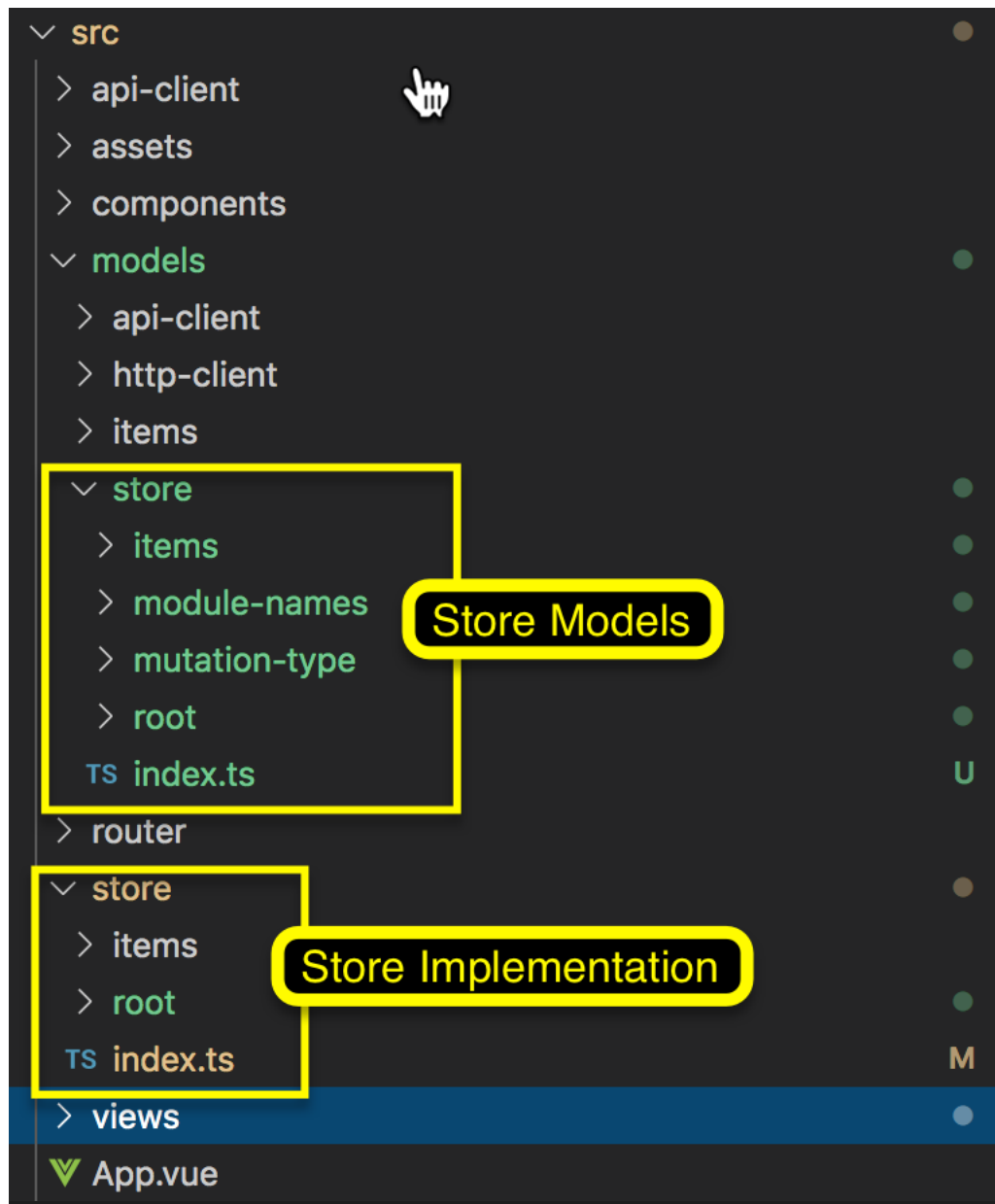
```
// export the MutationType namespace
export * from './mutation-type/MutationType'
// export the StoreModuleNames namespace
export * from './module-names/StoreModuleNames'
// export RooState and RootStore interfaces
export * from './root/RootState.interface'
export * from './root/RootStore.interface'
// export the RootStore model
export * from './root/RootStore.model'
// GEN-EXPORTS
// as you add more state modules, add additional exports for
those here as well
export * from './items/ItemsState.interface'
export * from './locales/LocalesState.interface'
```

Let's now move to the store implementation code under **src/store** directory.

Store Instance

Similarly to what we did in `src/models/store`, we need to organize our folder/file structure for the actual store implementation files under `src/store`.

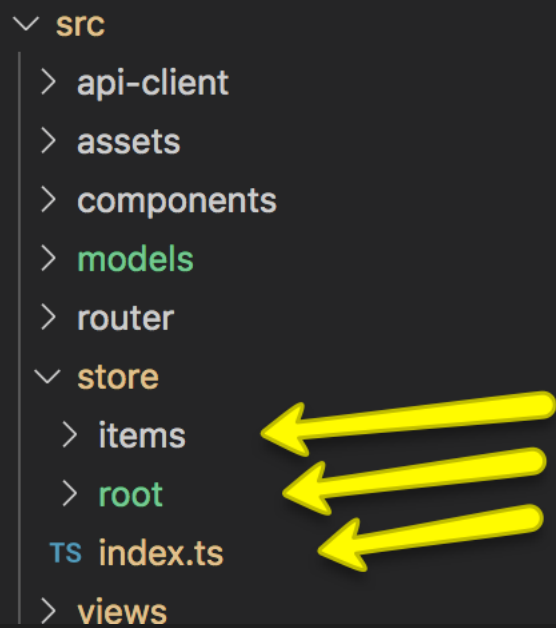
This might be confusing, so here is a screenshot to illustrate the locations for the models/interfaces and the one for the actual implementation:



Start by adding a file called **index.ts** and the following sub-directories:

- root
- items

Your **src/store** sub-directory structure should look like this:



```

✓ src
  > api-client
  > assets
  > components
  > models
  > router
  ✓ store
    > items
    > root
    TS index.ts
  > views

```

Items Store Instance

The way a **Vuex** module is structured is like how the root store is structured, except a few additional things. We'll be using the **Module** interface from **Vuex** and compose a module like this:

```
Module<ItemsStateInterface, RootStateInterface> = {  
  namespace,  
  state,  
  getters,  
  actions,  
  mutations  
}
```

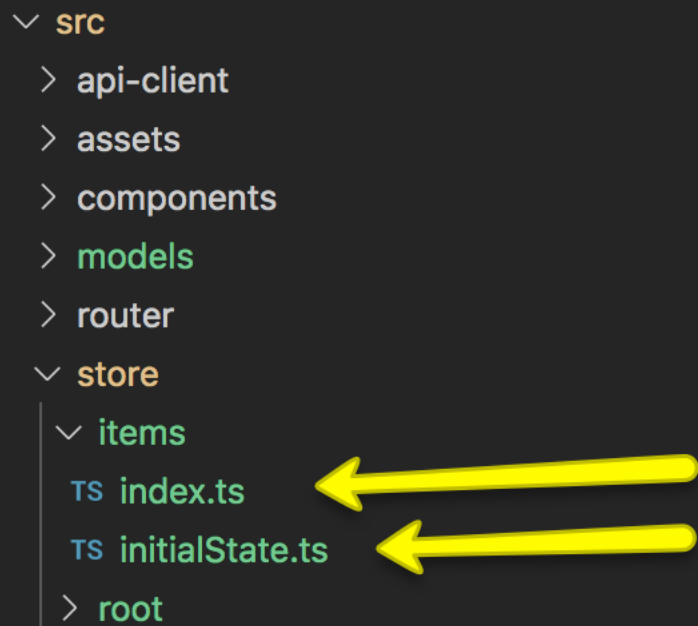
We'll be using our **TypeScript** interfaces here so we define our module with strong types by specifying both the module state interface (**ItemsStateInterface** in this case) and the interface for the root state (**RootStateInterface**):

```
Module<ItemsStateInterface, RootStateInterface>
```

Under the **src/store/items** folder, create the following 2 files:

- **initialState.ts**
- **index.ts**

Your **src/store/items** file structure should look like this:



```

  ▾ src
    > api-client
    > assets
    > components
    > models
    > router
  ▾ store
    ▾ items
      TS index.ts
      TS initialState.ts
    > root

```

Following is a description of each file and their code.

initialState.ts

Here we create and export an instance of our **ItemsStateInterface** with the initial state values:

```
import { ItemsStateInterface } from '@models/store'

/**
 * @name initialItemsState
 * @description
 * The Items state instance with the initial default values
 */
export const initialItemsState: ItemsStateInterface = {
  loading: false,
  items: []
}
```

index.ts

Here we create our **Items** module instance by first importing **Module**, **MutationTree**, **ActionTree** and **GetterTree** from **Vuex**:

```
import { Module, MutationTree, ActionTree, GetterTree } from
'vuex'
```

Then import our **MutationType**, **RootStateInterface**, **ItemsStateInterface** and our **initialItemsState**:

```
import {  
  MutationType,  
  RootStateInterface,  
  ItemsStateInterface  
} from '@models/store'
```

```
import {  
  initialItemsState  
} from './initialState'
```

Import also our **ItemInterface** and the **apiClient**:

```
import { ItemInterface } from '@models/items/Item.interface'  
import apiClient from '@api-client'
```

*Note: Some of the next code could also be extracted by the older file **src/store/index.ts** (as you implemented in Chapter 6)*

Now let's create our **mutations** as an instance of **MutationTree<ItemsStateInterface>** (note that the code for the mutations methods like *loadingItems* etc is the same as in your older *src/store/index.ts* file):


```

export const mutations: MutationTree<ItemsStateInterface> = {
  loadingItems(state: ItemsStateInterface) {
    state.loading = true
  },
  loadedItems(state: ItemsStateInterface, items: ItemInterface[])
  {
    state.items = items
    state.loading = false
  },
  selectItem(state: ItemsStateInterface, params: {
    id: number
    selected: boolean
  }) {
    const { id, selected } = params
    const item = state.items.find(o => o.id === id)
    if (item) {
      item.selected = selected
    }
  }
}

```

Next, let's create our **actions** as an instance of **ActionTree<ItemsStateInterface, RootStateInterface>** (note that the code for the actions methods like *loadItems* etc is very similar as in your older *src/store/index.ts* file. But note that we replaced the hard-coded strings like 'loadedItems' etc for the commits calls, with our *MutationType* constants):

```

export const actions: ActionTree<ItemsStateInterface,
RootStateInterface> = {
  loadItems({ commit }) {
    commit(MutationType.items.loadingItems)

    // let's pretend we called some API end-point
    // and it takes 1 second to return the data
    // by using javascript setTimeout with 1000 for the
    milliseconds option
    setTimeout(() => {
      apiClient.items.fetchItems().then((data: any) => {
        commit(MutationType.items.loadedItems, data)
      })
    }, 1000)
  },
  selectItem(
    { commit }: any,
    params: {
      id: number
      selected: boolean
    }
  ) {
    commit(MutationType.items.selectItem, params)
  }
}

```

Now, let's create our **getters** as an instance of **GetterTree<ItemsStateInterface, RootStateInterface>** (note: this for now is blank as we have not coded any getters yet, however we should include it for completeness):

```
export const getters: GetterTree<ItemsStateInterface,  
RootStateInterface> = {}
```

Let's now compose our **Module** by putting together our **mutations**, **actions**, **getters**. We will use our **initialItemsState** for the **state** and also tell **Vuex** that this module has to be name-spaced by setting the **namespaced** flag to true:

```
// create our Items store instance  
const namespaced: boolean = true  
const state: ItemsStateInterface = initialItemsState  
  
export const itemsState: Module<ItemsStateInterface,  
RootStateInterface> = {  
  namespaced,  
  state,  
  getters,  
  actions,  
  mutations  
}
```

We are using **namespaced** true because we want our module to be more **self-contained** and **reusable** as per the official [vuexjs.org](https://vuex.vuejs.org/guide/modules.html) guidance²¹:

²¹ <https://vuex.vuejs.org/guide/modules.html>

Namespacing

*By default, actions, mutations and getters inside modules are still registered under the **global namespace** - this allows multiple modules to react to the same mutation/action type.*

If you want your modules to be more self-contained or reusable, you can mark it as namespaced with `namespaced: true`. When the module is registered, all of its getters, actions and mutations will be automatically namespaced based on the path the module is registered at.

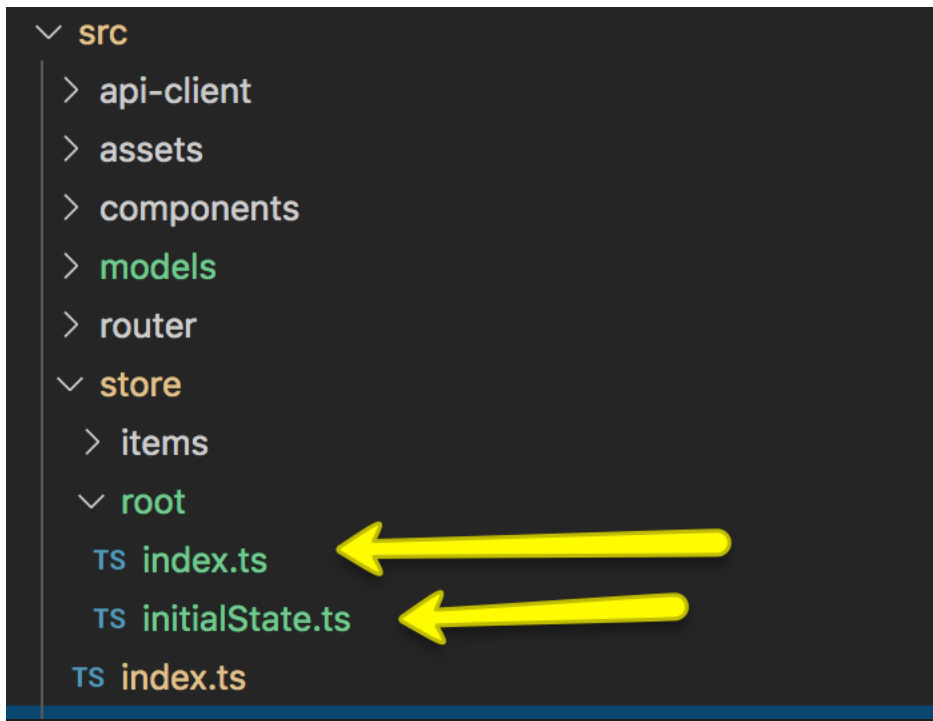
This has implications on how we'll dispatch actions to our **Vuex** module in our components because it will require us to use the 'module-name/mutation-name' syntax.

Root Store Instance

Under the `src/store/root` folder, create the following 2 files:

- `initialState.ts`
- `index.ts`

Your `src/store/root` file structure should look like this:



Following is a description of each file and their code.

`initialState.ts`

Here we create and export an instance of our **RootStateInterface** with the initial state values:

```
import { RootStateInterface } from '@models/store'

/**
 * @name initialRootState
 * @description
 * The Root state instance with the initial default values
 */
export const initialRootState: RootStateInterface = {
  loading: false
}
```

index.ts

Here we create our **Vuex** root instance which will implement our **RootStoreModel<RootStateInterface>** which put all our modules together:

```
import { createStore, StoreOptions } from 'vuex'
```

```
import {  
  RootStateInterface,  
  RootStoreInterface,  
  RootStoreModel  
} from '@models/store'
```

```
import { initialRootState } from './initialState'
```

```
// import each Vuex module
```

```
import { itemsState } from '@store/items'
```

```
// Vuex store options to build our modularized namespaced store
```

```
const storeOptions: StoreOptions<RootStateInterface> = {  
  state: initialRootState,
```

```
  modules: {  
    itemsState  
    // as you create additional modules, you'll add them here  
    // similar to the itemsState  
  }  
}
```

```
// Vuex Root store instance
```

```
export const store: RootStoreModel<RootStateInterface> =  
<any>createStore(storeOptions)
```

src/store/index.ts

Finally lets open again the older Vuex Store instance code under src/store/index.ts and replace all the code in there with one line that export our **root** instance:

```
export * from './root'
```

Let's now change our **Vue** components to leverage our store changes.

src/main.ts Updates

Note: we are no longer exporting **store** as default, so we need to import with `{ store }`:

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'
import { store } from './store'

createApp(App)
  .use(store)
  .use(router)
  .mount('#app')
```

Home.vue Updates

Let's add an import for our **MutationType**, **StoreModuleNames**, and **store** instance (Note: we are no longer exporting **store** as default, so we need to import with `{ store }`):

```
<script lang="ts">
  import { defineComponent, computed, onMounted } from 'vue'
  import store from '@store'
  import { store } from '@store'
  import { MutationType, StoreModuleNames } from '@models/store'
  ...
</script>
```

In the `setup()` method, within the computed properties, our access to `store.state.items` will no longer work. We have to change it to `store.state.itemsState.items` (similar thing for `state.loading`):

```
setup() {  
  const items = computed(() => {  
    return store.state.items  
    return store.state.itemsState.items  
  })  
  const loading = computed(() => {  
    return store.state.loading  
    return store.state.itemsState.loading  
  })  
  ...  
}
```

We need to change how we dispatch the `loadItems` action within the `onMounted` to use the **Vuex** name-space syntax ('store-modulename/mutation-name'). Here I am using JavaScript string interpolation to build the fully qualified mutation name:

```
onMounted(() => {  
  store.dispatch(MutationType.items.loadItems)  
  store.dispatch(`${StoreModuleNames.itemsState}/${  
    MutationType.items.loadItems}`)  
})
```

Similarly, we need to also change the code that dispatch the **selectItem** action:

```
const onSelectItem = (item: ItemInterface) => {  
  store.dispatch(MutationType.items.selectItem, {  
    store.dispatch(`${StoreModuleNames.itemsState}/${  
    {MutationType.items.selectItem}`, {  
      id: item.id,  
      selected: !item.selected  
    })  
}
```

Note how we have to use the namespace prefix **itemsState** with the convention ``store-modulename/mutation-name`` when dispatching our action. Also, for the name of the **action** we replaced the hard-coded string with **MutationType.items.selectingItem**.

Summary

All this might seem overkill especially to less experienced developers. It might initially create a bit of confusion when accessing your store. But in a large code base, it will help keeping things encapsulated by app domain and ultimately the benefits of better code readability, maintainability, unit testing etc will outweigh the bit of extra work that this requires.

Furthermore, it is possible to eventually auto-generate code for the API client and store modules interfaces/models and constants namespaces with tools like **plop** or **Swagger** (I will either add a Chapter here on how to do that, or write a blog post about it).

If you encounter any trouble, please refer to the companion code on GitHub, or reach me on twitter with questions or comments.

Chapter 9 Recap

What We Learned

- How to create a modularized **Vuex** store
- How to use a **MutationType** object to keep all the mutation types related to each store module in one place
- Make code more readable by avoid using hard-coded strings and enforcing constant values checks through interfaces
- How to use a code generation script to quickly stub all the files for a new store module

Observations

- We did not write unit tests against the **RootStore actions** and **mutations**

Based on these observations, there are a few improvements that you can do.

Improvements

- Add unit tests against the **RootStore actions** and **mutations**

Chapter 10

LOCALIZATION AND INTERNATIONALIZATION – LANGUAGE LOCALIZATION – part 1

“Localization refers to the adaptation of a product, application or document content to meet the language, cultural and other requirements of a specific target market (a locale)...”

“...Internationalization is the design and development of a product, application or document content that enables easy localization for target audiences that vary in culture, region, or language”²²

NOTE: This chapter applies to you only if the application you are working on will be used or marketed to multiple countries and it is desired to present labels in the local language, as well as date/number formats in the local culture.

Most modern applications that target multiple countries or cultures are architected in a way that is easy to present the UI in different languages and also present values like numbers or dates formatted as expected by the culture

²² Reference: <https://www.w3.org/International/questions/qa-i18n>

specific to that country (hence, localized).

In this book we'll first leverage a plugin that allows us to present labels in different languages (**vue-i18n**) and later we'll add also a custom plugin based on the **Intl API** (supported by most modern browsers) to provide for numbers/date formatting functionality based on different locales (cultures).

*NOTE: **vue-i18n-next** offers also formatting, but is still under work and we would rather show you here how to use the browser built-in **Intl API** instead. Later, as **vue-i18n-next** formatting capabilities are more solid, you might just use those if you prefer.*

Plugin: vue-i18n

There are many JavaScript libraries out there that simplify localization of a frontend app, but the most widely used is the **i18n** library. GitHub user **Kazuya Kawaguchi (kazupon)**²³ maintains a very nice **Vue** plugin called **vue-i18n**, which is published on NPM here <https://www.npmjs.com/package/vue-i18n>

In this book we'll be creating an additional plugin that wrap around the **vue-i18n** which will allow us to avoid code cluttering and greatly simplify how we localize our components in our **Vue** application.

You are free to call the additional plugin as you prefer if you want to avoid confusion with other existing plugins out there. In this book we'll call it **vue-i18n-next-plugin** to make it easier to remember what really is.

Let's start by first adding the **vue-i18n** NPM package to our application. We need to use the command **npm install --save vue-i18n@next** since we are using **Vue 3** (*NOTE: The **vue-i18n next** is currently at version **9.0.0-alpha.13** as of this writing, but you might get a newer version. If you want to avoid errors, you can get the exact same version with “**npm install --save vue-i18n@9.0.0-alpha.13**”*)²⁴.

We need to use the **--save** option as we want this to be saved as part of the app “dependencies” in the **package.json**:

```
npm install --save vue-i18n@next
```

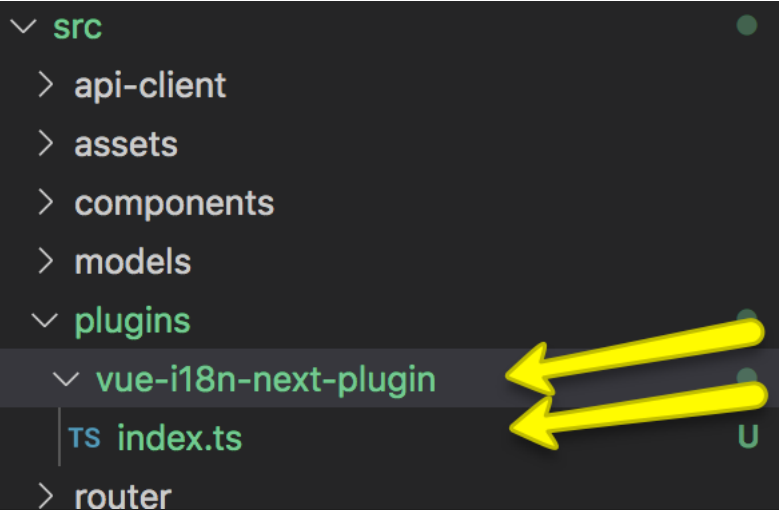
²³ <https://github.com/kazupon>

²⁴ See <https://github.com/intlify/vue-i18n-next>

Let's also install the **vue-i18n** types (these are only for TypeScript and not required to run the application, so we can use **—save-dev** to add only to the **devDependencies**):

```
npm install --save-dev @types/vue-i18n
```

Then we need to add a new folder called **plugins** within our project **src** directory. Inside the **plugins** folder we'll add another folder called **vue-i18n-next-plugin** and inside this add a TypeScript file called **index.ts**. The directory structure should look like this:



For the **index.ts** code, let's start importing **createI18n** and **LocaleMessages** from **vue-i18n**. Let's also declare an interface called **LocalesDataInterface** that defines a property called **messages** of the **vue-i18n** type **LocaleMessages**. This will simplify things later when we'll be loading the messages from static **JSON** files.

```
import { createI18n, LocaleMessages } from 'vue-i18n'

interface LocalesDataInterface {
  messages: LocaleMessages
}

...
```

For now let's instantiate a **const** called **data** with some initial data for a welcome message that we'll display in a different language for 4 different locales:

```

...
const data: LocalesDataInterface = {
  messages: {
    'en-US': {
      welcome: 'Welcome: this message is localized in English'
    },
    'it-IT': {
      welcome: 'Benvenuti: this message is localized in Italian'
    },
    'fr-FR': {
      welcome: 'Bienvenue: this message is localized in French'
    },
    'es-ES': {
      welcome: 'Bienvenido: this message is localized in Spanish'
    }
  }
}

```

Note how messages is a key-value pair lookup (or strategy pattern) that allows the **vue-i18n** plugin to know about localized text strings for specific “locales” (or cultures).

NOTE: You can learn more about locales and their standard and definitions here https://en.wikipedia.org/wiki/Language_localisation and here <https://github.com/ladjs/i18n-locale> or here <https://github.com/mashpie/i18n-node>

In our code above, we added four entries, each one related to a specific culture (or language):

- English (en-us)
- Italian (it-IT)
- French (fr-FR)
- Spanish (es-ES)

And each of them has only one key called **welcome** which holds the value for each specific locale.

Finally we export a **const** named just **i18n** for brevity which will hold a reference to a new instance of **VueI18n**. Note that the locale property is set to 'it-IT' and this will be the currently selected locale:

```
...  
  
export const i18n = createI18n({  
  locale: 'it-IT',  
  fallbackLocale: 'en-US',  
  messages: data.messages  
}) as any
```

*Note that we are only going to need one instance of our plugin and we could also use `Object.freeze` if we want to be extra careful, but since it is a **const** we would be warned by TypeScript if we accidentally try to somehow overwrite this in other parts of our code.*

Let's now switch to the main TypeScript file for our **Vue** app which is located at **src/main.ts** and import a reference to our **vue-i18n-next-plugin** and let the

Vue app instance know about it (again added code is in yellow):

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import { store } from './store'
import { i18n } from '@plugins/vue-i18n-next-plugin'

createApp(App)
  .use(store)
  .use(router)
  .use(i18n)
  .mount('#app')
```

Now if you serve the app with **npm run serve** it should build and run without errors. But, of course, the plugin is not doing anything yet.

Now, we'll soon be adding more code in the **vue-i18n-next-plugin** directory to better structure and define the different locales as this book is about writing code that can grow large and is as clean as possible, but for the time being let's go change one of our views to demonstrate how it works.

App.vue

Let's open our main **App.vue** file and within the **<script>** section let's add an import to **useI18n** from **vue-i18n**:

```

<script lang="ts">
  import { defineComponent } from 'vue'
  import { useI18n } from 'vue-i18n'
  ...

```

Within our component definition, add a `setup()` method and within this get a reference to `i18n` with `useI18n()` and return it as part of the component properties:

```

...

export default defineComponent({
  name: 'App',
  setup() {
    const i18n = useI18n()

    return {
      i18n
    }
  }
})
</script>

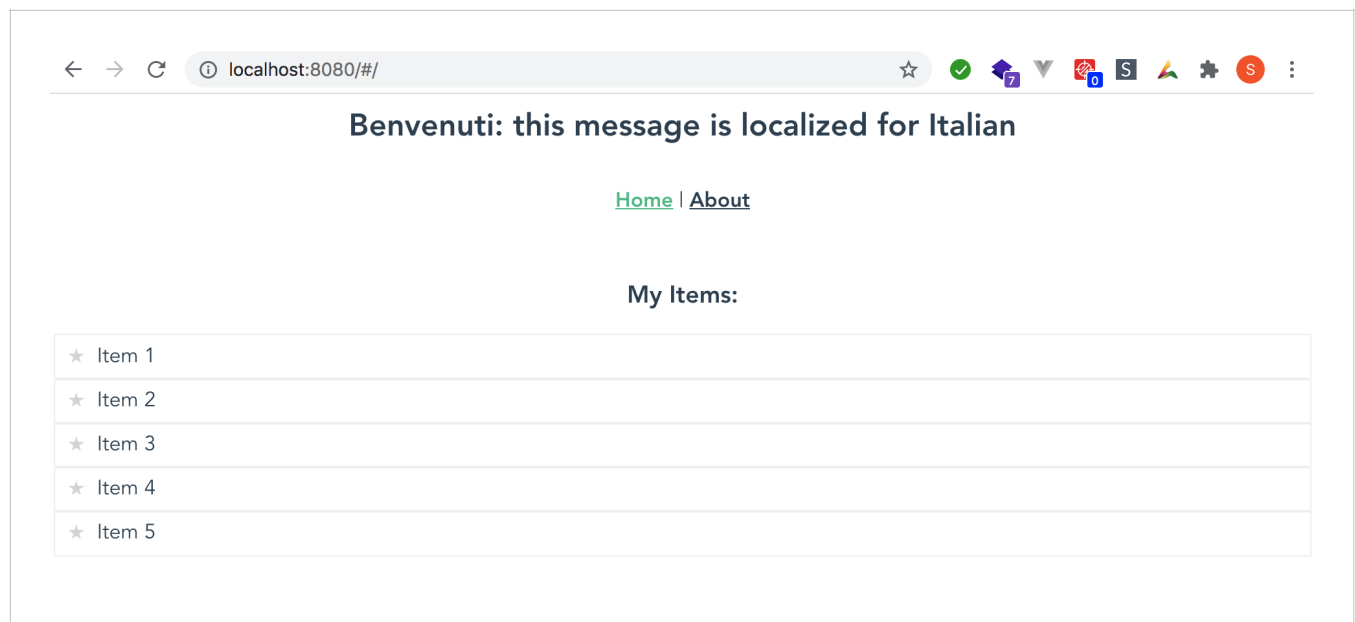
```

Within the `<template>` section, just before the navigation element there should be either a `<h1>` or `<h2>` element. Replace the hard-coded text message with the one from the `i18n` plugin using the handle-bars binding and

the `i18n.t()` function:

```
<template>
  <div id="app">
    <h2>{{ i18n.t('welcome') }}</h2>
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
    </div>
    <router-view />
  </div>
</template>
```

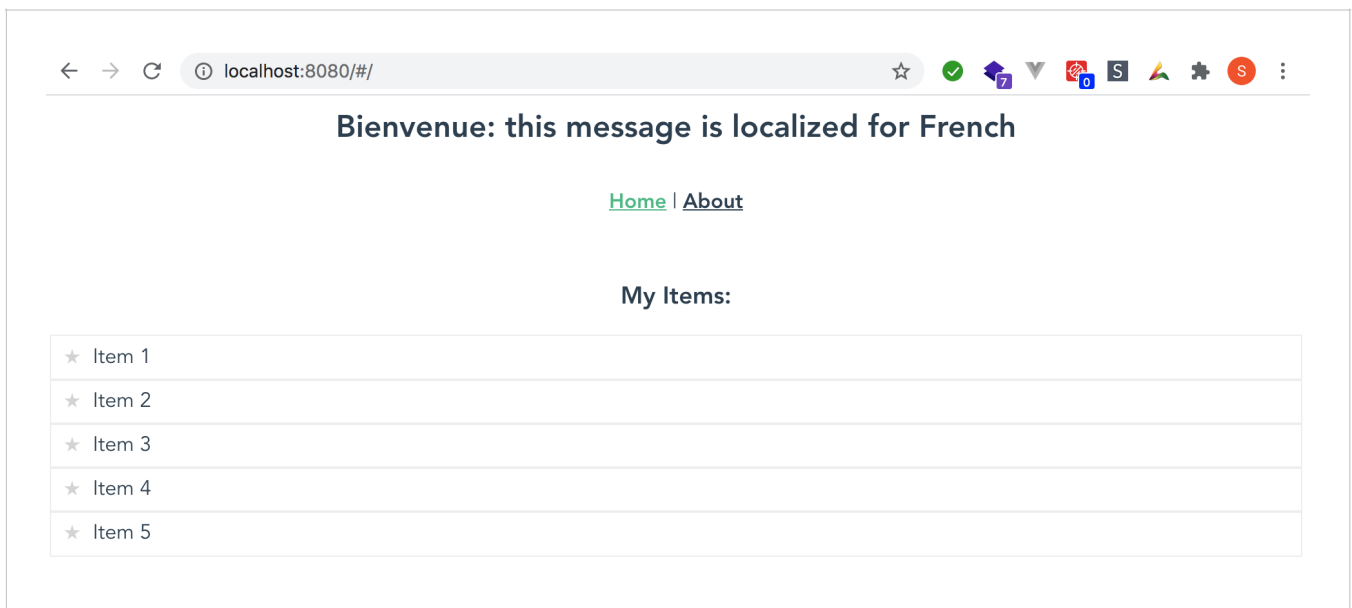
Save the file again, and make sure this time the **message** from **it-IT** is displayed
(NOTE: it default to it-IT because we specified this as the default locale within `createI18n()` in our `vue-i18n-next-plugin`):



In the next chapter we'll build a simple component that will allow us to easily switch locale from the UI so we can quickly test the different locale messages. For the time being, let's edit the **vue-i18n-next-plugin/index.ts** code once more so that it uses the **French** locale:

```
export const i18n = new VueI18n({  
  locale: 'fr-FR',  
  fallbackLocale: 'en-US',  
  messages: ...
```

Again save the file, and you should see the French version of the message in the browser:



Chapter 10 Recap

What We Learned

- How to add the **vue-i18n** plugin to our application using an additional wrapper plugin
- How to use multiple **locale** settings for text translation in order to localize our UI labels
- How to switch to test a different locale by manually (for now) updating the default locale parameter passed as one of the options to the **vue-i18n createI18n()**

Observations

- We hard-coded the localized strings for each locale
- We do not have a dynamic way to switch to a different locale at run-time

Based on these observations, there are a few improvements that we can do in the next two chapters:

Improvements

- Better structure our **vue-i18n-next-plugin** code so that it can load each locale messages from a static **JSON** file automatically
- Build a small component that allow to switch locale at run-time to visually test the different locales

Chapter 11

LOCALIZATION AND INTERNATIONALIZATION - LANGUAGE LOCALIZATION - part 2

The LocaleSelector Component

Let's start by creating an interface that define a data structure that will hold information specific to a **locale**.

Create the directory **src/models/localization** and here add a new file called **LocaleInfo.interface.ts** with this code:

```
export interface LocaleInfoInterface {  
  name: string // the friendly name of the locale, i.e. USA  
  locale: string // the locale code, i.e. en-US  
  flag: string // the 2 char code used to build the icon name  
  selected: boolean // if this locale is currently  
}
```

Vuex State Module for Locales

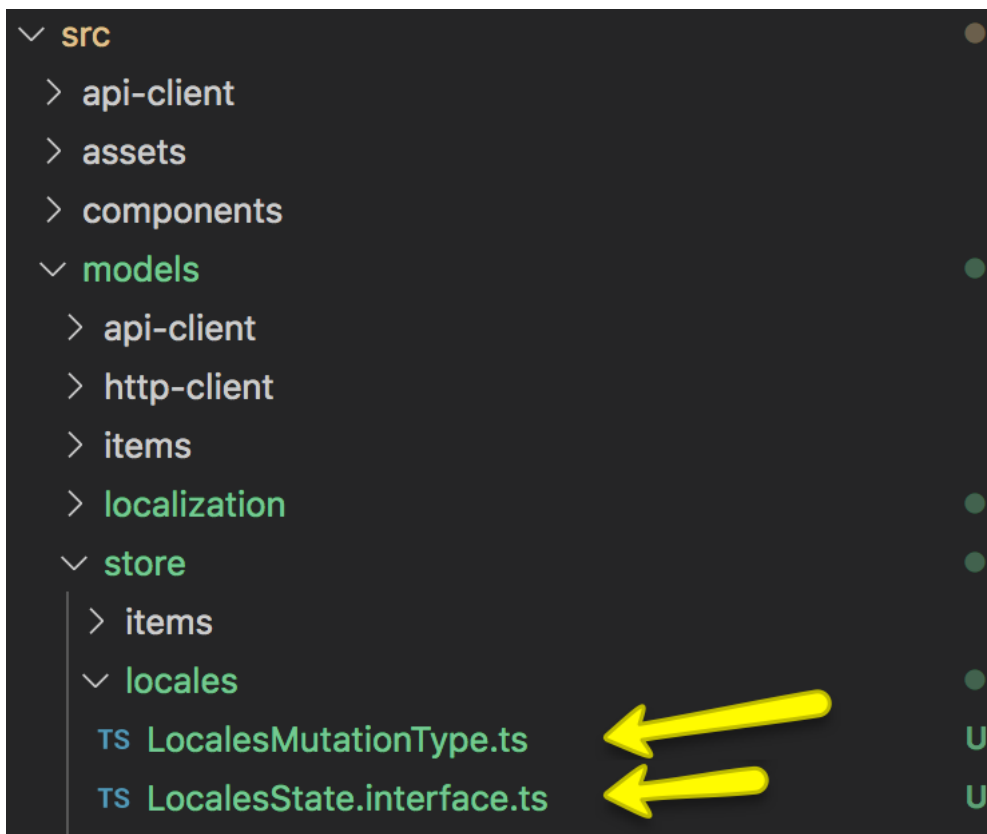
This will be a similar process to the one in Chapter 9 when we create the

ItemsStateModule and related interface/models.

Create the directory **src/models/store/locales** and here add 2 files:

- **LocalesMutationType.ts**
- **LocalesState.interface.ts**

Your **src/models/store/locales** file structure should look like this:



Following, is a description and the code for each of the **locales** store files.

LocalesMutationType.ts

The **LocalesMutationType** is a namespace that groups and exports our mutation type constants names in an organized way (this is similar to how we did the **ItemsMutationType** in Chapter 9):

```
// group our constants in a namespace
export namespace LocalesMutationType {
  export const selectLocale: string = 'selectLocale'
  // as you add more mutations to the Locales store module, keep
  adding their names here as above
}
```

LocalesState.interface.ts

The **LocalesStateInterface** represent our Locales state. From now, this will have only one property called **availableLocales** which will return an array of objects of type **LocaleInfoInterface**:

```
import { LocaleInfoInterface } from '@models/localization/
LocaleInfo.interface'

/**
 * @name LocalesStateInterface
 * @description
 * Interface for the Locales state
 */
export interface LocalesStateInterface {
  availableLocales: LocaleInfoInterface[]
}
```

src/models/store/index.ts Updates

Let's add an export at the end also for our new **LocalesState.interface**:

```
...  
// as you add more state modules, add additional exports for  
those here as well  
export * from './items/ItemsState.interface'  
export * from './locales/LocalesState.interface'
```

StoreModuleNames.ts Updates

Expand our **StoreModuleNames** namespace to include the new
LocalesMutationTypeInterface

```
// group our constants in a namespace:  
export namespace StoreModuleNames {  
  export const itemsState: string = 'itemsState'  
  export const localesState: string = 'localesState'  
  // as you add more state modules, add additional properties  
  here following the same convention  
}
```

MutationType.ts Updates

Expand our **MutationType** namespace to include the new

LocalesMutationType namespace as well:

```
// group our constants in a namespace
import { ItemsMutationType } from '../items/ItemsMutationType'
import { LocalesMutationType } from '../locales/
LocalesMutationType'

export namespace MutationType {
  export const items = ItemsMutationType
  export const locales = LocalesMutationType
  // as you add more domain-specific mutation types, add them
  here following the same convention
}
```

RootStore.interface.ts Updates

Now we need to update our **RootStoreInterface** to include the additional **LocalesStateInterface**:

```
import {
  ItemsStateInterface,
  LocalesStateInterface
} from '@models/store'

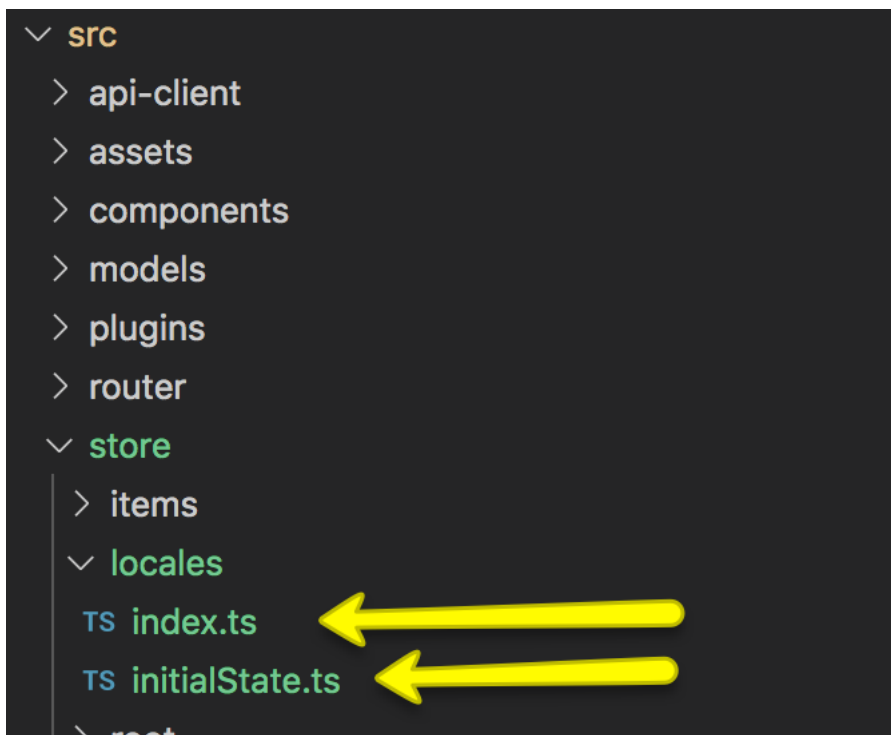
/**
 * @name RootStoreInterface
 * @description
 * Wraps together each store module interface in one place
 */
export interface RootStoreInterface {
  itemsState: ItemsStateInterface,
  localesState: LocalesStateInterface
  // additional domain-specific module interfaces we'll be added
  here in the next book chapters
}
```

LocalesStateInterface implementation

Create the directory **src/store/locales** and here add 2 files:

- initialState.ts
- index.ts

Your **src/store/locales** file structure should look like this:



Following, is a description and the code for each of the **locales** implementation files.

initialState.ts

This creates an instance of **LocalesStateInterface** with our initial data for availableLocales (note that it-IT has selected: true) and exports it with name

initialLocalesState:

```
import { LocalesStateInterface } from '@models/store'

/**
 * @name initialLocalesState
 * @description
 * The Locales state instance
 */
export const initialLocalesState: LocalesStateInterface = {
  availableLocales: [
    {
      name: 'USA',
      locale: 'en-US',
      flag: 'us',
      selected: false
    },
    {
      name: 'Italy',
      locale: 'it-IT',
      flag: 'it',
      selected: true // this is selected by default
    },
    {
      name: 'Spain',
      locale: 'es-ES',
      flag: 'es',
      selected: false
    },
    {
      name: 'France',
      locale: 'fr-FR',
      flag: 'fr',
      selected: false
    }
  ]
}
```

index.ts

This is the implementation of our **Locales Vuex** store module. We start importing all the things we need from **Vuex**, our models, our **initialLocalesState**, and our **i18n** plugin instance:

```
import { Module, MutationTree, ActionTree, GetterTree } from
'vuex'

import {
  MutationType,
  RootStateInterface,
  LocalesStateInterface
} from '@models/store'

import {
  initialLocalesState
} from './initialState'

import { LocaleInfoInterface } from '@models/localization/
LocaleInfo.interface'
import { i18n } from '@plugins/vue-i18n-next-plugin'
```

...

Then we create our **mutations** (as instance of **MutationTree<LocalesStateInterface>**), **actions** (as instance of **ActionTree<LocalesStateInterface, RootStateInterface>**), and **getters** (as

instance of `GetterTree<LocalesStateInterface, RootStateInterface>`).

Note that our `selectLocale` action will just commit the mutation `selectLocale`, which in turn will loop through all `availableLocales` and will set the `i18n.locale.value` to the currently selected locale in our state:

```
...
export const mutations: MutationTree<LocalesStateInterface> = {
  selectLocale(state: LocalesStateInterface, model:
LocaleInfoInterface) {
    // set only the model selected to true
    state.availableLocales.forEach(localeInfo => {
      localeInfo.selected = localeInfo.locale === model.locale
      if (localeInfo.selected) {
        // switch i18n selected locale
        i18n.global.locale.value = localeInfo.locale
      }
    })
  }
}

export const actions: ActionTree<LocalesStateInterface,
RootStateInterface> = {
  selectLocale({ commit }, localeInfo: LocaleInfoInterface) {
    commit(MutationType.locales.selectLocale, localeInfo)
  }
}

export const getters: GetterTree<LocalesStateInterface,
RootStateInterface> = {}
...
```

Finally, we compose our **Vuex** module (as we did in previous chapters for the **Items** module):

```
...  
  
const namespaced: boolean = true  
const state: LocalesStateInterface = initialLocalesState  
  
export const localesState: Module<LocalesStateInterface,  
RootStateInterface> = {  
  namespaced,  
  state,  
  getters,  
  actions,  
  mutations  
}
```

src/root/index.ts updates

Now we have to expand our **RootStore** implementation to include the new **LocalesState** module:

```
...  
// import each Vuex module  
import { itemsState } from '@store/items'  
import { localesState } from '@store/locales'  
  
// Vuex store options to build our modularized namespaced store  
const storeOptions: StoreOptions<RootStateInterface> = {  
  state: initialRootState,  
  
  modules: {  
    itemsState,  
    localesState  
    // as you create additional modules, you'll add them here  
    // similar to the itemsState  
  }  
}  
...
```

LocaleSelector.component.vue

Create directory `src/components/locale-selector` and here add a new file called **LocaleSelector.component.vue**

At the top of the file add the following html template:

```
<template>
  <div class="locale-selector">
    <div class="locale-radio-group">
      <LocaleFlagRadio
        v-for="(localeInfo, index) in availableLocales"
        :key="index"
        :localeInfo="localeInfo"
        @clicked="onFlagClicked"
      />
    </div>
  </div>
</template>
```

The component will display a list of custom components of type **LocaleFlagRadio** that will act as radio buttons (only one will be selected at the time). We'll create this sub-component shortly.

Add also the following script section:

```
...
<script lang="ts">
  import { defineComponent, reactive, computed, ref } from 'vue'
  import { useI18n } from 'vue-i18n'
  import { LocaleInfoInterface } from '@models/localization/
LocaleInfo.interface'
  import LocaleFlagRadio from './LocaleFlagRadio.component.vue'

  const LocaleSelectorComponent = {
    components: {
      LocaleFlagRadio
    },
    props: {
      availableLocales: {
        type: Array
      }
    },
    setup(props: any, { emit }: any) {
      const i18n = useI18n()

      const onFlagClicked = (localeInfo: LocaleInfoInterface) =>
      {
        emit('clicked', localeInfo)
      }

      return {
        onFlagClicked
      }
    }
  }

  export default defineComponent(LocaleSelectorComponent)
</script>
...
```


Finally add the style section with this code:

```
<style lang="scss">
  .locale-selector {
    display: inline-flex;

    .locale-radio-group {
      display: inline-flex;
      justify-content: center;

      label.locale-radio {
        cursor: pointer;
        display: block;
        padding: 5px;

        &.selected {
          border-bottom: solid 5px #42b983;
        }
      }
    }

    input {
      display: none;
    }
  }
</style>
```

LocaleFlagRadio.component.vue

This is a sub-component for each individual radio displayed within the

LocaleSelector component. Within the directory **src/components/locale-selector/** create a new file called **LocaleFlagRadio.component.vue** with this code for the **<template>** section:

```
<template>
  <label
    role="radio"
    :class="`locale-radio ${localeInfo.selected ? 'selected' : ''}`.trim()"
  >
    <i :class="`flag-icons ${localeInfo.flag}`"></i>
    <input
      type="radio"
      class="icon-button"
      name="locale"
      :value="localeInfo.selected"
      v-model="localeInfo.selected"
      @click="onClick"
    />
  </label>
</template>
...
```

And this code for the **<script>** section:

```

...
<script lang="ts">
  import { defineComponent, reactive, computed, ref } from 'vue'
  import { useI18n } from 'vue-i18n'
  import { LocaleInfoInterface } from '@models/localization/
LocaleInfo.interface'
  import { i18n } from '@plugins/vue-i18n-next-plugin'

  const LocaleFlagRadioComponent = {
    props: {
      localeInfo: {
        type: Object
      },
    },
    setup(props: any, { emit }: any) {
      const i18n = useI18n()

      const onClick = () => {
        emit('clicked', props.localeInfo)
      }

      return {
        onClick
      }
    }
  }

  export default defineComponent(LocaleFlagRadioComponent)
</script>

```

This custom component will show a country flag icon with the `<i class="flag-icons xx"></i>` element. The `localeInfo.flag` will output the 2-char code for

the flag.

For this project, I used the **flags-icons** created by **Yefferson Marín**²⁵. Since this contains a lot of files, I ended up using only a modified version of the SCSS file and the **svg** files for each flag.

From the companion code for this book on GitHub (<https://github.com/damianof/large-scale-apps-my-vue3-project>), grab all the files from under **src/assets/svg-flags** and copy them in your local project under the same location.

Additionally, to avoid additional complicating our build setup for now, copy all the **svg** files from **src/assets/sv-flags** folder also to your **public/svg-flags** folder.

Then copy the file **src/assets/scss/flag-icons.scss** into your local project under the same directory location.

File src/main.ts

Let's now modify the **main.ts** file so we tell **Vue** to use the **flag-icons** SCSS. Here are the changes in yellow:

²⁵ <https://github.com/yammadev/flag-icons>

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import { store } from './store'
import { i18n } from '@plugins/vue-i18n-next-plugin'
import { FlagIconsScss } from '@plugins/flags-icons/'

createApp(App)
  .use(store)
  .use(router)
  .use(i18n)
  .use(FlagIconsScss)
  .mount('#app')
```

File src/App.vue

For the **App.vue** file, we need to import the newly created **LocaleSelector** component and also our store so we can set the properties we need on the **LocaleSelector** component and also respond to the click on the locale flag button. Here are the **<template>** section changes:

```
<template>
  <div id="app">
    <h2>{{ i18n.t('welcome') }}</h2>
    <LocaleSelector
      :availableLocales="availableLocales"
      @clicked="onLocaleClicked"
    />
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
    </div>
    <router-view />
  </div>
</template>
```

...

Here are the imports addition to the `<script>` section:

```
...  
<script lang="ts">  
  import { defineComponent, computed } from 'vue'  
  import { useI18n } from 'vue-i18n'  
  import {  
    MutationType,  
    StoreModuleNames  
  } from '@models/store'  
  import { store } from '@store'  
  import { LocaleInfoInterface } from '@models/localization/  
LocaleInfo.interface'  
  import LocaleSelector from '@components/locale-selector/  
LocaleSelector.component.vue'  
  ...  
</script>
```

Here are the changes without our component definition:

```

...
export default defineComponent({
  name: 'App',
  components: {
    LocaleSelector
  },
  setup() {
    const i18n = useI18n()

    const availableLocales = computed(() => {
      return store.state.localesState.availableLocales
    })

    const onLocaleClicked = (localeInfo: LocaleInfoInterface)
=> {
      store.dispatch(
        `${StoreModuleNames.localesState}/$
{MutationType.locales.selectLocale}`,
        localeInfo
      )
    }

    return {
      i18n,
      availableLocales,
      onLocaleClicked
    }
  }
})
</script>

```

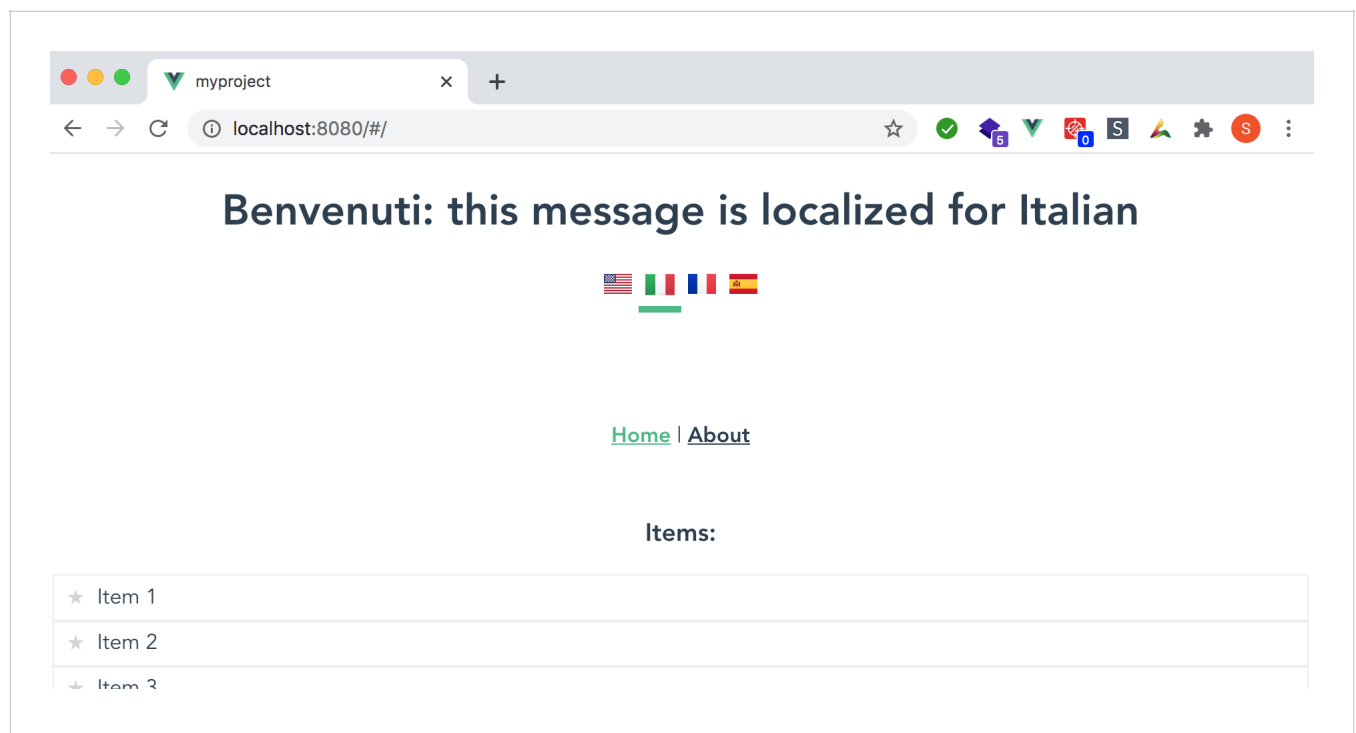
Note that you could display this component anywhere you like, but I decided to show in the main App.vue so it is always available through the whole app even when navigating to other

views.

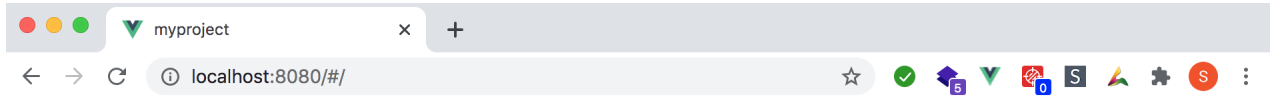
Browser

Run the app again with **npm run serve**

If everything is in the right place, now you should see the flag icons displayed at the top and be able to click on the different country flags to quickly switch between different locales:



Try clicking on the French flag for example , and it should display the welcome message at the top in French:



Bienvenue: this message is localized for French



[Home](#) | [About](#)

Items:

★ Item 1
★ Item 2
★ Item 3

Chapter 11 Recap

What We Learned

- How to add a custom component with custom radio buttons that allow selecting a different **Locale**
- How to use an external **SCSS** file with **.svg** icons
- How to setup an additional **Vuex** module to handle the selected Locale state and add ability to switch to different locales

Observations

- We did not add unit tests for our new **LocaleFlagRadio** and **LocaleSelector** components
- Our messages data of type **LocalesDataInterface** is hard-coded within the **vue-i18n-next-plugin/index.ts** and this would make it harder to maintain as we add more translated messages

Based on these observations, there are a few improvements that you could do on your own:

Improvements

- Add a unit test for the **LocaleFlagRadio** to ensure it renders the correct flag
- Add a unit test for the **LocaleSelector** component to ensure it renders a list of **LocalFlagRadio** buttons with the correct flags
- Improve how we create our instance of **LocalesDataInterface** data with the translated messages.

Chapter 12

LOCALIZATION AND INTERNATIONALIZATION – LANGUAGE LOCALIZATION – part 3

The very first thing we need to take care of, before we move to the number and date formatting chapters, is to improve how we create the **LocalesDataInterface** instance.

Currently we do this in the **vue-i18n-next-plugin/index.ts** like this:

```

...
interface LocalesDataInterface {
  messages: LocaleMessages
}

const data: LocalesDataInterface = {
  messages: {
    'en-US': {
      welcome: 'Welcome: this message is localized in English'
    },
    'it-IT': {
      welcome: 'Benvenuti: this message is localized in Italian'
    },
    'fr-FR': {
      welcome: 'Bienvenue: this message is localized in French'
    },
    'es-ES': {
      welcome: 'Bienvenido: this message is localized in Spanish'
    }
  }
}
...

```

But in a real app that has many components and a large code base, we might end up with hundreds, if not thousands of translated labels and text fragments.

We need to code this in a way that loads each locale translation from static JSON files that we can then maintain manually or through the use of third-party applications.

Here we'll be improving this by adding the static JSON files directly under a **src/**

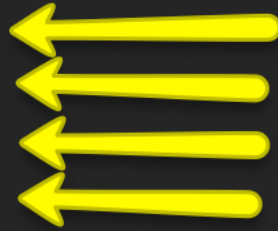
plugins/vue-i18n-next-plugin/locales/ sub-directory, with a naming convention like **[LCID].json**, i.e. **en-US.json**.

Create the directory **plugins/vue-i18n-next-plugin/locales/** and add the following 4 JSON files to it:

- en-US.json
- it-IT.json
- fr-FR.json
- es-ES.json

You **plugins/vue-i18n-next-plugin/locales/** files structure should look like this:

```
✓ src
  > api-client
  > assets
  > components
  > models
  ✓ plugins
    > flags-icons
    ✓ vue-i18n-next-plugin
      ✓ locales
        {} en-US.json
        {} es-ES.json
        {} fr-FR.json
        {} it-IT.json
TS index.ts
```



Here is the content of each file.

en-US.json:

```
{
  "messages": {
    "welcome": "Welcome: this message is localized in English"
  }
}
```

it-IT.json:

```
{
  "messages": {
    "welcome": "Benvenuti: this message is localized in Italian"
  }
}
```

fr-FR.json:

```
{
  "messages": {
    "welcome": "Bienvenue: this message is localized in French"
  }
}
```

es-ES.json:

```
{
  "messages": {
    "welcome": "Bienvenido: this message is localized in Spanish"
  }
}
```

Now we need to load these dynamically in our **LocalesDataInterface** instance called **data**. For this, let's first open the **vue-i18n-next-plugin/index.ts** file and

add an helper function just after our **LocalesDataInterface** declaration. This function will load all the JSON files that are currently under the locales sub-directory and set the messages property on our **data** accordingly:

```

import { createI18n, LocaleMessages } from 'vue-i18n'

interface LocalesDataInterface {
  messages: LocaleMessages
}

/**
 * @name: getLocalesData
 * @description: Helper to load the locale json files ...
 */
const getLocalesData = (): LocalesDataInterface => {
  // use require.context to get all the .json files ...
  const files = (require as any).context('./locales', true, /[A-
Za-z0-9-_,\s]+\\.json$/i)
  // create the instance that will hold the loaded data
  const localeData: LocalesDataInterface = {
    messages: {}
  }
  // loop through all the files
  const keys: string[] = files.keys()
  keys.forEach((key: string) => {
    // extract name without extension
    const matched = key.match(/([A-Za-z0-9-_,]+)\.\/i)
    if (matched && matched.length > 1) {
      const localeId = matched[1]
      // from each file, set the related messages property
      localeData.messages[localeId] = files(key).messages
    }
  })
  return localeData
}
...

```

Now let's remove the old code, and use our `getLocalesData` function to load the data from the JSON files:

```
...  
const data: LocalesDataInterface = {  
  messages: {  
    'en-US': {  
      welcome: 'Welcome: this message is localized in English'  
    },  
    'it-IT': {  
      welcome: 'Benvenuti: this message is localized in Italian'  
    },  
    'fr-FR': {  
      welcome: 'Bienvenue: this message is localized in French'  
    },  
    'es-ES': {  
      welcome: 'Bienvenido: this message is localized in Spanish'  
    }  
  }  
}  
  
// create our data dynamically by loading the JSON files through  
our getLocalesData helper  
const data: LocalesDataInterface = getLocalesData()  
  
...
```

The rest of the code stays the same. Save and look at the rendered page. Everything should still work as before our change.

Localize the Items Component

Now let's localize our **ItemsList** component as well. This currently has a hard-coded `<h3>` header that says "My Items:"

```
<template>
  <div>
    <h3>My Items:</h3>
    <Loader v-show="loading" />
    ...
  </div>
</template>
```

We are going to translate this text for each locale. Let's update our four JSON files like this:

en-US.json:

```
{
  "messages": {
    "welcome": "Welcome: this message is localized in English",
    "items": {
      "list": {
        "header": "My Items"
      }
    }
  }
}
```

it-IT.json:

```
{
  "messages": {
    "welcome": "Benvenuti: this message is localized in Italian",
    "items": {
      "list": {
        "header": "I miei articoli"
      }
    }
  }
}
```

fr-FR.json:

```
{
  "messages": {
    "welcome": "Bienvenue: this message is localized in French",
    "items": {
      "list": {
        "header": "Mes articles"
      }
    }
  }
}
```

es-ES.json:

```
{
  "messages": {
    "welcome": "Bienvenido: this message is localized in
Spanish",
    "items": {
      "list": {
        "header": "Mis cosas"
      }
    }
  }
}
```

ItemsList.component.vue

Now let's update our **ItemsList** component code to use the **il8n** plugin as we did for **App.vue**:

```

<template>
  <div>
    <h3>My Items:</h3>
    <h3>{{ i18n.t('items.list.header') }}:</h3>
    <Loader v-show="loading" />
  </div>
</template>

...

<script lang="ts">
  import { defineComponent, PropType } from 'vue'
  import { ItemInterface } from '@models/items/Item.interface'
  import ItemComponent from '@components/items/children/
Item.component.vue'
  import Loader from '@components/shared/Loader.component.vue'
  import { useI18n } from 'vue-i18n'

  ...

  setup(props: any, { emit }: any) {
    const i18n = useI18n()

    const onItemSelect = (item: ItemInterface) => {
      emit('selectItem', item)
    }

    return {
      i18n,
      onItemSelect
    }
  }
}

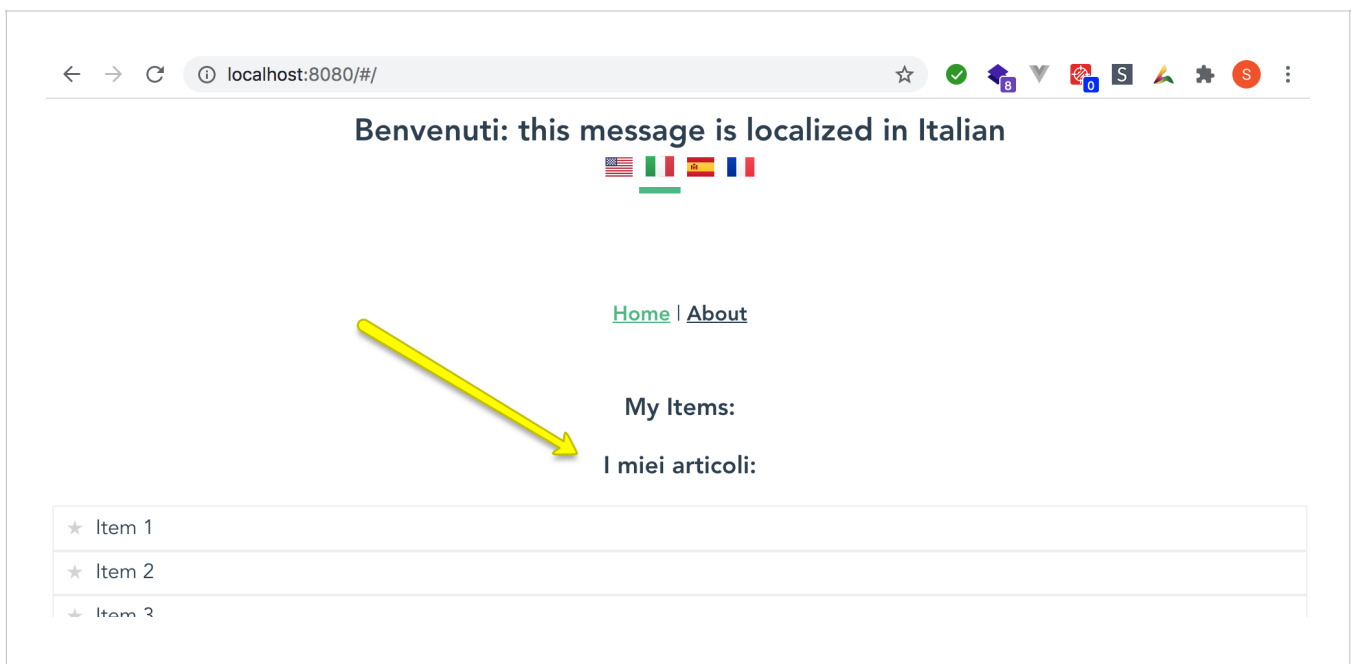
```

Not how here we have to use key **items.list.header** as that is how we structured our **JSON** data for that. We are doing this way to illustrate how to better

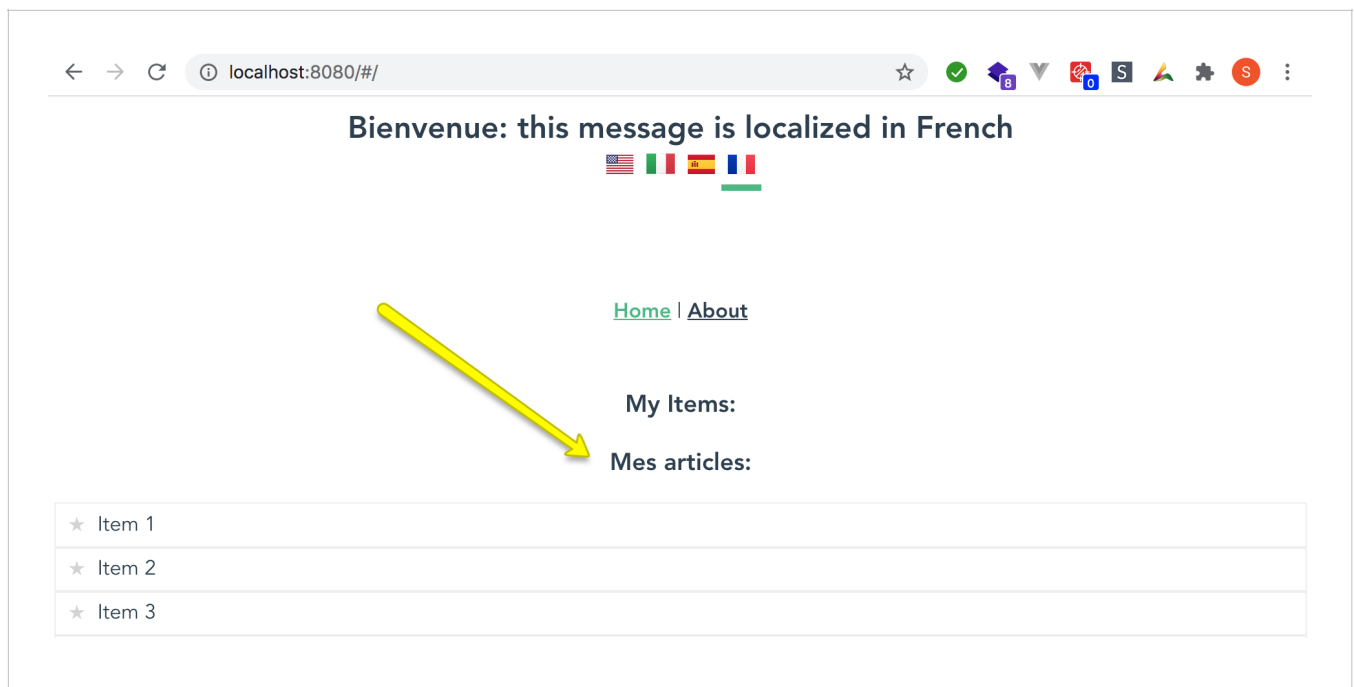
organize all the locale messages by app domain. If you prefer you can organized them in other way. Mind you, there is not going to be a perfect way that serves all scenarios, but if your code base will grow large with many components, I suggest you start using this way or some way of organizing them or your code will quickly become hard to read and maintain.

Browser

Head over your web browser and verify the page is displaying the **ItemsList** component header in the different locales. Here is in Italian:



Here is in French:



And verify also the other locales as well.

Chapter 12 Recap

What We Learned

- We learned how to localize labels for a component so that they are displayed in different languages
- We learned how to expand our locales **JSON** files in a structure and organized way

Observations

- We did not localize any number or date based on the selected **locale** yet
- We did not write any unit tests against our **getLocalesData** helper function
- We did not write any unit tests against our static **locales JSON** files to validate their structure and content

Based on these observations, there are a few improvements that we can:

Improvements

- In the next chapters we'll be adding a way to localize also numbers and dates formats
- Optional: You can write unit tests to ensure the **getLocalesData** helper function loads the **JSON** files correctly and set the messages as expected. You could also write some unit tests against the **JSON** files directly to make sure they contain the correct data structure.

LOCALIZATION AND INTERNATIONALIZATION – NUMBER AND DATETIME FORMATS

The **vue-i18n-next** also provide functionality for the formatting of dates/numbers according to the selected **locale**. While still in **alpha**, it has been recently updated to use the browser built-in **Intl API**²⁶. For this reason, it makes sense to use the **vue-i18n-next** plugin for number and date formatting as well.

If for any reason, you want more control or you need to display different formats for different locales at the same time, you can always build your own wrapper/plugin around **Intl API**.

²⁶ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl

JSON Locales Files Updates

The first this we need to do, is to add 2 additional sections to each of the **locale JSON** files.

One section, called **datetimeFormats**, defines the settings for a the **datetime** formats (**long** and **short**).

The other section, called **numberFormats**, defines the settings for the **number** formats (**currency**, **decimal**, **numeric**, **percent**).

Edit each file as follows

en-US.json:

```
{
  "datetimeFormats": {
    "long": {
      "year": "numeric",
      "month": "2-digit",
      "day": "2-digit",
      "hour": "2-digit",
      "minute": "2-digit",
      "second": "2-digit",
      "hour12": true
    },
    "short": {
      "year": "numeric",
      "month": "2-digit",
      "day": "2-digit"
    }
  },
  ...
}
```

```
...
"numberFormats": {
  "currency": {
    "style": "currency",
    "currency": "USD",
    "currencyDisplay": "symbol",
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 2
  },
  "decimal": {
    "style": "decimal",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 2
  },
  "numeric": {
    "style": "decimal",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 0
  },
  "percent": {
    "style": "percent",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 0
  }
},
"messages": {
  "welcome": "Welcome: this message is localized in English",
  "items": {
    "list": {
      "header": "My Items"
    }
  }
}
```

it-IT.json:

```
{
  "datetimeFormats": {
    "long": {
      "year": "numeric",
      "month": "2-digit",
      "day": "2-digit",
      "hour": "2-digit",
      "minute": "2-digit",
      "second": "2-digit",
      "hour12": false
    },
    "short": {
      "year": "numeric",
      "month": "2-digit",
      "day": "2-digit"
    }
  },
  ...
}
```

```
...
"numberFormats": {
  "currency": {
    "style": "currency",
    "currency": "EUR",
    "currencyDisplay": "symbol",
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 2
  },
  "decimal": {
    "style": "decimal",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 2
  },
  "numeric": {
    "style": "decimal",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 0
  },
  "percent": {
    "style": "percent",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 0
  }
},
"messages": {
  "welcome": "Benvenuti: this message is localized in Italian",
  "items": {
    "list": {
      "header": "I miei articoli"
    }
  }
}
}
```


fr-FR.json:

```
{
  "datetimeFormats": {
    "long": {
      "year": "numeric",
      "month": "2-digit",
      "day": "2-digit",
      "hour": "2-digit",
      "minute": "2-digit",
      "second": "2-digit",
      "hour12": false
    },
    "short": {
      "year": "numeric",
      "month": "2-digit",
      "day": "2-digit"
    }
  },
  ...
}
```

```
...
"numberFormats": {
  "currency": {
    "style": "currency",
    "currency": "EUR",
    "currencyDisplay": "symbol",
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 2
  },
  "decimal": {
    "style": "decimal",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 2
  },
  "numeric": {
    "style": "decimal",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 0
  },
  "percent": {
    "style": "percent",
    "useGrouping": false,
    "minimumFractionDigits": 0,
    "maximumFractionDigits": 0
  }
},
"messages": {
  "welcome": "Bienvenue: this message is localized in French",
  "items": {
    "list": {
      "header": "Mes articles"
    }
  }
}
}
```

es-ES.json:

```
{
  "datetimeFormats": {
    "long": {
      "year": "numeric",
      "month": "2-digit",
      "day": "2-digit",
      "hour": "2-digit",
      "minute": "2-digit",
      "second": "2-digit",
      "hour12": false
    },
    "short": {
      "year": "numeric",
      "month": "2-digit",
      "day": "2-digit"
    }
  },
  ...
}
```

```
...
  "numberFormats": {
    "currency": {
      "style": "currency",
      "currency": "EUR",
      "currencyDisplay": "symbol",
      "minimumFractionDigits": 0,
      "maximumFractionDigits": 2
    },
    "decimal": {
      "style": "decimal",
      "useGrouping": false,
      "minimumFractionDigits": 0,
      "maximumFractionDigits": 2
    },
    "numeric": {
      "style": "decimal",
      "useGrouping": false,
      "minimumFractionDigits": 0,
      "maximumFractionDigits": 0
    },
    "percent": {
      "style": "percent",
      "useGrouping": false,
      "minimumFractionDigits": 0,
      "maximumFractionDigits": 0
    }
  },
  "messages": {
    "welcome": "Bienvenido: this message is localized in Spanish",
    "items": {
      "list": {
        "header": "Mis cosas"
      }
    }
  }
}
```

vue-i18n-next-plugin/index.ts Updates

Now let's update our import section of the **index.ts** file like this:

```
import { createI18n } from 'vue-i18n'
```

```
interface LocalesDataInterface {  
  datetimeFormats: any  
  numberFormats: any  
  messages: any  
}
```

```
...
```

Update our **getLocalesData** function like this:

```

...
/**
 * @name: getLocalesData
 * @description: Helper to load the locale json files...
 */
const getLocalesData = (): LocalesDataInterface => {
  // use require.context to get all the json files...
  const files = (require as any).context('./locales', true, /[A-
Za-z0-9-_,\s]+\\.json$/i)
  // create the instance that will hold the loaded data
  const localeData: LocalesDataInterface = {
    datetimeFormats: {},
    numberFormats: {},
    messages: {}
  }
  // loop through all the files
  const keys: string[] = files.keys()
  keys.forEach((key: string) => {
    // extract name without extension
    const matched = key.match(/[A-Za-z0-9-_,]+\.\/i)
    if (matched && matched.length > 1) {
      const localeId = matched[1]
      // from each file, set the related messages property
      localeData.datetimeFormats[localeId] =
files(key).datetimeFormats
      localeData.numberFormats[localeId] =
files(key).numberFormats
      localeData.messages[localeId] = files(key).messages
    }
  })
  return localeData
}
...

```

Finally update the `createI18n` options like this:

...

```
// create our data dynamically by loading the JSON files...  
const data: LocalesDataInterface = getLocalesData()
```

```
// create out vue-i18n instance  
export const i18n = createI18n({  
  locale: 'it-IT',  
  fallbackLocale: 'en-US',  
  messages: data.messages,  
  datetimeFormats: data.datetimeFormats,  
  numberFormats: data.numberFormats  
})
```

App.vue

Now let's display the current date and time somewhere in our app. Open the `App.vue` file and add the following `<div>` just before the `<h2>`:

```
<template>  
  <div id="app">  
    <div class="long-date">{{ i18n.d(new Date(), 'long') }}</div>  
  </div>  
  <h2>{{ i18n.t('welcome') }}</h2>  
  ...  
</template>
```

That would be enough, but you can also use these update `<style>` section to make things look a bit better:

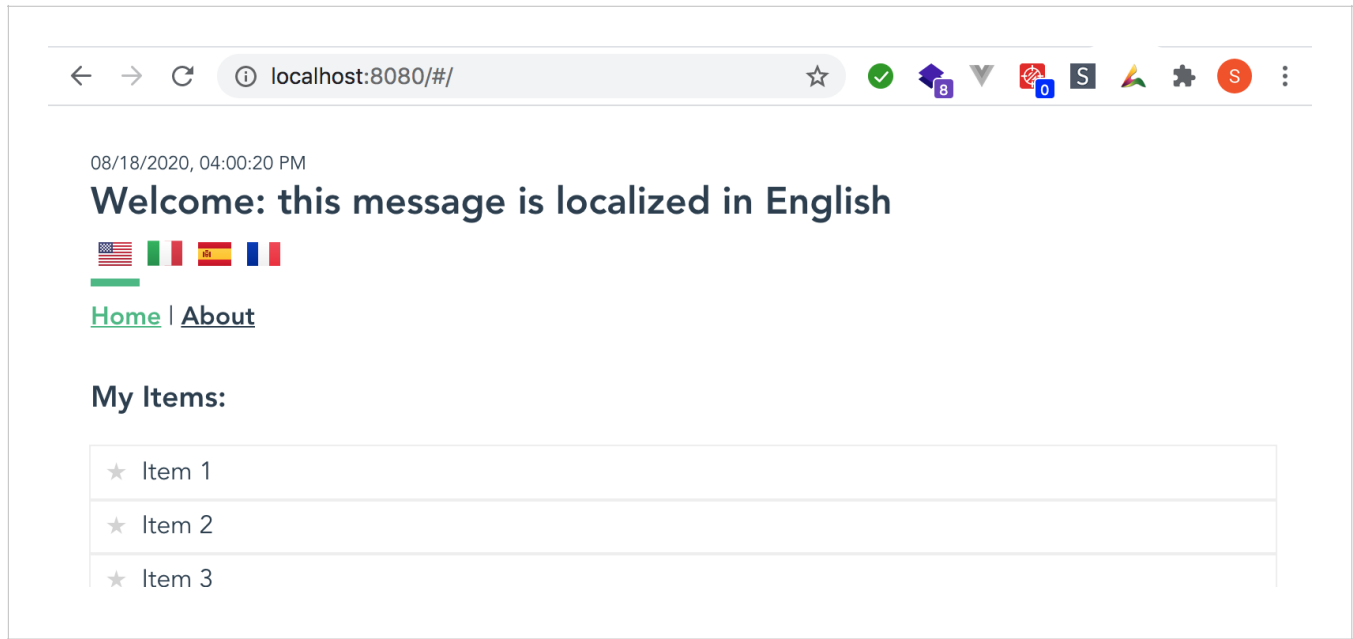
```
<style lang="scss">
  #app {
    font-family: Avenir, Helvetica, Arial, sans-serif;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    padding: 10px;
    color: #2c3e50;
    h2 {
      margin: 0;
    }

    .long-date {
      font-size: 12px;
    }
  }

  #nav {
    padding: 8px 0;
    a {
      font-weight: bold;
      color: #2c3e50;
      &.router-link-exact-active {
        color: #42b983;
      }
    }
  }
</style>
```


Web Browser

The web page should have now refreshed, and the date and time should be displaying on the top. Here is when clicking on the **US** icon:



Here is when clicking on the icon for **Italy**:

18/08/2020, 16:02:48

Benvenuti: this message is localized in Italian



[Home](#) | [About](#)

I miei articoli:

★ Item 1

★ Item 2

★ Item 3

Chapter 13 Recap

What We Learned

- We learned how to apply additional settings to our **vue-i18n-next-plugin** implementation to display **numbers** and **dates** in the **locale** formats

Observations

- We did not write any unit tests against our **getLocalesData** helper function
- We did not write any unit tests against our static **locales JSON** files to validate their structure and content

Based on these observations, there are a few improvements that we can:

Improvements

- Optional: You can write unit tests to ensure the **getLocalesData** helper function loads the **JSON** files correctly and set the messages as expected. You could also write some unit tests against the **JSON** files directly to make sure they contain the correct data structure.

Chapter 14

APP CONFIGURATION

We need now to add a way to configure our **Vue** app through configuration files for different environments (i.e. **mock**, **beta**, **live**, etc).

The code in this chapter is not just specific to **Vue**. These concepts can be applied to any front-end app.

We'll drive the configuration with JSON files. We'll have one configuration file for each environment and then load that dynamically at run-time (or build time) based on an environment variable.

File: src/shims-vue.d.ts

Let's start by modifying the **Vue shim** declaration file to extend `process.env` to include a new variable called **VUE_APP_CONFIG**.

*Note: we also remove the **VUE_APP_API_CLIENT** and **VUE_APP_TOKEN_KEY** as we'll be shortly replacing where we consume them by using our app config object.*

```
declare module '*.vue' {  
  import { defineComponent } from 'vue'  
  const component: ReturnType<typeof defineComponent>  
  export default component  
}  
  
declare interface process {  
  env: {  
    VUE_APP_API_CLIENT: string  
    VUE_APP_TOKEN_KEY: string  
    VUE_APP_CONFIG: string  
  }  
}
```

If you want to, you could also add this variable with a default value of **mock** to the **.env** file in the root of your project. But this is not required, as we'll be setting the value in our **package.json** scripts, or use a default value in the code for when a value has not been defined.

src/config

ConfigInterface and Configuration Files

Create the directory **src/config/** and under this directory create a file called **Config.interface.ts**. This contains the declaration for our **config** interface. You will keep expanding this as you add more **settings** or **app domains** (i.e. like Items), for now let's just have the interface contain four sections:

- **global**: this will be for settings that span all domains

- **httpClient**: this is for things related to the **HttpClient**
- **apiClient**: this is for things related to the **ApiClient**
- **items**: this is for the **Items** domain settings

We'll keep also the **global** section blank for now, but I wanted to add it from the beginning to show how things can be organized. As for the items section, we'll have only the **apiUrls** child section only for now. This will be of type **ItemsApiClientUrlsInterface**:

```
import {
  ItemsApiClientUrlsInterface
} from '@models/api-client/items'

/**
 * @Name ConfigInterface
 * @description
 *
 */
export interface ConfigInterface {
  global: {
    // ... things that are not specific to a single app domain
  }
  httpClient: {
    tokenKey: string
  }
  apiClient: {
    type: string
  }
  items: {
    apiUrls: ItemsApiClientUrlsInterface
  }
}
```

Now create a sub-directory called **config-files** under this directory. The full path for this will be **src/config/config-files/**

Inside this directory, add 4 JSON files:

- **mock.json**
- **local.json**
- **beta.json**
- **live.json**

The content of each file will have to match what is required by our **ConfigInterface** structure. In a little bit we'll be adding also **unit tests** against this files to make sure they are as expected.

Here is the content of each file:

mock.json

```
{
  "global": {
  },
  "httpClient": {
    "tokenKey": "myapp-token"
  },
  "apiClient": {
    "type": "mock"
  },
  "items": {
    "apiUrls": {
      "fetchItems": "/static/data/items.json"
    }
  }
}
```

local.json


```
{
  "global": {
  },
  "httpClient": {
    "tokenKey": "myapp-token"
  },
  "apiClient": {
    "type": "live"
  },
  "items": {
    "apiUrls": {
      "fetchItems": "/path/to/your/real/LOCAL/api/and-point"
    }
  }
}
```

beta.json

```
{
  "global": {
  },
  "httpClient": {
    "tokenKey": "myapp-token"
  },
  "apiClient": {
    "type": "live"
  },
  "items": {
    "apiUrls": {
      "fetchItems": "/path/to/your/real/BETA/api/and-point"
    }
  }
}
```

live.json

```
{
  "global": {
  },
  "httpClient": {
    "tokenKey": "myapp-token"
  },
  "apiClient": {
    "type": "live"
  },
  "items": {
    "apiUrls": {
      "fetchItems": "/path/to/your/real/LIVE/api/and-point"
    }
  }
}
```

tsconfig.json

In the next section, we'll be loading the individual **config** files through **import** statements. In order to enable this in **TypeScript**, we have to modify the **tsconfig.json** file located in the root of your project. We need to add option **resolveJsonModule** with **true** to the **compilerOptions** section:

```
{
  "compilerOptions": {
    ...
    "allowSyntheticDefaultImports": true,
    "resolveJsonModule": true, /* this allows to import .json
    file as if they were .ts files: using to load config files */
    "sourceMap": true,
    ...
  }
}
```

src/config/index.ts

Add a new file called **index.ts** under **src/config**. Here we'll be creating our **ConfigInterface** by using the **JSON** file for the environment specified by the current **VUE_APP_CONFIG** value.

Let's start by importing our **ConfigInterface** and all the individual **config JSON** files:

```
import { ConfigInterface } from './Config.interface'

// individual environments configs:
import configMock from './config-files/mock.json'
import configLocal from './config-files/local.json'
import configBeta from './config-files/beta.json'
import configLive from './config-files/live.json'
...

```

Create an **env** variable with a default value of **mock**. Then set the **env** variable with the current **VUE_APP_CONFIG** environment variable, if any:

...

```
// return appropriate config based on env VUE_APP_CONFIG  
let env: string = 'mock' /* by default we return the mock  
configuration */
```

```
// if our env VUE_APP_CONFIG variable is set, get its value  
if (process.env && process.env.VUE_APP_CONFIG) {  
  env = process.env.VUE_APP_CONFIG.trim().toLowerCase()  
}
```

...

Finally, you can either use a **strategy pattern** or a **JavaScript Map** that contains a map to our **JSON** files by environment key (here we are showing this with **Map**). We also add a check and throw an error if our map does not contain an entry for the current environment key. Then we export an instance of our **ConfigInterface** called **config**:

...

```
// you can use a strategy pattern, or a javascript Map()
export const configsMap: Map<string, ConfigInterface> = new
Map<string, ConfigInterface>([
  ['mock', configMock],
  ['local', configLocal],
  ['beta', configBeta],
  ['live', configLive]
])

if (!configsMap.has(env)) {
  throw Error(`Could not find config for VUE_APP_CONFIG key "${
    { env }}`)
}

export const config: ConfigInterface = configsMap.get(env) as
ConfigInterface
```

Unit Tests against Config

Let's now write a few **unit tests** to validate that our **config** is being set as expected. This will also validate that the config **JSON** files contains the expected data structure.

Unit Tests against configsMap

Create directory **tests/unit/config** and add a new file called **Config.configsMap.spec.ts**. Here we'll be testing that our **configsMap** instance contains at least one entry for each environment, as expected:

```

import { expect } from 'chai'
import { configsMap } from '@config'

describe('configsMap', () => {

  it('instance should have "mock" key', () => {
    expect(configsMap.has('mock')).to.equal(true)
  })

  it('instance should have "local" key', () => {
    expect(configsMap.has('local')).to.equal(true)
  })

  it('instance should have "beta" key', () => {
    expect(configsMap.has('beta')).to.equal(true)
  })

  it('instance should have "live" key', () => {
    expect(configsMap.has('live')).to.equal(true)
  })
})

```

Do not run the unit tests yet, let's create one more and then we'll have to update the **package.json** file shortly to set the appropriate **VUE_APP_CONFIG** values.

Unit Tests against Config instance

Also under **tests/unit/config** directory, add another file called **Config.spec.ts**. Here we'll be testing that the **config** instances contains all the expected sections

and properties:

```
import { expect } from 'chai'
import { config } from '@/config'

describe('config', () => {
  it('instance should have "global" section', () => {
    expect(config).to.have.own.property('global')
  })

  it('instance should have "httpClient" section', () => {
    expect(config).to.have.own.property('httpClient')
  })

  it('instance should have "items" section', () => {
    expect(config).to.have.own.property('items')
  })

  describe('global', () => {
    const section = config.global
    // tests against the global section eventually go here
  })

  describe('httpClient', () => {
    const section = config.httpClient
    it('section should have "tokenKey" property', () => {
      expect(section).to.have.own.property('tokenKey')
    })
  })
})
```

...

...

```
describe('apiClient', () => {  
  const section = config.apiClient  
  it('section should have "type" property', () => {  
    expect(section).to.have.own.property('type')  
  })  
})
```

```
describe('items', () => {  
  const section = config.items
```

```
    it('section should have "apiUrls" property', () => {  
      expect(section).to.have.own.property('apiUrls')  
    })
```

```
    describe('apiUrls', () => {  
      const apiUrls = section.apiUrls
```

```
        it('section should have "fetchItems" property', () => {  
          expect(apiUrls).to.have.own.property('fetchItems')  
          // verify that fetchItems url is a string and has a  
          reasonable length  
          expect(apiUrls.fetchItems).to.be.a('string')  
          expect(apiUrls.fetchItems).to.have.length.greaterThan(10)  
        })  
      })  
    })  
  })  
})
```

Update package.json Script section

Now modify the `package.json` scripts shortcuts for our **unit tests** to have one

entry for each different environment. We'll create 4 shortcuts:

- test-mock
- test-local
- test-beta
- test-live

We do this by using **cross-env** to set the **VUE_APP_CONFIG** to the appropriate environment key:

```
...  
  
"scripts": {  
  "serve": "cross-env VUE_APP_API_CLIENT=mock vue-cli-service  
serve --mode development",  
  "build": "cross-env VUE_APP_API_CLIENT=live vue-cli-service  
build --mode production",  
  "build-mock": "cross-env VUE_APP_API_CLIENT=mock vue-cli-  
service build --mode production",  
  "test:unit": "vue-cli-service test:unit",  
  "test-mock": "cross-env VUE_APP_CONFIG=mock vue-cli-service  
test:unit",  
  "test-local": "cross-env VUE_APP_CONFIG=local vue-cli-service  
test:unit",  
  "test-beta": "cross-env VUE_APP_CONFIG=beta vue-cli-service  
test:unit",  
  "test-live": "cross-env VUE_APP_CONFIG=live vue-cli-service  
test:unit",  
}
```

```
...
```

Now you can finally run these new **unit tests** using a different configuration **JSON** file for each environment by just using one of the 4 commands. I.e. **npm run test-mock**:

```
DONE Compiled successfully in 3172ms
```

```
...
```

```
MOCHA Testing...
```

```
  configsMap
```

- ✓ instance should have "mock" key
- ✓ instance should have "local" key
- ✓ instance should have "beta" key
- ✓ instance should have "live" key

```
  config
```

- ✓ instance should have "global" section
- ✓ instance should have "items" section

```
    items
```

- ✓ section should have "apiUrls" property

```
      apiUrls
```

- ✓ section should have "fetchItems" property

```
  HttpClient.get
```

- ✓ should succeed and return data (102ms)

```
  HttpClient.get
```

```
----- rejecting -----
```

- ✓ should reject and return 400 (103ms)

```
  Item.component.vue
```

- ✓ renders an Item correctly
- ✓ has expected css class when selected is false
- ✓ has selected css class when selected is true

```
  13 passing (231ms)
```

```
MOCHA Tests completed successfully
```

*NOTE: In a continuous integration (CI) setup, you might have a task that will run only the **text-xyz** command that is related to the CI environment. If you CI builds for both production (live) and QA (beta) you could add an additional entry that calls both of them at once: i.e. **npm run test-beta && npm run test-live***

Now that we have verified our configuration, we need to also update the other commands like **serve/build** etc. We'll be adding more commands in later chapters, but for now let's modify the existing 3 like this:

```
...  
  "scripts": {  
    "serve": "cross-env VUE_APP_API_CLIENT=mock  
VUE_APP_CONFIG=mock vue-cli-service serve --mode development",  
    "build": "cross-env VUE_APP_API_CLIENT=live  
VUE_APP_CONFIG=live vue-cli-service build --mode production",  
    "build-mock": "cross-env VUE_APP_API_CLIENT=mock  
VUE_APP_CONFIG=mock vue-cli-service build --mode production",  
    "test-mock": "cross-env VUE_APP_CONFIG=mock vue-cli-service  
test:unit",  
    "test-local": "cross-env VUE_APP_CONFIG=local vue-cli-service  
test:unit",  
    "test-beta": "cross-env VUE_APP_CONFIG=beta vue-cli-service  
test:unit",  
    "test-live": "cross-env VUE_APP_CONFIG=live vue-cli-service  
test:unit",  
  }  
  ...
```

Note how we are also removing the current **VUE_APP_API_CLIENT** env

variable. At this point, it does not make any more sense to drive our API client choice through the env variable like we first did, but we can drive it from values within our config files.

HttpClient.model.ts

Update the **HttpClientModel** to use the token key from **config** value instead of the old **VUE_APP_TOKEN_KEY**:

```
...
import { config } from '@config'
...
export class HttpClientModel implements HttpClientInterface {
  private getToken(): string {
    const TOKEN_KEY =
    process.env && process.env.VUE_APP_TOKEN_KEY
    ? process.env.VUE_APP_TOKEN_KEY
    : 'myapp-token'
    const TOKEN_KEY = config.httpClient.tokenKey || 'myapp-token'
    const token = localStorage.getItem(TOKEN_KEY) || ''
    return token
  }
}
```

src/api-client/index.ts

Update the code that returns the API client instance to use the **type** value from the **config** instead of the olde **VUE_APP_API_CLIENT** variable:

```

...
import { config } from '@config'
...
let env: string = 'mock'
if (process.env && process.env.VUE_APP_API_CLIENT) {
  env = process.env.VUE_APP_API_CLIENT.trim()
}

// return either the live or the mock client
let apiClient: ApiClientInterface
if (env === 'live') {
  apiClient = apiLiveClient
} else {
  apiClient = apiMockClient
}
if (config.apiClient.type === 'live') {
  apiClient = apiLiveClient
} else if (config.apiClient.type === 'mock') {
  apiClient = apiMockClient
} else {
  throw Error('Invalid or undefined config.apiClient.type')
}

export default apiClient

```

src/api-client/live/items/index.ts

Update the code that returns the Items API client instance to use the **urls** from the **config**:

```
...
import { config } from '@config'

// urls for this API client
const urls: ItemsApiClientUrlsInterface = {
  fetchItems: '/path/to/your/real/api/and-point'
}
const urls: ItemsApiClientUrlsInterface = config.items.apiUrl

// instantiate the ItemsApiClient pointing at the url that
// returns the live data from an API server
const itemsApiClient: ItemsApiClientInterface = new
ItemsApiClientModel(urls)
// export our instance
export default itemsApiClient
```

src/api-client/mock/items/index.ts

Update the code that returns the Items API client instance to use the **urls** from the **config**:


```

...
import { config } from '@config'

// urls for this API client
const urls: ItemsApiClientUrlsInterface = {
  fetchItems: '/static/data/items.json'
}
const urls: ItemsApiClientUrlsInterface = config.items.apiUrl

// instantiate the ItemsApiClient pointing at the url that
// returns static json mock data
const itemsApiClient: ItemsApiClientInterface = new
ItemsApiClientModel(urls)
// export our instance
export default itemsApiClient

```

*NOTE: At this point, thanks to the new way of driving things through the **config**, the code in both files **src/api-client/live/items/index.ts** and **src/api-client/mock/items/index.ts** is identical. In later chapters we will simplify and reduce the amount of code we initially created to serve either **mock** or **live** data.*

Now make sure you run all the unit tests again, then serve the app again to make sure all compiles and works as before.

Chapter 14 Recap

What We Learned

- We learned how to use static **JSON** files to have multiple configuration settings, one for each environment
- How to dynamically return the appropriate config file based on the new environment variable **VUE_APP_CONFIG**
- How to add option **resolveJsonModule** to the the **TypeScript** **tsconfig.json** file, section **compilerOptions** to allow importing static **JSON** files through import statement
- How to write unit tests against our configuration

Observations

- For now our configuration is pretty small, but might grow larger as the application itself grows and we need to add more configurable options.

Improvements

Going forward we'll be expanding the configuration as we keep growing our application components and logic.

Chapter 15

USING SCSS LIBRARIES

So far we have used very simple **SCSS** included within the `<style>` section of our `.vue` files. There is nothing wrong with this approach, but if we want to be able to apply a uniform style and branding throughout our entire application, it makes more sense to add the SCSS through the plugins.

This has a few advantages. The most important advantage is the ability to easily switch between CSS frameworks in the future, should we need to. Switching from one CSS framework to another is not necessarily easy as it also depends on the HTML structure and the names of the CSS classes applied to your elements. However, in this book we'll use the plugin approach to at least make it easy in your **Vue** app to switch between SCSS libraries (or use multiple libraries, like in the case of the **flag-icons** SCSS).

We start by removing all the SCSS code currently within `<style>` tags, and create our own custom SCSS library called **MyAppScss** (later in other projects you are free to use any SCSS library you like).

Since there is quite a bit of SCSS code here, it is not practical to try explaining all the changes with examples in this chapter. But you can download the latest version of the SCSS from the book companion code on GitHub.

SCSS Library

Download the content of the **src/assets/scss** directory from the GitHub repo and put it under your project at the same path.

You can see that the **scss** directory contains two folders:

- **flag-icons** (here I moved the previous **flag-icons.scss** file and renamed it **index.scss** to keep things organized in a consistent way)
- **myapp-scss** (here I put all the SCSS I extracted from each component **.vue** files, which has also been organized in a certain way plus updated to support different themes)

Remove all the `<style>` code from the components

Remove all the `<style>` sections and their content from all the components where we have it. For example, **Loader.component.vue**, **ItemsList.component.vuex**, **Ite.component.vue**, etc.

Flag-Icons Plugin Code update

Update the code within **src/plugins/flag-icons/index.ts** to refer to the new location for the **flag-icons scss**:

```
export const FlagIconsScss = {  
  install() {  
    require('../assets/scss/flag-icons.scss')  
    require('../assets/scss/flag-icons/index.scss')  
  }  
}
```

MyAppScss Plugin

Create the directory **src/plugins/myapp-scss** and inside here create a file called **index.ts** with the following code:

```
export const MyAppScss = {  
  install() {  
    require('../assets/scss/myapp-scss/index.scss')  
  }  
}
```

All we are doing here is use the install method of the plugin to include the main **index.scss** from our SCSS library, which is located within the **src/assets/scss/myapp-scss** folder.

main.ts Updates

Update the **src/main.ts** to import the new **MyAppScss** plugin and tell Vue to

“use” it:

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import { store } from './store'
import { i18n } from '@plugins/vue-i18n-next-plugin'
import { MyAppScss } from '@plugins/myapp-scss'
import { FlagIconsScss } from '@plugins/flags-icons/'

createApp(App)
  .use(store)
  .use(router)
  .use(i18n)
  .use(MyAppScss)
  .use(FlagIconsScss)
  .mount('#app')
```

public/index.html Updates

We could do this in a different way, but since we’ll be using the font **Open Sans** and **Material Icons** let’s just reference them from **fonts.googleapis.com**:

```
<!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><%= htmlWebpackPlugin.options.title %></title>
    <link rel="stylesheet" href="https://fonts.googleapis.com/
css?family=Open+Sans:300,400,600,700|
Material+Icons&lang=en&display=swap"/>
    ...
```

Now, if you run the app again it should all still work and use the centralized SCSS thanks to the **MyAppScss** plugin we created.

ThemeSelector component

Another thing we can do for fun, is to create a component called **ThemeSelector** that will allow us to switch between different themes.

Create the directory **src/components/theme-selector** and here add a new file called **ThemeSelector.component.vue** with the following **<template>** code:

```

<template>
  <div class="theme-selector">
    <div class="theme-radio-group">
      <label
        role="radio"
        v-for="(theme, index) in themes"
        :key="index"
        :class="'theme-radio ' + theme.name + ' ' + theme.selected ?
'selected' : ''".trim()"
      >
        <i class="material-icons">color_lens</i>
        <input
          type="radio"
          class="icon-button"
          name="locale"
          :value="theme.selected"
          v-model="theme.selected"
          @click="onThemeClicked(theme)"
        />
      </label>
    </div>
  </div>
</template>
...

```

Here we are creating a group of **radio** buttons using a **v-for** binding and setting their CSS **class** attribute dynamically adding the theme **name** and the **selected** class for the one that is currently selected.

For the **<script>** section, we define an interface called **ThemeInfoInterface** which defines the properties for each of our themes. These are

- **name** (the friendly name of our theme)
- **selected** (to track which one is currently selected)
- **bodyCss** (the name of the **body** **css** **class** we need to apply to the `<body>` element in order to change theme)

We also define an array called **themes** that hold information for our 3 different themes (**light**, **dark**, **navy**):

```
...  
<script lang="ts">  
  import { defineComponent, reactive, onMounted, ref } from 'vue'  
  
  interface ThemeInfoInterface {  
    selected: boolean  
    name: string  
    bodyCss: string  
  }  
  
  const themes: ThemeInfoInterface[] = reactive([  
    {  
      selected: false,  
      name: 'light',  
      bodyCss: 'default'  
    }, {  
      selected: false,  
      name: 'dark',  
      bodyCss: 'dark-theme'  
    }, {  
      selected: false,  
      name: 'navy',  
      bodyCss: 'navy-theme'  
    }  
  ])  
...  

```

Finally the component declaration. Here we exposes the **themes** array and the handler **onThemeClicked** to the template. Within the **click handler** we set the **document.body** class to the clicked theme **bodyCss** value, and also ensure that only the clicked theme is set to **selected**:

```

...
const ThemeSelectorComponent = {
  components: {
  },
  setup(props: any, { emit }: any) {
    const onThemeClicked = (themeClicked: ThemeInfoInterface)
=> {
      document.body.className = ''
      document.body.classList.add(themeClicked.bodyCss)
      // select only the clicked theme
      themes.forEach(theme => {
        theme.selected = theme.name === themeClicked.name
      })
    }
  },
  onMounted(() => {
    const defaultTheme = themes.find(theme => theme.name ===
'dark')
    if (defaultTheme) {
      onThemeClicked(defaultTheme)
    }
  })
  return {
    themes,
    onThemeClicked
  }
}

export default defineComponent(ThemeSelectorComponent)
</script>

```

Not how we also use `onMounted` to preselect a **default theme (dark)**.

App.vue updates

Now let's import and use the **ThemeComponent**. Modify the **App.vue** file

`<template>` section as follows:

```
<template>
  <div id="app">
    <div class="long-date">{{ i18n.d(new Date(), 'long') }}</div>
    <h2>{{ i18n.t('welcome') }}</h2>
    <ThemeSelector />
    <LocaleSelector
      :availableLocales="availableLocales"
      @clicked="onLocaleClicked"
    />
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
    </div>
    <router-view />
  </div>
</template>
...
```

And the `<script>` section as follows:

```

...
<script lang="ts">
  import { defineComponent, computed } from 'vue'
  import { useI18n } from 'vue-i18n'
  import { MutationType, StoreModuleNames } from '@models/store'
  import { store } from '@store'
  import { LocaleInfoInterface } from '@models/localization/
LocaleInfo.interface'
  import LocaleSelector from '@components/locale-selector/
LocaleSelector.component.vue'
  import ThemeSelector from '@components/theme-selector/
ThemeSelector.component.vue'

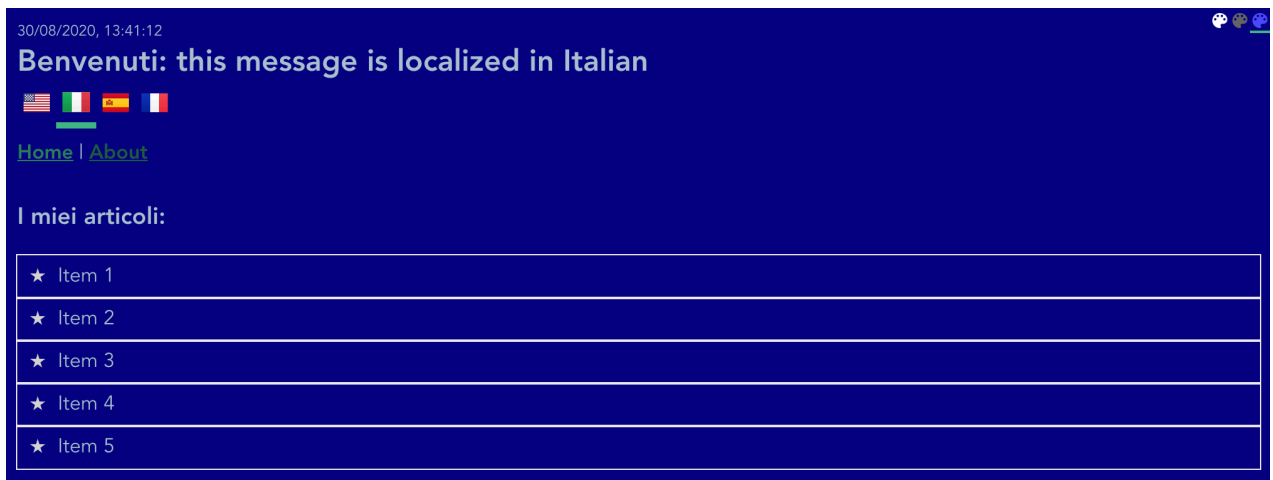
  export default defineComponent({
    name: 'App',
    components: {
      LocaleSelector,
      ThemeSelector
    },
    setup() {
...

```

If you now run the app and refresh your browser, it should default to the **dark** theme:



You can then click on the top-right icons to switch between themes. For example, here is after clicking on the icon for the **blue** theme:



Chapter 15 Recap

What We Learned

- We learned on how to organize SCSS libraries under our `src/assets/scss` folder
- We learned how to include those libraries by creating a plugin
- We learned hoe to create a component to switch between themes

Observations

- We did not write unit tests
- We did not show how to add a third-party SCSS library

Based on these observations, there are a few improvements that you could do on your own:

Improvements

- Write unit tests against the ThemeSelector component
- Try adding a third-party SCSS library and consume it with a custom plugin like we did for **MyAppCss**

THE VUE.CONFIG.JS FILE

I will describe the **vue.config.js** file here. For now just adding it to the companion code on **GitHub**. Here is what is currently used in the companion code:


```

const path = require('path');

module.exports = {
  devServer: {
    port: 8080
  },
  productionSourceMap: true,
  outputDir: 'dist',
  /* publicPath is used when we execute "npm run build" to prefix
  references for script/css files */
  publicPath: './',
  /* chainWebpack/configureWebpack so that npm run build does not
  produce hashed file names under dist/ folder */
  chainWebpack: (config) => {
    config.resolve.alias.set('@', path.resolve(__dirname, 'src'))

    if (config.plugins.has('html')) {
      const htmlPlugin = config.plugin('html');
      if (htmlPlugin) {
        htmlPlugin.tap(args => {
          args[0].title = 'Vue 3 Large Scale Apps with TypeScript
Sample Project';
          return args;
        });
      }
    }
  }
  ...

```

...

```
if (config.plugins.has('extract-css')) {  
  const extractCSSPlugin = config.plugin('extract-css');  
  extractCSSPlugin && extractCSSPlugin.tap(() => [{  
    filename: '[name].css',  
    chunkFilename: '[name].css'  
  }])  
}
```

```
if (process.env.NODE_ENV === 'production') {  
  // if we need to make exception for some things that should  
  not go into the dist folder  
}  
,  
configureWebpack: {  
  output: {  
    filename: '[name].js',  
    chunkFilename: '[name].js'  
  },  
  resolve: {  
    alias: {  
      "@": path.resolve(__dirname, 'src')  
    }  
  }  
}  
}
```

Naming Standards

NAMING CONVENTIONS

In this book we have been providing some direction on both naming standards for code elements like interface, classes etc, as well as for directory and file names. Here is a detailed description of the standard we followed in this book.

NOTE: These are mostly suggestions, a starting point. You should always agree with your team on the naming conventions and directory structure standards that will work best for your company. Then the whole team should commit to follow those conventions.

Coding Standards

TypeScript any

Avoid using **any** and rather always choose an interface/type/class

Interfaces

Interfaces are named with an **Interface** suffix. For example, an interface representing **Item** will be named **ItemInterface**.

Each interface will be contained in its own file. The file naming convention will be **Item.interface.ts**.

Directory/File Naming and Structure

Directory Names

In general, we'll use lower-case for naming directories. When this contains multiple words, they will be separated by a hyphen (dash). I.e. **items-with-**

children

We try to keep code files organized in directories by category (i.e. **components**, **models**, **plugins**) and sub-directories

Sub-directories are organized by app domain for models, i.e. **models/items**, **models/customers**, **models/locales** etc

For components, they are organized by component domain or functionality, i.e. **components/items**, **components/locales** etc.

In general, if a model or a component is used across all domains, then the sub-directory name is **shared** (or **common** if you prefer), i.e. **components/shared**

File Names

In general, files will be named with a pascal-case convention, I.e.

ItemsWithChildren.ts

Interface File Names

Files containing interfaces will follow the convention **[Name].interface.ts**, i.e. **Item.interface.ts**.

Vue single-file Components File Names

Vue components files will be under **src/components** directory.

Their names follow the convention **[ComponentName].component.vue**. I.e.

ItemsList.component.vue

Vue single-file Views File Names

Vue views files will be under **src/views** directory.

Their names follow the convention **[ViewName].vue**, without adding any suffix like for the components (*NOTE: in **Vue**, everything is really a component, including views. The separation is mostly for organization purposes. The way we consume views and components differs and we talk more about this throughout the book*).

Unit Tests file names

For unit tests, we'll follow the convention

[ClassOrComponentBeingTested].spec.ts. I.e. **ItemsListComponent.spec.ts**

NOTE: If you have to write many unit tests against the same class or component to test specific areas (i.e. security, permissions etc) might be a good idea to also split the code into additional files named with additional suffixes (as long as you adopt a standard that makes sense and it's easy to follow).

This could be a convention to follow in that case:

[ClassOrComponentBeingTested].[area-tested].[condition].[value].spec.ts

and here are a couple of examples:

- **ItemsListComponent.permissions.view.no.vue** (to test when user does not have View permissions)
- **ItemsListComponent.permissions.view.yes.vue** (to test when user has view permissions)

Directory src

Contains all the **Vue** source code

- **src/assets**: contains static assets like image files etc
 - **src/assets/images**: contains all the static images

- **src/api**: contains the API clients implementations
 - **src/api/mock**: contains the API clients that return mock data
 - **src/api/live**: contains the API clients that communicate with the real API end-points

- **src/components**: contains all the **Vue** single file components (extension **.vue**)
 - **src/components/[lowercase-component-name]**: contains all the files that make up a specific component. I.e. **src/components/items**
 - **src/components/[lowercase-component-name]/children**: contains all the sub-components, if any, consumed only by our main component. I.e. **src/components/items/children**

- **src/models**: contains all the pure TypeScript interface/types/classes/models/etc (extension **.ts**), with the exception of the **Vue main.ts** file which is located directly under **src/**
 - **src/models/[domain]**: contains all the interfaces/classes/etc that are related to a particular domain, I.e. **items**
 - **src/models/store**: contains all the interfaces for our **Vuex** store and store modules organized in sub-directories by specific domain:
 - **src/models/store/[domain]** contains all the interfaces for a specific **Vuex** store module

- **src/router**: contains the **Vue** router code implementation

- **src/store**: contains the **Vuex** store implementation

- `src/store/[domain]`: contains the store module implementation for a specific domain, I.e. **items**
- `src/views`: contains all the **Vue** single file views (extension **.vue**), except for the **App.vue** which is located directly under **src/**

Directory `tests/unit`

Contains all the unit and end-to-end tests.

- `tests/unit`: contains the unit tests
 - `tests/unit/components`: contains all the unit tests against Vue components
 - `tests/unit/models`: contains all the unit tests against the models/interfaces/types etc
- `tests/end-to-end`: contains the end-to-end tests (not used in this book)

Chapter XYZ

MORE CHAPTERS COMING SOON

(in progress)