NEW EDITION

LARGE SCALE APPS with VUE, VITE and TYPESCRIPT

```
const stack: [
    "Vue",
    "Vite",
    "TypeScript",
    "Vitest",
    "TailWind CSS"
]
```

DAMIANO FUSCO

Large Scale Apps with Vue, Vite and TypeScript

Build Large and Scalable front-ends that leverage component isolation, internationalization, localization, a modular state manager, component Libraries, API-client code that easily can switch between mocked data and live data and more.

Damiano Fusco

This book is for sale at http://leanpub.com/vue-vite-typescript

This version was published on 2023-01-30

Tweet This Book!

Please help Damiano Fusco by spreading the word about this book on Twitter!

The suggested hashtag for this book is #vuejs, vite, typescript.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#vuejs, vite, typescript

Contents

LARGE SCALE APPS WITH VUE, VITE, AND TYPESCRIPT

This book is a guide for developers looking to build large-scale front-end applications with Vue and TypeScript. With the growth of the web and mobile app development, there is an increasing demand for robust, scalable, and maintainable front-end solutions. This book provides a comprehensive approach to building large scale code bases that use Vue and TypeScript.

The book covers key concepts and best practices like:

- **Building front-ends** that can grow to a large code base that is organized and easy to expand and maintain.
- **Development of UI components in isolation** using an API client that can easily serve live data or mocked data.
- **Centralized State Manager** organized into domain/area modules, providing a unified and consistent way to manage the application state.
- **Internationalization and Localization** for language translation and number/dates formatting according to a specific culture, making it easier to reach a global audience.
- **TypeScript** type-checking at development time to decrease run-time bugs or errors, reducing the risk of costly bugs and enhancing the overall quality of the code.
- **Directory structure, file, and code naming conventions**, ensuring a consistent and organized project structure for both developers and future maintainers.
- **Hooks and Compositional Patterns**, providing a flexible and reusable way to implement functionality in components.
- **Components Libraries**, allowing developers to build a library of reusable components, reducing development time and increasing code quality.
- **Unit tests** for models and components, ensuring code quality and reducing the risk of regressions.

The book is designed for developers with intermediate to advanced Vue and TypeScript skills who are looking to take their applications to the next level. Whether you are building a new large-scale app or optimizing an existing one, this book will provide you with the tools and knowledge to achieve your goals. Throughout the book, practical examples and real-world scenarios are used to illustrate key concepts and best practices, providing you with a solid understanding of how to build large scale apps with Vue and TypeScript.

Preface

Why Vue, Vite and what we mean by “large scale apps” in this book.

Vue offers a simple and intuitive API that allows for a reactive and modular architecture, making it a popular choice for building user interfaces. Additionally, Vue is highly performant and boasts a large and active community of developers, making it a great choice for building applications of any scale.

Vite, on the other hand, is a relatively new player in the front-end development world but brings a number of important benefits to the table. With Vite, we can build and serve our application in near real-time, making it ideal for development. Additionally, Vite has a small build size and is highly optimized for speed, making it a great choice for large scale applications. It offers several benefits, including:

- Faster build times and development experience compared to traditional bundlers.
- Lower initial load times, as only the essential code is loaded
- Improved build size, as Vite only includes the necessary code
- Lightweight and optimized for modern web development.

When we refer to “**large scale apps**”, we mean applications that have a large code base, a large number of users, and a wide range of functionality. These applications typically require efficient and scalable code that can handle high traffic and large amounts of data.

In this kind of projects there are several common concerns that arise, such as:

- Code maintenance and scalability
- Code quality and performance
- Code organization and structure

To address these concerns, here we will outline best practices for code organization and structure, such as using a centralized state manager and implementing strong-type checking with TypeScript. Additionally, we will focus on writing unit tests for our models and components, which will help improve code quality and catch bugs early in the development process.

Our ultimate goal is to build a foundation that can handle the demands of our app and be easy to expand and maintain as the code base grows.

Goal

The primary aim of this book is to guide you through the process of building a scalable Vue application by following best practices for project structure, file organization, naming conventions, state management, type checking with TypeScript, and compositional approaches using hooks.

Throughout the chapters, we will grow our simple project into a robust, large-scale application that is easy to expand and maintain, showcasing how patterns, conventions, and strategies can lay a solid foundation and keep the code organized and uncluttered.

We will build a TypeScript API client that can seamlessly switch between serving static mock data and communicating with a live API, allowing for front-end development to commence even before the back-end API is fully functional. Additionally, we will delve into topics such as internationalization, localization, and advanced techniques, to round out our comprehensive guide to building a scalable Vue application.

IMPORTANT: We will initially write code that allows us to achieve the desired functionality quickly, even if it requires more code, but then we constantly “rework” it (**refactoring**) to improve it and find solutions that allow us to reduce the amount of code used, or to organize it in a clear and easy way that is easy to expand and maintain. So arm yourself with a lot of patience!

IMPORTANT: We will initially write code that allows us to achieve the desired functionality quickly, even if it requires more code, but then we constantly “rework” it (**refactoring**) to improve it and find solutions that allow us to reduce the amount of code used, or to organize it in a clear and easy way that is easy to expand and maintain. So arm yourself with a lot of patience!

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 want to thank my son for helping me proof read and validate the steps in each chapter by building the same project. I also would like to thank all the developers that over the time helped me correct things in the book and provided valuable feedback.

[1]*Evan You on Twitter*: https://twitter.com/youyuxi* ⊠
[2]*Official website:* https://vuejs.org

About me

I have worked as a software developer for more than 20 years. I switched career from being a full time musician when I was 30 years-old and then became a graphic designer, then transition to a web designer when internet became "a thing", and for many years after that worked as full-stack developer using Microsoft .NET, JavaScript, Node.js and many other technologies. You can read more about me on my personal website https://www.damianofusco.com and LinkedIn profile https://www.linkedin.com/in/damianofusco/. You will find me also on Twitter, @damianome, and GitHub github.com/damianof

Audience

The audience for this book is from beginners with some experience in **MV*** applications, to intermediate 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, architectures, and technologies.

Note: Some of the patterns illustrated here are not specific to Vue, but can applied in any application written in **TypeScript** or **JavaScript**. For example, most code from Chapters 3, 7, and others can also be used in Svelte/React/Angular or other front-end apps. Similarly, code from Chapters 3 and 14 can also be used in **NodeJS** apps.

Text Conventions

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 help specifically for Vue code. For VS Code for example, you could use extensions like **Volar**[4] (just search for it within the VS code extensions tab).

[3] https://www.typescriptlang.org
[4] https://marketplace.visualstudio.com/items?itemName=vue.volar

Companion Code

The entire companion code for the book can be found on **GitHub** at: https://github.com/damianof/large-scale-apps-my-vue-vite-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-vue-vite-project/issues

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

Chapter 1 - Setting Up The Project

***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/*

There are many different ways to create a Vue app. Here we'll be leveraging TypeScript and therefore will need to setup an project with a build/transpile process that will let us make changes and verify them in real time. You could manually create this project, install all the npm packages required, create each individual file. However, it is just much easier to do this by leveraging **vite**[5]

Create Project Wizard

To set up the project, use the terminal and execute the following node.js command:

```
npm init vite@latest
```

If you do not have already installed the package `create-vite@latest`[6] it will prompt you to install it. In this case, type **y** and then **enter** to proceed:

```
Need to install the following packages:
  create-vite@latest
Ok to proceed? (y)
```

The create-vite wizard will start and will ask you the name of the project. The default is vite-project, so change this to **my-vue-project** and hit enter:

```
? Project name: › my-vue-project
```

The second step will ask to select a framework. Use the keyboard arrows to scroll down the list and stop at vue, then hit enter:

[5] https://vitejs.dev
[6] https://www.npmjs.com/package/create-vite

```
? Select a framework: › - Use arrow-keys. Return to submit.
    vanilla
▯   vue
    react
    preact
    lit
    svelte
```

The third step will asking which “variant” you want o use. Scroll down to **vue-ts** (this is for the version that uses TypeScript) and hit enter:

```
? Select a variant: › - Use arrow-keys. Return to submit.
    vue
▯   vue-ts
```

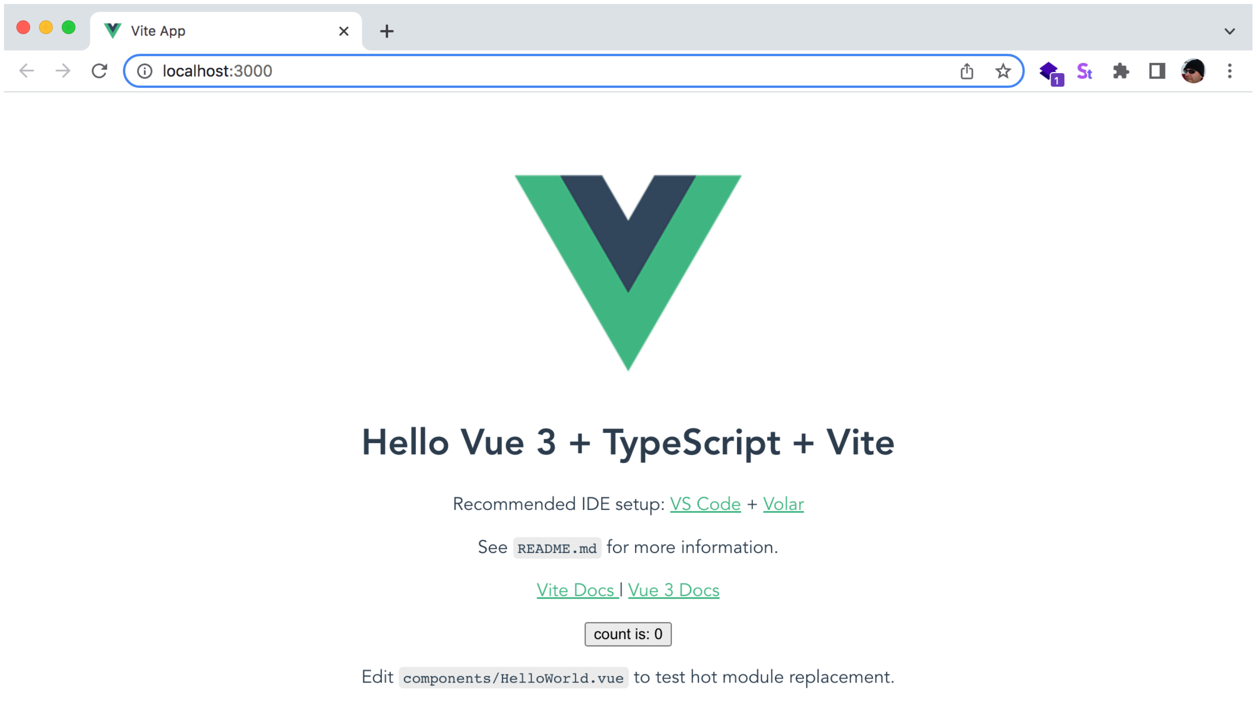
This will create a folder called **my-vue-project** which is also the name of our project.
At the end it should display a message similar to this:

```
Scaffolding project in /Volumes/projects/my-vue-project...

Done. Now run:

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

The first command will navigate to the current sub-directory called **my-vue-project**, the second one will install all the npm dependencies, the third one will serve the app locally. You’ll see a message similar to this displayed:

```
vite v2.9.7 dev server running at:

  > Local: http://localhost:3000/
  > Network: use `--host` to expose
```

From the web browser, navigate to the http://localhost:3000 address and you’ll see application home page rendered:



Hello Vue 3 + TypeScript + Vite

Recommended IDE setup: VS Code + Volar

See `README.md` for more information.

Vite Docs | Vue 3 Docs

count is: 0

Edit `components/HelloWorld.vue` to test hot module replacement.

The **my-vue-project** has been created with the main view called **App.vue** and a component called **HelloWorld.vue** which displays some static links to additional Vue resources plus an example of a simple counter.

Chapter 1 Recap

What We Learned

How to create the basic plumbing for a **Vue** app using the **vite** and `create-vite@latest`

- How to serve the app locally through the command **npm run dev**

Observations

- The app has been created with only preliminary code
- The app does not do much yet, has only the main App view with some static html in it and a reference to a simple component called Counter

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
- 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:

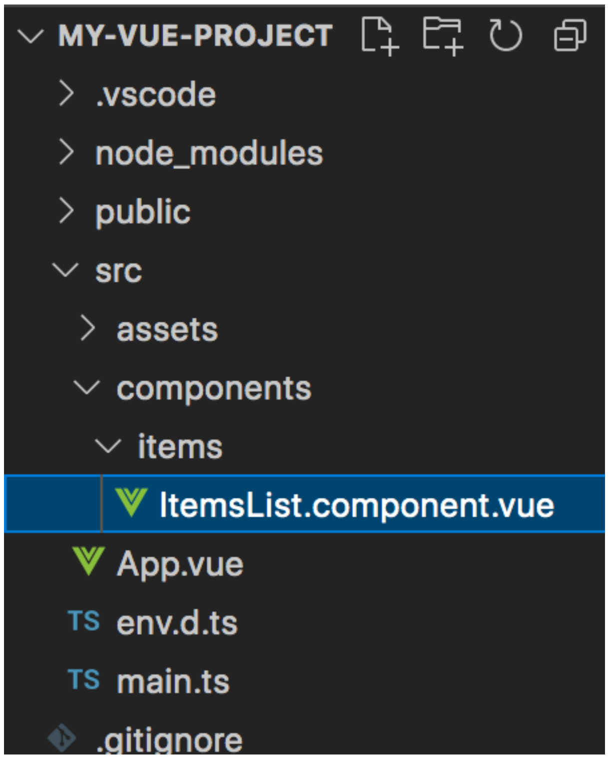
* **id**
* **name**
* **selected**

- 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**[7]

Your directory structure will now look like this:


```
MY-VUE-PROJECT
  > .vscode
  > node_modules
  > public
  v src
    > assets
    v components
      v items
        ItemsList.component.vue
    App.vue
    TS env.d.ts
    TS main.ts
  .gitignore
```

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

```
// file: src/components/items/ItemsList.component.vue
<script setup lang="ts">
  // expose a property called items with a default value of a blank array
  defineProps<{ items: any[] }>()
  // explicetely using any[] as we'll replace this with an interface in the next cha\
pters
</script>

<template>
  <div>
    <h3>Items:</h3>
```

[7] *We are following a file naming convention where higher level components' names are pascal-case and follow this format [Component-Name].component.vue (Reference: Naming Conventions section at the end of this book)*

```
    <ul>
      <li v-for="(item, index) in items"
        :key="item.id">
        {{item.name}}
      </li>
    </ul>
  </div>
</template>
```

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**. We define a property called **items** using Vue's **defineProps** [8]. When you using <script setup> this is how you create properties for your compopnent. If you prefer using **defineComponent** instead of <script setup> then this does not apply.

Note that **items** is just using the type **any**[9] for now (later we'll replace **any** with an interface we'll create).

For our html template, we added a **<h3>** element with hard-coded text just saying "**Items:**". Then a **<ul>** with a **v-for** binding that will render all our **items** within **<li>** elements.

App.vue

Open the existing **App.vue** file. Replace the existing code with this:

```
// file: src/App.vue

<script setup lang="ts">
 // TODO ....
</script>
<template>
  <div class="home">
   ... TODO
  </div
</template>
```

Within **<script>** the section, import a reference to the **ItemsListComponent**:

[8] *Vue's script setup and defineProps* https://vuejs.org/api/sfc-script-setup.html#defineprops-defineemits

[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'.*

```
// file: src/App.vue

<script setup lang="ts">
  // import a reference to our ItemsList component
  import ItemsListComponent from './components/items/ItemsList.component.vue'
...
```

For now, also mock some data for our list of items that we will feed to our **ItemsListComponent**. For this we instantiate a local const called **items** and initialize it with some hard-coded data[10]. Let;s just add it after the import statement:

```
// file: src/App.vue

<script setup lang="ts">
  // import a reference to our ItemsList component
  import ItemsListComponent from './components/items/ItemsList.component.vue'

  // mock some data:
  const items: any[] = [{ // explicetely using any[] as we'll replace this with an i\
nterface in the next chapters
    id: 1,
    name: 'Item 1'
  }, {
    id: 2,
    name: 'Item 2'
  }, {
    id: 3,
    name: 'Item 3'
  }]
</script>
...
```

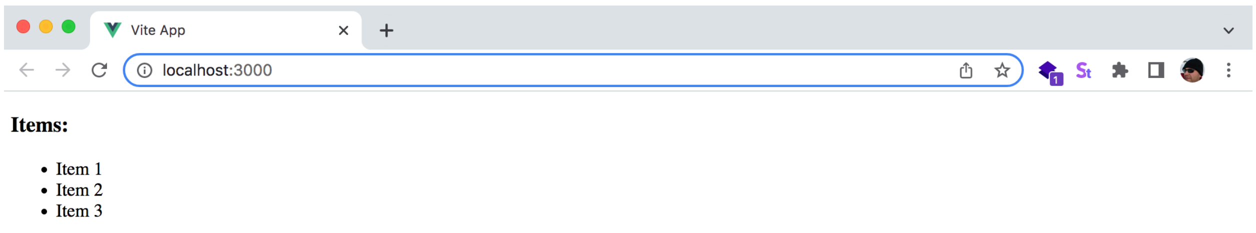
Finally, we add an **<ItemsListComponent>** element within the **<main>** markup. Insert it within the **<div class="home">** element. Add an attribute called **items** to our **<ItemsListComponent>**. This is how we pass properties to from parent to child components in Vue[11]. The **items** data is fed into the component **items** property this way. The complete code within the **App.vue** file should now look like this:

[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)*

[11]https://vue.dev/docs#template-syntax-attributes-and-props

```
// file: src/App.vue

<script lang="ts">
  // import a reference to our ItemsList component
  import ItemsListComponent from './components/items/ItemsList.component.vue'

  // mock some data:
  const items: any[] = [{
    id: 1,
    name: 'Item 1'
  }, {
    id: 2,
    name: 'Item 2'
  }, {
    id: 3,
    name: 'Item 3'
  }]
</script>

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

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



Items:

- Item 1
- Item 2
- Item 3

Note that when you pass properties by reference to other variables or properties you using the coon prefix like in `:items="items"`*. If instead you are passing a raw value like an hard-coded string, you would omit the colon. I.e.* `name="John"`

Chapter 2 Recap

What We Learned

- How to create a basic component that displays a list of items
- How to consume that component from another component/view

Observations

- The **items** property within the **ItemsList.component.vue** is declared as an array of type **any**
- The **App.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 delve into the power of **TypeScript** by leveraging its strong-type checking capabilities through the use of interfaces. One of the main challenges with pure JavaScript is its loosely typed nature, which can lead to unexpected behavior and bugs at run-time. This is due to the lack of checks on the type or properties of values or objects being passed around in the code. TypeScript solves this problem by providing developers with the ability to enforce strict type checking at development time through the use of interfaces, types, classes, and more.

By incorporating TypeScript into our project, we'll be able to catch potential issues and bugs before they reach the production environment, saving us time and resources in debugging and fixing. Moreover, strong-typing also improves the readability and maintainability of our code, making it easier for developers to understand the purpose and usage of values and objects in the codebase. In this chapter, we'll explore how to use interfaces and types to implement strong-typing in our project, and how it can help us ensure the reliability and quality of our code.

Models Directory

To lay the foundation for building large-scale applications, we will start by creating a new sub-directory under **src** called **models**. The organization of files and directories plays a critical role in the success of large-scale code bases. As such, it's essential to establish a consistent naming **convention** and directory structure from the outset. This will help ensure that the code remains organized, easy to understand, and maintainable as the application grows and the number of source files and directories increases.

You and your team are free to determine the standards that work best for you, but it's crucial to establish a set of conventions and stick to them. This will save you a significant amount of time and effort in the long run and prevent confusion and headaches as the application grows and evolves.

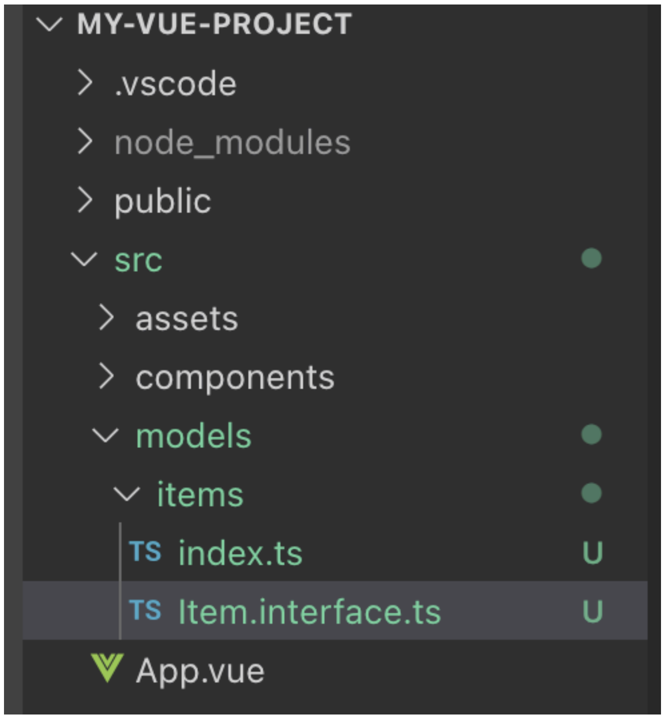
Interface ItemInterface

To create the interface for our items, we will create a new directory called **src/models/items** and add a TypeScript file named **Item.interface.ts**.

It's worth noting that there are different naming conventions for TypeScript interfaces, with some preferring to use a suffix like **Interface**, while others use a prefix like **I**. In this book, we will follow the suffix convention, using **Item.interface.ts** as the file name. However, you are free to choose your preferred naming convention or align with your organization's coding standards.

It's important to keep each interface in its own file, as this makes it easier to maintain and manage. For more information on naming conventions, please refer to the Naming Conventions section at the end of this book.

Your directory structure should now look similar to this:



```
MY-VUE-PROJECT
  > .vscode
  > node_modules
  > public
  v src
    > assets
    > components
    v models
      v items
        TS index.ts            U
        TS Item.interface.ts   U
    App.vue
```

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:

```
// file: src/models/items/Item.interface.ts

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

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 type[12] *and their guidance is to use interfaces for this.*

Finally, add a main index.ts barrel file at the root of `src/models` with this code:

```
// file: src/models/index.ts
export * from './items'
```

NOTE: barrel indexes files are not reqwuired, but make it a lot easier to import references in other code feils without having to specify the full path to the referenced file.

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

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** and modify our **items** property declaration from type **any[]** to type **ItemInterface[]**:

[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

```
// file: src/components/items/ItemsList.component.vue

<script setup lang="ts">
  // import a reference to our ItemInterace
  import type { ItemInterface } from '../../models/items/Item.interface'
  // expose a property called items with a default value of a blank array
  defineProps<{ items: ItemInterface[] }>() // here replace any[] with ItemInterace[]
</script>
...
```

The complete update code should look like this:

```
// file: src/components/items/ItemsList.component.vue
<script setup lang="ts">
  // import a reference to our ItemInterace
  import type { ItemInterface } from '../../models/items/Item.interface'

  // expose a property called items with a default value of a blank array
  defineProps<{ items: ItemInterface[] }>()
</script>

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

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

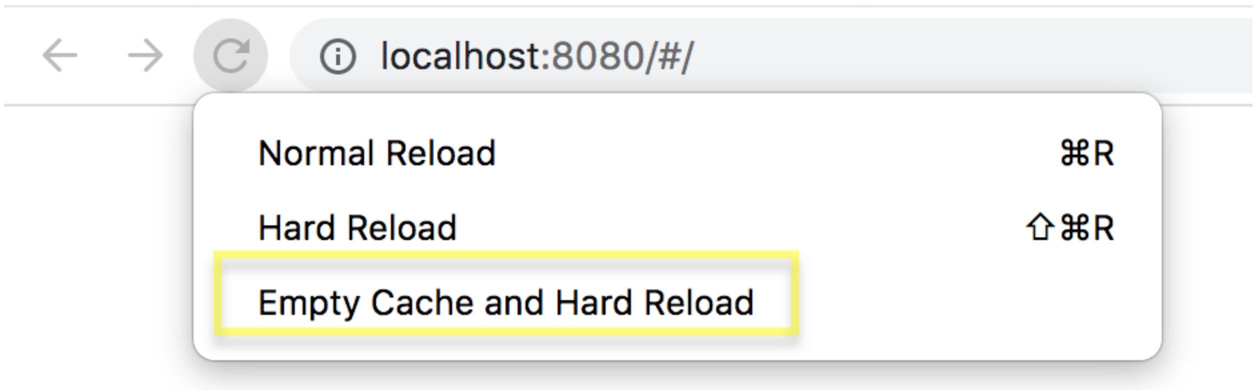
App.vue

We should also update the **App.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 catch errors like this at development time rather than run time, increase the code quality and make it less prone to bugs. So make sure each item has now also a selected field with a default of false.

```
// file: src/App.vue

<script setup lang="ts">
  // import a reference to our ItemInterface
  import type { ItemInterface } from './models/items/Item.interface'
  // import a reference to our ItemsList component
  import ItemsListComponent from './components/items/ItemsList.component.vue'

  // mock some data:
  const items: ItemInterface[] = [{ // here change any[] to ItemInterface[]
    id: 1,
    name: 'Item 1',
    selected: false // add selected: false to each item
  }, {
    id: 2,
    name: 'Item 2',
    selected: false // add selected: false to each item
  }, {
    id: 3,
    name: 'Item 3',
    selected: false // add selected: false to each item
  }]
</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:


localhost:8080/#/
Normal Reload ⌘R
Hard Reload ⇧⌘R
Empty Cache and Hard Reload

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 **App.vue** contains a local variable that holds hard-coded mocked data that enabled us to prototype our component quickly
- **ItemsList.component.vue** just displays 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 adding a function called **handleClick** . This function will handle a click on each of the **<li>** elements and will toggle the **item.selected** property from true to false or vice versa. It will also logs the item **id** and **selected** properties to the console for preliminary debugging:

```
// file: src/components/items/ItemsList.component.vue

<script setup lang="ts">
  // import a reference to our ItemInteface
  import type { ItemInterface } from '../../models/items/Item.interface'
  // expose a property called items with a default value of a blank array
  defineProps<{ items: ItemInterface[] }>()

  // item click handler
  const handleClick = (item: ItemInterface) => {
    item.selected = !item.selected
    console.log('handleItemClick', item.id, item.selected)
  }
</script>
...
```

And update the html by adding an **@click**[13] attribute to the **<li>** element, pointing to our handler **handleClick** and passing a reference to the **item** as the argument:

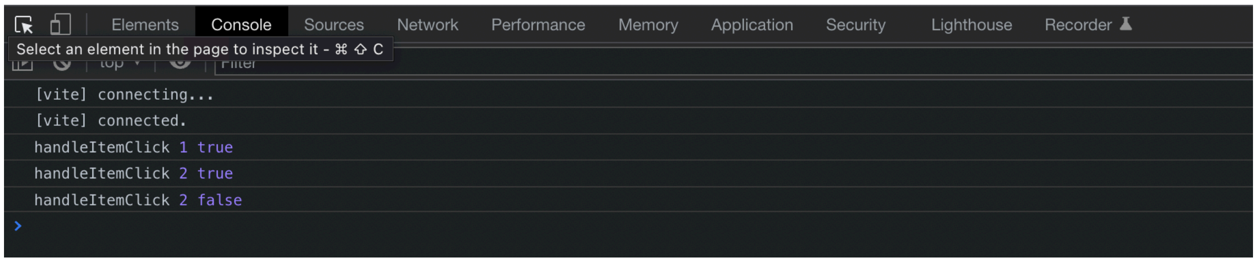
[13]https://vuejs.org/guide/essentials/event-handling.html

```
// file: src/components/items/ItemsList.component.vue

...

<div>
  <h3>Items:</h3>
  <ul>
    <li v-for="(item, index) in items"
        :key="item.id"
        @click="handleClick(item)">
      {{item.name}}
    </li>
  </ul>
</div>
```

Note the syntax we use in the `@click` attribute. We pass the item directly into the function. This is different from frameworks like React or Svelte where you have to either use an anonymous funciton or handle the MouseEvent as a parameter.

```
@click="handleClick(item)"
```

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



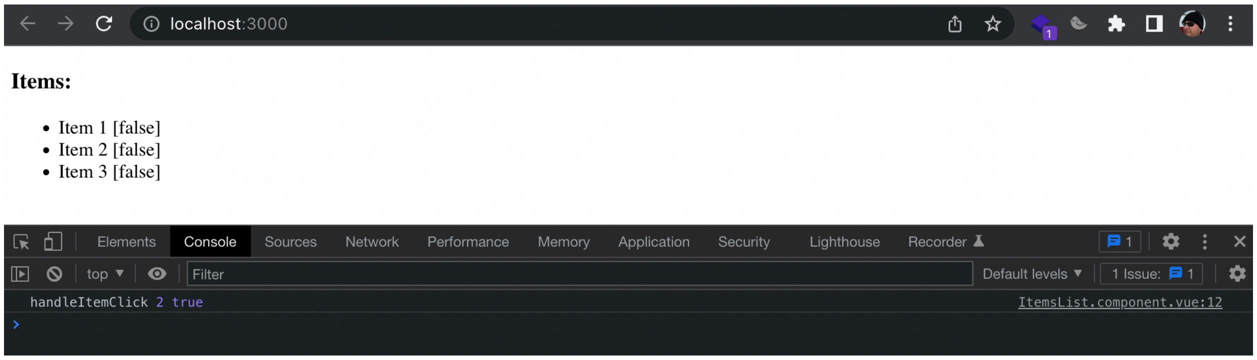
```
[vite] connecting...
[vite] connected.
handleItemClick 1 true
handleItemClick 2 true
handleItemClick 2 false
```

We just learned how to add a click handler to our component and changing the data item selected property that way. However, this in an anti-pattern in Vue as Vue follows a **one-way data flow** architecture where properties are only passed from parent to child components and can only be updated from the parent with reactive state (unless the reactive state is declared within the component itself).

Let's verify this. Start by slightly modifying the text output by our list element, outputting also the selected value within [] (square brackets) like "[}]":

```
...
      <li v-for="(item, index) in items"
        :key="item.id"
        @click="handleClick(item)">
        {{item.name}} [{{item.selected}}] <!-- add this -->
      </li>
...
```

Now the item.selected value will print next to the name and we can verify that Vue is not -re-rendering the html when we click on each item, even though the console.log statement outputs the correct value:


Items:

- Item 1 [false]
- Item 2 [false]
- Item 3 [false]

```
handleItemClick 2 true                                    ItemsList.component.vue:12
```

Let's fix this.

Open the the ItemsList.component.vue file and add a `defineEmits` just after the defineProps line. There are 2 ways to defineEmits within <script setup>:

- Just passing an array of strings with the name of each emit
- Using type-only declarations (with this you get all the power of TypeScript strong typing, so we'll use this)

Also modify the `handleClick` method to just emit the custom `selectItem` event by passing the `id` of the item as the argument:

```
// file: src/components/items/ItemsList.component.vue
<script setup lang="ts">
  // import a reference to our ItemInterace
  import type { ItemInterface } from '../../models/items'

  // expose a property called items with a default value of a blank array
  defineProps<{ items: ItemInterface[] }>()

  // define emits for custom events
  // const emit = defineEmits(['selectItem']) // <-- using only an array of strings
  const emit = defineEmits<{
    (e: 'selectItem', id: number): any // <-- using type-only declaration
  }>()

  // item click handler
  const handleClick = (item: ItemInterface) => {
    emit('selectItem', item.id) // <-- modify the body of the method by just emittin\
g "itemClicked"
  }
</script>

...
```

Note: the name of the custom event does not have to be ***selectItem****. If you prefer to follow a standard like* ***onSelectItem*** *to better identify that it is an event, feel free to do so)*

Now open the App.vue file and let’s make some changes in here too.

First, let’s create a `reactive` state out of our `items` array:

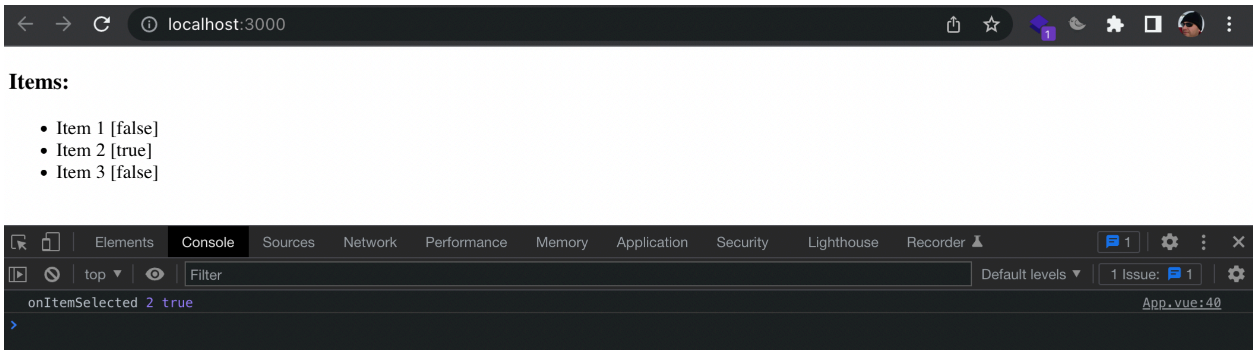
```
// file: App.vue
<script setup lang="ts">
  // import vue's reactive:
  import { reactive } from 'vue'

  ...

  // mock some data:
  const items = reactive([ // <-- now items array will be reactive
    {
      id: 1,
      name: 'Item 1',
      selected: false // add selected: false to each item
    }, {
      id: 2,
      name: 'Item 2',
      selected: false
    }, {
      id: 3,
      name: 'Item 3',
      selected: false
    }
  ])

  ...
```

Then, just before the closing </script> tag, add an `onSelectItem` event handler, and wireup the `@selectItem` event on the ItemsListComponent to point at this:

```
// file: App.vue
<script setup lang="ts">
  ...

  // add this block:
  // selectItem handler
  const onSelectItem = (id: number) => {
    // retrieve the item from our local data
    const item = items.find(o => o.id === id)
    // sanity check:
    if (!item) {
      console.warn(`onSelectItem: could not find item with id: ${ id }`)
      return
    }
    // update the item property
    item.selected = !item.selected
    console.log('onSelectItem', item.id, item.selected)
  }
</script>

<template>
  <div class="home">
    <ItemsListComponent :items="items" @selectItem="onSelectItem"/> <!-- wireup onSe\
lectItem handler -->
  </div>
</template>
```

Note: in Vue we wire up handling event using the @ character like in @selectItem or @click etc.

Save once more and check the browser again. This time you can verify the value of the selected property by looking at the output within the square brackets next to the name:



Items:

- Item 1 [false]
- Item 2 [true]
- Item 3 [false]

```
onItemSelected 2 true
```

Note: I personally prefer to always declare a local reactive state by calling it `state` *or* `internalState` *and use an object with properties for it. So in this case I would have declared* `const state = reactive({ items: [] })` *so that* `items` *is a property of state. But you can follow an convention you prefer of course.*

In the next chapter we'll talk more in depth on how to better manage the application state using centralized place and a State Manager

Chapter 4 Recap

What We Learned

- How to add a click handler to our **ItemsList** component
- How to define custom events within <script setup>
- How to use Vue's `reactive` to quickly create a local reactive state (items)
- How to manipulate the **item.selected** property by emitting a custom event (@selectItem) and handling it in our parent component (App.vue)

Observations

- 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 state manager to control our application state from a centralized place

Chapter 5 - Intro to Unit Testing While Refactoring a Bit

We will now delve into writing unit tests for our project. Unit tests serve as a critical aspect of ensuring the stability and reliability of our code. In this book, we will cover two main categories of unit tests:

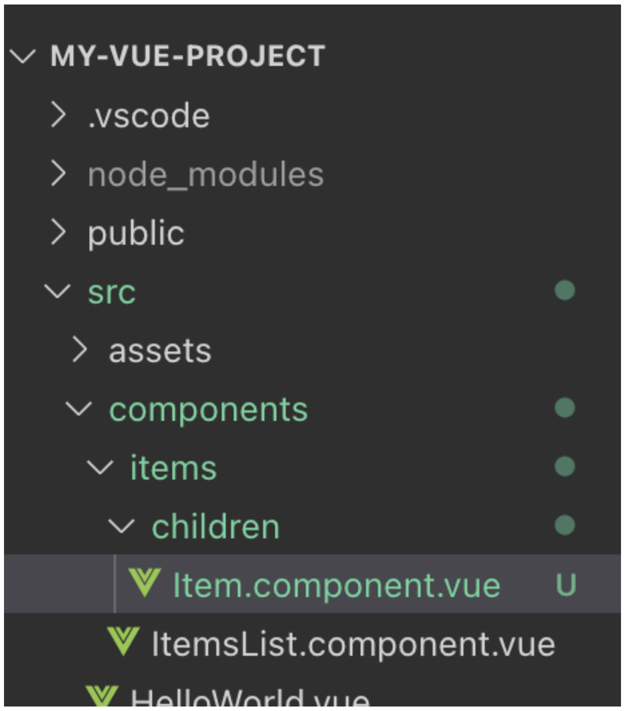
- Unit tests for models, classes, structures, and interfaces (such as the API client and helpers)
- Unit tests for Vue components

Note: It's worth mentioning that there is a third category of tests, known as end-to-end (e2e) tests, but we will not be covering those in this book.

Our first step will be to write unit tests for our Vue components. We will start with the **ItemsList** component and while doing so, we will make some refactors to improve its implementation. The unit tests will validate the changes we make, ensuring that our code remains functional and free of bugs.

ItemComponent

Remember how in our **ItemsList** component we have a loop that creates **<li>** elements, one for each item in our **items** property? Let's extract the code for the **<li>** 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:



Note: So far I have used the <script setup> feature to create components. For the Item.component.vue I am going to show you how to use Vue's **defineComponent** and **Composition API**. You are welcome to keep using <script setup> if you prefer. There is no particular recommended way, both ways are fine, but there seem to be a trend in our industry where developers want to write less code and that is what <script setup> tries to accomplish.

Paste the following code in the file:

```
// file: src/components/items/children/Item.component.vue
<template>
  <li :data-testid="testid" :class="cssClass" @click="handleClick">
    <div class="selected-indicator">*</div>
    <div class="name">{{model.name}} [{{model.selected}}]</div>
  </li>
</template>

<script lang="ts">
  // import reference to Vue's defineComponent
  import { defineComponent, computed, PropType } from 'vue'
  // import a reference to our ItemInterace
  import type { ItemInterface } from '../../../models/items/Item.interface'

  // create our component with defineComponent
  export default defineComponent({
    name: 'ItemComponent',
    emits: ['selectItem'],
    props: {
      // add testid prop here as well
      testid: {
        default: 'not-set'
      },
      // expose a property called model of type ItemInterface
      model: {
        type: Object as PropType<ItemInterface>,
        default: () => {
          return {}
        }
      }
    },
    setup(props, { emit }) {
      // a computed property to return a different css class based on the selected v\
alue
                  const cssClass = computed(() => {
        let css = 'item'
        if (props.model.selected) {
          css += ' selected'
        }
        return css.trim()
      })

      // on click event handler that will emit a @select custom event
      const handleClick = () => {
        emit('selectItem', props.model.id)
      }

      return {
        cssClass,
        handleClick
      }
    }
  })
</script>

<style>
  li.item {
    padding: 5px;
    outline: solid 1px #eee;
    display: flex;
    align-items: center;
    height: 30px;
    cursor: pointer;
    transition: background-color 0.3s ease;
  }
  li.item .name {
    margin-left: 6px;
  }
  li.item .selected-indicator {
    font-size: 2em;
    line-height: 0.5em;
    margin: 10px 8px 0 8px;
    color: lightgray;
  }
  li.item.selected .selected-indicator {
    color: skyblue;
  }
  li.item:hover {
    background-color: #eee;
  }
</style>
```

We just created a template for a single **<li>** element. We also enhanced this a bit by replacing the rendering of the name with the binding to name with 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 in the next chapters we'll be replacing this with real icons*)

Then we added a computed property called **cssClass** that will return the string "**item**" or "**item selected**". We then bind this to the **<li> class** attribute, based on whether the **model.selected** property is true or false: **<li :class="cssClass" @click="handleClick">**

This will have the effect to render the **<li>** element in two possible ways:

- <li class="item"> (when not selected)
- <li class="item selected"> (when selected)

We also bind to the click event with **@click** binding and in the local **handleClick** handler we just invoke the parent handler by emitting the custom event "selectItem" with Vue's **emit**[14] and passing the item id as the event argument (props.model.id). We will then handle this in the parent component (**ItemsList** component).

[14]Options: State | Vue.js (vuejs.org)

Additionally, we have starting to add a property called **testid** that will bind to html attribute **data-testid**. This is an industry standard that allow us to more easily retrieve a particual DOM element through automation and we will leverage this in our Vitest unit tests to more easily select the rendered elements.

Note that we also added a **<style>** section with some css to render our <li> element a little better. The css above is just a quick-and-dirty bit of styling so we can make our list look a bit prettier for now. In later chapters we'll introduce TailwindCSS and keep working with that instead of writing our own css.

ItemsList.component.vue updates

Now let's modify the ItemsList component so we can import and consume the newly created Item component:

```
// file: src/components/items/ItemsList.component.vue
<script setup lang="ts">

  ...

  // import a reference to the Item component:
  import ItemComponent from './children/Item.component.vue'

  ...
```

Modify the <template> section as follows:

```
// file: src/components/items/ItemsList.component.vue

  ...

<template>
  <div>
    <h3>Items:</h3>
    <ul>
      <ItemComponent v-for="(item, index) in items"
        :key="item.id"
        :model="item"
        @selectItem="handleClick(item)" />
    </ul>
  </div>
</template>
```

Let's also add some css at the end of our ItemsList.component.vue:

```
// file: src/components/items/ItemsList.component.vue

...

<style>
  ul {
    padding-inline-start: 0;
    margin-block-start: 0;
    margin-block-end: 0;
    margin-inline-start: 0px;
    margin-inline-end: 0px;
    padding-inline-start: 0px;
  }
</style>
```

And add some css at the end of our App.vue:

```
// file: src/App.vue

...

<style>
  .home {
    padding: 20px;
  }
</style>
```

Note: we are not consuming our new Item.component.vue anywhere yet. Let's proceed first to create a unit test against it and validate that it renders and behaves as we expect.

Add unit tests support to our project

We need to configure our project to be able to run unit tests. We need to add dependencies on a few npm packages and some configuration in order to be able to add unit tests for our components.

Dependencies

Vitest[15] has become popular very quicly recently as makes unit-testing in a Vite app much easier than Jest. Please read more about it on the official website to learn more about its features.

[15]https://vitest.dev

Let's start installing our npm dependencies first.

Install the packages @vitest + @testing-library/vue + @testing-library/user-event + @js-dom:

```
npm install --save-dev vitest @testing-library/vue @testing-library/user-event jsdom
```

test-utils code

Add directory **src/test-utils** and within it create file **index.ts**. Here we are going to export all we need in our unit tests from one place, and export also a wrapper around testing-library **render** (customRender) that will avoid having to repeat code in our unit tests:

```
// file:

/* eslint-disable import/export */
import { render } from '@testing-library/vue'

const customRender = <T>(ui: T, options = {}) =>
  render(ui, {
    ...options
  })

export * from '@testing-library/vue'
export { default as userEvent } from '@testing-library/user-event'
// override render export
export { customRender as render }
```

Configuration

Now we need to configure a few things to be able to run unit tests.

package.json

Within the package.json file, add the following command shortcuts within the script section:

```
...

  "scripts": {
    ...
    "test": "vitest run",
    "test-watch": "vitest watch",
    "test-coverage": "vitest run --coverage"
  }
...
```

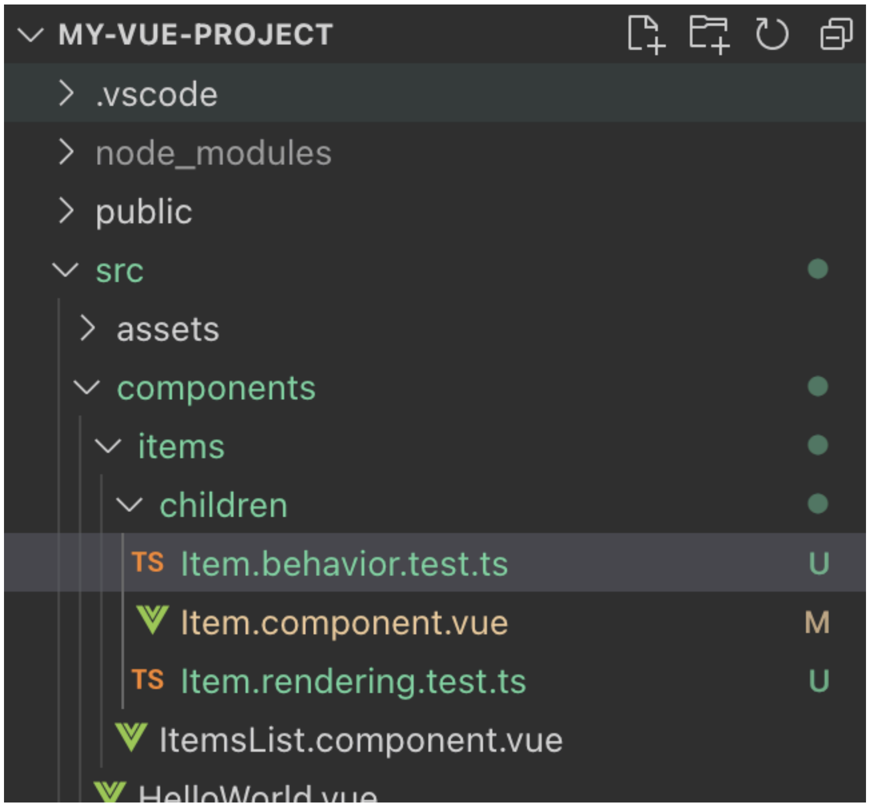
ItemComponent Unit Tests

Add our first two **unit tests** again our newly created component ItemComponent.

Within the same directory where our **Item.component.vue** is located, add two new files:

- one called **Item.rendering.test.ts**
- one called **Item.behavior.test.ts**

Your directory structure will look now ike this:



```
MY-VUE-PROJECT
  > .vscode
  > node_modules
  > public
  v src
    > assets
    v components
      v items
        v children
          TS Item.behavior.test.ts      U
          Item.component.vue            M
          TS Item.rendering.test.ts     U
        ItemsList.component.vue
      HelloWorld.vue
```

Item.rendering.test.ts

Open the file Item.rendering.test.ts and paste the following code in it:

```
// file: src/components/items/children/Item.rendering.test.ts
// directive to instruct vitest to use the jsdom environment:
// @vitest-environment jsdom
// import references to what we need from our test-utils:
import { render, screen } from '../../../test-utils'
import { describe, it, expect } from 'vitest'

// import reference to our interface
import { ItemInterface } from '../../..//models/items/Item.interface'
// import reference to your Item component:
import ItemComponent from './Item.component.vue'

describe('Item.component: rendering', () => {
  it('renders an Item text correctly', () => {
    const model: ItemInterface = {
      id: 1,
      name: 'Unit test item 1',
      selected: false
    }

    const testid = 'unit-test-appearance-1'

    // render component
    render<typeof ItemComponent>(ItemComponent, {
      props: {
        testid,
        model
      }
    })

    // get element reference by testid
    const liElement = screen.getByTestId(testid)

    // test
    expect(liElement).not.toBeNull()
    expect(liElement.innerHTML).toContain('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 html rendered by the component contains the item.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.

Note: These example are just to get you started. Later you can look at more precise ways to test what our component has rendered or even trigger events on them.

Run our unit tests from the terminal with this command:

```
npm run test
```

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

```
...

> my-vue-project@0.0.0 test
> vitest run

 RUN  v0.10.2 /Users/damianofusco/Documents/OutOfTheCage/private-my-vue3-project-2nd\
/my-vue-project

 √ src/components/items/children/Item.rendering.test.ts (1)

Test Files  1 passed (1)
     Tests  1 passed (1)
      Time  784ms (in thread 14ms, 5601.05%)

...
```

Let’s add two more tests within the same file to check that the component has the expected CSS classes.
Test to check that it has the class ”selected” when **item.selected** is **true**, and that does NOT have the css class ”selected” when **item.selected** is **false**:

```
// file: src/components/items/children/Item.rendering.test.ts

...

describe('Item.component: rendering', () => {

  ...

  it('has expected css class when selected is true', () => {
    const model: ItemInterface = {
      id: 1,
      name: 'Unit test item 2',
      selected: true /* note this is true */
    }

    const testid = 'unit-test-appearance-2'

    // render component
    render<typeof ItemComponent>(ItemComponent, {
      props: {
        testid,
        model
      }
    })

    // get element reference by testid
    const liElement = screen.getByTestId(testid)

    // test
    expect(liElement).not.toBeNull()
    // check that the element className attribute has the expected value
    expect(liElement.className).toContain('selected')
  })

  it('has expected css class when selected is false', () => {
    const model: ItemInterface = {
      id: 1,
      name: 'Unit test item 3',
      selected: false /* note this is false */
    }

    const testid = 'unit-test-appearance-3'
    // render component
    render<typeof ItemComponent>(ItemComponent, {
      props: {
        testid,
        model
      }
    })

    // get element reference by testid
    const liElement = screen.getByTestId(testid)

    // test
    expect(liElement).not.toBeNull()
    // check that the element className attribute has the expected value
    expect(liElement.className).not.toContain('selected')
  })
})
```

Item.behavior.test.ts

We can also test the behavior of our component by programmatifcally triggering the click event. Open the file Item.behavior.test.ts and paste the following code in it:

```
// file: src/components/items/children/Item.behavior.test.ts
// directive to instruct vitest to use the jsdom environment:
// @vitest-environment jsdom
// import references to what we need from our test-utils:
import { render, screen, fireEvent } from '../../../test-utils'
import { describe, it, expect } from 'vitest'

// import reference to our interface
import { ItemInterface } from '../../../models/items'
// import reference to your Item component:
import ItemComponent from './Item.component.vue'

describe('Item.component: behavior', () => {
  // test our component click event
  it('click event invokes selectItem handler as expected', () => {
    const model: ItemInterface = {
      id: 1,
      name: 'Unit test item 1',
      selected: false
    }

    const testid = 'unit-test-behavior-1'

    // render component
    const { emitted } = render<typeof ItemComponent>(ItemComponent, {
      props: {
        testid,
        model
      }
    })

    // get element reference by testid
    const liElement = screen.getByTestId(testid)

    // fire click
    fireEvent.click(liElement)

    // check that it emitted the "selected" event and passed the expected param valu\
e (the model of type ItemInterface)
    // this will be an object like {"selectItem":[[1]],"click":[[{"isTrusted":false}\
]]}
    const result = emitted() as { selectItem: any[][] }
    //console.log('emitted result', JSON.stringify(result))
    expect(result.selectItem).not.toBeUndefined
    const expectedParam = result.selectItem[0][0]
    expect(expectedParam).not.toBeUndefined
    expect(expectedParam).toEqual(model.id)
  })
})
```

Save and check the test results and make sure all pass (if you had stopped it, run `npm run test` again).

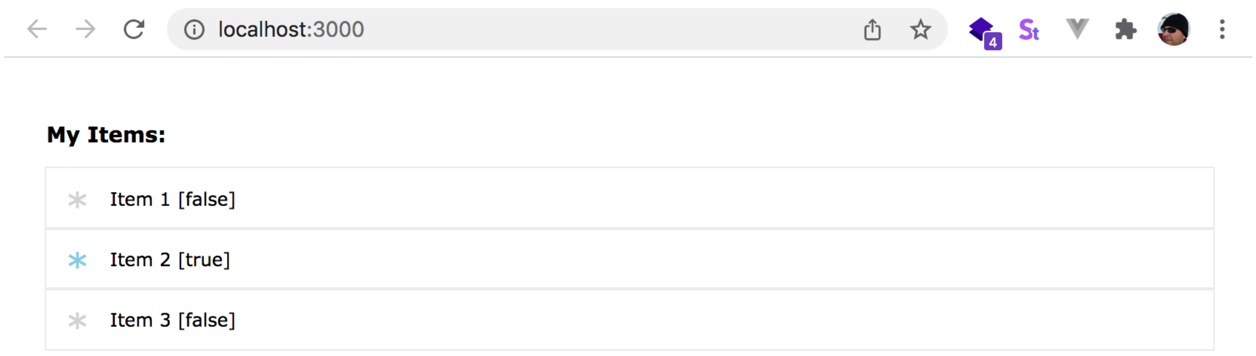
ItemsList component updates

Now we can finally modify our **ItemsList.component.vue** to consume our newly created Item component. Import a reference to ItemComponent, then replace the **<li>** element within the loop with our <ItemComponent>:

```
// file: src/components/items/ItemsList.component.vue
<script setup lang="ts">
  // import a reference to our ItemInterace
  import type { ItemInterface } from '../../models/items'
  // import a reference to the Item component:
  import ItemComponent from './children/Item.component.vue' // <-- add this line

  // expose a property called items with a default value of a blank array
  defineProps<{ items: ItemInterface[] }>()

  // define emits for custom events
  const emit = defineEmits<{
    (e: 'selectItem', id: number): any
  }>()

  // selectItem handler
  const onSelectItem = (id: number) => {  // <-- update this block
    emit('selectItem', id)
  }
</script>

<template>
  <div>
    <h3>Items:</h3>
    <ul>
      <!-- replace the <li> withit his block: -->
      <ItemComponent v-for="(item, index) in items"
        :key="item.id"
        :model="item"
        @selectItem="onSelectItem" />
    </ul>
  </div>
</template>
```

Note how we are handling the `selectItem` event using the `@selectItem="onSelectItem"` binding and just re-emit to the parent view (App.vue).

If you are not already running the app, run it. In the web browser, the list should now render similar to this (here we are showing it after we clicked on the 2nd item element and is now selected)



My Items:

Item 1 [false]

Item 2 [true]

Item 3 [false]

Additional Configuration

You probably noticed a little annoying thing where we have to use relatives paths when importing references, i.e. `import X from '../../../test-utils'`.

Before we finally conclude this chapter, let's do s a bit more configuration so we can start using a shortcut like @ and avoid relative paths, i.e. `import X from '@/test-utils'`.

First, install @types/node (needed for TypeScript type references that will use within vite.config.ts file):

```
npm install -D @types/node
```

vite.config.ts

Let's add type references to **vitest** and **vite/client** and then configure the **test** section so that we won't have to import things like describe/it/expectfrom vitest in each unit test file. Let's also add **envDir** and **resolve** sections in order to use the @ shortcut. Finally, add also a **server** section eith the **open** option so that it automatically opens the web browser to our app url when we run the app:

```
// file: vite.config.ts

/// <reference types="vitest" />
/// <reference types="vite/client" />
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import { fileURLToPath, URL } from 'url'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  server: {
    port: 3000,
    open: 'http://localhost:3000'
  },
  envDir: './src/',
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    },
  },
  test: {
    globals: true,
    environment: 'jsdom',
    exclude: [
      'node_modules'
    ]
  }
})
```

tsconfig.json updates

Add these at the end of the compilerOptions section:

```
// file: tsconfig.json

{
  "compilerOptions": {

    ...

    "baseUrl": ".",
    "paths": {
      "@/*": [
        "src/*"
      ]
    },
    "types": [
      "vue",
      "vite/client",
      "vitest/globals"
    ]
  }
  ...

}
```

You can now go back and update reltive paths imports like '../../../test-utils' to just '@/test-utils', or '../../../models' to '@/models' etc. Furthermore, you can remove the line `import { describe, it, expect } from 'vitest'` from both `Item.behavior.tests.ts` and `Item.rendering.test.ts`

Chapter 5 Recap

What We Learned

- How to write **unit tests** against a component
- How to test that components render specific DOM elements, or have specific text, or attributes like CSS classes, etc.
- How to test events on our components by programmatically triggering them with **fireEvent** (from Vue Testing Library [16])
- 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 our ItemsList.component.vue or more advanced behaviors

Based on these observations, there are a few improvements that you could make:

Improvements

- Add additional unit tests for ItemsList.component.vue as well

[16]https://vitest.dev

Chapter 6 - State Management

One of the most important part of an app that will grow large is to decided how to manage its state.

For many years in MV* frameworks like Vue or **React**[17] etc. that meant using a state manager that usually implemented the **Flux**[18] State Management pattern.

With React that usually meant using **Redux**[19], while with Vue it meant using **Vuex**[20] (or more recently **Pinia**[21]), even though nowadays there are other alternatives (inluding building your own custom state In Vue using just Vue **reactive**[22], or the **useState**[23] hooks in React, etc).

Flux offers an architectural pattern that is a slight modification of the observer-observable pattern and it is not a library or a framework.

Single source of truth:

The most important reason to implement a centralized state manager is to have a "single source of truth" for the application state/data. This simply means that our application state has only one global, centralized source. The responsibility of changing that state is only in the hand of our state manager. That means you can expect a consistent behavior in your app as the source of your data cannot be changed outside the state manager.

In this book, we'll implement our own peculiar centralized state manager that will help us deliver the goals of this book. For this, we'll create a set of interfaces and a structure that will allow use to keep our state manager organized into modules/domains. We'll leverage Vue's reactive to build a custom state manager instead of relying on frameworks like Vuex or Pinia (note: you can read about Vuex in the first edition of this book).

Note: Just remember to be open minded to different ideas, but also challenge them, and take time to explore your own ideas as well. Different patterns and code organization strategy can be implemented, and some might be better or worse than others. This is always the case when writing code in general, but even more important when writing state managements patterns.

Now let's proceed creating our store interfaces and implementations.

[17]https://reactjs.org
[18]https://facebook.github.io/flux
[19]https://redux.js.org
[20]https://vuex.vuejs.org
[21]https://pinia.vuejs.org
[22]https://vuejs.org/api/reactivity-core.html
[23]https://reactjs.org/docs/hooks-state.html

Store Architecture

One thing I learned from my past experience using React, Angular, Vue, Svelte, and more, is that there are some advantages adopting a certain flow that is closer to Flux, but does not have to follow it to the letter. We definitely won't need this in every component, as in some cases using just local state is the right thing to do. But we'll need it for global state changes on which many components within the same app will depend on.

One things that often drives the code pattern is the framework itself. Especially React has a peculiar way as a lot of plumbing has to happen within the React context itself for React to be aware of changes. Other frameworks are more flexible in this (Vue 3 reactive for example) and are less prone to drive your architectural and patterns decisisions, thus allowing more easily to decouple your state manager from the actual framework. In Vue, we have definitely much more flexibility than in React as you will see shortly.

In this chapter we'll offer a bit of an opinionated structure, but I found that is helps better understanding how the data and events flow, especially to beginners.

Let's try to implement a state manager that follow more or less this pattern:

- we will invoke an action (async) on our state manager from a component
- the state manager will perform some tasks within that action
- the state manager will commit (sync) a change to our state
- the state manager will be organized into modules (each module will represent a odmain/area of the application. I.e. items, authors, companies, projects, products, categories, etc)

One of the main core concept to keep in mind about this custom architecture for our state manager is that:

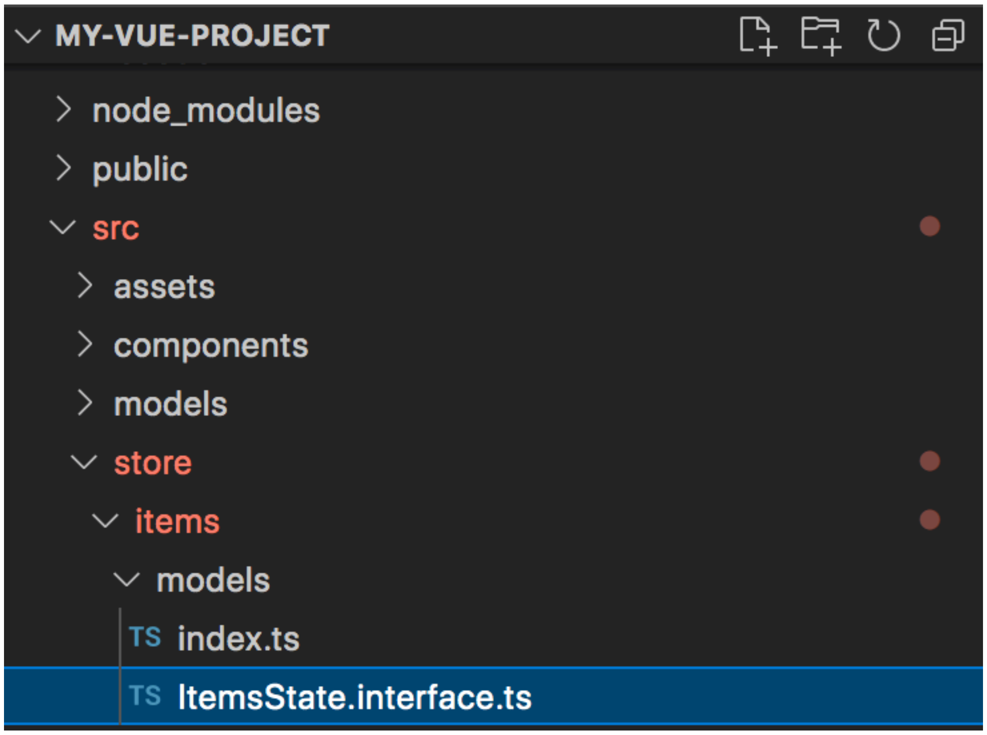
- actions will always be async functions
- mutations will always be sync functions

Items Store

Let's implement the **items** store module.

Items State interface

We will create first the interfaces for the **items** store module. Create the directory **src/store/items**. Inside here, create a directory called **models**. Inside create a **ItemsState.interface.ts** and an **index.ts** files:



Paste the following code within the **ItemsState.interface.ts** file:

```
// file: src/store/items/models/ItemsState.interface.ts
import { ItemInterface } from '@/models/items'

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

In the code above we just export an interface that represents our **Items** domain state. This will be an object with a property called **items** which will contain an array of objects of type

ItemInterface., and one called **loading** which is a boolean and will indicate if we are loading data or not (so that we can eventually display a loading indicator/component in the UI).

Within the **index.ts**[^chap6-9] file, just export our interface:

```
// file: src/store/items/models/index.ts
export * from './ItemsState.interface'
```

Items Store instance

Within the store/items directory add a file called **Items.store.ts** and paste the following code in it:

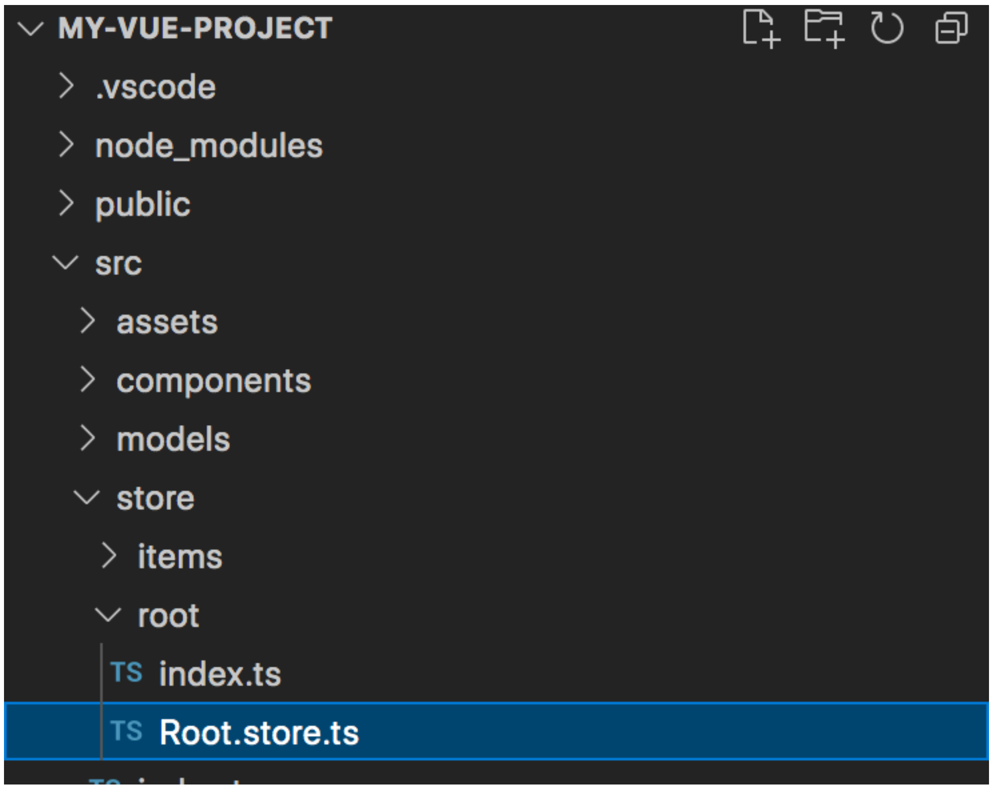
```
// file: src/store/items/Items.store.ts

// import a reference to Vue's reactive
import { reactive } from 'vue'

// import interfaces references
import { ItemsStateInterface } from './models'
// import a reference to our ItemInterface
import { ItemInterface } from '@/models'

// the items module state
const itemsState = reactive<ItemsStateInterface>({
  loading: false,
  items: []
})

// the items store actions implementation:
const actions = {
  // action that we invoke to load the items from an api:
  loadItems: async () => {
    // set loading to true and clear current data:
    itemsState.loading = true
    itemsState.items = []

    // mock some data:
    let mockData: ItemInterface[] = [{
      id: 1,
      name: 'Item 1',
      selected: false
    }, {
      id: 2,
      name: 'Item 2',
      selected: false
    }, {
      id: 3,
      name: 'Item 3',
      selected: false
    }, {
      id: 4,
      name: 'Item 4',
      selected: false
    }, {
      id: 5,
      name: 'Item 5',
      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(() => {
      // set items data and loading to false
      itemsState.items = mockData
      itemsState.loading = false
    }, 1000)
  },

  // action that we'll invoke from our component to select/unselect a specific item
  toggleItemSelected: async (id: number) => {
    const item = (itemsState.items || []).find((o) => o.id === id)
    if (item) {
      item.selected = !item.selected
    }
  }
}

// the items store getters implementation
// these will be consumed in the components
const getters = {
  get loading() {
    return itemsState.loading
  },
  get items() {
    return itemsState.items
  }
}

// infer the items store interface:
export interface ItemsStoreInterface {
  getters: typeof getters
  actions: typeof actions
}

// hook that will use to consume the items store in other code
export function useItemsStore(): ItemsStoreInterface {
  // return our store intance implementation
  return {
    getters,
    actions
  }
}
```

Still within the store/items directory, add an index.ts file to export our Items store hook:

```
// file: src/store/items/index.ts
export * from './Items.store'
```

Root Store

Create now the root store interfaces. Create the directory **src/store/root**. Inside here, create the files **Root.store.ts** and **index.ts**:



Within the **Root.store.ts** file, paste the following code:

```
// file: src/store/root/Root.store.ts

// import the items store hook
import { useItemsStore } from '../items'

// hook that returns our root store instance and will allow us to consume our app st\
ore from our components
export function useAppStore() {
  return {
    itemsStore: useItemsStore()
    // additional domain store modules will be eventually added here
  }
}

// infer store interface from useAppStore return type
export type RootStoreInterface = ReturnType<typeof useAppStore>
```

*Note how we jst export a hook called **useAppStore** that return an object with several modules (for now, we only have one called **itemsStore**). Also note how we create the RootStoreInterface by leveraging TypeScript type-inference.*

Within the **index.ts** file to just export the cdode from our Root.store.ts:

```
// file: src/store/root/index.ts
export * from './Root.store'
```

Finally, add one last **index.ts** file at the root of **src/store/** with this content:

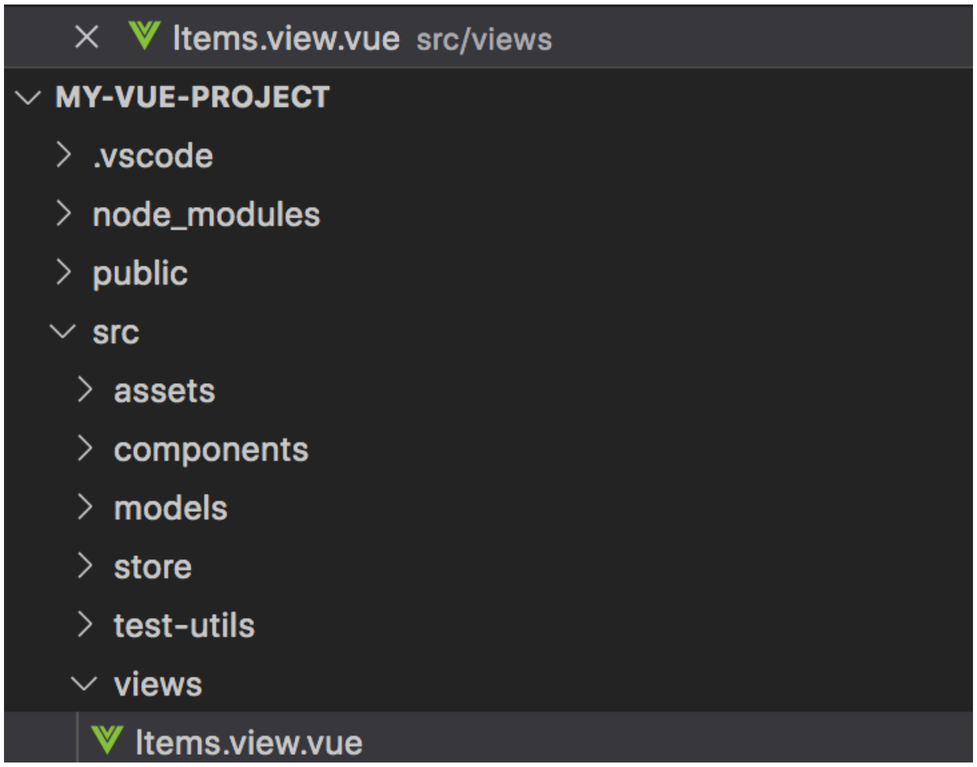
```
// file: src/store/index.ts
export * from './root'
```

Let's now go back to our components and start consuming our state.

Items.view.vue

Add a new directory called **views** under **src**.

Here we add a new higher-level component called **Items.view.vue**. Your directory structure will be like this:



```
× Items.view.vue  src/views
MY-VUE-PROJECT
  > .vscode
  > node_modules
  > public
  ∨ src
    > assets
    > components
    > models
    > store
    > test-utils
    ∨ views
      Items.view.vue
```

Note that in Vue anything is a component and we could have just called this Items.component.vue and put it under component/items. This is only for organizational purposes. We are really free *to organize the code as we see fit. In this case I also wanted to better separate what the lower components are doing and accessing the global state only in the higher-level component.*

Paste the following code within the file **Items.view.vue**:

```
// file: src/views/Items.view.vue
<template>
  <div>
    <ItemsListComponent :items="items" :loading="loading" @selectItem="onSelectItem"\
 />
  </div>
</template>

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

  import { useAppStore } from '@/store'
  import { ItemInterface } from '@/models'

  export default defineComponent({
    name: 'Home',
    components: {
      ItemsListComponent
    },
    setup() {
      // private:
      const { itemsStore } = useAppStore()
      const { actions: itemsStoreActions } = itemsStore

      // computed:
      const items = computed((): ItemInterface[] => {
        return itemsStore.getters.items
      })
      const loading = computed((): boolean => {
        return itemsStore.getters.loading
      })

      // methods:
      const onSelectItem = async (id: number) => {
              // always await store actions because they are async
        await itemsStoreActions.toggleItemSelected(id)
      }
      // lifecycle event handlers:
      onMounted(async () => {
              // always await store actions because they are async
        await itemsStoreActions.loadItems()
      })

      return {
        // computed:
        items,
        loading,
        // methods:
        onSelectItem
      }
    }
  })
</script>
```

In the code above we are basically rendering the same itemsListComponent as we did earlier in the App.vue file. However, here we are consuming the data from our items store and invoking our store actions that will mutate our data.

App.vue

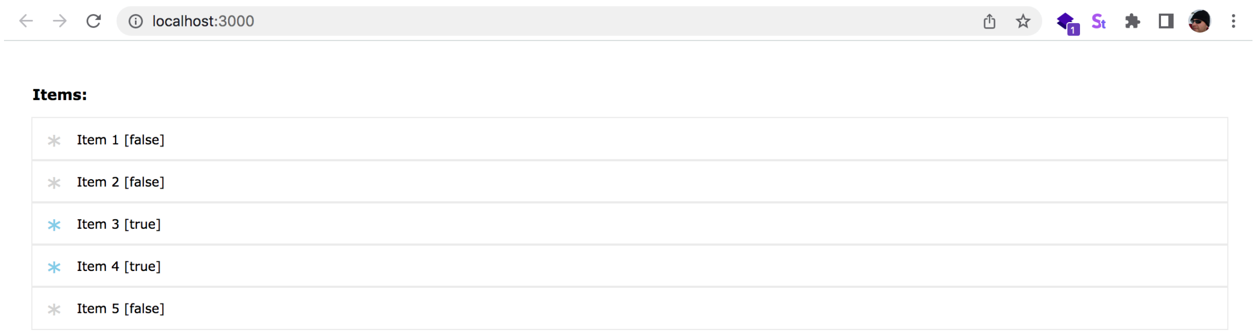
Finally, replace the entire code within App.vue so we can consume our new Items.view.vue here:

```
// file: App.vue
<script setup lang="ts">
  // import a reference to our ItemsView component
  import ItemsView from './views/Items.view.vue'
</script>

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

<style>
  .home {
    padding: 20px;
    font-family: Verdana, Geneva, Tahoma, sans-serif;
    font-size: 12px;
  }
</style>
```

Save the file.

Web Browser

The web browser should refresh and display the content similar to before. Notice that now it will take about 1 second before the items will be rendered. This is because in our **loadItems** action implementation we used a **setTimeout** with a 1 second delay to simulate a possible call to an API for example.



Items:

- Item 1 [false]
- Item 2 [false]
- Item 3 [true]
- Item 4 [true]
- Item 5 [false]

ItemsList.component.vue updates

Add a **loading** property of type **boolean** with a default value of **false**:

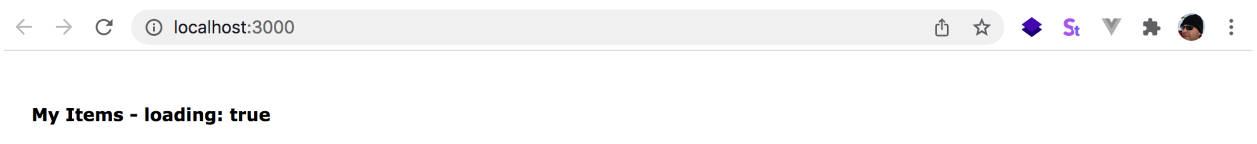
```
// file: src/components/items/ItemsList.component.vue
<script setup lang="ts">
  // import a reference to our ItemInteface
  import type { ItemInterface } from '@/models/items'
  // import a reference to the Item component:
  import ItemComponent from './children/Item.component.vue'

  // expose our properties
  defineProps<{
    items: ItemInterface[]
    loading: boolean // <-- add this property
  })()

  ...
```

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

```
// file: src/components/items/ItemsList.component.vue

...

<template>
  <div>
    <h3>Items - loading: {{loading}}:</h3> <!-- add "- loading: {{loading}}" -->

    ...
```

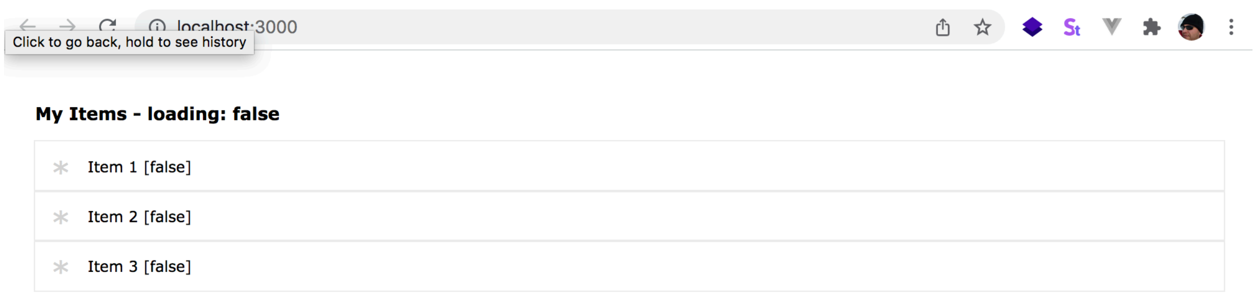
Web Browser

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**:


localhost:3000

My Items - loading: true

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


localhost:3000

Click to go back, hold to see history

My Items - loading: false

Item 1 [false]

Item 2 [false]

Item 3 [false]

Loader Component

Let's create a quick-and-dirty loader component that we can show to indicate a loading operation.

Create the directory **src/components/shared**. Within this directory create a file called **Loader.component.vue**. Within the file, paste the following code:

```
// file: src/components/shared/Loader.component.vue
<template>
  <div class="loader" v-show="show">
    Loading...
    <div class="bounceball"></div>
  </div>
</template>

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

  export default defineComponent({
    name: 'ElLoader',
    props: {
      show: {
        type: Boolean
      }
    }
  })
</script>

<style>
  .loader {
    display: inline-block;
  }
  .loader .bounceball {
    position: relative;
    width: 30px;
  }
  .loader .bounceball:before {
    position: absolute;
    content: '';
    top: 0;
    width: 30px;
    height: 30px;
    border-radius: 50%;
    background-color: #ff3e00;
    transform-origin: 50%;
    animation: bounce 500ms alternate infinite ease;
  }
  @keyframes bounce {
    0% {
      top: 60px;
      height: 10px;
      border-radius: 60px 60px 20px 20px;
      transform: scaleX(2);
    }
    25% {
      height: 60px;
      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. In later chapter we might modify this to implement a versin that uses TailwindCSS.

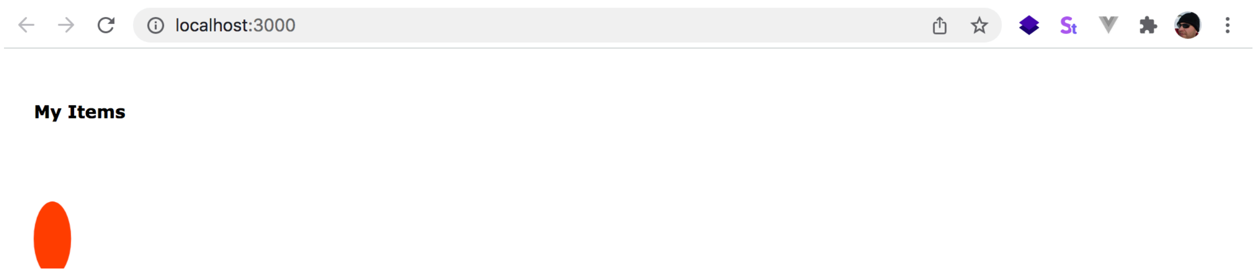
Now, lets go back into our **ItemsList.component.vue** code and import a reference to our new **Loader** component, and update our code as follow (complete code):

```
// file: src/components/items/ItemsList.component.vue
<script setup lang="ts">
  // import a reference to our ItemInteace
  import type { ItemInterface } from '@/models'
  // import a reference to the Item component:
  import ItemComponent from './children/Item.component.vue'
  // import a reference to the Laoder component
  import Loader from '@/components/shared/Loader.component.vue'

  ...
</script>
```

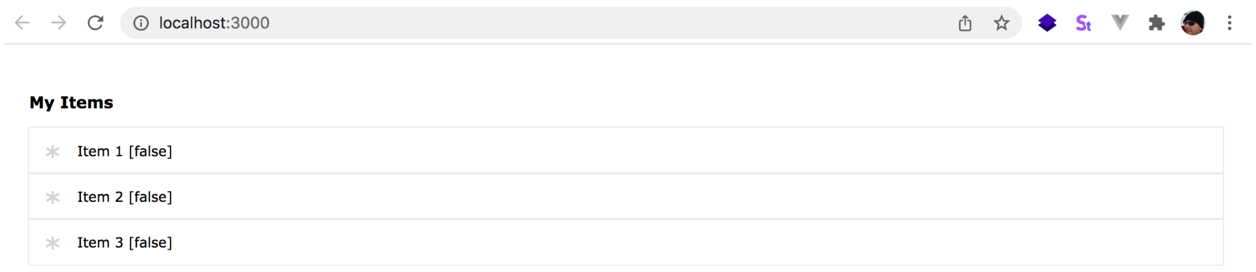
```
<template>
  <div>
    <h3>Items:</h3>
    <Loader v-show="loading" /> <!-- add this line -->
    <ul v-show="!loading"> <!-- add v-show="!loading" here -->

      ...
```

Note: we also removed the `- loading: {{loading}}` *from within the <h3> element since now we are displaying our Loader.*

Save the file and the refreshed the web page will show the loader bouncing for about 1 second before it renders the items:



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



My Items

Item 1 [false]

Item 2 [false]

Item 3 [false]

Congratulations on completing this chapter and learning how to build a state manager organized into domains to easily manage the application state in a consistent and predictable way. It's a long chapter, the concepts outlined here require a lot of code to implement, and not everyone gets through it in a straightforward fashion the first time around. In the next chapters we will try to improve this code even more so arm yourself with a lot of patience!

Chapter 6 Recap

What We Learned

- How to create a centralized state manager organized into modules, leveraging Vue writable store
- How to use our state manager to update our **Items** state
- How to create **actions** that will update our state
- How to invoke state **actions** from our components
- How to use a **loading** property on our state to provide feedback to the user about long-running processes through a **loader** (animation)
- How to create a simple and reusable **Loader** component

Observations

- We are still using hard-coded data (**mockItems** within the **actions** in the **store/items/Items.store.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

- 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 by manipulating the app state/data through our state manager (store). However, we are still ”pretending” to load data by using a **mockItems** variable with hard-coded mock data within our **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 load the data from a server-side 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-side 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 this[24]). 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.

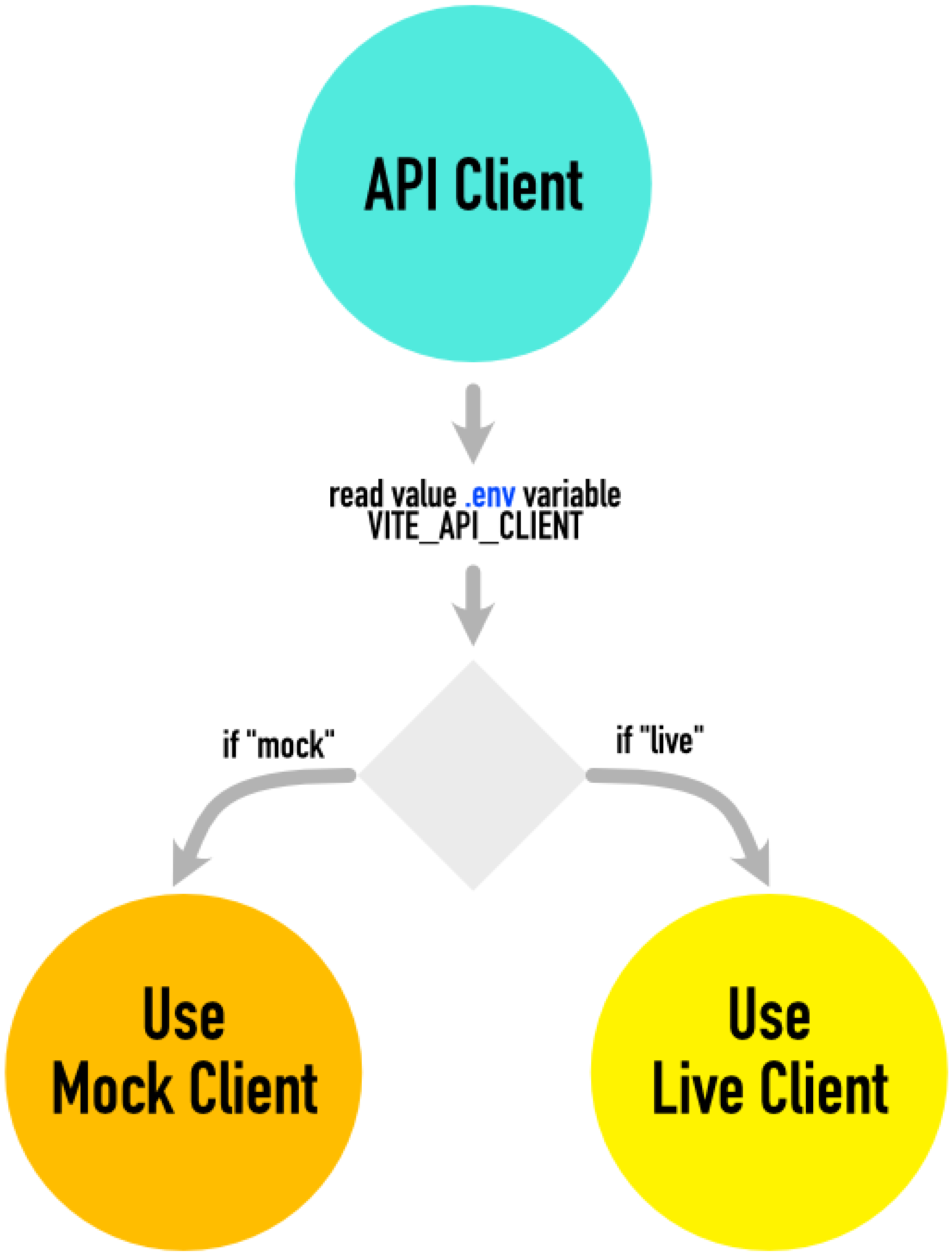
Another pattern is to create a separate API client for each area of our application. This will enable for better separation of concerns, avoid code cluttering, more easily write unit tests against each client. This is the pattern we'll be following in this book, but remember this is not the only way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

[24]JsonPlaceHolder *or* miragejs *for example*



API Client module will read the custom environment variable called VITE_API_CLIENT and there are two possible outcomes:

- when VITE_API_CLIENT is mock: it will return the Mock API Client
- when VITE_API_CLIENT is live: it will return the Live API Client

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

```
// file: src/api-client/models/ApiClient.interface.ts

import { ItemsApiClientInterface } from './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 the **Items** API client.

Items domain Api Client

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

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

- index.ts
- ItemsApiClient.interface.ts
- ItemsApiClient.model.ts
- ItemsApiClientOptions.interface.ts

Your directory structure will look like this:

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

ItemsApiClientOptions.interface.ts

In order to avoid using hard-coded strings, and to enforce type-checking at development time, we'll be using interface **ItemsApiClientOptionsInterface** for the values that indicates the API end-points consumed by the **ItemsApiClient**. Also, we'll have a **mockDelay** parameter that we can use to simulate the delay when loading data from static json files. Here is the code:

```
// file: src/api-client/models/items/ItemsApiClientOptions.interface.ts

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

/**
 * @Name ItemsApiClientOptions
 * @description
 * Interface for the Items api client options (includes endpoints used to avoid hard\
-coded strings)
 */
 export interface ItemsApiClientOptions {
  mockDelay?: number
  endpoints: ItemsApiClientEndpoints
}
```

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 **ItemsApi-Client.interface.ts**:

```
// file: src/api-client/models/items/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-vue-project** directory, and install **axios** with the command:

```
npm i -D axios --save
```

NOTE: we will improve this even more later to avoid having references to a third-party NPM package spread throughout the code. Also note, we are showing here to use a 3rd party package like axios on purpose, instead of the browser built-in fetch api, to show in later chapters how we should always try to abstract and encapsulate dependencies to avoid polluting our code.

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

```
// file: src/api-client/models/items/ItemsApiClient.model.ts

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

import { ItemsApiClientOptions, ItemsApiClientEndpoints } from './ItemsApiClientOpti\
ons.interface'
import { ItemsApiClientInterface } from './ItemsApiClient.interface'
import { ItemInterface } from '@/models'

...
```

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

```
// file: src/api-client/models/items/ItemsApiClient.model.ts

...

/**
 * @Name ItemsApiClientModel
 * @description
 * Implements the ItemsApiClientInterface interface
 */
export class ItemsApiClientModel implements ItemsApiClientInterface {
  private readonly endpoints!: ItemsApiClientEndpoints
  private readonly mockDelay: number = 0

  constructor(options: ItemsApiClientOptions) {
    this.endpoints = options.endpoints
    if (options.mockDelay) {
      this.mockDelay = options.mockDelay
    }
  }

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

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

      axios
        .get(endpoint, options)
        .then((response: AxiosResponse) => {
            if (!this.mockDelay) {
              resolve(response.data as ItemInterface[])
            } else {
              setTimeout(() => {
                resolve(response.data as ItemInterface[])
              }, this.mockDelay)
            }
        })
        .catch((error: any) => {
            console.error('ItemsApiClient: HttpClient: Get: error', error)
        })
    })
  }
}
```

api-client/items/index.ts (barrel file)

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:

```
// file: src/api-client/models/items/index.ts

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

api-client/index.ts (barrel file)

Up one more directory, add one more index.ts file with this:

```
// file: src/api-client/models/index.ts

export * from './ApiClient.interface'
export * from './items'
```

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/api-client** directory, create two new sub-directories called:

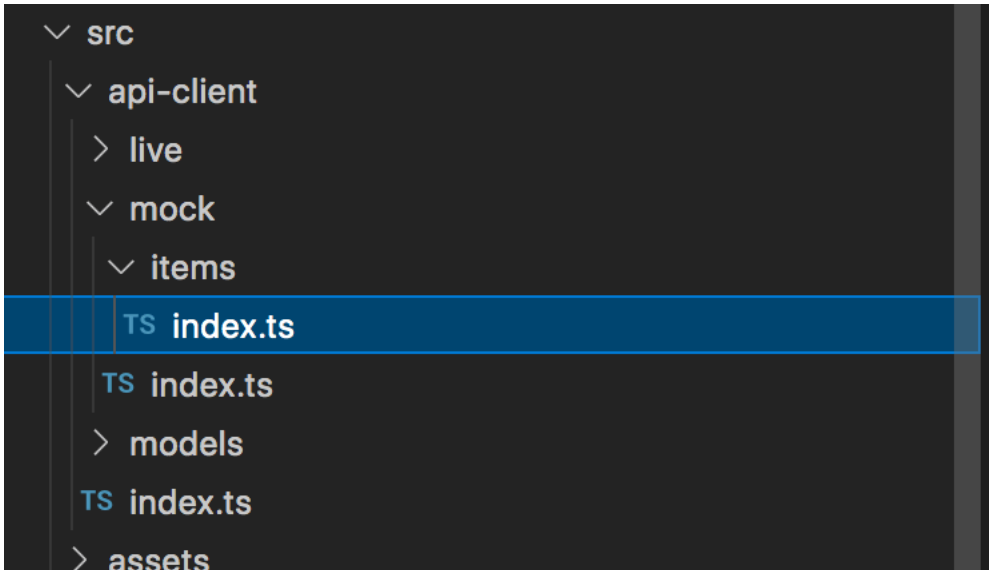
- **mock** (this will contain our mock implementations to return static **json** data)
- **live** (this will contain the implementation that call the real API end-points)

We'll be writing a mock implementation of our **ApiClientInterface** and its child **ItemsApiClientInterface**. We'll be also instantiating either the mock or live api-client based on config..

Mock Api Client

Items domain mock API instance

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:

```
// file: src/api-client/mock/items/index.ts

import {
  ItemsApiClientOptions,
  ItemsApiClientInterface,
  ItemsApiClientModel
} from '../../models/items'

const options: ItemsApiClientOptions = {
  endpoints: {
    fetchItems: '/static/mock-data/items/items.json'
  },
  mockDelay: 1000
}

// instantiate the ItemsApiClient pointing at the url that returns static json mock \
data
const itemsApiClient: ItemsApiClientInterface = new ItemsApiClientModel(options)

// export our instance
export {
  itemsApiClient
}
```

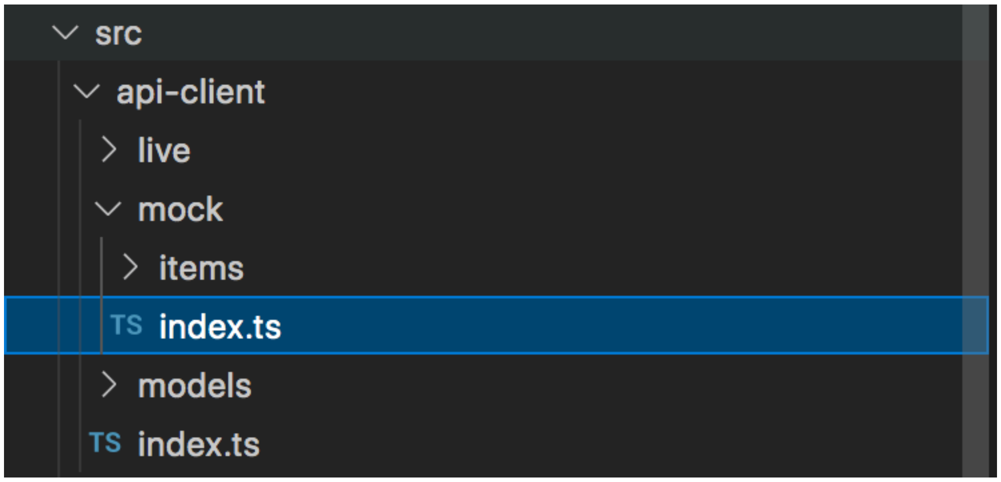
Here we import all our interfaces and models, then we instantiate a variable called **options** of type **ItemsApiClientOptions** that holds the API end-points values and the mockDelay option. In this case, since this is the mock implementation, for fetchItems we will point to some static **json** file 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 **options** instance into the constructor (as you can see, later in our live implementation we'll pass an instance of ItemsApiClientOptions that contains the paths/urls to the real end-points)

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

Mock API instance

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:



```
src
  api-client
    live
    mock
      items
      index.ts
    models
    index.ts
```

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

```
// file: src/api-client/mock/index.ts

import { ApiClientInterface } from '../models/ApiClient.interface'
import { itemsApiClient } from './items'

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

// export our instance
```

```
export {
  apiMockClient
}
```

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

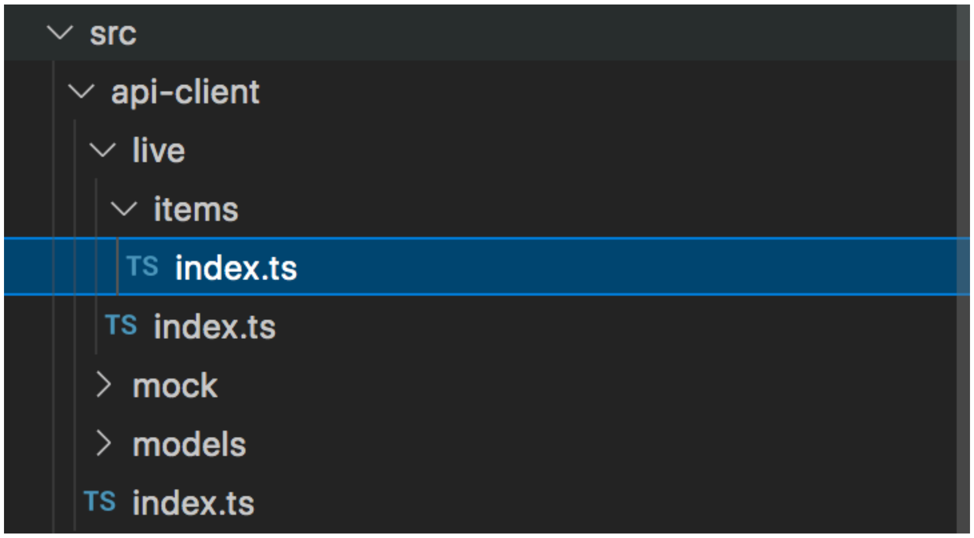
Here we import our **ApiClientInterface** interface, and our mock instance of **ItemsApiClient**. We then create an instance of our **ApiClientInterface** that is called **apiMockClient** because it will use the mock implementation of the **ItemsApiClient**.

Live Api Client

Items domain live API instance

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.

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
        index.ts
      index.ts
    mock
    models
    index.ts
```

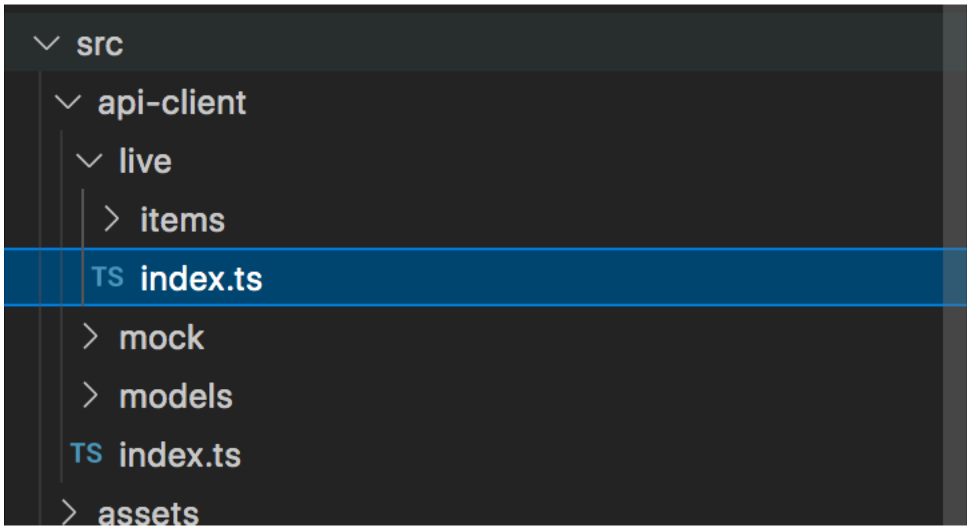
Inside the **src/api-client/live/items/index.ts** file, paste the following code:

```
// file: src/api-client/live/items/index.ts

import {
  ItemsApiClientOptions,
  ItemsApiClientInterface,
  ItemsApiClientModel
} from '../../models/items'

const options: ItemsApiClientOptions = {
  endpoints: {
    // this should be pointing to the live API end-point
    fetchItems: '/path/to/your/real/api/end-point'
  }
}

// instantiate the ItemsApiClient pointing at the url that returns live data
const itemsApiClient: ItemsApiClientInterface = new ItemsApiClientModel(options)

// export our instance
export {
  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/end-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.*

Live API instance

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:

```
// file: src/api-client/live/index.ts

import { ApiClientInterface } from '../models'
// import module instances
import { itemsApiClient } from './items'

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

// export our instance
export {
  apiLiveClient
}
```

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.

In a bit we'll be adding one final index.ts at the root of src/api-client that will act as our API client "provider". This will return either the mock or the live instance based on an environemnt variable. So let's first setup some things to work with environment variables.

Environment Variables

Since Vite uses **dotenv**[25] to load environment variables, we'll have to create two **.env** files[26] at root of your src directory:

```
.env.dev          # loaded when mode is dev for local development
.env.production              # loaded when mode is production
```

Inside the .env.mock put the following:

```
# file src/.env.dev

VITE_API_CLIENT=mock
```

Inside the .env.production put the following:

```
# file src/.env.production

VITE_API_CLIENT=live
```

You might have to add type declarations for the **import.meta.env** types within the **src/vite-env.d.ts** file[27] (note: this file might already exists and just be called **env.d.ts**):

```
// file: src/vite-env.d.ts (or src/env.d.ts)

/// <reference types="vue" />
/// <reference types="vite/client" />

// types for Vite env variables:
// (reference: https://vitejs.dev/guide/env-and-mode.html#intellisense-for-typescrip\
t)
interface ImportMetaEnv {
  readonly VITE_API_CLIENT: string
  // more env variables...
}

interface ImportMeta {
  readonly env: ImportMetaEnv
}
```

[25]https://github.com/motdotla/dotenv
[26]https://vitejs.dev/guide/env-and-mode.html#production-replacement
[27]https://vitejs.dev/guide/env-and-mode.html#intellisense-for-typescript

NOTE: Only variables prefixed with `VITE_` are exposed to the Vite-processed code.[28]

We'll be now able to access the value of our environment variables in TypeScript with **import.meta.env** (i.e. import.meta.dev.VITE_API_CLIENT). Before we can do this, we need to do one final change in our **package.json scripts** configurationso it will correctly set the expected environment variables when running locally for development with **npm start**, or when building for production with **npm run build**. The current content of your script section should be like this:

```
file: package.json
...
  "scripts": {
    "start": "npm run dev",
    "dev": "vite --mode mock", // here add --mode mock
    "build": "vue-tsc --noEmit && vite build --mode production", // here add --mode \
production
    ...
  },
...
```

Change the **dev** command to:

```
"dev": "vite --mode mock",
```

Change the **build** command to:

```
"build": "vue-tsc --noEmit && vite 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": "vue-tsc --noEmit && vite build --mode mock"
```

*Note: when running the app, if you make a change to the –mode value in the package.json, or the values within the .env files, you'll have to stop it with CTRL+C and restart with **npm start** for changes to take into effect.*

One last thing: we put our .env files within the src/ directory for now. To make sure Vite is aware of where they are, open the **vite.config.ts** file and make sure the **envDir** option is configured with the following value (we added this at the end of Chapter 5, but is a good idea to verify that is there):

[28] import.meta.env

```
// file: vite.config.ts
/// <reference types="vitest" />
/// <reference types="vite/client" />

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import { fileURLToPath, URL } from 'url'

// https://vitejs.dev/config/
export default defineConfig({

  ...

  envDir: './src/' // <-- make sure this is there

  ...

})
```

To test that the configuration is working, temporarily modify the **App.vue** code to ourput all the content of the import.meta.env like this:

```
// file: src/App.vue

<script setup lang="ts">
  // import a reference to our ItemsView component
  import ItemsView from './views/Items.view.vue'
  // store a reference to import meta env into a var
  const metaEnv = import.meta.env
</script>

<template>
  <div class="home">
    [{{metaEnv}}] <!-- add this to output the current content of import.meta.env -->
    <ItemsView />
  </div>
</template>

...
```

Stop the app with CTRL+C and run it again with npm start. Verify that in the browser our App.vue renders something like this at the top:

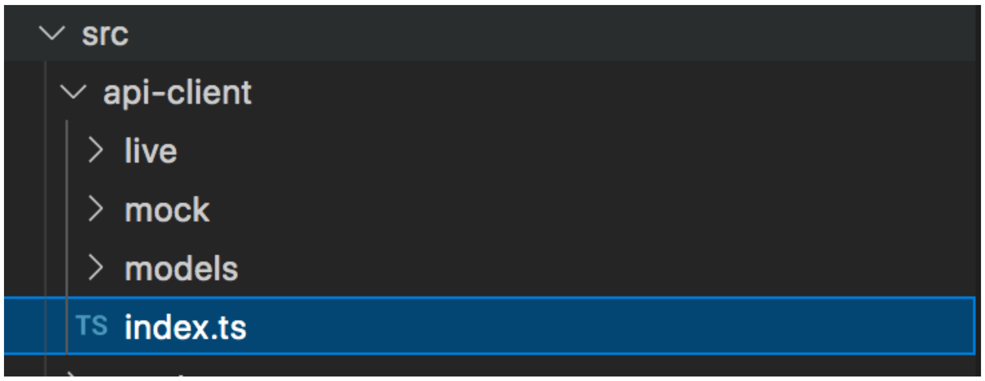
```
[{ "BASE_URL": "/", "MODE": "mock", "DEV": true, "PROD": false, ...
```

As you can see, our VITE_API_CLIENT environment variable contains the correct value "mock" and we are able to read this in our views or other client-side code.

Now remove the code we just added to App.vue and let's proceed creating our Api Client Provider.

Api Client Provider

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). Create an the file at the root of **src/api-client**:



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

```
// file: src/api-client/index.ts

import { ApiClientInterface } from './models'
import { apiMockClient } from './mock'
import { apiLiveClient } from './live'

...
```

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

```
// file: src/api-client/index.ts

...

let env: string = 'mock'
// Note: Vite uses import.meta.env (reference: https://vitejs.dev/guide/env-and-mode\
.html)
// optional: you can console.log the content of import.meta.env to inspect its value\
s like this: console.log('import.meta.env', JSON.stringify(import.meta.env))
if (import.meta.env && import.meta.env.VITE_API_CLIENT) {
  env = import.meta.env.VITE_API_CLIENT.trim()
}
// return either the live or the mock client
let apiClient: ApiClientInterface
if (env === 'live') {
  apiClient = apiLiveClient
} else {
  // default is always apiMockClient
  apiClient = apiMockClient
}

export {
  apiClient
}
```

Now let's proceed to update our store to consume the data from our newly created Api Client.

Store Instance updates

Back into our **src/store/items/Items.store.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 **VITE_API_CLIENT** environment variable we created earlier):

```
// src/store/items/Items.store.ts
...
// import a reference to our apiClient instance
import { apiClient } from '@/api-client'
...
```

Then, within the **loadItems** action, remove the hard-coded **mockItems** variable and its data. Then remove the **setTimeout** lines with the call to **commit(setItems(mockItems))**.

Replace the loadItems code with a call to **apiClient.items.fetchItems** and this time dispatch/-commit **setItems** passing it the **data** returned by our **fetchItems**:

```
// src/store/items/Items.store.ts
...

// the items store actions implementation:
const actions = {
  // action that we invoke to load the items from an api:
  loadItems: async () => {
    // set loading to true and clear current data:
    itemsState.loading = true
    itemsState.items = []

                // begin: remove code
~~    // mock some data:~~
~~    let mockData: ItemInterface[] = [{~~
~~      id: 1,~~
~~      name: 'Item 1',~~
~~      selected: false~~
~~    }, {~~
~~      id: 2,~~
~~      name: 'Item 2',~~
~~      selected: false~~
~~    }, {~~
~~      id: 3,~~
~~      name: 'Item 3',~~
~~      selected: false~~
~~    }, {~~
~~      id: 4,~~
~~      name: 'Item 4',~~
~~      selected: false~~
~~    }, {~~
~~      id: 5,~~
```

```
      ~~name: 'Item 5',~~
      ~~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(() => {~~
      ~~// set items data and loading to false~~
      ~~itemsState.items = mockData~~
      ~~itemsState.loading = false~~
      ~~console.log('itemsStore: loadItems: state updated')~~
    ~~}, 1000)~~
    // end: remove code

    // begin: add code
    // invoke our API cient fetchItems to load the data from an API end-point
    const data = await apiClient.items.fetchItems()
    itemsState.items = data
    itemsState.loading = false
    // end: add code
  },
  ...
```

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

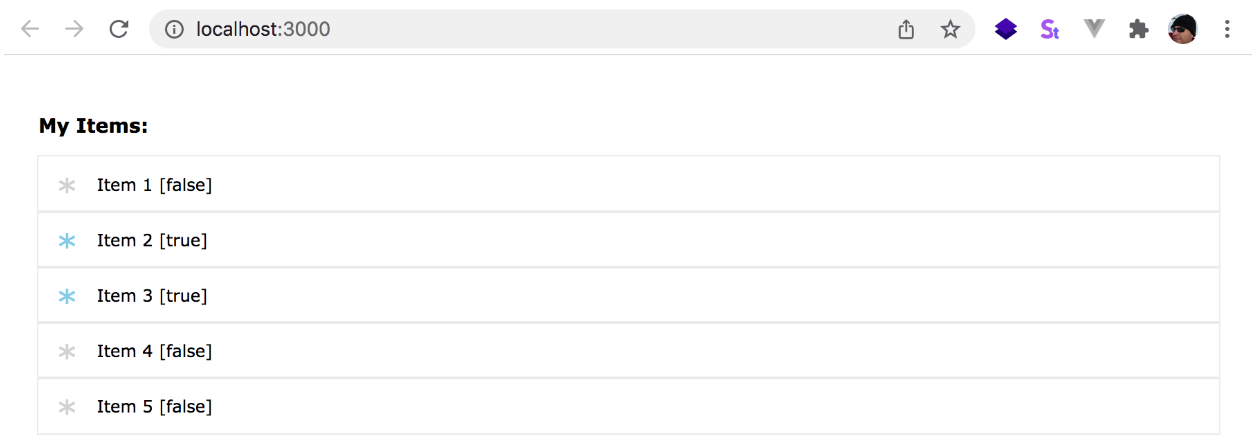
If you remember, earlier during our Mock Api Client implementation we set the **urls fetchItems** end-point path to be **/static/mock-data/items/items.json**.

We need to create a directory called **static** under our **public** folder, because that is what app considers our root directory to be when running the application. Within the static directory create a sub-directory called **mock-data**, within mock-data add one more sub-directory called **items**. Here create a file called **items.json**.

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

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

[{
  "id": 1,
  "name": "Item 1",
  "selected": false
}, {
  "id": 2,
  "name": "Item 2",
  "selected": false
}, {
  "id": 3,
  "name": "Item 3",
  "selected": false
}, {
  "id": 4,
  "name": "Item 4",
  "selected": false
}, {
  "id": 5,
  "name": "Item 5",
  "selected": false
}]
```

Make sure there are no errors in the terminal. If needed stop it with **CTRL-C** and run again with **npm start**. The browser should display a loader, then render our items list as before, but this time should display 5 items (because the data now is loaded through our Api client from the file public/static/mock-data/items/items.json):



My Items:

Item 1 [false]

Item 2 [true]

Item 3 [true]

Item 4 [false]

Item 5 [false]

Notice how powerful is this pattern we just implemented as it allows us to easily build our front-end components in isolation without a real API, and later everything we'll just work with a live API client that returns the same data structure as our static json data.

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 like **Miragejs** or **JSONPlaceHolder**[29], 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. Some of these alternatives require running an additional server app that will serve your mocked API.

I opted to show you how you can do this using static **.json** files that are located in the same project under **public/static/mock-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 deleted 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 reloaded as in its original state).

Additionally, this gives you the flexibility to use either: static JSON files, or maybe for the url end points use something like Miragejs etc for some of the API clients.

You can research alternatives as you see fit and make the decision you feel works better for you, but remember you are not confined to one way or another if you keep following the patterns I am showing you in this book. Indeed, let me finish by adding a few more instructions on how to use for example an NPM package called **json-server**.

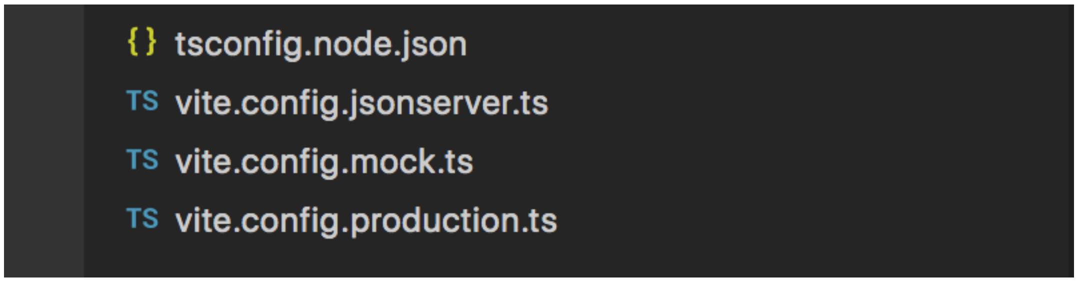
Alternative: using json-server

Let's start by install json-server:

```
npm i -D json-server
```

Now let's rename the **vite.config.ts** file to **vite.config.jsonserver.ts**. Make 2 more copies of this file and name one **vite.config.mock.ts** and the other **vite.config.production.ts**:

[29] JsonPlaceHolder *or* miragejs *for example*


{} tsconfig.node.json
TS vite.config.jsonserver.ts
TS vite.config.mock.ts
TS vite.config.production.ts

The content for **vite.config.mock.ts** will be:

```
/// <reference types="vitest" />
/// <reference types="vite/client" />

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import { fileURLToPath, URL } from 'url'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    vue()
  ],
  server: {
    port: 3000,
    origin: 'http://localhost:3000',
    open: 'http://localhost:3000'
  },
  envDir: './src/',
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    },
  },
  test: {
    globals: true,
    environment: 'jsdom',
    exclude: [
      'node_modules'
    ]
  }
})
```

The content for **vite.config.production.ts** will be:

```
/// <reference types="vitest" />
/// <reference types="vite/client" />

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import { fileURLToPath, URL } from 'url'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    vue()
  ],
  envDir: './src/',
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    },
  },
  test: {
    globals: true,
    environment: 'jsdom',
    exclude: [
      'node_modules'
    ]
  }
})
```

The content for **vite.config.jsonserver.ts** will be:

```
/// <reference types="vitest" />
/// <reference types="vite/client" />

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import { fileURLToPath, URL } from 'url'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  server: {
    port: 3000,
    origin: 'http://localhost:3000',
    open: 'http://localhost:3000',
    proxy: {
      '/jsonserver': {
        target: 'http://localhost:3111',
        changeOrigin: true,
        secure: false,
        ws: false,
        rewrite: (path) => path.replace(/^\/jsonserver/, '')
      }
    }
  },
  envDir: './src/',
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    },
  },
  test: {
    globals: true,
    environment: 'jsdom',
    exclude: [
      'node_modules'
    ]
  }
})
```

Note how the main difference in the vite.config.jsonserver.ts is the addition of the proxy section:

```
...

    proxy: {
      '/jsonserver': {
        target: 'http://localhost:3111',
        changeOrigin: true,
        secure: false,
        ws: false,
        rewrite: (path) => path.replace(/^\/jsonserver/, '')
      }
    }

...
```

This is telling Vite to proxy all the requests for endpoints that start with `/jsonserver` to the url `http://localhost:3111` (this is where the json-server API will run from)

Modify **tsconfig.node.json** include section like this (if you don't have this file, please create it):

```json
{
  "compilerOptions": {
    "composite": true,
    "module": "esnext",
    "moduleResolution": "node"
  },
  "include": [
    "vite.config.jsonserver.ts",
    "vite.config.mock.ts",
    "vite.config.production.ts"
  ]
}
```

Modify tsconfig.json to reference also tsconfig.node.json:

```json
{
  ...

  "references": [{ "path": "./tsconfig.node.json" }]
}
```

Modify the script section of the package.json file to have two additional commands:

- with-jsonserver (we'll use this to run the app using the vite.config.jsonserver.ts)
- json-server-api (with this we'll start json-server on port 3111)

Also update the current command to explicitely set which Vite config file to use with –config:

```
   "scripts": {
     "start": "npm run dev",
     "dev": "vite --config vite.config.mock.ts --mode mock",
     "build": "vue-tsc --noEmit && vite build --config vite.config.production.ts --m\
ode production",
     "build-mock": "vue-tsc --noEmit && vite build --config vite.config.mock.ts --mo\
de mock",
     "preview": "vite preview",
     "with-jsonserver": "vite --config vite.config.jsonserver.ts --mode mock",
     "json-server-api": "json-server --port 3111 --watch json-server/db.json",

     ...
```

Create json-server data by adding a new file at `json-server/db.json` with this code:

```
// file: json-server/db.json
{

  "items": [
    {
      "id": 1,
      "name": "Item 1 from json-server",
      "selected": false
    },
    {
      "id": 2,
      "name": "Item 2 from json-server",
      "selected": false
    },
    {
      "id": 3,
      "name": "Item 3 from json-server",
      "selected": false
    },
    {
      "id": 4,
      "name": "Item 4 from json-server",
      "selected": false
    },
    {
      "id": 5,
      "name": "Item 5 from json-server",
      "selected": false
    }
  ]

}
```

Now to finally test it, temporarily modify the file `src/api-client/mock/items/index.ts` to use `/jsonserver/items` for the fetchItems url:

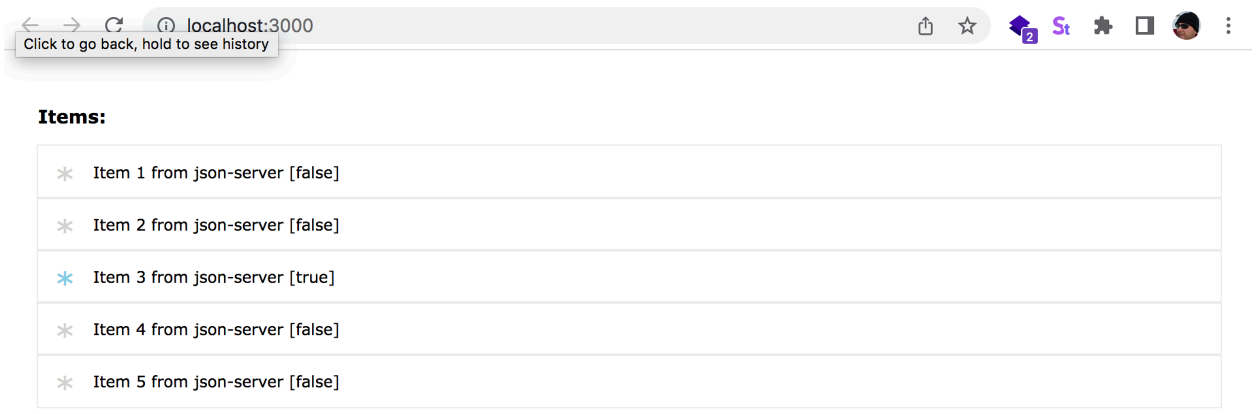
```
// file: src/api-client/mock/items/index.ts

...

const options: ItemsApiClientOptions = {
  endpoints: {
    //fetchItems: '/static/mock-data/items/items.json' // <-- comment this line out
    fetchItems: '/jsonserver/items' // <-- add this line
  },
  mockDelay: 1000
}

...
```

Note: we'll drive the API urls end-points through a much better configuration strategy in the next chapters.

Now stop the app, and this time open 2 terminal windows:

- in terminal one, execute **npm run json-server-api** (this will run json-server API on port 3111)
- in therminal two, execute **npm run with-jsonserver** (this will start our app but tell Vite to use the vite.config.jsonserver.ts which contains our proxy configuration)

The browser should now display:



Items:

Item 1 from json-server [false]

Item 2 from json-server [false]

Item 3 from json-server [true]

Item 4 from json-server [false]

Item 5 from json-server [false]

NOTE: Do not forget to revert your change for the URL end-point within the file `src/api-client/mock/items/`
Later, when we introduce the apllication configuration in the next chapters, we'll drive the end-points from configuration and will not have to modify eny code to test different environments.

Chapter 7 Recap

What We Learned

- How to implement an **apiClient** that automatically can serve either mock or real data depending on environment variables configuration
- 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 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 o 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.
- We had to modify code for the URL end-point of the mock api client for items to test with json-server.

Based on these observations, there are a few improvements that will be making into the next 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.
- Implement a better application configuration pattern to drive the app for different environemtns and avoid to have to modify code to test different scenarios.

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.

Imagine if we had built a huge code base with many components and state modules and now we wanted to using something like **Fetch Api**[30] or another library insteaf of **axios**. We would have to replace all the calls that use **axios** in our entire code base.

What we need to do is abstract the http client 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, but the most straigh-forward way is to create a class that wraps our calls done with **axios** in one place. We'll call this the **HttpClient** class and here we'll implement code that allow us to perform http requests using axios for now. If later we have to switch to a different NPM library or use the Fetch API etc, we'll jsut need to update the code without our HttpClient. Ass long as we do not change the signature of our HttpClient methods, everything should still work as before without having to change the code that consumes our HttpClient throughout our application.

Here I will show you how this pattern works by offering both an implementation that uses axios and one that uses the browser Fetch API. Then in the net chapter will drive which client we use through the app configuration.

HttpClient Interfaces and Models

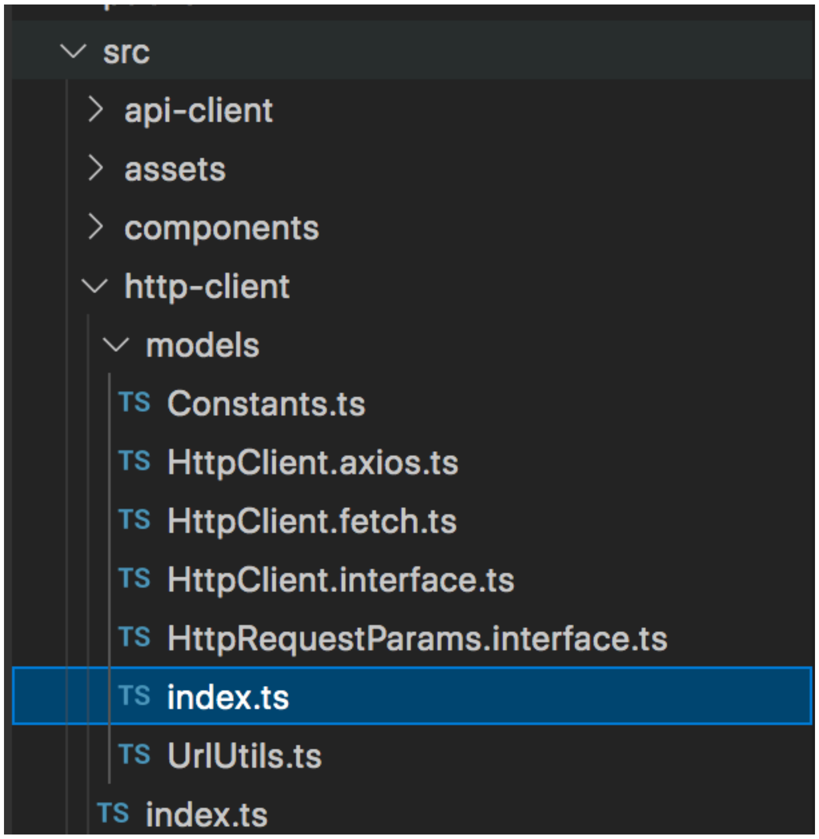
Create the directory **src/http-client/models**. Within this directory, create the following files

- Constants.ts
- HttpRequestParams.interface.ts
- UrlUtils.ts
- HttpClient.interface.ts
- HttpClient.axios.ts

[30] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

- HttpClient.fetch.ts
- index.ts

Your directory structure will look like this:



```
src
  api-client
  assets
  components
  http-client
    models
      Constants.ts
      HttpClient.axios.ts
      HttpClient.fetch.ts
      HttpClient.interface.ts
      HttpRequestParams.interface.ts
      index.ts
      UrlUtils.ts
    index.ts
```

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

Constants.ts

Within the Constants.ts file, we'll add an enum representing the type of http request we want our HttpClient to execute. For now we just add the 4 most common http verbs: get/post/put/delete:

```
// file: src/http-client/models/HttpRequestType.ts

/**
 * @name HttpRequestType
 * @description
 * The type of http request we need to execute in our HttpClient request method
 */
export const enum HttpRequestType {
  get,
  post,
  put,
  delete,
  patch
}

...
```

We'll also add two readonly objects to avoid using hard-coded strings later:

```
// file: src/http-client/models/HttpRequestType.ts

...

// http content types
export const HttpContentTypes = Object.freeze({
  applicationJson: 'application/json',
  formUrlEncoded: 'application/x-www-form-urlencoded;charset=UTF-8'
})

// constant for http request methods names
export const HttpRequestMethods = Object.freeze({
  get: 'GET',
  post: 'POST',
  put: 'PUT',
  delete: 'DELETE',
  patch: 'PATCH'
})
```

HttpRequestParams.interface.ts

The **HttpRequestParamsInterface** will allow us to pass parameters to the HttpClient request method. These are things like the type of request (GET/POST/etc), the API **endpoint**, an optional **payload** (if POST or PUT), and a flag that indicates if the request must include an authentication token.

```
// file: src/http-client/models/HttpRequestParams.interface.ts

import { HttpRequestType } from './Constants'

/**
 * @name HttpRequestParamsInterface
 * @description
 * Interface represents an object we'll use to pass arguments into our HttpClient re\
quest method.
 * This allow us to specify the type of request we want to execute, the end-point ur\
l,
 * if the request should include an authentication token, and an optional payload (i\
f POST or PUT for example)
 */
export interface HttpRequestParamsInterface<P = void> {
  requestType: HttpRequestType
  endpoint: string
  requiresToken: boolean
  headers?: { [key: string]: string }
  payload?: P
  mockDelay?: number
}
```

*NOTE: With **P ** we are trying to enfore more type-checking with TypeScript when we'll consume this, at the same time we need to add as **P = void** as this is not always required.*

UrlUtils.ts

This mainly contains an helper to dynamically build urls with parameters:

```
// file: src/http-client/models/UrlUtils.ts
export interface UrlUtilsInterface {
  getFullUrlWithParams(baseUrl: string, params: { [key: string]: number | string }):\
 string
}

export const UrlUtils: UrlUtilsInterface = {
  /**
   * @name getFullUrlWithParams
   * @description Returns the full formatted url for an API end-point
   * by replacing parameters place holder with the actual values.
   * @param baseUrl The base API end-point witht he params placeholders like {projec\
tId}
   * @param params The request params object with the key/value entries for each par\
ameter
   * @returns The fully formatted API end-point url with the actual parameter values
   */
  getFullUrlWithParams: (baseUrl: string, params: { [key: string]: number | string }\
): string => {
    const keys: string[] = Object.keys(params || {})
    if ((baseUrl || '').indexOf('[') === -1 || keys.length === 0) {
      return baseUrl
    }
    let fullUrl = baseUrl
    keys.forEach((key) => {
      fullUrl = fullUrl.replace(`[${key}]`, (params[key] || 'null').toString())
    })
    return fullUrl
  }
}
```

Note: you could alternatively implement getFullUrlWithParams using the JavaScript built-in Url.

HttpClient.interface.ts

The **HttpClientInterface** is the interface that defines the methods that the **HttpClient** will have to implement. There will be only one method called **request** which can execute different types of http request based on the parameters argument provided, and returns a **Promise**[31] with the results (if any):

[31] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

```
// files: src/http-client/models/HttpClient.interface.ts

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

/**
 * @name HttpClientConfigInterface
 * @description
 * We'll drive the HttpClient from configuration in later chapters.
 */
export interface HttpClientConfigInterface {
  tokenKey: string
  clientType: string
}

/**
 * @name HttpClientInterface
 * @description
 * Represents our HttpClient.
 */
 export interface HttpClientInterface {
  /**
   * @name request
   * @description
   * A method that executes different types of http requests (i.e. GET/POST/etc)
   * based on the parameters argument.
   * The type R specify the type of the result returned
   * The type P specify the type of payload if any
   * @returns A Promise<R> as the implementation of this method will be async.
   */
  request<R, P = void>(parameters: HttpRequestParamsInterface<P>): Promise<R>
}
```

*Note: in the code above the request method can take 2 generic types. The first one, **R**, define the type of the result/data returned. The second, **P**, is optional and defines the type of the **payload** (if any) passed with the **parameters** argument.*

HttpClient.axios.ts

The **HttpClientAxios** is the class that implements our **HttpClientInterface** using axios. Since the code here is longer, let me split in multiple parts.

First the import section:

```
// file: src/http-client/models/HttpClient.axios.ts

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

import { HttpRequestParamsInterface } from './HttpRequestParams.interface'
import { HttpClientInterface, HttpClientConfigInterface } from './HttpClient.interfa\
ce'
import { HttpRequestType, HttpContentTypes } from './Constants'
import { UrlUtils } from './UrlUtils'

/**
 * @name HttpClientAxios
 * @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 devel\
oped or other reasons
 */
export class HttpClientAxios implements HttpClientInterface {

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

  /**
   * @name request
   * @description
   * A method that executes different types of http requests (i.e. GET/POST/etc)
   * based on the parameters argument.
   * The type R specify the type of the result returned
   * The type P specify the type of payload if any
   * @returns A Promise<R> as the implementation of this method will be async.
   */
  async request<R, P>(parameters: HttpRequestParamsInterface<P>): Promise<R> {
    // use destructuring to extract our parameters into local variables
    const { requestType, endpoint, requiresToken, payload, headers, mockDelay } = pa\
rameters

    // use helper to build the fullUrl with request parameters derived from the payl\
oad
    const fullUrl = UrlUtils.getFullUrlWithParams(endpoint, payload as any)
    console.log('HttpClientAxios: fullUrl: ', fullUrl, payload)

    // set axios options
    const options: AxiosRequestConfig = {
      headers: {},
      maxRedirects: 0
    }

    if (headers) {
      options.headers = {
        //...options.headers,
        ...headers
      }
    }

    // set headers Authorization
    if (requiresToken && options.headers) {
      options.withCredentials = true
      // optional: you could add coded here to set the Authorization header with a b\
earer token
      // options.headers.Authorization = `bearer ${ JwtHelpers.getJwtToken() }`
    }

    let result!: R

    try {
      switch(requestType) {

        // TODO: implement a case statement for each request type

        default: {
          console.warn('HttpClientAxios: invalid requestType argument or request typ\
e not implemented')
        }
      }
    } catch (e) {
      console.error('HttpClientAxios: exception', e)
      throw Error('HttpClientAxios: exception')
    }

    if ((mockDelay || 0) > 0) {
      return new Promise<R>((resolve) => {
        setTimeout(() => {
          resolve(result)
        }, mockDelay)
      })
    }

    return result
  }
}
```

Note how we added a constructor placeholder, but not doing anything with it yet. Later, you could add things like request interceptors within the contructor so you can log or capture errors in one place. One more thing to notice is that we are using a try/catch block and just log the error in the console, but we are not gracefully rejecting the promise return by our request method. You are welcome to enhance and improve this code as you see fit based on your sepcific requirements.

The implementation of the request method starts by destructuring our request **parameters**, creates the fullUrl, setting some axios **options**, optionally setting an **Authorization** header (commented out for now, but to show how you can do that if you need it), and a **switch** statement that will execute the type of request we want. Let's implement now the different type of requests within each case block of our switch statement.

The **get** implementation:

```
// file: src/http-client/models/HttpClient.axios.ts

...

// executes a get request:
case HttpRequestType.get: {
  const response = await axios.get(fullUrl, options)
  result = response?.data as R
  break
}

...
```

The **post** implementation:

```
// file: src/http-client/models/HttpClient.axios.ts

...

// executes a post request:
case HttpRequestType.post: {
  const response = await axios.post(fullUrl, payload, options)
  result = response?.data as R
  break
}

...
```

The **put** implementation:

```
// file: src/http-client/models/HttpClient.axios.ts

...

// executes a put request:
case HttpRequestType.put: {
  const response = await axios.put(fullUrl, payload, options)
  result = response?.data as R
  break
}

...
```

The **delete** implementation:

```
// file: src/http-client/models/HttpClient.axios.ts

...

// executes a delete request:
case HttpRequestType.delete: {
  const response = await axios.delete(fullUrl, options)
  result = response?.data as R
  break
}

...
```

The **patch** implementation:

```
// file: src/http-client/models/HttpClient.axios.ts

...

// executes a patch request:
case HttpRequestType.patch: {
  const response = await axios.patch(fullUrl, payload, options)
  result = response?.data as R
  break
}

...
```

HttpClient.fetch.ts

The **HttpClientFetch** is the class that implements our **HttpClientInterface** using fetch. Since the code here is longer, let me split in multiple parts.

```
// file: src/http-client/models/HttpClient.fetch.ts
import { HttpRequestParamsInterface } from './HttpRequestParams.interface'
import { HttpClientInterface, HttpClientConfigInterface } from './HttpClient.interfa\
ce'
import { HttpRequestType, HttpRequestMethods, HttpContentTypes } from './Constants'
import { UrlUtils } from './UrlUtils'

/**
 * @name HttpClientFetch
 * @description
 * Wraps http client functionality to avoid directly using fetch
 * and simplify replacement in the future if such npm package would stop being devel\
oped or other reasons
 */
export class HttpClientFetch implements HttpClientInterface {

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

  /**
   * @name request
   * @description
   * A method that executes different types of http requests (i.e. GET/POST/etc)
   * based on the parameters argument.
   * The type R specify the type of the result returned
   * The type P specify the type of payload if any
   * @returns A Promise<R> as the implementation of this method will be async.
   */
  async request<R, P = void>(parameters: HttpRequestParamsInterface<P>): Promise<R> {
    // use destructuring to extract our parameters into local variables
    const { requestType, endpoint, requiresToken, payload, headers, mockDelay } = pa\
rameters

    // use helper to build the fullUrl with request parameters derived from the payl\
oad
    const fullUrl = UrlUtils.getFullUrlWithParams(endpoint, payload as any)
    console.log('HttpClientFetch: fullUrl: ', fullUrl, payload)

    // set fetch options
    const options: RequestInit = {
      credentials: 'include',
      redirect: 'follow',
      headers: {}
    }

    if (headers) {
      options.headers = {
        ...headers
      }
    }

    if (!options.headers?.hasOwnProperty('Content-Type')) {
      // default to content-type json
      options.headers = {
        ...headers,
        'Content-Type': HttpContentTypes.applicationJson
      }
    }

    // set headers Authorization
    if (requiresToken && options.headers) {
      // optional: you could add coded here to set the Authorization header with a b\
earer token
      // options.headers.Authorization = `bearer ${ JwtHelpers.getJwtToken() }`
    }
    let result!: R

    // helper for checking if response is being redirected (302) in fetch
    const checkRedirect = async (resp: any) => {
      if (resp.redirected) {
        // if so, redirect to response url
        document.location = resp.url
        return true
      }
      return false
    }

    try {
      switch (requestType) {

        // TODO: implement a case statement for each request type

        default: {
          console.warn('HttpClientFetch: invalid requestType argument or request typ\
e not implemented')
        }
      }
    } catch (e) {
      //console.error('HttpClientFetch: exception', e)
      throw Error('HttpClientFetch: exception')
    }

    if ((mockDelay || 0) > 0) {
      return new Promise<R>((resolve) => {
        setTimeout(() => {
          resolve(result)
        }, mockDelay)
      })
    }

    return result
  }
}
```

The implementation of the request method starts by destructuring our request **parameters**, creates the fullUrl, setting some fetch **options**, optionally setting an **Authorization** header (commented out for now, but to show how you can do that if you need it), and a **switch** statement that will execute the type of request we want. Let's implement now the different type of requests within each case block of our switch statement.

The **get** implementation:

```
// file: src/http-client/models/HttpClient.fetch.ts

...

// executes a get request:
case HttpRequestType.get: {
  options.method = HttpRequestMethods.get
  const response = (await fetch(fullUrl, options)) as any
  const redirected = await checkRedirect(response)
  if (!redirected) {
    result = (await response.json()) as R
  }
  break
}

...
```

The **post** implementation:

```
// file: src/http-client/models/HttpClient.fetch.ts

...

// executes a post request:
case HttpRequestType.post: {
  options.method = HttpRequestMethods.post
  options.body = typeof payload === 'string' ? payload : JSON.stringify(payload)
  const response = (await fetch(fullUrl, options)) as any
  const redirected = await checkRedirect(response)
  if (!redirected) {
    result = (await response.json()) as R
  }
  break
}

...
```

The **put** implementation:

```
// file: src/http-client/models/HttpClient.fetch.ts

...

// executes a put request:
case HttpRequestType.put: {
  options.method = HttpRequestMethods.put
  options.body = typeof payload === 'string' ? payload : JSON.stringify(payload)
  const response = (await fetch(fullUrl, options)) as any
  const redirected = await checkRedirect(response)
  if (!redirected) {
    result = (await response.json()) as R
  }
  break
}

...
```

The **delete** implementation:

```
// file: src/http-client/models/HttpClient.fetch.ts

...

// executes a delete request:
case HttpRequestType.delete: {
  options.method = HttpRequestMethods.delete
  const response = (await fetch(fullUrl, options)) as any
  const redirected = await checkRedirect(response)
  if (!redirected) {
    result = (await response.json()) as R
  }
  break
}

...
```

The **patch** implementation:

```
// file: src/http-client/models/HttpClient.fetch.ts

...

// executes a patch request:
case HttpRequestType.patch: {
  options.method = HttpRequestMethods.patch
  options.body = typeof payload === 'string' ? payload : JSON.stringify(payload)
  const response = (await fetch(fullUrl, options)) as any
  const redirected = await checkRedirect(response)
  if (!redirected) {
    result = (await response.json()) as R
  }
  break
}

...
```

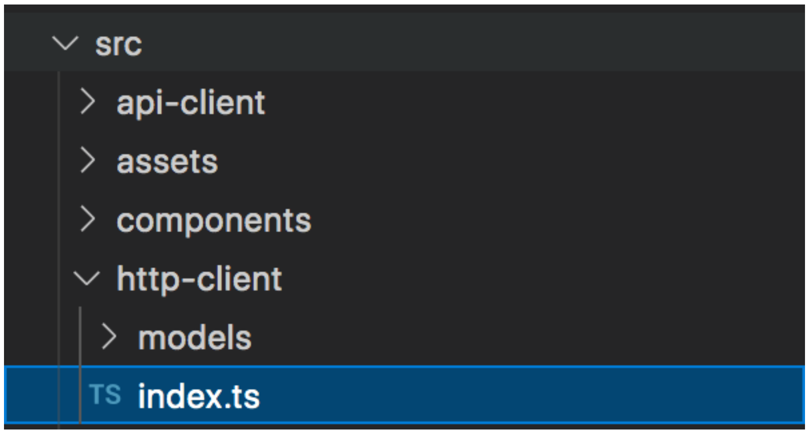
http-client/models/index.ts (barrel file)

Inside the index file paste the following to export all the enums/ interfaces/models:

```
// file: src/http-client/models/index.ts

export * from './Constants'
export * from './HttpClient.axios'
export * from './HttpClient.fetch'
export * from './HttpClient.interface'
export * from './HttpRequestParams.interface'
export * from './UrlUtils'
```

http-client/index.ts (client factory)

Add another index file under src/http-client:



This file contains the export of a single instance of our **HttpClient**. This is what we'll be consuming in our API client. For now, we'll create an instance of the HttpClient.fetch implementation, but in later chapters we'll drive this from configuration (appConfig):

```
// file: src/http-client/index.ts

import { HttpClientInterface } from './models/HttpClient.interface'
// import { config } from '@/config'

import { HttpClientAxios } from './models/HttpClient.axios'
import { HttpClientFetch } from './models/HttpClient.fetch'

// export all our interfaces/models/enums
export * from './models'

let _httpClient: HttpClientInterface | undefined = undefined

// export out hook
export const useHttpClient = () => {
  if (!_httpClient) {
    // export instance of HttpClientInterface
    const clientType = 'fetch'
    // const clientType = config.httpClient.clientType // later will drive from conf\
ig

    // if you'd like to use axios, set "clientType": "axios" within the config files\
 httpClient section
    if (clientType === 'fetch') {
      _httpClient = new HttpClientFetch()
    } else if (clientType === 'axios') {
      _httpClient = new HttpClientAxios()
    }
  }

  return _httpClient as HttpClientInterface
}
```

UrlUtils Unit Tests

Create the directory **tests/unit/http-client** directory and add a new file called **UrlUtils.getFullUrlWithParams.test.ts** with the following:

```
// file: src/tests/unit/http-client/UrlUtils.getFullUrlWithParams.test.ts

import { UrlUtils } from '@/http-client'

describe('UrlUtils: getFullUrlWithParams', () => {
  it('should return fullUrl formatted as expected with one param', () => {
    const endpoint = 'https://unit-test-api/v1/domain/[catalogId]/[partId]'
    const params = {
      catalogId: 5346782,
      partId: 'abcde23'
    }
    const result = UrlUtils.getFullUrlWithParams(endpoint, params)

    expect('https://unit-test-api/v1/domain/5346782/abcde23').toEqual(result)
  })

  // test our component click event
  it('should return fullUrl formatted as expected with multiple params', () => {
    const endpoint = 'https://unit-test-api/v1/domain/[country]/[state]/[cityId]'
    const params = {
      country: 'USA',
      state: 'NY',
      cityId: 'gtref345ytr'
    }
    const result = UrlUtils.getFullUrlWithParams(endpoint, params)

    expect('https://unit-test-api/v1/domain/USA/NY/gtref345ytr').toEqual(result)
  })
})
```

HttpClient: Unit Tests

We need to add unit tests against **HttpClientAxios** and **HttpClientFetch** before we can re-factor the **ItemApiClient** code to use it.

HttpClientAxios tests

Testing a successful "get" response

Within the directory **tests/unit/http-client** directory create a sub-directory called **axios-client** and here and add a new file called **AxiosClient.request.get.test.ts**. Within the file, paste the following code:

```
// file: src/tests/unit/http-client/axios-client/AxiosClient.request.get.test.ts

import axios from 'axios'
import { HttpClientAxios, HttpRequestType, HttpRequestParamsInterface } from '@/http\
-client'

let mockRequestParams: HttpRequestParamsInterface<any> = {
  requestType: HttpRequestType.get,
  endpoint: 'path/to/a/get/api/endpoint',
  requiresToken: false
}

describe('HttpClient: axios-client: request: get', () => {
  const httpClient = new HttpClientAxios()

  it('should execute get request succesfully', () => {
    vitest
      .spyOn(axios, 'get')
      .mockImplementation(async () => Promise.resolve({ data: `request completed: ${\
mockRequestParams.endpoint}` }))

    httpClient
      .request(mockRequestParams)
      .then((response) => {
        //console.debug('response:', response)
        expect(response).toEqual(`request completed: ${mockRequestParams.endpoint}`)
      })
      .catch((error) => {
        console.info('AxiosClient.request.get.test.ts: error', error)
      })
  })

  ...
```

Testing an unsuccessful "get" response

Within the same file, add the following code:

```
// file: src/tests/unit/http-client/axios-client/AxiosClient.request.get.test.ts

...

describe('HttpClient: axios-client: request: get', () => {

  ...

  it('get should throw error on rejection', () => {
    vitest
      .spyOn(axios, 'get')
      .mockImplementation(async () => Promise.reject({ data: `request completed: ${m\
ockRequestParams.endpoint}` }))

    httpClient.request(mockRequestParams).catch((error) => {
      expect(error).toBeDefined()
      expect(error.toString()).toEqual('Error: HttpClientAxios: exception')
    })
  })
})
```

Testing a successful "post" response

Within the directory **tests/unit/http-client/axios-client** directory and add a new file called **AxiosClient.request.post.test.ts**. Within the file, paste the following code:

```
// file: src/tests/unit/http-client/axios-client/AxiosClient.request.post.test.ts

import axios from 'axios'
import { HttpClientAxios, HttpRequestType, HttpRequestParamsInterface } from '@/http\
-client'

let mockRequestParams: HttpRequestParamsInterface<any> = {
  requestType: HttpRequestType.post,
  endpoint: 'path/to/a/post/api/endpoint',
  requiresToken: false,
  payload: {}
}

type P = typeof mockRequestParams.payload

describe('HttpClient: axios-client: request: post', () => {
  const httpClient = new HttpClientAxios()

  it('should execute post request succesfully', () => {
    vitest
      .spyOn(axios, 'post')
      .mockImplementation(async () => Promise.resolve({ data: `request completed: ${\
mockRequestParams.endpoint}` }))

    httpClient
      .request<string, P>(mockRequestParams)
      .then((response) => {
        //console.debug('response:', response)
        expect(response).toEqual(`request completed: ${mockRequestParams.endpoint}`)
      })
      .catch((error) => {
        console.info('AxiosClient.request.post.test.ts: post error', error)
      })
  })
})
```

Note: you can keep adding more test in a similar way for the rest of the request type like PUT/DELETE/etc

HttpClientFetch tests

Testing "get" responses

Within the directory **tests/unit/http-client** directory create a sub-directory called **fetch-client** and here and add a new file called **FetchClient.request.get.test.ts**. Within the file, paste the following code:

```
// file: src/tests/unit/http-client/fetch-client/FetchClient.request.get.test.ts

import { HttpClientFetch, HttpRequestType, HttpRequestParamsInterface, HttpRequestMe\
thods } from '@/http-client'

let mockRequestParams: HttpRequestParamsInterface<any> = {
  requestType: HttpRequestType.get,
  endpoint: 'path/to/a/get/api/endpoint',
  requiresToken: false
}

describe('HttpClient: axios-client: request: get', (done) => {
  const httpClient = new HttpClientFetch()

  it('should execute get request succesfully', async () => {

    // could not find an easy way to use spyOn for fetch so overriding global.fetch
    // save original fetch
    const unmockedFetch = global.fetch || (() => {})
    global.fetch = unmockedFetch

    const expectedResult = {
      result: `request completed: ${mockRequestParams.endpoint}`
    }

    vitest
      .spyOn(global, 'fetch')
      .mockImplementation(async () => Promise.resolve({
        redirected: false,
        json: () => Promise.resolve(JSON.stringify(expectedResult))
      } as any))

    try {
      const response = await httpClient.request(mockRequestParams)
      expect(response).not.toBeNull()
      expect(response).toEqual(expectedResult)
    } catch (error) {
      console.info('AxiosClient.request.get.test.ts: error', error)
    }

    // restore globa.fetch
    global.fetch = unmockedFetch
  })

  it('get should throw error on rejection', () => {
    // could not find an easy way to use spyOn for fetch so overriding global.fetch
    // save original fetch
    const unmockedFetch = global.fetch || (() => {})
    global.fetch = unmockedFetch

    vitest
      .spyOn(global, 'fetch')
      .mockImplementation(async () => Promise.reject())

    httpClient.request(mockRequestParams).catch((error) => {
      expect(error).toBeDefined()
      expect(error.toString()).toEqual('Error: HttpClientFetch: exception')
    })
  })
})
```

And so on, you can keep adding more unit tests for each request type like you did for the axios-client.

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

ItemsApiClientModel Update

Open the file **src/api-client/models/items/ItemsApiClient.model.ts**.
Remove the import **axios** line and replace it with an import for our **HttpClient** instance and the **HttpRequestParamsInterface**:

```
// file: src/api-client/models/items/ItemsApiClient.model.ts

import axios, { AxiosRequestConfig, AxiosError, AxiosResponse } from 'axios' // <-- \
remove this line
import { useHttpClient, HttpRequestType, HttpRequestParamsInterface } from '@/http-c\
lient' // <-- add this line
```

Then replace the **fetchItems** implementation with the following:

```
// file: src/api-client/models/items/ItemsApiClient.model.ts

...

  fetchItems(): Promise<ItemInterface[]> {
    const requestParameters: HttpRequestParamsInterface = {
      requestType: HttpRequestType.get,
      endpoint: this.endpoints.fetchItems,
      requiresToken: false,
      mockDelay: this.mockDelay
    }

    return useHttpClient().request<ItemInterface[]>(requestParameters)
  }

...
```

This creates a const variable to hold our **HttpRequestParamsInterface** parameters, and then return the call to **useHttpClient().request** (which is already a Promise, so we do not have to do anything else here):

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 an http client into interfaces and models that are generic
- How to implement the **HttpClientInterface** into a model that **encapsulate** the use of a 3rd party package in one place. We show this by implement two different clients: HttpClientAxios and HttpClientFetch.
- How to use **vitest.spyON** for stubs so we can test the HttpClient **request** method responses for different scenarios.

Observations

- We did not write unit tests against the **HttpClient** put/delete/patch methods
- We did not write unit tests against the **ItemsApiClientModel**

Based on these observations, there are a few improvements that you could make on your own:

Improvements

- Add unit tests against the **HttpClient** put/delete/patch methods as well
- Add unit tests against the **ItemsApiClient** methods as well
- Experiment by adding another HttpClient implementation that uses another Ajax library other than axios or fetch and then and modify the file src/http-client/index.ts so that it instantiate this one instead of the axios or fetch implementation. Then verify that the app still run as expected.

Chapter 9 - App Configuration

We need now to add a way to configure our app through configuration files for different environments (i.e. **mock**, **beta**, **production**, etc).

*NOTE: The code in this chapter is not just specific to **Vue**. These concepts can be applied to any front-end app (i.e. React/Vue/Svelte/Angular/etc).*

As you recall from Chapter 7, we extended **import.meta.env** declaration types (file src/vite-env.d.ts) to include a new variable called **VITE_API_CLIENT**. This currently drives the selection of the API client at run time (mock or live). As you can imagine, as we add more configuration, we might end adding a lot of new variables prefixed with ***VITE_***. This works, but can quickly become very hard to manage, especially for large configurations that will drive many settings.

A better approach is to drive the entire configuration through only one variable that we are going to call **VITE_APP_CONFIG**. We'll store all the settings in dedicated JSON files. We'll have one configuration file for each environment (mock/beta/production/etc) and then load that dynamically at run-time (or build time) based on our new **VITE_APP_CONFIG** environment variable.

vite-env.d.ts updates (or env.d.ts)

Let's start by modifying the code within the **Vite** types declarations file. Rename the current variable VITE_API_CLIENT to **VITE_APP_CONFIG**:

```
// file: src/vite-env.d.ts (or src/env.d.ts)

...


// types for Vite env variables:
// (reference: https://vitejs.dev/guide/env-and-mode.html#intellisense-for-typescrip\
t)
interface ImportMetaEnv {
  readonly VITE_APP_CONFIG: string // rename this from VITE_API_CLIENT to VITE_APP_C\
ONFIG
  // more env variables...
}

interface ImportMeta {
  readonly env: ImportMetaEnv
}
```

.env files updates

Make sure to also update each '.env' file by renaming VITE_API_CLIENT to **VITE_APP_-CONFIG**:

```
// file: src/.env.mock

VITE_APP_CONFIG=mock
```

```
// file: src/.env.production

VITE_APP_CONFIG=production
```

Add also three additional files, `.env.jsonserver`, `.env.localapis` and `.env.beta`:

```
// file: src/.env.jsonserver

VITE_APP_CONFIG=jsonserver
```

```
// file: src/.env.localapis

VITE_APP_CONFIG=localapis
```

```
// file: src/.env.beta

VITE_APP_CONFIG=beta
```

Note: remember that VITE read from the .env files based on the **–mode** flag specified in the scripts shortcut (within the package.json file). Here we did not add any command for **localapis** or **beta**. But you could add things like **start-local** or build-beta etc:

```
// file: package.json

...

  "scripts": {
    "start": "npm run dev",
    "dev": "vite --config vite.config.mock.ts --mode mock",
    "build": "vue-tsc --noEmit && vite build --config vite.config.production.ts --mo\
de production",
    "build-mock": "vite build --config vite.config.mock.ts --mode mock",
    "build-beta": "vite build --config vite.config.production.ts --mode beta", /* yo\
u could add this */
    "start-local": "vite --config vite.config.production.ts --mode localapis", /* yo\
u could add this */
    "preview": "vite preview --config vite.config.mock.ts --mode mock",
    "test": "vitest run --config vite.config.mock.ts --mode mock",
    "test-watch": "vitest watch --config vite.config.mock.ts --mode mock",
    "test-coverage": "vitest run --coverage --config vite.config.mock.ts --mode mock\
",
  }

...
```

Config Interface

Create the directory **src/config/models/** 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 (as we add more functionality/components etc we will add more areas/domains similar to this)

For the items section, we'll have only the **apiClientOptions** child section for now. This will be of type **ItemsApiClientOptions**.

Here is the code for the **src/config/models/Config.interface.ts** file:

```
// file: src/config/models/Config.interface.ts

import {
  ItemsApiClientOptions // NOTE: we'll create this a bit later
} from '@/api-client/models'

export interface HttpClientConfigInterface {
  tokenKey: string
  clientType: string
}

/**
 * @Name ConfigInterface
 * @description
 * Describes the structure of a configuration file
 */
export interface ConfigInterface {
  global: {
    // ... things that are not specific to a single app domain
    version: number
  }

  httpClient: HttpClientConfigInterface,

  apiClient: {
    type: string
  }

  items: {
    apiClientOptions: ItemsApiClientOptions
  }

}
```

Config files

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 with the following names:

- **mock.json**

- **jsonserver.json**
- **localapis.json**
- **beta.json**
- **production.json**

The content of each file will have to match what is required by our **ConfigInterface**. In a little bit we'll be also adding some unit tests against this files to make sure they are as expected.

Here is the content of each file:

mock.json

```
// file: src/config/config-files/mock.json
{
  "global": {
    "version": 0.103
  },

  "httpClient": {
    "tokenKey": "myapp-token",
    "clientType": "fetch"
  },

  "apiClient": {
    "type": "mock"
  },

  "items": {
    "apiClientOptions": {
      "endpoints": {
        "fetchItems": "/static/mock-data/items/items.json"
      },
      "mockDelay": 250
    }
  }
}
```

jsonserver.json

```
// file: src/config/config-files/jsonserver.json
{
  "global": {
    "version": 0.1
  },

  "httpClient": {
    "tokenKey": "myapp-token",
    "clientType": "fetch"
  },

  "apiClient": {
    "type": "live"
  },

  "items": {
    "apiClientOptions": {
      "endpoints": {
        "fetchItems": "/jsonserver/items"
      },
      "mockDelay": 0
    }
  }
}
```

localapis.json

```
// file: src/config/config-files/localapis.json
{
  "global": {
    "version": 0.1
  },

  "httpClient": {
    "tokenKey": "myapp-token",
    "clientType": "fetch"
  },

  "apiClient": {
    "type": "live"
  },
  "items": {
    "apiClientOptions": {
      "endpoints": {
        "fetchItems": "http://api.localhost:4111/items"
      },
      "mockDelay": 0
    }
  }
}
```

beta.json

```
// file: src/config/config-files/beta.json
{
  "global": {
    "version": 0.1
  },

  "httpClient": {
    "tokenKey": "myapp-token",
    "clientType": "fetch"
  },

  "apiClient": {
    "type": "live"
  },

  "items": {
    "apiClientOptions": {
      "endpoints": {
        "fetchItems": "/path/to/your/real/BETA/api/and-point"
      },
      "mockDelay": 0
    }
  }
}
```

production.json

```
// file: src/config/config-files/production.json
{
  "global": {
    "version": 0.1
  },

  "httpClient": {
    "tokenKey": "myapp-token",
    "clientType": "fetch"
  },

  "apiClient": {
    "type": "live"
  },

  "items": {
    "apiClientOptions": {
      "endpoints": {
        "fetchItems": "/path/to/your/real/PRODUCTION/api/and-point"
      },
      "mockDelay": 0
    }
  }
}
```

tsconfig.json updates

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": {

...
    "resolveJsonModule": true, /* this allows to import .json file as if they were .\
ts files: using to load config files */
  }
...
```

NOTE: your tsconfig.json might already have the resolveJsonModule flag, if so just make sure that is set to true.

Config files map

Within the directory **src/config/** add a file called **config-files-map.ts**.
Here we just import a reference to each of the configuration JSON files and create either a **strategy pattern** or a **JavaScript Map** that contains a map to our files by environment key (here we are showing this with **Map**):

```
// file: src/config/config-files-map.ts

// import a reference to our Config interface:
import { ConfigInterface } from './models/Config.interface'

// individual environments configs:
import configMock from './config-files/mock.json'
import configJsonServer from './config-files/jsonserver.json'
import configLocal from './config-files/localapis.json'
import configBeta from './config-files/beta.json'
import configProduction from './config-files/production.json'

// example with javascript Map()
export const configFilesMap: Map<string, ConfigInterface> = new Map<string, ConfigIn\
terface>([
  ['mock', configMock],
  ['jsonserver', configJsonServer],
  ['localapis', configLocal],
  ['beta', configBeta],
  ['production', configProduction]
])
```

Config provider

File utils.ts

Add a new file called **utils.ts** under src/config. Here we implement an helper function called **getAppConfigKey** that will return the value of our **VITE_APP_CONFIG** environment variable.

```
// file: src/config/utils.ts

// helper to read the value of VUE_APP_CONFIG (or VITE_APP_CONFIG if using vite)
export function getAppConfigKey() {
  // if using create-react-ap (webpack):
  // let env: string = 'mock'
  // // @ts-ignore
  // if (process.env && process.env.VUE_APP_CONFIG) {
  //  // @ts-ignore
  //   env = process.env.VUE_APP_CONFIG.trim()
  // }
  // return env

  // Note: Vite uses import.meta.env (reference: https://vitejs.dev/guide/env-and-mo\
de.html)
  // optional: you can console.log the content of import.meta.env to inspect its val\
ues like this: console.log('import.meta.env', JSON.stringify(import.meta.env))
  // @ts-ignore
  return (import.meta.env.VITE_APP_CONFIG || '').trim()
}
```

Note that by wrapping this in one place, we could easily re-use this code in a project created with webpack. Since that uses process.env instead of Vite's import.meta.env we just need to uncomment the related code above and comment out the one that uses import.meta.env.

File index.ts (config provider)

Still within **src/config** directory, add another file called **index.ts**. Here we'll be consuming the **JSON** file that matches the environment specified by the current **VITE_APP_CONFIG** value.

Let's start by importing a reference to **ConfigInterface**, **configFilesMap**, and our helper **getAppConfigKey**:

```
// file: src/config/index.ts
// returns appropriate config based on env VITE_APP_CONFIG

// import a reference to our Config interface:
import { ConfigInterface } from './models/Config.interface'

// import reference to configFilesMap
import { configFilesMap } from './config-files-map'

// import reference to our getAppConfigKey helper function
import { getAppConfigKey } from './utils'

...
```

Then add a check and throw and error if our map does not contain an entry for the current environment key:

```
// file: src/config/index.ts

...

if (!configFilesMap.has(getAppConfigKey())) {
  throw Error(`Could not find config for VITE_APP_CONFIG key "${ getAppConfigKey() }\
"`)
}

...
```

Finally we export an instance of our **ConfigInterface** called **config**:

```
// file: src/config/index.ts

...

export const config = configFilesMap.get(getAppConfigKey()) as ConfigInterface
```

Here is the entire content of src/config/index.ts:

```
// file: src/config/index.ts
// returns appropriate config based on env VITE_APP_CONFIG

// import a reference to our Config interface:
import { ConfigInterface } from './models/Config.interface'

// import reference to configFilesMap
import { configFilesMap } from './config-files-map'

// import reference to our getAppConfigKey helper function
import { getAppConfigKey } from './utils'

// optional: you can console.log the content of import.meta.env to inspect its value:
console.log(`------ env ---- "${getAppConfigKey()}"`)

if (!configFilesMap.has(getAppConfigKey())) {
  throw Error(`Could not find config for VITE_APP_CONFIG key "${getAppConfigKey()}"`)
}

export const config = configFilesMap.get(getAppConfigKey()) as ConfigInterface
```

Unit Tests

Le'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:

```
// file: src/tests/unit/config/config-files-map.test.ts

import { configFilesMap } from '@/config/config-files-map'

describe('configFilesMap', () => {

  it('instance should have "mock" key', () => {
    expect(configFilesMap.has('mock')).toBe(true)
  })

  it('instance should have "jsonserver" key', () => {
    expect(configFilesMap.has('jsonserver')).toBe(true)
  })

  it('instance should have "localapis" key', () => {
    expect(configFilesMap.has('localapis')).toBe(true)
  })

  it('instance should have "beta" key', () => {
    expect(configFilesMap.has('beta')).toBe(true)
  })

  it('instance should have "production" key', () => {
    expect(configFilesMap.has('production')).toBe(true)
  })
}
```

Unit Tests against Config instances by environment

Note, if using Jest, we cannot just write unit tests against the config instance already created in the **src/config/index.ts** file because Jest/Vitest will throw the following error:

```
SyntaxError: Cannot use 'import.meta' outside a module
```

Note: Jest does not understand VITE's import.meta.dev our of the box. There are discussion on the web if you google this and some people are using plugins or specific babel config etc. Here will simply do not test the config instance from the src/config/index.ts file but create a new one in our unit tests

Vitest does not have this issue but I thought it wa worth mentioning it.

To be safe in either case, we will write our unit tests by creating an instance of ConfigInterface within the unit tests. We can easily do this by specifying the environment key as a hard-coded string to the map.get method, i.e. `configFilesMap.get('mock')`.

Tests config.mock.test.ts

Still under directory **tests/unit/config** add another file called **config.mock.test.ts** with the following code:

```
// file: src/tests/unit/config/config.mock.test.ts

// import the Config interface
import { ConfigInterface } from '@/config/models/Config.interface'
// import a reference to the confiFilesMap
import { configFilesMap } from '@/config/config-files-map'

describe('config: mock', () => {
  const config: ConfigInterface = configFilesMap.get('mock') as ConfigInterface

  it('instance should have "global" section', () => {
    expect(config).toHaveProperty('global')
  })

  it('instance should have "httpClient" section', () => {
    expect(config).toHaveProperty('httpClient')
  })

  it('instance should have "items" section', () => {
    expect(config).toHaveProperty('items')
  })

  describe('global', () => {
    const section = config.global
    it('section should have "version" property', () => {
      expect(section).toHaveProperty('version')
      expect(typeof section.version).toBe('number')
      expect(section.version).toBeGreaterThan(0)
    })
  })

  describe('httpClient', () => {
    const section = config.httpClient
    it('section should have "tokenKey" property', () => {
      expect(section).toHaveProperty('tokenKey')
    })

    it('section should have "clientType" property', () => {
      expect(section).toHaveProperty('clientType')
    })
  })

  describe('apiClient', () => {
    const section = config.apiClient
    it('section should have "type" property', () => {
      expect(section).toHaveProperty('type')
    })
  })

  describe('items', () => {
    const section = config.items

    it('section should have "apiClientOptions" property', () => {
      expect(section).toHaveProperty('apiClientOptions')
    })

    describe('apiClientOptions', () => {
      const apiClientOptions = section.apiClientOptions

      describe('endpoints', () => {
        const endpoints = apiClientOptions.endpoints

        it('section should have "fetchItems" property', () => {
          expect(endpoints).toHaveProperty('fetchItems')
          // verify that fetchItems url is a string and has a reasonable length
          expect(typeof endpoints.fetchItems).toBe('string')
          expect(endpoints.fetchItems.length).toBeGreaterThan(10)
        })
      })
    })
  })
})
```

Run the unit tests with **npm run test**: and verify all succeed:

```
// terminal output:

...

Test Files  8 passed (8)
     Tests  25 passed (24)
      Time  1.18s (in thread 77ms, 1537.16%)

...
```

Please keep adding additional unit tests for each environment (i.e. config.jsonserver.test.ts, config.production.test.ts etc).

HttpClient code updates

file src/http-client/index.ts

Now we need to update the file **src/http-client/index.ts** and remove the hard-coded value for the **clientType** variable. We'll be instead reading the value from the config instance (**config.httpClient.type**):

```
// file: src/http-client/index.ts

import { HttpClientInterface } from './models/HttpClient.interface'
import { config } from '@/config' // <-- uncomment (or add) this line

import { HttpClientAxios } from './models/HttpClient.axios'
import { HttpClientFetch } from './models/HttpClient.fetch'

// export all our interfaces/models/enums
export * from './models'

let _httpClient: HttpClientInterface | undefined = undefined

// export out hook
export const useHttpClient = () => {
  if (!_httpClient) {
    // export instance of HttpClientInterface
    const clientType = config.httpClient.clientType // <-- uncomment this line
    // const clientType = 'fetch' // <-- remove this line
    // if you'd like to use axios, set "clientType": "axios" within the config files\
 (section "httpClient")
    if (clientType === 'fetch') {
      _httpClient = new HttpClientFetch()
    } else if (clientType === 'axios') {
      _httpClient = new HttpClientAxios()
    }
  }

  return _httpClient as HttpClientInterface
}
```

Api Client code updates

file src/api-client/index.ts

Update also the file **src/api-client/index.ts** and remove the code currently using the previous env variable and replace it by consuming our value from the config instance (**config.apiClient.type**). Here is the updated code:

```
// file: src/api-client/index.ts

import { ApiClientInterface } from './models'
import { apiMockClient } from './mock'
import { apiLiveClient } from './live'

import { config } from '@/config'

// return either the live or the mock client
let apiClient: ApiClientInterface
if (config.apiClient.type === 'live') { // this time we just read our config.apiClie\
nt.type
  apiClient = apiLiveClient
} else {
  // default is always apiMockClient
  apiClient = apiMockClient
}

export { apiClient }
```

file src/api-client/mock/items/index.ts

Update the code that returns the mock Items API client instance to use the **apiClientOptions** from the **config**:

```
// file: src/api-client/mock/items/index.ts

import { config } from '@/config' // <-- add this line

import {
  ItemsApiClientInterface,
  ItemsApiClientModel
} from '../../models/items'

// remove this block:
// const options: ItemsApiClientOptions = {
//   mockDelay: 250,
//   endpoints: {
//     fetchItems: '/static/mock-data/items/items.json'
//   }
// }

// instantiate the ItemsApiClient pointing at the url that returns static json mock \
data
const itemsApiClient: ItemsApiClientInterface = new ItemsApiClientModel(config.items\
.apiClientOptions) // <-- this time we'll pass the options from the config

// export our instance
export {
  itemsApiClient
}
```

file src/api-client/live/items/index.ts

Similarly, update the code that returns the live Items API client instance to use the **apiClientOptions** from the **config**:

```
// file: src/api-client/live/items/index.ts

// import a reference to the app config
import { config } from '@/config'

import { ItemsApiClientInterface, ItemsApiClientModel } from '../../models'

// instantiate the ItemsApiClient pointing at the url that returns static json live \
data
const itemsApiClient: ItemsApiClientInterface = new ItemsApiClientModel(config.items\
.apiClientOptions)

// export our instance
export { itemsApiClient }
```

*IMPORTANT: 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 basically identical. In later chapters we will simplify and reduce the amount of code we initially created to serve either **mock** or **live** data. But for now, we'll keep the duplicated code to avoid making this chapter too long.*

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 9 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 **VITE_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 code

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.
- We did not write unit tests again each config file like we did for config.mock.test.ts

Improvements

- Going forward we'll be expanding the configuration as we keep growing our application components and logic.
- You can write additional unit tests similar to config.mock.test.ts for beta and production as well (i.e. config.beta.test.ts, config.mock.production.ts, etc)

Chapter 10 - Localization and Internationalization - Language Localization

”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”[32]

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 specific to that country (hence, localized).

In this book we’ll first leverage plugins that allows us to present labels in different languages (**i18next**, **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).

Plugins: i18next, 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. The organization **i18next**[33] maintains several plugins for React/Vue/etc but at the time of writing this book nothing was available for **Vue**.

Github user @keisermann (under organization intlify) developed **vue-i18n**[34] which wraps some of the reactive tools that Vue already provides. This is is published on NPM here https://www.npmjs.com/package/vue-i18n

[32]https://www.w3.org/International/questions/qa-i18n
[33]https://github.com/i18next
[34]https://www.npmjs.com/package/vue-i18n

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

Let's start by first adding the **i18next** and **vue-i18n** NPM packages to our application. In particular, we'll be installing vue-18n-next which is for Vue 3.

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 i18next vue-i18n@9
```

Config updates

ConfigInterface

We will be introducing a concept of versioning here to dynamically drive different versions of data, introduce/retire views and components overtime, or expire cached data on the browser. You will see this in action first shortly when we'll use to expire our translation data stored in the browser cache.

Let's add a field called **version** to our **global** section (note: we might have added this already in the one of previous chapters):

```
// file: src/config/models/Config.interface.ts

...

export interface ConfigInterface {
  global: {
    version: number // add this line
  }

...
```

Let's also add a new section called **localization** like this:

```
// file: src/config/models/Config.interface.ts

...

export interface ConfigInterface {
  global: {
    version: number
  }

  ...
  // add this block:
  localization: {
    apiClientOptions: LocalizationApiClientOptions
    locales: { key: string, isDefault: boolean }[]
    localStorageCache: { enabled: boolean, expirationInMinutes: number }
  }

...
```

Here we reference a **apiClientOptions**. We don't have this yet. We'll add a new API client module called **localization** shortly.

First, let's finish updating the code related to the configuration.

file mock.json

Let's add the data we need in the **src/config/files/mock.json** as per our ConfigInterface updates:

```
// file: src/config/config-files/mock.json

{
  "global": {
    "version": 0.1 // add this line
  },

  ...

  // begin: add the localization section
  "localization": {
    "apiClientOptions": {
      "endpoints": {
        "fetchTranslation": "/static/mock-data/localization/[namespace]/[key].json"
      },
      "mockDelay": 250
    },
    "locales": [
      // each of this objects represent a locale available in our app
      { "key": "en-US", "isDefault": true },
      { "key": "it-IT", "isDefault": false },
      { "key": "fr-FR", "isDefault": false },
      { "key": "es-ES", "isDefault": false }
    ],
    "localStorageCache": {
      // these are settings we'll use to cache JSON locale translation data into loc\
aleStorage
      "enabled": true,
      "expirationInMinutes": 60
    }
  }
  // end: add the localization section
}
```

Please feel free to also update the beta.json/production.json/localapis.json files as well, and possibly add unit tests to validate your changes. Note that we have also an array called **locales** which hold a list of object that represent each of the locales available in or application. We also have a section caled localeStorageCache that we'll use to drive how we cache the locale translation JSON data into the browser **localStorage.**[35]

Translation JSON data

Note that for the **fetchTranslation** end-point, we'll use two parameters: **[namespace]** and **[key]**.
We'll create the files under `/public/static/mock-data/localization/`.
For the [namespace] parameter we'll always use 'translation' in our case, so go ahead and create a sub-directory called **translation** at the path /public/static/mock-data/localization/translation.
Then, add 4 files:

- en-US.json
- es-ES.json
- fr-FR.json

[35]https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

- it-IT.json

Here is the content for the **en-US.json** one:

```
// file: public/static/mock-data/localization/translation/en-US.json
{
  "locale.selector.en-US": "English",
  "locale.selector.it-IT": "Italian",
  "locale.selector.fr-FR": "French",
  "locale.selector.es-ES": "Spanish",

  "home.welcome": "Welcome: this message is localized in English",

  "navigation.home": "Home",
  "navigation.about": "About",

  "items.list.header": "My Items"
}
```

Here is the content for the **es-ES.json** one:

```
// file: public/static/mock-data/localization/translation/es-ES.json
{
  "locale.selector.en-US": "Inglés",
  "locale.selector.it-IT": "Italiano",
  "locale.selector.fr-FR": "Francés",
  "locale.selector.es-ES": "Español",

  "home.welcome": "Bienvenido: this message is localized in Spanish",

  "navigation.home": "Inicio",
  "navigation.about": "Acerca de",

  "items.list.header": "Mis cosas"
}
```

Here is the content for the **fr-FR.json** one:

```
// file: public/static/mock-data/localization/translation/fr-FR.json
{
  "locale.selector.en-US": "Anglais",
  "locale.selector.it-IT": "Italien",
  "locale.selector.fr-FR": "Français",
  "locale.selector.es-ES": "Espagnol",

  "home.welcome": "Bienvenue: this message is localized in French",

  "navigation.home": "Accueil",
  "navigation.about": "À propos de nous",

  "items.list.header": "Mes articles"
}
```

Here is the content for the **it-IT.json** one:

```
// file: public/static/mock-data/localization/translation/it-IT.json
{
  "locale.selector.en-US": "Inglese",
  "locale.selector.it-IT": "Italiano",
  "locale.selector.fr-FR": "Francese",
  "locale.selector.es-ES": "Spagnolo",

  "home.welcome": "Benvenuti: this message is localized in Italian",

  "navigation.home": "Home",
  "navigation.about": "Chi Siamo",

  "items.list.header": "I miei articoli"
}
```

We now have to add a new API client module for loading our **localization** data.

API Client updates

Create the directory **src/api-client/models/localization**.
Here we'll create the interfaces and model for our localization API module.
Add the following 4 files:

- LocalizationApiClient.interface.ts

- LocalizationApiClientOptions.interface.ts
- LocalizationApiClient.model.ts
- index.ts

file LocalizationApiClient.interface.ts

Our localization API client will exposes one method called fetchTranslation:

```
// file: src/api-client/models/localization/LocalizationApiClient.interface.ts

/**
 * @Name LocalizationApiClientInterface
 * @description
 * Interface for the Localization api client module
 */
export interface LocalizationApiClientInterface {
  fetchTranslation: (namespace: string, key: string) => Promise<{ [key: string]: str\
ing }>
}
```

file LocalizationApiClientOptions.interface.ts

Here we have the itnerfaces for the API client configuration:

```
// file: src/api-client/models/localization/LocalizationApiClientOptions.interface.ts

export interface LocalizationApiClientEndpoints {
  fetchTranslation: string
}

/**
 * @Name LocalizationApiClientOptions
 * @description
 * Interface for the Localization api client options (includes endpoints used to avo\
id hard-coded strings)
 */
export interface LocalizationApiClientOptions {
  mockDelay?: number
  endpoints: LocalizationApiClientEndpoints
}
```

file LocalizationApiClient.model.ts

Here is the implementation of our localization API client:

```
// file: src/api-client/models/localization/LocalizationApiClient.model.ts

import { useHttpClient, HttpRequestParamsInterface, HttpRequestType } from '@/http-c\
lient'

import { LocalizationApiClientOptions, LocalizationApiClientEndpoints } from './Loca\
lizationApiClientOptions.interface'
import { LocalizationApiClientInterface } from './LocalizationApiClient.interface'

/**
 * @Name LocalizationApiClientModel
 * @description
 * Implements the LocalizationApiClientInterface interface
 */
export class LocalizationApiClientModel implements LocalizationApiClientInterface {
  private readonly endpoints!: LocalizationApiClientEndpoints
  private readonly mockDelay: number = 0

  constructor(options: LocalizationApiClientOptions) {
    this.endpoints = options.endpoints
    if (options.mockDelay) {
      this.mockDelay = options.mockDelay
    }
  }

  fetchTranslation(namespace: string, key: string): Promise<{ [key: string]: string \
}> {
    const requestParameters: HttpRequestParamsInterface = {
      requestType: HttpRequestType.get,
      endpoint: this.endpoints.fetchTranslation,
      requiresToken: false,
      payload: {
        namespace,
        key
      } as any,
      mockDelay: this.mockDelay
    }

    return useHttpClient().request<{ [key: string]: string }>(requestParameters)
  }
}
```

file src/api-client/models/localization/index.ts

Just a barrel file:

```
// file: src/api-client/models/localization/index.ts

export * from './LocalizationApiClientOptions.interface'
export * from './LocalizationApiClient.interface'
export * from './LocalizationApiClient.model'
```

Updates to ApiClient.interface.ts

Import and add our localization module:

```
// file: src/api-client/models/ApiClient.interface.ts

import { LocalizationApiClientInterface } from './localization'
import { ItemsApiClientInterface } from './items'

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

Updates to the main models barrel file (api-client/models/index.ts)

```
// file: src/api-client/models/index.ts

export * from './ApiClient.interface'
export * from './localization'
export * from './items'
```

Updates to ApiClient instances

localization mock instance (api-client/mock/localization/index.ts)

Create the file api-client/mock/localization/index.ts with this code:

```
// file: src/api-client/mock/localization/index.ts

// import a reference to the app config
import { config } from '@/config'

import { LocalizationApiClientInterface, LocalizationApiClientModel } from '../../mo\
dels'

// instantiate the LocalizationApiClient pointing at the url that returns static jso\
n mock data
const localizationApiClient: LocalizationApiClientInterface = new LocalizationApiCli\
entModel(config.localization.apiClientOptions)

// export our instance
export { localizationApiClient }
```

mock instance (api-client/mock/index.ts)

Update the code within the api-client/mock/index.ts file like this:

```
// file: src/api-client/mock/index.ts

import { ApiClientInterface } from '../models'
// import module instances
import { localizationApiClient } from './localization'
import { itemsApiClient } from './items'

// create an instance of our main ApiClient that wraps the mock child clients
const apiMockClient: ApiClientInterface = {
  localization: localizationApiClient,
  items: itemsApiClient
}

// export our instance
export { apiMockClient }
```

live instance (api-client/live/index.ts)

Update the code within the api-client/live/index.ts file like this :

```
// file: src/api-client/live/index.ts

// import a reference to the app config
import { config } from '@/config'

import {
  ApiClientInterface,
  LocalizationApiClientModel,
  ItemsApiClientModel
} from '../models'

// create an instance of our main ApiClient that wraps the live child clients
const apiLiveClient: ApiClientInterface = {
  localization: new LocalizationApiClientModel(config.localization.apiClientOptions),
  items: new ItemsApiClientModel(config.items.apiClientOptions)
}
// export our instance
export { apiLiveClient }
```

Note: for the live instance going forward we'll just initialize the client modules within this file so you can go ahead and delete theapi-client/live/items sub-directory.

i18n initialization and useLocalization hook

Create the directory **src/localization**. Inside the localization folder we'll create the following files:

- utils.ts
- I18n.init.ts
- useLocalization.ts
- index.ts

file utils.ts

Here we'll have three helper methods:

- **getDefaultLocale** used to retrieve the default locale
- **getUserPreferredLocale** used to retrieve the user preferred locale from local storage
- **setUserPreferredLocale** used to save the user preferred locale to local storage

```
// file: src/localization/utils.ts

import { config } from '@/config'

// key that will use to save the user preferred locale id
const userPreferredLocaleStorageKey = 'user-lcid'

// helper to returnt he default locael form config
export const getDefaultLocale = () => {
  // get a reference from the available locales array from our config
  const availableLocales = config.localization.locales
  // return the one marked isDefault
  return availableLocales.find((o) => o.isDefault) as {
    key: string
    isDefault: boolean
  }
}

// helper method to retrieve the user preferred locale from localStorage
export const getUserPreferredLocale = () => {
  // try to retrive from local storage if they have one saved
  const preferredLocale = localStorage.getItem(userPreferredLocaleStorageKey)
  if (!preferredLocale) {
    // if not, use the default locale from config
    const defaultLocale = getDefaultLocale().key
    return defaultLocale
  }
  return preferredLocale
}

// helper to save the user preferred locale to localStorage
export const setUserPreferredLocale = (lcid: string) => {
  localStorage.setItem(userPreferredLocaleStorageKey, lcid)
}
```

Note that getUserPreferredLocale will return the default locale from config if there is not preferred locale in localStorage yet.

file i18n.init

Here we'll initialize the vue-i18n instance by making sure to use our getUserPreferredLocale method to retrieve the default locale:

```
// file: src/localization/i18n.init.ts

import { createI18n } from 'vue-i18n'
import { getUserPreferredLocale } from './utils'

const defaultLocale = getUserPreferredLocale()

export const i18n = createI18n({
  locale: defaultLocale, // set locale
  fallbackLocale: defaultLocale, // set fallback locale
  silentFallbackWarn: true,
  silentMissingWarn: true,
  silentTranslationWarn: true
})
```

file useLocalization.ts

For our hook, let's start importing references to our **config**, **apiClient**, **ref/nextTick** from Vue, the i18n instance from our i18n.init code, and the newly created helpers from our utils.ts:

```
// file: src/localization/useLocalization.ts

import { config } from '@/config'
import { apiClient } from '../api-client'
import { ref, nextTick } from 'vue'

import { i18n } from './i18n.init'
import { getUserPreferredLocale, setUserPreferredLocale } from './utils'


...
```

Then get a reference to the localeStorageCache configuration and save the current default locale id in a local variable:

```
// file: src/localization/useLocalization.ts
...

// get a reference to the localStorageCache config
const localStorageConfig = config.localization.localStorageCache
// save the default localeId in a local variable:
const defaultLocaleId = getUserPreferredLocale()

...
```

Create a few reactive variables (code comment on each should be self-explanatory):

```
// file: src/localization/useLocalization.ts
...

// state
const isLoadingLocale = ref(true) // will set to true while a locale is being loaded
const currentLocale = ref(defaultLocaleId) // will be set to the current lcid value

...
```

Add an helper called **changeLocale** that will help us switch local but also load JSON translation data for a specific locale from an API and cache it into localStorage (so subsequent calls to this method will retrieve the JSON data from cache):

```
// file: src/localization/useLocalization.ts
...


// helper to change the current 18n locale
const changeLocale = async (lcid: string) => {
  // set our loading flag to true:
  isLoadingLocale.value = true

  // try to get it from locale storage
  // dynamic key we use to cache the actual locale JSON data in the browser local st\
orage
  const localeStorageKey = `lcid-data-${lcid}`

  const cacheEntryModel = {
    appVersion: -1,
    expiresAt: 0,
    json: ''
  }

  // retrieve JSON as string
  const cacheEntryStr = localStorage.getItem(localeStorageKey) || JSON.stringify(cac\
heEntryModel)
  // a variable to hold the parsed JSON data:
  let cacheEntry = cacheEntryModel

  // if localeStorage is enabled through config, then proced trying parsing the  cac\
heEntryStr
  if (localStorageConfig.enabled) {
    try {
      cacheEntry = JSON.parse(cacheEntryStr)
    } catch (e) {
      console.warn('error parsing data', cacheEntryStr)
    }
  }

  //console.log('cacheEntry?.expiresAt - Date.now()', cacheEntry?.expiresAt - Date.n\
ow())
  //console.log('typeof cacheEntry.json', typeof cacheEntry.json)

  // check if we have cacheEntry and if matches app version and also did not expire
  if (cacheEntry && cacheEntry.appVersion === config.global.version && cacheEntry.ex\
piresAt - Date.now() > 0) {
    // then switch locale by invoking i18n setLocaleMessage
    i18n.global.setLocaleMessage(lcid, <any>cacheEntry.json)
    // then switch locale using i18n
    i18n.global.locale = lcid
    // update tate
    currentLocale.value = lcid
  } else {
    console.log(`loading from api: lcid: "${lcid}"`)
    // retrieve data from API end point (or CDN etc)
    const localeData = await apiClient.localization.fetchTranslation('translation', \
lcid)
    // use the data returned y the API and pass it toi18n setLocaleMessage
    i18n.global.setLocaleMessage(lcid, localeData)
    // then switch locale using i18n
    i18n.global.locale = lcid
    // update tate
    currentLocale.value = lcid

    // update our cache
    const dt = new Date()
    if (localStorageConfig.enabled) {
      // calculate new expiration date
      const expiresAt = dt.setMinutes(dt.getMinutes() + Number(localStorageConfig.ex\
pirationInMinutes))
      // update local storage
      localStorage.setItem(
        localeStorageKey,
        JSON.stringify({
          appVersion: config.global.version,
          expiresAt: expiresAt,
          json: localeData
        })
      )
    }
  }

  // also save the user preference
  setUserPreferredLocale(lcid)

  // set our loading flag to false
  isLoadingLocale.value = false

  nextTick()
```

```
}

...
```

Finally we'll just export a **useLocalization** hook that has all we need, and enable us to easily consume the different locale translations in our components:

```
// file: src/localization/useLocalization.ts
...

// export all we need as a hook
export function useLocalization() {
  const availableLocales = config.localization.locales

  return {
    locales: availableLocales,
    isLoadingLocale,
    currentLocale,
    changeLocale,
    t: i18n.global.t // expose i18n global translation functoin
  }
}
```

file src/localization/index.ts (barrel file)

Just a barrel file to export our hook (in one of the next chapters will be exporting additional things like formatters here as well):

```
// file: src/localization/index.ts

export * from './useLocalization'
```

App.vue updates

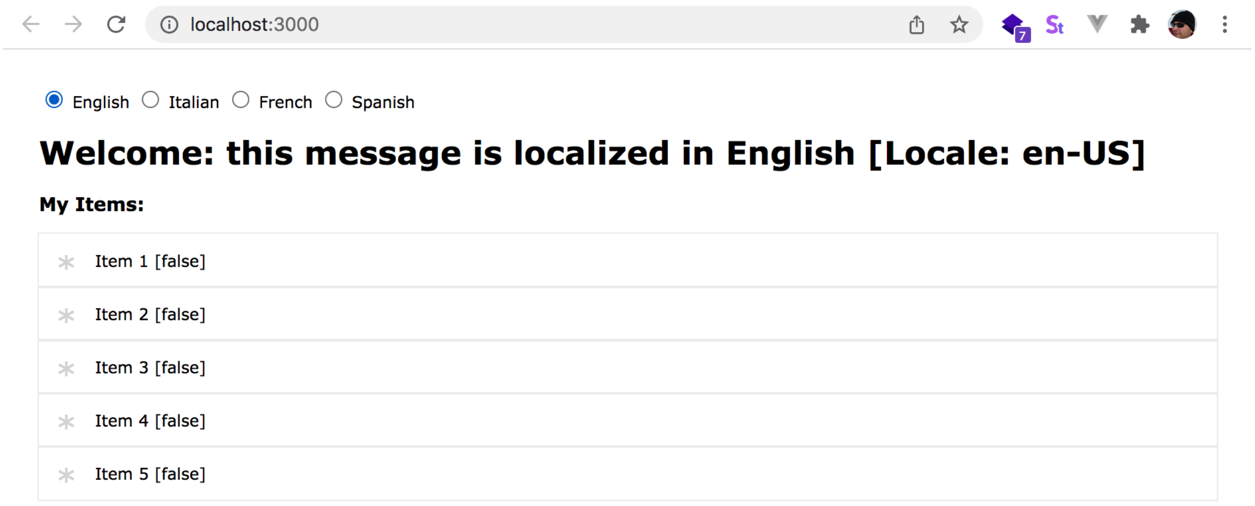
Let's now consume our **useLocalization** within the App.vue file by adding a quick way to change locale and also display the translated home welcome message:

```
// file: src/App.vue
<script setup lang="ts">
  // import onMounted from Vue
  import { onMounted } from 'vue' <!-- add this line

  ...

  // import a reference to useLocalization
  import { useLocalization } from '@/localization' // <-- add this line

  ...

  // get what we need from useLocalization:
  const { t, locales, currentLocale, isLoadingLocale, changeLocale } = useLocalizati\
on()

  // an event handler from changing the locale from our locale-selector
  const onLocaleClick = (lcid: string) => {
    changeLocale(lcid)
  }

  // invoke changeLocale also when component is mounted
  onMounted(() => {
    changeLocale(currentLocale.value)
  })
</script>

<template>
  <div class="home">
    <div class="locale-selector">
      <div v-if="isLoadingLocale">Loading locale data...</div>
      <label
        v-for="item in locales"
        :key="item.key"
        :for="`radio-locale-${item.key}`"
        class="cursor-pointer"
        @click="onLocaleClick(item.key)"
      >
        <input
          type="radio"
          :id="`radio-locale-${item.key}`"
          :radioGroup="currentLocale"
          name="locale"
          :value="item.key"
          :checked="currentLocale === item.key"
        />
        {{ t(`locale.selector.${item.key}`) }}
      </label>
    </div>
    <h3>{{ t('home.welcome') }} [{{ currentLocale }}]</h3>
    <ItemsView />
  </div>
</template>
```

You can similarly update the ItemsList.component.vue code as well to use the translations. I'll let you do that on your own. You can always refer tot he github repo if you need help.

Browser

Now run the app and you will see something like this:



localhost:3000

[✓ English] Italian French Spanish

Welcome: this message is localized in English [Locale: en-US]

My Items:

Item 1 [false]

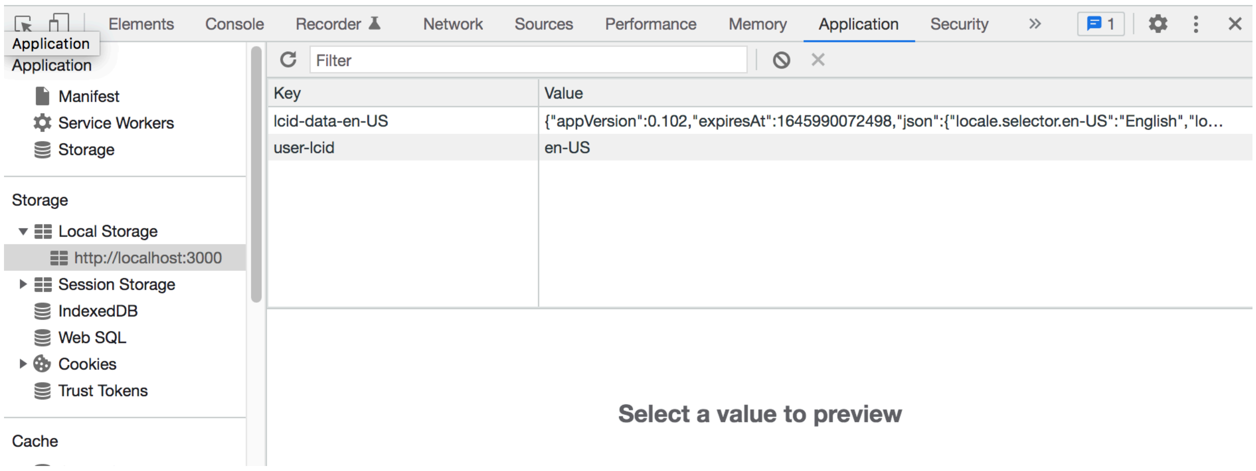
Item 2 [false]

Item 3 [false]

Item 4 [false]

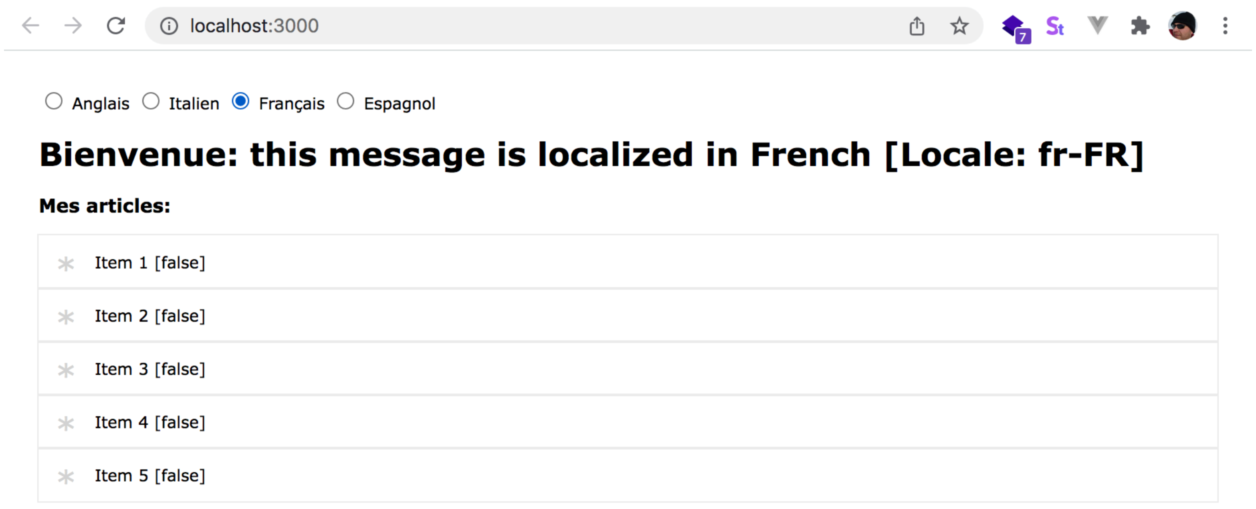
Item 5 [false]

Now right-click and select inspect to open the Chrome dev tools, then select the Application tab, then Local storage > http://localhost:3000 note how our code has cached the en-US JSON data in localStorage and saved the current locale id under user-lcid:



Key	Value
lcid-data-en-US	{"appVersion":0.102,"expiresAt":1645990072498,"json":{"locale.selector.en-US":"English","lo…
user-lcid	en-US

If you select a different locale, i.e. French, it should display the translated labels:



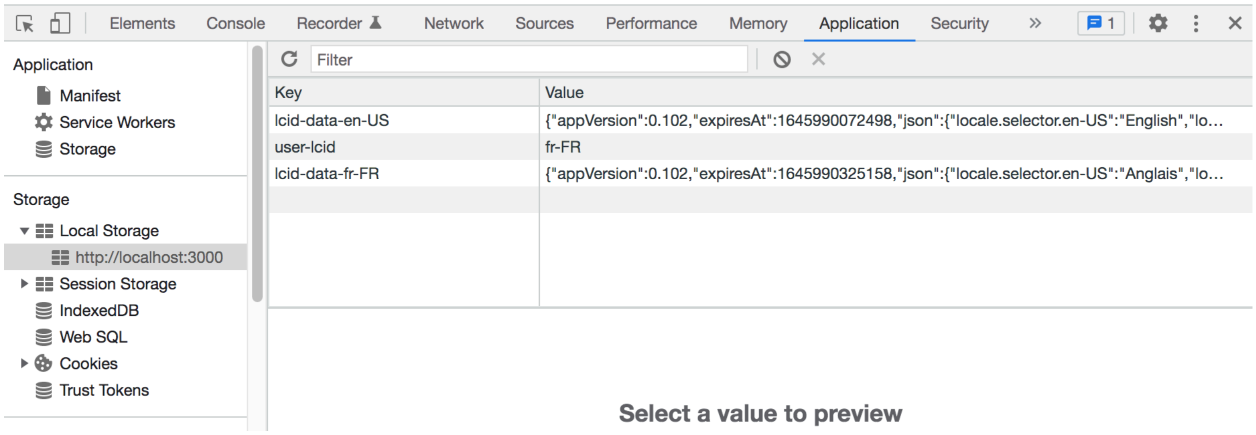
( ) Anglais ( ) Italien [✓ Français] ( ) Espagnol

Bienvenue: this message is localized in French [Locale: fr-FR]

Mes articles:

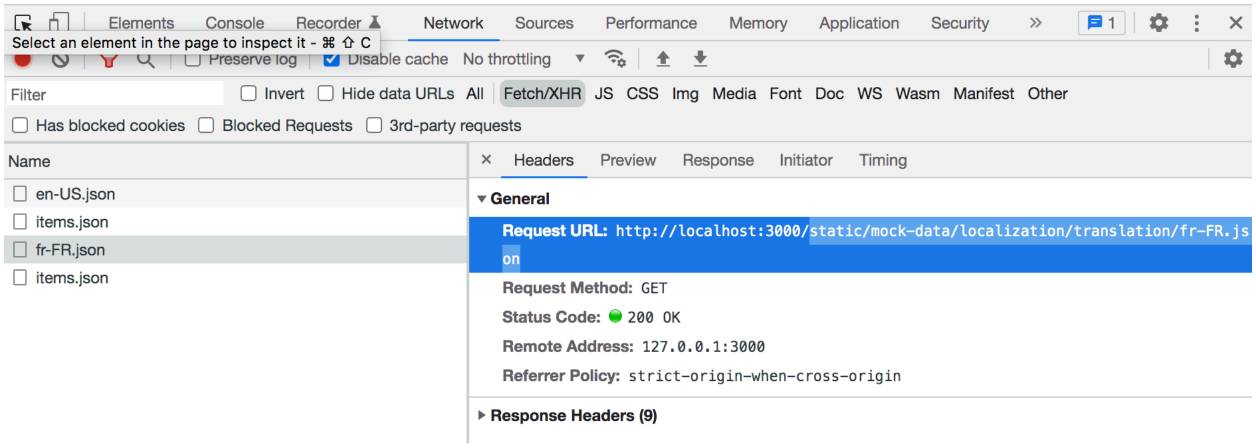
Item 1 [false]
Item 2 [false]
Item 3 [false]
Item 4 [false]
Item 5 [false]

Now in the Local Storage inspector in the Chrome console, again notice how the fr-FR locale data has been also saved to localeStorage and the user preferred locale saved under user-lcid has been also updated:



Key	Value
lcid-data-en-US	{"appVersion":0.102,"expiresAt":1645990072498,"json":{"locale.selector.en-US":"English","lo…
user-lcid	fr-FR
lcid-data-fr-FR	{"appVersion":0.102,"expiresAt":1645990325158,"json":{"locale.selector.en-US":"Anglais","lo…

If you switch on the Network tab in the Chrome/Edge console, you can see how our API client has loaded the data from static/mock-data/localization/translation/fr-FR.json:



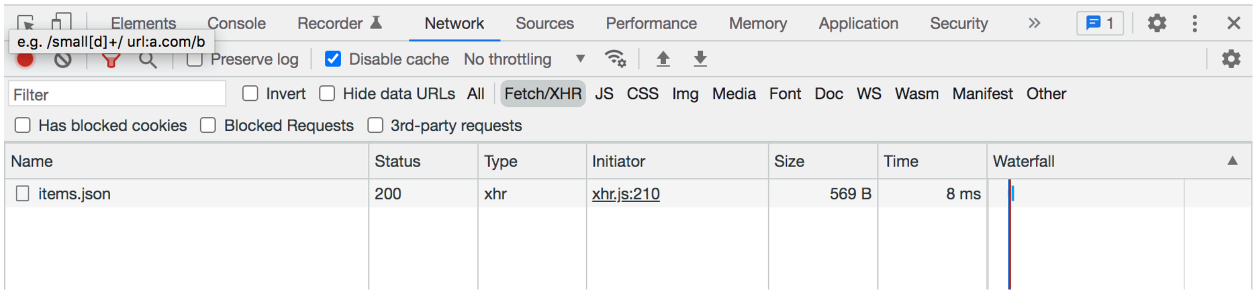
General

Request URL: http://localhost:3000/static/mock-data/localization/translation/fr-FR.json
Request Method: GET
Status Code: 200 OK
Remote Address: 127.0.0.1:3000
Referrer Policy: strict-origin-when-cross-origin

Response Headers (9)

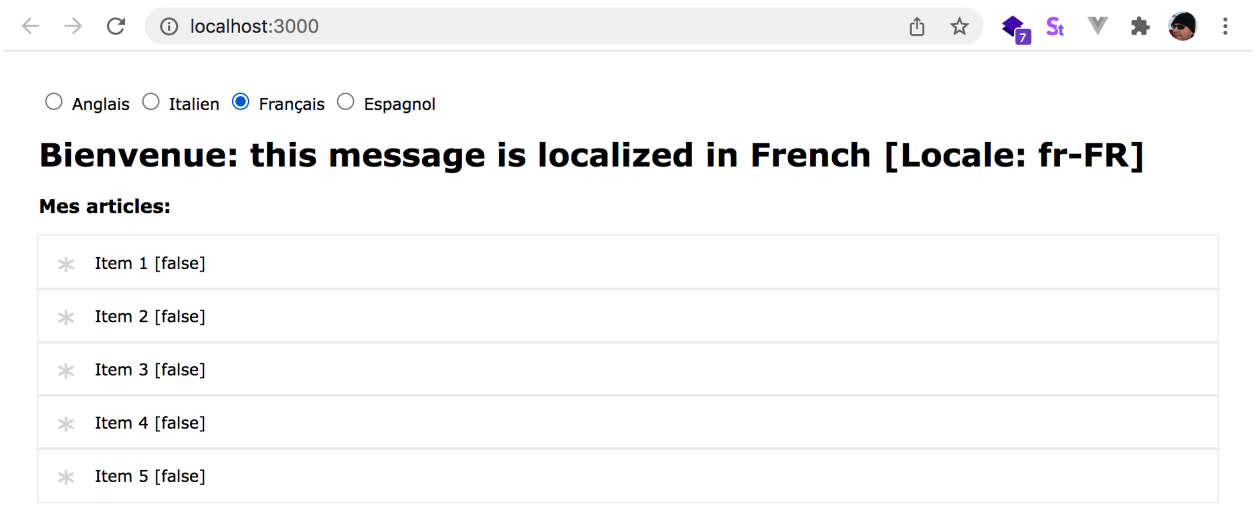
Test that our caching logic is working:

To test that our code will load the user-preferred locale from local storage, along with the save JSON translation data, clear the Network tab and then refresh the Chrome tab with F5. Note how the network tab will NOT show a call to the static JSON file this time because our code is actually loading the data from localStorage this time:



Name	Status	Type	Initiator	Size	Time	Waterfall
items.json	200	xhr	xhr.js:210	569 B	8 ms	

Also note how the selected locale is French:



Anglais Italien [✓ Français] Espagnol

Bienvenue: this message is localized in French [Locale: fr-FR]

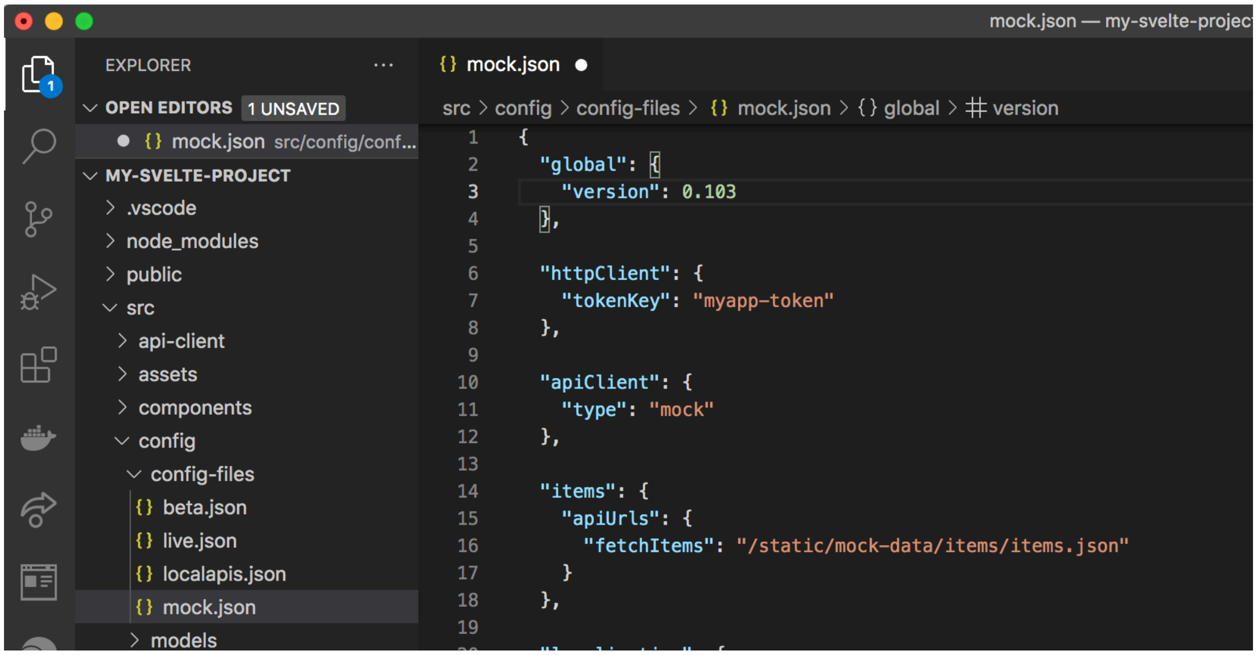
Mes articles:

- Item 1 [false]
- Item 2 [false]
- Item 3 [false]
- Item 4 [false]
- Item 5 [false]

Important

Remember that if you add additional keys to your translation files, or modify the data in any way, you'll have to either clear your local storage (if you are just playing with things locally), or increment that Application Configuration version in the config files (global section). The localStorage cache will also expire eventually based on the config.localization.localStorageCache.expirationInMinutes value.

For a new deploy, incrementing the version will make sure that the logic we added in our useLocalization code will ignore the currently cached data from localStorage and re-load fresh data through the AP. Here for example I increase it from 0.102 to 0.103:



```
{
  "global": {
    "version": 0.103
  },

  "httpClient": {
    "tokenKey": "myapp-token"
  },

  "apiClient": {
    "type": "mock"
  },

  "items": {
    "apiUrls": {
      "fetchItems": "/static/mock-data/items/items.json"
    }
  },
```

Note: you dont have to use decimals for your version. You are free to just use integers like 1, 2, etc.

Chapter 10 Recap

What We Learned

- How to add the **i18next** and **vue-i18n** plugins to our application
- How to wrap code for vue-i18n and lazy-loading of JSON translation data through our API client
- How to cache translation JSON data into localStorage with versioning and expiration
- How to drive available locales through configuration
- How to use multiple **locale** settings for text translation in order to localize our UI labels
- How to switch to different locales

Observations

- We did not add unit tests around switching locale through the radio buttons
- We did not create a component for switching locale

Improvements

- Add additional unit tests
- Extract the code that loops through each locale and adds a radio button (div with className set to locale-selector) into its own component and add unit tests against this. Maybe in your application requirements this has to be a dropdown instead of a radio group, so it is up to you how you will implement this.

Chapter 11 - Localization and Internationalization - Number and DateTime Formatters

In this chapter we are going to expand our support for localization by adding Number and DateTime value formatters. We'll leverage the **Intl API**[36] which is supported by all major web browsers.

We'll build a hook called useFormatters that will make it easier to consume different kind of formatters based on the currently selected locale.

Note: this is the same code from a plugin I published here and you could use it in other apps in the future without coding it yourself if you prefer: @builtwithjavascript/formatters

Note also that the code in this chapter is re-usable in any framework, not just Vue, as it does not have any dependency on Vue.

Directory localization/formatters

Start by creating a directory under **src/localization** called **formatters**. Inside this directory create the following files:

- useDateTimeFormatters.ts
- useNumberFormatters.ts
- index.ts

We'll create the 2 main hooks **useDateTimeFormatters** and **useNumberFormatters** and then just export them together as **useFormatters** from the index.ts file.

File: useDateTimeFormatters.ts

For date-time formatters we'll wrap around **Intl.DateTimeFormat** and make it easier to consume it. We'll cache each instance of **Intl.DateTimeFormat** by **localeId** and the different options to avoid keeping re-instantiating it everytime (this is for performance reasons).

[36]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl

For this reason, we need to first add a method that return a valid and unique cache key that uses localeId and the different options that might be passed when consuming it. The cache key will be a string in the format [localeId]-[dateStyle]-[timeStyle]:

```
// file: src/formatters/useDateTimeFormatters.ts

export type DayNameFormatType = 'long' | 'short' | 'narrow' | undefined
export type MonthNameFormatType = 'long' | 'short' | 'narrow' | 'numeric' | '2-digit\
' | undefined

const defaultDateStyle = 'short' // 'full', 'long', 'medium', 'short'

// helper to calculate the cache key for the datetime Intl.DateTimeFormat instances
export const getDateTimeFormattersCacheKey = (params: { lcid: string; dateStyle?: st\
ring; timeStyle?: string }) => {
  let { lcid, dateStyle, timeStyle } = params
  dateStyle = (dateStyle || defaultDateStyle).trim().toLowerCase()
  timeStyle = (timeStyle || '').trim().toLowerCase()

  let cacheKey = `${lcid}-${dateStyle}`
  if (timeStyle.length) {
    cacheKey = `${cacheKey}-${timeStyle}`
  }
  return cacheKey.trim().toLowerCase()
}

...
```

We can then add the code for the **useDateTimeFormatters** hook. This will return an object with 3 methods: **datetime**, **dayNames**, **monthNames**. Here is the preliminary implementation with just **datetime**:

```
// file: src/formatters/useDateTimeFormatters.ts

...

// hook to export the datetime, dayNames, monthNames utils
export const useDateTimeFormatters = (localeId: string) => {
  const _lcid = localeId
  const _cache = new Map<string, Intl.DateTimeFormat>()

  return {
    dateTime: (dateStyle?: string, timeStyle?: string) => {
      dateStyle = (dateStyle || defaultDateStyle).trim().toLowerCase()
      timeStyle = (timeStyle || '').trim().toLowerCase()

      const cacheKey = getDateTimeFormattersCacheKey({
        lcid: _lcid,
        dateStyle,
        timeStyle
      })

      if (!_cache.has(cacheKey)) {
        // if not in our cache yet, create it and cache it
        let options: { dateStyle?: string; timeStyle?: string } = {}
        if (dateStyle.length) {
          options.dateStyle = dateStyle
        }
        if (timeStyle.length) {
          options.timeStyle = timeStyle
        }
        // cache instance
        const instance = new Intl.DateTimeFormat(_lcid, options as Intl.DateTimeForm\
atOptions)
        _cache.set(cacheKey, instance)
      }
      // return instance from cache
      return _cache.get(cacheKey) as Intl.DateTimeFormat
    },

    // ... we'll be adding also dayNames and monthNames here shortly

  }
}
```

Now let’s also return dayNames and monthNames:

```
// file: src/formatters/useDateTimeFormatters.ts

...

// hook to export the datetime, dayNames, monthNames utils
export const useDateTimeFormatters = (localeId: string) => {
  const _lcid = localeId
  const _cache = new Map<string, Intl.DateTimeFormat>()

  // add these two to cache also dayName and monthNames
  const _cacheDayNames = new Map<string, { id: number; name: string }[]>()
  const _cacheMonthNames = new Map<string, { id: number; name: string }[]>()

  return {
    dateTime: (dateStyle?: string, timeStyle?: string) => {
      ...
    },

    dayNames: (format: DayNameFormatType = 'long') => {
      if (!_cacheDayNames.has(format)) {
        // if not in our cache yet, create it and cache it
        const items: { id: number; name: string }[] = []
        for (let i = 0; i < 7; i++) {
                                        // start from March 1st 1970 which is a Sunday
          // calculate day and pad string start with zero
          const strDay = (i + 1).toString().padStart(2, '0')
          const date = new Date(`1970-03-${ strDay }T00:00:00.000Z`)
          const name = date.toLocaleString(_lcid, { weekday: format, timeZone: 'UTC'\
 })
          items.push({ id: i, name })
        }
        _cacheDayNames.set(format, items)
      }
      // return cached items
      return _cacheDayNames.get(format) as { id: number; name: string }[]
    },

    monthNames: (format: MonthNameFormatType = 'long') => {
      if (!_cacheMonthNames.has(format)) {
        // if not in our cache yet, create it and cache it
        const items: { id: number; name: string }[] = []
        for (let i = 0; i < 12; i++) {
          // calculate month and pad string start with zero
          const strMonth = (i + 1).toString().padStart(2, '0')
          const date = new Date(`1970-${ strMonth }-01T00:00:00.000Z`)
          const name = date.toLocaleString(_lcid, { month: format, timeZone: 'UTC' })
          items.push({ id: i, name })
        }
        _cacheMonthNames.set(format, items)
      }
      // return cached items
      return _cacheMonthNames.get(format) as { id: number; name: string }[]
    }

  }
}
```

As you can see in the code above we leverage **date.toLocaleString** to get either the day or month name. We use a calculated date from March 1st 1970 (which is a Sunday) to get the correct weekday name, and from January 1st 1970 to get the correct month name (irrelevant of the current user time zone).

File: useNumberFormatters.ts

For number formatters, similar to what we did for datetime in the previous section, we'll wrap around **Intl.NumberFormat** . We'll cache each instance of **Intl.NumberFormat** by **localeId** and the different options to avoid keeping re-instantiating it everytime (again for performance reasons).

Similaor to the datetime formatters hook, we'll need a function here as well that calculate the cache key dynamically. This is a bit more complex as it takes into account a few more parameters:

```
// file: src/formatters/useNumberFormatters.ts

const defaultcurrencyDisplay = 'symbol' // 'symbol', 'narrowSymbol', 'code', 'name'

// helper to calculate the cache key for the datetime Intl.NumberFormat instances
export const getNumberFormattersCacheKey = (params: {
  lcid: string
  style?: string
  currency?: string
  currencyDisplay?: string
  minimumFractionDigits: number
  maximumFractionDigits: number
}) => {
  let { lcid, style, currency, currencyDisplay, minimumFractionDigits, maximumFracti\
onDigits } = params
  style = (style || 'decimal').trim().toLowerCase()
  currency = (currency || '').trim()
  currencyDisplay = (currencyDisplay || defaultcurrencyDisplay).trim()

  let cacheKey = `${lcid}-${style}`
  if (currency.length > 0) {
    cacheKey = `${cacheKey}-${currency}`
    if (currencyDisplay.length > 0) {
      cacheKey = `${cacheKey}-${currencyDisplay}`
    }
  }
  cacheKey = `${cacheKey}-${minimumFractionDigits}-${maximumFractionDigits}`.trim().\
toLowerCase()
  return cacheKey
}

...
```

We can then add the code for the **useNumberFormatters** hook. This will return an object with 4 methods: **whole**, **decimal**, **currency**, **percent**. Here we start by implementing a private method called **_privateGetFormatter** that will use to avoid code duplication. This method also contains the logic to retrieve/set the instance into the cache:

```
// file: src/formatters/useNumberFormatters.ts

...

// hook to export the various number formatters utils
export const useNumberFormatters = (localeId: string) => {
  const _lcid = localeId
  const _cache = new Map<string, Intl.NumberFormat>()

  const _privateGetFormatter = (params: {
    style?: string
    currency?: string
    currencyDisplay?: string
    minimumFractionDigits: number
    maximumFractionDigits: number
  }) => {
    let { style, currency, currencyDisplay, minimumFractionDigits, maximumFractionDi\
gits } = params

    style = (style || 'decimal').trim().toLowerCase()
    currency = (currency || '').trim()
    currencyDisplay = (currencyDisplay || defaultcurrencyDisplay).trim()

    let cacheKey = getNumberFormattersCacheKey({
      lcid: _lcid,
      style,
      currency,
      currencyDisplay,
      minimumFractionDigits,
      maximumFractionDigits
    })

    if (!_cache.has(cacheKey)) {
      // if not in our cache yet, create it and cache it
      let options: Intl.NumberFormatOptions = {
        style,
        minimumFractionDigits,
        maximumFractionDigits
      }
      if (currency.length > 0) {
        options.currency = currency
        if (currencyDisplay.length > 0) {
          options.currencyDisplay = currencyDisplay
        }
      }
      // cache instance
      const instance = new Intl.NumberFormat(_lcid, options)
      _cache.set(cacheKey, instance)
    }
    // return instance from cache
    return _cache.get(cacheKey) as Intl.NumberFormat
  }

 ...
```

Then we can add the code to export our 4 utility methods by using the private method to construct each instance with the various options:

```
// file: src/formatters/useNumberFormatters.ts

...

  return {
    whole: () => {
      return _privateGetFormatter({
        style: 'decimal',
        minimumFractionDigits: 0,
        maximumFractionDigits: 0
      })
    },
    decimal: (minimumFractionDigits: number = 0, maximumFractionDigits: number = 2) \
=> {
      return _privateGetFormatter({
        style: 'decimal',
        minimumFractionDigits,
        maximumFractionDigits
      })
    },
    currency: (
      currency: string,
      currencyDisplay?: string,
      minimumFractionDigits: number = 0,
      maximumFractionDigits: number = 2
    ) => {
      return _privateGetFormatter({
        style: 'currency',
        currency,
        currencyDisplay,
        minimumFractionDigits,
        maximumFractionDigits
      })
    },
    percent: (minimumFractionDigits: number = 0, maximumFractionDigits: number = 2) \
=> {
      return _privateGetFormatter({
        style: 'percent',
        minimumFractionDigits,
        maximumFractionDigits
      })
    },
    unescapeResult(result: string) {
      return (result || '').replace(/\xa0/g, ' ').replace(/\u202f/g, ' ')
    }
  }
}
```

File: index.ts

For convenience, here we just export a global hook called **useFormatters**:

```
// file: src/formatters/index.ts

import { useDateTimeFormatters } from './useDateTimeFormatters'
import { useNumberFormatters } from './useNumberFormatters'

export const useFormatters = () => {
  return {
    useDateTimeFormatters,
    useNumberFormatters
  }
}
```

Note: this step is optional. You could just import individually either useDateTimeFormatters or useNumberFormatters when consuming them.

Later, when we need to consume our formatters, we can just import them as:

```
  import {
    useLocalization,
    useDateTimeFormatters,
    useNumberFormatters
  } from '@/localization/formatters'
```

Component DebugFormatters.component.vue

Let's now create a component that we can just use to visually debug the output of the formatters. Here will also use the **useLocalization** hook to get the **currentLocale**. Then we'll have some computed properties that will return the correct formatters based on the currentLocale:

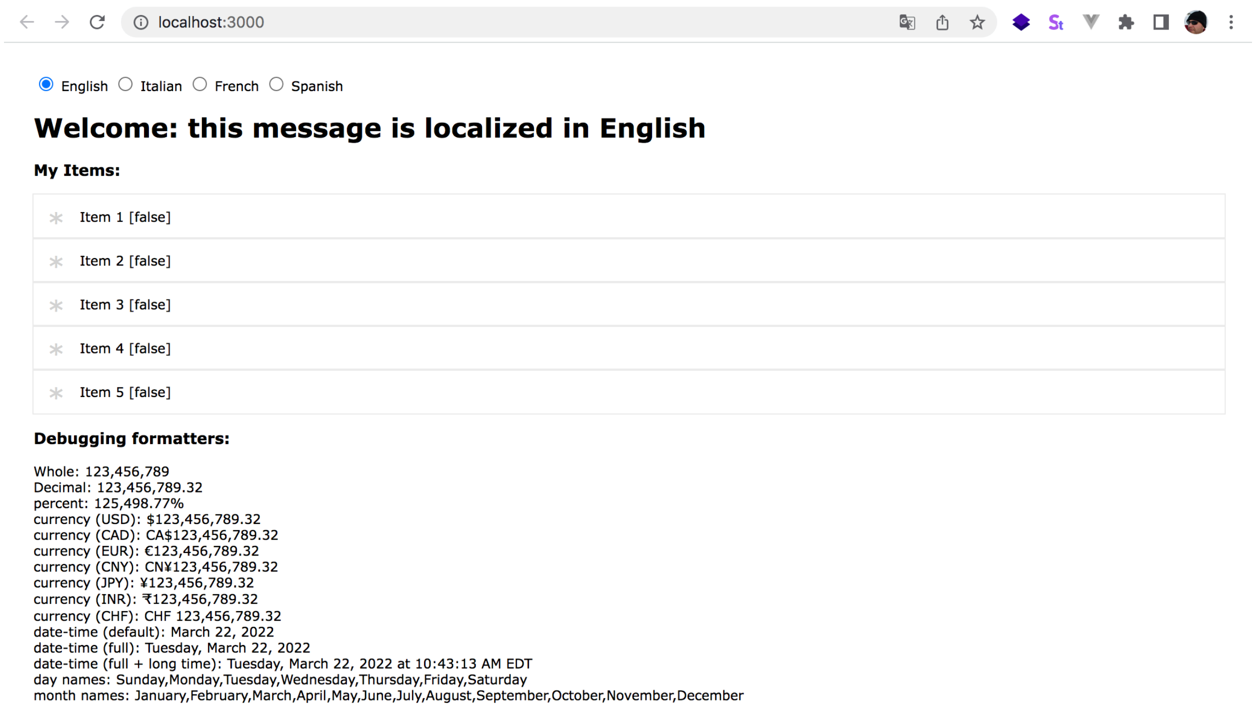
```
// file: src/components/shared/DebugFormatters.component.vue

<template>
  <div v-if="show">
    <h3>Debugging formatters:</h3>
    <div>Whole: {{ wholeNumberFormatter().format(123456789.321654) }}</div>
    <div>Decimal: {{ decimalNumberFormatter().format(123456789.321654) }}</div>
    <div>percent: {{ percentNumberFormatter().format(1254.987654) }}</div>

    <div>currency (USD): {{ currencyNumberFormatter().format(123456789.321654) }}</d\
iv>
    <div>currency (CAD): {{ currencyNumberFormatter('CAD').format(123456789.321654) \
}}</div>
    <div>currency (EUR): {{ currencyNumberFormatter('EUR').format(123456789.321654) \
}}</div>
    <div>currency (CNY): {{ currencyNumberFormatter('CNY').format(123456789.321654) \
}}</div>
    <div>currency (JPY): {{ currencyNumberFormatter('JPY').format(123456789.321654) \
}}</div>
    <div>currency (INR): {{ currencyNumberFormatter('INR').format(123456789.321654) \
}}</div>
    <div>currency (CHF): {{ currencyNumberFormatter('CHF').format(123456789.321654) \
}}</div>

    <div>date-time (default): {{ dateTime().format(new Date()) }}</div>
    <div>date-time (full): {{ dateTime('full').format(new Date()) }}</div>
    <div>date-time (full + long time): {{ dateTime('full', 'long').format(new Date()\
) }}</div>

    <div>day names: {{ dayNames() }}</div>
    <div>month names: {{ monthNames() }}</div>
  </div>
</template>

<script setup lang="ts">
  import { defineProps, computed } from 'vue'
  // import a reference to useLocalization and useFormatters
  import { useLocalization, useDateTimeFormatters, useNumberFormatters } from '@/loc\
alization'

  // define our props:
  defineProps<{ show: boolean }>()
  // get what we need from useLocalization:
  const { currentLocale } = useLocalization()

  // return formatters based on selected locale
  const dateTimeFormatters = computed(() => {
    return useDateTimeFormatters(currentLocale.value)
  })
  const numberFormatters = computed(() => {
    return useNumberFormatters(currentLocale.value)
  })

  const wholeNumberFormatter = () => {
    return numberFormatters.value.whole()
  }
  const decimalNumberFormatter = () => {
    return numberFormatters.value.decimal()
  }
  const currencyNumberFormatter = (currency: string = 'USD') => {
    return numberFormatters.value.currency(currency)
  }
  const percentNumberFormatter = () => {
    return numberFormatters.value.percent()
  }

  const dateTime = (dateStyle: string = 'long', timeStyle: string = '') => {
    return dateTimeFormatters.value.dateTime(dateStyle, timeStyle)
  }
  const dayNames = () => {
    return dateTimeFormatters.value.dayNames() //.map((o) => o.name)
  }
  const monthNames = () => {
    return dateTimeFormatters.value.monthNames() //.map((o) => o.name)
  }
</script>
```

Updates to App.vue

Now we can import the DebugFormatters component and render it within our App.vue to quickly visually debug that the formatters are working as expected:

```
<script setup lang="ts">

  ...

  // import a reference to the DebugFormatters component
  import DebugFormatters from '@/components/shared/DebugFormatters.component.vue'

  ...
</script>

<template>
  <div class="home">
    <LocaleSelector />
    <h3>{{ t('home.welcome') }} [{{ currentLocale }}]</h3>
    <ItemsView />
    <DebugFormatters :show="true" />
  </div>
</template>
```

If you now run the app you will see the DebugFormatters rendering information at the bottom of the page:



Welcome: this message is localized in English

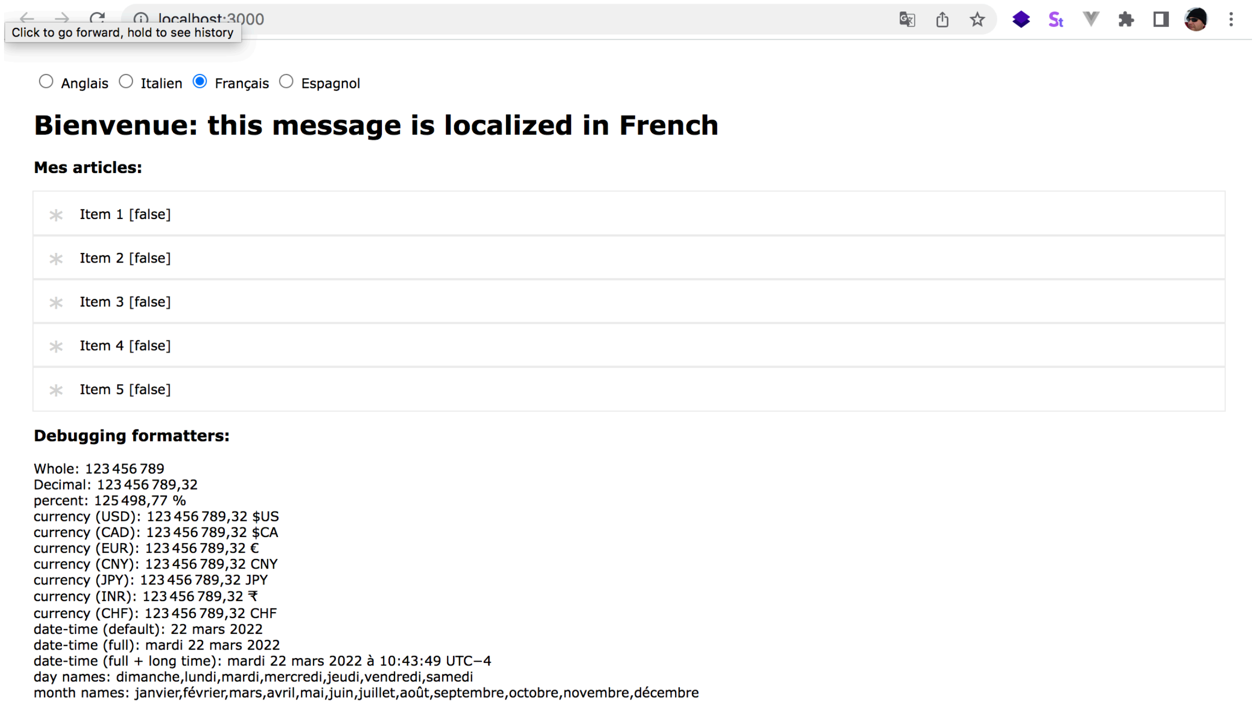
My Items:

Item 1 [false]
Item 2 [false]
Item 3 [false]
Item 4 [false]
Item 5 [false]

Debugging formatters:

```
Whole: 123,456,789
Decimal: 123,456,789.32
percent: 125,498.77%
currency (USD): $123,456,789.32
currency (CAD): CA$123,456,789.32
currency (EUR): €123,456,789.32
currency (CNY): CN¥123,456,789.32
currency (JPY): ¥123,456,789.32
currency (INR): ₹123,456,789.32
currency (CHF): CHF 123,456,789.32
date-time (default): March 22, 2022
date-time (full): Tuesday, March 22, 2022
date-time (full + long time): Tuesday, March 22, 2022 at 10:43:13 AM EDT
day names: Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
month names: January,February,March,April,May,June,July,August,September,October,November,December
```

And of course, if you select a different locale (i.e. French) you'll see the formatters displaying the value as per the current locale culture:



Bienvenue: this message is localized in French

Mes articles:

Item 1 [false]
Item 2 [false]
Item 3 [false]
Item 4 [false]
Item 5 [false]

Debugging formatters:

```
Whole: 123 456 789
Decimal: 123 456 789,32
percent: 125 498,77 %
currency (USD): 123 456 789,32 $US
currency (CAD): 123 456 789,32 $CA
currency (EUR): 123 456 789,32 €
currency (CNY): 123 456 789,32 CNY
currency (JPY): 123 456 789,32 JPY
currency (INR): 123 456 789,32 ₹
currency (CHF): 123 456 789,32 CHF
date-time (default): 22 mars 2022
date-time (full): mardi 22 mars 2022
date-time (full + long time): mardi 22 mars 2022 à 10:43:49 UTC−4
day names: dimanche,lundi,mardi,mercredi,jeudi,vendredi,samedi
month names: janvier,février,mars,avril,mai,juin,juillet,août,septembre,octobre,novembre,décembre
```

Chapter 11 Recap

What We Learned

- How to add code that wraps around Intl API **DateTimeFormat** and **NumberFormat**
- How to format values according to the current locale using the formatters we created

Observations

- We did not add unit tests against our formatters

Improvements

- Add unit tests again the formatters hooks

Chapter 12 - Adding Tailwind CSS

Going forward, we are going to use **Tailwind CSS**[37] as it makes it so easy to design components without having to mess with the CSS directly. We would eventually need to remove also the older CSS we wrote during the previous chapter, but there is no harm for now in leaving that there. But for our primitives library and new higher-level components, it will be written exclusively using Tailwind CSS.

To add TailwindCSS to our existing project execute this command. This will install the latest tailwindcss, postcss and autoprefixer npm packages:

```
npm install -D tailwindcss@latest postcss@latest autoprefixer@latest
```

Then let's init the Tailwind CSS configuration files. The following command will create the preliminary **tailwind.config.js** and **postcss.config.js**:

```
npx tailwindcss init -p
```

Opent the file **tailwind.config.js** and verify it has been created with this content:

```
//file: src/tailwind.config.js
module.exports = {
  content: [
    "./index.html", /* you might have to add this */
    "./src/**/*.{vue,js,ts,jsx,tsx}", /* you might have to add this */
  ],
  theme: {
    extend: {},
  },
  plugins: [
    // ...
  ]
};
```

also verify that **postcss.config.js** has been created with this:

[37]Tailwind CSS Official Website

```
// file: src/postcss.config.js
module.exports = {
  plugins: {
    tailwindcss: {},
    autoprefixer: {},
  },
}
```

Then, under src, create a new directory called **tailwind** and move the **app.css** file into it. The final path of this will be **src/tailwind/app.css**. Open the file and make sure contains this code:

```
/* file: src/tailwind/app.css */
@import 'tailwindcss/base';
@import 'tailwindcss/components';
@import 'tailwindcss/utilities';

@import './other.css'; /* note: we create this file in one of the next steps below */
```

Within the **main.ts** file: remove any existing references to previous .css files and import a reference to the new **tailwind/app.css** file:

```
// file: src/main.ts
import { createApp } from 'vue'
// import tailwind main css file
import './tailwind/app.css' // <-- add this line
import App from './App.vue'

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

Create the file tailwind/other.css and move the old css code from the App.vue <style> section of the into it:

```
// file: src/tailwind/other.css

.home {
  padding: 20px;
  font-family: Verdana, Geneva, Tahoma, sans-serif;
  font-size: 12px;
}
```

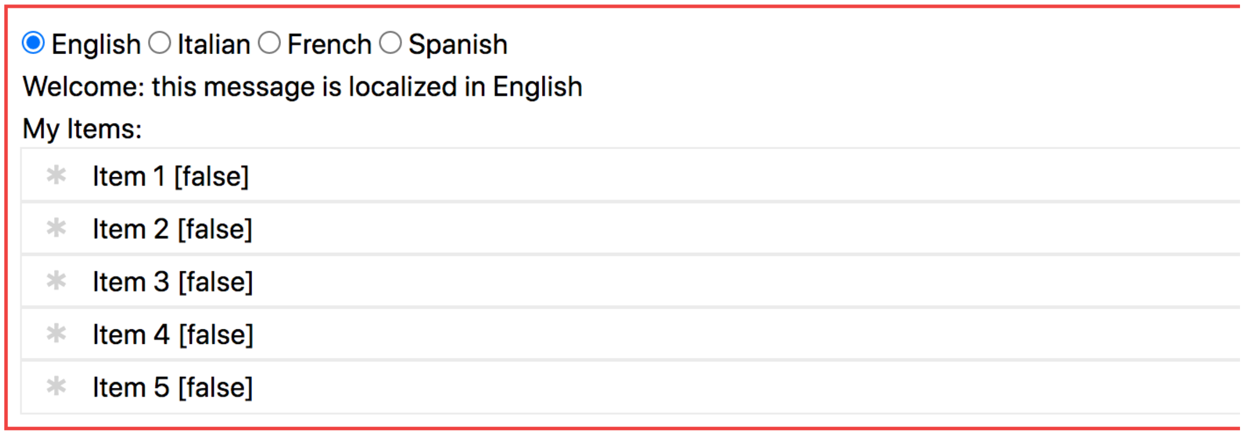
and remove the <style> block found at the end of the App.vue code:

```
...

// remove the following <style> block:
~~<style>~~
~~  .home {~~
~~    padding: 20px;~~
~~    font-family: Verdana, Geneva, Tahoma, sans-serif;~~
~~    font-size: 12px;~~
~~  }~~
~~</style>~~
```

Finally, to test that Tailwind CSS has been added and it is working, add this classes to the first div child of the <div> element within the App.vue file:

```
// file: App.vue

...

<template>
  <div class="home m-2 p-2 border-2 border-red-500">

...
```

To confirm that Tailwind CSS is being add correctly, run the application and verify that it renders like this:



● English ○ Italian ○ French ○ Spanish

Welcome: this message is localized in English

My Items:

Item 1 [false]

Item 2 [false]

Item 3 [false]

Item 4 [false]

Item 5 [false]

Chapter 12 Recap

What We Learned

- We learned how to add Tailwind CSS to our existing project

Observations

- We did not talk about how to add support for something like Sass/Scss

Based on these observations, there are a few improvements that can be done:

Improvements

- You could learn how to add Sass/Scss if you do not want to use Tailwind CSS

Chapter 13 - Intro to Primitives

This chapter covers concepts on how to write and organize the most primitive components (i.e. Inputs, Buttons etc) in order to create a foundation for higher-level and more complex components.

Atomic Design and Similar Approaches

The way you can think of and organize your components might follow one or more methodologies. One methodology that has seen an increase in adoption recently is **Atomic Design** originally introduced by **Brad Frost**[38]. The great **Alba Silvente**[39] has also a terrific post about this that I strongly recommend you check out. You are free to follow this or other methodologies either strictly or more losely, as well as chose to implement your own or even use a mix of ideas from different ones.

In my personal and pragmatical way, I found over the years that all I really need is a **foundation** of the most **primitive** elements liek buttons/textboxes/dropdowns/etc. These primitives should be as simple as possible, even though in some cases the might contain quite a bit of logic to determine how they render. In my world, **primitives** are more or less the same as the **Atoms** in Atomic Design.

Then, you can build your higher level components by "composing" them from the **primitives**.

This is what I'll be describing in this chapter. We'll build a collection of primitives that are simply Buttons, Inputs and similar and see what better strategies we can use there to reduce the amount of code we have to write and maintain. We'll then explore in the next chapters how to build higher-level components from these.

Conventions

One of the convention we will follow is to put all our primitive components under the directory **src/components/primitives**

[38] *Brad Frost - Atomic Design https://bradfrost.com/blog/post/atomic-web-design/*

[39] *Alba Silvente - How to structure a Vue.js app using Atomic Design and TailwindCSS https://vuedose.tips/how-to-structure-a-vue-js-app-using-atomic-design-and-tailwindcss*

Within this directory we'll have sub-directories (folders) that leep the components organized by category. I.e. buttons will be under **src/components/primitives/buttons**, inputs will be under **src/components/primitives/inputs** etc.

We'll follow also a naming convention where each .vue file that represents a primitive will start with the **El** prefix. I.e. **ElText.vue**, **ElIcon.vue**, etc. In this case El is for "element". You are of course free to decide your own naming convention. But I strongly suggest using some kind of prefix to more quickly identify a **primitive** by just looking at its file name when it is open in your editor.

Create the following 5 sub-directories to get started:

- **src/components/primitives/buttons**
- **src/components/primitives/toggles**
- **src/components/primitives/text**
- **src/components/primitives/modals**
- **src/components/primitives/inputs**
- **src/components/primitives/icons**

General Strategies

One of the things we are going to consistently need in each **primitive** is the main CSS class property. Often, this will have to be a computed property that returns the appropriate value based on other conditions. For example, a **Button** might have to render with an additional "**disabled**" CSS class if its **disabled** property is true.

For consistency, every time a primitive needs a dynamic CSS class, we'll add a computed property called **cssClass** that will return the appropriate value based on various conditions.

Here is a code example for an hypotethical **Button** component:

```
<script setup lang="ts">
  // a computed property to return a different css class based on the selected value
  const cssClass = computed((): string => {
    // here we concatenate the default CSS with 'disabled' only if disabled is true
    const defaultClasses = 'p-6' // in TailwindCSS this means we want a padding of 6
    return `${ defaultClasses } ${ this.disabled ? 'disabled' : '' }`.trim()

    // alternatively you could use an array that is initialized with
    // the default CSS, and if disabled is true, then add 'disabled'
    // and return the result by joining the array with space as the separator
    // (I usually feavor this approach especially when there
    // is more than one check and additional logic)
    const cssClasses = ['p-6']
    if (this.disabled) {
      cssClasses.push('disabled')
    }
    return cssClasses.join(' ')
  })
</script>

<template>
  <button type="button" :class="cssClass">
    <span class="name">{{ label }}</span>
  </button>
</template>
```

Text Elements

Let's start creating one element for each group as a starting point and then we'll keep building more elements from there.

Create a file called **ElText.vue** under directory **src/components/primitives/text**.

For the code, use the following:

```
// file: src/components/primitives/text/ElText.vue
<script setup lang="ts">
  import { computed } from 'vue'

  interface Props {
    tag: string,
    text: string,
    addCss: string
  }

  const props = withDefaults(defineProps<Props>(), {
    tag: 'p',
    text: '',
    addCss: ''
  })

  const cssClass = computed((): string => {
    const classes = ['p-1']
    if ((props.addCss || '').trim().length > 0) {
      classes.push(props.addCss.trim())
    }
    return classes.join(' ')
  })
</script>

<template>
  <component :is="tag" :class="cssClass">{{ text }}</component>
</template>
```

Here our logic within the computed cssClass property is a bit more complex. Our component also has a property called **addCss** (for additional Css) that can be used to specify **CSS classes** for our component in addition to the ones internally set initially to the **const cssClasses** variable. Within the **cssClass** computed property we check if a value for the property **addCss** has been provided. If so, we add its value to our computed value.

Here is an example on how we'll consume our ElText component:

```
<ElText tag="h2" addCss="text-red-500" text="Here ElText will render an <h2> element\
" />
```

As you can see we specified the value "**text-red-500**" for the addCss property. Thus, the final computed value for the cssClass will be "**p-1 text-red-500**".

Furthermore, since we are rendering the component dynamically based on the **tag** property specified, we can render our text as any valid Html element we wish for. In the example above, we specified the **tag** property to be "**h2**" and thus it will render an **<h2>** element. Or we could have specified a tag value of "**p**" and it will render as a **<p>** element etc. We are able to do this thanks to the Vue's **<component :is>**[40]

Primitives View

Let's create a view where we can consume our primitives so that we can visually debug and prototype them as we develop them. This view can become apoint of reference for our basic library of primitives from which we will build more complex components later.

Create the following file:

[40]https://vuejs.org/guide/essentials/component-basics.html#dynamic-components

- **src/views/Primitives.view.vue**

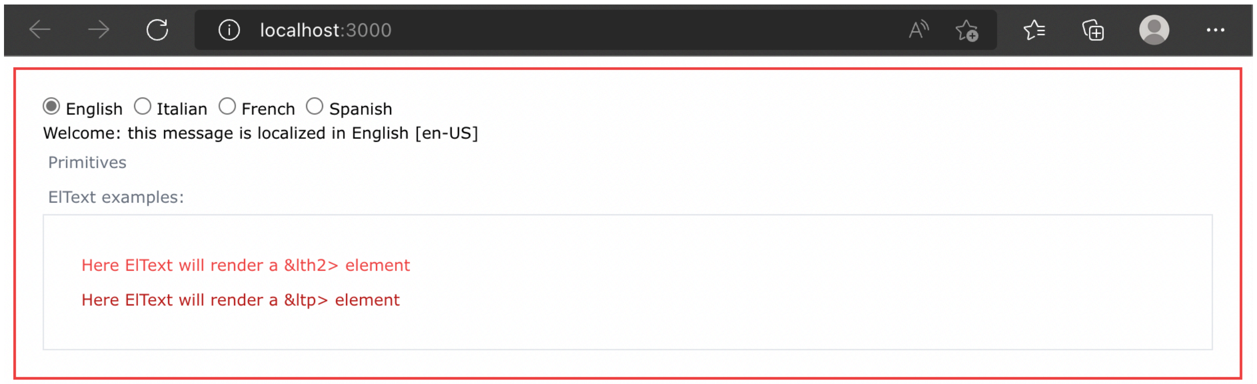
The initial code for this file is the following:

```
//file: src/views/Primitives.view.vue
<script setup lang="ts">
  // import a reference to the ElText primitive
  import ElText from '@/components/primitives/text/ElText.vue'
</script>

<template>
  <div class="about">
    <ElText tag="h1" addCss="text-gray-500" text="Primitives"/>
    <ElText tag="h2" addCss="text-gray-500" text="ElText examples:"/>
    <div class="p-6 border">
      <ElText tag="h2" addCss="text-red-500" text="Here ElText will render a &lth2&g\
t element"/>
      <ElText tag="p" addCss="text-red-700" text="Here ElText will render a &ltp&gt \
element"/>
    </div>
  </div>
</template>
```

Let's now temporarily import the **PrimitesView** in our App.vue and add it after ItemsView so we can verify it renders correctly (also temporarily add a v-if false to ItemsView to hide it):

```
// file: rc/App.vue
<script setup lang="ts">

  ...

  // import a reference to the Primitives view
  import PrimitivesView from '@/views/Primitives.view.vue'

  ....

<template>
  <div class="home m-2 p-2 border-2 border-red-500">
    <LocaleSelector />
    <h3>{{ t('home.welcome') }} [{{ currentLocale }}]</h3>
    <ItemsView v-if="false"/> <!-- temporarily add a v-if false to hide this -->
    <PrimitivesView /> <!-- add this -->

    ....
```

Now run the application and navigate to the **Primitives** view to see what we got. If all worked as expected, you should see something like this:



◉ English ○ Italian ○ French ○ Spanish
Welcome: this message is localized in English [en-US]
Primitives
ElText examples:
Here ElText will render a &lth2> element
Here ElText will render a &ltp> element

As you can see, the two ElText elements are rendered with different HTML tags. The first one as <h2> while the second one as a <p> element. We also specified some additional CSS class through their **addCss** property. The first has "text-red-500" which is a red, and the second one "text-red-700" which is a darker red in the default TailwindCSS colors.

Chapter 13 Recap

What We Learned

- We talked a bit about about atomic design
- We learned how to structure a directory of primitive elements from which we'll build higher-level components that are more complex

Observations

- We only created one primitive called ElText

Based on these observations, there are a few improvements that can be done:

Improvements

- We need to create a few more primitives
- We need to start consuming these primitives in higher-level components

Chapter 14 - More Primitives

Let's add now a few more primitives. This is just to give you some idea about the direction to take to build your own foundation library from which you can then derive all your higher level components.

Button Elements

Let's start for now by creating a button element for our primitives library, similar to how we created the **ElText** in the previous chapter.

Create a file called **ElButton.vue** under directory **src/components/primitives/buttons**.

For the code, use the following:

```
// file: src/components/primitives/buttons/Elbutton.vue
<script setup lang="ts">
  import { computed } from 'vue'

  interface Props {
    testid?: string
    id: string
    disabled?: boolean
    label: string
    addCss?: string
  }

  const props = withDefaults(defineProps<Props>(), {
    testid: 'not-set',
    id: 'not-set',
    disabled: false,
    label: 'label text',
    addCss: ''
  })

  // define emits for custom events
  const emit = defineEmits<{
    (e: 'clicked', id: string): any
  }>()

  // on click handler
  const handleClick = () => {
    // proceed only if the button is not disabled, otherwise ignore the click
    if (!props.disabled) {
      emit('clicked', props.id)
    }
  }

  // a computed property to return a different css class based on the selected value
  const cssClass = computed((): string => {
    let classes = ['font-bold py-1 px-2 border rounded']

    if (props.disabled) {
      // these are the button CSS classes when disabled
      classes.push('bg-gray-500 text-gray-300 opacity-50 cursor-not-allowed')
    } else {
      // these are the button CSS classes when enabled
      classes.push('bg-blue-500 text-white hover:bg-blue-400')
    }

    // addCss will have additional CSS class
    // we want to apply from where we consume this component
    const addCss= (props.addCss || '').trim()
    if (addCss.length > 0) {
      classes.push(addCss)
    }
    return classes.join(' ')
  })
</script>

<template>
  <button type="button"
    :data-testid="testid"
    :aria-label="label"
    :disabled="disabled"
    :class="cssClass"
    @click="handleClick">
    <span>{{ label }}</span>
  </button>
</template>
```

Here as you can see we start having a bit more complexity than what we had in **ElText**. First, we have an **handleClick** method that emits the **clicked** event when the button is clicked. This way, we can handle the click in the parent component where we will consume this primitive.

We have an **addCss** property (like we have in the ElText primitive), and a **label** property which is for the text of the button label. We also have a disabled **boolean** property to render the button either as enabled or disabled. Then we use this property in two places:

- Within the **handleClick** method, we make sure we proceed only if the button is not disabled
- Within the computed **cssClass** property, we check if the disabled property value is true to render a different set of CSS classes (with TailwindCSS here we set the text to gray with text-gray-300 for example, and a few other changes)

Here is an example on how we'll consume our **ElButton** component:

```
<ElButton id="my-button" label="This is a button" @clicked={onButtonClicked} />
```

*NOTE: We emit (dispatch) a custom event called **clicked** (not "click") so we can consume with **@clicked** binding. If we were to emit "click" and consume with on:click there is a chance that the event would fire twice as "click" is a native Vue event and therefore would fire both inside ElButton and on the actual <ElButton> instance itself.*

I hope you start seeing the power of organizing and building primitives this way. Ahead we'll soon compose higher-level components out of these primitives and you will see how easier to manage this will be, plus the code will be much cleaner and encapsulated.

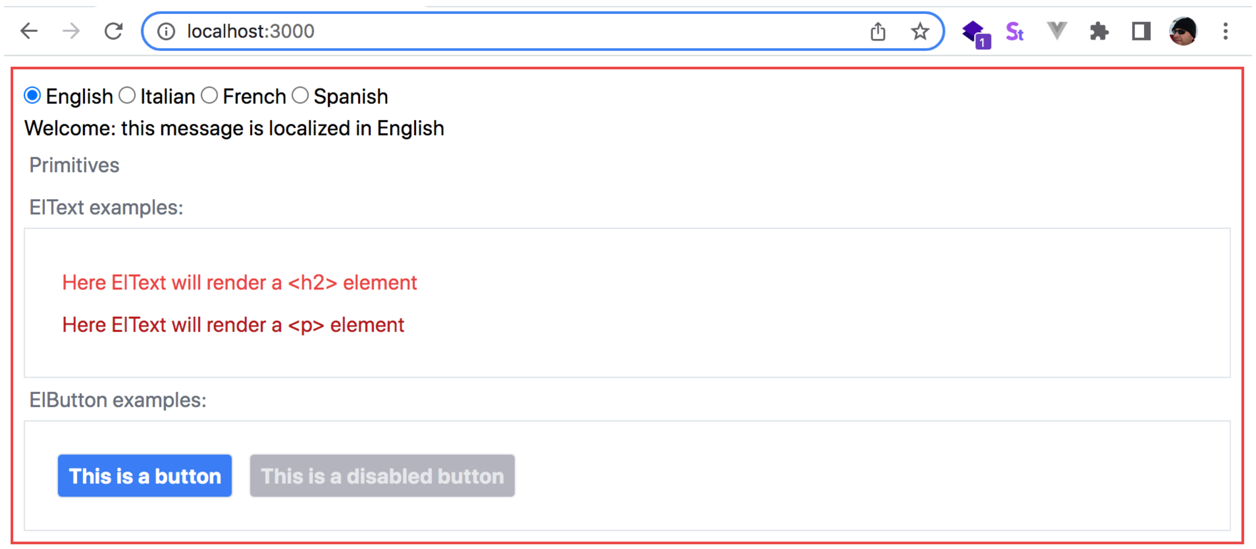
Primitives View - update

Within the primitives view, let's now consume the ElButton as in the example above so we can visually prototype the different button states.

Modify the file code within **src/views/Primitives.vue**:

```
// file src/views/Primitives.view.vue
<script setup lang="ts">
  import ElText from '@/components/primitives/text/ElText.vue'
  import ElButton from '@/components/primitives/buttons/ElButton.vue'

  const onButtonClicked = (id: string) => {
    console.log('PrimitivesView: onButtonClicked', id)
  }
</script>

<template>
  <div class="about">
    <ElText tag="h1" addCss="text-gray-500" text="Primitives" />
    <ElText tag="h2" addCss="text-gray-500" text="ElText examples:" />
    <div class="p-6 border">
      <ElText tag="h2" addCss="text-red-500" text="Here ElText will render a &lth2&g\
t element" />
      <ElText tag="p" addCss="text-red-700" text="Here ElText will render a &ltp&gt \
element" />
    </div>
                <!-- begin: add code block: -->
    <ElText tag="h2" addCss="text-gray-500" text="ElButton examples:" />
    <div class="p-6 border">
      <ElButton
        id="my-button-1"
        :disabled="false"
        label="This is a button"
        @clicked="onButtonClicked"/>
      <ElButton
        id="my-button-2"
        :disabled="true"
        label="This is a disabled button"
        addCss="ml-2"
        @clicked="onButtonClicked"/>
    </div>
                <!-- end: add code block: -->
  </div>
</template>
```

Now run the application and navigate to the **Primitives** view to see what we got. If all worked as expected, you should see something like this:



● English ○ Italian ○ French ○ Spanish

Welcome: this message is localized in English

Primitives

ElText examples:

Here ElText will render a <h2> element

Here ElText will render a <p> element

ElButton examples:

As you can see, the buttons are rendered with our specified label text, and the one on the right is rendering as “disabled” (Note that we also specified a margin-left with the **addCss** property using TailwindCSS **ml-2** value).

NOTE: I did not add a handler for @clicked event yet, but you are welcome to add one in the Primitives view and log a message to the console to test that is working

Toggle/Checkbox Elements

Let’s add one more primitive called **ElToggle** that will behave like a checkbox but looks like a toggle.

Create a file called **ElToggle.vue** under directory **src/components/primitives/toggles**.

For the code, use the following:

```
// file: src/components/primitives/toggles/ElToggle.vue
<script setup lang="ts">
  import { computed } from 'vue'

  interface Props {
    testid?: string
    id: string
    disabled?: boolean
    checked: boolean
    addCss?: string
  }
  const props = withDefaults(defineProps<Props>(), {
    testid: 'not-set',
    id: 'not-set',
    disabled: false,
    checked: false,
    addCss: ''
  })

  // define emits for custom events
  const emit = defineEmits<{
    (e: 'clicked', id: string): any
  }>()

  // a computed property to return a different css class based on the selected value
  const cssClass = computed((): string => {
    const classes = [
      'relative inline-flex flex-shrink-0 h-6 w-12 border-1 rounded-full cursor-poin\
ter transition-colors duration-200 focus:outline-none'
    ]
    if (props.checked) {
      classes.push('bg-green-400')
    } else {
      classes.push('bg-gray-300')
    }
    if (props.disabled) {
      classes.push('opacity-40 cursor-not-allowed')
    }
    const addCss = (props.addCss || '').trim()
    if (addCss.length > 0) {
      classes.push(addCss)
    }
    return classes.join(' ')
  })

  // css class for the innser span element
  const innerCssClass = computed((): string => {
    const classes = [
      'bg-white shadow pointer-events-none inline-block h-6 w-6 rounded-full transfo\
rm ring-0 transition duration-200'
    ]
    if (props.checked) {
      classes.push('translate-x-6')
    } else {
      classes.push('translate-x-0')
    }
    return classes.join(' ')
  })

  // click handler
  const handleClick = () => {
    // proceed only if it is not disabled, otherwise ignore the click
    if (!props.disabled) {
      emit('clicked', props.id)
    }
  }
</script>

<template>
  <button
    type="button"
    role="checkbox"
    :data-testid="testid"
    :aria-checked="checked"
    :disabled="disabled"
    :class="cssClass"
    @click="handleClick"
  >
    <span :class="innerCssClass"></span>
  </button>
</template>
```

As you can see this looks a lot similar to the ElButton primitive we created earlier.
Here too we have an **handleClick** method that emits the **clicked** event when the toggle is clicked. This way, we can handle the click in the parent component where we will consume this primitive.

We have an **addCss** property (like we have in the ElButton primitive), and a **checked** property which we'll use to specify whether the toggle is on or off. We also have a disabled **boolean** property to render the toggle as either enabled or disabled. Then we use this property in two places:

- Within the **handleClick** method, we make sure we proceed only if the toggle is not disabled

- Within the computed **cssClass** property, we check if the disabled property value is true to render a different set of CSS classes, we do the something similar for the **checked** property as well to change the background color of the toggle track
- We have an additional computed property called **innerCssClass** (this is for the inner <span> of the toggle which will render as a white circle). Inside here we check for the **checked** property value to determine how much we have to shift the circle horizontally (through the TailwindCSS **translate-x** property)

Here is an example on how we'll consume our **ElButton** component:

```
<ElToggle id="toggle-a" :checked="state.checked" :disabled="false" @clicked="myToggl\
eClickHandler" />
```

Primitives View - one more update

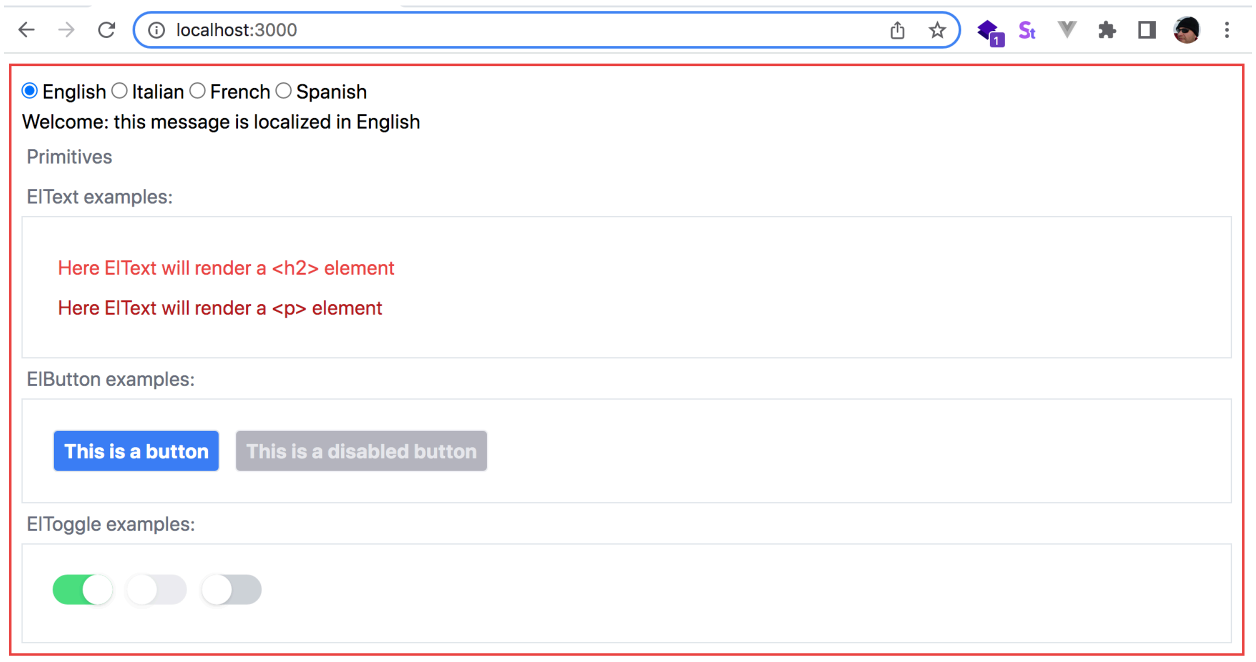
Within the primitives view, let's now consume the ElToggle as in the example above so we can visually prototype the **ElToggle**.

Modify the file code within **src/views/Primitives.vue** so that we add the ElToggle just created, and some state to better track multiple instance and verify visually that is working. Here is the full updated code:

```
// file: src/views/Primitives.view.vue
<script setup lang="ts">
  // import reactive to create some local state
  import { reactive } from 'vue'
  // import a reference to the ElToggle primitive
  import ElToggle from '@/components/primitives/toggles/ElToggle.vue'

  ...

  // create a local reactive state to track which toggle is checked
  const state = reactive({
    toggles: [
      {
        id: 'toggle-a',
        checked: true
      },
      {
        id: 'toggle-b',
        checked: false
      },
      {
        id: 'toggle-c',
        checked: false
      }
    ]
  })

  ...

  // the toggle click handler (we'll use this on all the toggles below)
  const onToggleClicked = (id: string) => {
    const stateItem = state.toggles.find((item) => item.id === id)
    if (stateItem) {
      // toggle the value of the ElToggle that was clicked
      stateItem.checked = !stateItem.checked
    }
  }
</script>

<template>
  <div class="about">

    ...

    <ElText tag="h2" addCss="text-gray-500" text="ElToggle examples:" />
    <div class="p-6 border">
      <ElToggle
        id="toggle-a"
        :checked="state.toggles.find((item) => item.id === 'toggle-a').checked"
        :disabled="false"
        @clicked="onToggleClicked"
      />
      <ElToggle
        id="toggle-b"
        :checked="state.toggles.find((item) => item.id === 'toggle-b').checked"
        :disabled="true"
        addCss="ml-2"
        @clicked="onToggleClicked"
      />
      <ElToggle
        id="toggle-c"
        :checked="state.toggles.find((item) => item.id === 'toggle-c').checked"
        :disabled="false"
        addCss="ml-2"
        @clicked="onToggleClicked"
      />
    </div>
  </div>
</template>
```

Note how here we are using Vue's reactive to create a simple local state to track the checked state of all toggles. Then in the **onToggleClicked** method we retrieve the state information for that toggle and invert its current **checked** value.

Now run the application and navigate to the **Primitives** view to see what we got. If all worked as expected, you should see something like this:



● English ○ Italian ○ French ○ Spanish
Welcome: this message is localized in English
Primitives
ElText examples:
Here ElText will render a <h2> element
Here ElText will render a <p> element
ElButton examples:

ElToggle examples:

You are welcome to keep following this pattern and start creating more complex primitives like icons, textboxes, dropdowns, lists etc. For now we'll stop here, and in the next chapter we'll start consuming the primitives we have created to compose higher level components.

Optional: You might also want to add a barrel index.ts file under components/primitives and export all primitives in an organized fashion so that you can simplify your imports like, i.e. `import { ElText, ElButton, ElToggle } from '@/components/primitives':`

```
// file: src/components/primitives/index.ts

// text
import ElText from './text/ElText.vue'

// buttons
import ElButton from './buttons/ElButton.vue'

// toggles
import ElToggle from './toggles/ElToggle.vue'

export {
  // text
  ElText,
  // buttons
  ElButton,
  // toggles
  ElToggle
}
```

Chapter 14 Recap

What We Learned

- We learned how to add additional components to our custom library by adding an ElButton and ElToggle primitives
- We learned how to render these primitives with different CSS classes conditionally to other properties like disabled, selected etc

Observations

- We did not consume these primitives in higher level components yet (besides the Primitive view use to visually prototype them)

Based on these observations, there are a few improvements that can be done:

Improvements

- We need to start consuming these primitives in higher-level coponents
- We need to start consuming the primitives and higher-level components in other existing component like our initial **ItemsList** and **Item** components

Chapter 15 - A Primitive Modal

I wanted to dedicate an additional chapter to creating a Modal component. There are many ways to create modals in Vue. There are also plug-ins created by various authors out there. You are free to choose anything you like of course and skip this chapter completely.

Here, I wanted to introduce a way of creating it a modal component that, in my experience over the years, has worked out to be one of the best ways in any front-end frameworks, including **Svelte** or **React**.

One of the main difference between a Modal component and a traditional component is that a Modal must **prevent interaction** with the rest of the application until the user dismisses the dialog.

The main use case for a dialog is to prompt the user to confirm an action, which might be usually destructive, like deleting a record or updating data (thus overwriting existing data, etc). The Modal will usually present a dialog box with a message and two buttons: one to **confirm** the action (primary) and one to **cancel** the action (secondary).

Our goal is to have a hook called **useModalDialog** that will return a reference to a shared instance of a Modal component and we can consume like this:

```
const onDeleteClick = async () => {
  const title = 'Confirm Delete'
  const message = 'Are you sure you want to delete this record? This action cannot b\
e be undone.'
  // note that prompt() will block code execution to this line
  const result = await useModalDialog().prompt(title, message)
  // code will resume after user clicks on Cancel or Confirm.
  // "result" will be true if the user clicked on Confirm button, false if Cancel bu\
tton
  // If result is true, execute the dangerous action, otherwise ignore
  if (result) {
    // execute the delete action
  }
}

...
```

We'll expect the prompt() method to be **async** and block execution of our code at that line till a **result** is returned. Similar to how the native JavaScript prompt works.

We also want to pass an icon to our dialog that will be rendered thanks to Vue's **<component :is>**[41]. The icon will be an optional parmaeter and if no icon is passed, then no icon will be rendered.

Icon: ElIconAlert

Before we start working on the modal code itself, let's create a preliminary icon icon primitive. Create the file **ElIconAlert.vue** under **directory src/components/primitives/icons/** and put the following code in it:

```
// file: src/components/primitives/icons/ElIconAlert.vue
<script setup lang="ts">
  import { computed } from 'vue'

  interface Props {
    testid?: string
    addCss?: string
  }

  const props = withDefaults(defineProps<Props>(), {
    testid: 'not-set',
    addCss: ''
  })

  const cssClass = computed(() => {
    const result = ['h-6 w-6 ']
    if ((props.addCss || '').trim().length > 0) {
      result.push(props.addCss)
    }
    return result.join(' ').trim()
  })
</script>

<template>
  <svg :class="cssClass" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0\
 24 24" stroke="currentColor" aria-hidden="true">
    <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M12 9v2\
```

[41]https://vuejs.org/guide/essentials/component-basics.html#dynamic-components

```
m0 4h.01m-6.938 4h13.856c1.54 0 2.502-1.667 1.732-3L13.732 4c-.77-1.333-2.694-1.333-\
3.464 0L3.34 16c-.77 1.333.192 3 1.732 3z" />
  </svg>
</template>
```

Create also a barrel indexts file within the icons directory with this:

```
// file: src/components/primitives/icons/index.ts
import ElIconAlert from './ElIconAlert.vue'

export {
  ElIconAlert
}
```

We'll be dynamically adding the icon as one of the possible properties passed to our modal component.

File ElModal.vue

Within the directory **src/components/primitives/modals**, create a file called **ElModal.vue** with the following code for the <script> section:

```
// file: src/components/primitives/modals
<script setup lang="ts">
  import { ref, defineComponent } from 'vue'
  import ElButton from '../buttons/ElButton.vue'

  interface ModalProps {
    testid?: string
    title: string
    message?: string
    cancelLabel?: string
    confirmLabel?: string
    primaryButtonType?: string
    icon?: ReturnType<typeof defineComponent>
    iconAddCss?: string
    resolve: (value: boolean | PromiseLike<boolean>) => void
  }

  const props = withDefaults(defineProps<ModalProps>(), {
    testid: 'not-set',
    message: '',
    cancelLabel: 'Cancel',
    confirmLabel: 'Confirm?',
    primaryButtonType: 'primary'
  })

  const isVisible = ref(true)

  const close = async () => {
    isVisible.value = false
  }

  const confirm = async () => {
    await close()
    await props.resolve(true)
  }

  const discard = async () => {
    await close()
    await props.resolve(false)
  }
</script>

<template>
  <div
    v-show="isVisible"
    class="fixed z-10 inset-0 overflow-y-auto transform transition-all"
    role="dialog"
    aria-modale="true"
    :aria-labelledby="title"
  >
    <div class="flex items-end justify-center min-h-screen pt-4 px-4 pb-20 text-cent\
er sm:block sm:p-0">
      <!-- Background overlay -->
      <div class="fixed inset-0 bg-gray-400 bg-opacity-75" aria-hidden="true" />
      <span class="hidden sm:inline-block sm:align-middle sm:h-screen" aria-hidden="\
true">&#8203;</span>
      <!--  Modal panel -->
      <div
        id="modal-panel"
        class="relative inline-block align-bottom bg-white rounded-lg px-4 pt-5 pb-4\
 text-left overflow-hidden shadow-xl sm:my-8 sm:align-middle sm:max-w-lg sm:w-full s\
m:p-6"
      >
        <div>
          <div
            v-if="icon"
            id="icon-container"
            class="mx-auto flex items-center justify-center h-12 w-12 rounded-full b\
g-green-100"
          >
            <component :is="icon" :addCss="iconAddCss" />
          </div>
          <div id="text-container" class="mt-3 text-center sm:mt-5">
            <h3 class="text-lg leading-6 font-medium text-gray-900">{{ props.title }\
}</h3>
            <div v-if="props.message" class="'mt-2">
              <p class="text-sm text-gray-500">{{ props.message }}</p>
            </div>
          </div>
          <div id="button-continer" class="mt-5 sm:mt-6 sm:grid sm:grid-cols-2 sm:ga\
p-3 sm:grid-flow-row-dense">
            <ElButton id="btn-cancel" :label="props.cancelLabel" buttonType="seconda\
ry" @clicked="discard" />
            <ElButton id="btn-confirm" :label="props.confirmLabel" :buttonType="prim\
aryButtonType" @clicked="confirm" />
          </div>
        </div>
      </div>
    </div>
  </div>
</template>
```

NOTE: We will not consume ElModal directly in our views/components, but instead rely on the useModalDialog hook. Additionally, I made some enhancements to the ElButton to render with different css based on a type classification like primary/secondary/danger etc. Please see the public GitHub repository for the additional changes.

File src/index.html

Modify the html by adding another <div> with id "modal" that will be the target element where we'll mount our ElModal component:

```
// file: src/index.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" href="/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
  </head>
  <body>
    <div id="app"></div>
    <div id="modal"></div> <!-- add this line here -->
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

File useModalDialog.ts

Within the same directory, create a file called **useModalDialog.ts** with the following code:

```
// file: src/components/primitives/modals/useModalDialog.ts
import { defineComponent } from 'vue'
import ElModal from './ElModal.vue'

interface ModalOptions {
  targetId?: string
  title?: string
  message?: string
  cancelLabel?: string
  confirmLabel?: string
  primaryButtonType?: string
  icon?: ReturnType<typeof defineComponent>
  iconAddCss?: string
}

import { nextTick, createApp } from 'vue'

export const useModalDialog = (options?: ModalOptions) => {
  return {
    prompt: (title: string, message?: string): Promise<boolean> => {
      const targetId = options?.targetId ? options?.targetId : '#modal'
      // return a Promise and pass the 'resolve" as a property to the ElModal
      // ElModal will invoke "resolve" from Confirm with true or from Cancel with fa\
lse
      return new Promise<boolean>((resolve) => {
        const cancelLabel = options?.cancelLabel ? options.cancelLabel : 'Cancel'
        const confirmLabel = options?.confirmLabel ? options.confirmLabel : 'Confirm\
?'

        nextTick(() => {
          return createApp(ElModal, {
            title: title || options?.title,
            message: message || options?.message,
            confirmLabel,
            cancelLabel,
            primaryButtonType: options?.primaryButtonType,
            icon: options?.icon,
            iconAddCss: options?.iconAddCss,
            resolve
          }).mount(targetId)
        })
      })
    }
  }
}
```

Note: We could have use Vue Teleport feature here but we opted for createApp in this case. Feel free to use Teleport if you prefer.

The core concept here is to return a **Promise** from the prompt() method that will be **awaited** in the consuming code till the user clicks on either **Cancel** or **Confirm**. The promise will be resolved when the user clicks on Cancel or Confirm and the result will be either **false** (cancelled) or **true** (confirmed).

One more thing to notice is how in our ElModal we make use of vue's <component :is> to dynamically render the icon from the props.icon passed into our ElModal. Mind you, <component :is> will render any component, not just an icon. In this case however, we most likely always use it to render an optional icon within our modal:

```
<component :is="icon" :addCss="iconAddCss"/>
```

Let's now modify the Primitives.view.vue so we can test a couple of different scenarios, for two different modals.

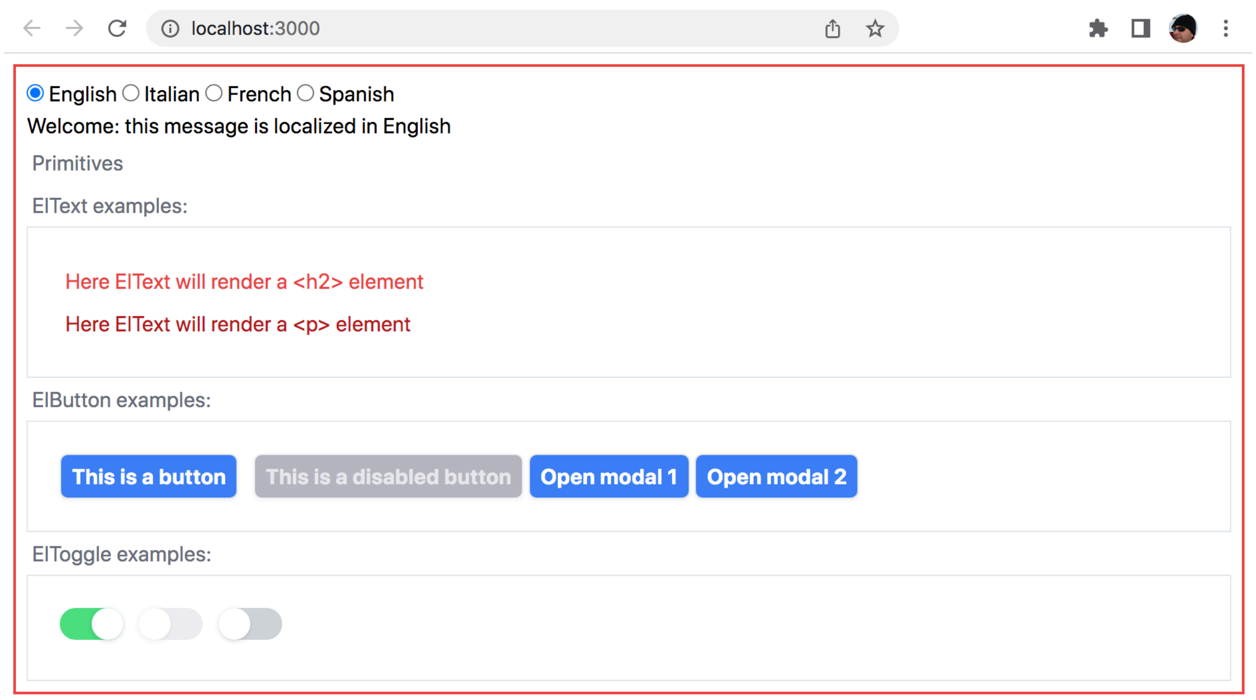
Updates to Primitives.view.vue

Now let's consume our useModal hook. Open the Primitives.view.vue file and make the following changes:

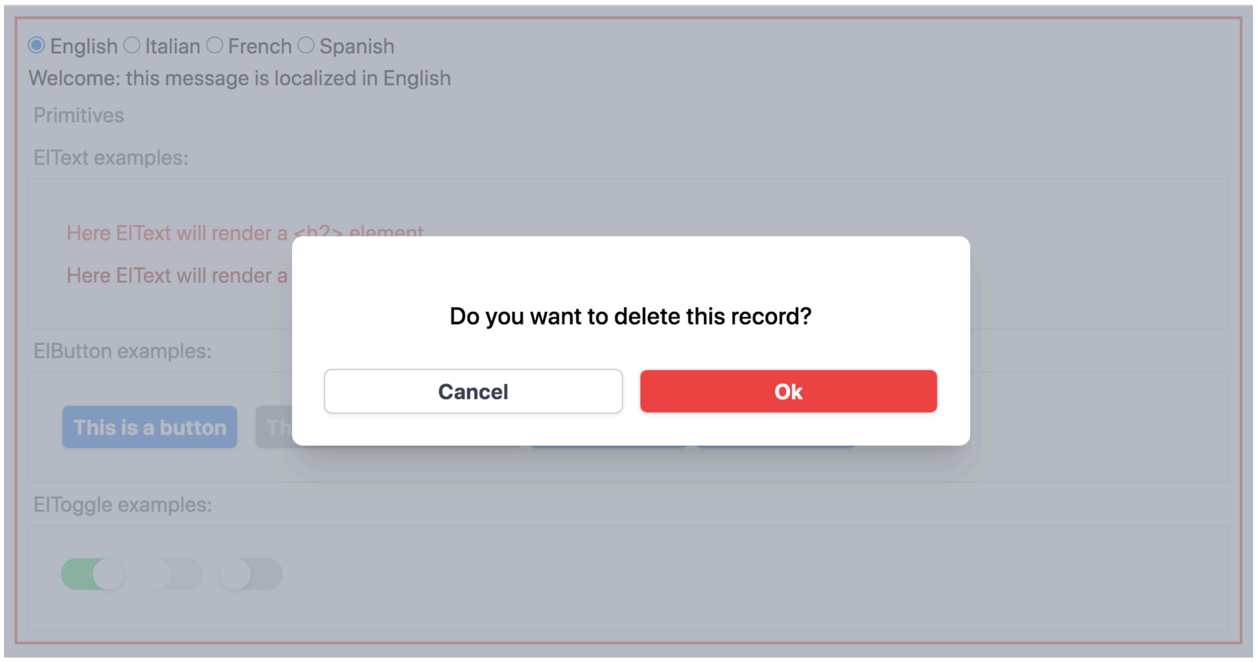
```
// file: src/views/Primitives.view.vue
<script setup lang="ts">
  import { reactive } from 'vue'
  // import a reference to the ElText primitive
  import {
    ElText,
    ElButton,
    ElToggle,
    ElIconAlert, // import a reference to the alert icon
    useModalDialog // import a reference to useModalDialog
  } from '@/components/primitives'

  ...

  // add this new handler for the two new Open Modal X buttons we'll add shortly
    const onOpenDialogClicked = async (id: string) => {
    console.log('PrimitivesView: onOpeanDialogClicked', id)
    // handle the new buttons with id "open-modal-x" (we'll be adding shortly)
    if (id === 'open-modal-1') {
      // here we invoke our useModal with the custom labels for the buttons
      // then we invoke modal.prompt() and await it
      const result = await useModalDialog({
        cancelLabel: 'Cancel',
        confirmLabel: 'Ok',
        primaryButtonType: 'danger'
      }).prompt('Do you want to delete this record?')
      // the result will be true if the user click on COnfirm, or false if click on \
Cancel
      console.log('----- PrimitivesView: onButtonClicked: modal-1 prompt result', re\
sult)
    } else if (id === 'open-modal-2') {
      // here we invoke our useModal with the custom labels for the buttons
      // then we invoke modal.prompt() and await it
      const result = await useModalDialog({
        cancelLabel: 'Cancel',
        confirmLabel: 'Confirm',
        icon: ElIconAlert, // here we use the icon component created earlier
        iconAddCss: 'text-red-600'
      }).prompt('Do you want to delete this record?', '')
      // the result will be true if the user click on COnfirm, or false if click on \
Cancel
      console.log('----- PrimitivesView: onButtonClicked: modal-2 prompt result', re\
sult)
    }
  }
</script>

<template>
  <div class="about">

  ...

  <ElText tag="h2" addCss="text-gray-500" text="ElButton examples:" />
    <div class="p-6 border space-x-2">
      <ElButton id="my-button-1" :disabled="false" label="This is a button" @clicked\
="onButtonClicked" />
      <ElButton
        id="my-button-2"
        :disabled="true"
        label="This is a disabled button"
        @clicked="onButtonClicked"
      />
      <!-- add these two buttons: -->
      <ElButton id="open-modal-1" :disabled="false" label="Open modal 1" @clicked="o\
nOpenDialogClicked" />
      <ElButton id="open-modal-2" :disabled="false" label="Open modal 2" @clicked="o\
nOpenDialogClicked" />
    </div>

    ...
```

Browser

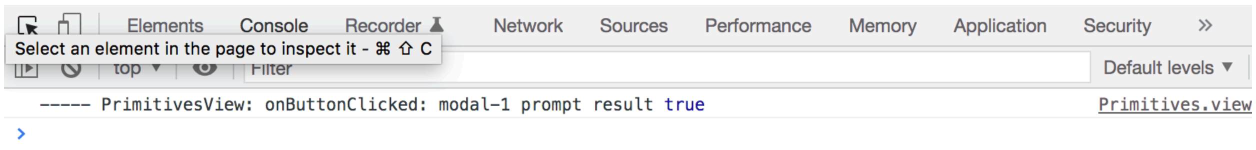
The app will now render our 3rd button:


localhost:3000
English Italian French Spanish
Welcome: this message is localized in English
Primitives
ElText examples:
Here ElText will render a <h2> element
Here ElText will render a <p> element
ElButton examples:
This is a button
This is a disabled button
Open modal 1
Open modal 2
ElToggle examples:

After clicking on the "**Open modal 1**" button, you will see a modal rendered without an icon:

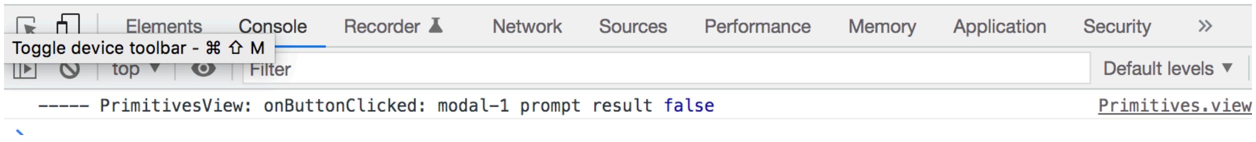


The Modal will block the execution at the await line where we call modal.prompt(). After clicking Ok, you should see it logging "result true" in the console:



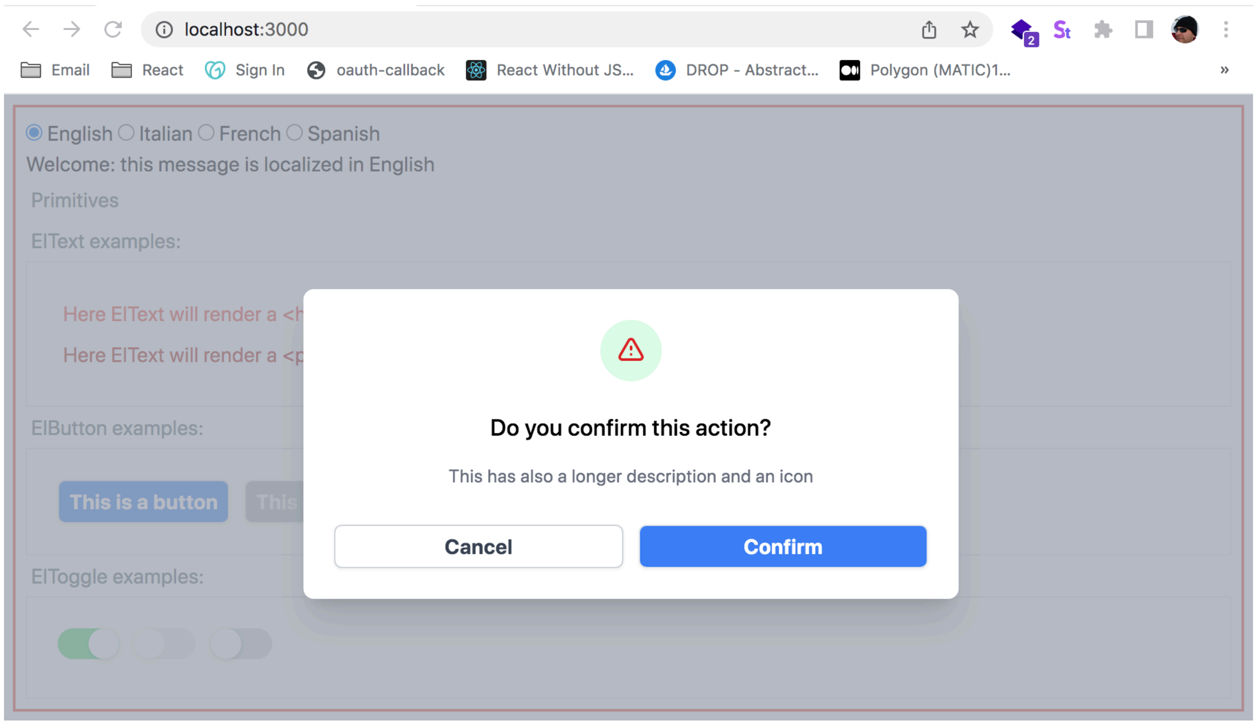
```
----- PrimitivesView: onButtonClicked: modal-1 prompt result true        Primitives.view
```

If instead you click on Cancel, it will log “result false”:



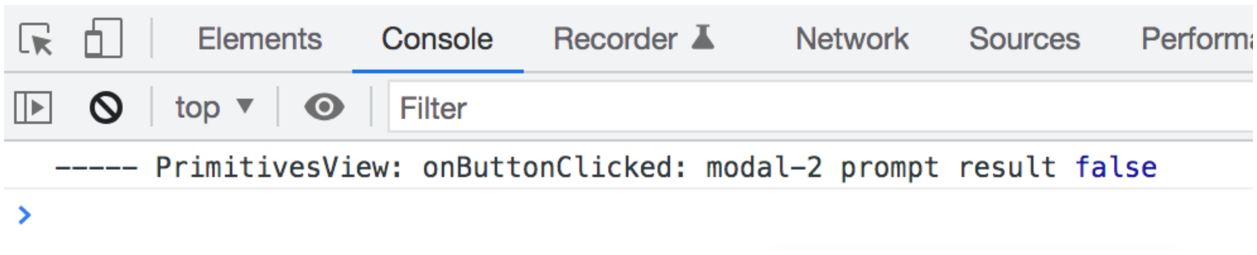
```
----- PrimitivesView: onButtonClicked: modal-1 prompt result false       Primitives.view
```

If you click on “**Open modal 2**” button, you will see a modal rendered with the alert icon:

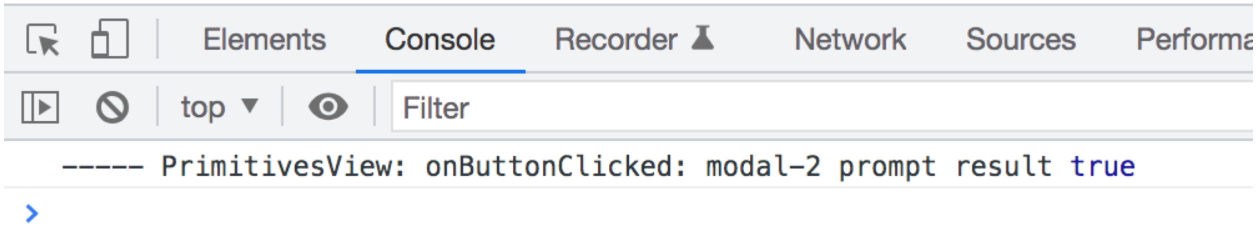


Note: here the Confirm button type is “primary”, which is teh default for Modal property “primaryButtonType” when we do not explicitely pass a value for it.

Here too click on Cancel/Confirm and make sure that the console logs “modal-2 result true/false” as well:



```
----- PrimitivesView: onButtonClicked: modal-2 prompt result false
```



```
----- PrimitivesView: onButtonClicked: modal-2 prompt result true
```

Chapter 15 Recap

What We Learned

- We learned how to build a Modal component that leverages some of our previous primitives like ElText and ElButton and uses a technique with Promises to block the execution when we invoke modal.prompt() from the consuming code and return true or false once the Promise is resolved by clicking on Cancel or Confirm

Observations

- We did not add unit tests against the Modal component

Based on these observations, there are a few improvements that can be done:

Improvements

- You could write unit tests against the Modal component to verify it renders and behaves as expected

Chapter 16 - Higher-level components

Let's now consume the primitives we created so far within the **Item** component.
As we do this, we'll make any additional adjustment we discover necessary.
Finally, if needed, we might be creating additional primitives that we do not have yet (i.e. a list)

Item Component - updates

Let's start by opening the file **src/components/items/children/Item.component.vue** and observe the current HTML template:

```
// file: src/components/items/children/Item.component.vue

...

<li :data-testid="testid" :class="cssClass" @click="handleClick">
  <div class="selected-indicator">*</div>
  <div class="name">{{ model.name }} [{{ model.selected }}]</div>
</li>

...
```

There are two elements that can be replaced with our primitives. We could use an **ElToggle** for the selected indicator, and an **ElText** for the name.

First, let's udpate our imports and also add a components block to the component definition that includes ElButton and ElText and also add their references tot he components section within defineComponent:

```
// file: src/components/items/children/Item.component.vue

...

<script lang="ts">

  ...

  // add the following two lines:
  import ElText from '@/components/primitives/text/ElText.vue'
  import ElToggle from '@/components/primitives/toggles/ElToggle.vue'

  ...

  // create our component with defineComponent
  export default defineComponent({
    name: 'ItemComponent',
    emits: ['selectItem'],
    components: {
      ElText, // <- add this
      ElToggle // <- add this
    },

    ...
```

Then start updating the HTML template as follows:

```
// file: src/components/items/children/Item.component.vue

...

<li :data-testid="testid" :class="cssClass" @click="handleClick">
  <div class="selected-indicator">*</div>
  ~~<!-- remove this line: -->~~
  ~~<div class="name">{{ model.name }}</div>~~
  <!-- add this line: -->
  <ElText :testid="`${testid}-text`" tag="div" :text="model.name"/>
</li>

...
```

Run the application and make sure it still renders the list of items without errors.

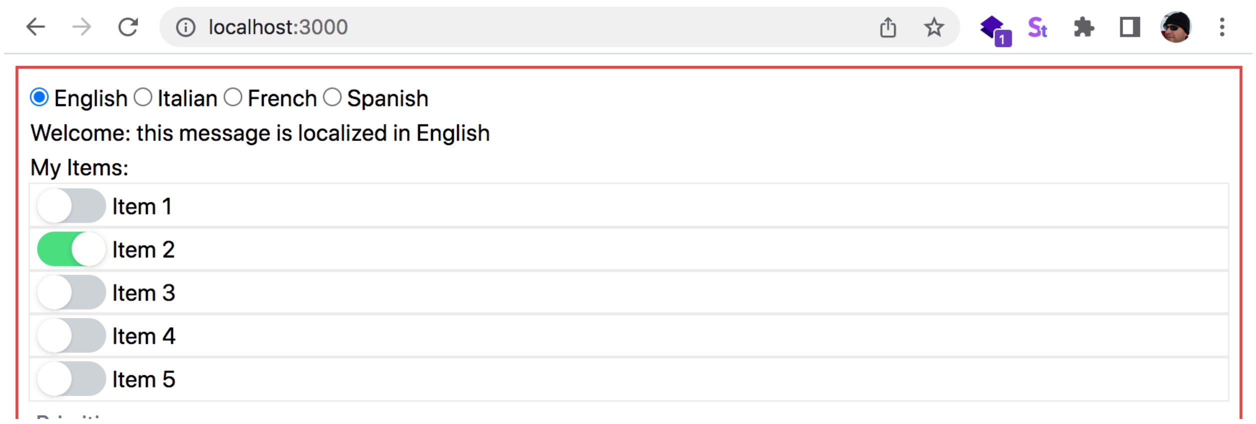
Now let's finish updating the HTML template by replacing the selected indicator with our **ElToggle**:

```
// file: src/components/items/children/Item.component.vue

...

<li :data-testid="testid" :class="cssClass" @click="handleClick">
~~  <!-- remove this line: -->~~
~~  <div class="selected-indicator">*</div>~~
  <!-- add this line: -->
  <ElToggle :testid="`${testid}-toggle`" :checked="model.selected" />
  <ElText :testid="`${testid}-text`" tag="div" :text="model.name"/>
</li>

...
```

NOTE: we do not have to handle the on:clicked event on the ElToggle here as we are already handling a on:click event on the entire <li> element.

Again, refresh the browser and make sure everything still renders without errors.

It should look currently like this (the layout will be a bit off, so we'll need to tweak the Item component CSS and start using TailwindCSS here as well):



● English ○ Italian ○ French ○ Spanish
Welcome: this message is localized in English
My Items:
Item 1
Item 2
Item 3
Item 4
Item 5

Let's move the toggle to the right side:

```
// file: src/components/items/children/Item.component.vue

...

<li :data-testid="testid" :class="cssClass" @click="handleClick">
  <ElText :testid="`${testid}-text`" tag="div" :text="model.name"/>
  <ElToggle :testid="`${testid}-toggle`" :checked="model.selected" />
</li>

...
```

Let's add a new property called isLast that we'll use to better control the border style:

```
// file: src/components/items/children/Item.component.vue

...

  // expose a property called testid
  export let testid: string = 'not-set'
  // expose a property called isLast // <-- add this line
  export let isLast: boolean = false // <-- add this line

    ...
```

Modify the logic within the computed **cssClass** property:

```
...

  const cssClass = computed(() => {
~~    // begin: remove code block~~
~~    let css = 'item'~~
~~    if (props.model.selected) {~~
~~      css += ' selected'~~
~~    }~~
~~    // end: remove code block~~
    // begin: add code block
    let css = 'item flex items-center justify-between cursor-pointer border border-l\
-4 list-none rounded-sm px-3 py-3'
    if (props.model?.selected) {
      css += ' font-bold bg-pink-200 hover:bg-pink-100 selected'
    } else {
      css += ' text-gray-500 hover:bg-gray-100'
    }
    if (!props.isLast) {
      css += ' border-b-0'
    }
    // end: add code block
    return css.trim()
  }

...
```

Now, before we proceed, lets remove out all the custom SCSS we wrote at the beginning of this book for the ItemsList and Item component by removing the entire <style> blocks from both files.

```
// remove entire blocks of code like this
<style>
  ...
</style>
```

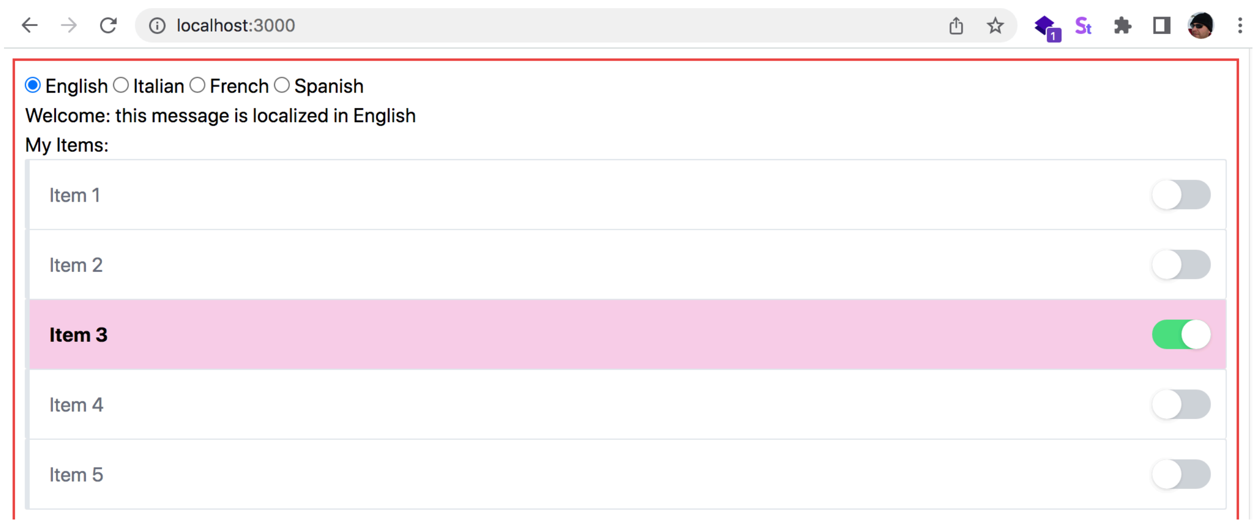
ItemsList Component - updates

We need to do a small update to the ItemsList.component.vue code as well to pass a value for the new **isLast** property of the Item component. We are going to use the **index** property for this and comparing it against the total number of items. Modify the HTML template as this:

```
// file: src/components/items/ItemsList.component.vue

...

  <ul v-show="!loading>
    <ItemComponent v-for="(item, index) in items" <!-- after item, add index here -->
      :key="item.id"
      :model="item"
      <!-- add the following line: -->
      :isLast="index === items.length - 1"
      @selectItem="onSelectItem" />
  </ul>

...
```

NOTE: You will have to update also the unit tests accordingly as they would be now failing. Please see the GitHub repo for the updated unit tests if you need help.

Refresh the browser, and if everything is correct it should render like this:



● English ○ Italian ○ French ○ Spanish
Welcome: this message is localized in English
My Items:

Item 1
Item 2
Item 3
Item 4
Item 5

Summary

Let’s reflect a little bit on what we just did. We replaced two HTML elements within the **Item.component.vue** with our new primitives. By doing this, we effectively “composed” the higher-level component “Item” from those primitives. I hope you start seeing the pattern here. Even though this was a very simple example, the sky is really the limit on how you can better structure your primitives and copose more complex primitives out of those, and ultimately the higher-level components that consume them.

Chapter 16 Recap

What We Learned

- We started to learn how to compose higher-level components by putting together the primitives we created in the previous chapters.

Observations

- We did not leverage localization and internationalization in our primitives.
- We did not write unit tests against our primitives

Based on these observations, there are a few improvements that can be done:

Improvements

- You can add localization and internationalization support through the i18n plugin as shown in other chapters
- You can write unit tests against the primitives to further create a solid foundation for your primitives library

Chapter 17 - Creating a Component Library

In this chapter we'll leverage Vite to create a component library that can be shared across different projects. Once you know how to create a component library, you could choose to publish it as an NPM package (either public or private) for more easily sharing it between different projects, or across departments in your organization.

When creating a component library, there are different approaches and architecture decision to be made, depending on different factors. One of the main thing to keep in mind is the dependencies that your library will have (i.e. web framework, state, css frameworks, other frameworks, etc).

In this chapter we'll worry about only creating a library with a couple of simple components, we'll learn how to build it and package it and how to consume it into our sample project.

In the next chapter we'll build a more complex component that might require additional things like state etc.

Create my-component-library

To setup the library project, use the terminal and execute the following command (make sure you are at the same level of your my-vue-project folder):

```
npm init vite@latest
```

The create-vite wizard will start and will ask you the name of the project. The default is vite-project, so change this to my-component-library and hit enter:

```
? Project name: › my-component-library
```

The second step will ask to select a framework. Use the keyboard arrows to scroll down the list and stop at vue, then hit enter:

```
? Select a framework: › - Use arrow-keys. Return to submit.
    vanilla
▯   vue
    react
    preact
    lit
    svelte
```

The third step will asking which “variant” you want o use. Scroll down to **vue-ts** (this is for the version that uses TypeScript) and hit enter:

```
? Select a variant: › - Use arrow-keys. Return to submit.
    vue
▯   vue-ts
```

This will create a folder called **my-component-library** which is also the name of our project. At the end it should display a message similar to this:

```
Scaffolding project in /Volumes/projects/my-component-library...

Done. Now run:

  cd my-component-library
  npm install
  npm run dev
```

The first command will navigate to the current sub-directory called **my-component-library**, the second one will install all the npm dependencies, and we do not need to run the third one in this case.

Now let’s clean up a few things. We need to remove a few files that we are not going to need since this is a library, and we’ll also need to update the project configuration so that it can correctly build and bundle our library.

Remove obsolete files

Remove the following files as they are not needed in a component library:

- index.html
- src/App.vue
- src/app.css (or style.css)

- src/main.ts
- src/assets/logo.svg (or vue.svg)

Remove also the public directory and all its contents.

Add main entry index.ts file

Add new new file under src/ called index.ts that will just export everything from the components/ sub-directory

```
// file: src/index.ts

export * from './components'
```

Update vite.config.ts

Update the Vite's config file as follows:

```
// file: vite.config.ts

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import path from 'path' // <- add this line

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  envDir: './src/', // <- add this line
  resolve: { // <- add this block
    alias: {
      '@': path.resolve(__dirname, 'src/')
    },
  },
  build: { // <- add this block
    lib: {
      entry: path.resolve(__dirname, 'src/index.ts'),
      name: "MyComponentLib",
      fileName: (format) => `my-component-lib.${format}.js`,
    },
    rollupOptions: {
      // Vue should not be bundled with the cmoponent library
      // tell vite that this is an external dependency
      external: ['vue'],
      output: {
        // To expose global variables for use in the UMD builds
        // for external dependencies
        globals: {
          vue: 'Vue'
        }
      }
    }
  }
})
```

A few things to notice in the config changes above:

- we are telling Vite that the environment directory for the source code is ./src/
- we added a "resolve" block so we can use the @ shortcut to point to src/ and avoid imports with relative paths (i.e. ../../../)
- we added a "build" block, and this is the most important change for setting up the project as a library. Here:
  - we tell Vite which is the main entry file for our library (src/index.ts),
  - set the name of our library to MyComponentLib
  - set the name of the main built files to be `my-component-lib.${format}.js` (where format will be set dynamically to `es` or `umd`)
  - set the rollupOptions so that Vue will not be bundled with our library (we'll assume this library is consumed in a project that already uses Vue thus we do not want to include it multiple times)

Update package.json commands

Update the package.json file. First, make sure we update the following scripts commands so that we can correctly build both JavaScript and the TypeScript types:

```
// file: src/package.json

{
  "name": "my-component-library",
  "version": "0.1.2",
  "scripts": {
    "clean": "rm -rf ./dist; rm -rf my-component-library-0.1.2.tgz; rm -rf ../my-com\
ponent-library-0.1.2.tgz",
    "build-types": "vue-tsc --declaration --emitDeclarationOnly --outDir ./dist",
    "build-lib": "vite build",
    "build": "npm run clean && npm run build-lib && npm run build-types",
    "pack": "npm pack; mv my-component-library-0.1.2.tgz ../my-component-library-0.1\
.2.tgz",
    "all": "npm run build && npm run pack",
    "preversion": "npm run clean",
    "version": "npm run build",
    "postversion": "npm run pack",
    "version-patch": "npm version patch -m \"Patch version\""
  }

...
```

The most important thing to notice here is that we have a master command called "all" that will run the build and then the pack command. The pack command is optional and will create a single compressed (tgz) file with all our library code, then copy this up to one directory so we could more easily consume it from our my-vue-project.

The sub-commands run buy the build command are:

- clean: this will just remove the dist/ folder and the previously packed tgx file
- build-types: this will build the TypeScript types declarations
- build-lib: this will build our Vue library code
- build: this will run the clean + build-lib + build-types sub-commands

We need to also make changes to package.json so that it can correctly build the project as a library. We need to add these sectoins/properties:

- files: this tells which directory is the destinatio for the built JavaScript files (dist in our case)
- types: this indicates the entry file for the TypeScript definitions
- main: this indicates the main entry file for our library (umd module)
- module: this indicates the main entry file for our library (es module)
- exports: this section indicates what our package will export

```
// file: src/package.json

  ...

  "files": [
    "dist"
  ],
  "types": "./dist/src/index.d.ts",
  "main": "./dist/my-component-lib.umd.js",
  "module": "./dist/my-component-lib.es.js",
  "exports": {
    ".": {
      "import": [
        "./dist/my-component-lib.es.js"
      ],
      "require": "./dist/my-component-lib.umd.js"
    },
    "./package.json": "./package.json"
  },

  ...
```

Now let's add a couple of simple components to our library.

Create Counter.vue component

Create a new file at src/components/counter/Counter.vue with the following code

```
// file: src/components/counter/Counter.vue

<script setup lang="ts">
  import { ref } from 'vue'
  const count = ref(0)
</script>

<template>
  <button type="button" @click="count++">count is {{ count }}</button>
</template>
```

Create SampleComp.vue component

Create a new file at src/components/sample-component/SampleComp.vue with the following code

```
// file: src/components/sample-component/SampleComp.vue

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

  type SampleCompProps = {
    testid?: string
    text?: string
  }

  const props = withDefaults(defineProps<SampleCompProps>(), {
    testid: 'not-set',
    text: 'not-set'
  })

  const cssClass = () => {
    return `p-2 border border-green-500`
  }
</script>

<template>
  <div :data-testid="testid" :class="cssClass()">
   <span>{{ text }}</span>
  </div>
</template>
```

Add components/index.ts barrel file

Under components/, add a barrel index.ts file and just export all our components in an organized way:

```
// file: src/components/index.ts

import Counter from './counter/Counter.vue'
import SampleComp from './sample-component/SampleComp.vue'

export {
  Counter,
  SampleComp
}
```

Build our library

Now finally run the “build” command (or you could run the “all” command) to compile and build our library:

```
npm run build
```

Consuming our library

To consume our library locally, let’s switch now to our my-vue-project and install a reference to our library by running this command:

```
npm install -D file:../my-component-library
```

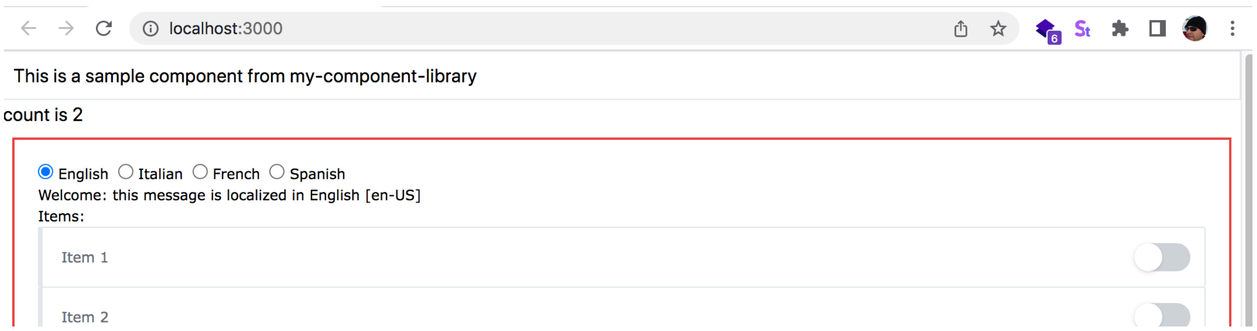
Then open the file App.vue and add the following imports at the top:

```
// file: src/App.vue

<script setup lang="ts">
  import {
    Counter,
    SampleComp
  } from 'my-component-library'

  ...
```

In the template section, let’s consume our library components:

```
// file: src/App.vue

  ...

  <template>
    <div>
      <SampleComp text="This is a sample component from my-component-library" />
      <Counter />
    </div>

  ...
```

Save and run the application. If everything worked and there are no errors, you should see something like this in the browser (here shown after I clicked two times on the count button):



This is a sample component from my-component-library

count is 2

English Italian French Spanish

Welcome: this message is localized in English [en-US]

Items:

Item 1

Item 2

Chapter 17 Recap

What We Learned

- We create a new project called my-component-library that will export a couple of simple components
- We then consumed these components in our my-vue-project

Observations

- We did not write unit tests against our components within my-component-library
- We did not publish our component library to NPM.
- We did not write more complex components that leverage other dependencies like application state or other libraries

Based on these observations, there are a few more things that can be done:

Improvements

- You can add unit tests within the my-component-library and test your components
- You could keep adding more complex components to your library that use application state or other dependencies

Chapter 18 - Creating a JavaScript library

Similarly to what we discussed in Chapter 17, we can create a library that we can publish as an NPM package that does not necessarily contains Vue components. This might be a collection of helpers, or a plugin, etc.

As you start building more complex application that will grow to a large code base, it starts to make sense to be more strict about following principles like Single Responsibility and Separation of Concerns[42].

Separating code that can be shared across different applications/projects into its own NPM package has many advantages, and if you publish it as an open-source project with a permissive license, other developers might start using it as well, providing more feedback and reporting or even helping with bugs. This might result in your package growing even stronger as time goes by.

There are a few downsides as well, like having to maintain a separate code base, having to publish a new version whenever you add new functionality or fix a bug.

Unless you are working only on one small application, and/or the code within your NPM package has not utility in other applications (or does not offer much benefits to other developers), usually the advantages will make it worth it to have it as an NPM package.

Create my-js-helpers

We'll create a new project called my-js-helpers by following similar to those as at the beginning of Chapter 17 for my-component-library (just make sure you use the name my-js-helpers this time).

Please note, this chapter will just illustrate how to create a simple NPM package that exposes some simple JavaScript helpers, thus the name my-js-helpers. But, of course, you are welcome to choose whatever name you wish for your NPM package.

One main difference after you run `npm init vite@latest` and set `my-js-helpers` as the name, is to choose `vanilla` for the framework selection:

[42] https://en.wikipedia.org/wiki/Separation_of_concerns

```
? Select a framework: › - Use arrow-keys. Return to submit.
▯   vanilla
    vue
    react
    preact
    lit
    svelte
```

And then `vanilla-ts` for the framework “variant”:

```
? Select a variant: › - Use arrow-keys. Return to submit.
    vanilla
▯   vanilla-ts
```

After you are done creating the project and have run “npm install”, let’s continue by removing unecessary files (similarly to Chapter 17)

Remove obsolete files

Remove the following files as they are not needed in a NPM package:

- favicon.svg
- typescript.svg
- app.css (or style.css)
- index.html
- main.ts
- counter.ts

Remove also the public directory and all its contents.

Add main entry index.ts file

Add new new file under src/ called `index.ts` that export all the source code we want to exposes from our NPM package. In our case, we’ll export everything from the sub-directory called `helpers` (which will create in a bit):

```
// file: src/index.ts
export * from './helpers'
```

Update vite.config.ts

Update the Vite's config file similarly to what we did in Chapter 17 (just make sure to replace my-component-library with my-js-library). Here is what it should look like:

```
// file: vite.config.ts

/// <reference types="vitest" />
/// <reference types="vite/client" />

import { defineConfig } from 'vite'
import path from 'path'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
  ],
  envDir: './src/',
  resolve: {
    alias: {
      '@': path.resolve(__dirname, 'src/')
    },
  },
  test: {
    globals: true,
    environment: 'jsdom',
    exclude: [
      'node_modules'
    ]
  },
  build: {
    lib: {
      entry: path.resolve(__dirname, 'src/index.ts'),
      name: 'MyJsHelpers',
      fileName: (format) => `my-js-helpers.${format}.js`,
    },
    rollupOptions: {
      external: [],
      output: {
        // Provide global variables to use in the UMD build
        // Add external deps here
        globals: {
        },
      },
    },
  }
})
```

Update package.json commands

Update the package.json file. First, make sure we update the following scripts commands so that we can correctly build both JavaScript and the TypeScript types:

```
// file: src/package.json

{
  "name": "my-component-library",
  "version": "0.1.2",
  "scripts": {
    "clean": "rm -rf ./dist; rm -rf my-js-helpers-0.1.2.tgz; rm -rf ../my-js-helpers\
-0.1.2.tgz",
    "build-types": "tsc --declaration --emitDeclarationOnly --outDir ./dist",
    "build-lib": "vite build",
    "build": "npm run clean && npm run build-lib && npm run build-types",
    "pack": "npm pack; mv my-js-helpers-0.1.2.tgz ../my-js-helpers-0.1.2.tgz",
    "all": "npm run build && npm run pack",
    "preversion": "npm run clean",
    "version": "npm run build",
    "postversion": "npm run pack",
    "version-patch": "npm version patch -m \"Patch version\""
  }

...
```

And similarly to Chapter 17, lets add additional configuration so that the project will build as a library:

```
// file: src/package.json

  ...

  "files": [
    "dist"
  ],
  "types": "./dist/src/index.d.ts",
  "main": "./dist/my-js-helpers.umd.js",
  "module": "./dist/my-js-helpers.es.js",
  "exports": {
    ".": {
      "import": [
        "./dist/my-js-helpers.es.js"
      ],
      "require": "./dist/my-js-helpers.umd.js"
    },
    "./package.json": "./package.json"
  },

  ...
```

Now let's add a some JavaScript helpers to our NPM package.

random-id

Create the directory `src\helpers\random-id` and inside here add a file called `random-id.ts` (the full location path will be `src/helpers/random-id/random-id.ts`) with the following code:

```
// file: src/helpers/random-id/random-id.ts

export const randomid = (): string => {
  let result: string = ''
  if (typeof window !== 'undefined' && window.crypto && window.crypto.getRandomValue\
s) {
    const array: Uint32Array = new Uint32Array(1)
    window.crypto.getRandomValues(array)
    result = array[0].toString()
  } else {
    // throw error
    // throw Error('Browser does not support window.crypto.getRandomValues')
    // if node, we could use crypto to do the same thing
    result = require('crypto').randomBytes(5).toString('hex')
  }

  // pad the result with zero to make sure is always the same length (11 chars in ou\
r case)
  if (result.length < 11) {
    result = result.padStart(11, '0')
  }

  return result
}
```

Note: this is a very simple function that leverage the web browser native `crypto` api to generate a rabdin string. If the browser does not support `crypto` (or maybe you want to consume this package in a node.js app), you could either throw an error (commented out in the above code) or leverage node.js crypto library. We also make syre the string is always 11 chars long, and if not we leverage the string `padStart` method to pad the start of the string with zeros

Add also a barrel index.ts file to just export the code from random-id/random-id.ts.

random-id unit tests

Create the directory `src\tests\random-id` and here add a file called `randomid.test.ts` with the following:

```
// file: src/tests/random-id/randomid.test.ts

import { randomid } from '../../helpers'

describe('id', () => {

  it('should return value with expected length', () => {
    const result = randomid()
    expect(result.length).toEqual(11)
  })

  it('should return expected value', () => {
    // testing 10,000 different ids
    const attempts = 10000
    const results = []
    for (let i = 0; i < attempts; i++) {
      const value = randomid()
      results.push(value)
    }

    const distinctResults = new Set([...results])
    expect(results.length).toEqual(distinctResults.size)
  })
})
```

Here we have a unit test that ensure the result from randomid() is of the expected length, which is 11 chars. We also have a unit test that invokes randomid() ten thousand times and then checks that the distinct results count matches the results count. If these do not match it means that randomid is in some cases returning a non-unique id and thus fail.

Note: we leverage the JavaScript Set to get rid of potential duplicates.[43]

Before we try to run our tests, let's install `vitest` and additional unit-test depedencies we need with `npm install -D vitest @types/jest jsdom` and then add the following 2 new commands to the package.json scripts section:

```
// file: package.json

{
  "name": "@largescaleapps/my-js-helpers",
  "version": "0.1.2",
  "type": "module",
  "scripts": {

    ...

    "test": "vitest run",
    "test-watch": "vitest watch",

    ...
```

Now finally execute the command `npm run test` and it should output something like this:

[43] https://dev.to/soyleninjs/3-ways-to-remove-duplicates-in-an-array-in-javascript-259o

```
iMacRetina:my-js-helpers damiano$ npm run test

> @largescaleapps/my-js-helpers@0.1.2 test
> vitest run

 RUN  v0.19.0 /Volumes/code/large-scale-apps-my-vue-vite-project/my-js-helpers

 ✓ src/tests/random-id/randomid.test.ts (2)

Test Files  1 passed (1)
     Tests  2 passed (2)
      Time  3.29s (in thread 65ms, 5063.52%)
```

Build the library

To build the library, just run the command `npm run all` (not this will also pack the library into a compressed file with .tgz extension and we could later consume from there or just by referencing the local directory)

Consuming the my-js-helpers library

Now we have to open the my-vue-project and consume our helpers library by referencing it from a local path. The easiest way is to run the following command `npm install -D file:../my-js-helpers`.

Note that we are referencing our helpers library with a relative directory path so it is important that oyu have create the my-js-helpers project at the same level of my-vue-project.

To test that we can consume our library without problems, open one of the views, maybe App.vue, and import a reference to randomid:

```
import {
    randomid
  } from 'my-js-helpers'
```

And then output the value in the UI with some HTML like:

```
<p>[randomid() result (from my-js-helpers): ${ randomid() }]</p>
```

Or maybe you could add the output to the `text` property of the SampleComp created in the previous chapter:

```
<SampleComp :text="`This is a sample component from my-component-library: {{ randomi\
d() }}`" />
```

The first one will output in the browser something like:

```
[randomid() result (from my-js-helpers): 03627536338]
```

And the second one should output something like this:

```
This is a sample component from my-component-library: 00244391593
```

Chapter 18 Recap

What We Learned

- We created a new project called my-js-helpers that will export an helper method called randomid that returns a unique id value
- We wrote some basic unit tests against our randomid helper function
- We then built this library and consumed it in our my-vue-project to display in the UI the value returned by the randomid() helper

Observations

- We did not publish our library to NPM yet.

Based on these observations, there are a few more things that can be done:

Improvements

- In the next chapter will learn how to publish our library to the NPM registry and then consume it from there

Chapter 19 - Publish a library as a NPM package

Publishing to the NPM registry is pretty straigh forward. However, there are many different options like publishing private packages etc that might also interested you. For this, is a good idea to review the official documentation here: https://docs.npmjs.com/packages-and-modules/contributing-packages-to-the-registry.

Here we'll explain only how to publish scoped public packages but will not cover private packages or unscoped packages.

Create an NPM user account

The first step will be for you to create an NPM user account, if you do not already have one. You can do this on the NPM signup page at https://www.npmjs.com/signup. If you need further help with that please see here https://docs.npmjs.com/creating-a-new-npm-user-account

Create an Organization under your NPM profile

To publish a scoped public package, I would suggest to create a fictitious organization that you can use to learn how to publish NPM packages. Once you have mastered this and are more confident, you can better organize your packages under a real organization name or publish them using your NPM username as the scope (which will have to be prefixed with the @ char).

On NPM, once you are logged in, click on your avatar and select "Add Organization +" (alternatively, you can click on Profile, then on the Organizations tab, then on the "+ Add New Organization" button). Enter a name of your choice in the Organization field and click on the Create button next to the "Unlimited public packages" option. In the next screen, where it asks if you want to invite other developers, just click Skip. Your organization is now created and will show under your Profile (Organizations tab).

Update my-js-helpers package.json

We need to scope our library name. In order to do this, you have to add a prefix to the name property in the package.json field follow by a slash character. Here you could either use your NPM username or organization name (note: you have to include the @ char at the beginning):

```
// file: package.json

{
  "name": "@your-org-name/my-js-helpers", // prefix is in the form @username/ or @or\
gname/
  "version": "0.1.21",

  ...
```

Publishing the library

First, you'll have to login to NPM with the command:

```
npm login
```

It will prompt you for username, password and email (careful: this email will be public so feel free to use an email that is different from the one used in the NPM account):

```
npm notice Log in on https://registry.npmjs.org/
Username: yourusername
Password: yourpassword
Email: (this IS public) youremail
```

Note: if you have 2FA (two-factor authentication) setup in NPM, it will also prompt you to enter an OTP code:

```
npm notice Please use the one-time password (OTP) from your authenticator application
Enter one-time password: [yourOtpCode]
```

Now you can publish the my-js-helpers package by first navigating to the root of the my-js-helpers directory, and then execute the command:

```
npm publish --access public
```

If everything goes well, your package will be published on NPM.

Consuming your NPM package

Let’s switch back to the my-vue-project code.
Here, we’ll first uninstall the current local references to the my-js-helpers library:

```
npm uninstall my-js-helpers
```

Then we can install the one form the NPM registry with:

```
npm install -D @your-org-name/my-js-helpers
```

If you run the my-vue-project everything should still work as before.

Chapter 18 Recap

What We Learned

- We learned how to publish our library as an NPM package on the NPM registry using a user-scope or organization-scope
- We learned how to install our NPM package from the NPM registry and consume it as we did before when it was installed form the local directory
- We learned how to bump the version of our NPM package and publish a new version on NPM

Observations

- We did not publish the other library we created in Chapter 17 which is a component library (my-component-library)

Based on these observations, there are a few more things that you could try:

Improvements

- You can try to publish also the my-component-library as an NPM package and then consume it from NPM

(More Chapters Coming Soon)

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. **order-history**

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/order-history**, **models/locales** etc

For components, they are organized by component domain or functionality, i.e. **components/items**, **components/locales**, **components/order-history** 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**

Primitive components will be under the directory **primitives** (src/primitives).

File Names

In general, files will be named with a pascal-case convention, I.e. **OrderHistory.ts**

Barrel files will always be named **index.ts** (all lower case)

Files that export one instance of a class, or serve as a provider/factory will be also named **index.ts** (as long as the folder in which they are contained specify the domain/rea name, i.e. **http-client/index.ts**)

Interface File Names

Files containining interfaces will follow the convention **[Name].interface.ts**, i.e. **Item.interface.ts**.

Components File Names

Higher-order components files will be under **src/components** directory. Their names follow the convention **[ComponentName].component.vue**. I.e. **ItemsList.component.vue**

Primitive components will be under **src/primitives**. Their names follow the convention **El[ComponentName].vue**. I.e. **ElButton.vue**, **ElTextbox.vue**, etc

Views/Pages File Names

Views files will be under **src/views** directory.
Their names follow the convention **[ViewName].vue** (*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].test.ts**. I.e. **ItemsList.test.ts** *(NOTE that test against models/classes will be stored under tests/unit directory, while a test against a component will be located where each corresponding component is)*

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].test.ts** and here are a couple of examples:

- **ItemsList.permissions.view.no.ts** (to test when user does not have View permisions)
- **ItemsList.permissions.view.yes.ts** (to test when user has View permisions)

Directory src

Contains the **Vue** source code

- src/assets: contains static assets like image files etc (organized in further sub-directories)

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 live API end-points

src/components: contains the higher order components (while primitives are within a sub-directory)

- **src/components/[lowercase-component-name]**: directory 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** *(NOTE: this is not a strict requirement. Might have multiple sub-directory at the same level as children with more specific names for more complex component that have many child components)*

src/components/primitives: contains all the primitives (i.e. buttons, inputs, etc) organized in sub-directories by families:

- buttons
- icons
- etc // add more directories as you keep building your primitives foundation

src/models: contains all the pure TypeScript interface/types/classes/models/etc (extension .ts)

- **src/models/[domain]**: contains all the interfaces/classes/etc that are related to a particular domain, I.e. **items**

src/store: contains the state manager implementation

- **src/store/[domain]**: contains the store module implementation for a specific domain, I.e. **items**

src/views: contains all the views, except for the **App.vue** which is located directly under **src/**

Directory tests/unit

Contains all the unit tests that are not for components. (each component unit test will be located at the same level of the corresponding component)

- **tests/unit**: contains the unit tests against TypeScript models (not components) organized in sub-directories by domain/area or however you see fit

Resources

Websites

Official Website:
https://vuejs.org

Official Vite Website:
https://vitejs.dev

Official TailwindCSS Website:
https://tailwindcss.com

Official Vitest Website

https://vitest.dev

Vue Testing Library

https://testing-library.com/docs/vue-testing-library/intro

Tutorials

Official Vue Getting Started

https://vitejs.dev/guide

… more coming soon

Blogs

Atomic Design

… coming soon

Books

… coming soon