

Quickly Quasar

Quasar, a Vue Widget Framework

[Haris Hashim](#)

© 2018 harishashim@gmail.com

Overview

A step by step training manual style instruction to accompany mentor and mentee discussion and learning session. This is a simple introduction to Quasar.

IMPORTANT This document does not include explanation that is done during one to one mentorship session.

Table of Contents

Part 1: Introduction to Vue	2
Prerequisite	2
Steps	2
Exercise	3
Part 2: Getting Started	4
Prerequisite	4
Steps	4
Part 3: Template, Declaring Data, and Handling Click using Methods	5
Prerequisite	5
Configuring ESLint	5
Steps	6
Part 4: Layout, Page, Routes and Components	7
Prerequisite	7
Wall of Texts (Here be Dragons!)	7
Layout and Page Structure	7
Importing Built-in Quasar Components	8
Camel Case and Kebab Case	8
Steps	9
Part 5: Watch, Computed, Props & Events	11
Prerequisite	11
Steps	11
Part 6: State Using Vuex and REST Using Axios	15
Prerequisite	15
Steps	15

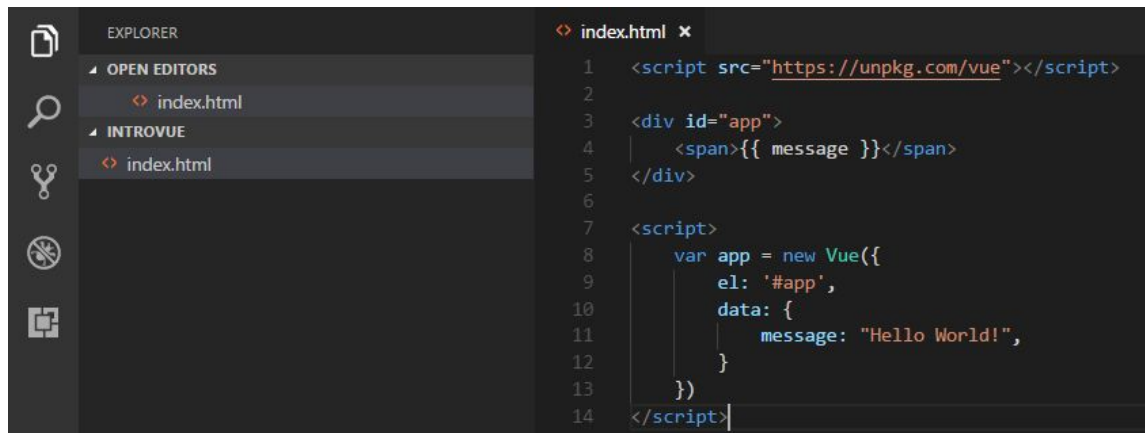
Part 1: Introduction to Vue

Prerequisite

1. VS Code installed
2. Chrome Browser installed
3. Internet connection

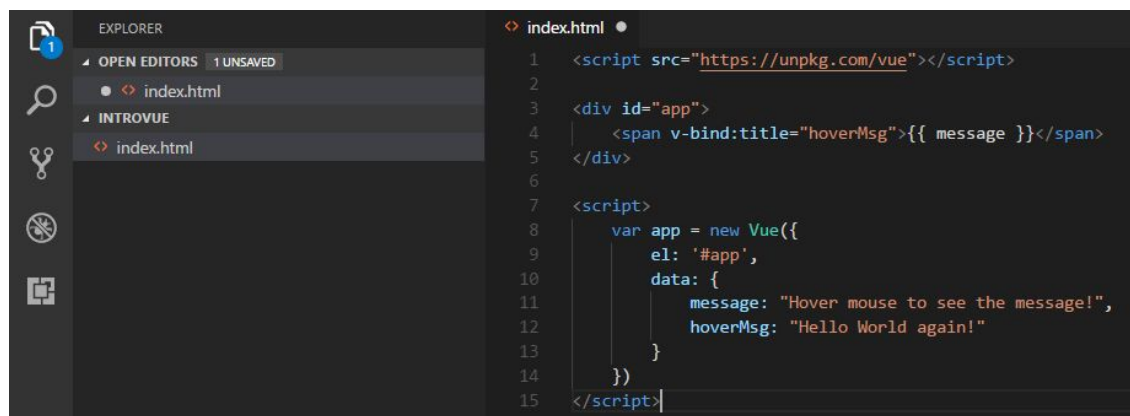
Steps

1. Create a root folder to contain all of our project. Let's name this folder QuicklyQuasar.
2. Create another folder inside QuicklyQuasar for this the tutorial and name it introvue.
3. Open introvue folder using VS Code and Create index.html file with the following contents:



```
1 <script src="https://unpkg.com/vue"></script>
2
3 <div id="app">
4   <span>{{ message }}</span>
5 </div>
6
7 <script>
8   var app = new Vue({
9     el: '#app',
10    data: {
11      message: "Hello World!",
12    }
13  })
14 </script>
```

4. Save and open file in Chrome to see result. If the file is already opened, refresh using F5.
5. Open Chrome Dev Tool by pressing F12 key on the keyboard and open the Console tab so that you can type some JS code interactively.
6. Check the message variable value using app.message
7. Change message value by setting it to "Hi World!". Observe what is meant by reactive.
8. In above step text interpolation is used to reactively render variable value on a page. Another technique is bind element attribute. Do bellow code to implement this technique.



```
1 <script src="https://unpkg.com/vue"></script>
2
3 <div id="app">
4   <span v-bind:title="hoverMsg">{{ message }}</span>
5 </div>
6
7 <script>
8   var app = new Vue({
9     el: '#app',
10    data: {
11      message: "Hover mouse to see the message!",
12      hoverMsg: "Hello World again!"
13    }
14  })
15 </script>
```

9. Refresh Chrome and repeat step to check and change hoverMsg variable value to "Hi World again!"
10. The usage of v-bind as attribute is called **directive**. This is special attribute provided by Vue. Let's practice another directive called v-if directive.
11. Add bellow code after the message element.

```

3   <div id="app">
4     <span v-bind:title="hoverMsg">{{ message }}</span>
5     <span v-if="seen">Now you see me</span>
6   </div>
7

```

12. To show your understanding, add another variable to data called seen and set the value to false.
13. Save and refresh chrome, you will not see the new element because seen is false. Change the value to true and observe what will happen.
14. And v-for directive to loop an element base on array variable.

```

7   <ol>
8     <li v-for="todo in todos">
9       {{ todo.text }}
10    </li>
11  </ol>

```

15. With bellow array variable in data.

```

22  todos: [
23    { text: 'Learn JavaScript' },
24    { text: 'Learn Vue' },
25    { text: 'Build something awesome' }
26  ]

```

16. Save and refresh Chrome to see the to do list.
17. Add a new to do list item by running bellow code in Chrome Developer Tools.

```
app.todos.push({ text: 'New item' })
```

18. Observe Vue reactive nature when the new item is added immediately!

Exercise

1. This part of the training manual is a shorter version of [Introduction To Vue](#). As a take home exercise, continue with the link to learn more. Instead of redoing the introduction as in the link, adapt it to all of the above step in introvue index.html file.

Part 2: Getting Started

Prerequisite

1. Node JS and NPM installed. Node version must be 8.9.0 or newer.

Steps

1. Open command prompt or console in QuicklyQuasar folder as created in part 1.
2. Install Quasar command line interface by executing in command prompt

```
npm install -g quasar-cli
```

3. Install Vue command line interface by executing in command prompt

```
npm install -g vue-cli
```

4. Install the default starter kit. This command will also create the quickly folder.

```
quasar init quickly
```

5. Changing directory to quickly folder generated above and execute.

```
quasar dev
```

6. Open quickly folder as project workspace using VS Code and try to understand the codes.

- a. src/App.vue
- b. src/router/router.js
- c. src/layouts/default.vue
- d. src/pages/index.vue
- e. src/pages/404.vue

7. Rather than following above complex structure, we are going to do something simpler.

- a. Open another terminal (if needed) and create a new .vue file by doing

```
quasar new component HelloWorld
```

- b. In src/router/router.js, change the route for path "/" to point to HelloWorld

```
1 export default [  
2   {  
3     path: "/",  
4     component: () => import("components/HelloWorld")  
5   },  
6 ]
```

- c. Save file to see hot reload in action.

8. Change title in HelloWorld.vue from "My component" to "Hello World!", save file to see hot reload in action again.

Part 3: Template, Declaring Data, and Handling Click using Methods

Prerequisite

This part is continuing part 2. Code changes for layout and script is done in `src/components/HelloWorld.vue` file.

Configuring Eslint

Problem with eslint is that it give lots of error. This error does not confirm with formatting style executed by code formatter (when we use the ALT-SHIFT-F key). Paste bellow config in `eslintrc.js` file in rules section.

```
// Custom config
"space-before-function-paren": [0, "never"],
'semi': 0,
'brace-style': 0,
'quotes': 0,
'indent': 0,
'key-spacing': 0,
'no-tabs': 0,
'no-mixed-spaces-and-tabs': 0,
```

```
37     "import/export": 2,
38     "import/extensions": 0,
39     "import/no-unresolved": 0,
40     "import/no-extraneous-dependencies": 0,
41
42     // Custom config
43     "space-before-function-paren": [0, "never"],
44     semi: 0,
45     "brace-style": 0,
46     quotes: 0,
47     indent: 0,
48     "key-spacing": 0,
49     "no-tabs": 0,
50     "no-mixed-spaces-and-tabs": 0,
51
52     // allow debugger during development
53     "no-debugger": process.env.NODE_ENV === "production" ? 2 : 0
54   }
55   };
```

For every file with script tag, add following configuration exactly and after the opening script tag :

```
/* eslint brace-style: 0 */
```

Steps

1. **Template.** Change the template to add a button.
 - a. Put the button codes after "Hello World!" text:

```

1  <template>
2    <div>
3      Hello World!
4      <q-btn round color="secondary" @click="btnClick()">
5        <q-icon name="card_giftcard"/>
6      </q-btn>
7    </div>
8  </template>

```

- b. Save and check in browser that button is shown after the title.
2. **Declaring Data.** Rather than "hard coding" the title. We will use data.
 - a. Add title variable to data() with previously specified value.

```

13  data() {
14    return { title: "Hello there world!" };
15  }

```

- b. Replace "Hello World!" with title. Add a line break to make it look nice!

```

1  <template>
2    <div>
3      {{title}}<br>
4      <q-btn round color="secondary" @click="btnClick()">

```

3. **Handling Click.** Nothing happened if button is clicked. We need to write code in **methods**.
 - a. Add methods section and btnClick function to <script>. The code will toggle title to "Bye!" just so that we can see change happened when clicking button.

```

13  data() {
14    return { title: "Hello there world!" };
15  },
16  methods: {
17    btnClick() {
18      if (this.title === "Bye!") {
19        this.title = "Hello there world!";
20      } else {
21        this.title = "Bye!";
22      }
23    }
24  }

```

- b. Save and test the button.

Part 4: Layout, Page, Routes and Components

Prerequisite

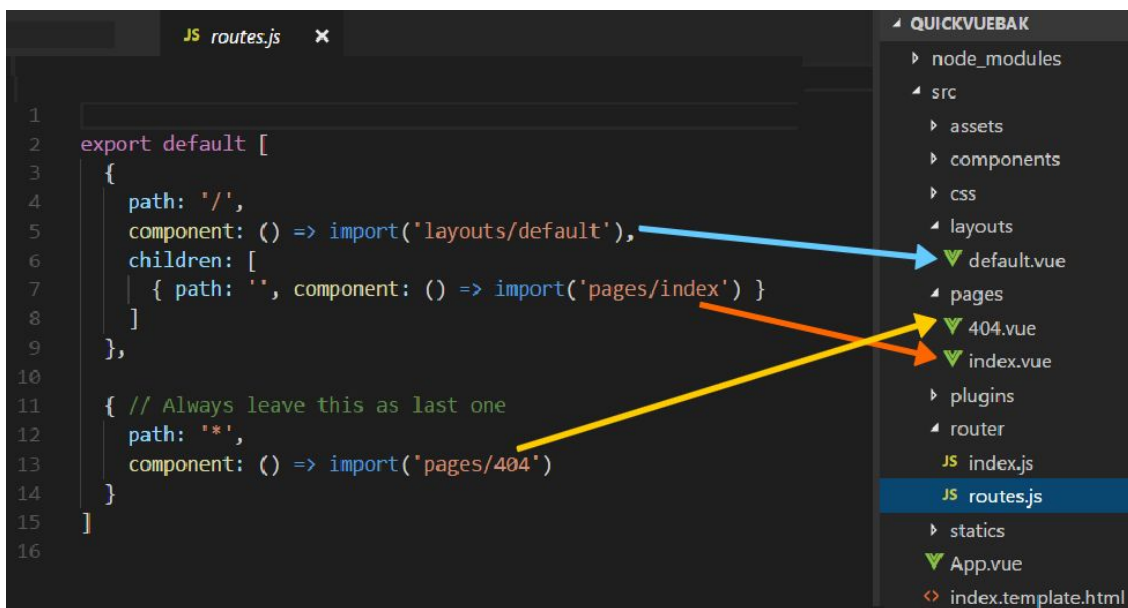
This part is continuing part 3. We will learn about proper structure for Quasar and later integrate HelloWorld component created in previous part into a page.

Wall of Texts (Here be Dragons!)

Layout and Page Structure

In Quasar 0.15, the starter kit (or simply code generated by “quasar init”) is structured into layouts and pages. Layout is the unchangeable part of a webpage while the page is the embedded contents that are different between various webpages.

Hence the source code structure is as below:



Basically, the above picture is saying that:

- The default route will have src\layouts\default.vue as the layout.
- The default page of the default route is src\pages\index.vue.
- Non existent route will display a 404 page that does not use the default layout!

Content of page will be embedded inside default vue router-view component. Refer to below picture and find the code in src\layouts\default.vue. Read more about router-view [here](#)!

```

54 <q-page-container>
55   <router-view />
56 </q-page-container>
57 </q-layout>
58 </template>
59

```


Importing Built-in Quasar Components

Quasar 0.15 add a new config file (quasar.conf.js) where built-in Quasar component can be “included” for use by .vue files. Example of already included component in quasar.conf.js file:

```

44   framework: {
45     components: [
46       'QLayout',
47       'QLayoutHeader',
48       'QLayoutDrawer',
49       'QPageContainer',
50       'QPage',
51       'QToolbar',
52       'QToolbarTitle',
53       'QBtn',
54       'QIcon',
55       'QList',
56       'QListHeader',
57       'QItem',
58       'QItemMain',
59       'QItemSide'
60     ],

```

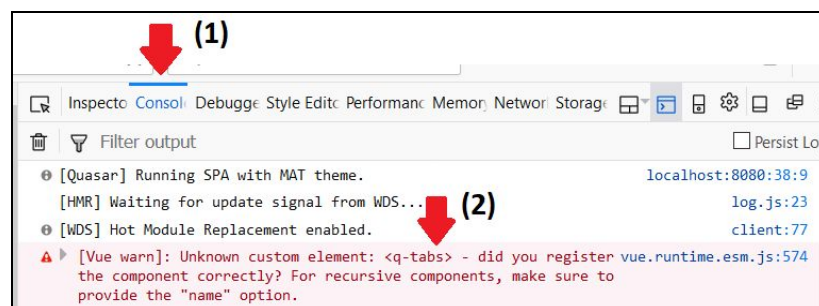
If these built-in components are not included, components will be missing when the project is run and viewed in browser. Case in point is that at step 4 in this part. We will not see footer and tabs when my_layout.vue is displayed in browser.

There are two ways to effectively handle this situation:

1. Everytime a new Quasar built-in component is added to a template, add that component to quasar.conf.js. Make sure that the component is not already in quasar.conf.js file.

OR

2. In browser, open console by pressing F12 on the keyboard. Make sure that console tab is displayed and look for “[Vue warn]: Unknown custom element: <component name>”. Add the missing component to quasar.conf.js file.



Camel Case and Kebab Case

The standard of giving variable name in script is CamelCase (for class name) or camelCase (for object name). In template, this need to be converted to kebab-case. Read more about this [here](#).

The application of this concept in the case of importing components is that <component-name> need to be converted to ComponentName. Or as per above picture, <q-tabs> will become QTabs when inserted into quasar.conf.js.

Steps

1. We will create a new **layout** by executing bellow command.

```
quasar new layout my_layout
```

2. After that add a new **page** by doing:

```
quasar new page home
```

3. Change the **route** in src\router\routes.js to use my_layout and home as the child.

```
1  export default [
2    {
3      path: "/",
4      component: () => import('layouts/my_layout'),
5      children: [
6        { path: '', component: () => import('pages/home') }
7      ]
8    },
9  ]
```

4. Run and save the route to see layout with empty page in browser. Press F12 to check for warning in browser console. Refer to "Importing Built-in Quasar Components" and fix them.
5. In the browser, type url <http://localhost:8080/about> to see the 404 page defined in route.
6. Let's change that to show 404 page inside my_layout rather than having no layout.

```
10  {
11    // Always leave this as last one
12    path: "**",
13    component: () => import('layouts/my_layout'),
14    children: [
15      { path: '', component: () => import('pages/404') }
16    ]
17  }
```

7. Save and view in browser. Take a deep breath and try to understand what you've just done!
8. As exercise, let's simplify the page:
 - a. Remove duplicate tabs in the header - simply delete <q-tabs> and everything in it.
 - b. Modify footer. Bellow code is just for footer:

```
22  <q-layout-footer>
23    <q-tabs>
24      <q-route-tab
25        slot="title"
26        icon="home"
27        to="/"
28        replace
29        label="Home"
30      />
31      <q-route-tab
32        slot="title"
33        icon="info"
34        to="/about"
35        replace
36        label="About"
37      />
38    </q-tabs>
39  </q-layout-footer>
```

9. Save and check the browser. Obviously clicking the About tab button still result to 404.
10. To add About page, again we do the new page command:

quasar new page about

11. Add "THIS IS ABOUT PAGE!" as content for src\pages\about.vue. Save the file.
12. At the child route for "/about" in src\router\routes.js. Take note child path is without /.

```

2   {
3     path: "/",
4     component: () => import("layouts/my_layout"),
5     children: [
6       {
7         path: "", component: () => import("pages/home")
8       },
9       {
10        path: "about", component: () => import("pages/about")
11      }
12    ]
13  },

```

13. Save and test to make sure that About tab button already work to navigate to about page.
14. Test by clicking Home tab button which display an empty page because we left it empty.
15. Let's change that by embedding HelloWorld **component** created in previous part:
 - a. Change src/pages/home.vue to import HelloWorld component from src/components/HelloWorld.vue and export it so that it can be used in template:

```

8   <script>
9     import HelloWorld from "components/HelloWorld";
10
11   export default {
12     // name: 'PageName',
13     components:{
14       HelloWorld
15     }
16   }
17 </script>

```

- b. Add HelloWorld to home.vue template. Take note that the tag is <hello-world> :

```

1   <template>
2     <q-page padding>
3       <!-- content -->
4       <hello-world></hello-world>
5     </q-page>
6   </template>
7

```

- c. Save and check the changes in browser. Home now display HelloWorld component.
16. Feel free to take a deep breath and realize how easy it is to create components and add them to a page. This is the power of Quasar!

Part 5: Watch, Computed, Props & Events

Prerequisite

This part is continuing Part 4. Where we will learn more about applying Vue JS in Quasar.

Steps

1. In Part 3 we have seen **methods**. Let's do **watch** by changing HelloWorld component.
 - a. We need a field (QField) and an input (QInput) component to let user key in a value. Add the 2 components to quasar.conf.js.

```

44     // framework: 'all' --- includes everything; for dev
45     framework: {
46       components: [
47         'QField',
48         'QInput',
49         'QLayout',

```

- b. In src\components\HelloWorld.vue, add the components before {{title}}

```

2     <div>
3       <q-field><q-input v-model="model" /></q-field>
4       {{title}}<br>
5       <q-btn round color="secondary" @click="btnClick()">

```

- c. Basically the input will go straight to variable called model. We need to declare it!

```

14     data() {
15       return {
16         title: "Hello there world!",
17         model: ""
18       };
19     },

```

So when user enter text in the QInput component, text is stored in model variable.

- d. Watcher or watch is simply a function that wait for changes and do something.

```

17     methods: {
18       // ...
19     },
20     watch: {
21       model: function(newModel, oldModel) {
22         this.title = newModel;
23       }
24     }

```

So whenever there is input, title will change to reflect user input. Save the file and run quasar dev to see what is happening on the browser!

2. While watch is quite straightforward, **computed** is more complicated. Computed function is executed when property change. At other time, the function will just return a cached value of data. To demonstrate this let's change HelloWorld to display message and count.
 - a. First of all, let's add message to the template, just below {{title}}.

```

1  <template>
2    <div>
3      <q-field><q-input v-model="model" /></q-field>
4      {{title}}<br>
5      {{message}}<br>
6      <q-btn round color="secondary" @click="btnClick()">

```

- b. Add count data and initialize it to 0.

```

15  data() {
16    return {
17      title: "Hello there world!",
18      model: "",
19      count: 0
20    };
21  },

```

- c. The idea is to generate a message computed from model and count data.

```

21  watch: {
22    // ...
23  },
24  computed: {
25    message: function() {
26      return "Hello " + this.model + "! count is " + this.count++;
27    }
28  }

```

Whenever model change, message will be calculated as above and returned.

- d. To cover the case where no property change but we return a message. Let's set title to message using btnClick in methods.

```

15  data() {
16    return {
17      // ...
18    };
19  },
20  methods: {
21    btnClick() {
22      this.title = this.message
23    }
24  },

```

When button is clicked, message is constructed using cached value. Count will not increase, since count++ is not executed. Rather the cached value of count is used.

- e. Save and go to browser. Verify above behavior by typing input and pressing button.

Recap: So far we have seen **data**, **method**, **watch** and **computed**. Which have internal scope to a component. The next two item, Props and Events are different. In the sense that they are the bridge between a component and its parent. From Vue own [guide](#):

"In Vue, the parent-child component relationship can be summarized as props down, events up. The parent passes data down to the child via props, and the child sends messages to the parent via events.."

Note that props down means data flow is one way. When parent update props value, child will get the value. Don't write code to set props value in child. That violate the one way data flow principle.

3. Take note that in HelloWorld component, model is initialized to be an empty string. Let's make it so that parent pass a value through props that will initialize model data.
 - a. In src\components\HelloWorld.vue define who props and set model to it.

```

15   props: [
16     "who"
17   ],
18   data() {
19     return {
20       title: "Hello there world!",
21       model: this.who,
22       count: 0
23     };
24   },

```

- b. Set the props in parent template. So in src\pages\home.vue.

```

1   <template>
2     <q-page padding>
3       <!-- content -->
4       <hello-world who="PERSON" >/hello-world>
5     </q-page>
6   </template>
7

```

- c. Take note that props can be used inside the child component template as usual.

```

1   <template>
2     <div>
3       <q-field><q-input v-model="model" /></q-field>
4       {{title}}<br>
5       {{message}}<br>
6       Who = {{this.who}}<br>
7       <q-btn round color="secondary" @click="btnClick()">
8         <q-icon name="card_giftcard"/>
9       </q-btn>
10    </div>
11  </template>

```

- d. Save and refer to browser to see what's happening.

4. **Events** are used to send data or for parent to react to what happening in the children. In the following steps, we will display user input in HelloWorld component (child) inside home page (parent).

a. Let's create a data called parentWho in src\pages\home.vue.

```
17  data(){
18    return{
19      parentWho: "PERSON"
20    }
21  }
```

- b. In the template add some text to display the value of parentWho. Also change the child component to set its who prop by using parentWho. Take note that :who is a shortcut for v-bind:who="parentWho".

```
1  <template>
2    <q-page padding>
3      <!-- content -->
4      In parent, who = {{parentWho}}<p/>
5      <hello-world :who="parentWho" ></hello-world>
6    </q-page>
7  </template>
```

- c. Let's allow HelloWorld (child) component to fire an event whenever user input change. In src\components\HelloWorld.vue, event is fired by \$emit as follows:

```
31  watch: {
32    model: function(newModel, oldModel) {
33      this.title = newModel;
34      this.$emit("onModelChange", newModel);
35    }
36  },
```

The event name is onModelChange and newModel value is passed to the parent.

- d. Parent have to handle above event in src\page\home. Declare this in template:

```
1  <template>
2    <q-page padding>
3      <!-- content -->
4      In parent, who = {{parentWho}}<p/>
5      <hello-world :who="parentWho" v-on:onModelChange="onWhoChange"></hello-world>
6    </q-page>
7  </template>
```

- e. And declare onWhoChange function in methods that set the value of parentWho.

```
22  methods: {
23    onWhoChange: function(newWho) {
24      this.parentWho = newWho;
25    }
26  }
```

- f. Save and check browser to see how parent change whenever user enter an input.



Part 6: State Using Vuex and REST Using Axios

Prerequisite

This part is continuing Part 5. Where we will learn more about application global state management using Vuex and how to use that to perform REST using Axios.

Steps