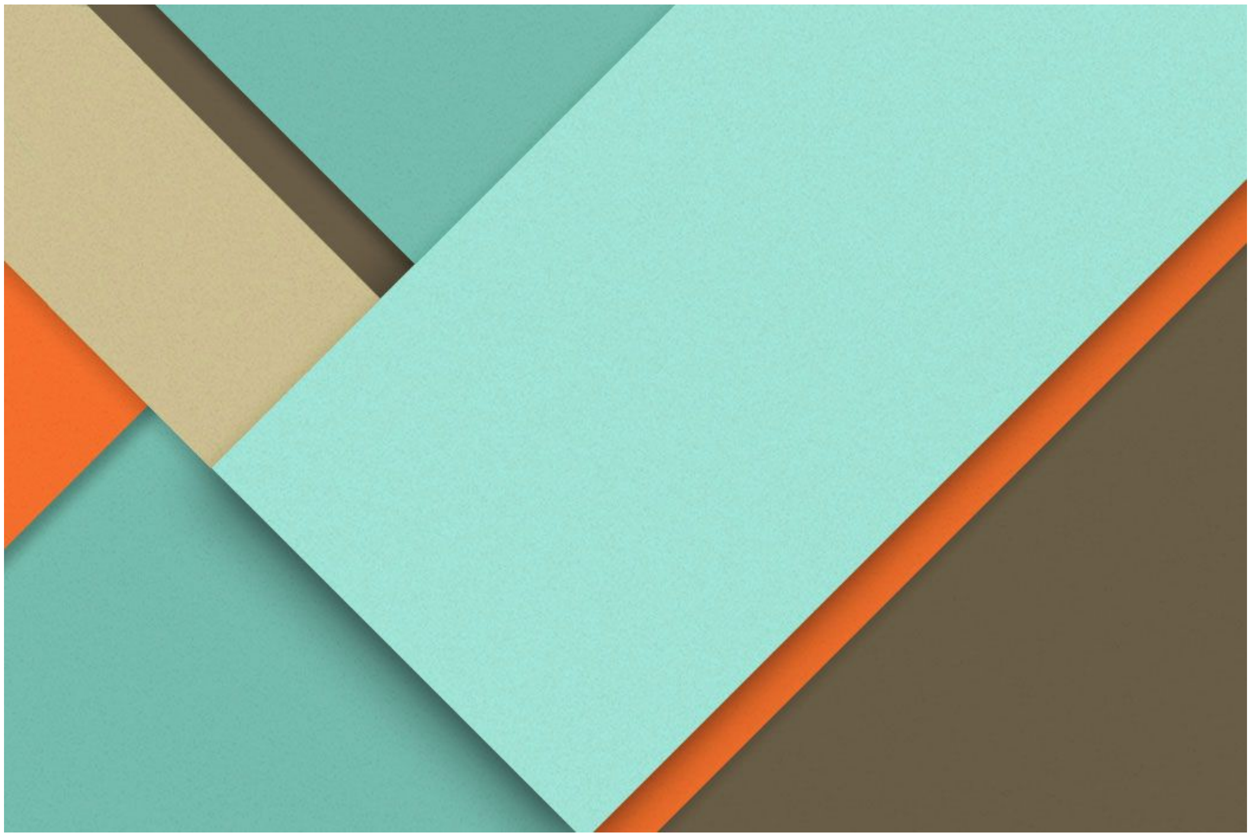


Quickly Quasar

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

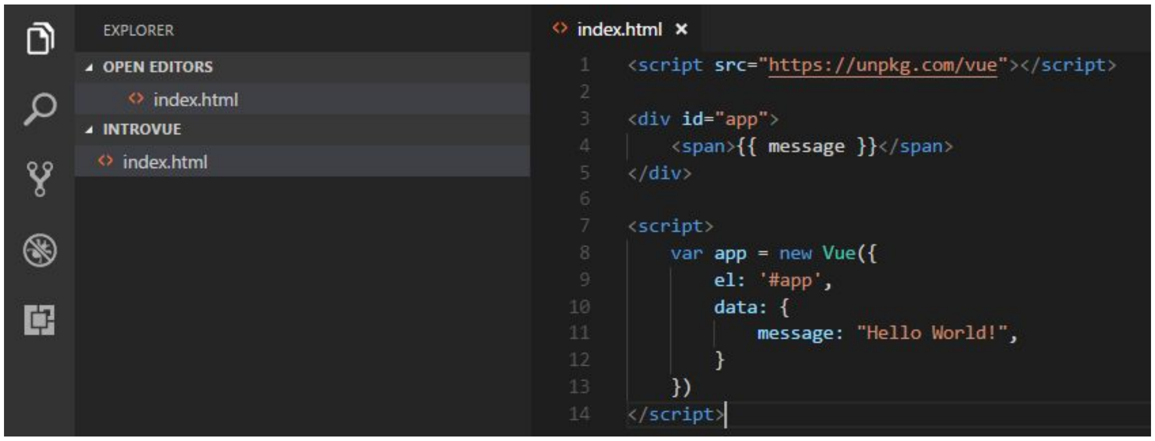
Part 1: Introduction to Vue

Prerequisite

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

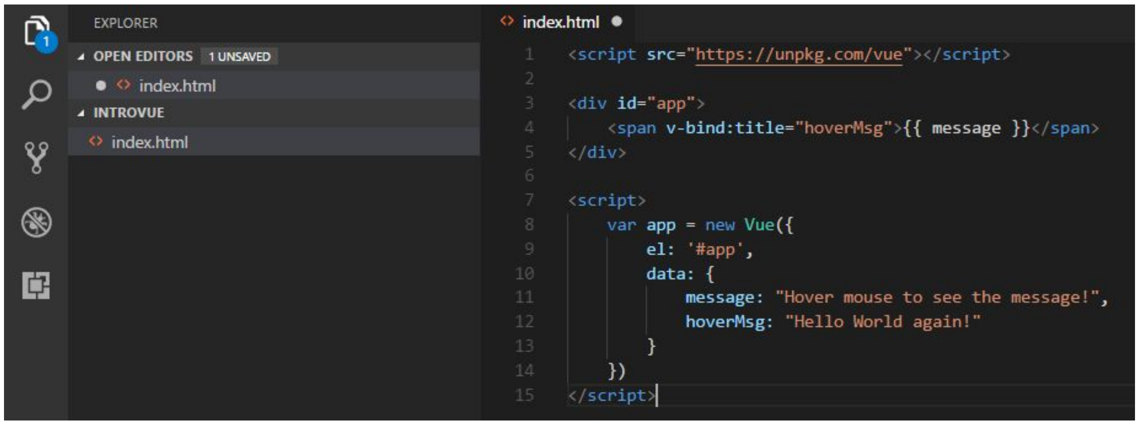
Steps

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:



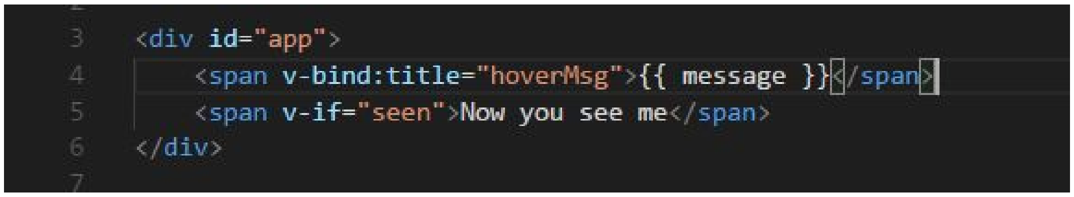
```
<script src="https://unpkg.com/vue"></script>

<div id="app">
    <span>{{ message }}</span>
</div>

<script>
    var app = new Vue({
        el: '#app',
        data: {
            message: "Hello World!",
        }
    })
</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.



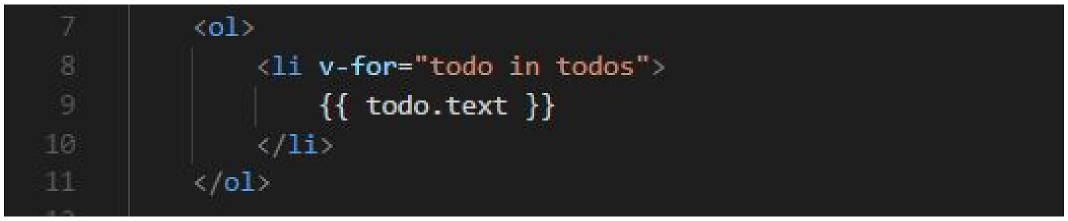
```
<script src="https://unpkg.com/vue"></script>

<div id="app">
    <span v-bind:title="hoverMsg">{{ message }}</span>
</div>

<script>
    var app = new Vue({
        el: '#app',
        data: {
            message: "Hover mouse to see the message!",
            hoverMsg: "Hello World again!"
        }
    })
</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 <span> element.



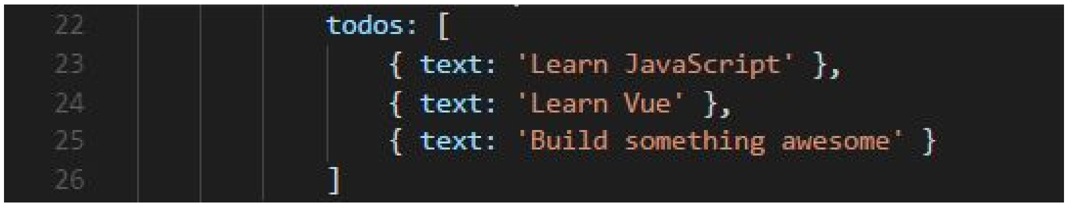
```
<div id="app">
    <span v-bind:title="hoverMsg">{{ message }}</span>
    <span v-if="seen">Now you see me</span>
</div>
```

12. To show your understanding, add another variable to data called seen and set the value to false.
13. Safe and refresh chrome, you will not see the new <span> 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.



```
    <ol>
        <li v-for="todo in todos">
            {{ todo.text }}
        </li>
    </ol>
```

15. With bellow array variable in data.



```
            todos: [
                { text: 'Learn JavaScript' },
                { text: 'Learn Vue' },
                { text: 'Build something awesome' }
            ]
```

16. Safe 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
```

   NOTES: When asked about features, best to choose all among the choices (ESLint, Vuex, Axios & Vue-i18n). When asked about Yarn or NPM, choose NPM!

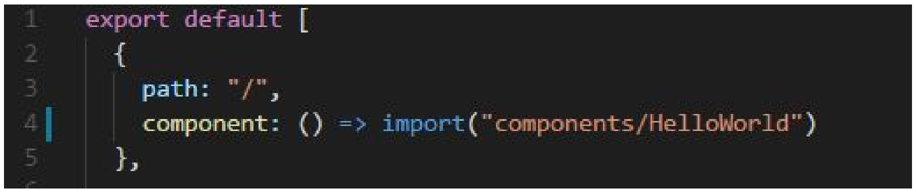
5. Changing directory to quickly folder generated above and execute.

```
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



```
export default [
  {
    path: "/",
    component: () => import("components/HelloWorld")
  },
```

   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.

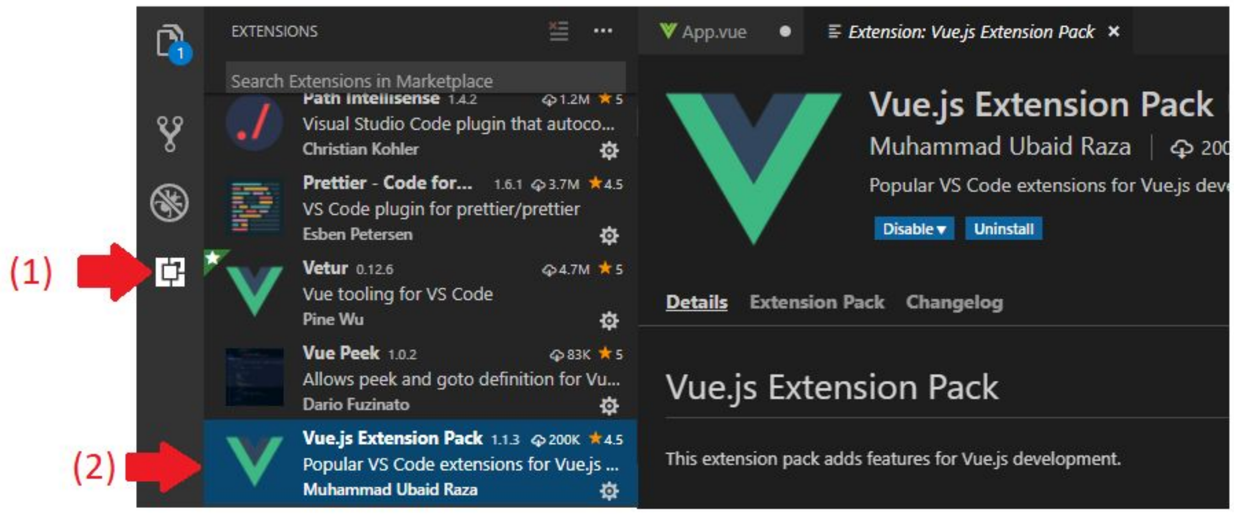
Part 3: Template, Declaring Data, and Handling Click using Methods

Prerequisite

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

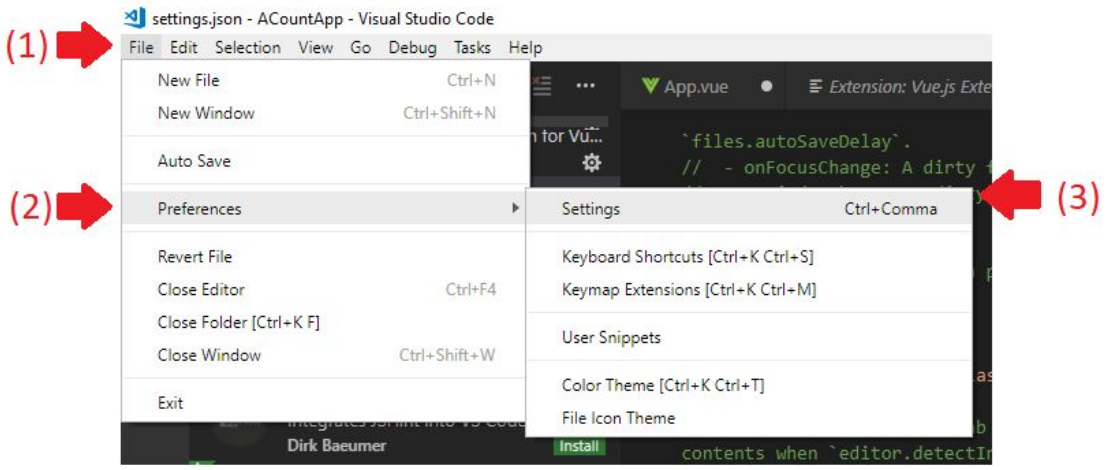
Configuring VSCode Support for Vue

To make code formatting and highlighting work for Vue in VS Code, open up the Extensions panel and install Vue JS Extension pack.


(1)
(2)
EXTENSIONS
Search Extensions in Marketplace
Path Intellisense 1.4.2 1.2M 5
Visual Studio Code plugin that autoco...
Christian Kohler
Prettier - Code for... 1.6.1 3.7M 4.5
VS Code plugin for prettier/prettier
Esben Petersen
Vetur 0.12.6 4.7M 5
Vue tooling for VS Code
Pine Wu
Vue Peek 1.0.2 83K 5
Allows peek and goto definition for Vu...
Dario Fuzinato
Vue.js Extension Pack 1.1.3 200K 4.5
Popular VS Code extensions for Vue.js ...
Muhammad Ubaid Raza
App.vue
Extension: Vue.js Extension Pack
Vue.js Extension Pack
Muhammad Ubaid Raza
Popular VS Code extensions for Vue.js deve
Disable
Uninstall
Details
Extension Pack
Changelog
Vue.js Extension Pack
This extension pack adds features for Vue.js development.

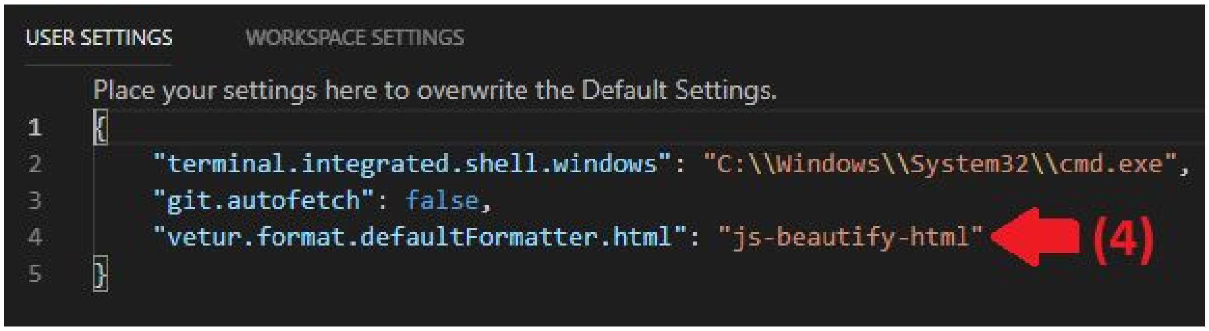
However this will supply only half of the solution. Since only Javascript part of the code will be formatted. Vue file part that is HTML template will not receive auto formatting treatment when ALT-SHIFT-F is pressed!

This is PITA because linter will complain empty line in the HTML template indented by space. Work around for this is to enable js-beautify-html by doing bellow steps.



```
`files.autoSaveDelay`.
//  - onFocusChange: A dirty
contents when `editor.detectI
```

And doing the configuration: "vetur.format.defaultFormatter.html": "js-beautify-html"



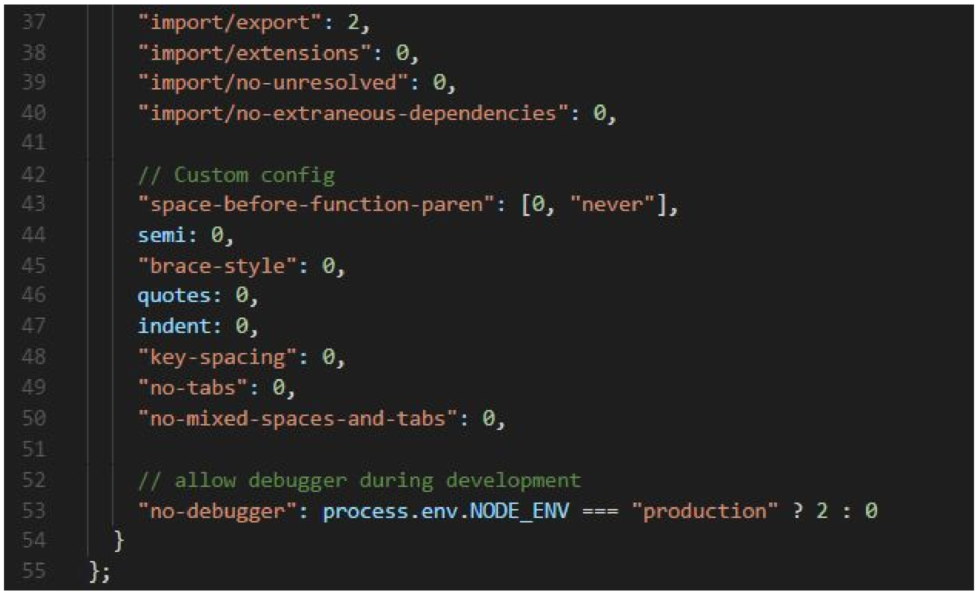
Place your settings here to overwrite the Default Settings.

```
{
    "terminal.integrated.shell.windows": "C:\\Windows\\System32\\cmd.exe",
    "git.autofetch": false,
    "vetur.format.defaultFormatter.html": "js-beautify-html"
}
```

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,
```



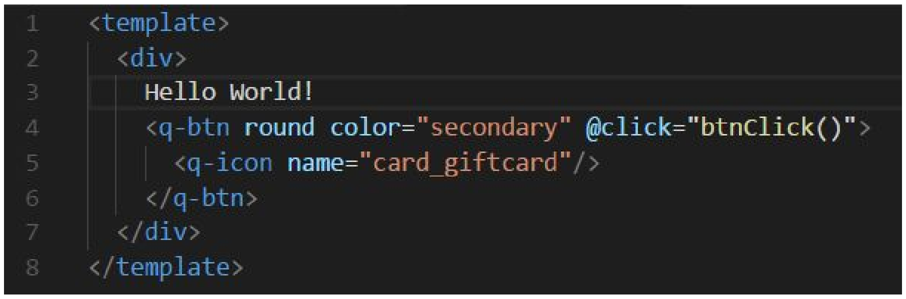
```
    "import/export": 2,
    "import/extensions": 0,
    "import/no-unresolved": 0,
    "import/no-extraneous-dependencies": 0,

    // 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,

    // allow debugger during development
    "no-debugger": process.env.NODE_ENV === "production" ? 2 : 0
  }
};
```

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

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

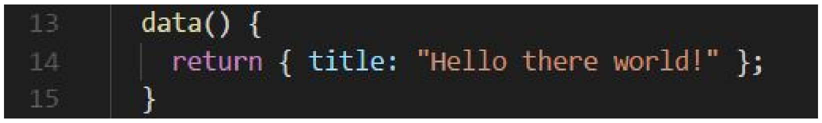
Steps

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



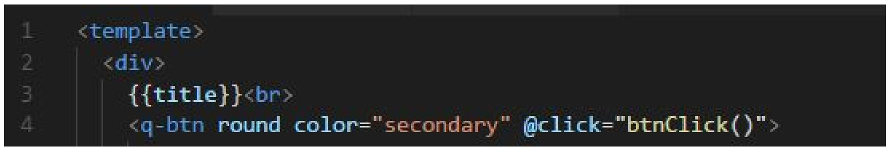
```
<template>
  <div>
    Hello World!
    <q-btn round color="secondary" @click="btnClick()">
      <q-icon name="card_giftcard"/>
    </q-btn>
  </div>
</template>
```

   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.



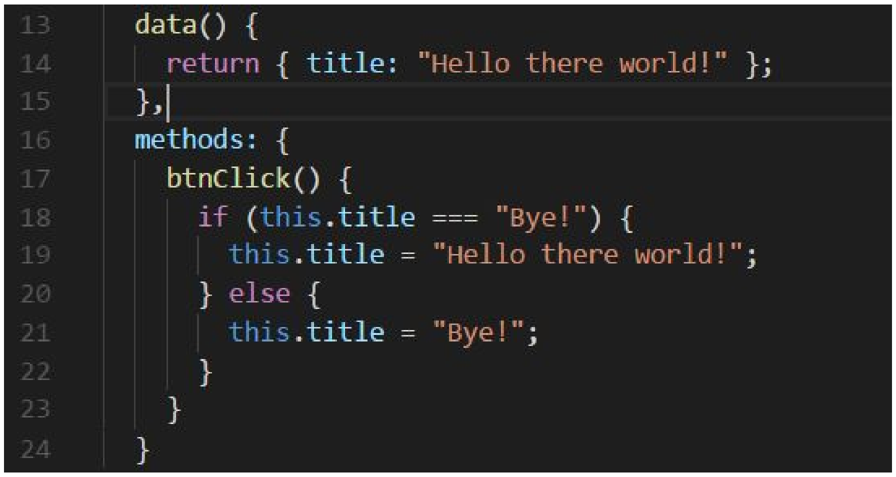
```
data() {
  return { title: "Hello there world!" };
}
```

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



```
<template>
  <div>
    {{title}}<br>
    <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.



```
data() {
  return { title: "Hello there world!" };
},
methods: {
  btnClick() {
    if (this.title === "Bye!") {
      this.title = "Hello there world!";
    } else {
      this.title = "Bye!";
    }
  }
}
```

   b. Save and test the button.

Part 4: Layout, Page, Routes and Components

Prerequisite

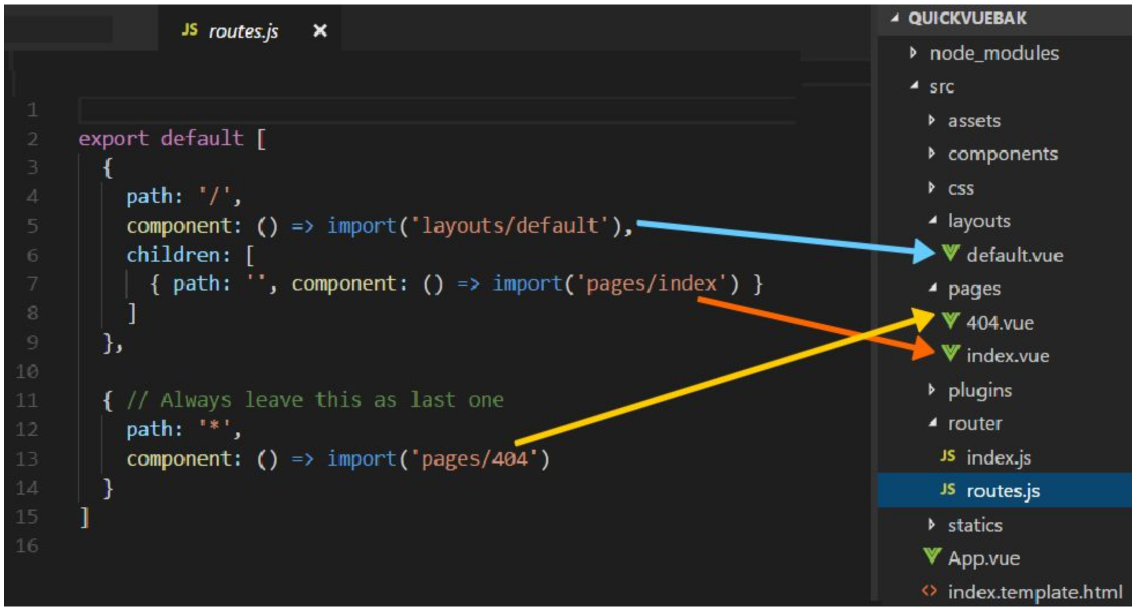
This part is continuing part 3. We will learn about proper structure for Quasar and later integrate HelloWorld component created in previous part into a page.

Wall of Texts (Here be Dragons!)

Layout and Page Structure

In Quasar 0.15 (or 0.16 or 0.17. Well you get the flow!). 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 web pages.

Hence the source code structure is as below as seen in src\router\routes.js:

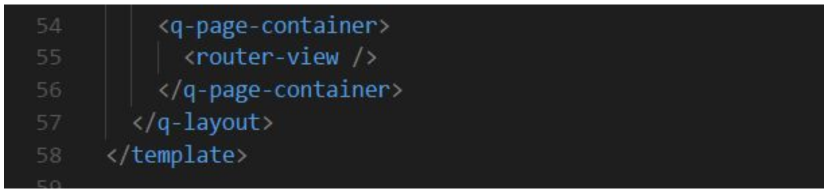


```
export default [
  {
    path: '/',
    component: () => import('layouts/default'),
    children: [
      { path: '', component: () => import('pages/index') }
    ]
  },

  { // Always leave this as last one
    path: '*',
    component: () => import('pages/404')
  }
]
```

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!

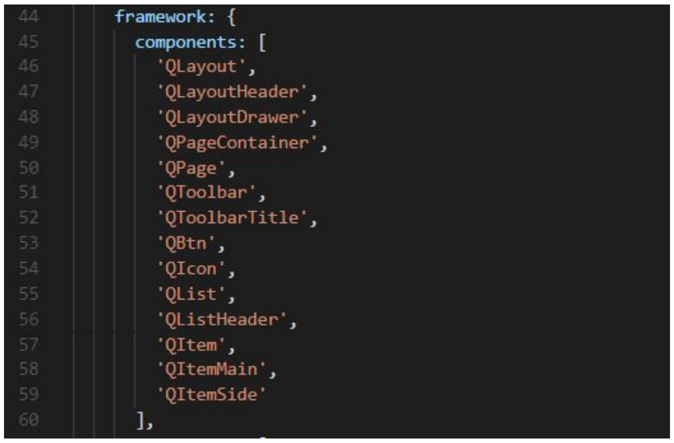
Content of page will be embedded inside default.vue router-view component. Refer to below picture and find the code in src\layouts\default.vue. Read more about router-view here!



```
    <q-page-container>
      <router-view />
    </q-page-container>
  </q-layout>
</template>
```

Importing Built-in Quasar Components

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



```
    framework: {
      components: [
        'QLayout',
        'QLayoutHeader',
        'QLayoutDrawer',
        'QPageContainer',
        'QPage',
        'QToolbar',
        'QToolbarTitle',
        'QBtn',
        'QIcon',
        'QList',
        'QListHeader',
        'QItem',
        'QItemMain',
        'QItemSide'
      ],
```

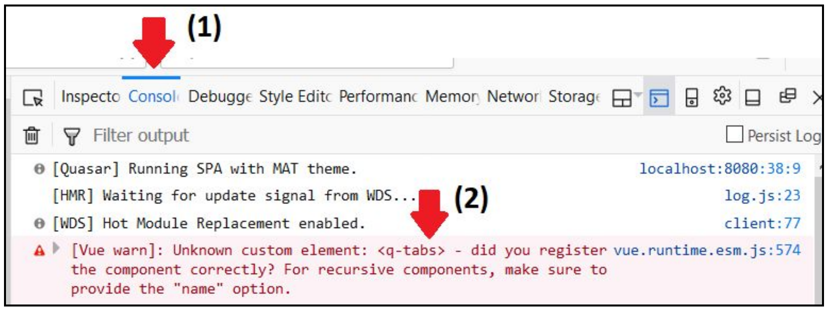
If these built-in components are not included, components will be missing when the project is run and viewed in browser. Case in point is that at step 4 in this part. We will not see footer and tabs when my_layout.vue is displayed in browser.

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.



```
[Quasar] Running SPA with MAT theme.                     localhost:8080:38:9
[HMR] Waiting for update signal from WDS...                        log.js:23
[WDS] Hot Module Replacement enabled.                             client:77
[Vue warn]: Unknown custom element: <q-tabs> - did you register vue.runtime.esm.js:574
the component correctly? For recursive components, make sure to
provide the "name" option.
```

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

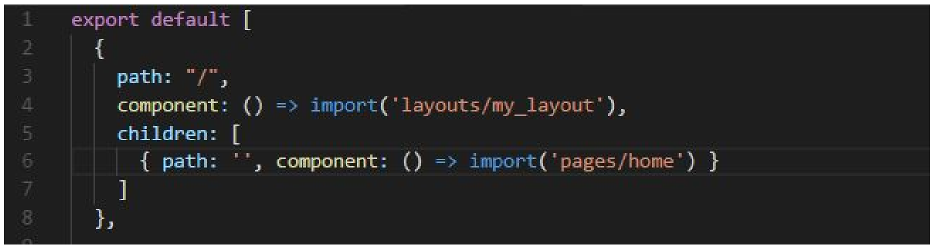
1. We will create a new **layout** by executing bellow command.

   quasar new layout my_layout

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

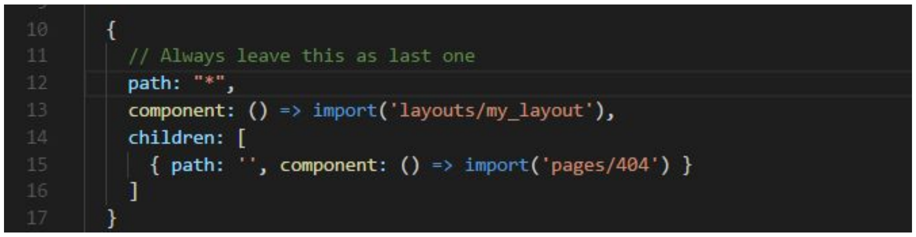
   quasar new page home

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



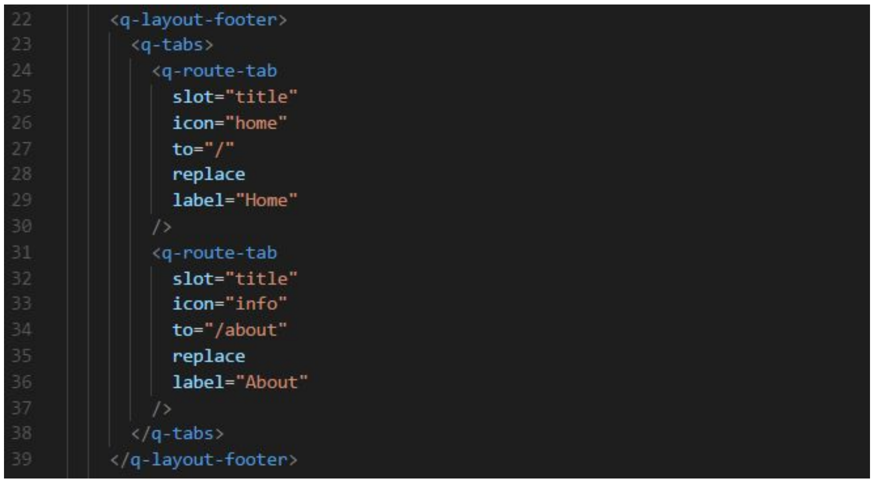
```
export default [
  {
    path: "/",
    component: () => import('layouts/my_layout'),
    children: [
      { path: '', component: () => import('pages/home') }
    ]
  },
```

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.



```
  {
    // Always leave this as last one
    path: "*",
    component: () => import('layouts/my_layout'),
    children: [
      { path: '', component: () => import('pages/404') }
    ]
  }
```

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:

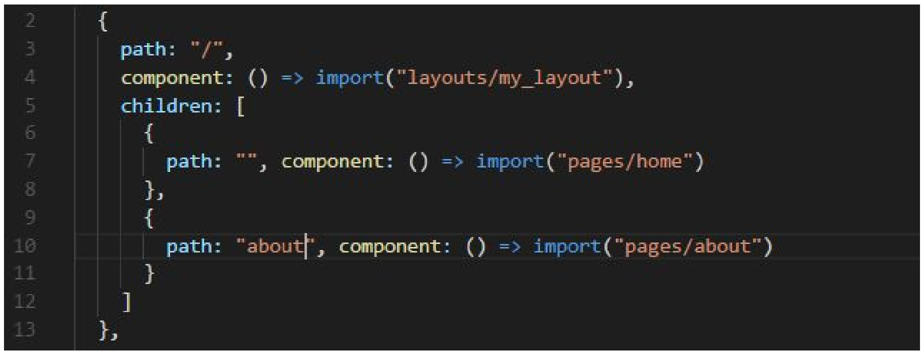


```
    <q-layout-footer>
      <q-tabs>
        <q-route-tab
          slot="title"
          icon="home"
          to="/"
          replace
          label="Home"
        />
        <q-route-tab
          slot="title"
          icon="info"
          to="/about"
          replace
          label="About"
        />
      </q-tabs>
    </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:

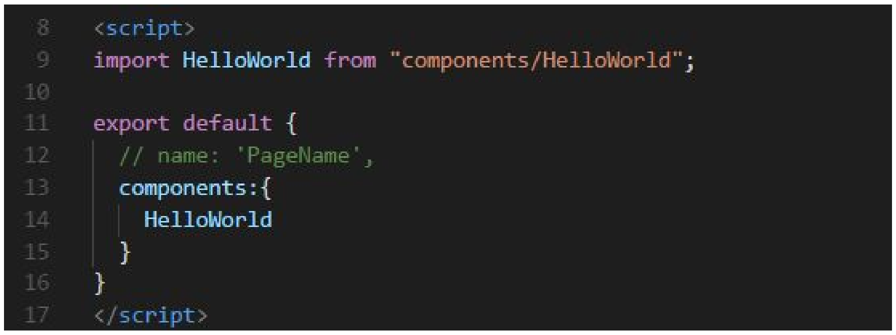
    quasar new page about

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 /.



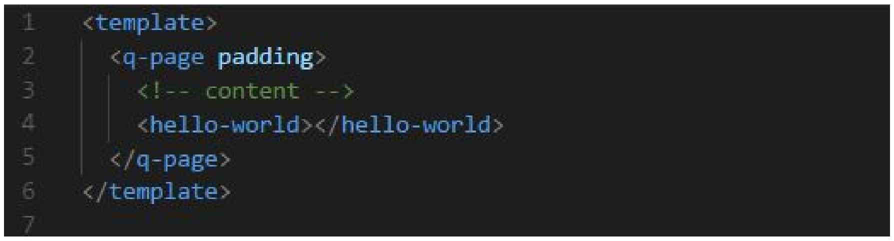
```
  {
    path: "/",
    component: () => import("layouts/my_layout"),
    children: [
      {
        path: "", component: () => import("pages/home")
      },
      {
        path: "about", component: () => import("pages/about")
      }
    ]
  },
```

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:



```
<script>
import HelloWorld from "components/HelloWorld";

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

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



```
<template>
  <q-page padding>
    <!-- content -->
    <hello-world></hello-world>
  </q-page>
</template>
```

    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!

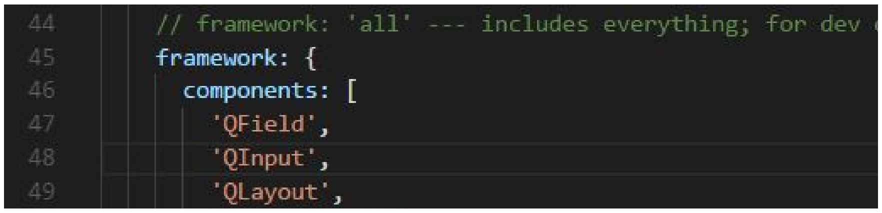
Part 5: Watch, Computed, Props & Events

Prerequisite

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

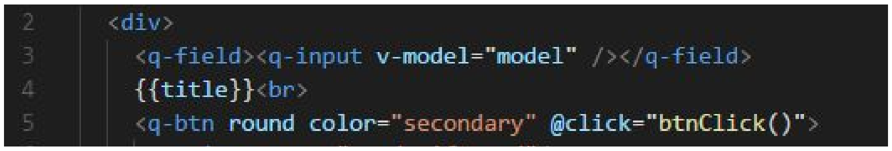
Steps

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.



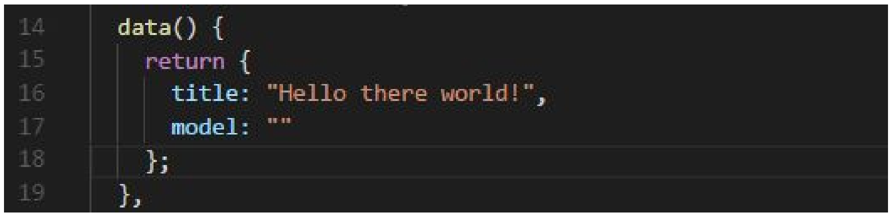
```
// framework: 'all' --- includes everything; for dev 
framework: {
  components: [
    'QField',
    'QInput',
    'QLayout',
```

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



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

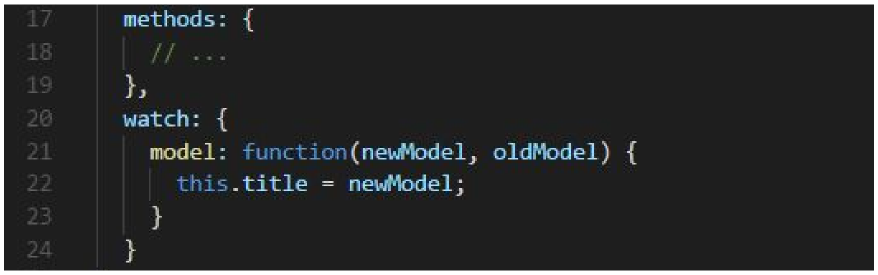
   c. Basically the input will go straight to variable called model. We need to declare it!



```
data() {
  return {
    title: "Hello there world!",
    model: ""
  };
},
```

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

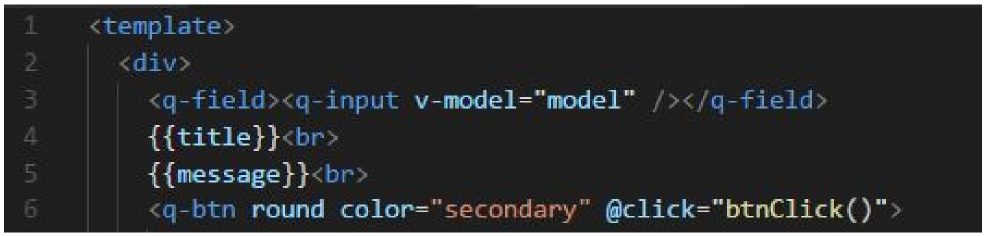
   d. Watcher or watch is simply a function that wait for changes and do something.



```
methods: {
  // ...
},
watch: {
  model: function(newModel, oldModel) {
    this.title = newModel;
  }
}
```

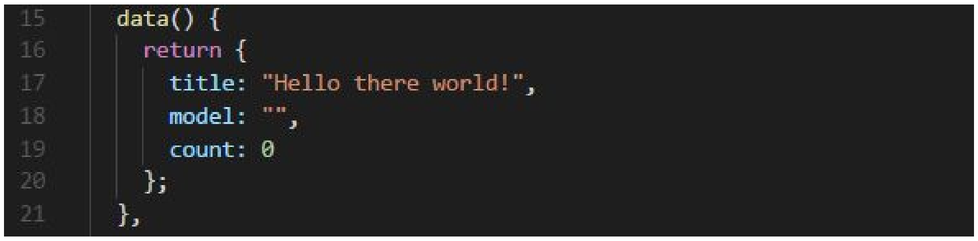
   So whenever there is input, title will change to reflect user input. Save the file and run quasar dev to see what is happening on the browser!

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}}.



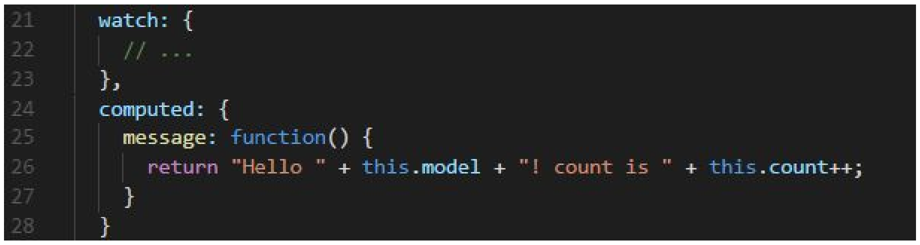
```
<template>
  <div>
    <q-field><q-input v-model="model" /></q-field>
    {{title}}<br>
    {{message}}<br>
    <q-btn round color="secondary" @click="btnClick()">
```

   b. Add count data and initialize it to 0.



```
  data() {
    return {
      title: "Hello there world!",
      model: "",
      count: 0
    };
  },
```

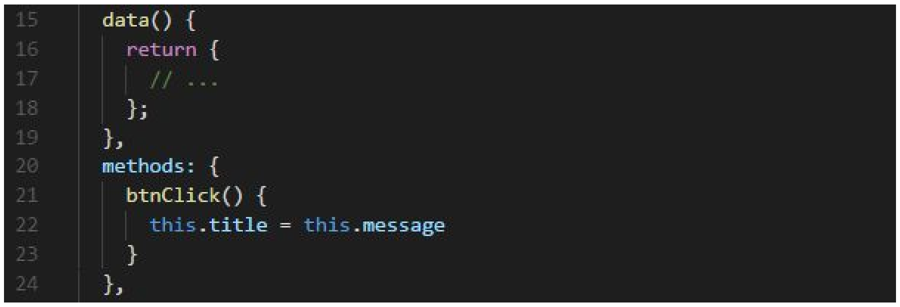
   c. The idea is to generate a message computed from model and count data.



```
  watch: {
    // ...
  },
  computed: {
    message: function() {
      return "Hello " + this.model + "! count is " + this.count++;
    }
  }
```

   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.



```
  data() {
    return {
      // ...
    };
  },
  methods: {
    btnClick() {
      this.title = this.message
    }
  },
```

   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.

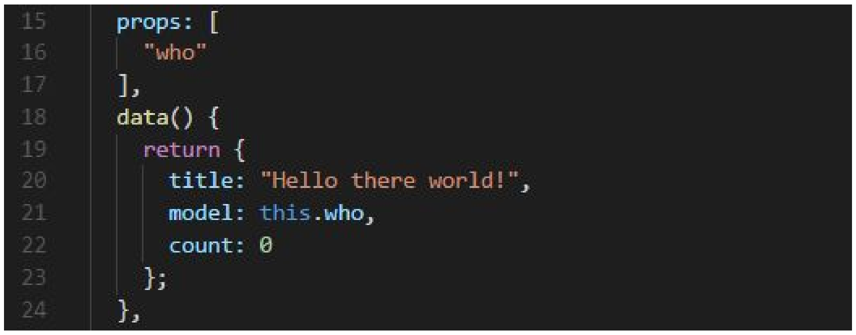
   e. Save and go to browser. Verify above behavior by typing input and pressing button.

Recap: So far we have seen **data**, **method**, **watch** and **computed**. Which have internal scope to a component. The next two items, 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..”

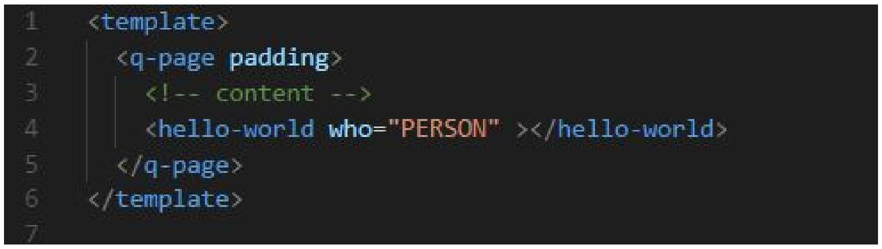
Note that props down means data flow is one way. When parent update props value, child will get the value. Don’t write code to set props value in child. That violate the one way data flow principle.

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.



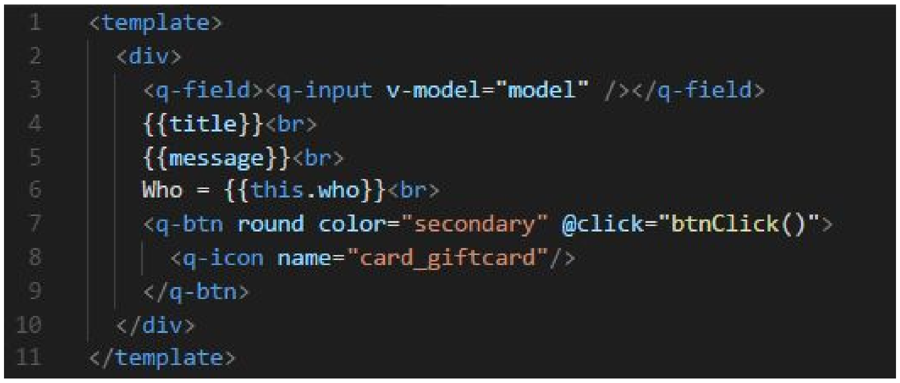
```
  props: [
    "who"
  ],
  data() {
    return {
      title: "Hello there world!",
      model: this.who,
      count: 0
    };
  },
```

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



```
<template>
  <q-page padding>
    <!-- content -->
    <hello-world who="PERSON" ></hello-world>
  </q-page>
</template>
```

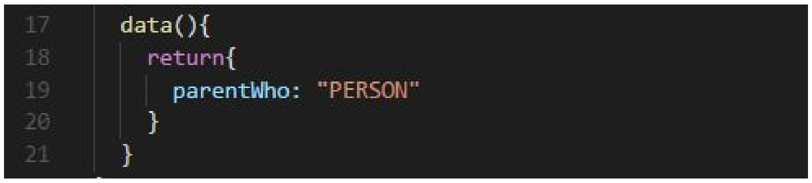
    c. Take note that props can be used inside the child component template as usual.



```
<template>
  <div>
    <q-field><q-input v-model="model" /></q-field>
    {{title}}<br>
    {{message}}<br>
    Who = {{this.who}}<br>
    <q-btn round color="secondary" @click="btnClick()">
      <q-icon name="card_giftcard"/>
    </q-btn>
  </div>
</template>
```

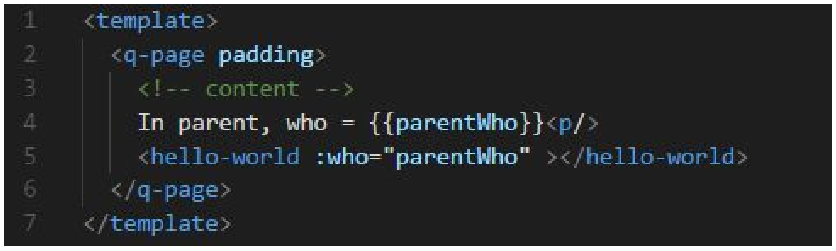
    d. Save and refer to browser to see what’s happening.

4. **Events** are used to send data or for parents to react to what is 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.



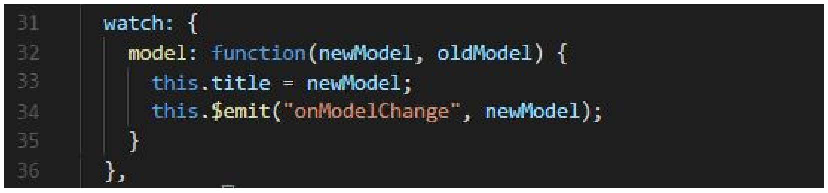
```
  data(){
    return{
      parentWho: "PERSON"
    }
  }
```

   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".



```
<template>
  <q-page padding>
    <!-- content -->
    In parent, who = {{parentWho}}<p/>
    <hello-world :who="parentWho" ></hello-world>
  </q-page>
</template>
```

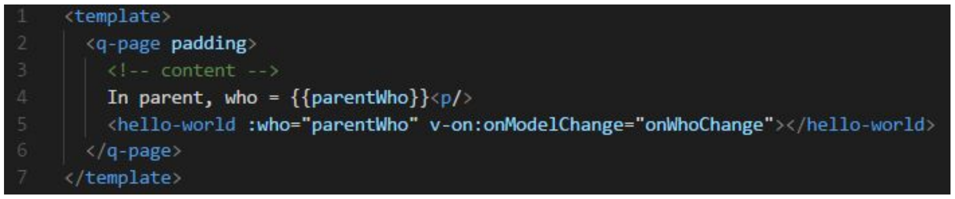
   c. Let's allow HelloWorld (child) component to fire an event whenever user input change. In src\components\HelloWorld.vue, event is fired by $emit as follows:



```
  watch: {
    model: function(newModel, oldModel) {
      this.title = newModel;
      this.$emit("onModelChange", newModel);
    }
  },
```

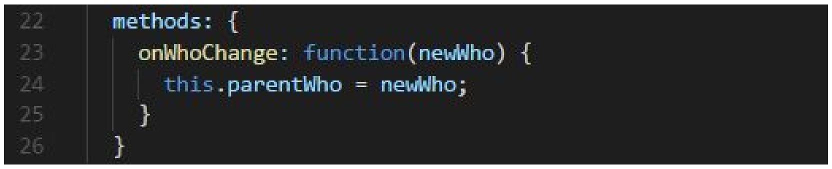
   The event name is onModelChange and newModel value is passed to the parent.

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



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

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



```
  methods: {
    onWhoChange: function(newWho) {
      this.parentWho = newWho;
    }
  }
```

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

Part 6: REST Using Axios and Global State Using VueX

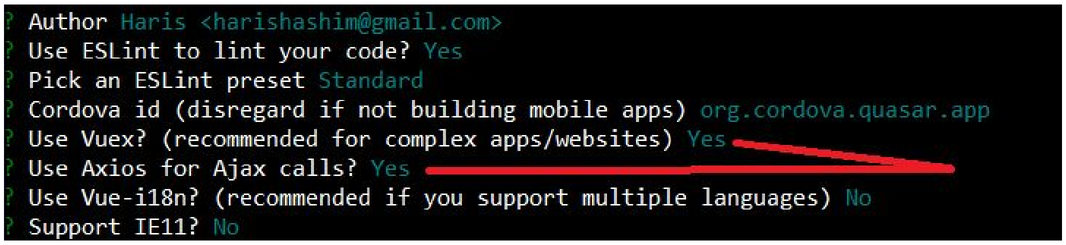
Prerequisite

This part is continuing Part 5. Where we will learn more about consuming REST service using Axios.

1. We will be using a mockup REST service from https://jsonplaceholder.typicode.com/
2. Reference to Axios is a bit hard to find but it is here!

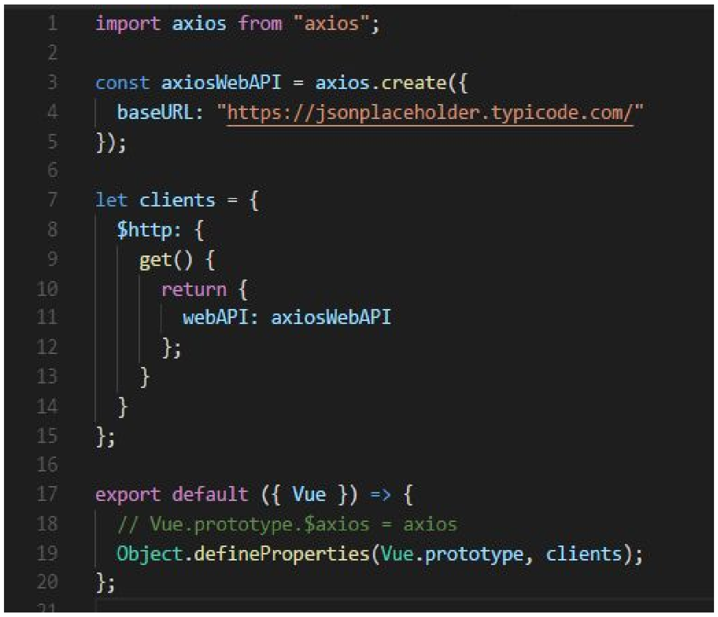
Steps

1. Create a new project named quickly2. Make sure Vuex and Axios plugin is turned on.



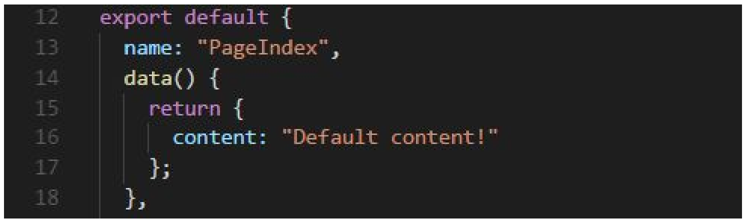
```
? Author Haris <harishashim@gmail.com>
? Use ESLint to lint your code? Yes
? Pick an ESLint preset Standard
? Cordova id (disregard if not building mobile apps) org.cordova.quasar.app
? Use Vuex? (recommended for complex apps/websites) Yes
? Use Axios for Ajax calls? Yes
? Use Vue-i18n? (recommended if you support multiple languages) No
? Support IE11? No
```

2. After the project is created, do the usual by opening up the workspace in VSCode and adding custom eslint rule in eslintrc.js file as done in Part 3.
3. Inspect the following items that is added when VueX and Axios is turned on:
    a. File src\plugins\axios.js that is added for Axios. We will be modifying this file later.
4. Modify the Axios plugin to include a reference to a mock REST service



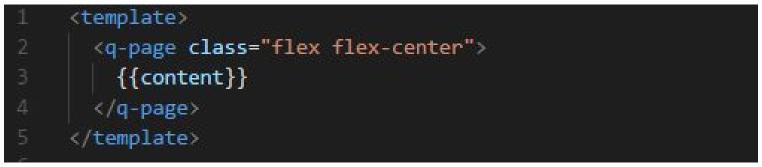
```
import axios from "axios";

const axiosWebAPI = axios.create({
  baseURL: "https://jsonplaceholder.typicode.com/"
});

let clients = {
  $http: {
    get() {
      return {
        webAPI: axiosWebAPI
      };
    }
  }
};

export default ({ Vue }) => {
  // Vue.prototype.$axios = axios
  Object.defineProperties(Vue.prototype, clients);
};
```

5. Let's test the service in our default index.vue page.
    a. Add content data into the page script.



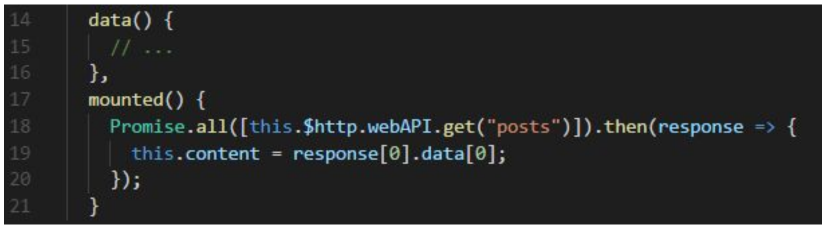
```
export default {
  name: "PageIndex",
  data() {
    return {
      content: "Default content!"
    };
  },
```

b. Modify page template to display the content.



```
<template>
  <q-page class="flex flex-center">
    {{content}}
  </q-page>
</template>
```

c. Write a function called mounted that change content data value based on REST GET result to the mock service. This function is run when page is displayed.



```
data() {
  // ...
},
mounted() {
  Promise.all([this.$http.webAPI.get("posts")]).then(response => {
    this.content = response[0].data[0];
  });
}
```

d. Save and check the page in browser.

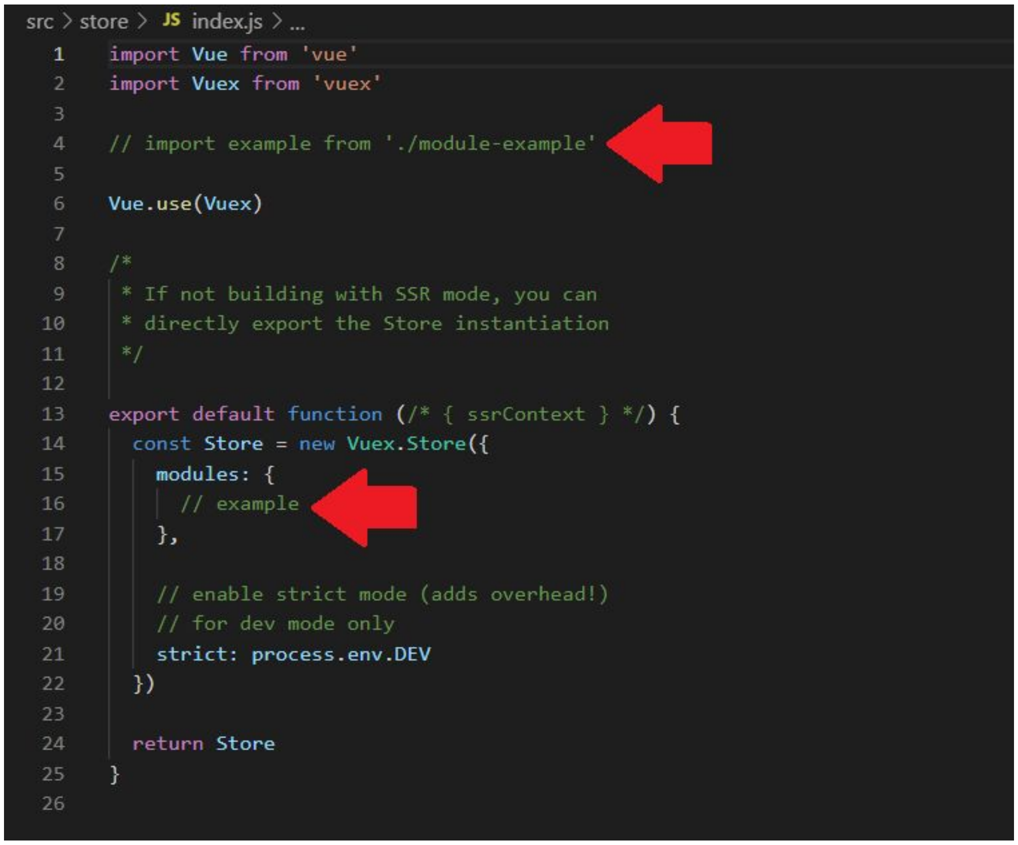
Part 7: App State Using VueX

Prerequisite

This part is continuing Part 6. Where we will learn about application global state management using Vuex. Quasar reference to Vuex can be found here.

Steps

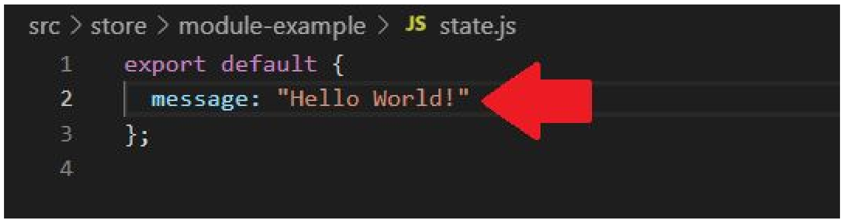
1. Inspect the following items that is added when VueX option is turned on:
    a. Folder src\store that is added for Vuex. Subfolder inside this folder is a Vuex Module. This is a way to implement multiple Vuex store rather than having only one.
2. View index.js code in store folder. Take note from the generated code that module example is disabled by default. Enable that module.



src > store > JS index.js > ...

```
import Vue from 'vue'
import Vuex from 'vuex'

// import example from './module-example'

Vue.use(Vuex)

/*
 * If not building with SSR mode, you can
 * directly export the Store instantiation
 */

export default function (/* { ssrContext } */) {
  const Store = new Vuex.Store({
    modules: {
      // example
    },

    // enable strict mode (adds overhead!)
    // for dev mode only
    strict: process.env.DEV
  })

  return Store
}
```

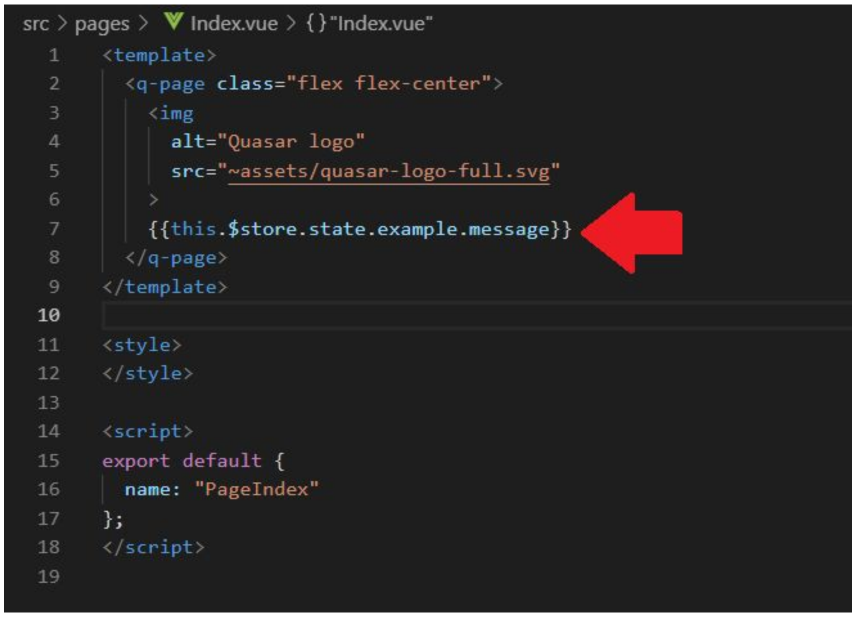
3. Now we can use example module located in module-example subfolder. Edit file state.js in module-example folder to add a message as app global state.



src > store > module-example > JS state.js

```
export default {
  message: "Hello World!"
};
```

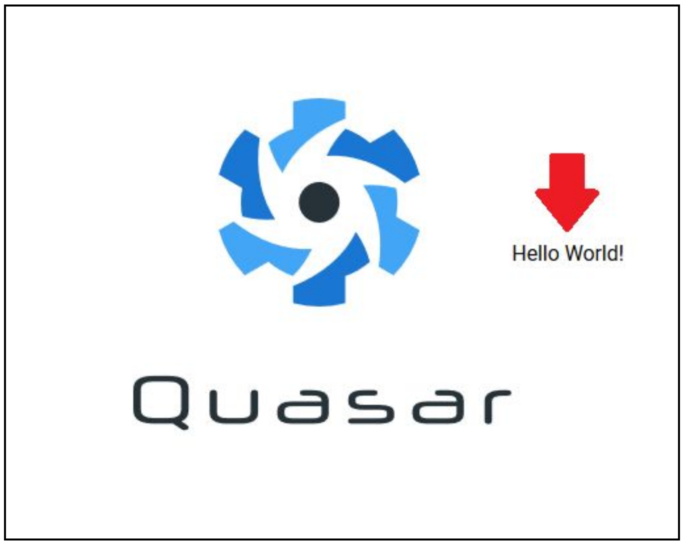
4. Add the code to use above state in index.vue file.



src > pages > Index.vue > {}"Index.vue"

```
<template>
  <q-page class="flex flex-center">
    <img
      alt="Quasar logo"
      src="~assets/quasar-logo-full.svg"
    >
    {{this.$store.state.example.message}}
  </q-page>
</template>

<style>
</style>

<script>
export default {
  name: "PageIndex"
};
</script>
```

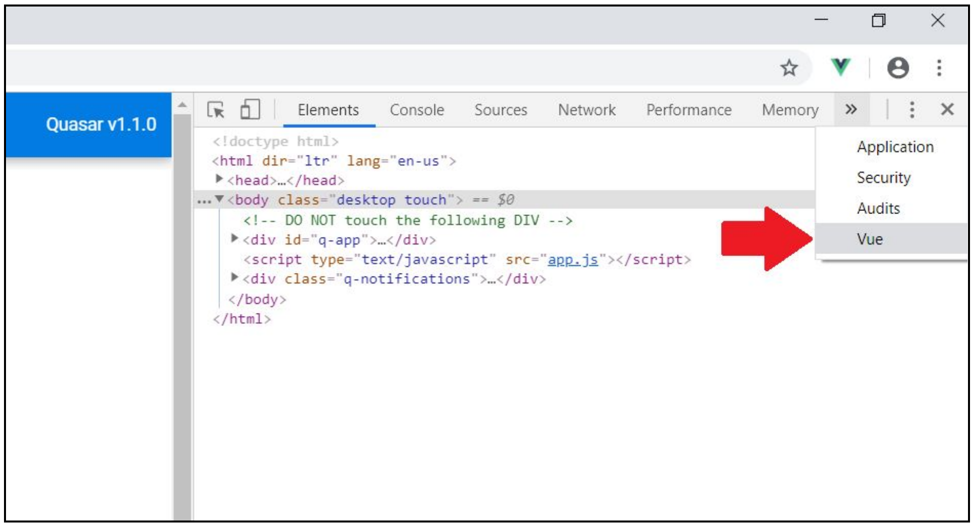
5. Notice the hello world message when we run the code.



6. Install Vue.js Devtools chrome extension as tools to debug and see VueX app state value.

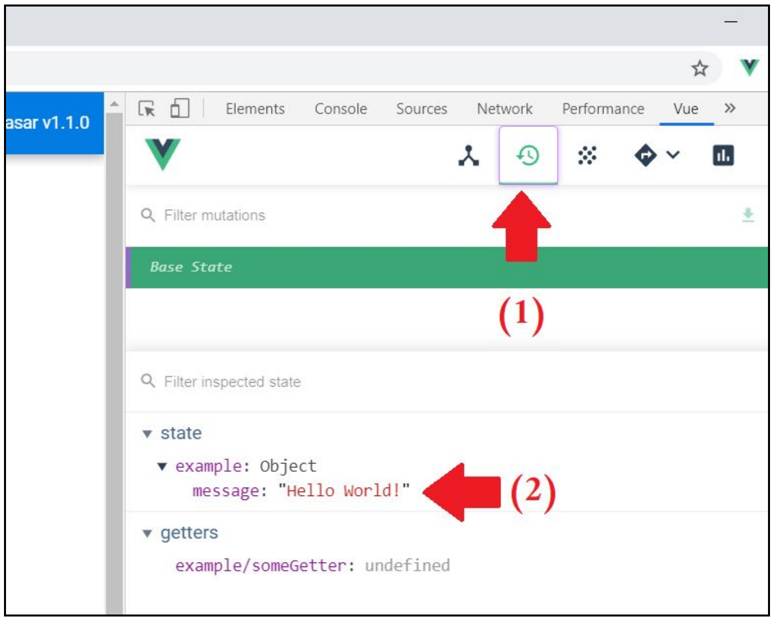
   https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd?hl=en

7. After Vue Devtools is installed and in tab where a Vue or Quasar project is running. Open up Chrome Developer Tools and navigate to Vue tab as below picture.



```
<!doctype html>
<html dir="ltr" lang="en-us">
▶<head>…</head>
…▼<body class="desktop touch"> == $0
    <!-- DO NOT touch the following DIV -->
  ▶<div id="q-app">…</div>
    <script type="text/javascript" src="app.js"></script>
  ▶<div class="q-notifications">…</div>
  </body>
</html>
```

8. Change to VueX sub tab and observe the app state as below picture.

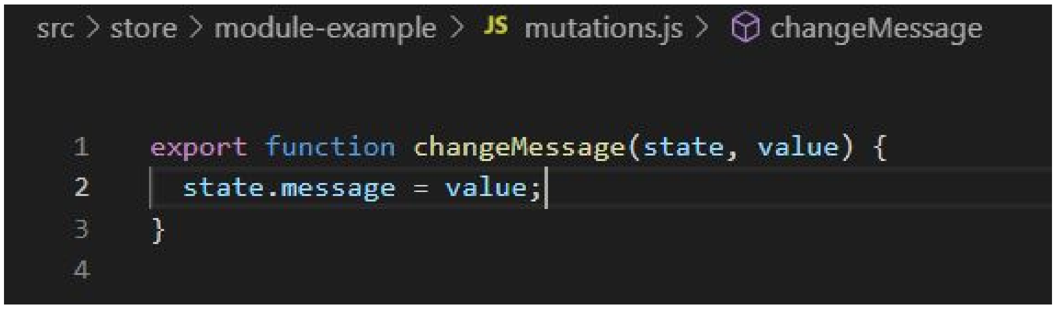


```
▼ state
  ▼ example: Object
      message: "Hello World!"
▼ getters
    example/someGetter: undefined
```

Keep referring to this Developer Tools tab as we go along in this step by step instruction!

9. Changing or mutating app state. This is done by writing a function in mutation.js inside the store module. The function is then committed in Index.vue as below example.

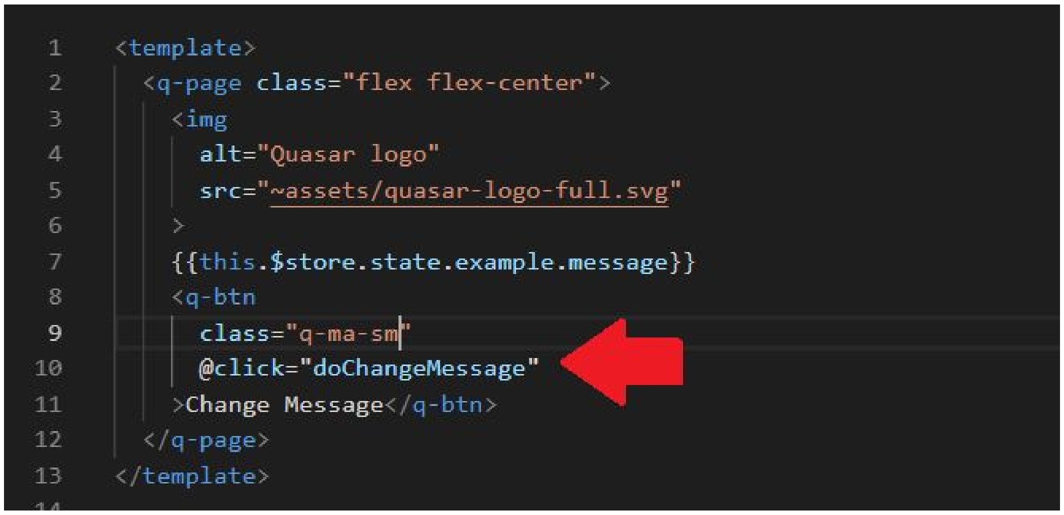
Module mutation.js



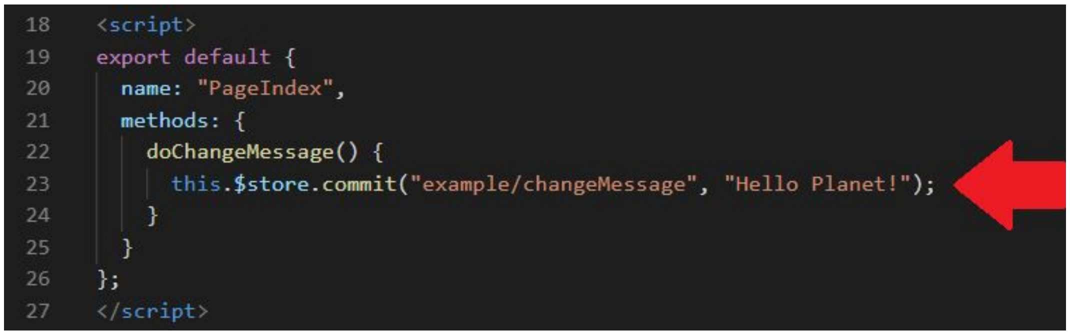
src > store > module-example > JS mutations.js > changeMessage

```
export function changeMessage(state, value) {
  state.message = value;
}
```

Index.vue page



```
<template>
  <q-page class="flex flex-center">
    <img
      alt="Quasar logo"
      src="~assets/quasar-logo-full.svg"
    >
    {{this.$store.state.example.message}}
    <q-btn
      class="q-ma-sm"
      @click="doChangeMessage"
    >Change Message</q-btn>
  </q-page>
</template>
```

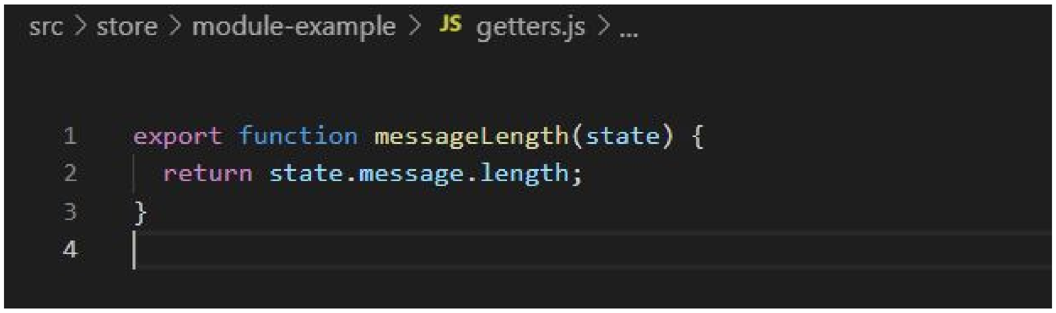


```
<script>
export default {
  name: "PageIndex",
  methods: {
    doChangeMessage() {
      this.$store.commit("example/changeMessage", "Hello Planet!");
    }
  }
};
</script>
```

Save and test the Change Message button.

10. Getter is similar in function to computed in a vue file. We will define a getter function that return message length in module getters.js file and use it inside Index.vue.

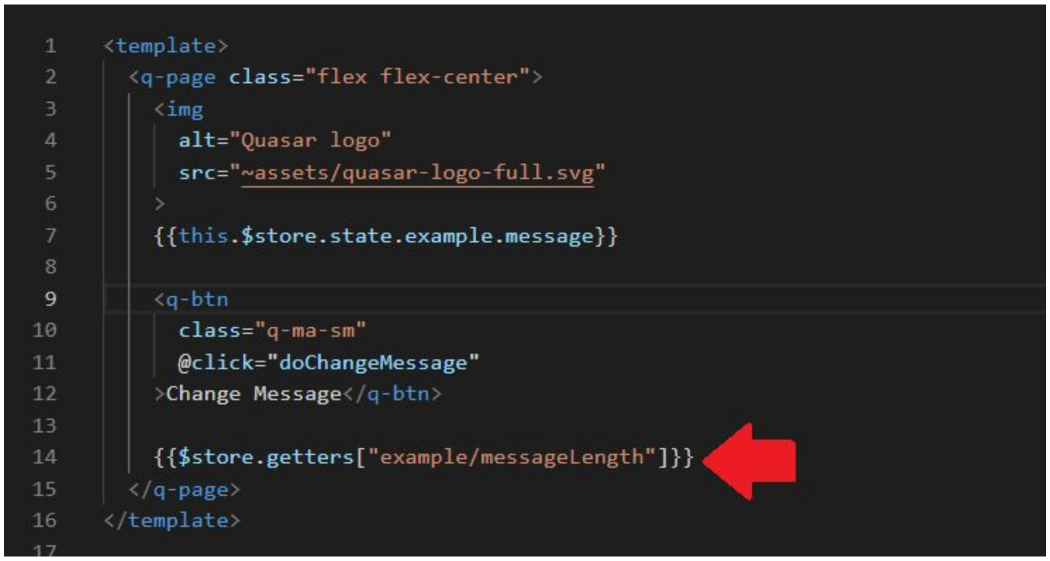
Module getters.js



src > store > module-example > JS getters.js > ...

```
export function messageLength(state) {
  return state.message.length;
}

```

Index.vue page

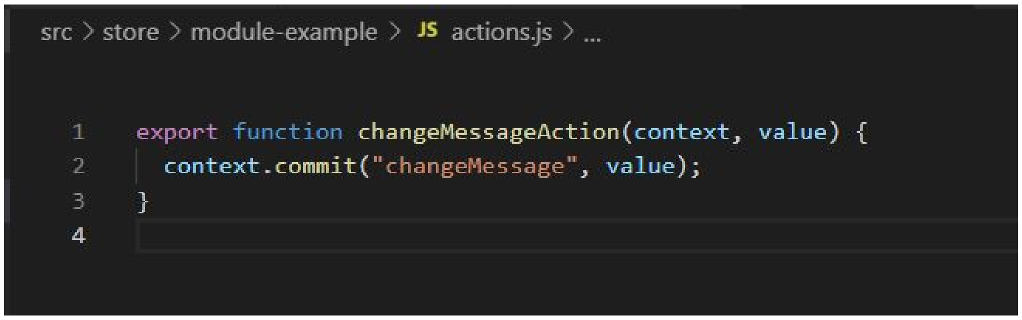


```
<template>
  <q-page class="flex flex-center">
    <img
      alt="Quasar logo"
      src="~assets/quasar-logo-full.svg"
    >
    {{this.$store.state.example.message}}

    <q-btn
      class="q-ma-sm"
      @click="doChangeMessage"
    >Change Message</q-btn>

    {{$store.getters["example/messageLength"]}}
  </q-page>
</template>
```

Save and test by clicking the Change Message button.

11. Action is a way to perform or dispatch mutation. Let's create an action function in module actions.js and dispatch it from Index.vue.

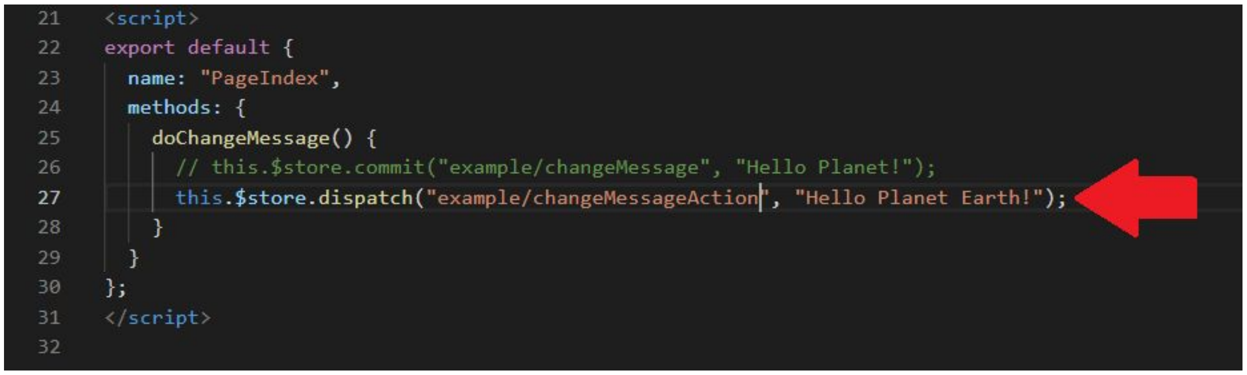
Module actions.js



src > store > module-example > JS actions.js > ...

```
export function changeMessageAction(context, value) {
  context.commit("changeMessage", value);
}

```

Index.vue page



```
<script>
export default {
  name: "PageIndex",
  methods: {
    doChangeMessage() {
      // this.$store.commit("example/changeMessage", "Hello Planet!");
      this.$store.dispatch("example/changeMessageAction", "Hello Planet Earth!");
    }
  }
};
</script>

```

Save and test by clicking the Change Message button.

The takeaway here is that, if commit (mutation) and dispatch (action) is doing the same thing. Why is action is needed?

Take note that mutation is synchronous while action can be asynchronous. This is another way to say that we can call REST API inside action!

We can also perform multiple mutation inside an action. So action is indeed something kewl and needed.

A more rigorous example can be seen in this url => VueX Actions.

(TO BE CONTINUED)