Table of Contents

Introduction 1.1

Basics 1.2

- The simplest use case 1.2.1
- Vue-router 1.2.2

Advanced 1.3

- Vuex 1.3.1

Welcome to Learning Vue.js

Vue.js is a Javascript library to help building complex user interfaces for the browser. It implements the pattern MVVM (Model View - View Model). Out-of-the-box it provides rich features such as **two-way data binding** and **directives**. You can create **components**, encapsulating behavior, structure (HTML) and styling (CSS), allowing code reuse.

It is light weight, plays nicely with jQuery and can be used with module bundlers such as Webpack or Browserify.

There are two use cases for Vue.js:

1. Enhance an existent pages by adding reactive components
2. Create a single page application (requires add-ons)

In both cases it is a good choice for its simplicity of use and flat learning curve.

This book means to be a collection of short how-to like tutorials.

The content is released in no particular order.

Author

Fabio Vedovelli

Basics

1. The simplest use case
2. Vue-router

The simplest use case

Vue.js is very good to work with data. It provides means to access, modify and validate it. Furthermore it acts as the glue between HTML elements and the data, removing from the developer the annoying task of getting references to DOM elements just to show/hide them or read their values.

This bare bones example will just enable/disable the submit button when all form fields are filled. It is a simple HTML page with Vue.js included in a script tag.

The setup:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <script src="http://cdnjs.cloudflare.com/ajax/libs/vue/1.0.25/vue.min.js"></script>

    <script>
        new Vue({
            el: 'body'
        })
    </script>
</body>
</html>
```

Nothing special is going on at the moment: close to the `</body>` tag is a javascript include and the instantiation of a Vue object. As the only parameter passed to the Vue() method is an object literal containing some configuration information. In the example above the property `el` attaches the Vue object to the `<body>`. Everything contained within body will be available to Vue.js.

Next we need a form:

```
...
<body>
    <form action="#">
        <div>
            <label for="name">Name</label>
            <input type="text" id="name" name="name">
        </div>
        <div>
            <label for="email">E-mail</label>
            <input type="email" id="email" name="email">
        </div>
        <div><button type="submit">Submit</button></div>
    </form>
</body>
...
```

What is the our goal here? **To keep the submit button disabled until both fields are filled**. Think for a moment: "- How would you accomplish this if working with Vanilla JS or jQuery?" Probably you'd get a reference to both fields and read their values, right? Well, with Vue.js it is a little bit different.

First we are going back to the Vue object instantiation to create an object with the data we are interested in...

```
...
    new Vue({
        el: 'body',
        data: {
          name: '',
          email: ''
        }
    })
...
```

... and then attach that data to our form:

```
...
  <input type="text" id="name" name="name" v-model="name">
  <input type="email" id="email" name="email" v-model="email">
...
```

Pay attention to the new attribute `v-model` included in both input fields. They contain a direct reference to the properties contained in our data object living inside the Vue object.

This property is a **directive** and its purpose is to tie the data to the HTML element. By doing this, everything typed in the field will change the data in the Vue object. The opposite is also true and if the information is changed in the Javascript, let's say, by the return of an AJAX call, the new value will automatically be displayed in the form field.

This handy behaviour is called **two-way data binding**.

We are getting there!

Now it is time to work on the button, afterall, he is our main character here. To enable the desired behaviour we are going to use another directive: `v-bind`. This directive conditionally add properties to HTML elements based on the state of our data. In this particular case we'll make use of the property `disabled`:

```
...
<div><button type="submit" v-bind:disabled="(name != '' && email != '')">Submit</button
></div>
...
```

By the configuration above Vue is instructed to do the following: add the disabled attribute to the `<button>` when both **name** and **email** are not empty. Thanks to the two-way data binding, this is performed live and automatically. **This is the power of Vue!**.

But this is kind of dirty, don't you think so? What if we want to validate e-mail structure before considering the form as valid for submission? Well, there's a better way to implement this validation: **computed property**.

```
...
    new Vue({
        el: 'body',
        data: {
          name: '',
          email: ''
        },
        computed: {
          isValid: function () {
            return this.name != '' && this.email != ''
          }
        }
    })
...
```

```
...
<div><button type="submit" v-bind:disabled="!isValid">Submit</button></div>
...
```

Once again thanks to the reactivity nature of Vue.js every time one of the properties mentioned in the method `isValid` changes the method gets executed, returning true or false. Because this is a regular method you can perform any other validation inside it.

Below you find the complete source code. If you wanna see it in action there's a fiddle available here: https://jsfiddle.net/vedovelli/focs85v3/

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <form action="#">
        <div>
            <label for="name">Name</label>
            <input type="text" id="name" name="name" v-model="name">
        </div>
        <div>
            <label for="email">E-mail</label>
            <input type="email" id="email" name="email" v-model="email">
        </div>
        <div><button type="submit" v-bind:disabled="!isValid">Submit</button></div>
    </form>
    <script src="http://cdnjs.cloudflare.com/ajax/libs/vue/1.0.25/vue.min.js"></script>

    <script>
    new Vue({
        el: 'body',
        data: {
          name: '',
          email: ''
        },
        computed: {
          isValid: function () {
            return this.name != '' && this.email != ''
          }
        }
    })
    </script>
</body>
</html>
```

Vue-router

This article was first posted on Vue.js Brasil (http://www.vuejs-brasil.com)

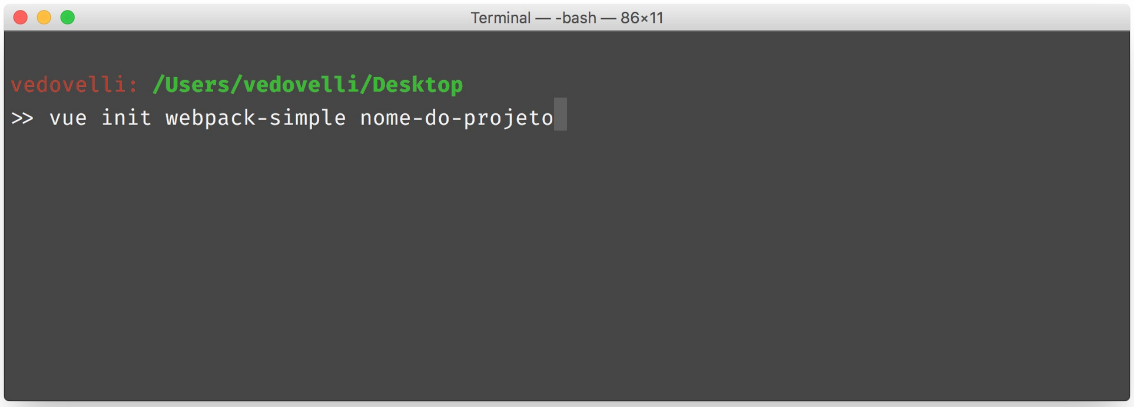
The most notable characteristic of a ***Single Page Application*** is the performance. All interface elements are in place and when your server infrastructure is well setup (the web service that provides the data) the feeling user gets while using the application is the application is locally installed, because of the short response time.

An essential component in a SPA is the **Router** which is responsible for showing/hidding one or more interface elements depending on the URL. This is its single responsibility but it doesn't mean it is a simple tool!

To support the article a new project is going to be created with **Vue-cli**, a command line tool which allows you to bootstrap a new Vue project. See more on https://github.com/vuejs/vue-cli.

Starting the project

Assuming you already have all necessary tools installed (node.js, npm and vue-cli) just `cd` to one of your system's folder and execute `vue init webpack-simple project-name` .



```
vedovelli: /Users/vedovelli/Desktop
>> vue init webpack-simple nome-do-projeto
```

After answeting to the 4 questions about the projets, just `cd` to the project's folder and execute `npm install` . After the general installation of all project's default dependencies, it is time to install the Vue-router (it is NOT listed as one of the project's default dependencies): `npm install --save-dev vue-router` .

> All files are now stored on **/node_modules** directory and will be available for your own script for import

Open up the project in your favorite code editor and locate the file `/src/main.js` to see the basic structure of a Vue object:

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

new Vue({
  el: 'body',
  components: { App }
})
```

Pay attention to **App** component because it is gonna be the parent of all other components that might be created during development.

Vue-router configuration

Before we begin we need at least 2 components so we can mimic some navigation. So, within the `/src` folder just create **ComponentA.vue** and **ComponentB.vue**. As their contents just add something that visually differentiate them:

```
<template>
    <h1>Componente A</h1>
</template>
```

```
<template>
    <h1>Componente B</h1>
</template>
```

Now going back to `/src/main.js` we start to configure the Vue-router:

```
import Vue from 'vue'
import VueRouter from 'vue-router' // << here
import App from './App.vue'

Vue.use(VueRouter) // << and here

new Vue({
  el: 'body',
  components: { App }
})
```

At this point we just import Vue-router straight from **node_modules** folder and make Vue aware of it.

Now we import both components...

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import App from './App.vue'
import ComponentA from './ComponentA.vue'
import ComponentB from './ComponentB.vue'

Vue.use(VueRouter)

new Vue({
  el: 'body',
  components: { App }
})
```

... and then we map them both to their routes:

```
...
Vue.use(VueRouter)

const router = new VueRouter()

router.map({
    '/component-a': {
        component: ComponentA
    },
    '/component-b': {
        component: ComponentB
    },
})

new Vue({
...
```

In the object passed as the only param to the method **map()** we assign the desired URLs to the components to be displayed. Now we need to adapt our App.vue component so it will display the right component and the correspondent URL is accessed:

Remove everything contained in App.vue replacing with the HTML below:

```
<template>
    <div>
        <router-view></router-view>
    </div>
</template>
```

The special tag `<router-view` introduced by the Vue-router package is the placeholder for the mapped components. You're free to add other tags close to and even encapsulate it within a regular HTML tag.

The last step is to replace the Vue object creation for the `router.start()` method. This part sometimes causes confusion because it is not clear where the Vue object is being created.

```
...

router.map({
    '/componente-a': {
        component: ComponentA
    },
    '/componente-b': {
        component: ComponentB
    },
})

router.start(App, '#container')
...
```

Please note that the `start()` method links the application's main component (the one with the tag `<router-view` >) to the DOM element that needs to be observed by the Vue object. So where is this element? Nowhere! We are going to add it to **/index.html**:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>route</title>
  </head>
  <body>
    <div id="container">
        <router-view></router-view>
    </div>
    <script src="dist/build.js"></script>
  </body>
</html>
```

Note the div with the id of "container". It also contain a `<router-view>` and this is the catch to use a single file component (*.vue) as in the instantiator of the Vue-router. In the official documentation you'll see you have to create a generic Vue component, but this is not at all necessary and you can keep using the structure you're used to.

Below you find the complete main.js:

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import App from './App.vue'
import ComponenteA from './ComponenteA.vue'
import ComponenteB from './ComponenteB.vue'

Vue.use(VueRouter)

const router = new VueRouter()

router.map({
    '/component-a': {
        component: ComponentA
    },
    '/component-b': {
        component: ComponentB
    },
})

router.start(App, '#container')
```

To finish this tutorial return to console and execute `npm run dev` and in your favorite browser point it to `http://localhost:8080/#!/component-a` or `http://localhost:8080/#!/component-b` .

Because in a real world application the quantity of router tend to be big, it is advised to add this router configuration on it own folder/files, making use of the module bundler to add them together in a way that makes sense to the Vue object to consume it.

Here's the link to the official documentation for the Vue-router: http://router.vuejs.org/en/index.html.

Advanced

1. Vuex

Vuex

Understanding the problem it solves

When you're building a browser based application it is not unusual to lose control over the flow of the data it manages.

In those kinds of application data is not only what the user sends to the server but also what controls the display of interface controls. So for instance you change a value in a component and another one that is supposed to be visible, for any reason it is now hidden. This is what we call **side effects**.

As your application grows it becomes a nightmare to deal with all the side effects that arise. The usual way to share data across you tree of components is by using events, up and down the tree. It is OK when you immediately capture a dispatched event but when the event is captured higher or lower in the tree, you'll quickly get yourself wondering *"where that event comes from..."*.

Enter the Single Source of Truth

More on Wikipedia

Single source of truth is to structure your data in a way that it is not duplicated anywhere within your application. Think about it as a dorsal spine of your application. All your components will get data from it and save back to it, making it easy to know where the change came from.

There are several advantages of taking this approach:

1. You have a centralized place to add/change its data;
2. It is available for all your components;
3. No component changes its data directly thus assuring data consistency;
4. Related tools make debugging a smoother experience.

Vuex

Official documentation here

It is the Vue implementation of Flux which in turn is the Facebook's implementation for Single Source of Truth. The integration with Vue's reactivity system is transparent and requires some simple configuration steps before it is ready to use.

The first step is to install it using a regular npm command `npm install vuex --save-dev`. Afterwards go to the file in which you've created the Vue instance, instruct Vue to use Vuex and attach a store to it:

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

Vue.use(Vuex)

new Vue({
  el: 'body',
  store: new Vuex.Store({})
})
```

Inside the Vuex.Store() method you'll have to pass an object two properties as follows:

1. The **State** which is a regular Javascript object containing the data you want to share across your application;
2. The methods that change the state. They're called **Mutation Methods**.

As an example take the snippet of code above:

```
...
new Vue({
  el: 'body',
  store: new Vuex.Store({
    state: {
      user: {
        name: '',
        email: ''
      }
    },
    mutations: {
      SET_USER (store, obj) {
        store.user = obj.user
      }
    }
  })
})
```

As you might have imagined the quantity of both state properties and mutation methods will grow larger than this, so we rely on module bundlers to setup Vuex somewhere else in our file system and just import it here:

create the file: /src/vuex/store.js

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

Vue.use(Vuex)

export default new Vuex.Store({
    state: {
      user: {
        name: '',
        email: ''
      }
    },
    mutations: {
      SET_USER (store, obj) {
        store.user = obj.user
      }
    }
  })
```

and in your **/src/main.js**:

```
import Vue from 'vue'
import store from './vuex/store'

Vue.use(Vuex)

new Vue({
  el: 'body',
  store
})
```

Reading and Setting data in a store

Now that we have our store fully configured and attached to the Vue object, it is time to start using it!

The beauty is that it is immediately available to all your components. The same way you're used to properties such as **methods: {}**, **data: {}** and **computed: {}** you now have the **vuex: {}** property. Let's get started:

```
<script>
  export default {
    props: ['some-property-here'],

    vuex: {
      getters: {},
      actions: {}
    },

    data () { // << some local state
      return {
        whatever: ''
      }
    }
  }
</script>
```

If we want to access the user property of our Vuex Store, it is just a matter of setting up a getter...

```
...
vuex: {
  getters: {
    user: store => store.user
  },
  actions: {}
},
...
```

... and then use it as a normal internal property: `this.user` .

Now you need to change user data. Before moving forward a note to keep in mind: **you CANNOT change Vuex State directly**. If you ask me if it is possible I'd say "yes it is" but highly discouraged. This is because if you want to find our how a property is set, you go for a single place instead of opening up all you component files to discover where the change came from.

So how to change it?

By using **actions**. They are methods like the ones you create inside the **methods: {}** property of your components but you declare them in a special place: within **vuex: { actions: {} }** property. By doing this you assure the first parameter received by your method will be an instance of the Vuex Store.

We are interested in a special method contained in that Store object, a method called **dispatch()**. We will use it to invoke a mutation, this one responsible for setting up date in our Vuex Store.

```
...
vuex: {
  getters: {
    user: store => store.user
  },
  actions: {
    setUser ({dispatch}, obj) {
      dispatch('SET_USER', obj)
    }
  }
},

methods: {
  ordinaryButtonClickHandler () {
    let user = {
      user: {
        username: 'New username',
        email: 'email@email.com'
      }
    }
    this.setUser(user) // << the action gets called after a button click
  }
}
...
```

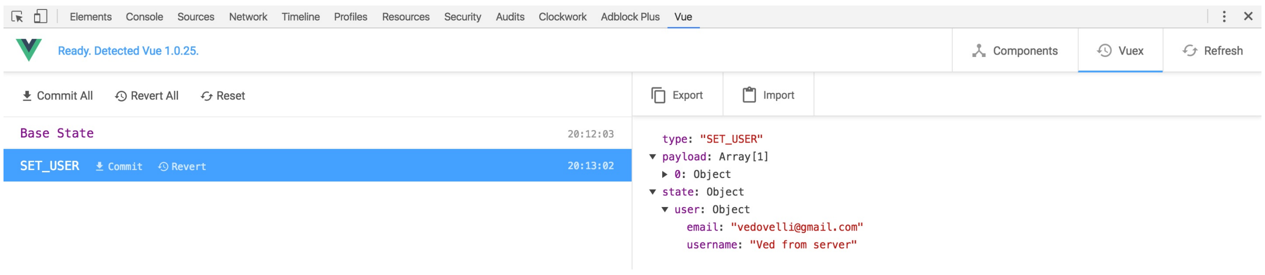
Pay attention to this specific part because this the most confusing part: `dispatch('SET_USER', obj)`. First we receive this method dispatch() by using Destructuring Assignment, a new and very useful feature of ES2015. Think about it as an event dispatcher with a single purpose: **invoke a mutation**. You don't manually capture it anywhere as you would do with a regular dispatched event. Thanks to Vue it goes straight to your **mutations object** in your Vuex configuration and the proper method is invoked.

The name of the mutation method is the first parameter and the object the mutation will receive is the second one. When it gets to the mutation, your store is finally changed and all your components observing the Vuex Store will automatically be updated.

Supporting Tools

There is a powerful tool to support development with Vue.js called Vue Devtools, a Google Chrome extension that lets you inspect your tree of components, interact with them in the Console and interact with your Vuex Store, taking advantage of Time Travel for the data.

It shows all the invoked mutations and you can navigate through it, seeing changes in real time.



```
  type: "SET_USER"
▼ payload: Array[1]
  ▶ 0: Object
▼ state: Object
  ▼ user: Object
      email: "vedovelli@gmail.com"
      username: "Ved from server"
```

Conclusion

The official documentation states that ***Vuex is not suitable for all kinds of project*** but I found it so simple to use that I include it in all of my projects. It feels natural.

Now that it is clear that Vuex is **a centralized store for your data** which makes it easier to handle **the state of your application**, keep in mind you can also have **a local state in all your components**. The decision to add it to Vuex Store or to keep it local is for you to make. Do you need this piece of data anywhere else in my application? If the answer is YES than you should add it to Vuex Store.