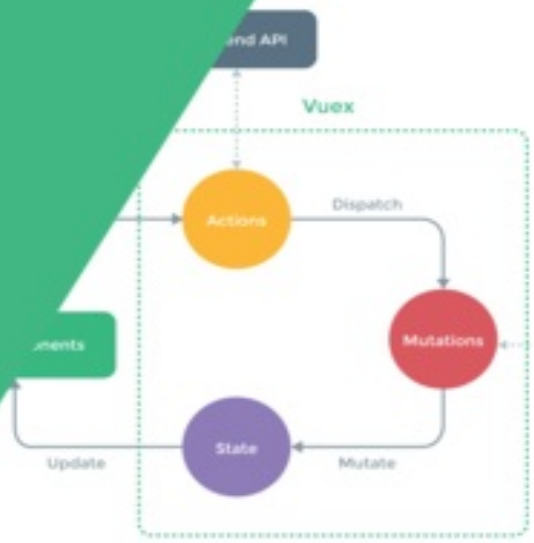
2016

# Vuex Concepts

## The Flux Application Architecture for Vue.js




Daniel Schmitz
Daniel Georgii

# Vuex Concepts

## The Flux Application Architecture for Vue.js

**Daniel Schmitz and Daniel Pedrinha Georgii**

This book is for sale at http://leanpub.com/vuex

This version was published on 2016-07-08


* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

# Table of Contents


**I** Introduction

**1.** What is Vue.js?

**2.** What is Flux?

**3.** Why use it

**4.** When to use it

**5.** Basic concepts

**II** A complete example

**5.1** Creating the project

**5.2** Creating components

**5.3** Including Vuex

**5.4** Creating a state variable

**5.5** Creating mutations

**5.6** Creating Actions

**5.7** Creating getters

**5.8** Changing the Display component

**5.9** Changing the Increment component

**5.10** Testing the application

**6.** Chrome vue-devtools

**7.** Passing data through Vuex

**8.** Dealing with Errors

**9.** Working with async methods

**10.** Notifying the user of the application state

**11.** Using Vuex to control the `waitMessage`

**III** Modular Vuex

# I Introduction

In this book we will talk about Vuex, the implementation of the Flux in Vue.

## 1. What is Vue.js?

Vue.js is a framework based on reactive components, used specially for web interfaces creation. Vue.js was created to be simple, reactive, based on components and compact.

## 2. What is Flux?

Flux is not a framework, architecture or language. It's a concept. When we use Flux in Vue (or any other language/framework) we create an unique way to update data that ensures every thing is right. This way to update data also ensures that the view components also keep the right state. This is really important to complex SPAs (Single Page Applications), that has plenty tables and screens, and of course events and data changes all the time.

## 3. Why use it

When creating complex SPAs we can't work with all the application data in global variables taken that in a certain moment they would be too many to control. And besides that, keeping the right state of each system point becomes a real trouble for the developer.

For example, with a shopping cart it's necessary to test all the different interface variations after including items to it. With Vuex it's possible to have a better control of those states. It's possible, for instance, to load a state with 10 items in the cart without adding them manually, saving time. It's also possible to work "unplugged" from the API.

Unit tests also benefit from Vuex since you can go to a specific state of the app by quickly changing its data.

## 4. When to use it

Use Vuex in production applications. To learn Vue basics it’s not necessary to use Vuex. But it’s recommended for any production application.

## 5. Basic concepts

To understand Vuex it’s necessary to know the following concepts:

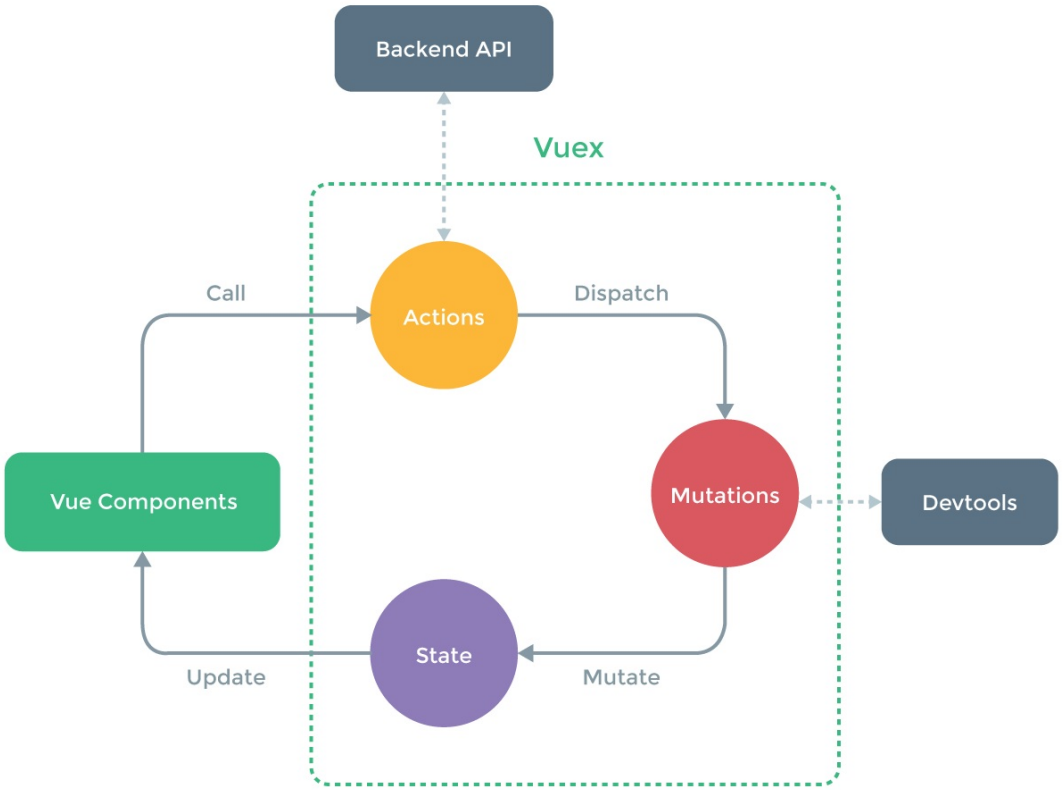
Store
: Store is the main Flux/Vuex concept. It’s where the application state is stored. When we say **state** we are talking about a group of dozens of variables to store data. As we know, Vue has a reactive visualization layer that observes variables and changes its content accordingly. The Store in Vuex is where the variables are stored, where the states are stored and where the View will observe for changes.

Action
: Defined as the action that can change a state. An Action is the **only** thing that can change a state’s information. This means that instead of changing the Store directly, you will call an Action to change it.

Mutation
: A “mutation” can be described as the “event that changes the store”. The Action will change the Store via “mutations”. Only the Mutations should directly access the Store variables.

Getters
: Getters are methods responsible for observing the variables in the Store and provide it to the application. It ensures that changes to the Store will reflect to the application.

As you can see, the application never access or changes a Store variable directly. There’s a flux to be followed in order to keep every thing in its place. The next image illustrates this flux:


Backend API
Vuex
Call
Actions
Dispatch
Vue Components
Mutations
Devtools
State
Update
Mutate

# II A complete example

In this first example let's create a simple counter, showing its value and a button to add or remove a unit.

## 5.1 Creating the project

Let's create a new project called `vuex-counter` with vue-cli:

```
1 vue init browserify-simple vuex-counter
```

Install vue-cli with `npm i -g vue-cli`

Open the directory and add Vuex:

```
1 cd vuex-counter
2 npm i -S vuex
```

And install all the remaining libraries:

```
1 npm i
```

Compile the code and start the web server:

```
1 npm run dev
```

Accessing http://localhost:8080/ you should get the "Hello Vue" message. The vuex-counter project is ready for Vuex.

## 5.2 Creating components

Let's create two components called `Increment` and `Display`.

Create the `Display.vue` component in the `src` directory with the following code:

```
1 <template>
2   <div>
```

```
    <h3>Count is 0</h3>
  </div>
</template>
<script>
export default {

}
</script>
```

And the `Increment.vue` in the same directory with two buttons:

```
<template>
  <div>
    <button>+1</button>
    <button>-1</button>
  </div>
</template>

<script>
export default {
}
</script>
```

Add them to the Application's main component (App.vue):

```
<template>
  <div id="app">
    <Display></Display>
    <Increment></Increment>
  </div>
</template>


<script>
import Display from './Display.vue'
import Increment from './Increment.vue'

export default {
    components: {
        Display, Increment
    },
  data () {
    return {
      msg: 'Hello Vue!'
    }
  }
}
</script>
```

Reload the page on your browser and you should see the new interface.

## 5.3 Including Vuex

With Vuex installed we can create the Store file for the application.

Create the `store.js` file inside the `src` directory with the following code:

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

Vue.use(Vuex)

const state = {

}

const mutations = {

}

export default new Vuex.Store({
  state,
  mutations
})
```

Notice that here we are also importing the Vuex, and telling Vue to use it. We also created two constants called `state` and `mutations` to store the app's variables and the events responsible to changing the `states`. And in the end, we create and export the Store with `export default new Vuex.Store`. This way we can import it into the main application like following:

```
<template>
  <div id="app">
    <Display></Display>
    <Increment></Increment>
  </div>
</template>

<script>
import Display from './Display.vue'
import Increment from './Increment.vue'
import store from './store.js'

export default {
    components: {
        Display, Increment
    },
  data () {
    return {
      msg: 'Hello Vue!'
    }
  },
  store
}
</script>
```

## 5.4 Creating a state variable

In the Store we have the `state` constant that contains the application state variables. For our counter we need a variable to store its value. Let's call this counter `count`:

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

Vue.use(Vuex)

const state = {
    count: 0
}

const mutations = {

}

export default new Vuex.Store({
  state,
  mutations
})
```

## 5.5 Creating mutations

With the variable ready we can define mutations to access it. Remember, only mutations can change a state. Check out the code to create the `INCREMENT` and `DECREMENT` mutations:

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

Vue.use(Vuex)

const state = {
    count: 0
}

const mutations = {
    INCREMENT(state){
        state.count++;
    },
    DECREMENT(state){
        state.count--;
    }
}

export default new Vuex.Store({
  state,
  mutations
})
```

The mutations, by convention, use capital letters. Their first parameter must be the `state` to be able to access the variables.

## 5.6 Creating Actions

The Actions are responsible for calling the mutations. By convention, the Actions are defined in a different file. Create the `actions.js` file inside the `src` directory with the following code:

```
export const incrementCounter = function ({ dispatch, state }) {
  dispatch('INCREMENT')
}

export const decrementCounter = function ({ dispatch, state }) {
  dispatch('DECREMENT')
}
```

At this moment, the *actions* only dispatch the mutation. It may look like an unnecessary step, but will be very useful when working with Ajax.

## 5.7 Creating getters

The Getters are responsible for returning the variables values of a state. The view components use the getters to observe the changes to the state

Create the `getters.js` in the `src` directory:

```
export function getCount (state) {
  return state.count
}
```

## 5.8 Changing the Display component

With the Flux structure ready (state, mutations, actions and getters) we can get back to the components. First let's change the `Display` component to use a getter to update its value:

```
<template>
  <div>
    <h3>Count is {{getCount}}</h3>
  </div>
</template>

<script>

import { getCount } from './getters'

export default {
    vuex: {
        getters: {
            getCount
        }
    }
}
</script>
```

Don’t forget to use braces in the import. This is necessary due to the new *destructuring array* from ECMAScript 6.

The `Display` component now imports the `getCount` method of the `getters.js` file, referenced in the `Vuex` object and finally used in the template with a bind. This setup will ensure that when the `state.count` value is changed by a mutation, the template will be updated.

## 5.9 Changing the Increment component

Back to the `Increment.vue` file, let’s add actions to the buttons:

```
<template>
  <div>
    <button @click="incrementCounter">+1</button>
    <button @click="decrementCounter">-1</button>
  </div>
</template>

<script>
import { incrementCounter, decrementCounter } from './actions'

export default {
    vuex: {
        actions: {
            incrementCounter,decrementCounter
        }
    }
}
</script>
```

Notice that we imported the `incrementCounter` and `decrementCounter` actions and referenced them in the `vuex` property. With that, Vuex will take care of every thing necessary for the bind. Then the actions are also referenced in the buttons.

When the user clicks on the `+1` button, the `incrementCounter` action will be called, calling the `INCREMENT` mutation that will change the `state.count` variable. The getter is notified, changing the `Display` component value.

## 5.10 Testing the application

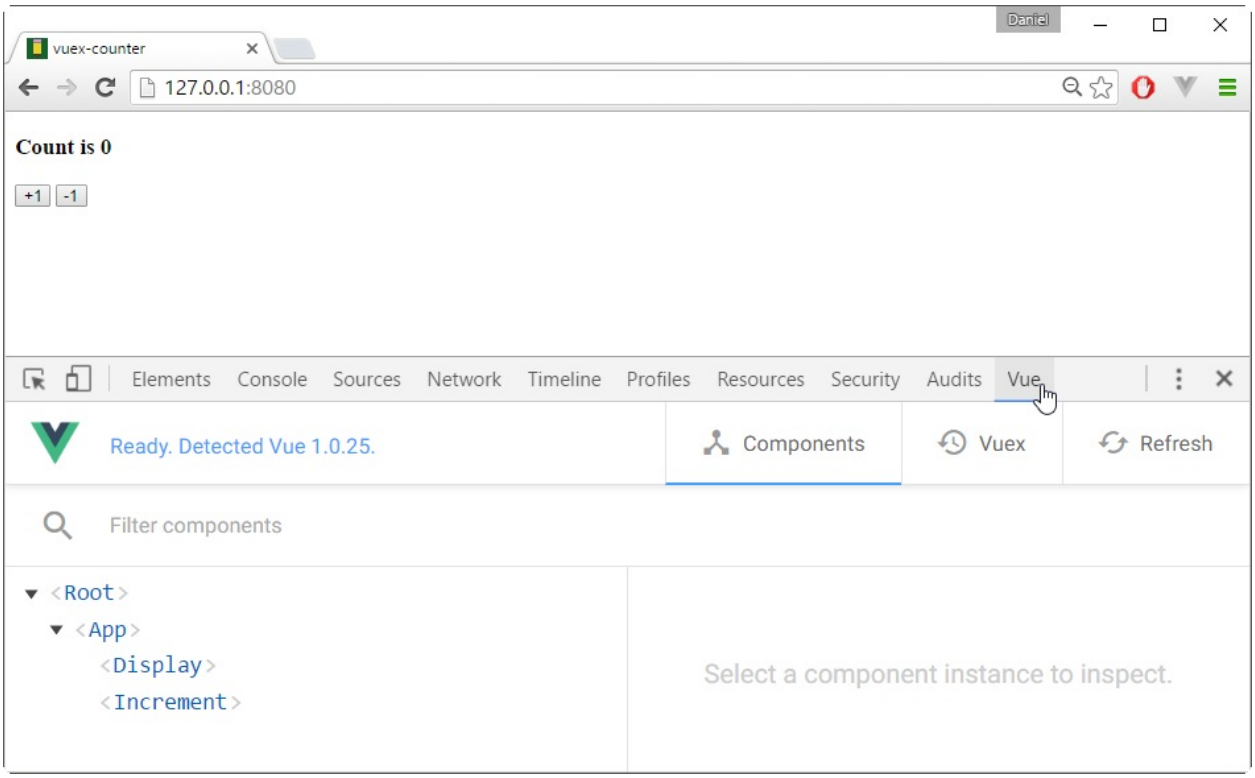
Make sure that the app is compiled and test it. If it doesn’t behave as expected, check the console (F12 in Chrome) to see if there’s any error.

## 6. Chrome vue-devtools

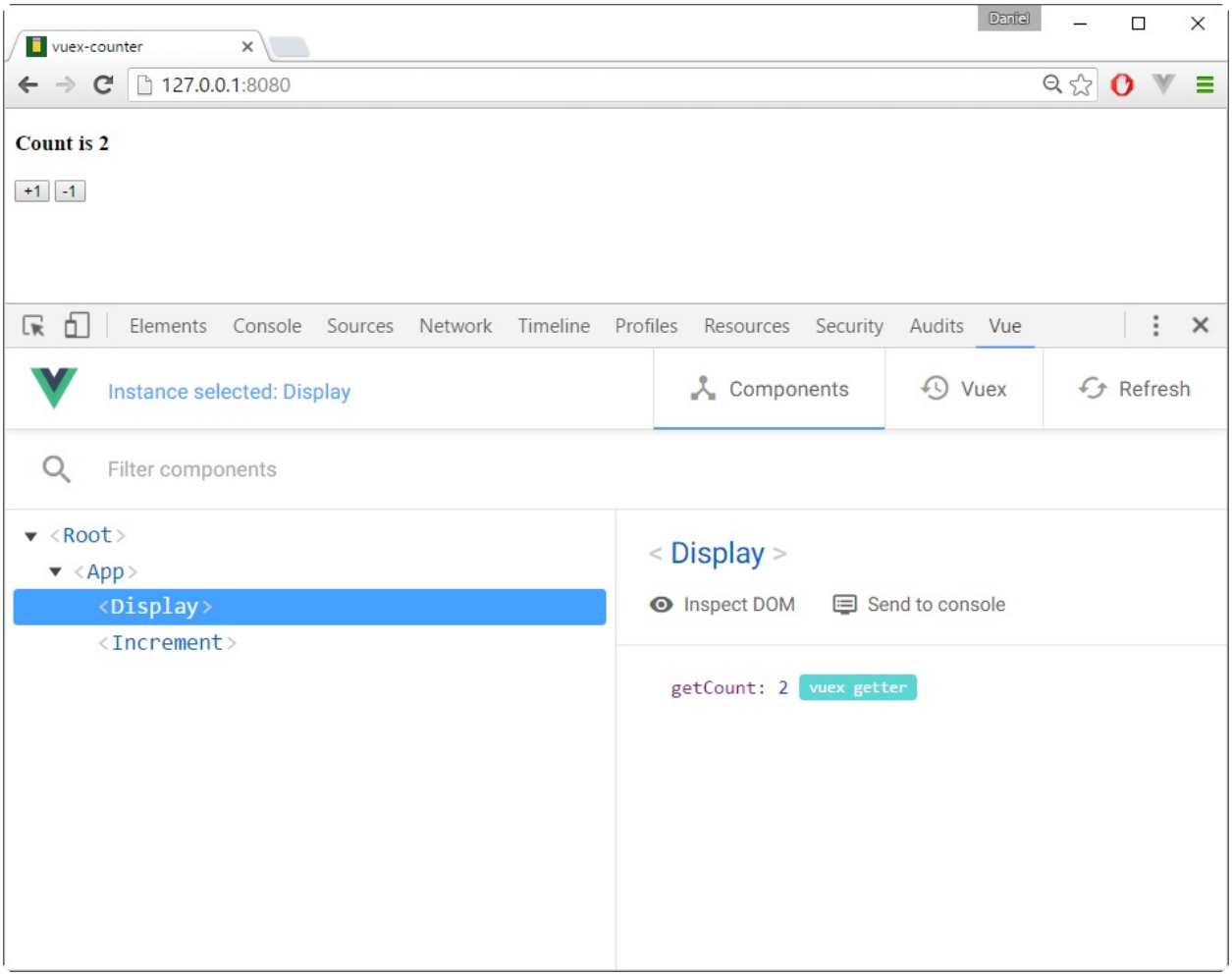
The Chrome vue-devtools is a Google Chrome plug-in to help debugging Vue Applications. You can get it in the Chrome Web Store.

After installing it, restart your browser and access the vuex-counter app.

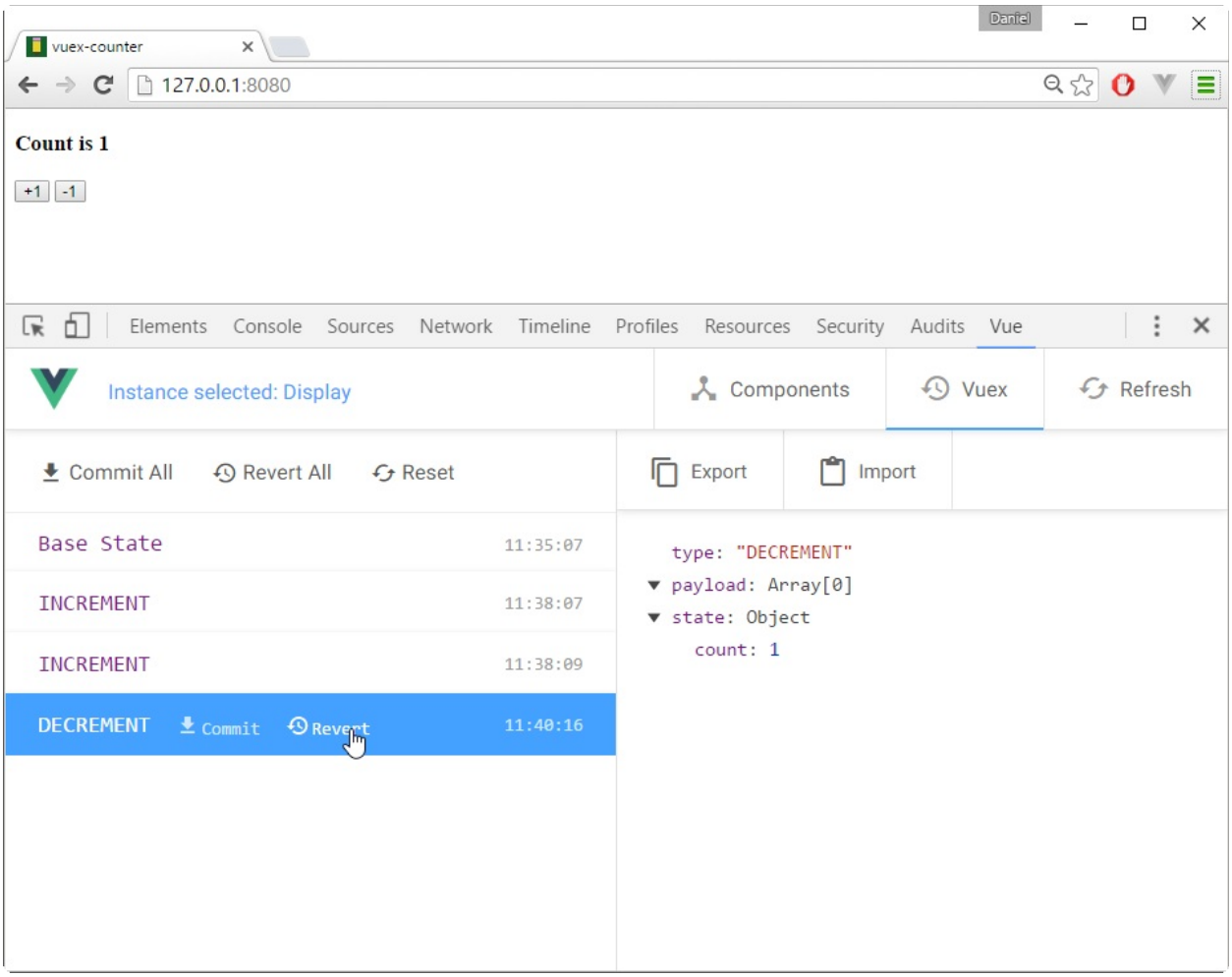
Press F12 to open the Chrome Dev Tools and check out the Vue tab:




In the Vue tab you can verify how the components are being inserted in the page, and for each component it's possible to evaluate each variable state:




In the Vuex tab it’s possible to see the mutations being executed and its changes to the state. It’s even possible to rollback a mutation:




## 7. Passing data through Vuex

Most of the times it's necessary to pass data from the view component to the Store. This can be done respecting the parameter flow between each step.

Back to the `vuex-counter` project let's add a text box to the Increment component and a button to increment the amount typed by the user.

```
<template>
  <div>
    <button @click="incrementCounter">+1</button>
    <button @click="decrementCounter">-1</button>
  </div>
  <div>
      <input type="text" v-model="incrementValue">
      <button @click="tryIncrementCounterWithValue">increment</button>
  </div>
</template>
...
```

Notice that the text box has the v-model pointing to the `incrementValue` variable and that the button click event is calling `tryIncrementCounterWithValue`. Here's the new script code of the component:

```
<script>
import { incrementCounter, decrementCounter, incrementCounterWithValue  } from './actior

export default {
    vuex: {
        actions: {
            incrementCounter,decrementCounter,incrementCounterWithValue
        }
    },
    data () {
        return{
            incrementValue:0
        }
    },
    methods: {
        tryIncrementCounterWithValue(){
            this.incrementCounterWithValue(this.incrementValue)
        }
    }
}
</script>
```

Notice that we imported a third Action called `incrementCounterWithValue`, that we will create soon. This Action is called by the `tryIncrementCounterWithValue` method to show you that it's also possible to call Actions from component methods. The `tryIncrementCounterWithValue` method passes the `this.incrementValue` value defined in the component data.

Create the new Action:

```
export const incrementCounter = function ({ dispatch, state }) {
  dispatch('INCREMENT')
}

export const decrementCounter = function ({ dispatch, state }) {
  dispatch('DECREMENT')
}

export const incrementCounterWithValue = function ({ dispatch, state },value) {
  dispatch('INCREMENTVALUE',parseInt(value))
}
```

Notice that the `incrementCounterWithValue` method has the `value` parameter that will be passed to the `INCREMENTVALUE` mutation after being converted to `int`.

Here's the new mutation:

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

```
3
4 Vue.use(Vuex)
5
6 const state = {
7     count: 0
8 }
9
10 const mutations = {
11     INCREMENT(state){
12         state.count++;
13     },
14     DECREMENT(state){
15         state.count--;
16     },
17     INCREMENTVALUE(state,value){
18         state.count=state.count+value
19     },
20 }
21
22 export default new Vuex.Store({
23   state,
24   mutations
25 })
```

Notice that the `INCREMENTVALUE` mutation has a second parameter called `value` to increment the `state.count` variable value.

## 8. Dealing with Errors

The Vuex Actions are responsible for dealing with possible errors that can invalidate the mutation call. One way to deal with that is using the `try...catch` block. In our case we could have:

```
1 ...
2 export const incrementCounterWithValue = function ({ dispatch, state }, value) {
3
4     let intValue = parseInt(value);
5     if (isNaN(intValue)) {
6         throw "Not an Integer"
7     } else {
8         dispatch('INCREMENTVALUE', intValue)
9     }
10 }
```

If the value can’t be converted to an integer an exception is thrown and must be catch in the component:

```
1 ...
2  tryIncrementCounterWithValue(){
3     try{
4         this.incrementCounterWithValue(this.incrementValue)
5     } catch (error) {
6         alert(error)
7     }
8 }
9 ...
```

Notice that only synchronous methods will work with `try...catch` blocks. Asynchronous methods like when reading files and Ajax calls among others will work only with callbacks.

## 9. Working with async methods

One of the most common async methods used is the Ajax call to the server, where the java-script code flux won't wait for the response. And this behavior must be dealt with in the Action.

For that we need to use callbacks, and we will use the `incrementCounter` Action as an example.

Let's add two seconds to the mutation call:

```
export const incrementCounter = function ({ dispatch, state }) {
    setTimeout(function(){
        dispatch('INCREMENT')
    },2000)
}
```

Now, when clicking on the `+1` button, the value is updated only after two seconds. In this mean time we need to let the user know that something is happening. In other words, we need to add a `Loading...` message to the component and remove it when the mutation is triggered.

## 10. Notifying the user of the application state

To notify the user that a method is being executed add another `div` to the `Increment` component:

```
<template>
    <div>
        <button @click="incrementCounter">+1</button>
        <button @click="decrementCounter">-1</button>
    </div>
    <div>
        <input type="text" v-model="incrementValue">
        <button @click="tryIncrementCounterWithValue">increment</button>
    </div>
    <div v-show="waitMessage">
      Wait...
      </div>
</template>
```

The "Wait…" message is shown if `waitMessage` is `true`. It's declared initially as `false`:

```
...
 data () {
        return{
            incrementValue:0,
            waitMessage:false
        }
    },
...
```

To show the `waitMessage` let's make the `+1` button call the `tryIncrementCounter` method instead of `incrementCounter`:

```
<template>
...
<button @click="tryIncrementCounter">+1</button>
...
</template>
<script>
...
methods: {
      tryIncrementCounter(){
        this.waitMessage=true;
        this.incrementCounter();
      },
        tryIncrementCounterWithValue(){
            ...
        }
    }
...
</script>
```

When the mutation changes the `state.count` value it's necessary to hide the "Wait…" message. This is done with a callback:

```
  export const incrementCounter = function ({ dispatch, state },callback) {
    setTimeout(function(){
        dispatch('INCREMENT')
        callback();
    },2000)
}
...
```

Now the `incrementCounter` action has another parameter called `callback`, that is called right after the operation ends. In this case, when `setTimeOut` expires, simulating an Ajax request.

In the template, it's necessary to use the callback called by the action to hide the "Wait…" message:

```
...
   tryIncrementCounter(){
        let t = this;
        this.waitMessage=true;
        this.incrementCounter(function(){
```

```
6           t.waitMessage=false;
7         });
8       },
9 ...
```

Notice that in the `tryIncrementCounter` method we created the `t` variable to reference `this`, since it will be lost due to the scope change.

Now, when the user clicks on the `+1` button, the “Wait…” message shows and when the counter is updated, the message disappears.

Looks like every thing is write, but not quite… This callback approach is commonly used in Java-script, but we are not using the right Flux to control the `waitMessage` variable.

## 11. Using Vuex to control the `waitMessage`

To properly control the “Wait…” message we need to follow the next steps:

- Create a variable in the State to store the visibility state of the message.
- Create mutations to show and hide the message.
- Create a getter to return the state of the variable.
- Change the component to observe its state.

First let’s change the Store to include the new state and mutations:

```
 1 import Vue from 'vue'
 2 import Vuex from 'vuex'
 3
 4 Vue.use(Vuex)
 5
 6 const state = {
 7     count: 0,
 8     showWaitMessage:false
 9 }
10
11 const mutations = {
12     INCREMENT(state){
13         state.count++;
14     },
15     DECREMENT(state){
16         state.count--;
17     },
18     INCREMENTVALUE(state,value){
19         state.count=state.count+value
20     },
21     SHOW_WAIT_MESSAGE(state){
22       state.showWaitMessage = true;
23     },
24     HIDE_WAIT_MESSAGE(state){
25       state.showWaitMessage = false;
```

```
    }
}

export default new Vuex.Store({
  state,
  mutations
})
```

Create the getter to return the `state.showWaitMessage` variable value:

```
export function getCount (state) {
  return state.count
}

export function getShowWaitMessage(state){
   return state.showWaitMessage;
}
```

Change the action to dispatch the mutations:

```
export const incrementCounter = function ({ dispatch, state }) {
   dispatch('SHOW_WAIT_MESSAGE')
    setTimeout(function(){
       dispatch('HIDE_WAIT_MESSAGE')
       dispatch('INCREMENT')
    },2000)
}
```

The `incrementCounter` dispatches the `SHOW_WAIT_MESSAGE` before simulating the Ajax call. And after the two seconds, we dispatch both `HIDE_WAIT_MESSAGE` and `INCREMENT` mutations.

And finally we change the `Increment` component:

```
<template>
    <div>
        <button @click="incrementCounter">+1</button>
        <button @click="decrementCounter">-1</button>
    </div>
    <div>
        <input type="text" v-model="incrementValue">
        <button @click="tryIncrementCounterWithValue">increment</button>
    </div>
     <div v-show="getShowWaitMessage">
      Aguarde...
      </div>
</template>

<script>
import { incrementCounter, decrementCounter, incrementCounterWithValue  } from './actior
import {getShowWaitMessage} from './getters'

export default {
    vuex: {
        actions: {
            incrementCounter,
```

```
            decrementCounter,
            incrementCounterWithValue
        },
        getters: {
          getShowWaitMessage
        }
    },
    data () {
        return{
            incrementValue:0
          }
    },
    methods: {
        tryIncrementCounterWithValue(){
            try{
                this.incrementCounterWithValue(this.incrementValue)
            } catch (error) {
                alert(error)
            }
        }
    }
}
</script>
```

Notice that the changes make the code cleaner, with less methods on the component. But it can still be improved, changing the `div` with the “Wait…” message to its own component. But we will leave it as an exercise to you.

# III Modular Vuex

It’s possible to work with modules where each concept is fiscally separated by files. Suppose an example of a system with user log-in, a group of screens to work with user information (users table), another group of screens to work with products (products table) and another group about suppliers (suppliers table).

You can use a single store, action and getters files or separate those concepts in modules.

Let’s create another project for this situation:

```
vue init browserify-simple vuex-modules
cd vuex-modules
npm install
```

And don’t forget to add the extra vue libraries:

```
npm i -S vuex vue-resource vue-route
```

With every thing installed let’s start with the application structure. And since we are dividing the app in modules, let’s create a *modules* directory and the three other modules in it:

Remember that this is not the only way to use Vuex, it’s just a suggestion.

Structure:

```
  src
  |- modules
        |- login
        |- user
        |- product
        |- supplier
  |- App.vue
  |- main.js
```

After creating the directories create each Store. But in this case, each Store file will have the `index.js` name:

```
src
  |- modules
```

```
        |- login
            |- index.js
        |- user
            |- index.js
        |- product
            |- index.js
        |- supplier
            |- index.js
  |- App.vue
  |- main.js
```

Now, each Store contains its own state and mutations:

src/modules/login/index.js

```
const state = {
    username : "",
    email : "",
    token : ""
}

const mutations = {

}

export default { state, mutations }
```

The login module has three properties. The mutations will be created as demanded. Here are the basic code for the three other Stores:

src/modules/user/index.js

```
const state = {
    list : [],
    selected : {}
}

const mutations = {

}

export default { state, mutations }
```

src/modules/product/index.js

```
const state = {
    list : [],
    selected : {}
}

const mutations = {

}

export default { state, mutations }
```

src/modules/supplier/index.js

```
const state = {
    list : [],
    selected : {}
```

```
}

const mutations = {

}

export default { state, mutations }
```

Now we need to create the main Store to expose the others:

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

import login from './modules/login'
import user from './modules/user'
import product from './modules/product'
import supplier from './modules/supplier'

Vue.use(Vuex)

export default new Vuex.Store({
 modules: {
     login,
     user,
     product,
     supplier
    }
})
```

With the Stores ready, let's get to the main component `App.vue`:

src/App.vue

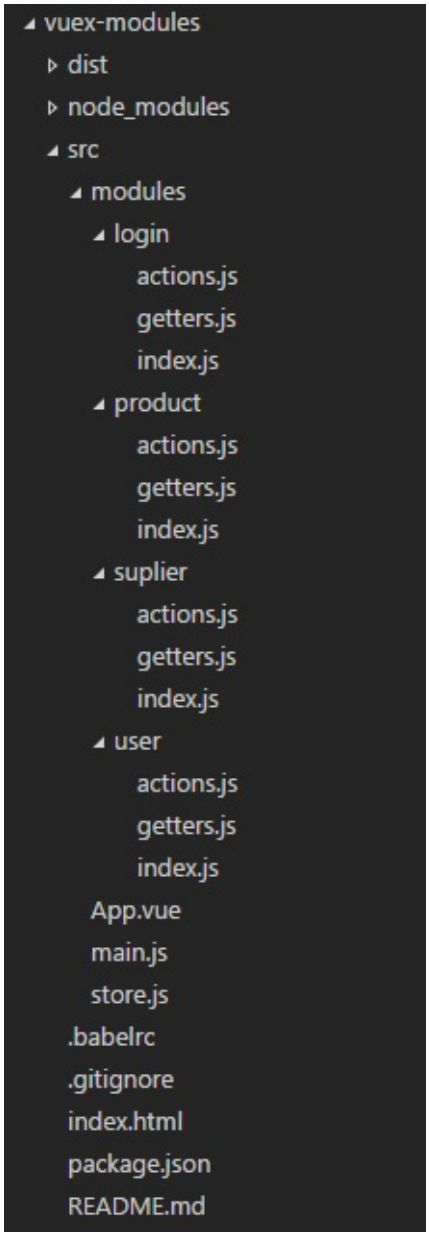
```
<template>
  <div id="app">
    <h1>{{ msg }}</h1>
  </div>
</template>

<script>
import store from './store'

export default {
  data () {
    return {
      msg: 'Hello Vue!'
    }
  },
  store
}
</script>

<style>
body {
  font-family: Helvetica, sans-serif;
}
</style>
```

Now we need to create the Actions and Getters for each module (at first empty) as on the next image:




With the basic structure ready, let’s start using Vuex. First let’s create a button to simulate log-in/log-out. The button will make an Ajax request to the “login.json” file with the following content:

login.json

```
{
   "username": "foo",
   "email": "foo@gmail.com",
   "token": "abigtext"
}
```

Notice that we are just testing the modules, so there's no need to implement the back-end services.

Add the "login" button to the `App.vue` file:

src/App.vue

```
 <template>
  <div id="app">
    <h1>{{ msg }}</h1>
    <button @click="doLogin">Login</button>
  </div>
</template>

<script>
import store from './store'

import {doLogin} from './modules/login/actions'

export default {
  data () {
    return {
      msg: 'Hello Vue!'
    }
  },
  vuex :{
      actions :{
          doLogin
      }
  },
  store
}
</script>
```

The button calls the `doLogin` action from the login module that needs the following code:

```
 export function doLogin({dispatch}){
    this.$http.get("/login.json").then(
        (response)=>{
           dispatch("SETLOGIN",JSON.parse(response.data))
        },
        (error)=>{
            console.error(error.statusText)
        }
    )
}
```

Since the action uses VueResource for the Ajax connection it's necessary to load this plug-in to the `main.js` file:

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

import VueResource from 'vue-resource'
```

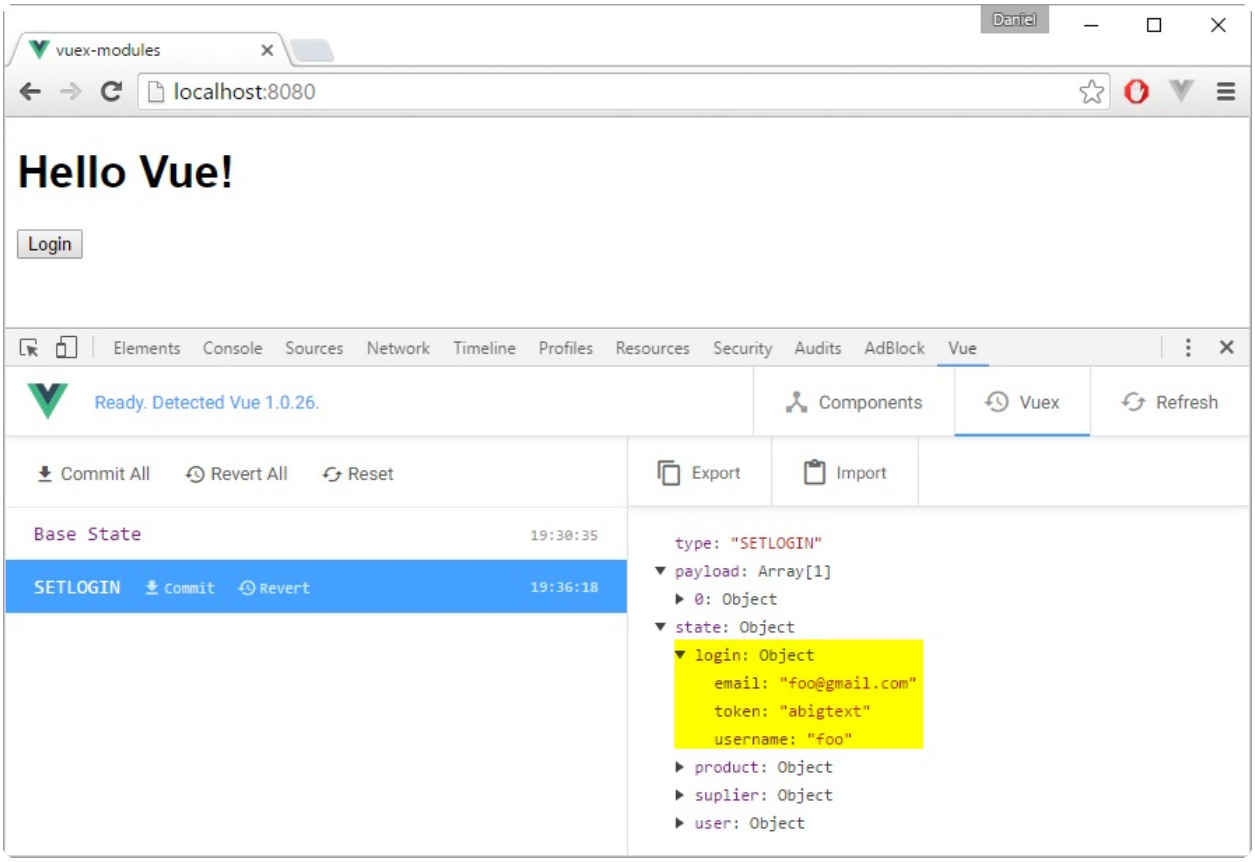
```
5
6 Vue.use(VueResource)
7
8 new Vue({
9   el: 'body',
10  components: { App }
11 })
```

Notice that when the Ajax request is successful the “SETLOGIN” mutation is dispatched, passing the `response.data` as the second parameter, which is the text from the `login.json` file. The “SETLOGIN” mutation is shown next:

src/modules/login/index.js

```
1 const state = {
2     username : "",
3     email : "",
4     token : ""
5 }
6
7 const mutations = {
8     SETLOGIN (state, data) {
9         console.log(data)
10        state.username = data.username;
11        state.email = data.email;
12        state.token = data.token;
13    }
14 }
15
16 export default { state, mutations }
```

Now it’s possible to test the application and verify that the login data was updated on the Vuex:




With the data being stored in the state we can create the “isLogged” getter to control if the user is logged or not. We will use this getter to control some other buttons.

src/modules/login/getters.js

```
1 export function isLogged(state){
2     return state.login.token != ""
3 }
```

To check if the user is logged or not it’s necessary to check if the authentication token is null. To log-out we will create another action called `doLogout`:

src/modules/login/actions.js

```
 1 export function doLogin({dispatch}){
 2     this.$http.get("/login.json").then(
 3         (response)=>{
 4            dispatch("SETLOGIN",JSON.parse(response.data))
 5         },
 6         (error)=>{
 7             console.error(error.statusText)
 8         }
 9     )
10 }
11 export function doLogout({dispatch}){
12       let login = {
```

```
            username : "",
            email : "",
            token : ""
        }
      dispatch("SETLOGIN",login)
}
```

By logging-out we call `SETLOGIN` with empty values.

With the log-in and log-out and the `isLogged` getter ready we can use it:

src/App.vue

```
<template>
  <div id="app">
    <button v-show="!isLogged" @click="doLogin">Login</button>
    <div v-show="isLogged">
        <button  @click="doLogout"> Logout</button>
        <button  > Users</button>
        <button  > Products</button>
        <button  > Suppliers</button>
    </div>
  </div>
</template>

<script>
import store from './store'
import {doLogin,doLogout} from './modules/login/actions'
import {isLogged} from './modules/login/getters'

export default {
  data () {
    return {
      msg: 'Hello Vue!'
    }
  },
  vuex :{
      actions :{
          doLogin,doLogout
      },
      getters : {
          isLogged
      }
  },
  store
}
</script>
```

The result is that now the Logout, Users, Products and Suppliers buttons appear only if the user is logged-in.

Now let’s use the “Products” button to show data that can be requested from the server. And for that we need to create:

- the loadProducts action
- the SETPRODUCTS mutation

- the SETPRODUCT mutation
- the getProducts getter
- the getProduct getter

The `loadProcuts` action will read the following json file:

products.json

```
[
    {
        "id": 1,
        "name": "product1",
        "quantity": 200
    },
    {
        "id": 2,
        "name": "product2",
        "quantity": 100
    },
    {
        "id": 3,
        "name": "product3",
        "quantity": 50
    },
    {
        "id": 4,
        "name": "product4",
        "quantity": 10
    },
    {
        "id": 5,
        "name": "product5",
        "quantity": 20
    },
    {
        "id": 6,
        "name": "product6",
        "quantity": 300
    }
]
```

src/modules/product/actions.js

```
export function loadProducts({dispatch}){
    this.$http.get("/products.json").then(
        (response)=>{
           dispatch("SETPRODUCTS",JSON.parse(response.data))
        },
        (error)=>{
            console.error(error.statusText)
        }
    )
}
```

After reading the file the “SETPRODUCTS” mutation is called passing the `response.data`. Here’s the mutation:

src/modules/product/index.js

```
const state = {
    list : [],
    selected : {}
}

const mutations = {
    SETPRODUCTS(state, data) {
        state.list = data;
    },
}

export default { state, mutations }
```

The “SETPRODUCTS” mutation will set the `list` variable that can be read via the getter:

src/modules/product/getters.js

```
export function getProducts(state){
    return state.product.list;
}

export function hasProducts(state){
    return state.product.list.length>0
}
```

For this getter we created two methods. The first returns the products list. The second will return if the list is empty. We will use it in the `App.vue` file:

src/App.vue

```
<template>
  <div id="app">
    <button v-show="!isLogged" @click="doLogin">Login</button>
    <div v-show="isLogged">
        <button  @click="doLogout"> Logout</button>
        <button  > Users</button>
        <button  @click="loadProducts">Products</button>
        <button  > Suppliers</button>

        <br>

        <div v-show="hasProducts">
            <h4>Products</h4>
            <table border="1">
                <thead>
                    <tr>
                        <td>id</td>
                        <td>name</td>
                        <td>quantity</td>
                        </tr>
                </thead>
                <tbody>
                    <tr v-for="p in getProducts">
                        <td>{{p.id}}</td>
                        <td>{{p.name}}</td>
                        <td>{{p.quantity}}</td>
                    </tr>
                </tbody>
            </table>
```

```
        </div>

    </div>
  </div>
</template>

<script>
import store from './store'
import {doLogin,doLogout} from './modules/login/actions'
import {loadProducts} from './modules/product/actions'
import {isLogged} from './modules/login/getters'
import {hasProducts,getProducts} from './modules/product/getters'

export default {
  data () {
    return {
      msg: 'Hello Vue!'
    }
  },
  vuex :{
      actions :{
          doLogin,doLogout,
          loadProducts
      },
      getters : {
          isLogged,
          hasProducts,getProducts
      }
  },
  store
}
</script>

<style>
body {
  font-family: Helvetica, sans-serif;
}
</style>
```

Notice that the button to load the products calls the `loadProcuts` action to make the Ajax call and the `SETPRODUCTS` mutation that sets the state list. The getters are updated changing `getProducts` and `hasProducts` variables. With the reactive design the div with `v-show="hasProducts"` will show as well as the table with the `v-for`. It will iterate the `getProducts` getter to create the DOM and show the products.

With that we finished a full cycle of the modular Vuex. And again, the directory structure is just a suggestion, you can change it as you will. The use of modular Vuex is also a project design suggestion, you don’t need to have multiple files for your states, actions and getters.

Here are some exercises for you:

- Create a button to remove all the products from the state.

- Load all the users.
- Load all the suppliers.