

## O que é o Vuex?

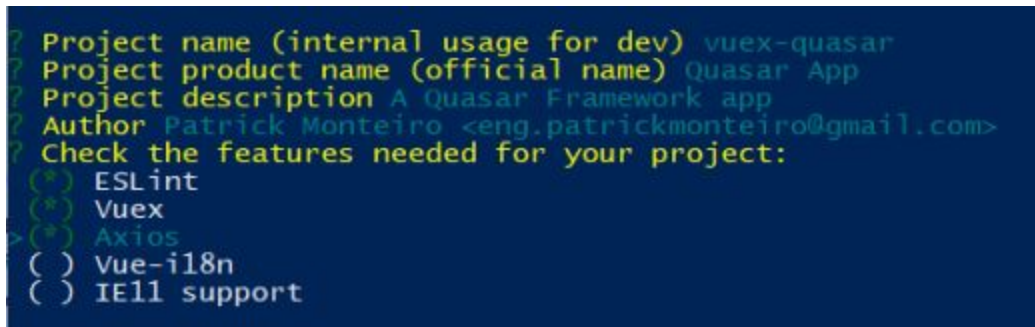
Segundo a sua documentação, o **Vuex** é um padrão de gerenciamento de estado. Na prática isto significa ter acesso/visualização de uma "variável" em qualquer componente/parte do software de maneira reativa.

## Como instalar o Vuex?

a) No caso do **quasar**, a instalação ocorre junto a instalação do quasar via terminal com o seguinte comando:

```
quasar init vuex-quasar
```

No instalador do quasar deve ser selecionada a opção Vuex.



```
? Project name (internal usage for dev) vuex-quasar
? Project product name (official name) Quasar App
? Project description A Quasar Framework app
? Author Patrick Monteiro <eng.patrickmonteiro@gmail.com>
? Check the features needed for your project:
  (*) ESLint
  (*) Vuex
  (*) Axios
  ( ) Vue-i18n
  ( ) IE11 support
```

b) No caso do Vue, o instalador

```
npm install vuex --save
```

## Estrutura do Vuex

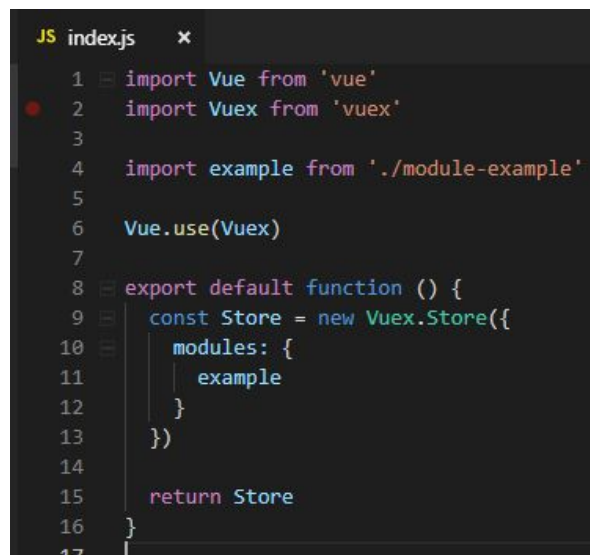
O Vuex é composto por 4 elementos sendo eles as **Actions**, **Getters**, **Mutations** e **State**. Isso pode ser feito em apenas um arquivo que é o **Store.js**, ou em caso de modularização do Vuex, são feitos em arquivos separados, cada um no seu próprio arquivo .js.

No caso do **Vue**, para ter acesso ao Vuex no sistema é necessário fazer a importação do mesmo no Store.js, arquivo único com a instância Vuex e os elementos do mesmo(State, Mutations, Getters e Actions).



```
1 import Vue from "vue"
2 import Vuex from "vuex"
3
4 Vue.use(Vuex);
5
6 export default new Vuex.Store({
7   state: {
8     user: "Lobo"
9   }
10 });
```

No caso do **quasar**, para ter acesso ao Vuex no sistema é necessário fazer a importação do mesmo no index.js. No exemplo abaixo, o Vuex está modularizado, devido a essa divisão é necessário definir dentro do Store o nome do módulo que está sendo importado.



```
JS index.js x
1 import Vue from 'vue'
2 import Vuex from 'vuex'
3
4 import example from './module-example'
5
6 Vue.use(Vuex)
7
8 export default function () {
9   const Store = new Vuex.Store({
10     modules: {
11       example
12     }
13   })
14
15   return Store
16 }
17
```

## State

Essa estrutura é responsável por definir as variáveis que serão usadas por qualquer parte do software ao importar o módulo em questão, similar ao **data()** que estamos acostumados a usar.

### ***a) Como acessar uma variavel que está dentro do State no meu componente?***

A maneira “raiz” de acessar a variável definida dentro do State é usando a instância Vuex e o nome do objeto.

**Exemplo:** `<div>{{ $store.state.variavel }}</div>`

Para evitar escrever todo esse nome da instância + state + variável, existe o auxiliar **mapState**. Por meio do **mapState** é possível acessar a variável apenas pelo nome em que foi definido no próprio State, o mesmo é definido no computed e deve ser importado na página/componente que vai ser usado.

## Exemplo:

```
<template>
  <p>{{id}}: {{user}}</p>
</template>

<script>
import { mapState } from "vuex";

export default {
  name: "Aulas",
  computed: mapState(["user", "id"]),
};
```

Caso exista mais de um método no Computed é necessário usar o **Spread Operator** -> ...  
(funciona como um separador de valores de um array, porém nesse caso, separador de métodos).

```
computed: {
  ...mapState(["user", "aulasCompletas"]),
  nomeMaiusculo() {
    return this.nome.toUpperCase();
  }
},
```

## Mutations

A única maneira de mudar o estado(variável) é utilizando o commit de uma mutação, essa estrutura faz a função do emit/eventbus. As mutations são definidas no Store como funções, podendo essa receber ou não parâmetros.

```
mutations: {
  changeUser(state, payload) {
    state.user = payload
  },
  completarAula(state) {
    state.aulasCompletas++
  }
}
```

### a) Como fazer o commit de uma mutation?

O commit é feito a partir de uma método na sua pagina/componente que commita a alteração para passando o nome da mutation e os parâmetros caso haja necessidade. Nesse exemplo abaixo, handleClick é um método normal da página, e quando ele é acionado ele commita a mutation de nome “changeUser” e depois da vírgula os parâmetros são passados logo em seguida como um objeto

```
methods: {  
  handleClick() {  
    this.$store.commit("changeUser", {  
      user: this.novoUser,  
      totalAulas: this.totalAulas  
    });  
  },  
}
```

Assim como no State, existe um facilitador para não precisar ficar escrevendo o commit, nesse caso é o **mapMutations**. O mesmo é definido em **methods** e é definido com o nome da função que foi definida no Store.js. Para usar a função que foi descrita na **mapMutations** basta usar como se fosse uma função normal definida dentro de methods.

```
methods: {  
  ...mapMutations(["changeUser", "completarAula"]),  
  handleClick() {  
    this.$store.commit("changeUser", {  
      user: this.novoUser,  
      totalAulas: this.totalAulas  
    });  
  },  
}
```

Obs: O **mapMutations** deve ser importado no script.

Sobre nomenclatura e padrão, o mais comum a ser usado para definir as funções é o **ALL\_CAPS**, ou seja, nome da função toda maiúscula e caso seja composto a separação de palavras de da pelo underline \_

```
mutations: {  
  CHANGE_USER(state, payload) {  
    state.name = payload.name  
  }  
}  
)
```

## Actions

As ações são semelhantes às mutações, as diferenças são as seguintes:

- Em vez de mudar o estado, as ações confirmam(ou fazem *commit*) de uma ou mais mutações.
- As ações podem conter **operações assíncronas** arbitrárias.

As actions são definidas no Store.js assim como o State e as Mutations, e deve ter o **context** (esse parâmetro contém todos os elementos da página Store.js) e os outros parâmetros caso sejam necessários.

```
actions: {  
  changeUser(context, payload) {  
    context.commit("CHANGE_USER", payload);  
    context.commit("CHANGE_ID", payload);  
  }  
}
```

Na página vue ou no seu componente, ele deve ser chamado por meio do método **dispatch**, assim como no exemplo abaixo

```
methods: {  
  ...mapMutations(["COMPLETAR_AULA"]),  
  handleClick(aula) {  
    this.$store.dispatch("completarAula", aula);  
  }  
}
```

Existe o helper para as Actions também, que é definido dentro de **methods**

```
methods: {  
  ...mapActions(["changeUser"]),  
  handleClick() {  
    this.changeUser({  
      name: "Ave",  
      id: "3321"  
    });  
  }  
}
```

A grande vantagem de se usar **Actions** é poder fazer chamadas **assíncronas**, funcionalidade que as Mutations não tem.

```
actions: {
  fetchAcao(context) {
    fetch("https://api.iextrading.com/1.0/stock/aapl/quote")
      .then(r => r.json())
      .then(r => {
        context.commit("UPDATE_ACAO", r)
      })
  }
}
```

## Getters

O Vuex nos permite definir **getters** no *store*. Você pode pensar neles como dados tratados para os **stores**. Como os dados computados, o resultado de um **getter** é armazenado em cache com base em suas dependências e só será reavaliado quando algumas de suas dependências forem alteradas.

```
getters: {
  livrosLidos(state) {
    return state.livros.filter(livro => livro.lido)
  }
},
```

Nesse exemplo acima, o **getter** retorna os livros que tem a propriedade “lido” como true.

A chamada do getter é similar a chamada do state:

```
<ul>
  <li v-for="livro in $store.getters.livros" :key="livro.nome">
    <p>{{livro.nome}}</p>
  </li>
</ul>
```

E assim como nós outros elementos, existe um helper chamado **mapGetters** que deve ser declarado dentro do **computed**.

```
computed: {
  ...mapGetters(["livrosLidos"])
}
```

Uma grande vantagem de se usar getters é poder passar parâmetros para a função e obter dados filtrados de acordo com o parâmetro.

Nesse exemplo abaixo, o getter livrosLidos retorna uma função que recebe um parâmetro(lido), esse no caso pode ser passado como true ou false, caso seja passado como true, ele irá retornar os livros que tem a propriedade lido como true, e vice-versa.

```
getters: {  
  livrosLidos(state) {  
    return function(lido) {  
      return state.livros.filter(livro => livro.lido === lido)  
    }  
  }  
},
```

Na sua página vue/componente o getter é chamado da seguinte forma:

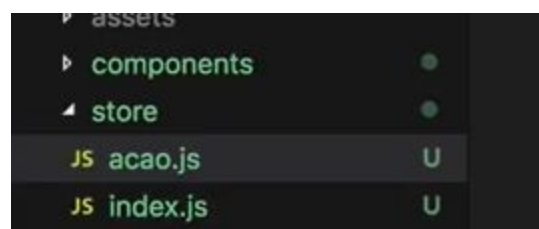
```
<ul>  
  <li v-for="livro in livrosLidos(false)" :key="livro.nome">  
    <p>{{livro.nome}}</p>  
  </li>  
</ul>
```

## Mutations

Devido ao uso de uma única árvore(único arquivo) de estado, todo o estado de nossa aplicação está contido dentro de um grande objeto. No entanto, à medida que nossa aplicação cresce em escala, o *store* pode ficar realmente inchado.

Para ajudar com isso, o Vuex nos permite dividir nosso **store** em módulos. Cada módulo pode conter seu próprio **state**, **mutations**, **actions**, **getters** e até módulos aninhados

Os novos módulos devem ser registrados dentro da pasta Store assim como o index.js(arquivo principal). Abaixo acao é o novo módulo do vuex.



O arquivo principal **index.js** deve importar os módulos do seu projeto e deve declarar dentro do **export** na propriedade **modules**.

```
import Vue from 'vue'
import Vuex from 'vuex'
import acao from '@/store/acao.js'

Vue.use(Vuex)

export default new Vuex.Store({
  modules: {
    acao
  },
})
```

Para evitar o problema de ter **mutations**, **actions** e **getters** com o mesmo nome de outros módulos, pois no Vuex essas propriedades ficam “juntas” com as do arquivo raiz(index.js), foi criado o **namespaced**. O mesmo é incluído com o intuito de exigir que o usuário referencie pelo “caminho” a função que ele está chamando. Para incluir o **namespaced** no seu código basta incluir ele dentro de **export default** e setá-lo como **true**.

```
export default {
  namespaced: true,
  state: {
    carrinho: []
  },
  mutations: {
    ADD_CART(state, payload) {
      state.carrinho.push(payload)
    }
  }
}
```

Exemplo de uso utilizando uma **Mutation**(puxarAcao) de um **Modulo**(acao).

```
created() {
  this.$store.dispatch("acao/puxarAcao");
}
```