

EXERCISES QUE TRABAJAREMOS EN LA CUE:

- EXERCISE 1: ACTIONS.
- EXERCISE 2: MAPACTIONS.
- EXERCISE 3: LLAMADAS ASÍNCRONAS EN ACTIONS.

EXERCISE 1: ACTIONS

INSTRUCCIONES

Para realizar el siguiente ejercicio, copia el proyecto hecho en el CUE anterior, y crea una carpeta vacía, con el nombre que prefieras, en este caso la llamaremos "cue13".

```
~/Documents mdkir cue13
```

Luego, nos dirigiremos a donde está el proyecto creado anteriormente. En este caso, tenemos una carpeta llamada cue12, y dentro de ella, el proyecto con el mismo nombre.

```
~/Documents/cue12 ls
CUE12_ejercicio1_getters.docx
CUE12_ejercicio3_mutations.docx
Cue12_TextClassReview.docx
Cue12_ejercicio2_mutation_eliminar_producto.docx
Cue12rebound_exercise.docx
cue12
```

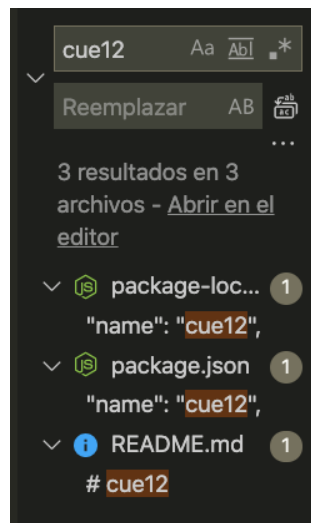
Ahora, teniendo la carpeta cue12 visualizada, vamos a copiarla a la nueva carpeta "cue13".

```
~/Documents/cue12 cp -a cue12/. ../cue13/cue13
```

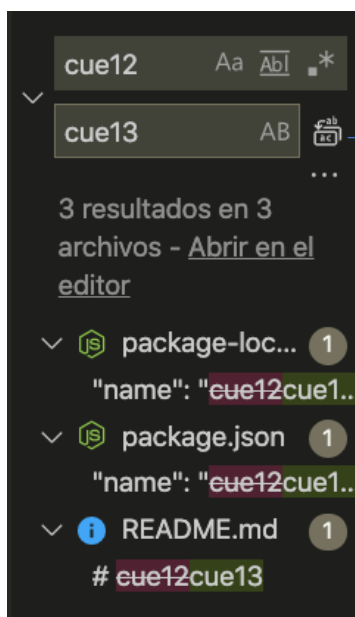
De esta forma, se copiará todo el contenido de la carpeta cue12, en la nueva carpeta "cue13".

Con esto listo, solo necesitamos abrir el proyecto con Visual Studio Code.

Para cambiar el nombre del proyecto, debemos hacer una búsqueda donde aparezca cue12.



Y vamos a reemplazarlo por “cue13”.



Presionamos el botón
“reemplazar todo”.

Actions:

Según la documentación oficial de Vuex, “Actions son similares a mutaciones”, la diferencia es que:

- En vez de mutar los states, las Actions hacen commit de las mutaciones (Podemos invocar cuantas mutaciones necesitemos dentro de una Action).

- Las Actions pueden contener operaciones asíncronas (Por ejemplo, hacer una petición a una Base Datos, y luego almacenar los datos en el store de Vuex).

EJERCICIO

Para comenzar a aprender sobre cómo utilizar actions, veremos como estas pueden invocar mutaciones haciendo commit. A través de los commit, es como se invocan las mutaciones que necesitemos utilizar, y estas a su vez, cambiarán los estados del store de Vuex.

Iremos a la carpeta store, y abriremos el archivo index.js. Dentro de éste, nos dirigiremos a la sección "actions".

La primera action que crearemos se llamará: **removeProduct**, ésta contendrá dos argumentos: **context**, el cual nos permitirá utilizar la función commit para poder invocar la mutación "REMOVE_PRODUCT"; y como segundo argumento, entra la id del producto.

```
1 actions: {  
2   removeProduct: (context, id) => {  
3     context.commit('REMOVE_PRODUCT', id);  
4   },  
5 }
```

Teniendo la action creada, vamos a utilizarla. Para ello, iremos a la carpeta "components", y abriremos el archivo ProductList.vue, luego al método "remove", y en vez de invocar a la mutación, usaremos un dispatch y llamaremos a la action **removeProduct**.

```
1 methods: {  
2   remove(id) {  
3     let response = confirm("¿Estas seguro de Eliminar el  
4     producto?");  
5     if(response) {  
6       this.$store.dispatch('removeProduct', id);  
7     }  
8   },  
9 }
```

Ahora, vamos a volver a store de Vuex, y crearemos otra Actions, llamada **"addProduct"**; ésta será la encargada de invocar a través de commit, a la mutación **"ADD_PRODUCT"**.

```
1 actions: {  
2   removeProduct: (context, id) => {  
3     context.commit('REMOVE_PRODUCT', id);  
4   },  
5   addProduct: ({commit}, product) => {  
6     commit('ADD_PRODUCT', product);  
7   }  
8 },
```

Si nos fijamos, la diferencia entre las dos Actions, es el primer argumento. En la primera, utilizamos "context"; y en la segunda, "**commit**", esto lo podemos hacer ya que context es un objeto con distintos métodos dentro, entre esos: commit, y es por ello que para solo obtener esa función, usaremos los paréntesis de llaves, con el nombre de la requerida {commit}. De esta manera, el código queda un poco más reducido, pero el funcionamiento es el mismo.

Teniendo Action creada, procederemos a utilizarla. Para ello, iremos a la carpeta "components", y abriremos el archivo "**ProductNew.vue**".

En el método add(), vamos a hacer un dispatch con el nombre de la Action: "**addProduct**".

```
1 add() {  
2   if(  
3     this.form.code !== ""  
4     && this.form.product !== ""  
5     && this.form.quantity !== ""  
6     && this.form.price !== ""  
7   ) {  
8     let data = {...this.form}  
9     this.$store.dispatch('addProduct', data);  
10    this.clean();  
11  }  
12 },
```




Si guardamos, veremos que podemos agregar o eliminar productos.



Navbar



Productos 3

Código	Producto	Cantidad	Precio	Acción
23452323	Funda papel	2	100	
23443323	Bandeja Carton	2	200	
24452329	Papel de regalo	2	500	
Total: \$1600				

Nuevo Producto

Código

Producto

Cantidad

Precio

 Agregar

EXERCISE 2: MAPACTIONS

INSTRUCCIONES

Para realizar el siguiente ejercicio, continuaremos trabajando en el mismo proyecto creado anteriormente: "cue13".

MapActions:

Al igual que hemos visto anteriormente, con `mapState` y `mapGetters`, también podemos utilizar `mapActions`, de esta manera, podremos utilizar e invocar un Action, como si fuera un método del componente.

Editar Producto:

Para comenzar, vamos a ir a la carpeta "store", y abriremos el archivo `index.js`.

En primer lugar, crearemos un nuevo state, llamado: `"id_product_edit"`.

```
1 id_product_edit:null,
```

El state completo quedará de esta manera:

```
1 state: {  
2   user_id:"325",  
3   name: 'Juan',  
4   last_name: 'Ramirez',  
5   products:[  
6     {id:1, code:'23452323', product: "Funda papel" , quantity:2,  
7 price:100},  
8     {id:2, code:'23443323', product: "Bandeja Carton" ,  
9 quantity:2, price:200},  
10    {id:3, code:'24452329', product: "Papel de regalo" ,  
11 quantity:2, price:500},  
12  ],  
13   id_product_edit:null,  
14 },
```

Lo segundo que haremos, será crear un nuevo getter llamado `"getProductByID"`, el cual nos retornará un producto, según la id que entreguemos como argumento.

```
1 getProductByID: (state) => (id) => {  
2   return state.products.find(prod => prod.id === id);  
3 }
```

Los getters completos quedarán de esta forma:

```
1 getters: {  
2   userName: state => {  
3     return state.name + " " + state.last_name;  
4   },  
5   countProducts: state => {  
6     return state.products.length;  
7   },  
8   totalProducts: state => {  
9     return state.products.reduce((total, prod) => {  
10      return total + (prod.quantity * prod.price)  
11    }, 0)  
12   },  
13   getProductByID: (state) => (id) => {  
14     return state.products.find(prod => prod.id === id);  
15   }  
16 }
```

```
17 },
```

Crearemos dos mutaciones:

1. **SET_ID_PRODUCT_ID**: nos servirá para guardar la id del producto a editar.
2. **EDIT_PRODUCT**: nos servirá para guardar el producto editado en el state.

```
1 SET_ID_PRODUCT_ID: (state, id) => {
2   state.id_product_edit = id;
3 },
4 EDIT_PRODUCT: (state, product) => {
5   let index = state.products.findIndex(prod => prod.id === product.id);
6   Vue.set(state.products, index, product);
7
8   state.id_product_edit = null;
9 }
```

Las mutaciones quedarán de esta forma:

```
1 mutations: {
2   REMOVE_PRODUCT: (state, id) => {
3     let index = state.products.findIndex(prod => prod.id === id);
4     state.products.splice(index, 1);
5   },
6   ADD_PRODUCT: (state, product) => {
7     product.id = Math.floor(Math.random() * 1000);
8     state.products.push(product);
9   },
10  SET_ID_PRODUCT_ID: (state, id) => {
11    state.id_product_edit = id;
12  },
13  EDIT_PRODUCT: (state, product) => {
14    let index =
15    state.products.findIndex(prod => prod.id === product.id);
16    Vue.set(state.products, index, product);
17
18    state.id_product_edit = null;
19  }
20 },
```

En la sección de Actions, crearemos 2 para invocar a las mutaciones anteriores.

```
1 setIdProductEdit: ({ commit }, id) => {
2   commit('SET_ID_PRODUCT_ID', id);
3 },
```

```

4 editProduct: ({commit}, product) => {
5   commit('EDIT_PRODUCT', product);
6 }

```

Las Actions quedarán de la siguiente forma:

```

1 actions: {
2   removeProduct: (context, id) => {
3     context.commit('REMOVE_PRODUCT', id);
4   },
5   addProduct: ({commit}, product) => {
6     commit('ADD_PRODUCT', product);
7   },
8   setIdProductEdit: ({commit}, id) => {
9     commit('SET_ID_PRODUCT_ID', id);
10  },
11  editProduct: ({commit}, product) => {
12    commit('EDIT_PRODUCT', product);
13  }
14 },

```

Una vez creada Actions, generaremos un nuevo archivo dentro de la carpeta "components/products", llamado: **ProductEdit.vue**.

Para crearlo, nos vamos a basar en el componente: **ProductNew.vue**.

```

1 <template>
2   <div class="container">
3     <div class="title"><h2>Editar Producto</h2></div>
4     <div class="code">
5       <label for="">Código</label>
6       <input type="text" v-model="form.code">
7     </div>
8     <div class="product">
9       <label for="">Producto</label>
10      <input type="text" v-model="form.product">
11    </div>
12    <div class="quantity">
13      <label for="">Cantidad</label>
14      <input type="text" v-model="form.quantity">
15    </div>
16    <div class="price">
17      <label for="">Precio</label>
18      <input type="text" v-model="form.price">
19    </div>
20    <div class="add">
21      <button @click="edit">
22        <i class="fas fa-edit"></i>

```




```
23     Editar
24     </button>
25   </div>
26 </div>
27 </template>
28
29 <script>
30 import {mapGetters,mapState,mapActions} from 'vuex';
31 export default {
32   name: 'ProductEdit',
33   // props: {},
34   data: function() {
35     return {
36       form: {
37         code: "",
38         product: "",
39         quantity: "",
40         price: "",
41       }
42     }
43   },
44   computed: {
45     ...mapGetters(['getProductByID']),
46     ...mapState(['id_product_edit'])
47   },
48   methods: {
49     ...mapActions(['editProduct']),
50     edit() {
51       if(
52         this.form.code !== ""
53         && this.form.product !== ""
54         && this.form.quantity !== ""
55         && this.form.price !== ""
56       ) {
57         let data = {...this.form, id: this.id_product_edit};
58         this.editProduct(data);
59         this.clean();
60       }
61     },
62     clean() {
63       this.form.code = ""
64       this.form.product = ""
65       this.form.quantity = ""
66       this.form.price = ""
67     },
68     setProduct() {
69       let prod = this.getProductByID(this.id_product_edit);
70       this.form.code = prod.code;
71       this.form.product = prod.product;
72       this.form.quantity = prod.quantity;
73       this.form.price = prod.price;
74     }
75   }
76 }
```



```
75 },
76 // components: {},
77 mounted(){
78   this.setProduct();
79 }
80 }
81 </script>
82
83 <style scoped>
84   .container{
85     display:grid;
86     grid-template-columns: repeat(5,1fr);
87     text-align:start;
88     padding:10px;
89     width:500px;
90     margin: 0 auto;
91     background:rgb(140, 163, 9);
92     margin-top:10px;
93   }
94   .title{
95     grid-column:1/4;
96     grid-row:1/2;
97   }
98   .code{
99     grid-column:1/3;
100    grid-row:2/3;
101  }
102  .product{
103    grid-column:1/4;
104    grid-row:3/4;
105  }
106  .quantity{
107    grid-column:1/2;
108    grid-row:4/5;
109  }
110  .price{
111    grid-column:2/4;
112    grid-row:4/5;
113  }
114  .add{
115    grid-column:1/5;
116    grid-row:5/6;
117    margin:auto 0;
118  }
119  input{
120    width: 90%;
121  }
122  h2{
123    color:white;
124  }
125  label{
126    color:white;
```

```
127 }  
128 </style>
```

Este componente nos permitirá recuperar los datos de un producto en específico, desde Vuex, y lo mostrará para poder editarlo.

Para visualizar este nuevo componente, debemos dirigirnos a la carpeta "views", y abriremos el archivo **Products.vue**, importaremos el nuevo componente, y crearemos un condicional para mostrar lo siguiente:

```
1 <template>  
2   <div>  
3     <ProductCount></ProductCount>  
4     <ProductList></ProductList>  
5     <ProductTotal></ProductTotal>  
6     <ProductNew v-if="id_product_edit===null"></ProductNew>  
7     <ProductEdit v-else></ProductEdit>  
8   </div>  
9 </template>  
10  
11 <script>  
12 import ProductCount from '@components/products/ProductCount.vue'  
13 import ProductList from '@components/products/ProductList.vue'  
14 import ProductTotal from '@components/products/ProductTotal.vue'  
15 import ProductNew from '@components/products/ProductNew.vue'  
16 import ProductEdit from '@components/products/ProductEdit.vue'  
17 import {mapState} from 'vuex';  
18 export default {  
19   name: 'Product-view',  
20   // props: {},  
21   data: function() {  
22     return {}  
23   },  
24   computed: {  
25     ...mapState(['id_product_edit']),  
26   },  
27   methods: {  
28     // -- Metodos  
29   },  
30   components: {  
31     ProductCount,  
32     ProductList,  
33     ProductTotal,  
34     ProductNew,  
35     ProductEdit  
36   },  
37 }  
38 </script>  
39
```

```
40 <style scoped>
41
42 </style>
```

Para mostrar tanto el componente **"ProductNew.vue"**, como **"ProductEdit"**, utilizaremos un state de Vuex, llamado: **"id_product_edit"**.

Por último, nos quedará agregar un botón para poder editar el producto. Esto lo haremos en **"components/products/"**, y el nombre del archivo es: **"ProductList.vue"**.

En la sección de template, agregaremos un nuevo botón para editar, y le ligaremos un evento clic con un método de Vuex, llamado **"SetIdProductEdit"**, el cual debe recibir la id del producto.

```
1 <template>
2   <div>
3     <div class="container">
4       <div class="tab">Código</div>
5       <div class="tab">Producto</div>
6       <div class="tab">Cantidad</div>
7       <div class="tab">Precio</div>
8       <div class="tab">Acción</div>
9     </div>
10    <div class="container" v-for="product in products"
11      :key="product.id">
12      <div class="tab_content">{{product.code}}</div>
13      <div class="tab_content">{{product.product}}</div>
14      <div class="tab_content">{{product.quantity}}</div>
15      <div class="tab_content">{{product.price}}</div>
16      <div class="tab_content">
17        <button @click="remove(product.id)">
18          <i class="fas fa-trash"></i>
19        </button>
20        <button @click="setIdProductEdit(product.id)">
21          <i class="fas fa-edit"></i>
22        </button>
23      </div>
24    </div>
25  </div>
26 </template>
```

En la sección de script, utilizaremos un **mapAction**, para traer la función utilizada en el template y poder editar.

```
1 ...mapActions(['setIdProductEdit']),
```

```

1 <script>
2 import {mapState,mapActions} from 'vuex'
3 export default {
4   name: 'Product-list',
5   // props: {},
6   data: function(){
7     return {}
8   },
9   computed: {
10    ...mapState(['products'])
11  },
12  methods: {
13    ...mapActions(['setIdProductEdit']),
14    remove(id){
15      let response =confirm("¿Estas seguro de Eliminar el
16 producto?");
17      if(response){
18        this.$store.dispatch('removeProduct',id);
19      }
20    },
21  },
22 },
23 // components: {},
24 }
25 </script>

```

El componente **ProductEdit.vue**, al final debe quedar de esta forma:

```

1 <template>
2   <div>
3     <div class="container">
4       <div class="tab">Código</div>
5       <div class="tab">Producto</div>
6       <div class="tab">Cantidad</div>
7       <div class="tab">Precio</div>
8       <div class="tab">Acción</div>
9     </div>
10    <div class="container" v-for="product in products"
11    :key="product.id">
12      <div class="tab_content">{{product.code}}</div>
13      <div class="tab_content">{{product.product}}</div>
14      <div class="tab_content">{{product.quantity}}</div>
15      <div class="tab_content">{{product.price}}</div>
16      <div class="tab_content">
17        <button @click="remove(product.id)">
18          <i class="fas fa-trash"></i>
19        </button>
20        <button @click="setIdProductEdit(product.id)">

```







```
21     <i class="fas fa-edit"></i>
22   </button>
23 </div>
24 </div>
25 </div>
26 </template>
27
28 <script>
29 import {mapState,mapActions} from 'vuex'
30 export default {
31   name: 'Product-list',
32   // props: {},
33   data: function(){
34     return {}
35   },
36   computed: {
37     ...mapState(['products'])
38   },
39   methods: {
40     ...mapActions(['setIdProductEdit']),
41     remove(id){
42       let response =confirm("¿Estas seguro de Eliminar el
43 producto?");
44       if(response){
45         this.$store.dispatch('removeProduct',id);
46       }
47     },
48   },
49   // components: {},
50 }
51 </script>
52
53
54 <style scoped>
55   .container{
56     display:grid;
57     grid-template-columns: 1fr 4fr 1fr 2fr 1.5fr;
58   }
59   .tab_content{
60     background:orange;
61   }
62   .tab{
63     background:grey;
64     color:white;
65   }
66 </style>
```

Si guardamos, y visitamos la ruta `"/products"`, podremos ver un nuevo botón editar en la lista.

Navbar



Productos 3

Código	Producto	Cantidad	Precio	Acción
23452323	Funda papel	2	100	 
23443323	Bandeja Carton	2	200	 
24452329	Papel de regalo	2	500	 
Total: \$1600				


Nuevo Producto

Código

Producto

Cantidad

Precio

 Agregar

Si lo presionamos en cualquiera de los productos de la lista, el componente **ProductNew** será remplazado por **ProductEdit**.

Vamos a hacer clic en editar el primer producto.

Productos 3

Código	Producto	Cantidad	Precio	Acción
23452323	Funda papel	2	100	
23443323	Bandeja Carton	2	200	
24452329	Papel de regalo	2	500	
Total: \$1600				

Editar Producto

Código

Producto

Cantidad

Precio

Editar

Editaremos el producto "Funda papel 2", y cambiaremos la cantidad a 10.

Editar Producto

Código

Producto


Cantidad

Precio

Editar

Si presionamos en editar, veremos que la lista se refrescará, y el componente Editar será ocultado para mostrar el que crea un nuevo producto.

Productos 3

Código	Producto	Cantidad	Precio	Acción
23452323	Funda papel 2	10	100	 
23443323	Bandeja Carton	2	200	 
24452329	Papel de regalo	2	500	 
Total: \$2400				

Nuevo Producto

Código

Producto

Cantidad

Precio

 Agregar

EXERCISE 3: LLAMADAS ASÍNCRONAS EN ACTIONS

INSTRUCCIONES

Para realizar este ejercicio, seguiremos trabajando en el mismo proyecto que hemos utilizado hasta ahora: "cue13".

Instalando Axios

Como realizaremos llamadas asíncronas, debemos instalar en nuestro proyecto la librería Axios, la cual nos permite hacer peticiones HTTP.

Para ello, abriremos el terminal de Visual Studio Code, teniendo nuestro proyecto abierto.

"npm install axios"

```
$ npm install axios
```

Simulando un servidor

Para simular un servidor backend, utilizaremos la librería "json-server".

JSON

O JavaScript Object Notation, es un formato de texto sencillo, que se utiliza para representar datos estructurados en la sintaxis de objetos de JavaScript. Comúnmente, se emplea para transmitir datos en aplicaciones web, por ejemplo, desde el servidor hacia una página. A pesar de su nombre, este formato de datos se puede utilizar independientemente de JavaScript.

Un JSON, es una cadena cuya estructura es semejante a la de los objetos de JavaScript, y permite los mismos tipos de datos que un objeto (cadenas, números, booleanos, arreglos, entre otros). Es importante señalar que éstos solo contienen propiedades, no métodos.

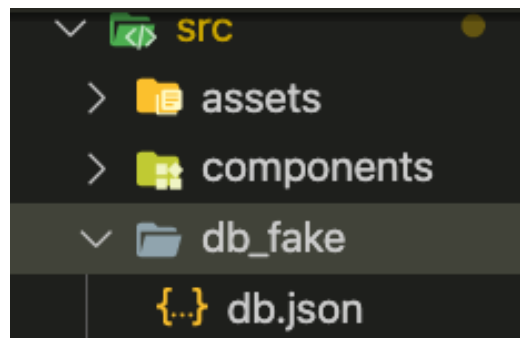
[Documentación JSON-SERVER](#)

Para utilizarla, lo primero que debemos hacer es instalar json-server de manera global, en nuestro terminal. Utilizaremos el comando:

npm install -g json-server

```
npm install -g json-server
```

Una vez instalado, vamos a crear una carpeta nueva dentro de "src", llamada "db_fake", y dentro de ella, un archivo llamado "db.json".



Dentro de este archivo, traeremos los mismos productos almacenados en Vuex, solo debemos tener la precaución de agregar las comillas a los nombres de cada elemento json.

```

1 {
2   "products": [
3     {
4       "code": "23452323",
5       "product": "Funda papel 2 ",
6       "quantity": 2,
7       "price": 100,
8       "id": 1
9     },
10    {
11      "id": 3,
12      "code": "24452329",
13      "product": "Papel de regalo",
14      "quantity": 2,
15      "price": 500
16    },
17    {
18      "code": "643454",
19      "product": "Hojas de Papel",
20      "quantity": "100",
21      "price": "10",
22      "id": 4
23    }
24  ]
25 }
```

Para poder levantar el servidor, iremos al terminal, y nos posicionaremos en `"src/db_fake"`.

```

alejandros-MacBook-Pro:cue13 bonilla$ cd src/db_fake
alejandros-MacBook-Pro:db_fake bonilla$ ls
db.json
```

Una vez ubicado en el directorio, utilizaremos el programa instalado "json-serve": `json-server --watch db.json`.

```
alejandros-MacBook-Pro:db_fake bonilla$ json-server --watch db.json
```

Esto nos levantará un servidor en el puerto 3000.

```
\{^_^}/ hi!

Loading db.json
Done

Resources
http://localhost:3000/products

Home
http://localhost:3000

Type a command to start a new route, or a command to create a new resource.
```

Si visitamos el recurso en el navegador, podremos obtener los productos, simulando una base de datos externa.

<http://localhost:3000/products>

```
localhost:3000/products

Aplicaciones Lyrics Sign in · GitLab

[
  - {
    code: "23452323",
    product: "Funda papel 2 ",
    quantity: 2,
    price: 100,
    id: 1
  },
  - {
    id: 3,
    code: "24452329",
    product: "Papel de regalo",
    quantity: 2,
    price: 500
  },
  - {
    code: "643454",
    product: "Hojas de Papel",
    quantity: "100",
    price: "10",
    id: 4
  }
]
```

Actions asíncronos:

En algunas ocasiones, necesitamos obtener información desde una base de datos, para luego almacenarla en Vuex, es por esto que las Actions nos permiten realizar esta operación.

Vamos a ir a la carpeta “store”, y abriremos el archivo index.js.

En la parte superior del archivo, importaremos Axios.

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

Axios, nos servirá para realizar peticiones al servidor fake.

En state, el dato **products**, debe quedar vacío.

```
state: {
  user_id: "325",
  name: 'Juan',
  last_name: 'Ramirez',
  products: [],
  id_product_edit: null,
},
```

Products, debe quedar vacío, ya que traeremos los datos desde el servidor que creamos.

En la sección de mutaciones, crearemos una llamada “**SET_PRODUCTS**”, la cual nos permitirá almacenar los datos en el state products, quedando así:

```
1 SET_PRODUCTS: (state, products) => {
2   state.products = products;
3 }
```

En la sección de Actions, crearemos una nueva, llamada “fetchProducts”, y en ella realizaremos la llamada asíncrona; cuando ésta se resuelva, llamaremos la mutación “**SET_PRODUCTS**”, para guardar los datos que vienen desde la API REST.

```
1 fetchProducts: ({commit}) => {
2   axios.get('http://localhost:3000/products')
3   .then(resp => {
4     console.log(resp)
5     commit('SET_PRODUCTS', resp.data);
```



```

6      })
7      .catch(error=>{
8          console.log(error)
9      })
10     }
  
```

Para utilizar esta action, iremos a la carpeta “Views”, y abriremos el archivo Products.vue. Importaremos mapActions, para poder utilizar la action “fetchProducts”.

```

1 import {mapState, mapActions} from 'vuex';
  
```

En los methods, llamaremos a este mapActions.

```

1 methods: {
2     // -- Metodos
3     ...mapActions(['fetchProducts'])
4 },
  
```

Luego, este método hará la petición cuando el componente sea creado. Es por ello que utilizaremos el ciclo de vida “created”.

```

1 created() {
2     this.fetchProducts();
3 }
  
```

De esta manera, al momento de ser creada la vista, todos los componentes asociados, podrán utilizar los datos que vienen desde el servidor de pruebas creado por nosotros.

Si guardamos, y vamos a la ruta “/products”, podremos ver que ahora los mismos datos han sido traídos desde el servidor.

Navbar				
Productos 3				
Código	Producto	Cantidad	Precio	Acción
23452323	Funda papel 2	2	100	 
24452329	Papel de regalo	2	500	 
643454	Hojas de Papel	100	10	 
Total: \$2200				

Ahora, nos queda modificar las acciones de: eliminar, crear, y editar productos, para que estos puedan ser almacenados en nuestro servidor.

Eliminar producto

Partiremos por la actions **"removeProduct"**.

```
1 removeProduct: (context, id) => {  
2   axios.delete(`http://localhost:3000/products/${id}`)  
3   .then(resp => {  
4     console.log(resp)  
5     context.commit('REMOVE_PRODUCT', id);  
6   })  
7   .catch(error => {  
8     console.log(error)  
9   })  
10  },
```

En esta utilizaremos Axios, con la cabecera delete, para indicarle al servidor que vamos a borrar la id enviada por la url.

Una vez el servidor devuelva la respuesta, invocaremos la mutación anteriormente creada.

Agregar nuevo producto

Ahora vamos a **"addProduct"**.

```
1 addProduct: ({commit}, product) => {  
2   axios.post('http://localhost:3000/products', product)  
3   .then(resp => {  
4     console.log(resp)  
5     commit('ADD_PRODUCT', resp.data);  
6   })  
7   .catch(error => {  
8     console.log(error);  
9   })  
10  },
```

En este caso, para poder crear un nuevo registro en la API-REST, debemos utilizar la cabecera post; ésta debe incluir como segundo parámetro los datos a agregar.

Esta llamada nos devuelve una respuesta si la petición es correcta. En ese caso, invocaremos a la mutación **"ADD_PRODUCT"**.

En caso contrario, por consola mostraremos el error capturado en catch.

Editar producto

```

1 editProduct: ({commit}, product) => {
2   axios.put(`http://localhost:3000/products/${product.id}`,
3   product)
4   .then(resp => {
5     console.log(resp)
6     commit('EDIT_PRODUCT', resp.data);
7   })
8   .catch(error => {
9     console.log(error)
10  })
11 },

```

Para poder editar, utilizaremos la cabecera “put”; la dirección del servidor debe ir acompañada por la id del producto a editar, y además, como segundo parámetro, los datos del producto editado.









En caso de la respuesta sea exitosa, llamaremos a la mutación “**EDIT_PRODUCT**”.

Si guardamos, y visitamos la ruta “/products”, podremos ver que los datos van a persistir, ya que hemos vinculado todas las acciones al servidor Api que instalamos al comienzo.

Navbar



Productos 4

Código	Producto	Cantidad	Precio	Acción
23452323	Funda papel 2	2	100	 
24452329	Papel de regalo	2	500	 
643454	Hojas de Papel	100	10	 
55453323	nuevo Producto	10	200	 

Total: \$4200

Nuevo Producto

Código

Producto

Cantidad

Precio

 Agregar