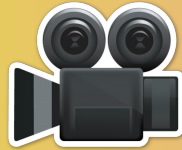




Clase 12. Vue JS

Vuex en Vue

RECORDÁ PONER A GRABAR LA CLASE





¿DUDAS DEL ON-BOARDING?

MIRALO AQUI



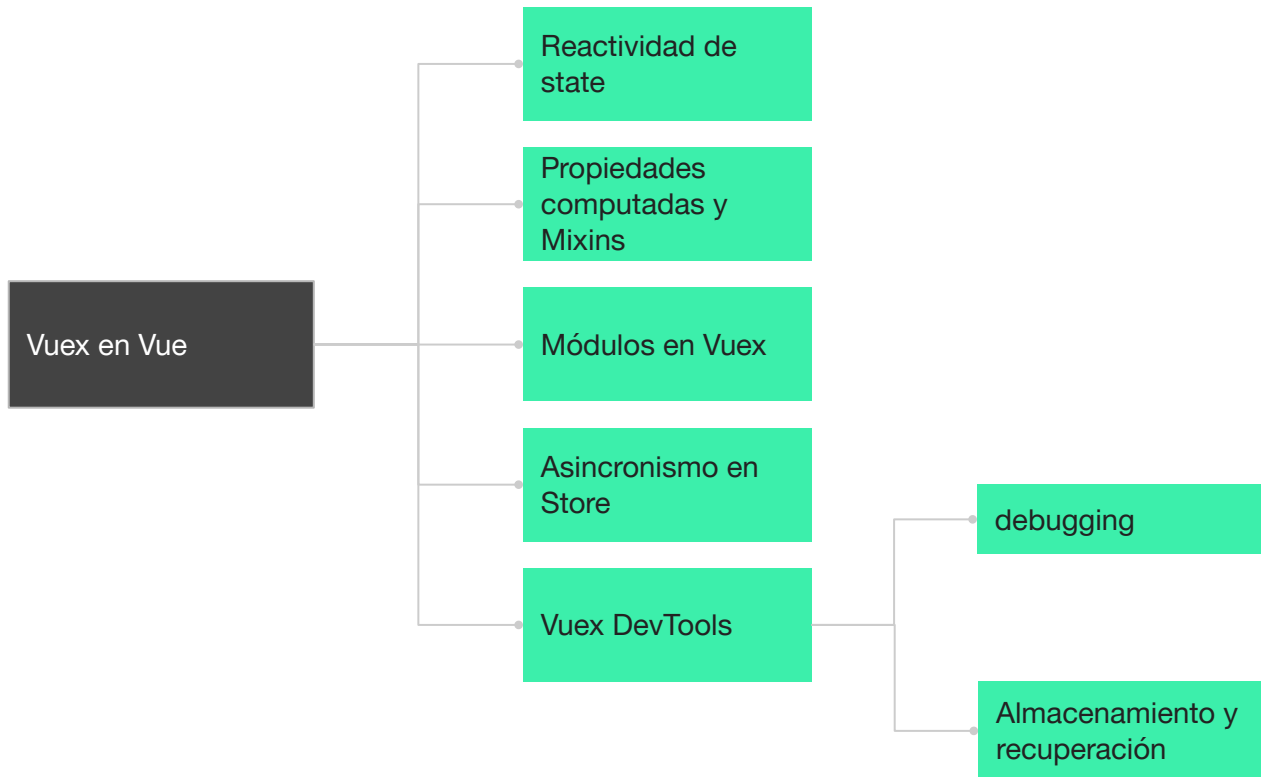
OBJETIVOS DE LA CLASE

- Comprender el mecanismo de conexión de Vuex con los componentes del proyecto y las características avanzadas de su uso.

MAPA DE CONCEPTOS

MAPA DE CONCEPTOS CLASE 12

¡Para
recordar!



CRONOGRAMA DEL CURSO

Clase 11



Consumo de API Rest



VUEX EN VUE CLI 2



ESTRUCTURA VUEX



EJEMPLO EN VIVO

Clase 12



Introducción a Vuex



VUEX Y ASINCRONISMO



EJEMPLO EN VIVO



CONECTAR STORE Y
CHROME DEVTOOLS



VUE CDN A VUE CLI

Clase 13



Vuex en Vue



CREAR UN BUILD DE
PROYECTO CLI



EJEMPLO EN VIVO



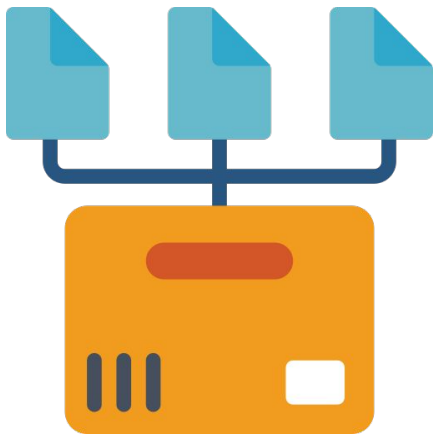
PUBLICAR EL BUILD EN
SERVIDOR DE NUBE



ENTREGA INTERMEDIA

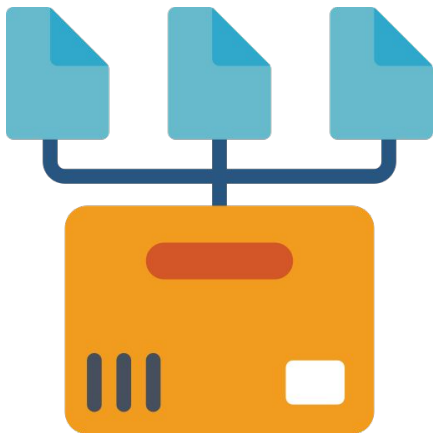
VUEX Y GETTERS

AJUSTE FINO



La clase anterior introdujimos el Concepto de Vuex, Store, state, mutations y actions, entendiendo cómo Vuex aporta un sistema de almacenamiento central para cualquier módulo de nuestras aplicaciones Vue/Cli.

AJUSTE FINO



Antes de sumergirnos en los conceptos más avanzados de Vuex, **hagamos un repaso de buenas prácticas de la P.O.O. que debemos traer a Vuex**, para securizar los datos que manipulamos todo el tiempo.

getters

GETTERS



Cuando trabajamos con `state`, sabemos que podemos acceder a estos mediante la ruta `$store.state.propiedad` pero, de acuerdo a las buenas prácticas de la P.O.O., no debemos acceder a estas propiedades de forma directa, sino que debemos hacerlo utilizando `getters`.



GETTERS

La integración de **getters** la realizamos dentro de la declaración del Store global de Vuex. Puede ser, seguida a cualquiera de los objetos declarados aquí: (**state**, **actions**, **mutations**).

Una vez agregado, definiremos un **getter** por cada propiedad que tengamos declarada en **state**.

```
export default new Vuex.Store({  
  state: {  
    msg: 'Variable de State  
          declarada en Store.',  
    nombreDelCurso: 'Vue y Vuex!'  
  },  
  ...  
},  
  getters: {  
    ...  
  }  
})
```



GETTERS

Si bien, dentro del mismo objeto (**store**) tenemos acceso a cualquier elemento de este, (**scope**), conviene pasarle **state** como parámetro a cada **getter** declarado para así definir, dentro de este, toda la ruta completa.

De esta forma, cuando miremos el código tiempo después sabremos el origen de lo que utilizamos internamente en él.

```
...  
getters: {  
  getMsg: (state)=> {  
    return state.msg  
  },  
  getNombreDelCurso: (state)=> {  
    return state.nombreDelCurso  
  }  
}  
...
```



GETTERS

```
1 <template>
2   <div class="hello">
3     <h1>{{ $store.state.msg }}</h1>
4     <h2>{{ $store.state.nombreDelCurso }}</h2>
5     <br><br>
6     <button @click="cambioNombre">CAMBIAR CURSO</button>
7   </div>
8 </template>
9
10 <script>
11 export default {
12   name: 'HelloWorld',
13   methods: {
14     cambioNombre: () => {
15       this.$store.dispatch('cambiarNombreDelCurso')
16     }
17   }
18 }
```

Nos queda reemplazar en el o los componentes donde consumimos `store.state` la ruta correspondiente hacia el o los `getters` creados.

Probemos nuestro proyecto luego del cambio, y veremos que llegamos al mismo resultado.



OPERACIONES CON GETTERS

Incluso, el uso de getters nos sirve también para hacer operaciones varias sobre propiedades almacenadas en **state**. Si, por ejemplo, tengo dos propiedades las cuales debo mostrar concatenadas, con un **getter** dedicado, resolvería esto fácilmente.

```
//propiedades en state
state: {
  msg: 'Variable de State
        declarada en Store.',
  nombreDelCurso: 'Vue y Vuex!',
  profesion: 'Coder',
  ubicacion: 'house'
},

//getter
getters: {
  ...
  getConcatStates: (state)=> {
    return `${state.profesion}
    ${state.ubicacion}`
  }
}
```




OPERACIONES CON GETTERS

```
index.js — vu  
src > store > index.js > default  
21   },  
22   },  
23   getters: {  
24     getMsg: (state) => {  
25       return state.msg  
26     },  
27     getNombreDelCurso: (state) => {  
28       return state.nombreDelCurso  
29     },  
30   },  
31   },  
32   },  
33   }
```

De igual forma en la cual realizamos operaciones como esta: una simple concatenación; también podemos integrar cualquier otro tipo de operación de conversión, cálculos matemáticos, etcétera.



MÓDULOS EN VUEX



MÓDULOS EN VUEX

Esta porción de código representa el archivo Store que trabajamos la clase anterior, incluyendo states, mutations y actions, además de los getters que agregamos al inicio de esta clase.

Como podemos ver, a lo largo del simple ejemplo que elaboramos alcanzamos poco más de 30 líneas de código...

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

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    msg: 'Variable de State declarada en Store.',
    nombreDelCurso: 'Vue y Vuex!',
    profesion: 'Coder',
    ubicacion: 'house'
  },
  mutations: {
    cambiarCurso: (state)=> {
      state.nombreDelCurso = 'Vuex en Vue'
    }
  },
  actions: {
    cambiarNombreDelCurso: ( context )=> {
      context.commit(`cambiarCurso`)
    }
  },
  getters: {
    getMsg: (state)=> {
      return state.msg
    },
    getNombreDelCurso: (state)=> {
      return state.nombreDelCurso
    },
    getConcatStates: (state)=> {
      return `${state.profesion} ${state.ubicacion}`
    }
  }
})
```



MÓDULOS EN VUEX

... ¿se imaginan qué pasaría si este Store debiera alimentar un proyecto más complejo, que maneje al menos 20 propiedades dentro de state, con sus respectivos getters, actions, mutations, más alguna que otra validación?

¡Exacto! El Store se tornaría algo difícil de mantener, ya que crecería alrededor de 300 líneas de código. 😞

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

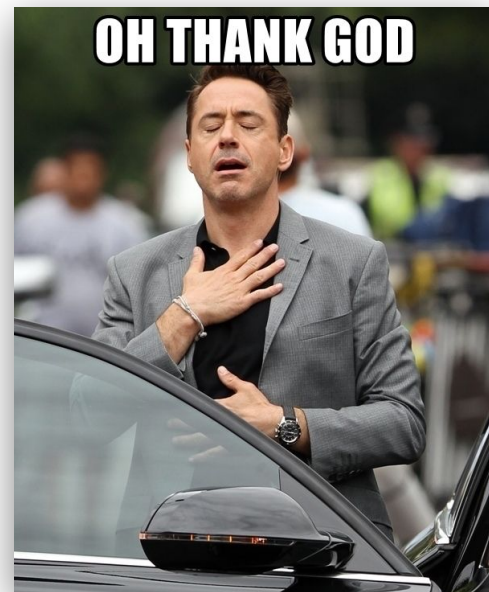
Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    msg: 'Variable de State declarada en Store.',
    nombreDelCurso: 'Vue y Vuex!',
    profesion: 'Coder',
    ubicacion: 'house'
  },
  mutations: {
    cambiarCurso: (state)=> {
      state.nombreDelCurso = 'Vuex en Vue'
    }
  },
  actions: {
    cambiarNombreDelCurso: ( context )=> {
      context.commit(`cambiarCurso`)
    }
  },
  getters: {
    getMsg: (state)=> {
      return state.msg
    },
    getNombreDelCurso: (state)=> {
      return state.nombreDelCurso
    },
    getConcatStates: (state)=> {
      return `${state.profesion} ${state.ubicacion}`
    }
  }
})
```

MÓDULOS EN VUEX

Por ello, para que esto no se vuelva un problema,
Vuex propone una pequeña modificación a su estructura, en pos de que la consistencia y longitud de cada uno de nuestros objetos del Store, sean mucho más fáciles de mantener.

A esto se le denomina: **Módulos**. 🙌

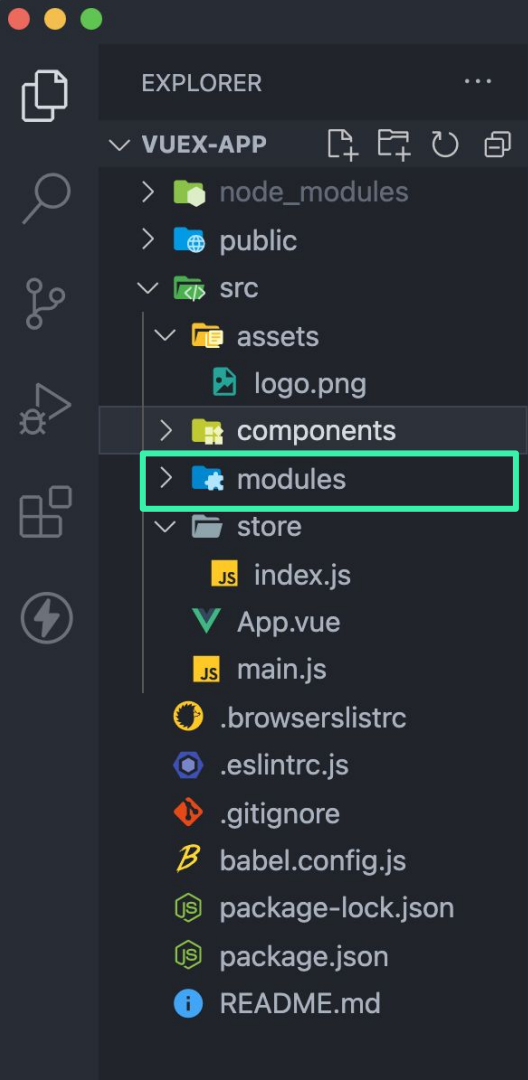


CREAR UNA SUBCARPETA PARA MÓDULOS



SUBCARPETA PARA MÓDULOS

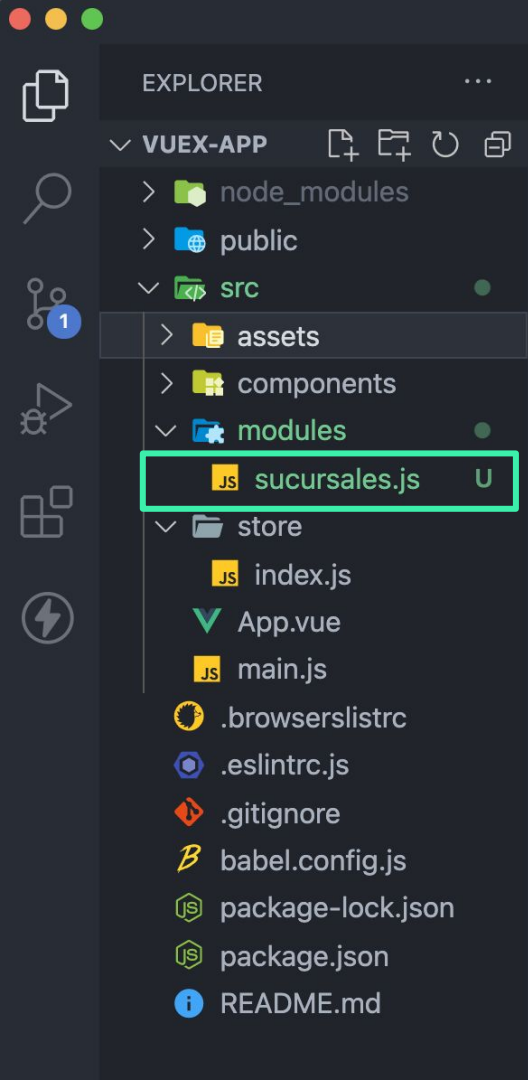
Lo primero que debemos realizar es crear dentro de la carpeta de nuestro proyecto, `/src/`, una subcarpeta a la cual podemos denominar, por ejemplo, `modules`.





ARCHIVO JAVASCRIPT

Ahora, crearemos dentro de esta subcarpeta, un archivo del tipo **JS**. Suponiendo que debemos crear un array que incluye datos claves en nuestra aplicaciones, creamos el archivo en cuestión con el nombre de lo que trabajaremos dentro de éste.





ESTRUCTURA DE UN OBJETO

JS sucursales.js ×

```
export default {  
  state: {  
  
  },  
  mutations: {  
  
  },  
  actions: {  
  
  },  
  getters: {  
  
  }  
}
```

En el archivo en cuestión, definimos un objeto exportable, a través de las palabras reservadas `export default` y, en su interior, la estructura base de un `Store`:

- State
- Mutations
- Actions
- getters

ESTRUCTURA DE UN OBJETO



De esta forma, terminamos exportando desde un archivo Javascript independiente, toda la información referente a un objeto individual que podíamos trabajar directamente en un **store**.

Pero, para facilitar su mantenimiento, lo separamos de este último.

ARMAR EL OBJETO INDEPENDIENTE



ESTRUCTURA DE UN OBJETO

```
export default {  
  state: {  
    sucursales =  
      [  
        'CABA', 'CÓRDOBA',  
        'MONTEVIDEO', 'LIMA',  
        'LAS CONDES'  
      ]  
  },  
  ...  
  getters: {  
    getSucursales: (state)=> {  
      return state.sucursales  
    }  
  }  
}
```

Ya podemos armar la estructura necesaria dentro de nuestro objeto: state, getters y demás secciones.

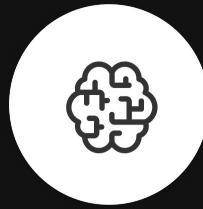
Así resolvemos todo en un espacio independiente, evitando incrementar notablemente nuestro Store, y que este termine colapsando.



IMPORTARLO HACIA EL STORE

Ahora nos queda indicarle al Store principal, cómo utilizar nuestro objeto independiente. Para ello, tenemos que importarlo primero hacia el Store.

```
//store/index.js
import Vue from 'vue'
import Vuex from 'vuex'
import tareas from '/src/modules/sucursales.js'
...
```



¡PARA PENSAR!

*En la diapositiva anterior importamos el archivo **sucursales.js**. Cuando usamos la sentencia **import... from**, para referenciar una dependencia, es necesario agregar la extensión de la misma para que funcione correctamente.*

¿VERDADERO O FALSO?
CONTESTA LA ENCUESTA DE ZOOM





IMPORTARLO HACIA EL STORE

Definimos una nueva sección dentro del `Store`,
denominada `modules:`.

El paso siguiente es declarar el módulo importado en la
sección creada, tal como muestra el código contiguo.



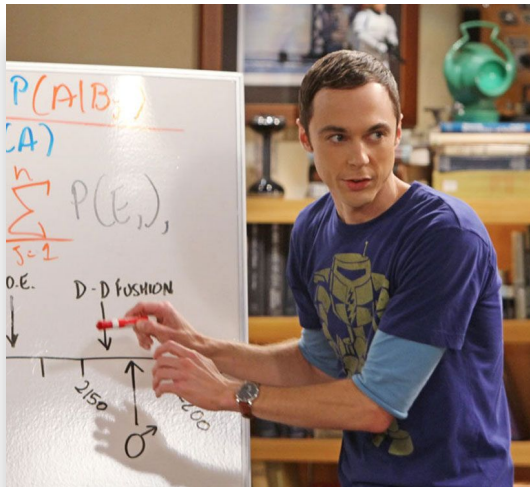
Aún así, no podemos usar nuestro array del objeto
`sucursales.js`, más allá de la
importación/exportación.

Pero... ¿por qué? 🤔

```
//store/index.js
export default new Vuex.Store({
  ...
},
modules: {
  sucursales
}
})
```



IMPORTARLO HACIA EL STORE



Porque en nuestro `Store` principal recibimos un array de objeto y no el `state` puro.

Para poder utilizarlo debemos **activar el espacio de nombres** en nuestro módulos, además de implementar `mapState`.

Y, ¿qué es `mapState`?



mapState

Vuex incluye una serie de helpers, entre ellos, **mapState**. Se ocupa del mapeo de estados hacia las propiedades computadas, a través de componentes.

mapState simplifica esta tarea, haciéndola incluso más fácil de integrar a nuestros **states**, cuando sea necesario.



INTEGRAR `mapState`

La integración de `mapState`
se realiza a través de
`spread operator`.

Como primer parámetro, va
el nombre del namespace a
mapear: en nuestro caso,
`sucursales`.

El segundo parámetro
corresponde a lo declarado en
dicho `store`.

```
...mapState('namespace', ['propiedad'])
```

Separado por comas, van el resto de declaraciones que
queramos agregar al mapeo.



mapState y namespaced

```
export default {  
  namespaced: true,  
  state: {  
    sucursales: ['CABA', 'CÓRDOBA'...
```

Agregamos **namespaced** 🖐 a nuestro módulo **sucursales.js** e importamos y sumamos **mapState** a la propiedad computada, para resolver esta necesidad en el componente web donde usaremos nuestro Store sucursales. 🖐

```
...  
</template>  
  
<script>  
import {mapState} from 'vuex'  
export default {  
  name: 'sucursales',  
  computed: {  
    ...mapState('sucursales',  
      ['sucursales'])  
  }  
  ...
```

REACTIVIDAD DEL STATE



REACTIVIDAD DEL STATE

```
//index.js
state: {
  msg: 'Variable de State
      declarada en Store'
}

-----

//HelloWorld.vue
<template>
  <div class="hello">
    <h1>{{ $store.getters.getMsg }}</h1>
  </div>
</template>
```

La clase pasada aprendimos que `state` es una evolución de data, dentro del ecosistema Vuex.

También aprendimos que podemos leer su contenido de forma directa, referenciando el path a través de la ruta `this.$store.state.propiedad` pero, a través de las buenas prácticas, debemos hacer su lectura mediante `getters` y no de forma directa.



REACTIVIDAD DEL STATE

```
mutations: {  
  cambiarCurso: (state)=> {  
    state.nombreDelCurso = 'Vuex en Vue'  
  },  
  ...  
actions: {  
  cambiarNombreDelCurso: ( context )=> {  
    context.commit(`cambiarCurso`)  
  },  
  ...  
}
```

Y, por supuesto, que toda modificación o necesidad de añadir datos en una propiedad declarada en state debe hacerse a través de **mutations**, quienes impactarán directamente el nuevo valor o la modificación de uno existente, pero siendo llamadas a través de **actions**, y no de forma directa.



REACTIVIDAD DEL STATE

```
mutations: {  
  cambiarCurso: (state)=> {  
    state.nombreDelCurso = 'Vuex en Vue'  
  },  
  ...  
actions: {  
  cambiarNombreDelCurso: ( context )=> {  
    context.commit(`cambiarCurso`)  
  },  
  ...  
}
```

Por lo tanto, cualquier cambio que hagamos mediante **mutations** - **actions** sobre una propiedad del state, o alguna petición API de datos remotos, los cuales son asignados a una propiedad del state apenas recibidos, harán que si la propiedad modificada está en ese momento mostrándose a través de una Vista, cambiará la información de dicha Vista en ese mismo instante.



REACTIVIDAD DEL STATE



Esto se debe a que, de igual manera que cualquier propiedad declarada en data, **las propiedades dentro de state son también reactivas.**

Tengamos presente estos manejos para el momento en el cual nos encontremos con una situación como la que comentamos, y no deseamos que el cambio se vea reflejado de manera inmediata.

VUEX y VUE DEVTOOLS

VUEX y VUE DEVTOOLS

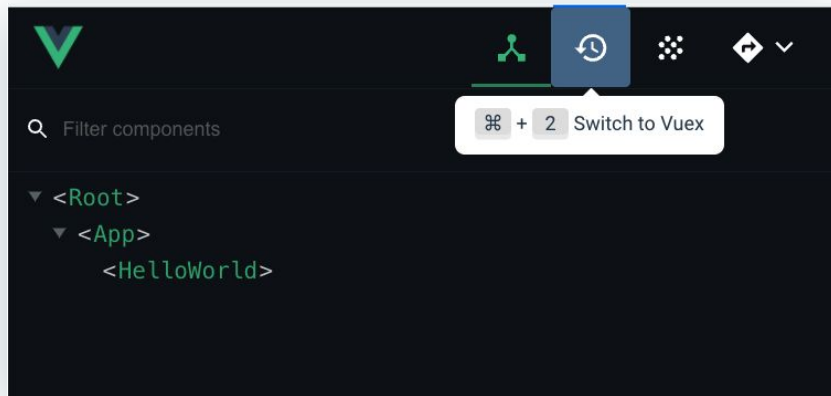


Vue Devtools cuenta con un apartado dedicado a la depuración de aplicaciones que integran Vuex.

Si aún no lo instalaste 😱, o lo quitaste por algún motivo, instalalo nuevamente desde la Tienda de aplicaciones de tu web browser. Está disponible para [Chrome](#), Edge y Firefox como extensión.

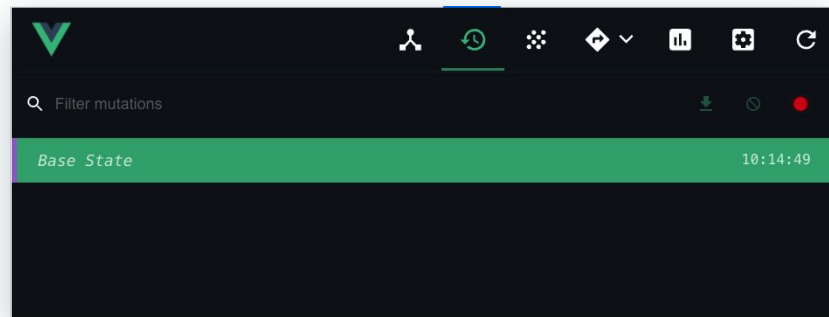


VUEX Y VUE DEVTOOLS



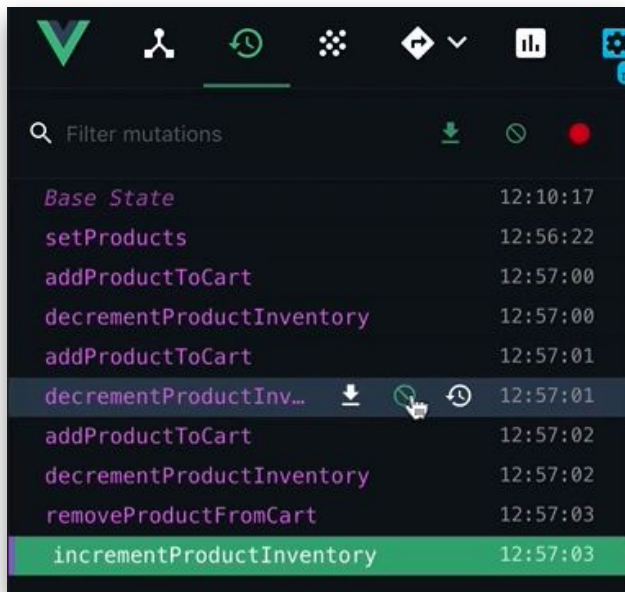
En el menú superior de Vue Devtools encontrarás un botón que permite cambiar a Vuex si la aplicación lo integra.

En la parte superior de esta sección, información sobre el state. A medida que se realizan operaciones sobre los datos, las mismas quedarán registradas en este timeline.





VUEX Y VUE DEVTOOLS



A medida que ejecutamos diferentes acciones sobre el **Store**, quedarán aquí registradas cada mutation invocada por cada operación.

¿Por qué mutations solamente?

Porque entre **actions**, **getters** y **mutations**, estas últimas son las únicas que se comunican con los **states**.



VUE DEVTOOLS: OPERACIONES

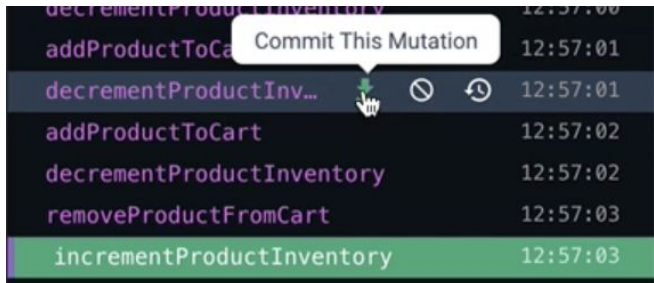
Además de poder registrar dentro de Vuex Devtools cada mutation, en el momento que estas se invocan, podemos también realizar una serie de operaciones sobre cada mutation ya ejecutada, directamente desde este apartado de Devtools.

Veamos, a continuación, cómo podemos sacar provecho de estas herramientas.



VUEX: *re-COMMIT*

Contamos con la posibilidad de re-ejecutar o confirmar una mutación específica.



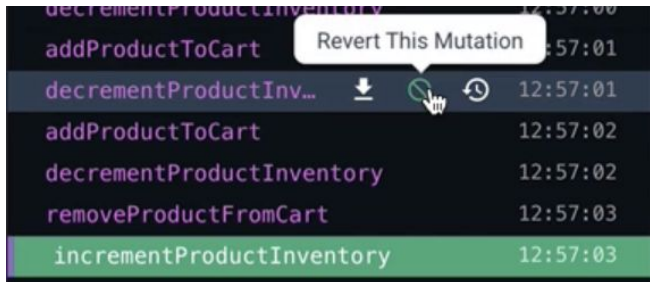
Esto lo realizamos ubicando y pulsando el botón **commit** ubicado a la izquierda del menú de operaciones de cada mutación. De esta forma volveremos el estado de la mutación a dicho momento, olvidando el resto de lo listado.



VUEX: ELIMINAR/REVERTIR

Una de las operaciones más efectivas que nos permite llevar adelante Vuex Devtools es eliminar o revertir una mutación determinada.

Solo debemos seleccionar desde DevTools la mutación en cuestión y ubicar y pulsar el botón central de la misma para revertir la operación realizada por dicha mutación.

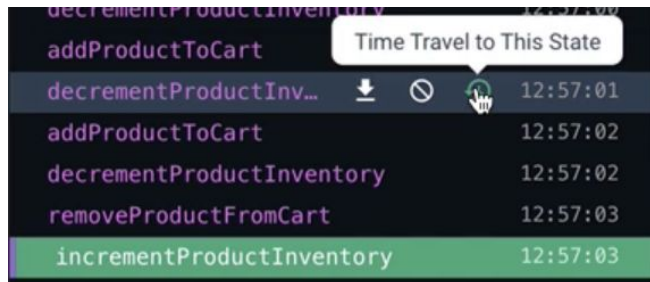




VUEX: TIME TRAVEL

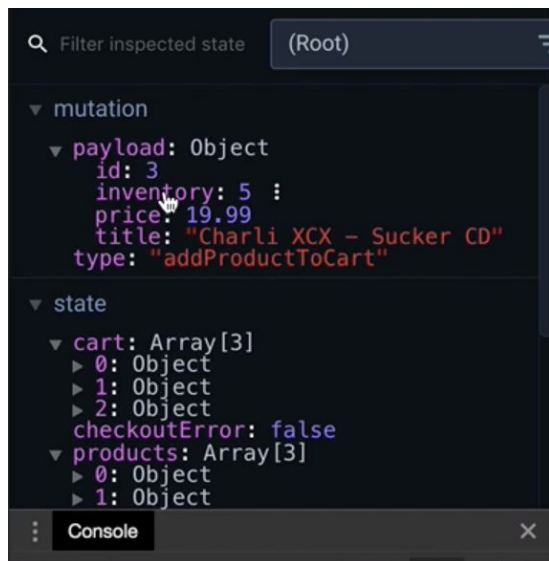
Y, por supuesto, la tercera opción nos permite ejecutar un “Viaje en el Tiempo”. De allí su nombre particular.

Esta acción nos permite desplazarnos en el tiempo hasta ese momento en la historia de operaciones ejecutadas. Debemos tomar como referencia la hora, minutos y segundos en el cual se realizó dicha operación.





VUEX: TIME TRAVEL

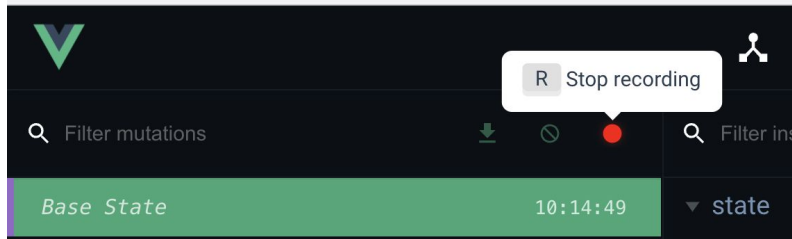


Al desplazarnos hasta un momento en particular, utilizando Time Travel, podemos visualizar en el detalle de dicha operación, la información puntual sobre **state**, el tipo de operación que realiza dicha mutación, y hasta el estado de **state**, **getters** y demás elementos.

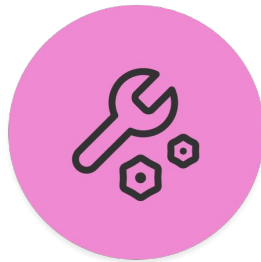
Toda esta información se encuentra anidada y puede ser desplegada e inspeccionada para ver qué había almacenado en ese preciso momento.



VUEX: DEVTOOLS



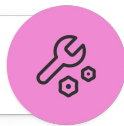
Recuerda que, para sacar provecho de **Time Travel** y demás operaciones realizadas por mutations, debes tener activo el botón **Recording** en el apartado superior de la pestaña **Switch to Vuex**.



VUEX DEVTOOLS

Conectar el store con Chrome Dev Tools.

Tiempo estimado: 15 minutos.



VUEX DEVTOOLS

Integra la herramienta Vue DevTools en tu proyecto actual y aplica la depuración sobre el mismo, realizando interacciones sobre la información que este manipula, e integra la importación/exportación de datos almacenados en state.

Tiempo estimado: 15 minutos.

ALMACENAMIENTO DE STATES

ALMACENAMIENTO DE STATES

Todo dato establecido en `store.state` puede ser configurado, no solo de forma predeterminada dentro de nuestra App, sino también a través de un archivo JSON.

Este archivo `JSON` puede ser accedido a través de una petición `fetch()` para obtener sus valores y luego aplicarlos en `state`.



ALMACENAMIENTO DE STATES

Podemos recurrir a **Axios** o al uso de **fetch()** para recuperar un set de elementos de array almacenados en un archivo JSON, y almacenarlos en un state predefinido.

Este proceso deberás tener en cuenta de realizarlo a través de una mutación.

```
export default new Vuex.Store({
  state: {
    ...
    ubicacion: 'house',
    productos: [],
  },
  mutations: {
    configurarProductos:
      (state, productos) => {
        state.productos = productos
      },
  })
  .....
const setProductosEnStore = () => {
  axios.get("json/products.json")
    .then(response => {
      store.commit(
        'setProductosEnStore',
        response.data.productos
      )
    })
}
setProductosEnStore()
```



ALMACENAMIENTO DE STATES

También, dada su función tan específica, sería muy buena práctica definir las mutaciones en Vuex capitalizando la misma.

```
mutations: {  
  CONFIGURAR_PRODUCTOS:  
    (state, productos) => {  
      state.productos = productos  
    }  
}
```

.....

```
const setProductosEnStore = () => {  
  axios.get(...)  
    .then(response => {  
      store.commit(  
        'CONFIGURAR_PRODUCTOS',  
        response.data.productos  
      )  
    })  
}
```

```
setProductosEnStore()
```


RECUPERACIÓN



RECUPERACIÓN

Otra alternativa válida para recuperar el contenido a utilizar en nuestra aplicación Vue desde un archivo JSON, sería importar directamente este archivo mediante `import...from`, y luego definirlo sobre la propiedad de state en cuestión.

```
import prods from './productos';

export default new Vuex.Store({
  state: {
    ...
    productos: prods
  }
})
```



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

ASINCRONISMO

ASINCRONISMO

Hasta ahora aplicamos las acciones (Actions) en diferentes operaciones, de forma 100% sincrónica.

Pero, en realidad, las acciones pueden contener, al menos, una **operación asincrónica** lo cual cambiaría el comportamiento predeterminado de nuestras aplicaciones Vue.

Veamos algunos ejemplos a continuación.



ASINCRONISMO

Recordando el comportamiento de las acciones, vimos que las mismas se disparan a través del método `dispatch()`.

Si bien podríamos llamar de forma directa a `store.commit('increment')` en lugar de `dispatch`, debemos tener presente que las mutaciones (`mutations`) tienen que ser sincrónicas pero las acciones (`Actions`), no.

```
store.dispatch('increment')
```

```
actions: {  
  incrementAsync ({ commit }) {  
    setTimeout(() => {  
      commit('increment')  
    }, 1000)  
  }  
}
```



ASINCRONISMO

Podemos simular una operación asincrónica a través de las Actions, utilizando la función JavaScript `setTimeout()`. De esta forma podemos evaluar cómo se comportaría nuestra aplicación Vue a través del asincronismo integrado en Vuex.

```
actions: {  
  incrementAsync ({ commit }) {  
    setTimeout(() => {  
      commit('increment')  
    }, 1000)  
  }  
}
```



ASINCRONISMO

Y cuando debemos involucrar asincronismo más complejo en un Store, una de las opciones más efectivas en este momento, es llamar a las operaciones complejas y con un asincronismo seguro, a través de acciones.

```
actions: {  
  carrito ({ commit, state }, productos) {  
    const itemsGuardados = [...state.cart.added]  
    commit(types.CHECKOUT_REQUEST)  
    shop.comprarProductos(  
      Productos,  
      () => commit(types.CHECKOUT_SUCCESS), // manejar una operación exitosa  
      () => commit(types.CHECKOUT_FAILURE, itemsGuardados) // manejar una falla  
    )  
  }  
}
```




ASINCRONISMO

Y, por supuesto, que integrar promesas para controlar acciones con asincronismo asegurado, es otra alternativa más que válida y útil. Sabemos que las promesas y el asincronismo en éstas, funciona de manera efectiva.

```
actions: {  
  actionA ({ commit }) {  
    return new Promise((resolve, reject) => {  
      setTimeout(() => {  
        commit('algunaMutation')  
        resolve()  
      }, 1850)  
    })  
  }  
}
```

MIXINS



MIXINS

Ya aprendimos que las mutaciones son el camino para cambiar un estado en Vuex Store. Y seguramente que, en proyectos medianos y grandes, la representación de este código tornará a duplicarse en casi todos los proyectos.

Por suerte, existe una vía para establecer una práctica de reutilización de mutaciones en Vuex.

```
mutations: {  
  setStaff (state, staffs) {  
    state.staffs = staffs  
  },  
  setRol (state, role) {  
    state.role = role  
  },  
  setPolíticas (state, policies) {  
    state.policies = policies  
  },  
  setSector (state, sector) {  
    state.sector = sector  
  }  
}
```



MIXINS

```
const mutations = {  
  mutate(state, payload) {  
    state[payload.property] = payload.with  
  }  
}
```

A través de una mutación simple, podemos definir como parámetros el `state` y su `payload`, para finalmente aplicar el state a cada propiedad de forma dinámica.

De esta otra forma, definimos una propiedad y su `payload`, de manera más simple, tal como un objeto, alcanzando el mismo resultado que el ejemplo anterior.

```
commit('mutate', {  
  property: <propertyNameHere>,  
  with: <valueGoesHere>  
})
```



MIXINS

Y el uso de `mixins` será la buena práctica ideal para poder propagar fácilmente una mutación estándar a través de todo el árbol de componentes que utilizan un mismo Store.

Para integrar mixins en Vuex, contamos con una dependencia que nos facilitará esta tarea: `Vuex Extensions`.

```
_> npm install vuex-extensions
```



MIXINS: VUEX EXTENSIONS

Instalada la dependencia, podemos importarla en primera instancia y, luego de ello, ya podemos sumar a nuestro Store global, el uso de `mixins: {}`.

```
import Vue from 'vue'
import Vuex from 'vuex'
import { createStore } from
'vuex-extensions'
Vue.use(Vuex)

export default
createStore(Vuex.Store, {
  plugins: [],
  modules: {
    moduloA,
    moduloB
  },
  mixins: {
    //Mixins disponibles! :)
  }
})
```



MIXINS

De esta forma ya podemos integrar mutaciones genéricas, que sean útiles evitando el código repetitivo, dentro del objeto mixins.

Así lograremos su disponibilidad global para cualquier de los componentes que importe este Store.

```
mixins: {  
  mutations: {  
    cambiarEstado: (state, changed) => {  
      Object.entries(changed)  
        .forEach(([name, value]) => {  
          state[name] = value  
        })  
    }  
  }  
}
```



VUEX EXTENSIONS

Además de soportar mixin mutations, Vuex Extensions brindan soporte de mixins con getters y mixins con actions.

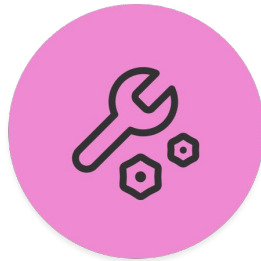
Esto nos permitirá escribir componentes web lo más simplificados posible, ahorrando tiempo de desarrollo además de implementar código 100% reutilizable.

Ejemplo
en vivo



¡VAMOS A PRACTICAR!

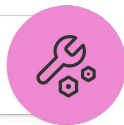
CODER HOUSE



ESTRUCTURAS COMPLEJAS EN VUEX

Comienza a sacar provecho de Vuex y la manipulación de estructuras de datos complejas.

Tiempo estimado: 10 minutos.



ESTRUCTURAS COMPLEJAS EN VUEX

Si ya tienes un proyecto encaminado con consumos de datos remotos o tal vez un JSON local que te brinda información para alimentar el Store, te proponemos que modifiques el mismo, estableciendo:

- Una operación asíncronica (lectura de datos vía fetch o axios).
- Definiendo módulos independientes para los datos de state.
- Integrando mixins globales con propiedades computadas.

Tiempo estimado: 10 minutos.



DESAFÍO DE VUE CDN A VUE CLI

Convierte tu desafío de la Clase 04 de Vue CDN a Vue Cli.

DESAFÍO DE FORMULARIOS CON VUE

Formato: Sube tu desafío complementario a la Plataforma de Coderhouse. Su nombre deberá ser “Formularios con Vue + Tu Apellido”.

Sugerencia: N/A.

Desafío
Complementario



>> Consigna: Recupera el desafío genérico realizado en la Clase 08, donde trabajamos con Formularios.

>>Aspectos a incluir en el entregable:

Aprovecha el backend que elaboramos en el portal Mockapi, y agrega un endpoint que permita recibir los datos del formulario. Además de esta simple operación, ten presente integrar Vuex en tu formulario.

En una segunda Vista del formulario, manejada a través de ruteos, deberás poder acceder a los datos de Mockapi, recuperando en el state del Store global dicha información para luego mostrarla en pantalla.

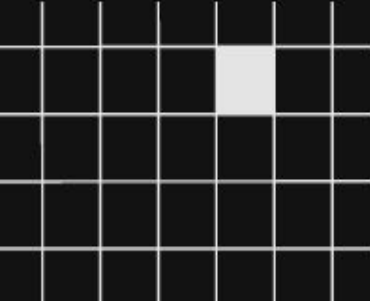
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- 
- Mixins en Vuex
 - Vuex en Vue Devtools
 - Módulos en Vuex
 - Asincronismo y Reactividad



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN

CODER HOUSE