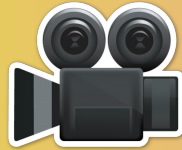




Clase 11. Vue JS

Introducción a Vuex

RECORDÁ PONER A GRABAR LA CLASE





¿DUDAS DEL ON-BOARDING?

MIRALO AQUI



OBJETIVOS DE LA CLASE

- Desarrollar el patrón Vuex e integrarlo en un proyecto frontend VueJS.

MAPA DE CONCEPTOS



CRONOGRAMA DEL CURSO

Clase 10



Consumo de API Rest



PETICIÓN REMOTA VÍA
FETCH



PETICIONES REMOTAS VÍA
AXIOS API



EJEMPLO EN VIVO



ENTREGA INTERMEDIA

Clase 11



Introducción a Vuex



VUEX EN VUE CLI 2



EJEMPLO EN VIVO



ESTRUCTURA VUEX

Clase 12



Vuex en Vue



ESTRUCTURA COMPLEJA
VUEX



EJEMPLO EN VIVO



CHROME DEV TOOLS



VUE CDN A VUE CLI

PATRÓN DE ESTADO GLOBAL

¿A qué nos referimos?

CONCEPTO DE PATRÓN DE ESTADO GLOBAL

Cuando hablamos de Estados de una aplicación nos referimos al conjunto de variables y constantes que conforman la base de la misma.

En sí, un Estado hace referencia a todas las formas posibles en las que una App Vue puede encontrarse en un determinado momento de su ciclo.

CONCEPTO DE PATRÓN DE ESTADO GLOBAL

La forma en la cual gestionamos y estructuramos el Estado de nuestra aplicación es clave para poder evitar bugs innecesarios.

Por ello, es clave saber cómo, cuándo, dónde y porqué un Estado en particular realiza una mutación.

Para eso existe...¡Vuex!



CONCEPTO DE PATRÓN DE ESTADO GLOBAL

Para poder controlar todo esto desde una App Vue, existe **Vuex**.

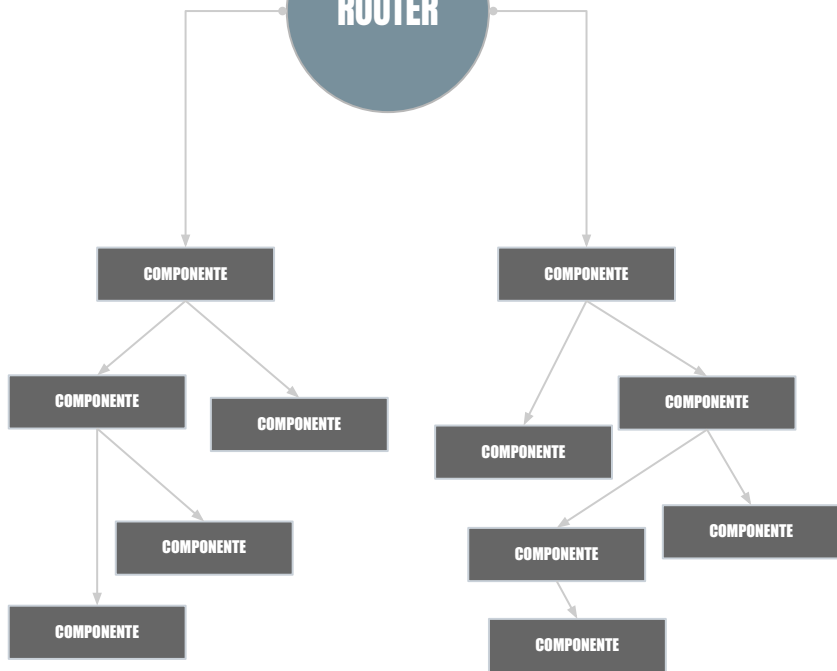
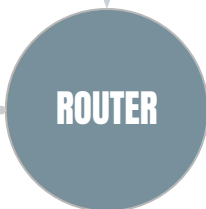
Este permite gestionar el Estado de las aplicaciones Vue.js, funcionando como un almacén que centraliza la información para todos los componentes que conforman una aplicación Vue e integra un set de reglas que garantizan que el Estado pueda cambiarse de forma predecible.

CONCEPTO DE PATRÓN DE ESTADO GLOBAL



Vuex cuenta con un soporte e integración completa con la **Extensión Devtools** oficial de Vue, lo cual permite adicionar a ésta una serie de funciones avanzadas, integrables al momento de depurar una aplicación y ante la necesidad de capturar Snapshots o instantáneas de Estado.

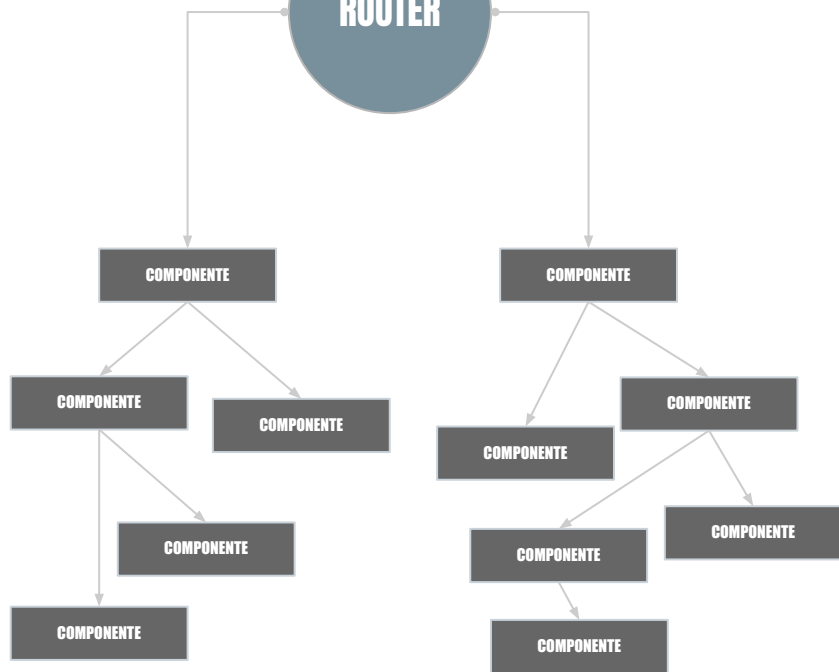
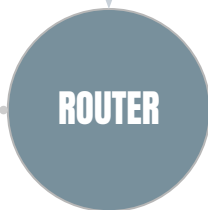
RESOLVIENDO LOS ESTADOS



RESOLVIENDO LOS ESTADOS

En el gráfico encontramos la arquitectura de cómo se estructura una App Vue.

Con la introducción de un **Router**, la distribución de componentes entre diferentes ruteos le da un tinte algo más complejo a la ecuación de una App.



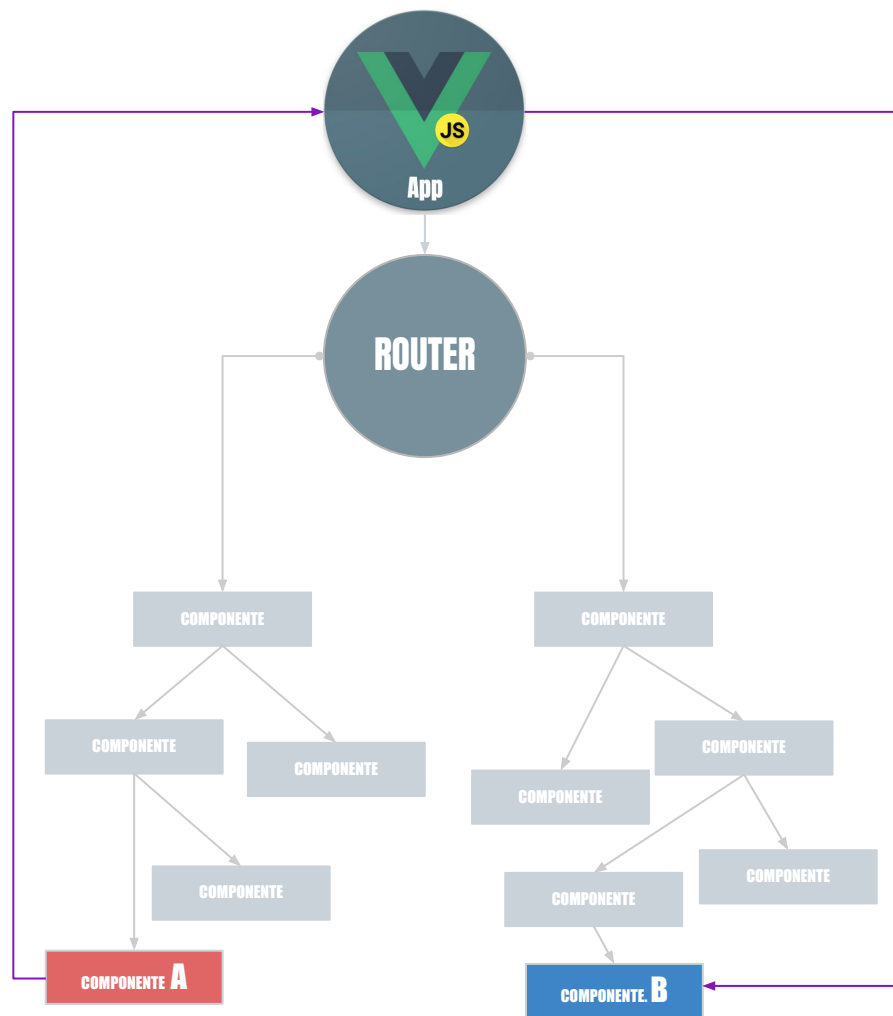
RESOLVIENDO LOS ESTADOS

A esto le sumamos que, en determinadas oportunidades, la información fluye en tantas direcciones que termina volviendo más complejo saber en dónde se encuentra la fuente de información actualizada o desde dónde debemos obtener la información a mostrar, entre otras cosas.

RESOLVIENDO LOS ESTADOS

Veamos el siguiente ejemplo en el gráfico:

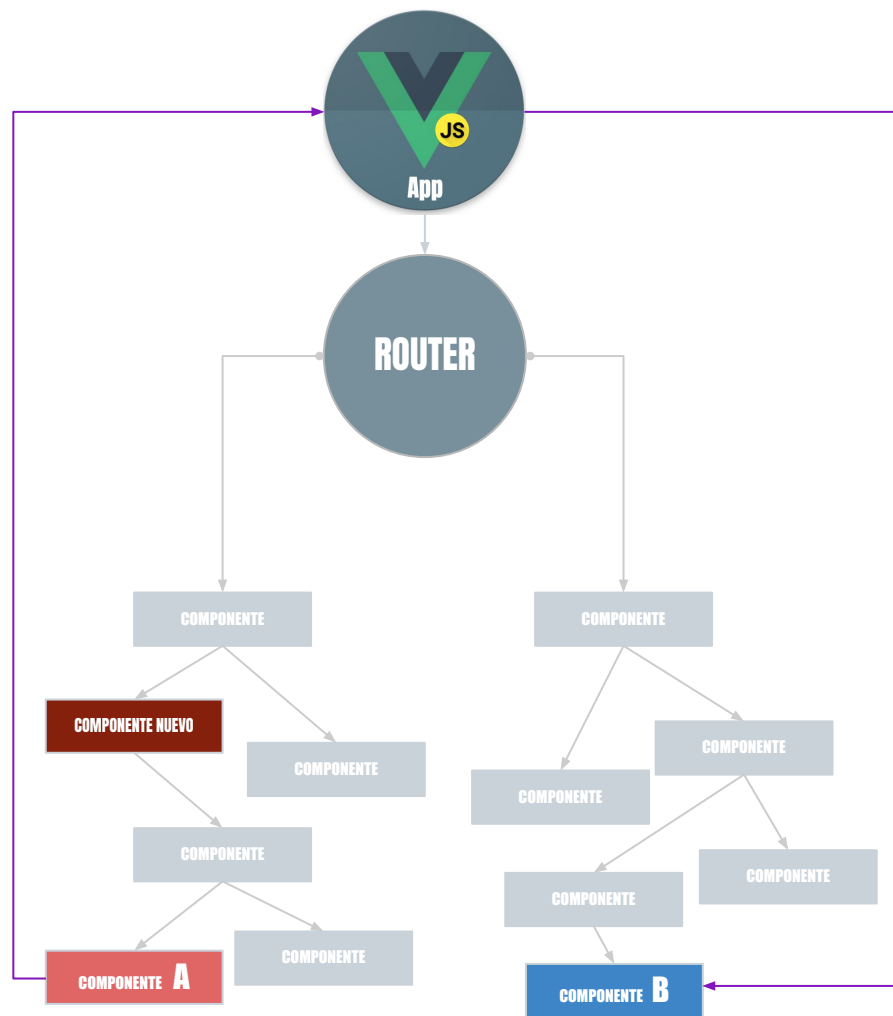
Si el **componente A** necesita comunicarle información al **componente B**, tendremos que trasladar la misma vía **props** desde componente A, como hijo, a su padre, y este otro a su otro padre, pasando por el **Router**, para finalmente bajar la información de componentes padres a sus hijos hasta llegar al componente B.



RESOLVIENDO LOS ESTADOS

Si bien esto es factible de hacer, tenemos que evaluar que, tal vez, el día de mañana surge la necesidad de sumar un nuevo componente a alguno de los árboles.

Esto hará ese traspaso de información algo más complejo, además de que puede aparecer alguna traba en la arquitectura de la aplicación, que ahora podemos no estar dimensionándola correctamente.

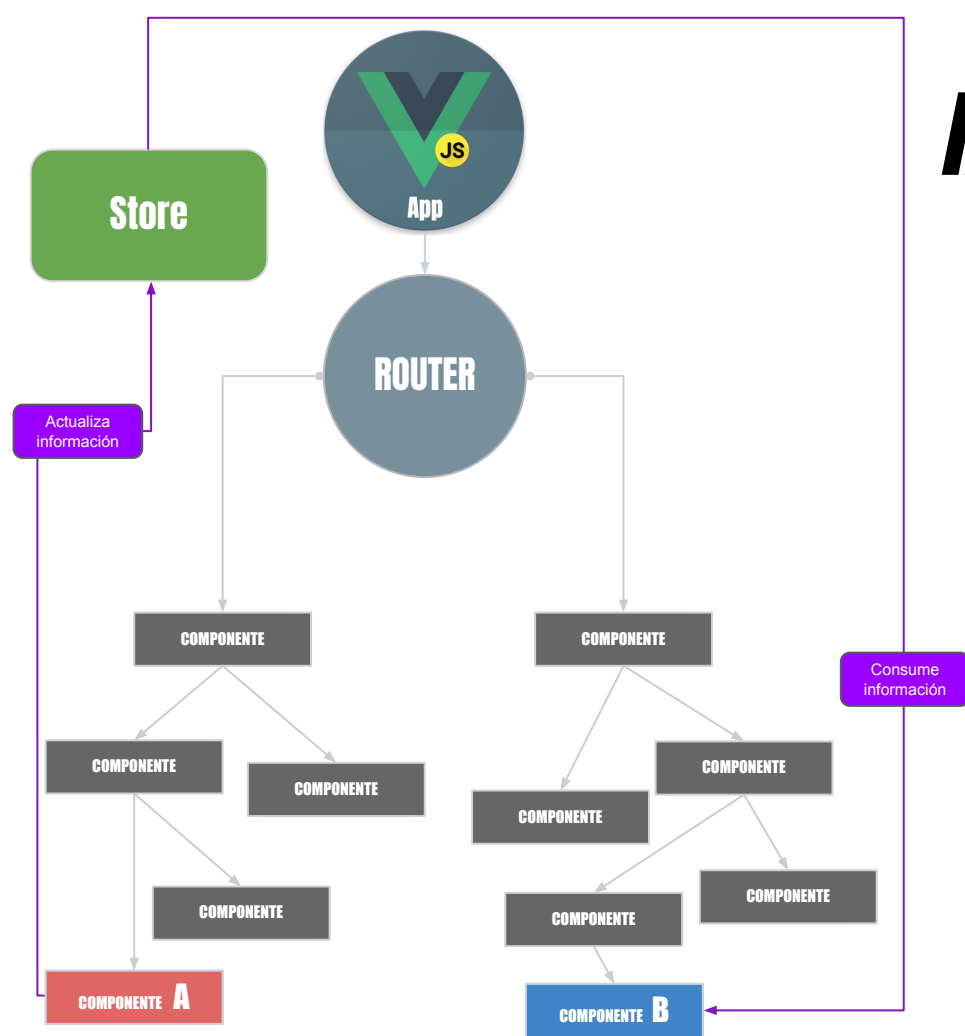


RESOLVIENDO LOS ESTADOS

Aquí es donde Vuex aporta la inteligencia. 💪

Su idea es que contemos con un Store dentro de nuestra aplicación Vue/Cli para que cada uno de nuestros componentes que necesite consumir información vaya a buscarla allí.

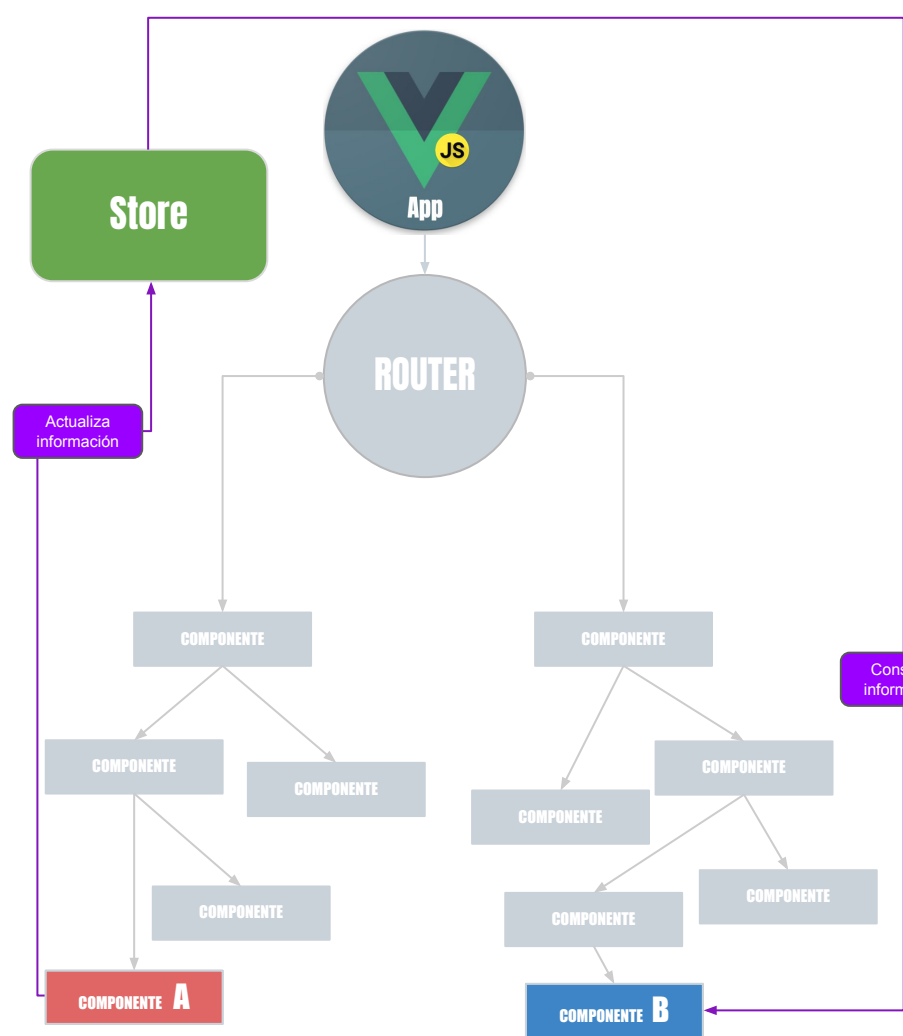
De igual forma para los componentes que necesiten actualizar información. Al hacerlo, lo harán directamente en el Store.



RESOLVIENDO LOS ESTADOS

De esta forma, cuando el **Store** se actualiza, el componente B, que está escuchando por cambios en el Store, va a enterarse de los mismos y consumirá dicha información una vez que esté actualizada.

Esto último tiene algo más de complejidad en su procesos de lo que aquí relatamos pero... ¡tranquilos!, los veremos en detalle a lo largo de estas clases.






RESOLVIENDO LOS CAMBIOS

Para que el **Store** funcione como nosotros esperamos, debemos de realizar ciertas configuraciones que permitan que esto ocurra.

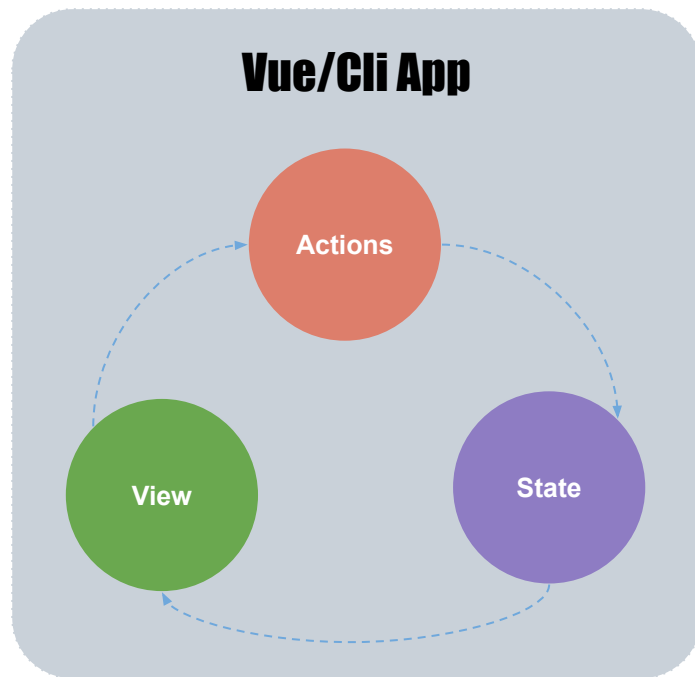
El Store en sí cuenta con un **state** “*para que toda la magia ocurra*”. Dicho state es reactivo y muchos de los conceptos que aprendimos hasta ahora en Vue son completamente aplicables casi de forma transparente al Store, lo que de alguna forma hará que trabajar con **Vuex** no sea tan rebuscado.

ACCIONES

ACCIONES

Si abstraemos y graficamos una pequeña porción de una App Vue, tendremos algo similar al siguiente gráfico 

La **Vista** originalmente es quien dispara acciones en la aplicación. Las **acciones** se ocupan de procesar, calcular, y modificar datos que luego son almacenados en **state** quien, a su vez, al ser reactivo, le notifica a la Vista sobre el cambio, repitiendo este ciclo de forma interactiva.



ACCIONES



Y dentro de las estrategias que propone Vue para resolver la problemática en el primer gráfico, implementa la comunicación entre componentes usando **bus de datos** interno.

Creando una nueva instancia de la clase Vue, lograremos trasladar información entre dos componentes sin parentesco, que incluso pueden formar parte de diferentes Vistas.

ACCIONES

Creamos una instancia de Vue, que cuenta con el método `$emit`, para emitir un evento en cuestión y, por otro lado, un método `$on` para registrarme y escuchar dicho evento.

El `componente B` registra el evento `incrementar`, cuando este ha sido creado, y espera a que el `componente A` emita cambios al ejecutar el método `accionArealizar()`.

```
const componenteA = {
  methods: {
    accionArealizar() {
      bus.$emit('incrementar', 1)
    }
  }
}

const componenteB = {
  data() {
    return {
      contador: 0
    }
  },
  evento() {
    bus.$on('incrementar', (numero) => {
      this.contador += numero
    })
  }
}
```

ACCIONES

Como podemos apreciar, la forma de comunicación entre ellos es similar a la cual un componente hijo dialoga con un componente padre excepto que, en este ejemplo, no existe parentesco alguno entre **componente A** y **componente B**.


```
const componenteA = {
  methods: {
    accionArealizar() {
      bus.$emit('incrementar', 1)
    }
  }
}

const componenteB = {
  data() {
    return {
      contador: 0
    }
  },
  evento() {
    bus.$on('incrementar', (numero) => {
      this.contador += numero
    })
  }
}
```


ACCIONES

La modalidad, anteriormente representada, funcionará correctamente en aplicaciones pequeñas o casos aislados pero, cuando la aplicación crece, esta propuesta se vuelve imposible de mantener, dificultando así las labores de reutilizar acciones (**actions**) y compartir estados (**states**) comunes.

ACCIONES

Entonces... ¿cómo debemos hacer para solucionar este aspecto en aplicaciones de media y gran escala? 

¿Qué debo tener en cuenta para no caer en esta práctica, la cual es buena pero limitada?

¡Allí es donde Vuex aporta su granito de arena, a través de la **Arquitectura del Patrón Vuex!** 

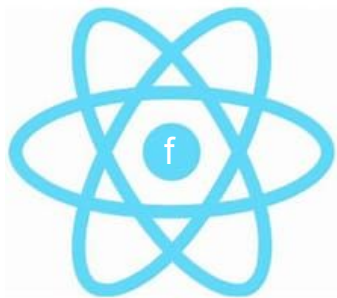
Veámoslo a continuación.



ARQUITECTURA DEL PATRÓN VUEX

ENTONCES... QUÉ ES VUEX

Vuex es una implementación del Patrón de Diseño, creado oportunamente por Facebook, llamado **Flux**.



Por si no tienes experiencia en un ecosistema más allá de JavaScript, Facebook es el desarrollador de la librería **React JS**, como también de Flux. Ambos fueron creados con el fin de encontrar una mejor solución al desarrollo de la plataforma Facebook, la cual estaba limitada a **HTML5 + JS** y **PHP + MySQL** como motor backend. Algo que para interacciones masivas que esta plataforma solía tener su performance general se tornaba escasa.

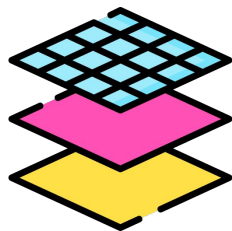
ENTONCES... QUÉ ES VUEX

En aplicaciones de mediana y gran escala, cuando llega el momento de compartir estados comunes entre sus componentes, la comunicación se torna difícil dentro del ecosistema en sí a través de los métodos y acciones.

ENTONCES... QUÉ ES VUEX

Estos últimos comienzan a repetirse o asemejarse en funcionalidad, lo cual conlleva a tener problemas de trazabilidad de los estados.

Entonces, esta problemática se transforma en una necesidad de plantearse cuál es la mejor forma de utilizar estas arquitecturas.



ENTONCES... QUÉ ES VUEX



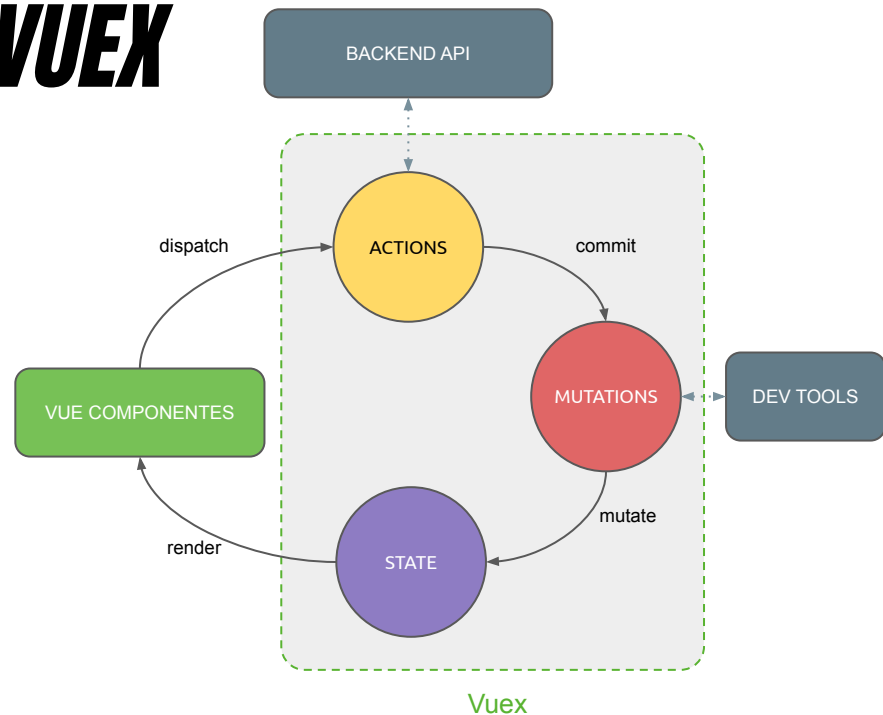
La comunidad de desarrolladores Facebook/React supo solventar esta problemática, a través de **flux**. Entonces, Vue optó por desarrollar una especificación similar a este Patrón, bautizada Vuex, la cual se acopla muy bien a la filosofía de este último framework.

👉 Recordemos que Vue mantiene su **Core** lo más ligero posible, por lo tanto Vuex debe acoplarse de forma externa y sólo en los casos donde se considere necesario contemplar este patrón para un escalamiento de arquitectura.

ARQUITECTURA DEL PATRÓN VUEX

Representación gráfica de la Arquitectura
Vuex.

A través de la misma podemos denotar un flujo unidireccional, el cual aporta una capa de claridad para entender bien qué es lo que ocurre dentro del ecosistema de la aplicación mientras sus elementos interactúan entre sí como también con actores externos.



ELEMENTOS DEL PATRÓN VUEX

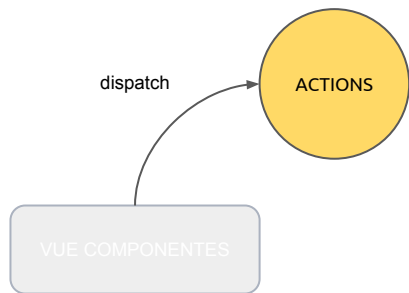
VUE COMPONENTES

- La Arquitectura de componentes se representan como la estructura de un edificio el cual espera ser habitado por los estados de la aplicación

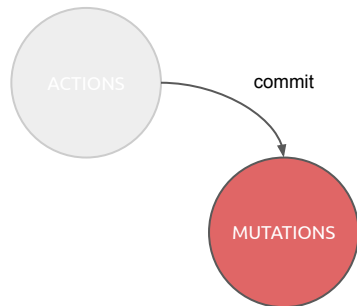
STATE

- Los estados que los componentes usan pasan a gestionarse por Vuex y vincularse a los componentes por medio de observadores (**watchers**).
Vuex se adecua al sistema reactivo de la plataforma y cuando un estado interno de su sistema de almacenamiento muta, y este, a su vez, se vincula con un componente, activa el renderizado del componente con el nuevo estado ya mutado

ELEMENTOS DEL PATRÓN VUEX



- Los componentes son capaces de lanzar acciones. En este momento y lugar se puede gestionar parte de la lógica más próxima a los datos de la aplicación. Las acciones permiten gestionar asincronía, por lo cual, son el lugar idóneo para invocar servidores externos (**backend APIs**)



- Cuando una acción finaliza sus tareas, se realizan las confirmaciones, o (**commits**), contra el estado. Éstas ejecutan métodos especializados en la mutación de cambios (**mutations**). En ese instante se desencadenan cambios de estado y se renderiza el HTML

COMUNICAR COMPONENTES EN APLICACIONES DE ESCALA MEDIA



COMUNICAR COMPONENTES

En este bloque de código, traemos una representación de las buenas prácticas aplicadas a la construcción de un espacio centralizado donde compartir métodos y estados.

Este modelo puede aplicar para aplicaciones de mediana escala, de forma efectiva y sin complicaciones extra a futuro.

```
const store = {
  debug: true,
  state: {
    mensaje: "Hola coders!"
  },
  setAccionDelMensaje (newValue) {
    this.debug
    console.log('Método disparado con ', newValue)
    this.state.mensaje = newValue
  },
  limpiarAccioneDelMensaje () {
    This.debug
    console.log('Método disparado')
    this.state.mensaje = ''
  }
}

const vmA = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})

const vmB = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})
```



COMUNICAR COMPONENTES

De esta forma, creamos un objeto el cual contiene almacenado el estado a compartir:

```
state: {mensaje: ...}
```

Además, cuenta con unos métodos que son los encargados de realizar el proceso de mutación de dicho estado. De esta forma quedan centralizados los estados y sus acciones.

```
const store = {
  debug: true,
  state: {
    mensaje: "Hola coders!"
  },
  setAccionDelMensaje (newValue) {
    this.debug
    console.log('Método disparado con ', newValue)
    this.state.mensaje = newValue
  },
  limpiarAccioneDelMensaje () {
    This.debug
    console.log('Método disparado')
    this.state.mensaje = ''
  }
}
```

```
const vmA = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})
```

```
const vmB = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})
```




COMUNICAR COMPONENTES

Con el objeto anterior definido, estamos en condiciones de crear instancias de componentes, las cuales pueden compartir dicho estado.

Si alguno de estos componentes muta el estado, el cual está compartido, usará para ello el método definido en el store. Dicho método, al estar referenciado en todos los componentes, hará que el cambio del **state** se refleje masivamente.

```
const store = {
  debug: true,
  state: {
    mensaje: "Hola coders!"
  },
  setAccionDelMensaje (newValue) {
    this.debug
    console.log('Método disparado con ', newValue)
    this.state.mensaje = newValue
  },
  limpiarAccioneDelMensaje () {
    This.debug
    console.log('Método disparado')
    this.state.mensaje = ''
  }
}

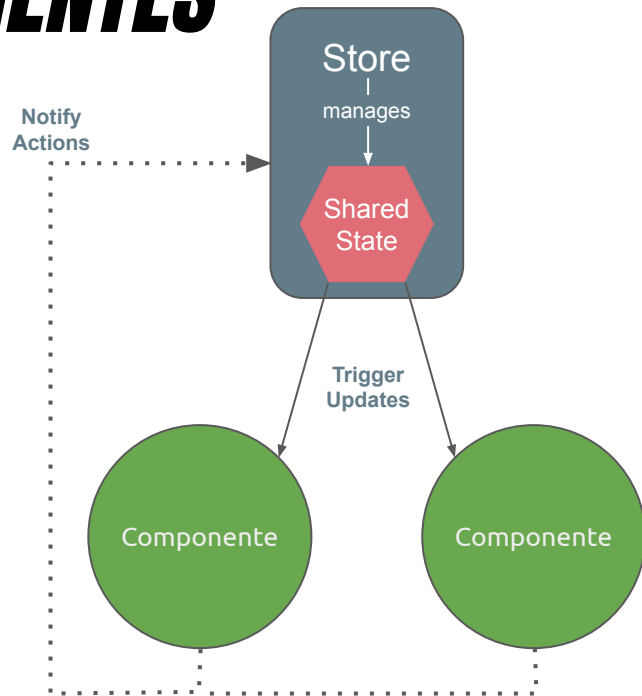
const vmA = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})

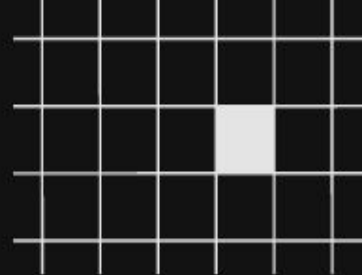
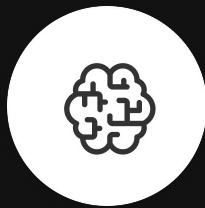
const vmB = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})
```

COMUNICAR COMPONENTES

El código anteriormente visto desencadena una representación gráfica como la que aquí podemos ver 🖱️

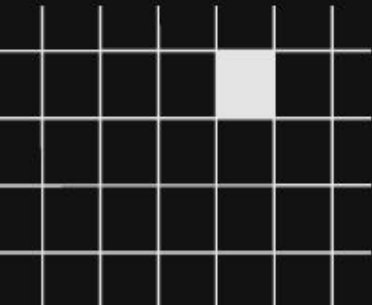
Este mecanismo funcionará correctamente pero, cuando contamos con aplicaciones de escala superior en el cual modularizar un Store también sea necesario, deberemos recurrir a una librería aún más potente, la cual deberá limitar la manipulación externa de estados. Allí es donde, alternativas como Vuex entran en juego.





¡PARA PENSAR!

*¿Conoces las arquitecturas Flux, Redux o Elm? o, al menos,
¿las sentiste nombrar alguna vez?*



CONTESTA POR EL CHAT DE ZOOM

Ejemplo
en vivo



¡VAMOS A PRACTICAR!

CODER HOUSE

INTEGRACIÓN DE VUEX EN VUE/CLI 2



VUEX EN VUE/CLI 2

Para implementar **Vuex** en nuestros proyectos Vue/Cli, lo primero que debemos hacer es instalar su dependencia en nuestro entorno de desarrollo.

→ Ejecutamos para esto, en la Terminal, el siguiente comando npm:

```
_> npm install vuex --save
```



VUEX EN VUE/CLI 2

→ Al crear un proyecto Vue/Cli 2 incluyendo Vuex, elegimos las siguientes opciones del menú:

- ☐ Manually select features
- ☐ Vuex
- ☐ Vue 2.x
- ☐ Eslint with error prevention only
- ☐ Lint on save
- ☐ In dedicated config files



>  node_modules

>  public

✓  src

>  assets

>  components

✓  store

 index.js

 App.vue

 main.js

 .browserslistrc

 .eslintrc.js

 .gitignore

 babel.config.js

 package-lock.json

 package.json

 README.md



VUEX EN VUE/CLI 2

→ Finalizado el proceso de instalación de nuestro proyecto integrando **Vuex**, podemos verificar en la rama del mismo los cambios que llegan de la mano de esta nueva configuración.

En primera instancia, encontraremos una carpeta denominada **./store/**, como subcarpeta de **/src/**, donde se establecen los archivos de nuestro proyecto.



VUEX EN VUE/CLI 2

Dentro de la subcarpeta `./store/`, encontramos el archivo `index.js`. El mismo, establece la configuración base para que funcione Vuex, importando esta librería desde `node_modules` y ejecutando la instancia `Vue.use()` para inicializarlo.

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

Vue.use(Vuex)

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



VUEX EN VUE/CLI 2

→ Finalmente, en el archivo `main.js`, encontramos referenciado el `store` desde la subcarpeta donde se creó, y su inicialización dentro de la instancia del objeto `new Vue({})`.

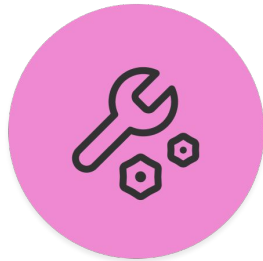
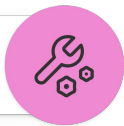
Con esto, ya tenemos integrado Vuex en una aplicación Vue/Cli.

Veamos, a continuación, cómo sacarle provecho.

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

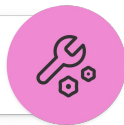
Vue.config.productionTip = false

new Vue({
  store,
  render: h => h(App)
}).$mount('#app')
```



VUEX en VUE CLI 2

Integra la estructura Vuex en un nuevo proyecto Vue/Cli 2.



APRENDE A SUMAR VUEX A TUS PROYECTOS VUE/CLI

Inicia un nuevo proyecto Vue/Cli 2 personalizado integrando Vuex, tal como vimos en este último tramo de la clase.

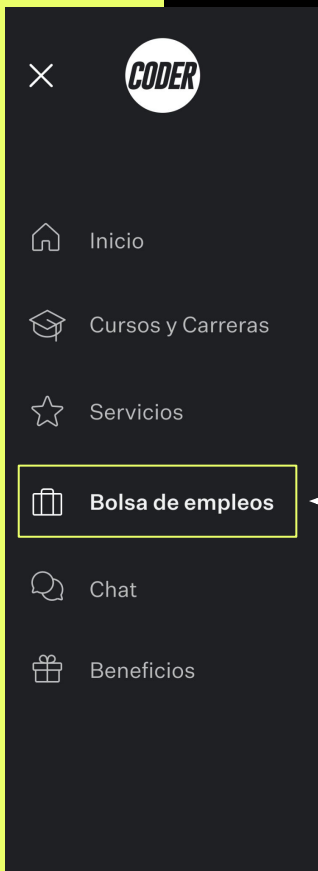
Ten presente los pasos que detallamos durante la configuración inicial del proyecto, lo cual te permitirá armar la estructura base de Vuex y su Store global que permitirá manejar la información de todos los componentes de tu proyecto Vue.

Tiempo estimado: 10 minutos.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!



Nuevo

¡Lanzamos la Bolsa de Empleos!

Un espacio para seguir **potenciando tu carrera** y que tengas más **oportunidades de inserción laboral**.

Podrás encontrar la **Bolsa de Empleos** en el menú izquierdo de la plataforma.

Te invitamos a conocerla y ¡postularte a tu futuro trabajo!

Conócela

CREACIÓN DEL STORE EN VUEX

CREACIÓN DEL STORE EN VUEX

→ Tenemos el proyecto ya construido y Vuex

integrado. Analizamos cada instancia de Vuex y las referencias a este para sacar provecho del mismo y ubicamos la declaración del código base del Store.

Solo nos queda entender cómo se estructura ahora la aplicación para poder preparar un proyecto 100% escalable.

Sumerjámonos entonces en el detalle del mismo.



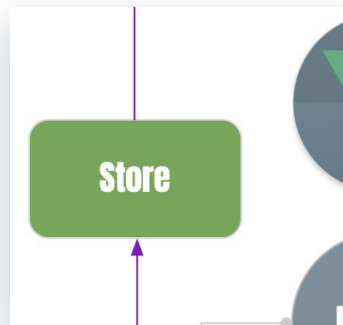


CREACIÓN DEL STORE EN VUEX

```
Vuex.Store({  
  state: {  
  },  
  mutations: {  
  },  
  actions: {  
  },  
  modules: {  
  }  
})
```

Volviendo al archivo `index.js`, encontramos la estructura base de `Vuex.Store()`, donde se definen los objetos que centralizarán estados, mutaciones, acciones y otros módulos comunes a todas las Vistas.

Aquí es donde se plasma la Visión presentada en los primeros slides de esta clase.





CREACIÓN DEL STORE EN VUEX

El Store integrado a nuestro proyecto funcionará como un árbol de estados simples dado que, desde un único objeto, gestionaremos todos los datos de la aplicación, estableciendo un hub o concentrador para toda la información que en este debe fluir.

De esta forma, será mucho más fácil realizar la depuración de aplicaciones, evitando tener que trackear un dato a lo largo de toda una App.

STATE



STORE EN VUEX: STATE

Aquí tenemos una comparativa de una **instancia de Vue** ➡, respecto a la **instancia que propone Vuex** ➡.

El modelo de datos el cual, en una aplicación Vue se maneja a través de **data**, en Vuex pasa a estar controlado por el objeto **state** (estados), como modelo global de toda la aplicación.

```
export default new Vuex.Store({  
  state: {  
    msg: ...  
  },  
  mutations: {  
  },  
  actions: {  
  },  
  modules: {  
  }  
})
```

```
const app = new Vue({  
  data: {  
    ...  
  },  
  methods: {  
    ...  
  },  
  computed: {  
    ...  
  }  
})...
```



STORE EN VUEX: STATE

Dentro del objeto `state` de `Vuex.Store()` comenzamos a declarar los elementos que necesitamos que estén disponibles para todo el ecosistema de nuestra aplicación Vue/Cli.

Tal como muestra el ejemplo, ya tenemos un objeto que puede ser accedido desde todos los componentes web.

```
export default new Vuex.Store({  
  state: {  
    msg: 'Variable de State  
declarada en Store'  
  },  
  mutations: {  
  },  
  actions: {  
  },  
  modules: {  
  }  
})
```



STORE EN VUEX: STATE

Para ver cómo se disponibiliza la propiedad declarada en `state`, aprovechemos el código generado mediante Vue/Cli, y editemos el archivo `HelloWorld.vue`.

Elimina la declaración de `props` como también el contenido HTML, dejando solo la declaración en `<template>` tal como vemos en el código contiguo.

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>
```



STORE EN VUEX: STATE

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

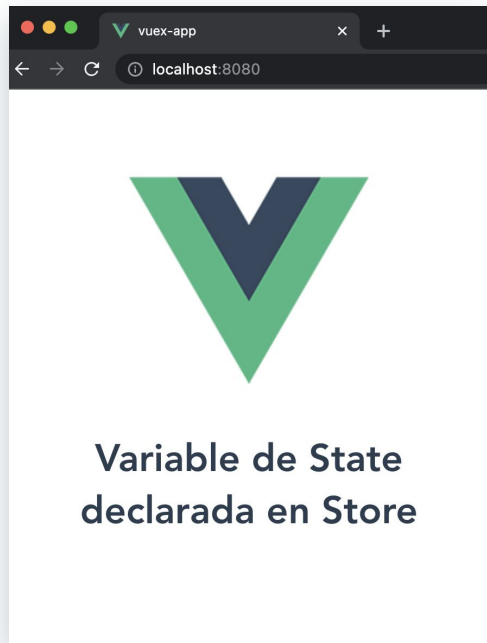
-----

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

👉 Si agregamos un texto en la propiedad `msg` del apartado `state`, tal como muestra este ejemplo...

👉 Y modificamos la propiedad `msg` dentro del `template`, referenciando la ruta completa hacia la propiedad `msg` declarada en el store...

STORE EN VUEX: STATE



...podrás verificar en tu aplicación que dicho cambio se realiza sin problema alguno.

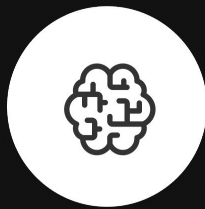
De esta forma, podemos corroborar cómo cualquier propiedad declarada dentro de **State**, estará disponible para cualquier Vista o componente web de nuestra aplicación Vue/Cli, gracias a **Vuex**.



STORE EN VUEX: STATE

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

Para llegar a msg, invocamos a la instancia global `$store`, luego a su objeto `state`, para finalmente alcanzar a leer la propiedad `msg`.



¡PARA PENSAR!

El objeto `$store` utilizado en el ejemplo para acceder al árbol de estados de Vuex, forma parte de las propiedades internas de Vue.

¿VERDADERO O FALSO?
CONTESTA LA ENCUESTA DE ZOOM





STORE EN VUEX: STATE

Si, de repente, vino a tu mente la clase no. 7
donde vimos propiedades internas y sus
similitudes con `$store`, ten presente que este
último no figura dentro del objeto global `this`
porque no forma parte del Core de Vue.

Recuerda que tuvimos que instalarlo por
separado. 😊

PROPIEDADES INTERNAS

- `$el`
- `$props`
- `$data`
- `$options`
- `$slots`
- `$refs`
- `$attrs`

Si hacemos una analogía entre los nombres y lo
que aprendimos hasta el momento con Vue,
veremos que cada propiedad interna referencia
directamente a una sección u objeto específico de
este Framework.

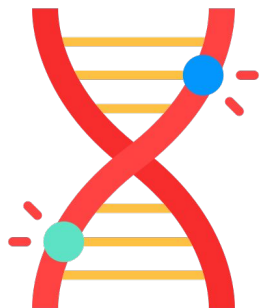
¡Analicemos cada una de ellas!

CODER HOUSE

MUTATIONS



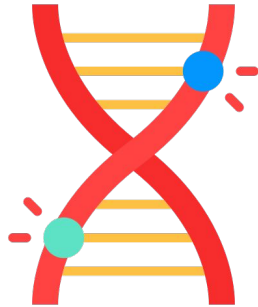
MUTATIONS 1/2



De acuerdo a la teoría vista algunas clases atrás, las mutaciones (**mutations**) se declaran exclusivamente para hacer cambios sobre los datos de los estados (**states**) de una aplicación.

Y, para que esto ocurra, debemos utilizar una acción (**actions**) que será quien invoque a la mutación en sí.

MUTATIONS 2/2



Por lo tanto, las mutaciones, quienes tienen acceso a los estados y datos almacenados en estos, **nos permitirán tener un flujo unidireccional sobre dicha información.**



MUTATIONS

```
//index.js
state: {
  msg: 'Variable de State declarada en
      Store',
  nombreDelCurso: 'jQuery a fondo!'
},
```

Supongamos que existe en `state`, una propiedad como la que 🖐️ aquí vemos, a la cual necesitamos cambiarle su valor.

Deberíamos para ello, declarar un método dentro del objeto `mutations`, para así poder llevar a cabo dicho cambio, de manera controlada.

```
mutations: {
  cambiarCurso: (state)=> {
    state.nombreDelCurso = 'Vue y Vuex'
  }
},
```



MUTATIONS

```
<h2>{{ $store.state.nombreDelCurso }}</h2>
```

Tenemos a ésta también definida en la Vista del componente, a través de toda la ruta específica, la cual nos permite llegar a la propiedad almacenada en `state`.





MUTATIONS

```
mutations: {  
  ...  
},  
incrementar: (state)=> {  
  state.valor++  
},
```

Si tenemos una propiedad en **state**, cuyo valor es numérico, y necesitamos incrementarlo, procederemos de la misma manera declarando la mutación.

Con estos ajustes dentro de **mutations**, ya tenemos el terreno preparado para ejecutar la mutación a través de un método contenido en **actions**.

ACTIONS



STORE EN VUEX: ACTIONS

```
const app = new Vue({  
  ...  
  methods: {  
    ...  
  }  
  -----  
  -  
  //En Vuex Store  
  export default new  
  Vuex.Store({  
    ...  
    actions: {  
      }  
    }  
  })  
})
```

De igual forma que vimos en **state** y **mutations**, los métodos (**methods**) utilizados en una aplicación Vue/Cli, pasan a ser controlados por las acciones (**Actions**), bajo el paraguas de Vuex.



STORE EN VUEX: ACTIONS

```
actions: {  
  cambiarNombreDelCurso: (context)=>  
  {  
    ...  
  }  
},
```

En Actions, debemos definir el método que se ocupará de aplicar al cambio que deseamos llevar a cabo.

El mismo debe recibir un parámetro, denominado comúnmente **context**, que hace referencia al contexto general de la aplicación.



STORE EN VUEX: ACTIONS

```
actions: {  
  cambiarNombreDelCurso: (context)=>  
  {  
    context.commit('cambiarCurso')  
  }  
},
```

`context` nos provee una serie de métodos para poder llevar un control sobre los cambios que realizaremos sobre el `state`.

`commit()` es el que debemos utilizar, pasándole como parámetro la mutación que deseamos ejecutar.

STORE EN VUEX: ACTIONS

A través de `context`, el manejador (`handler`), puede acceder a una serie de funcionalidades de este objeto, que exponen las siguientes propiedades:

HANDLER	DESCRIPCIÓN
<code>state</code>	Similar a <code>store.state</code> , o <code>state</code> local en los módulos JS.
<code>rootState</code>	Similar a <code>store.state</code> , sólo en módulos JS.
<code>commit</code>	Similar a <code>store.commit</code> , aportado por <code>context</code> .
<code>dispatch</code>	Similar a <code>store.dispatch</code> .
<code>getters</code>	Similar a <code>store.getters</code> , o <code>getters</code> locales si estamos en módulos JS.
<code>rootGetters</code>	Similar a <code>store.getters</code> sólo en módulos JS.



STORE EN VUEX: ACTIONS

Nos queda entonces invocar al método `dispatch`,
el cual se ocupa de invocar a la acción que se
ocupa de llevar a cabo la ejecución de la mutación
de dicho `state`.

```
methods: {  
  cambioNombre: () => {  
    this.$store.dispatch('cambiarNombreDelCurso')  
  }  
}
```

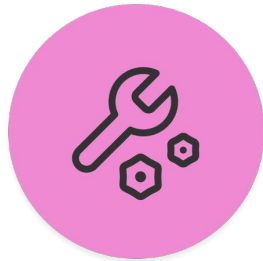


Ejemplo
en vivo



¡VAMOS A PRACTICAR!

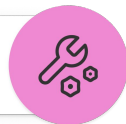
CODER HOUSE



ESTRUCTURA VUEX

Crear la estructura de Vuex con información global para ser manejada por los todos los componentes de un proyecto VueJS.

Tiempo estimado: 15 minutos



ESTRUCTURA VUEX

Dentro de un proyecto Vue/Cli 2, integra Vuex para contar con un store general que te permita manejar datos en states.

Te proponemos pensar en una app de dos Vistas diferentes. En una, puedes incluir un input type donde cargarás información que se almacenará en un array contenido en Vuex State.

En una segunda vista, navegable desde la aplicación, recorrerás dicho array para cargar su contenido en pantalla.

Tiempo estimado: 15 minutos.

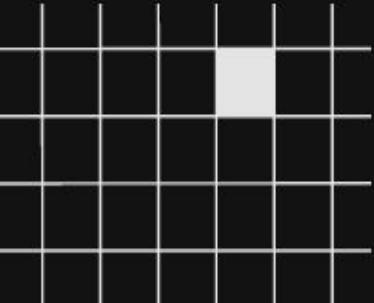
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- 
- Patrón de estado global
 - Arquitectura del patrón Vuex
 - Integrar Vuex en Vue/Cli 2
 - Acciones, Mutaciones, Estados



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN

CODER HOUSE