Guía de ejercicios - Vuex (I)

 ¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo practicar y ejercitar los contenidos que hemos visto en clase.

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

Actividad guiada: Contador con payload

A continuación, realizaremos una actividad que consiste en crear un contador de clicks. Dicho contador recibirá el valor a incrementar de manera dinámica a través de un payload.

¿Qué es un payload?
Un payload no es más que un objeto que contiene información, el cual es enviado a través de una acción. Dicha acción modifica el estado definido en nuestro store de manera dinámica.

Veamos entonces el ejemplo a través de los pasos que se listan a continuación:

- **Paso 1:** Para iniciar, empezaremos descargando el **"Material de Apoyo - Vuex-payload"** disponible en la plataforma. Una vez descargado, deberás correr el comando `npm install` para instalar todas las dependencias asociadas.

```
npm install
```

- **Paso 2:** Creamos un componente llamado `Contador.vue`, el cual tendrá el siguiente código en su interior.

```
<template lang="">
    <div>
        <h3>{{contador}}</h3>
    </div>
</template>
<script>
    import {mapState} from 'vuex'

export default {
    computed: {
        ...mapState(['contador'])
    }

}
</script>
<style lang="">

</style>
```

El código del componente `Contador.vue` está realizando un mapeo a un estado llamado "contador". Recordemos que mapState hace un mapeo de los estados que se tengan definidos en el store y luego es importado en el componente que se requiera.

- **Paso 3:** En el archivo index.js del /store, agregamos las siguientes líneas de código

para empezar con la preparación del estado global.

```
import { createStore } from 'vuex'

export default createStore({
  state: {
    contador: 0
  },
  getters: {
  },
  mutations: {
    incrementar(state, payload) {
      return state.contador = state.contador + payload
    }
  },
  actions: {
  },
  modules: {
  }
})
```

El código anterior está definiendo la estructura base de nuestro estado global. Recordemos que las mutaciones serán el medio por el cual accederemos al estado para luego modificarlo. Es importante que recordemos esto, dado que por buena práctica es el único camino que debemos seguir al momento de modificar el estado.

En este sentido tenemos:

1. Un estado inicial del contador en cero (0).
2. Una mutación que ejecutará un método incrementar que recibe el state como parámetro y un payload. Luego, retorna la lógica para que al contador se le sume el payload que le pasemos.

Te preguntarás, **¿Cómo ponemos a funcionar el payload?**

- **Paso 4:** Nos dirigimos al componente `Contador.vue` e insertamos un `<button>` que reaccionará a un método llamado `botonIncrementar`. Seguidamente, definiremos el method incrementar que realizará el commit a la mutación que definimos en el store. Veamos entonces cómo queda el código del componente.

```
<template lang="">
    <div>
        <h3>{{contador}}</h3>
    </div>
```

```
    <button @click="botonIncrementar">Incrementar contador</button>
</template>
<script>
    import {mapState} from 'vuex'

export default {
    computed: {
        ...mapState(['contador'])
    },
    methods: {
        botonIncrementar() {
            this.$store.commit('incrementar', 10)
        }
    }
}
</script>
```

Entendamos el código

- Creamos un botón que ejecutará un método llamado `botonIncrementar`.
- En el `methods` de nuestro componente estamos accediendo al store y a través del commit disparamos la mutación que definimos en el store.
- `Commit` es una palabra en inglés, que en el entorno de Vue podría interpretarse como ejecutar o cometer una acción. En este caso, la acción que ejecutará es la mutación que definimos.
- En el commit estamos pasando un segundo argumento, definido con el número 10. Este segundo argumento irá a la mutación y será pasado al `payload`.
- Al hacer `click` en el botón veremos que el contador modifica su estado inicial y va aumentando de 10 en 10.

Getters

Antes de abordar el concepto de getters, es importante recordar las computed properties en un componente. Como vimos en unidades anteriores, las computed properties son métodos que podemos ocupar para obtener valores depurados o procesados del estado. Además, son propiedades que idealmente utilizaremos al momento de mostrar información a partir de una determinada lógica en un componente.

Los getters tienen un comportamiento parecido a las propiedades computadas, en este caso, podemos interpretar que son métodos que estarán alojados en el store y que realizarán el procesamiento de lógica. El objetivo detrás de los getters es evitar la redundancia o repetición de código a través del árbol de componentes en una aplicación.

Veamos entonces cómo integrar los getters utilizando el mismo ejercicio que venimos trabajando.

- **Paso 5:** Dentro del archivo index.js del store incorporamos un nuevo estado que contendrá un arreglo de frutas.

```
state: {
    contador: 1,
    frutas: ['Fruta 1', 'Fruta 2', 'Fruta 3']
},
```

- **Paso 6:** Utilizamos el objeto getters para definir la lógica que permita calcular cuántas frutas posee el arreglo.

```
getters: {
    obtenerFrutas(state) {
      console.log(state.frutas.length)
    }
},
```

- **Paso 7:** En las propiedades computadas del componente Counter.vue llamamos al getter que definimos en el store.

```
computed: {
        ...mapState(['contador', 'frutas']),
        botonObtenerFrutas() {
            this.$store.getters.obtenerFrutas;
        },
},
```

Recuerda que dado que tenemos un nuevo estado en el store, debemos pasarlo en el mapState en el arreglo.

- **Paso 8:** Agregamos un botón que al dar click nos mostrará en consola cuántas frutas posee el arreglo del store.

```
<button @click="botonObtenerFrutas">Contar frutas</button>
```

Análisis de implementación de getters

Como puedes observar, el sentido detrás de la implementación de `getters` consiste en mantener lo más limpio posible de ejecuciones lógicas en nuestros componentes. A través

de `getters`, podemos tener en un contexto centralizado como el `store`, todas aquellas operaciones de orden lógico que luego serán mostradas en los componentes.

Actions

Como vimos en nuestro ejercicio, el estado lo modificamos directamente a través de las mutaciones que definimos. Ese planteamiento, podría parecer sencillo dado que ejecuta una tarea que no parece ser compleja. Sin embargo, hay conceptos y buenas prácticas que debemos considerar al momento de trabajar con Vuex:

1. Recordemos que las mutaciones son las encargadas de modificar el estado.
2. Las mutaciones **deben realizar** procesos síncronos, veamos un ejemplo:
    a. Supongamos que tenemos una mutación que realiza una consulta a una API para extraer información.
    b. Dado que las mutaciones son síncronas, el programa o la aplicación quedará bloqueada hasta que el proceso de consulta a la API finalice.
3. Ante el ejemplo anterior, es donde entran los procesos asíncronos y los Actions en Vuex.
    a. Las Actions (acciones) son la característica que nos provee Vuex para ejecutar tareas asíncronas como el consumo de un API.
    b. Además, las acciones **también pueden ejecutar mutaciones**.

Veamos un diagrama para visualizar de manera gráfica el flujo:

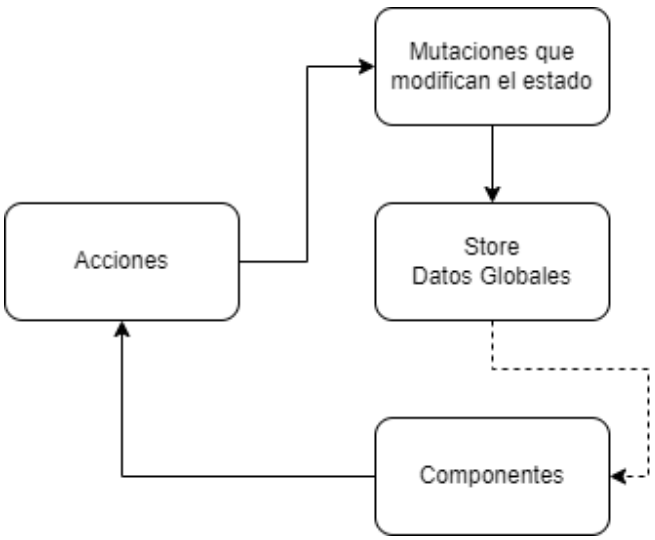

Imagen 1. Diagrama de flujo, acciones y mutaciones
Fuente: Desafío Latam

El diagrama anterior nos plantea de manera gráfica cómo es el flujo de interacción entre componentes, acciones, mutaciones y el store.

1. Tenemos un componente que ejecuta o llama una acción.
2. Esta acción recibe la instrucción del componente y activa una mutación.

3. Esta mutación recibe las instrucciones de la acción y procede a realizar los cambios requeridos en el estado.

Entonces, podemos concluir que las acciones según las buenas prácticas deben:

- Realizar procesos asíncronos como el consumo de una API.
- **Pueden** ejecutar mutaciones.

Continuemos entonces trabajando con el ejercicio anterior e ingresemos una acción que active una mutación.

- **Paso 9:** Definimos una acción llamada `accionIncrementar` y tendrá el siguiente código:

```
actions: {
    accionIncrementar(context) {
      context.commit('incrementar', 10)
    },
  },
```

Las acciones (Actions) al momento de ser definidas deben recibir como parámetro un objeto llamado `context`.

- `context` posee en su interior un conjunto de funciones útiles al momento de ejecutar mutaciones y acciones, estas son el `commit` y el `dispatch`, más adelante veremos qué es el dispatch.

Entonces, nuestra `accionIncrementar` está accediendo el contexto y está cometiendo una mutación que en este caso se llama incrementar y es el primer parámetro del `commit`. El segundo parámetro, corresponde al `payload`.

- **Paso 10:** Ahora, debemos dirigirnos al componente `Contador.vue` y hacer el `dispatch` de `accionIncrementar`. Para lograrlo, debemos modificar el methods en nuestro componente, veamos el código.

```
methods: {
        botonIncrementar() {
            this.$store.dispatch('accionIncrementar')
        },
    }
```

¿Pero qué es el dispatch?

Dispatch puede interpretarse como un disparador o detonador de acciones. Es la función

disponible en el `context` encargada de ejecutar acciones.

Hint: Para comprobar que dispatch es una función disponible en el context puedes escribir el siguiente código en el actions del store.

```
actions: {
    accionIncrementar(context) {
      console.log(typeof(context.dispatch))
      context.commit('incrementar', 10)
    },
  },
```

Con el console.log que se resalta, puedes comprobar que dispatch es una función.

Hasta este punto, ya vimos que las acciones pueden ejecutar mutaciones que modifiquen el estado, y esto es una buena práctica en el entorno de Vuex y Vue JS. Ahora, vamos a generar un proceso asíncrono en el actions dado que es lo que generalmente haremos con las acciones.

- **Paso 11:** Utilicemos setTimout() para que el commit a la mutación que modificará el estado se realice posterior a los 3 segundos.

```
actions: {
    accionIncrementar(context) {
      console.log(typeof(context.dispatch))
      setTimeout(() => {
        context.commit('incrementar', 10)
      }, 3000)
    },
  },
```

Con esta sencilla implementación de setTimeout(), evidenciamos que las acciones sí ejecutan procesos asíncronos y son según la buena práctica las que deberán realizar el consumo a APIs.

En la plataforma te compartimos el código finalizado de este ejercicio con el nombre **"Códigos Ejercicio - Contador con payload"**

¡Manos a la obra! - Agregar acción para disminuir

Aplicando los conocimientos adquiridos hasta el momento en esta guía de ejercicios, es necesario que incorpores un botón para disminuir el contador. Para lograrlo, realiza las siguientes acciones:

1. Crea una acción que podrías llamar `accionDisminuir`.
2. Genera una mutación que podría llamarse `disminuir`.
    a. Esta mutación debe capturar el estado y hacer la disminución numérica en cada clic.
    b. La disminución se realizará a partir del `payload` obtenido.
3. Crea el botón en el componente contador que haga el `dispatch` a la `accionDisminuir`.

Solución - Manos a la Obra

1. Código del store con la acción y la mutación solicitada.

```
import { createStore } from 'vuex'

export default createStore({
  state: {
    contador: 1,
    frutas: ['Fruta 1', 'Fruta 2', 'Fruta 3']
  },
  getters: {
    obtenerFrutas(state) {
      console.log(state.frutas.length)
    }
  },
  mutations: {
    incrementar(state, payload) {
      return state.contador = state.contador + payload
    },
    disminuir(state, payload) {
      return state.contador = state.contador - payload
    },
  },
  actions: {
    accionIncrementar(context) {
      console.log(typeof(context.dispatch))
      setTimeout(() => {
        context.commit('incrementar', 10)
      }, 3000)
    },
    accionDisminuir(context) {
      setTimeout(() => {
        context.commit('disminuir', 10)
      }, 3000)
    },
  },
  modules: {
  }
})
```

2. Código del componente `Contador.vue`

```
<template lang="">
    <div>
        <h3>{{contador}}</h3>
    </div>
    <button @click="botonIncrementar">Incrementar contador</button>
    <button @click="botonDisminuir">Disminuir contador</button>
    <button @click="botonObtenerFrutas">Contar frutas</button>
</template>
<script>
    import {mapState} from 'vuex'

export default {
    computed: {
        ...mapState(['contador', 'frutas']),
        botonObtenerFrutas() {
            this.$store.getters.obtenerFrutas;
        },
    },
    methods: {
        botonIncrementar() {
            this.$store.dispatch('accionIncrementar')
        },
        botonDisminuir() {
            this.$store.dispatch('accionDisminuir')
        },
    }

}
</script>
```