

## HINTS

### CONSEJOS CONCEPTUALES

Si vinculamos datos desde Vuex hacia el componente, al hacerlo de esta forma:

```
1 this.$store.state.name
```

Debemos crear una propiedad computada que lo contenga, para poder utilizarlo dentro del componente.

```
1 computed: {  
2   userName () {  
3     return this.$store.state.name;  
4   }  
5 },
```

Para evitar esto, podemos utilizar `mapState`, el cual nos permite importar cualquier dato desde Vuex, sin tener que crear propiedades computadas adicionales.

```
1 computed: {  
2   ...mapState(['name']),  
3 },
```

## RECOMENDACIÓN

### Cuando almacenar data en Vuex:

1. Si la data necesita ser accesible por múltiples componentes (independientes entre ellos).
2. Centralización de llamadas a una Api, de esta forma, un componente puede hacer la llamada, y los otros componentes solo consumir los datos desde el store de Vuex.
3. Si utilizar props y events entre componentes, complica demasiado el código.
  - a. El uso de props y events, es aconsejable cuando debemos comunicar datos entre un componente padre y un componente hijo.

### Cuando no utilizar Vuex:

1. Si solo necesitamos comunicarnos con componentes Padre/Hijo; lo recomendable es usar props y events.
2. Cuando los datos solo sean consumidos por un solo componente.

## COMPONENTES DINÁMICOS

Hay veces que, en una aplicación de Vue, tenemos una pantalla donde aparecen un número variable de componentes que el usuario puede configurar. Por ejemplo: en un dashboard, o en una pantalla de definición de filtros.

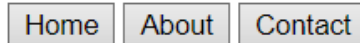
En estos casos, usar la sintaxis habitual para importar componentes, se hace tedioso y complicado, por lo que es más sencillo utilizar el componente de Vue component, y un sistema de carga dinámica.

Tenemos el siguiente código. Queremos que cada vez que le hacemos clic a un botón, éste cambie de vista, y para eso, hacemos uso de los componentes dinámicos, que se reduce en utilizar un componente llamado component

```
<div id="app">
  <button>Home</button>
  <button>About</button>
  <button>Contact</button>
</div>
```

```
new Vue({
  el: "#app",
  component: {
    home: {
    },
  },
  data: {
  },
  methods: {
  }
})
```

Y se verá en el navegador de la siguiente manera.



Para utilizar un componente dinámico, lo que haremos será definir algunos componentes en la instancia de Vue, con el contenido que tendrá cada uno de sus templates.

Entonces, cada vez que demos clic a uno de los botones, debemos reemplazar el contenido de los componentes que están en la instancia de Vue, en el div de nuestro html

```
new Vue({
  el: "#app",
  component: {
    home: {
      template: "<p>Home</p>"
    },
    about: {
      template: "<p>About</p>"
    },
    Contact: {
      template: "<p>Contact</p>"
    }
  },
})
```

Definiremos una acción, y cada vez que le demos clic, debemos pasar lo siguiente: crearemos la acción `switchComponent`, dentro de la que irá el nombre del componente, como se ve en la imagen.

```
<div id="app">
  <button @click="switchComponent('home')">Home</button>
  <button @click="switchComponent('about')">About</button>
  <button @click="switchComponent('contact')">Contact</button>
</div>
```

Ahora, lo que debemos hacer es definir un componente llamado `component`, y éste lo que recibe es una propiedad llamada `is`, al que se le da un valor, en este caso, el nombre del componente.

```
<div id="app">
  <button @click="switchComponent('home')">Home</button>
  <button @click="switchComponent('about')">About</button>
  <button @click="switchComponent('contact')">Contact</button>
  <div>
    <component v-bind:is="current"></component>
  </div>
</div>
```

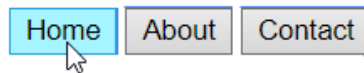
Definimos una propiedad llamada `current`, la cual se debe indicar en el apartado de `data`, y ahí se le dará el nombre del componente por defecto.

```
new Vue({
  el: "#app",
  component: {
    home: {
      template: "<p>Home</p>"
    },
    about: {
      template: "<p>About</p>"
    },
    Contact: {
      template: "<p>Contact</p>"
    }
  },
  data: {
    current: 'home'
  },
})
```

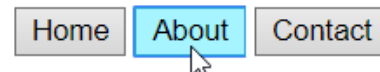
Ahora vamos a definir el método switchComponent, le pasaremos un parámetro value, y se lo asignaremos a la propiedad current.

```
new Vue({
  el: "#app",
  component: {
    home: {
      template: "<p>Home</p>"
    },
    about: {
      template: "<p>About</p>"
    },
    Contact: {
      template: "<p>Contact</p>"
    }
  },
  data: {
    current: 'home'
  },
  methods: {
    switchComponent(value) {
      this.current = value;
    }
  }
})
```

Si ejecutamos y vamos al navegador, vemos que al presionar cada uno de los botones, se mostrará el contenido correspondiente. Esto nos indica que al dar clic, está cambiando de componente.



Home



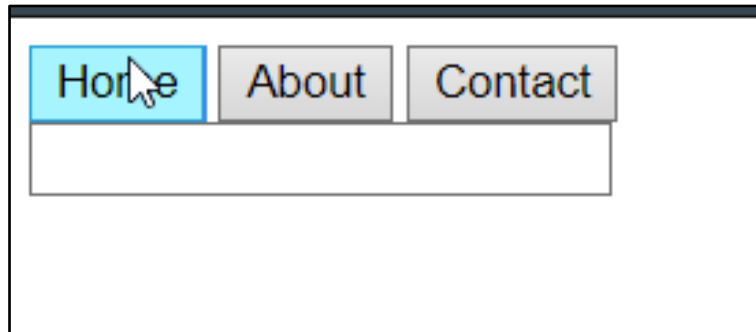
About



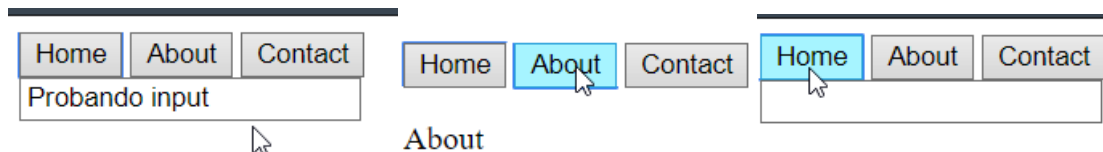
Contact

Ahora, cambiaremos el contenido del template del componente home, y agregaremos un input.

```
const app = new Vue({
  el: "#app",
  components: {
    home: {
      template: "<input type='text'/>"
    },
    about: {
      template: "<p>About</p>"
    },
    Contact: {
      template: "<p>Contact</p>"
    }
  }
})
```

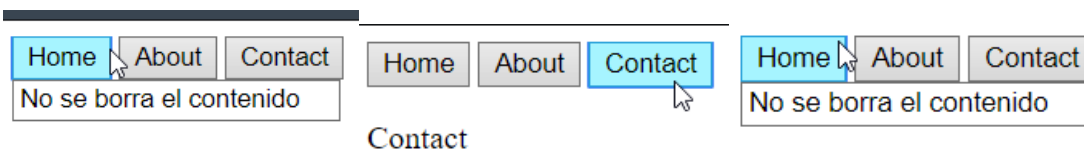


Se presenta un problema. Si escribimos algo en el input del home, cambiamos de vista a about, y volvemos al home, se borra la información que habíamos ingresado antes. Esto sucede porque al momento que cambiamos de componente, éstos se autodestruyen, es decir, cada vez que damos clic, se generará uno nuevo.



Para solucionarlo, debemos utilizar el elemento `<keep-alive>`, y envolver nuestro componente dinámico en éste, así le indicamos a Vue que no autodestruya a ese componente, y nos aseguramos de que el contenido que ingresamos no desaparezca.

```
<body>
  <div id="app">
    <button @click="switchComponent('home')">Home</button>
    <button @click="switchComponent('about')">About</button>
    <button @click="switchComponent('contact')">Contact</button>
    <div>
      <keep-alive>
        <component v-bind:is="current"></component>
      </keep-alive>
    </div>
  </div>
</body>
```



## PROPS

Se pueden utilizar para pasar datos a un componente. Por ejemplo: tendremos el componente padre, que quiere pasarle información a un componente hijo; será title con la frase “Mi título”, e images\_src que tendrá “images”, un Array de imágenes definido en la data.

```
<template>
  <div id="app">
    <componenteHijo
      title="Mi título"
      :images_src="images">
    </componenteHijo >
  </div>
</template>
<script>

import componenteHijo from '@components/componenteHijo.vue'
export default {
  name: 'App',
  components: {
    componenteHijo,
  },
  data: function(){
    return{
      images: [
        "http://lorempixel.com/200/200/sports",
        "http://lorempixel.com/200/200"
      ],
    }
  },
}
```

En el componente hijo debemos recibir los datos definidos en el componente padre, para eso necesitaremos utilizar props, y los declararemos dentro del script del componente hijo como un Array.

```
export default {
  name: "componente-hijo",
  props: ['title', 'images_src']
}
```



Ahora, para mostrar los datos que recibimos con props, en el template del componente hijo, lo haremos de la misma forma que con datos declarados dentro de data.

```
<template>
  <div>
    <h1>{{title}}</h1>
    
  </div>
</template>
```

La props title está siendo utilizada dentro de la etiqueta h1, y la props images\_src está dentro de un v-for, porque es un Array.

Puedes encontrar más información sobre props en el Drill 6: Desarrollo de interfaces interactivas con framework Vue, CUE 3.

## EMITIR EVENTOS

Para gatillar un evento, lo que debemos hacer es utilizar la palabra reservada \$emit, que nos permite generarlos y que puedan ser escuchados por un componente padre.

Por ejemplo, tenemos lo siguiente con dos botones.

```
<template>
  <div class="container">
    <h1>{{title}}</h1>
    <form>
      <label for="">Ingrese src imagen</label>
      <input type="text" v-model="newImage">
      <button class="btn btn-add" @click.prevent="agregar">Agregar</button>
    </form>

    <div class="card" v-for="(image,index) in images_src" :key="image" >
      
      <button class="btn btn-delete" @click="eliminar(index)">Eliminar</button>
    </div>
  </div>
</template>
```

Dichos botones están asociados a métodos. Si bien los eventos pueden ser gatillados desde el mismo template, en este caso, lo haremos más ordenado y lo asociaremos a un método.

```
methods:{
  eliminar(index){
    this.$emit('delete',index);
  },
  agregar(){
    this.$emit('add',this.newImage);
    this.newImage = "";
  }
}
```

Si notamos, los dos eventos tienen un dato asociado, el cual podrá ser leído posteriormente por el padre, cuando el evento se gatille.

En el componente padre podremos agregar los eventos delete y add, los cuales asociaremos a los métodos que serán creados.

```
<template>
  <div id="app">
    <componenteHijo
      title="Mi titulo"
      :images_src="images"
      v-on:delete="eliminarImagen"
      @add="agregarImagen">
    </componenteHijo>
  </div>
</template>
```

Si lo notas, el evento delete es llamado con la directiva v-on, y el evento "add" es llamado con @. Esto es solo para ejemplificar que @ es el atajo para v-on.

El componente hijo gatilla un evento, y el padre es el encargado de ejecutar la acción. En el caso del ejemplo, borrar o agregar un elemento a un Array.

```
import componenteHijo from '@components/componenteHijo.vue'
export default {
  name: 'App',
  components: {
    componenteHijo,
  },
  data: function(){
    return{
      images: ["http://lorempixel.com/200/200/sports", "http://lorempixel.com/200/200"],
    }
  },
  methods:{
    eliminarImagen(index){
      this.images.splice(index,1);
    },
    agregarImagen(src){
      this.images.push(src);
    }
  }
}
```

Puedes encontrar más detalles sobre eventos en el Drill 6: Desarrollo de interfaces interactivas con framework Vue, CUE 3.