Vue

Vue.js es, en palabras de Evan You, su creador, un **framework progresivo** para crear interfaces de usuario. Vue se basa en una implementación muy ligera del patrón *view-model* que nos ayudará a relacionar nuestra capa de presentación con nuestra capa de negocio de forma sencilla y eficiente. A lo que se refiere You con ?progresivo?, es a que es muy fácil añadir Vue.js a cualquier proyecto ya existente y poder aprovechar su funcionalidad casi sin complicaciones. Por el contrario, otros frameworks como Angular son mucho más orientados a comenzar proyectos desde cero, y con una arquitectura predeterminada que viene dirigida por el framework.

Vue comenzó originariamente como un proyecto que aspiraba a reproducir un conjunto mínimo de funcionalidades de Angular, pero sin tener que amoldarse a su arquitectura. Así, You, ingeniero de Google, comenzó a trabajar en Vue en 2013 y lo publicó en github en febrero de 2014. Actualmente Vue ha tenido un gran impacto y hay varias compañías de renombre que utilizan Vue en su front-end, como Gitlab, Alibaba o Nintendo.

Así, Vue.js se caracteriza por ser uno de los frameworks javaScript de mayor rendimiento, gracias a una implementación muy ligera del llamado **virtual DOM**.

Sumario

- 1 Introducción al virtual DOM
- 2 Comenzar a trabajar con Vue.js
    - 2.1 Ejemplos de introducción a Vue.js
        - 2.1.1 Cambio dinámico
        - 2.1.2 Unión de caja de texto con otro elemento
        - 2.1.3 Aplicar condiciones
        - 2.1.4 Definir el momento en el que se invoca un método
        - 2.1.5 **v-if** frente a **v-show**
        - 2.1.6 Bucles con v-for
        - 2.1.7 Enlazar atributos y propiedades de elementos DOM
        - 2.1.8 Enlace de datos bidireccional
    - 2.2 Capturar eventos
    - 2.3 Métodos
    - 2.4 this
    - 2.5 Watchers
- 3 Aplicaciones sencillas
    - 3.1 Administrador cuenta de banco muy sencilla con Vue.js
        - 3.1.1 Modificar etiquetas
        - 3.1.2 v-bind
        - 3.1.3 v-if -- v-else-if -- v-else
        - 3.1.4 v-for
        - 3.1.5 v-on
        - 3.1.6 Clases dinámicas
        - 3.1.7 Propiedades Computadas
        - 3.1.8 Introducción a Componentes
    - 3.2 Libreta de pagos pendientes con Vue.js
    - 3.3 CRUD PHP con Vue.js + Axios + MariaDB
- 4 Vue.js CLI
    - 4.1 Instalación Vue.js
    - 4.2 Archivos Proyecto CLI
    - 4.3 Módulos con CLI
    - 4.4 Compilar para publicar en un servidor

Introducción al virtual DOM

Como sabemos, el DOM es una estructura en forma de árbol que se crea una vez que el navegador parsea una archivo HTML, contiene todos los elementos que deben ser renderizados en la página. Esto puede funcionar bien cuando se trata de una aplicación pequeña, pero empieza a volverse costoso cuando hay que encontrar un elemento en particular en un árbol que contenga cientos de nodos de una aplicación más grande. Para solucionar esto VUE, y otros frameworks, implementan un DOM virtual. El DOM virtual es una representación del DOM real en forma de objeto JS. Se utiliza con el objetivo de minimizar las interacciones con el DOM real.

Ventajas y desventajas del DOM virtual

Ventajas:

- ◊ Te permite controlar cuando se actualiza el DOM real, si un elemento se modifica repetidas veces podemos hacer que se renderize solo su última modificación.
- ◊ Es mucho menos costoso modificar una propiedad de un objeto JS que un elemento de un árbol, por lo que una aplicación web con DOM virtual será mucho más eficiente.
- ◊ Sólo se vuelven a renderizar los elementos que hayan sufrido algún cambio, por ejemplo, si queremos cambiar de una página a otra sólo se volverán a renderizar los elementos que tengan alguna diferencia con el de la página anterior.

Desventajas:

- ◊ Consume mas recursos, ya que necesita tener una copia del DOM cargada en memoria mientras siga funcionando la aplicación.
- ◊ En un programa pequeño y con cambios pocos frecuentes en sus elementos el DOM virtual puede hacer que la aplicación funcione más lenta que con el DOM real, ya que agrega lógica a cada actualización.

Comenzar a trabajar con Vue.js

Vue.js tiene una documentación muy cuidada que nos permite ir aprendiendo desde cero cómo utilizar este framework de JavaScript.

El modo más sencillo de comenzar a utilizarlo es enlazar en nuestro proyecto el CDN de Vue.js. Para empezar a programar se enlazará la versión de desarrollo:

```
<script src="https://unpkg.com/vue@next"></script>
```

Ejemplos de introducción a Vue.js

Cambio dinámico

Vue nos permite de un modo dinámico cambiar el contenido de elementos de la web

- **HTML**

```
<body>

    <div id="app">

      {{ mensaje }}
    </div>


    <script src="https://unpkg.com/vue@next"></script>

    <script src="scripts.js"></script>

</body>
```

- **JavaScript**

```
// Archivo : "scripts.js"

// Creamos una instancia de objeto Vue
const App = {

 // Con 'data()' definimos el objeto que guardará todos los datos
    //con los que trabajará la instancia Vue
    data() {
      return {
        mensaje: 'Hola Vue.js'
      }
    }
  }
     // Proporcionamos ahora a la instancia Vue un elemento DOM existente para montarla
    //puede ser un selector CSS o un elemento HTML
    //Vue convertirá recursivamente sus propiedades en getters/setters para hacerlas "reactivas"
  Vue.createApp(App).mount('#app')
```

```
})
```

Unión de caja de texto con otro elemento

También podemos realizar una ?unión? entre una caja de texto y cualquier otro elemento empleando la directiva v-model

- **HTML**

```
<body>

    <div id="app">

        <h1>Saludo: {{ mensaje }}</h1>

        <input type="text" v-model='mensaje' />
    </div>

</body>
```

- **JavaScript**

```
// Archivo : "scripts.js"

const App = {

    data() {
      return {
        mensaje: 'Hola Vue.js'
      }
    }
  }
  Vue.createApp(App).mount('#app')
})
```

Aplicar condiciones

Con la directiva v-if podemos mostrar u ocultar elementos al cumplirse una condición:

- **HTML**

```
<body>

    <div id="app">

        <span v-if="ok">Ahora me ves!!</span>
    </div>

</body>
```

- **JavaScript**

```
const App = {
    data() {
      return {
        ok: true
      }
    }
  }

  Vue.createApp(App).mount('#app')
```

Un ejemplo más elaborado con **v-if** es el siguiente:

- **HTML**

```
<body>
```

```
    <div id="app">
        <p v-if="hora < 12">Buenos días!</p>
        <p v-if="hora >=12 && hora < 20">Buenas tardes!</p>
        <p v-if="hora >= 20">Buenas noches!</p>
    </div>

</body>
```

- **JavaScript**

```
const App = {
    data() {
      return {
        hora: new Date().getHours()
      }
    }
  }

  Vue.createApp(App).mount('#app')
```

Y otro ejemplo donde, además, se utiliza la función setInterval de JavaScript, puede ser el siguiente:

- **HTML**

```
<body>

    <div id="app">
        <p v-if="num < 12">Menor de 12</p>
        <p v-if="num >=12 && num < 20">Entre 12 y 20</p>
        <p v-if="num >= 20">Mayor de 20</p>
    </div>

</body>
```

- **JavaScript**

```
var app = new Vue({
el: "#app",
data: {
num: 0
},
methods: { //Definimos los métodos de nuestra instancia Vue
testFunction() {
//Utilizar function tipo "arrow" para acceder a las variables definidas en 'data'
setInterval(() => {
this.num++;
console.log(this.num);
}, 1000);
}
},
mounted() { //Se invoca después de que se ha montado la instancia
this.testFunction();
}
});
```

Definir el momento en el que se invoca un método

Vue.js nos permitirá definir acciones anteriores y posteriores a la transición desde o hacia cada estado interno del componente. Los métodos en cuestión para implementar estas acciones son:

- **beforeCreate** : evento lanzado antes de tener el componente cargado, implica no poder acceder al componente a nivel de DOM.
- **created** : evento donde se verifica si el componente tiene plantilla, entonces se compila y se observan las propiedades computads, data, métodos y eventos. Pero no podemos acceder al $el.
- **beforeMount** : evento que ocurre antes de representar el commponente.
- **mounted** : evento que implica que el componente está cargado por completo, se añade al DOM, quedando el componente accesible a través de $el.
- **beforeUpdate** : evento que se ejecuta cuando el valor del data del componente cambia.

- **updated** : evento invocado tras finalizar la modificación de valor del data.
- **beforeDestroy** : evento que elimina eventos que estuvieran activos en el componente, antes de eliminar la instacia.
- **destroyed** : evento lanzado tras desacoplar el componente.

v-if frente a v-show

Si **v-if** tiene un valor 'falsy', es decir, si es: **false**, **undefined**, **null**, 0, "" o **NaN**, el elemento asociado NO aparecerá en el DOM. Si, por ejemplo, tenemos esta plantilla:

```
<div v-if=?true?>Primero</div>
<div v-if=?false?>Segundo</div>
```

La salida será:

```
<div>Primero</div>
```

El funcionamiento de **v-show** es similar, pero distinto, pues utiliza CSS para mostrar u ocultar los elementos asociados. Si tenemos esta plantilla:

```
<div v-show=?true?>Primero</div>
<div v-show=?false?>Segundo</div>
```

La salida será:

```
<div>Primero</div>
<div style=?display: none?>Segundo</div>
```

Saber que **v-show** tiene menor coste de procesamiento que **v-if**. Además, si el elemento contiene alguna imagen, entonces ocultar el contenedor solo con CSS permite que el navegador descargue la imagen antes de que se muestre, lo que significa que se puede mostrar tan pronto como **v-show** se ponga en **true**. Utilizando **v-if**, no comenzaría a descargarse la imagen hasta que se necesitase mostrar.

Bucles con v-for

Otra directiva interesante es v-for, que nos permite realizar un bucle a través de un objeto o un array.

- **HTML**

```
<div id="app">
  <ol>

    <li v-for="todo in todos">
      {{ todo.text }}
    </li>
  </ol>
</div>
```

- **JavaScript**

```
var app = new Vue({
    el: "#app",
    data: {
      todos: [
        //Array sobre el que vamos a iterar
        //Se trata de un array de objetos (clave:valor)
        { text: "Learn JavaScript" },
        { text: "Learn Vue" },
        { text: "Build something awesome" },
      ],
    }
  });
```

Y, si queremos hacer un simple contador, lo podemos hacer del siguiente modo:

- **HTML**

```
<div id="app">
  <ul>
```

```
    <li v-for="n in 10">{{ n }}</li>
  </ul>
</div>
```

- **JavaScript**

```
var app = new Vue({
el: "#app",
});
```

Enlazar atributos y propiedades de elementos DOM

Podemos enlazar también atributos o propiedades de elementos del DOM:

- **HTML**

```
<div id="app">

  <span v-bind:title="mensaje">
    Al poner el puntero sobre mí. <br>
    Sale un mensaje (tooltip text)
  </span>
</div>
```

- **JavaScript**

```
var app = new Vue({
  el: '#app',
  data: {
    mensaje: 'Fecha actual: ' + new Date().toLocaleString()
  }
});
```

Podemos configurar también alguna propiedad de los elementos DOM. Veamos un ejemplo con un botón:

- **HTML**

```
<div id="app">
  <button v-bind:type="buttonType">Botón de test</button>
</div>
```

- **JavaScript**

```
var app = new Vue({
  el: '#app',
  data: {
   buttonType: 'submit'
  }
});
//Salida : <button type="submit">Botón de test</button>
```

Que funcionaría de igual modo con las propiedades **disabled** y **checked**.

También saber que, normalmente, se utiliza la versión corta a la hora de utilizar **v-bind**.

```
<button :type="buttonType">Botón de test</button>
```

Enlace de datos bidireccional

Veamos un ejemplo donde se actualizan los datos de la web empleando botones "radio".

- **HTML**

```
<div id="app">
<label><input type="radio" v-model="value" value="uno">Uno</label>
<label><input type="radio" v-model="value" value="dos">Dos</label>
<label><input type="radio" v-model="value" value="tres">Tres</label>
```

```
<br>
<p>El valor es : {{ value }}</p>
</div>
```

- **JavaScript**

```
var app = new Vue({
el: '#app',
data: {
  value: 'uno'
}
});
```

Capturar eventos

Para capturar eventos podemos utilizar la directiva v-on para escuchar eventos DOM y ejecutar código JavaScript cuando se activan.

- **HTML**

```
<div id="app">
  <p>{{ message }}</p>

  <button v-on:click="reverseMessage">Mensaje Inverso</button>
</div>
```

- **JavaScript**

```
var app = new Vue({
    el: "#app",
    data: {
      message: "Hola Vue.js!",
    },
    methods: {
      reverseMessage: function () {
        // 'this' hace referencia al 'div' donde está el botón
        this.message = this.message.split("").reverse().join("");
      }
    }
  });
```

Métodos

Vue nos permite crear funciones para no tener que repetir código y utilizar éstas como métodos. Veamos un ejemplo:

- **HTML**

```
<div id="app">
 <input type="text" v-on:keyup="estadoFromId(estadoId)" v-model="estadoId" />
 <p>Tu estado actual es: {{ estado }}</p>
</div>
```

- **JavaScript**

```
var app = new Vue({
    el: '#app',
    data: {
      estadoId: 2,
      estado: ''
    },
    created: function() {
      this.estado = this.estadoFromId(this.estadoId);
    },
    methods: {
      estadoFromId(id) {
        //console.log(id);
        const e = ({
          0: 'Durmiendo',
          1: 'Comiendo',
```

```
          2: 'Aprendiendo Vue'
        })[id];
        this.estado = e || 'Estado desconocido ' + id;
      }
    },
});
```

Veamos otro ejemplo interesante donde un método devuelve un array con los números positivos existentes en otro más amplio que se le pasa como parámetro. Con ese método se crea una lista:

- **HTML**

```
<div id="app">
  <ul>
    <li v-for="numero in filtraPositivos(numeros)">{{ numero }}</li>
  </ul>
</div>
```

- **JavaScript**

```
new Vue({
  el: "#app",
  data: {
    numeros: [-5, 0, 2, -1, 1, 0.5],
  },
  methods: {
    filtraPositivos(numeros) {
      return numeros.filter((numero) => numero >= 0);
    }
  }
});
```

this

Como vimos antes, en un método, **this** se refiere al componente al que está asociado el método. Se puede acceder también a las propiedades y a otros métodos también. Veamos un ejemplo donde se demuestra esto:

- **HTML**

```
<div id="app">
    <ul>
      <p>
        La suma de los números positivos de la lista: {{numeros.join(', ')}} -> Es: {{calculaSumaNumPos()}}
      </p>
    </ul>
</div>
```

- **JavaScript**

```
new Vue({
  el: "#app",
  data: {
    numeros: [-5, 0, 2, -1, 1, 0.5],
  },
  methods: {
    filtraPositivos() {
      return this.numeros.filter((numero) => numero >= 0);
    },
    calculaSumaNumPos() {
      return this.filtraPositivos().reduce((sum, val) => sum + val);
    },
  },
});
```

Watchers

Si bien las propiedades computadas son más apropiadas en la mayoría de los casos, hay ocasiones en que es necesario un observador personalizado. Es por eso que Vue proporciona una forma más genérica de reaccionar a los cambios de datos a través de la opción watch. Esto es más útil cuando

desea realizar operaciones asíncronas o costosas en respuesta al cambio de datos.

- **HTML**

```
<div id="app">
  <input type="text" v-model="inputValue" />
  <br />
  <p>Cinco segundos después, en la entrada: {{ oldInputValue }}</p>
</div>
```

- **JavaScript**

```
new Vue({
  el: "#app",
  data: {
    inputValue: "",
    oldInputValue: "",
  },
  watch: {
    inputValue() {
      const newValue = this.inputValue;
      setTimeout(() => {
        this.oldInputValue = newValue;
      }, 5000);
    },
  },
});
```

Aplicaciones sencillas

Administrador cuenta de banco muy sencilla con Vue.js

-> Ejemplo sacado del curso Vue.js de Udemy

Modificar etiquetas

- **index.html**

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">


    <script src="https://unpkg.com/vue@next"></script>

    <title>Document</title>
</head>
<body>


    <div id="app">

        <h1> {{ titulo }} </h1>
        <h2> Mi saldo disponible es: {{ cantidad }} </h2>

    </div>

    <script src="main.js"></script>

    <script>
        const mountedApp = App.mount('#app')
    </script>
</body>
</html>
```

- **scripts.js**

```
const App = Vue.createApp({

    data() { //Los datos que vamos utilizar en el app
        return {
            titulo: 'Hola mundo desde Vue.js',
            cantidad: 500
        }
    }
})
```

v-bind

La directiva v-bind (dos puntos) es una de las directivas más utilizadas y populares de Vue. Esta directiva permite enlazar (bindear) una variable de Vue con un atributo específico de una etiqueta HTML. De esta forma, podemos colocar como valor de un atributo HTML el contenido que tengamos almacenado en una variable de la lógica de Javascript.

- **index.html**

```
<div id="app">

        <h1>{{ titulo }}</h1>
        <h2> Mi saldo disponible es: {{ cantidad }} </h2>

        <a v-bind:href="enlace">IES San Clemente</a>



    </div>
```

- **scripts.js**

```
const App = Vue.createApp({

    data() { //Los datos que vamos utilizar en el app
        return {
            titulo: 'Hola mundo desde Vue.js',
            cantidad: 500,
            enlace: 'https://www.iessanclemente.net'
        }
    }
  })
```

v-if -- v-else-if -- v-else

Con v-if se muestra o elimina un elemento dependiendo una condición. También es interesante v-show que, aparentemente hace el mismo efecto pero, lo que hace es "ocultar" (eliminar la visibilidad) de ese objeto (Diferencias v-if vs v-show).

- **index.html'**

```
<div id="app">

        <h1>{{ titulo }}</h1>
        <h2> Mi saldo disponible es: {{ cantidad }} </h2>

        <a v-bind:href="enlace">IES San Clemente</a>



        <hr>
        <h2 v-if="estado">Cuenta activa</h2>
        <h2 v-else>Cuenta desactivada</h2>

        <hr>
        <h2 v-if="cantidad > 500">Cantidad: {{ cantidad }}</h2>
        <h2 v-else-if="cantidad <= 500 && cantidad > 0">
            Cantidad:
            <span style="color:red">{{cantidad}}</span>
```

```
        </h2>
        <h2 v-else>Sin cantidad {{cantidad}}</h2>

    </div>

    <script src="scripts.js"></script>
    <script>
        const mountedApp = App.mount('#app')
    </script>
```

- **scripts.js**

```
const App = Vue.createApp({

    data() { //Los datos que vamos utilizar en el app
        return {
            titulo: 'Mi banco con Vue.js',
            cantidad: 1000,
            enlace: 'https://www.iessanclemente.net',
            estado: true
        }

    }

  })
```

v-for

La directiva v-for es muy interesante para crear estructuras repetitivas de código HTML de una forma sencilla y sin que el código resulte excesivamente complejo (sobre todo en estructuras que se repiten muchas veces).

- **index.html'**

```
<div id="app">

        <h1>{{ titulo }}</h1>
        <h2> Mi saldo disponible es: {{ cantidad }} </h2>

        <a v-bind:href="enlace">IES San Clemente</a>



        <hr>
        <h2 v-if="estado">Cuenta activa</h2>
        <h2 v-else>Cuenta desactivada</h2>

        <hr>
        <h2 v-if="cantidad > 500">Cantidad: {{ cantidad }}</h2>
        <h2 v-else-if="cantidad <= 500 && cantidad > 0">
            Cantidad:
            <span style="color:red">{{cantidad}}</span>
        </h2>
        <h2 v-else>Sin cantidad {{cantidad}}</h2>

        <hr>

        <h2>Mis servicios disponibles: </h2>
    //The 'key' attribute tells Vue how your data relates to the HTML elements it's rendering to the screen.
    //When your data changes, Vue uses these 'keys' to know which HTML elements to remove or update, and if it needs to create any n
        <ul>
            <li v-for="(item, index) in servicios" :key="index">{{index + 1}} - {{item}}</li>
        </ul>

    </div>

    <script src="main.js"></script>
    <script>
        const mountedApp = App.mount('#app')
    </script>
```

- **scripts.js**

```
const App = Vue.createApp({

    data() { //Los datos que vamos utilizar en el app
        return {
            titulo: 'Mi banco con Vue.js',
            cantidad: 0,
            enlace: 'https://www.iessanclemente.net',
            estado: false,
            servicios: ['transferencias', 'pagos', 'giros', 'cheques']
        }

    }

  })
```

v-on

Podemos usar la directiva v-on para escuchar eventos DOM y ejecutar algunos JavaScript cuando se activan.

- **index.html**

```
...
        <h2>Mis servicios disponibles: </h2>
        <ul>
            <li v-for="(item, index) in servicios" :key="index">{{index + 1}} - {{item}}</li>
        </ul>

        <button v-on:click="agregarSaldo">Agregar saldo</button>
        <!-- <button @click="agregarSaldo">Agregar saldo</button> -->

    </div>

    <script src="main.js"></script>
    <script>
        const mountedApp = App.mount('#app')
    </script>
```

- **scripts.js'**

```
const App = Vue.createApp({

    data() { //Los datos que vamos utilizar en el app
        return {
            titulo: 'Mi banco con Vue.js',
            cantidad: 0,
            enlace: 'https://www.iessanclemente.net',
            estado: false,
            servicios: ['transferencias', 'pagos', 'giros', 'cheques']
        }
    },
    methods: {
        agregarSaldo() {
            this.cantidad = this.cantidad + 100
        }
    }

  })
```

Y programamos el botón 'Disminuir' configurando un método al que se le pasa un parámetro:

- **index.html**

```
        <button @click="agregarSaldo">Agregar saldo</button>

        <hr>
        <button @click="disminuirSaldo(100)" :disabled="desactivar">Disminuir saldo</button>
```

```
    </div>
```

- **scripts.js**

```
methods: {
        agregarSaldo() {
            this.cantidad = this.cantidad + 100
            if (this.cantidad > 0) this.desactivar = false
        },
        disminuirSaldo(valor) {
            this.cantidad = this.cantidad - valor
            if (this.cantidad === 0) this.desactivar = true
        }
    }
```

Clases dinámicas

Podemos hacer un cambio de estilos empleando clases dinámicas, en el siguiente ejemplo lo hacemos con clases de Bootstrap.

- **index.html**

```
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1a

    <title>Ejemplos Vue</title>
</head>

<body>


    <div id="app">

        <h1>{{ titulo }}</h1>
        <h2
            class="bg-dark"
            :class="[cantidad > 500 ? 'text-success' : 'text-danger']"
            >
            Mi saldo disponible es: {{ cantidad }}
        </h2>
...
```

- **scripts.js**

Sin cambios

Propiedades Computadas

Las propiedades computadas nos sirven para generar cálculos en nuestros componentes, por ejemplo no se recomienda colocar demasiada lógica en nuestras plantillas HTML, ya que dificulta la interpretación de los componentes.

Vemos que las variables computadas son como variables a las que antes se le pueden aplicar una serie de cálculos o transformaciones.

Las variables que se crean en el **data** no pueden depender de otras variables del *data*, por ejemplo no se puede crear una variable en el *data* cuyo valor sea el doble de otra variable. Las propiedades computadas vienen a resolver este problema.

Las computadas son funciones que SIEMPRE tienen que devolver un valor (actúan como un *getter*).

Es importante saber que las computadas nunca pueden recibir un parámetro desde fuera de la función. Si necesitas pasar un valor a una computada tienes que crear un método.

Dentro de cada computada con el this accedemos a los valores del data o a los métodos que necesitemos (esto último NO se recomienda).

Las computadas se comportan igual que las variables. Si por ejemplo llamas dos veces a una computadas, si el resultado es el mismo, **Vue** es lo suficientemente listo como para no tener que recalcular cada vez su valor.

Además, las variables computadas son reactivas también y actualizarán su valor en la vista cuando cambie su valor.

Otra cosa muy interesante de las computadas es que **Vue** las trata como una variable más y no como métodos. Esto quiere decir que dentro de los métodos o de una vista podemos llamar a las propiedades computadas como si fueran variables.

-> Métodos vs Computadas

A simple vista los métodos y las computadas se pueden confundir, porque un método puede usarse como una especie de computada si devolvemos valores también, pero hay diferencias.
En primer lugar, las computadas son reactivas y su valor se actualiza solo. Con un método que devuelva algo del **data** solo se ejecutaría la primera vez y no reaccionaría a un cambio en la variable del **data**.
Otra diferencia es que las propiedades computadas tienen caché, es decir, utilizar propiedades computadas es más óptimo porque si Vue detecta que la computada va a devolver el mismo valor, no ejecutará la computada ahorrando cálculos. Los métodos se ejecutan siempre cada vez aunque el resultado sea el mismo.
Aunque se puede hacer, dentro de las computadas se recomienda no llamar a los métodos del componente. Como hemos dicho el propósito de las computadas es devolver valores del definidos en el data pero con alguna transformación. Además se pueden usar las computadas para devolver un valor que dependa de varias variables del *data*.

- **index.html**

```
...
<div id="app">

        <h1>{{ mayusculasTexto }}</h1>
        <h2
            class="bg-dark"
            :class="colorCantidad"
            >
            Mi saldo disponible es: {{ cantidad }}
        </h2>
...
```

- **scripts.js**

```
...
,
    computed: {
        colorCantidad() {
            return this.cantidad > 500 ? 'text-success' : 'text-danger'
        },
        mayusculasTexto() {
            return this.titulo.toUpperCase()
        }
    }
...
```

Introducción a Componentes

Los Componentes son instancias reutilizables (html, css y javascript). Nos van a permitir estructurar la lógica de nuestro proyecto en diferentes secciones y partes. Podemos crear un *footer* para un sitio web.

1.- Modificamos el archivo **index.html** del siguiente modo:

```
        ...
             <hr>
                <button @click="disminuirSaldo(100)" :disabled="desactivar">Disminuir saldo</button>

                <div class="bg-dark py-3 mt-2 text-white">
                    <h3>Footer de mi sitio web</h3>
                </div>

            </div>
        ...
```

Comprobamos como se vería en el navegador... y pasamos a crear nuestro Módulo Vue.

2.- Creamos un nuevo directorio **"components"** en el directorio principal de nuestro proyecto. Dentro creamos un archivo **Footer.js**. Dentro de ese archivo pegamos el código que escribimos antes en el archivo **index.html** cortándolo de este último. El contenido de **Footer.js** tiene ahora el siguiente aspecto (recuerda eliminar ese código del archivo **index.html**). También vamos a aprovechar para instalar la extensión de VSCode **es6-string-html**

que nos va a colorear el código **html** pero utilizando unas etiquetas especiales que iremos viendo a continuación.

```
App.component('footer-banco', { //El nombre del componente siempre "minúsculas y separado por guiones"
    template: /*html*/`
    <div class="bg-dark py-3 mt-2 text-white">
            <h3>Footer de mi sitio web</h3>
    </div>
    `
})
```

3.- Volver al archivo **index.html** para enlazar el **Footer.js** del siguiente modo:

```
...
       <hr>
        <button @click="disminuirSaldo(100)" :disabled="desactivar">Disminuir saldo</button>

        <footer-banco></footer-banco>


    </div>

    <script src="main.js"></script>
    <script src="components/Footer.js"></script>
    <script>
        const mountedApp = App.mount('#app')
    </script>
```

4.- Añadir propiedades a módulos **props**:

- **index.html**

```
...
        <footer-banco
            cantidad="700"
        />
...
```

- **Footer.js**

```
App.component('footer-banco', { //El nombre del componente siempre "minúsculas y separado por guiones"
    props: ['cantidad'], //Array de props
    template: /*html*/`
    <div class="bg-dark py-3 mt-2 text-white">
            <h3>{{ texto }} - {{ cantidad }}</h3>
    </div>
    `,
    data() {
        return {
            texto: 'Footer de mi sitio web'
        }
    }
})
```

5.- También podemos añadir métodos:

- **Footer.js**

```
App.component('footer-banco', { //El nombre del componente siempre "minúsculas y separado por guiones"

    props: ['cantidad'],

    template: /*html*/`
    <div class="bg-dark py-3 mt-2 text-white">
            <h3>{{texto}} - {{cantidad}}</h3>
            <p>{{fecha()}}</p>
    </div>
    `,
    data() {
        return {
            texto: 'Footer de mi sitio web'
```

```
        }
    },

    methods: {
        fecha() {
            return new Date().toLocaleString()
        }
    }

})
```

6.- Configurar las **props** para que sean dinámicas, pasamos la información de "cantidad" del main.js al Footer.js:

- **index.html**

```
...
        <footer-banco
            :valor="cantidad"
        />
...
```

- **Footer.js**

```
App.component('footer-banco', { //El nombre del componente siempre "minúsculas y separado por guiones"

    props: ['valor'],

    template: /*html*/`
    <div class="bg-dark py-3 mt-2 text-white">
            <h3>{{texto}} - {{valor}}</h3>
            <p>{{fecha()}}</p>
    </div>
    `,
    data() {
        return {
            texto: 'Footer de mi sitio web'
        }
    },

    methods: {
        fecha() {
            return new Date().toLocaleString()
        }
    }

})
```

Libreta de pagos pendientes con Vue.js

Seguir el siguiente video paso a paso, para cualquier duda del código tienes, a continuación, el código final del ejemplo.

Código solución:

- **index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="s
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css">


    <title>Libreta de pagos con VueJs</title>
</head>
<body>
```

```
<div class="container">
    <div class="row">
        <div class="col-xs-12">
            <h1 class="jumbotron">Libreta de pagos con VueJs</h1>
            <div id="app">
                <table class="table table-striped">
                    <thead>
                        <tr>
                            <th class="text-center" style="width:40px;"></th>
                            <th>Descripción</th>
                            <th style="width:200px;">Cantidad</th>
                            <th class="text-center" style="width:100px;">¿Pagado?</th>
                        </tr>
                    </thead>
                    <tbody>


                        <tr>
                            <td></td>
                            <td>
                                <input type="text" class="form-control" v-model="newEntry.name" />
                            </td>
                            <td>
                                <input type="text" class="form-control" v-model="newEntry.amount" />
                            </td>
                            <td>
                                <button @click="add()"
                                type="button" class="btn btn-success btn-block">Agregar</button>
                            </td>
                        </tr>


                        <tr v-if="items.length === 0">
                            <td colspan="4" class="text-center">
                                No hay registros que mostrar
                            </td>
                        </tr>


                        <tr v-for="item, index in items">
                            <td>
                                <button
                                @click="remove(index)"
                                type="button" class="btn btn-danger btn-xs">
                                    <span class="glyphicon glyphicon-trash"></span>
                                </button>
                            </td>
                            <td>{{ item.name }}</td>
                            <td>{{ item.amount.toFixed(2) }}</td>
                            <td class="text-center" :title="item.paid? 'Si' : 'No'">
                                <button type="button"
                                @click="changeToPaid(item)"
                                class="btn btn-default btn-sm"
                                :class="{ 'btn-success' : item.paid}">
                                    <span v-if="item.paid" class="glyphicon glyphicon-ok"></span>
                                    <span v-if="!item.paid" class="glyphicon glyphicon-remove"></span>
                                </button>
                            </td>
                        </tr>
                    </tbody>

                    <tfoot>
                        <tr>
                            <td></td>
                            <td class="text-right">Por pagar</td>
                            <td>{{ totalAmount(0) }}</td>
                            <td></td>
                        </tr>
                        <tr>
                            <td></td>
                            <td class="text-right">Pagado</td>
                            <td>{{ totalAmount(1) }}</td>
                            <td></td>
```

```
                            </tr>
                            <tr>
                                <td></td>
                                <td class="text-right">Total</td>
                                <td>{{ totalAmount(2) }}</td>
                                <td></td>
                            </tr>
                        </tfoot>
                    </table>


                </div>
            </div>
        </div>
    </div>

    <script src="https://unpkg.com/vue@next"></script>
    <script src="script.js"></script>
    <script>
        //Se monta el objeto App en el div #app
        const mountedApp = App.mount('#app')
    </script>
</body>
</html>
```

- **script.js**

```
//Se crea un App Vue
const App = Vue.createApp({

    data() { //Datos del componente
        return {
            items: [
                { name: 'Servicios', amount: 200, paid: false},
                { name: 'Hosting', amount: 90, paid: true}
            ],
            newEntry: {
                name: '',
                amount: 0,
                paid: false
            }
        }
    },

    methods: {
        remove(index) {
            this.items.splice(index, 1)
        },
        add() {
            // console.log(this.newEntry)
            this.items.push({
                name: this.newEntry.name,
                amount: parseFloat(this.newEntry.amount),
                paid: false
            })
            this.newEntry.name = '',
            this.newEntry.amount = 0
        },
        changeToPaid(item) {
            item.paid = !(item.paid)
        },
        totalAmount(opt) {
            let total = this.items.reduce(function(a, b) {
                switch(opt) {
                    case 0: return a + (!b.paid ? b.amount : 0)
                    case 1: return a + (b.paid ? b.amount : 0)
                    case 2: return a + b.amount
                }
            }, 0)
            return total.toFixed(2)
        }
    }
```

```
})
```

CRUD PHP con Vue.js + Axios + MariaDB

Seguir este Video para adaptar Marvel a Vue.js

Tecnologías utilizadas:

- Front-End : Vue.js - Vuetify - Sweet Alert 2
- Back-End : Servidor Linux NGINX - PHP - MySQL - Axios

Creamos un CRUD PHP para interactuar con la BD.

Creamos un index.html y con CDNs referenciamos todas las librerías necesarias.

1.- Instalo Vuetify-vscode en el VSCode.
2.- Creamos un **index.html** y accediendo a la web de Vuetify copiamos la plantilla de CDN y la pegamos en nuestro **index.html**.
3.- Enlazamos Sweet Alert 2 para los mensajes de alerta.
4.- Enlazamos Axios para acceder a la BD PHP.

Ya tenemos el **index.html** preparado para empezar a programar:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://fonts.googleapis.com/css?family=Roboto:100,300,400,500,700,900" rel="stylesheet">
    <link href="https://cdn.jsdelivr.net/npm/@mdi/font@4.x/css/materialdesignicons.min.css" rel="stylesheet">
    <link href="https://cdn.jsdelivr.net/npm/vuetify@2.x/dist/vuetify.min.css" rel="stylesheet">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/sweetalert2@9.17.2/dist/sweetalert2.min.css">
    <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no, minimal
</head>
<body>
  <div id="app">
    <v-app>
      <v-main>
        <v-container>Hello world</v-container>
      </v-main>
    </v-app>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/vue@2.x/dist/vue.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/vuetify@2.x/dist/vuetify.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/sweetalert2@9.17.2/dist/sweetalert2.all.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.21.1/axios.min.js" integrity="sha512-bZS47S7sPOxkjU/4
  <script>
    new Vue({
      el: '#app',
      vuetify: new Vuetify(),
    })
  </script>
</body>
</html>
```

5.- Modificamos el código anterior del siguiente modo (el código base para la tabla se puede descargar de la página de Vuetify.

```
...
<body>
  <div id="app">
    <v-app>
      <v-main>
        <v-card class="mx-auto mt-5 max-width=1200">
            <v-btn rounded color="green acent-2" @click="formNuevo()">Crear</v-bin>
        </v-card>

        <v-simple-table class="mt-5">
            <template v-slot:default>
```

```
                    <thead>
                    <tr class="light-blue darken-2">
                        <th class="white--text">NAME</th>
                        <th class="white--text">ALIGNMENT</th>
                        <th class="white--text">HOMETOWN</th>
                        <th class="white--text">GENDER</th>
                        <th class="white--text">FIGHTING SKILLS</th>
                        <th class="white--text">EDITAR / BORRAR</th>
                    </tr>
                    </thead>

                    <tbody>
                    <tr v-for="marvel in marvels" :key="marvel.ID" :id="marvel.ID">
                        <td>{{ marvel.name }}</td>
                        <td>{{ marvel.alignment }}</td>
                        <td>{{ marvel.hometown }}</td>
                        <td>{{ marvel.gender }}</td>
                        <td>{{ marvel.fighting_Skills }}</td>
                        <td>
                            <v-btn>Editar</v-btn>
                            <v-btn>Borrar</v-btn>
                        </td>
                    </tr>
                    </tbody>

                </template>
            </v-simple-table>
          </v-main>
        </v-app>
      </div>
    ...
```



Vue Crud Base HTML

6.- Realizamos una petición get con la librería Axios a **buscar.php** con **alignment=Todos** de modo que nos enviará toda la tabla **marvels**.

```
    ...
      <script>
          const url="./php/buscar.php?alignment=Todos"

          new Vue({
            el: '#app',

            vuetify: new Vuetify(),

            data() {
                return {
                    marvels: [],
                    marvel: {
                        name:'',
                        alignment:'',
                        hometown:'',
                        gender:'',
                        fighting_Skills:''
                    }
                }
            },
            created() {
                this.mostrar()
            },
            methods: {
                mostrar: function() {
                    axios.get(url)
                    .then(res => {
                        this.marvels = res.data.resultados
                        // console.log(this.marvels)
                    }, (err) => {
                        console.log(err)
                    })
                }
```

```
        }
        })
    </script>
</body>
</html>
```

Vue.js CLI

Instalación Vue.js

Para instalar Vue.js lo primero es instalar **Node** bajándolo de nodejs.org.

Una vez instalado **Node**, reiniciamos el equipipo y pasamos a instalar **Vue**:

```
$ npm install -g @vue/cli
```

Ahora podemos ver la versión de Vue instalada

```
$ vue ?version
# @vue/cli 4.5.11
```

Ya estamos preparados para crear un proyecto

```
$ vue create hola-mundo
# Deseleccionamos Linter y dejamos Babel y Vue version
## Presionamos Enter y seleccionamos la versión 3 de Vue
## Seleccionamos en archivos separados
## No guardamos este proyecto para otros futuros
```

Ahora entramos en la carpeta del proyecto

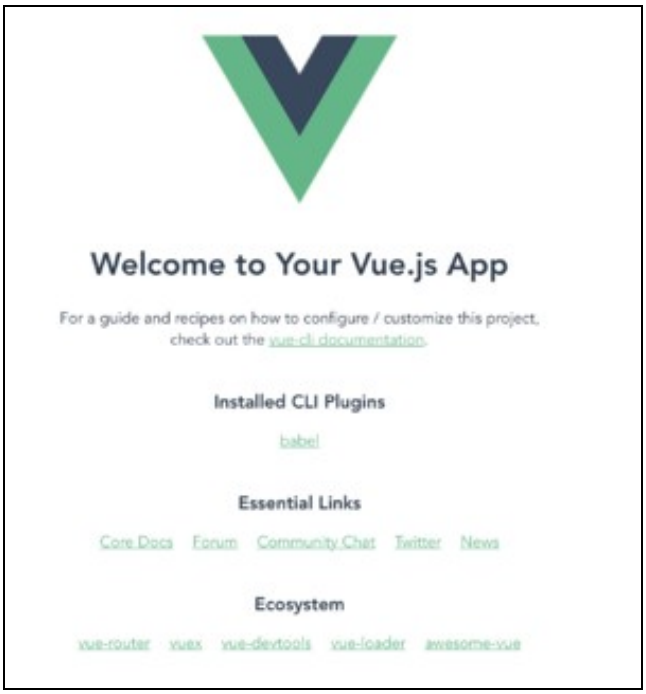
```
$ cd hola-mundo
```

y arrancamos el servidor del siguiente modo

```
$ npm run serve
```

Y, ahora, en un navegador

```
htpp://localhost:8080
# y veremos nuestro proyecto
```

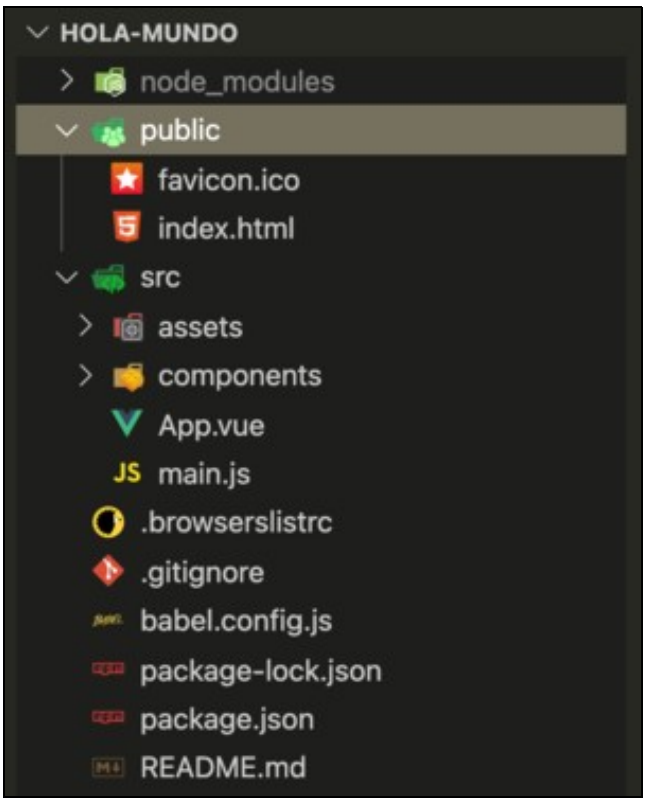


Base Proyecto Vue.js CLI
Para trabajar en VSCode, nos aseguraremos tener instaladas las extensiones:

- **Vetur de VSCode**
- **Vue VSCode Snippets**

Y, como detalle final, decir que no estaría de más tener unos pequeños conocimientos de **Node.js**

Archivos Proyecto CLI



Directorios y Archivos Vue.js CLI

- El archivo **index.html** es una estructura básica de HTML5, siendo la línea importante la línea:

```
<div id="app"></div>
```

- El archivo **main.js** tiene el siguiente contenido donde se hace referencia al *div* donde se va a montar el código Vue.js escrito en el archivo **App.vue**:

```
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

- El archivo **App.vue** tiene el siguiente contenido, donde vemos que se distinguen tres partes principales:

  ◊ **<template>** : El HTML del elemento .vue.
  ◊ **<script>** : La parte de JavaScript.
  ◊ **<style>** : Los estilos.

  Todos juntos en un mismo archivo pero, a la vez, separados en tres partes bien diferenciadas.
  Vemos que ***App.vue* es un componente general, el componente padre**.

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld msg="Welcome to Your Vue.js App"/>
</template>

<script>
```

```
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Vemos que en CLI sí se pueden utilizar *Camel case* tal y como se ve en la llamada al componente **HelloWorld** (línea 3 del código anterior), donde se le está enviando una **prop** denominada *msg*.

- El contenido del archivo **HelloWorld.vue** existente en el directorio **components** tiene, por defecto un contenido muy amplio, pero lo vamos a dejar tal y como se muestra a continuación para seguir con nuestros ejemplos:

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

  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>


<style scoped>
h3 {
  margin: 40px 0 0;
}
ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  margin: 0 10px;
}
a {
  color: #42b983;
}
</style>
```

Apariencia ahora de nuestro proyecto:



Apariencia ahora de nuestra base de Proyecto

- También vemos que existe en nuestro proyecto un directorio llamado **assets** donde se guardarán los archivos estáticos, en este caso vemos que, de momento, está el archivo **logo.png**.

- El resto de archivos existentes fuera de **src** son archivos de configuración que se irán viendo poco a poco.

    ◊ El archivo **package.json** es, sin duda, el más importante de todos. Tiene el siguiente el contenido:

```
{
  "name": "hola-mundo",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build"
  },
  "dependencies": {
    "core-js": "^3.6.5",
    "vue": "^3.0.0"
  },
  "devDependencies": {
    "@vue/cli-plugin-babel": "~4.5.0",
    "@vue/cli-service": "~4.5.0",
    "@vue/compiler-sfc": "^3.0.0"
  }
}
```

Módulos con CLI

Como se vio en el apartado anterior, tenemos ya creado el módulo principal padre **App.vue** que llama a otro módulo **HelloWorld.vue**.

Primeramente, podemos modificar en el módulo App.vue el prop de llamada del módulo HelloWorld.exe del siguiente modo:

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld msg="Hola mundo a CLI"/>
</template>
...
```

Vemos que se modifica el *prop* **msg** que se configuró en el módulo **HelloWorld.vue** indicando el tipo de datos que guardará:

```
...
<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>
...
```

y, en el apartado *<template>* de dicho componente llamamos para incorporar dicha *prop*:

```
<template>
```

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

  </div>
</template>
```

Vamos ahora a crear nosotros un nuevo componente para tener mucho más claro cómo se realizan.

1.- En el directorio componentes, creamos un nuevo archivo llamado **Titulo.vue**

2.- Si tenemos instaladas en VSCode las extensiones de Vue antes comentadas, escribiendo **vue** y pulsando **Tab** se creará automáticamente la estructura básica que podemos ver a continuación

```
<template>

</template>

<script>
export default {

}
</script>

<style>

</style>
```

3.- Ahora hacemos unos cambios en el archivo **App.vue** porque crearemos nuestro propio módulo para el título y tenemos que eliminar ciertas líneas y comentar otra para así hacer desaparecer el título

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">

</template>

<script>
// import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {

  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

4.- Tenemos que modificar dos archivos, el primero será **Titulo.vue** donde simplemente añadimos la línea que vemos a continuación

```
<template>
  <h1>Mi banco 2.0</h1>
</template>
...
```

También modificamos el archivo **App.vue** donde tendremos que configurar el enlace del submódulo **Titulo.vue** con este módulo principal

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <titulo />
</template>

<script>
// import HelloWorld from './components/HelloWorld.vue'
import Titulo from './components/Titulo.vue' //Con o sin extensión
```

```
export default {
  name: 'App',
  components: {
    Titulo
  }
}
</script>
...
```



Nuevo título

5.- Pero también podemos enviar **props** a la hora de llamar al módulo para así realizar módulos dinámicos.
Primeramente modificamos el módulo padre **App.vue**

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <titulo texto="Mi banco dinámico 2.0" />
</template>
...
```

Luego modificamos el submódulo **Titulo.vue** que estamos creando

```
<template>
  <h1>{{texto}}</h1>
</template>

<script>
export default {
    name: 'Titulo',
    props: {
        texto: String
    }
}
</script>

// Añadir scoped para que el estilo sólo sea para elementos de este componente
<style scoped>
    h1 {
        color: peru;
    }
</style>
```



Módulo dinámico

6.- Ahora vamos a hacer el módulo **Cuenta.vue** igual al realizado con **CDN**
Contenido del archivo **App.vue**

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <titulo texto="Mi banco dinámico 2.0" />
  <cuenta />
</template>

<script>
import Titulo from './components/Titulo.vue' //Con o sin extensión
import Cuenta from './components/Cuenta.vue'
export default {
  name: 'App',
  components: {
    Titulo,
    Cuenta
  }
}
</script>
...
```

Contenido del archivo **Cuenta.vue**

```
<template>
    <h2>Tipo de cuenta:  {{cuenta}}</h2>
    <h2>Saldo: {{saldo}}</h2>
    <h2>Cuenta {{ estado ? 'Activa' : 'Desactivada' }}</h2>
    <div v-for="(servicio, index) in servicios" :key="index">
        {{ index + 1 }} - {{ servicio }}
    </div>
</template>

<script>
export default {
    data() {
        return {
            saldo: 1000,
            cuenta: 'Visa',
            estado: true,
            servicios: ['giro', 'abono', 'transferencia']
        }
    },
}
</script>
...
```

7.- Creamos un componente llamado **AccionSaldo.vue** hijo del componente **Cuenta.vue**, la idea es que desde el componente padre modifiquemos el comportamiento del hijo y, luego, desde el hijo también modifiquemos el del padre.
Inicialmente el contenido de **Cuenta.vue** será modificado del siguiente modo

```
<template>
    <h2>Tipo de cuenta:  {{cuenta}}</h2>
    <h2>Saldo: {{saldo}}</h2>
    <h2>Cuenta {{ estado ? 'Activa' : 'Desactivada' }}</h2>
```

```
    <div v-for="(servicio, index) in servicios" :key="index">
        {{ index + 1 }} - {{ servicio }}
    </div>
    <AccionSaldo texto="Aumentar saldo" />
    <AccionSaldo texto="Disminuir saldo" />
</template>

<script>
import AccionSaldo from './AccionSaldo.vue'

export default {
    components: {
        AccionSaldo
    },
    data() {
        return {
            saldo: 1000,
            cuenta: 'Visa',
            estado: true,
            servicios: ['giro', 'abono', 'transferencia']
        }
    },
    methods: {
        aumentar() {
            this.saldo = this.saldo + 100
        },
        disminuir() {
            this.saldo = this.saldo - 100
        }
    }
}
</script>

<style>

</style>
```

El contenido de **AccionSaldo.vue** será el siguiente

```
<template>
  <button>{{ texto }}</button>
</template>

<script>
export default {
    props: {
        texto: String
    }
}
</script>

<style>

</style>
```

8.- Ahora veremos cómo un componente hijo modifica la lógica de un componente padre (**Custom Events**).

El contenido de **AccionSaldo.vue** será el siguiente

```
<template>
  <button @click="accionHijo">{{ texto }}</button>
</template>

<script>
export default {
    props: {
        texto: String
    },
    methods: {
        accionHijo() {
            this.$emit('accionPadre')
        }
    }
}
</script>

<style>
```

```
</style>
```

El contenido de **Cuenta.vue** será el siguiente

```
<template>
    <h2>Tipo de cuenta:  {{cuenta}}</h2>
    <h2>Saldo: {{saldo}}</h2>
    <h2>Cuenta {{ estado ? 'Activa' : 'Desactivada' }}</h2>
    <div v-for="(servicio, index) in servicios" :key="index">
        {{ index + 1 }} - {{ servicio }}
    </div>
    <AccionSaldo
        texto="Aumentar saldo"
        @accionPadre="aumentar"
    />
    <AccionSaldo
        texto="Disminuir saldo"
        @accionPadre="disminuir"
    />
</template>

<script>
import AccionSaldo from './AccionSaldo.vue'

export default {
    components: {
        AccionSaldo
    },
    data() {
        return {
            saldo: 1000,
            cuenta: 'Visa',
            estado: true,
            servicios: ['giro', 'abono', 'transferencia']
        }
    },
    methods: {
        aumentar() {
            this.saldo = this.saldo + 100
        },
        disminuir() {
            if (this.saldo === 0) alert('Saldo agotado')
            else this.saldo = this.saldo - 100
        }
    }
}
</script>

<style>

</style>
```

9.- Ahora hacemos las modificaciones necesarias para desactivar el botón disminuir cuando llegamos a un saldo = 0

El contenido de **AccionSaldo.vue** será el siguiente

```
<template>
  <button @click="accionHijo" :disabled="desactivar">{{ texto }}</button>
</template>

<script>
export default {
    props: {
        texto: String,
        desactivar: {
            type: Boolean, // Tipo
            default: false // Valor por defecto
        }
    },
    methods: {
        accionHijo() {
            this.$emit('accionPadre')
        }
    }
```

```
}
</script>

<style>

</style>
```

El contenido de **Cuenta.vue** será el siguiente

```
<template>
    <h2>Tipo de cuenta:  {{cuenta}}</h2>
    <h2>Saldo: {{saldo}}</h2>
    <h2>Cuenta {{ estado ? 'Activa' : 'Desactivada' }}</h2>
    <div v-for="(servicio, index) in servicios" :key="index">
        {{ index + 1 }} - {{ servicio }}
    </div>
    <AccionSaldo
        texto="Aumentar saldo"
        @accionPadre="aumentar"
    />
    <AccionSaldo
        texto="Disminuir saldo"
        @accionPadre="disminuir"
        :desactivar="desactivar"
    />
</template>

<script>
import AccionSaldo from './AccionSaldo.vue'

export default {
    components: {
        AccionSaldo
    },
    data() {
        return {
            saldo: 1000,
            cuenta: 'Visa',
            estado: true,
            servicios: ['giro', 'abono', 'transferencia'],
            desactivar: false
        }
    },
    methods: {
        aumentar() {
            this.saldo = this.saldo + 100
            this.desactivar = false
        },
        disminuir() {
            this.saldo = this.saldo - 100
            if (this.saldo === 0) this.desactivar = true
        }
    }
}
</script>

<style>

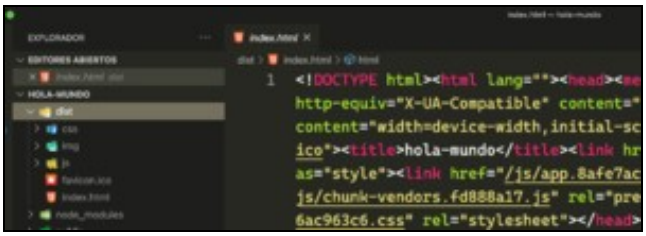
</style>
```

Compilar para publicar en un servidor

Ahora lo único que nos queda es compilar todo el proyecto para subirlo al servidor. Escribimos en la línea de comandos:

```
$ npm run build
```

Aparecerá en el interior de nuestra carpeta del proyecto un directorio denominado **dist** con todos los archivos a subir



Archivos para despliegue